Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

INFORME TÉCNICO

Introducción a la Simulación de Eventos Discretos

Gabriel A. Wainer

Report n.: 96-005

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

http://www.dc.uba.ar

Title: **Introducción a la simulación de sistemas de eventos discretos**

Authors: **Gabriel A. Wainer**

E-mail: **gabrielw@dc.uba.ar**

Report n.: **96-005**

Key-words: **Discrete events Simulation and Modelling, Parallel Simulation, Cellular Automata.**

Abstract: In this work we present a survey about different mechanisms and techniques for discrete event simulation. We analyze several features about modeling and simulation, paying special attention to two formalisms: DEVS and Cellular Automata. Finally, we study questions related with high performance simulation through parallel and distributed approaches.

To obtain a copy of this report please fill in your name and address and return this page to:

Infoteca
Departamento de Computación - FCEN
Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires - Argentina

TEL/FAX: (54)(1)783-0729
e-mail: infoteca@dc.uba.ar

You can also get a copy by anonymous ftp to: **zorzal.dc.uba.ar/pub/tr**

or visiting our web: **http://www.dc.uba.ar/people/proyinv/tr.html**

Name:.....

Address:.....

.....

Introducción a la simulación de sistemas de eventos discretos

Gabriel A. Wainer
Departamento de Computación - FCEN
Universidad de Buenos Aires.
gabrielw@dc.uba.ar

En este trabajo presentamos un informe acerca de distintos mecanismos y técnicas de simulación de sistemas de eventos discretos. Se analizan generalidades acerca de modelos y simulación, y se presta atención especial a dos formalismos: DEVS y autómatas celulares. Finalmente se analizan cuestiones relacionadas con simulación de alto rendimiento usando soluciones paralelas y distribuidas.

Palabras clave: eventos discretos, DEVS, simulación paralela, autómatas celulares.

1. INTRODUCCION

En general, para implementar sistemas automatizados flexibles, complejos y altamente precisos, debemos construir sistemas de test caros y complejos. Esta complejidad hace que el chequeo sea difícil (o imposible), así como la previsión de su comportamiento y comprensión global, indispensable para minimizar los riesgos en el sistema desarrollado. Para lograr estos objetivos con una aproximación efectiva con respecto a los costos, está difundido el uso de metodologías y herramientas de simulación.

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo del sistema, las decisiones 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. El sistema a construir tendrá como objetivo ayudar a los investigadores a modelar tales fenómenos complejos.

El objetivo de este trabajo es servir como introducción a algunos conceptos generales sobre modelización y simulación utilizando determinadas metodologías. Presentaremos un conjunto de técnicas importantes con especial utilidad en la simulación de sistemas de eventos discretos. El trabajo está basado en los que se mencionan en la bibliografía, y tiene como intención servir como material básico de estudio en Español. Está orientado a difundir los temas en este área de simulación, y se ha organizado en cuatro secciones. En la introducción se analizan conceptos generales de modelos, sistemas y simulación. En la segunda sección se atacan generalidades del formalismo DEVS (un formalismo para simulación de eventos discretos). Una tercera sección analiza otro formalismo: los autómatas celulares. En la cuarta sección se estudian con detalle mecanismos de simulación paralela y distribuida, con el fin de mejorar la desempeño de las simulaciones.

1.1. Sistemas, modelos y paradigmas

Para comenzar, daremos algunas definiciones. Primero, llamaremos **sistema** a una entidad real o artificial. De hecho no existe una definición de sistema que tenga aceptación general. Se llama sistema a una parte de una realidad, restringida por un entorno. Está compuesto por entidades que experimentan efectos espacio-tiempo y relaciones mutuas. También se dice que un sistema es un conjunto ordenado de objetos lógicamente relacionados que atraviesan ciertas actividades, interactuando para cumplir ciertos objetivos.

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

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

Llamaremos **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 razonamos sobre modelos. El proceso de pensar y razonar acerca de un sistema resaltando la reacción de un modelo se llama **modelado de sistemas**.

Ejemplo 1

Consideremos el diseño de un circuito digital. Si analizamos la medición de corrientes y tensiones estamos razonando sobre un modelo eléctrico, en cambio si se hace un estudio de las funciones booleanas que realiza, se razona sobre un modelo lógico.

Para estudiar sistemas complejos, la idea es partir haciendo un modelo del sistema que se quiere estudiar, y se estudian problemas del sistema real estudiando el modelo.

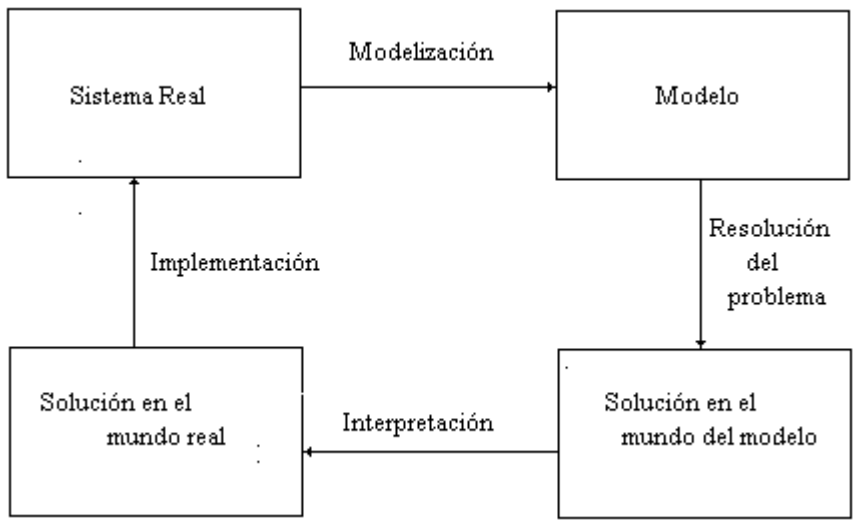

Figura 1. Pasos en el estudio de un sistema por medio de modelización [Gia96]

Llamaremos **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 2

Consideremos un lenguaje de programación algorítmico como Pascal. Aquí usamos el paradigma de modelización secuencial/procedural, con el fin de construir un modelo que en este caso es un programa modelado por una actividad secuencial. De la misma forma, el simbolismo de circuitos lógicos es un paradigma en el cual un esquema es el modelo de un circuito digital. En base a compuertas lógicas y un conjunto de reglas de conexión, permitimos la construcción de un sistema de circuitos lógicos.

1.2. Modelos y simulación

¿Porqué se hacen modelos de los sistemas? ¿Porqué usamos 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 en 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**: los modelos están basados en razonamiento. Suelen ser simbólicos, y permiten obtener soluciones generales al problema. 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 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.

El problema es que si consideramos sistemas complejos, 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. Frente a esta situación, la simulación ofrece otra aproximación de resolución de problemas que permite tratar cierta complejidad.

B) **Basados en simulación**: en ellos no existen soluciones generales sino que buscan soluciones particulares para el problema. Si el problema es simple, es conveniente el uso de métodos analíticos ya que nos 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 y obtener resultados de salida significativos. Así se obtienen datos que pueden estudiarse para analizar algún comportamiento que interese.

El uso de simulación permite experimentación controlada, compresión de tiempo (una simulación se realiza en mucho menos tiempo que el sistema real que modela), 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 su tiempo de desarrollo, en que los resultados pueden tener divergencia con la realidad (precisan validación), y en que para reproducir el comportamiento del sistema simulado se precisa colección extensiva de datos.

Definición:

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

Por ende, la simulación es el proceso de diseñar un modelo de un sistema real, y conducir experimentos basados en computadoras para describir, explicar y predecir el comportamiento del sistema real. En general, para hacer un simulador se siguen los siguientes pasos:

a) Planteo del problema: en esta fase se toma un sistema real y se trata de entenderlo. 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). En esta etapa también se trata de definir medidas de desempeño del sistema (como función de 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 presunciones y simplificaciones 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 vía observación. Por ende, 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 hay dos fases.

En una primera etapa se trata de comprender el sistema, ya sea siguiendo una aproximación de **flujo físico** basado en el flujo de entidades a través del sistema con sus puntos de procesamiento y reglas de decisión, o una aproximación de **eventos** (o cambio de estados), basado en la definición de variables de estado internas seguida por una descripción de la operación del sistema cuando ocurre un evento. En una segunda etapa se construye el modelo. Para ello se definen objetos, atributos, métodos, en el paradigma elegido. También en esta fase se elige un lenguaje de implementación.

d) Implementación: en esta etapa, en base al lenguaje elegido 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 de computadora (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) Experimento de simulación y optimización: 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 transiente 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 respuesta al comportamiento del sistema original.

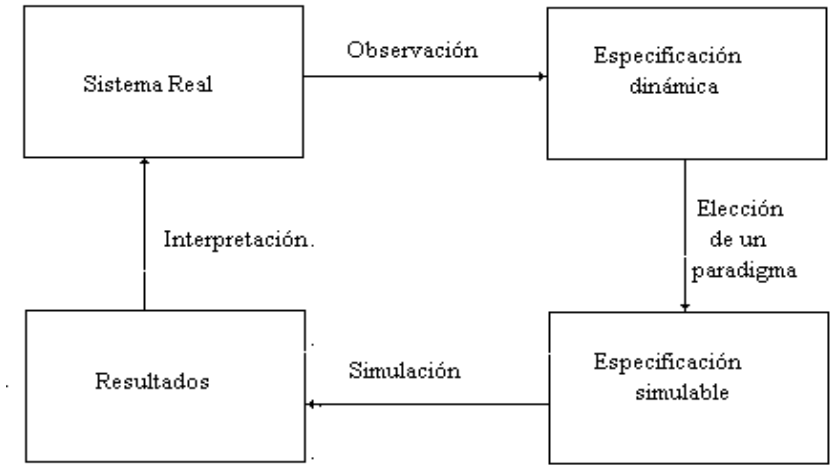

Figura 2. Pasos en el estudio de sistemas con simulación

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 usa simulación para adquirir una mejor comprensión de las operaciones del sistema real;

. de **predicción**: se usa un modelo para predecir el concepto futuro del sistema real;

. de **mejoras**: se utiliza para optimizar el desempeño del sistema real, y estudiar distintas alternativas (por ejemplo, un sistema de producción, de stocks, etc.);

. de **concepción**: el sistema no existe, y se usa simulación para chequear distintas soluciones posibles (por ejemplo, porque no se puede hacer un prototipo).

En cualquier caso, nos interesa tener **modelos ejecutables** de sistemas, o sea, descripciones inteligibles sobre las que se pueden ejecutar algoritmos en tiempo finito. Para poder construir un modelo simulable, el sistema en consideración deber obedecer a dos principios fundamentales:

. **Causalidad**: el futuro no puede influenciar sobre el pasado. El estado del sistema al instante presente t es independiente de todo lo que se pueda producir en horas futuras a t.

. **Determinismo**: el futuro del sistema puede estar determinado a partir de su estado presente y su pasado. En todo instante t existe un valor positivo **e** tal que el comportamiento del sistema puede calcularse hasta t+**e.**

1.3. Categorías de paradigmas de modelado

Durante siglos el desarrollo de sistemas dinámicos estuvo basado en el estudio de modelos de ecuaciones diferenciales ordinarias y parciales. Estas permitieron modelar exitosamente los sistemas dinámicos encontrados en la naturaleza (de hecho, los éxitos de la física y esta línea de investigación fueron tan grandes que penetraron casi todo el pensamiento científico). Pero 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 o ensamblado, las redes de computadoras y comunicaciones, los sistemas de control tráfico (en el aire y tierra), los sistemas de control militar, etc. En estos sistemas, la evolución en el tiempo depende de interacciones complejas de varios eventos discretos y de su temporalidad, tales como la llegada o partida de un trabajo, y la iniciación o finalización de una tarea, etc. El "estado" de tales sistemas sólo cambia en instantes discretos de tiempo en lugar de continuamente [Ho89]. La simulación aparece como una alternativa para estudiar el comportamiento de estos sistemas complejos.

Una de las primeras aplicaciones de simulación con computadoras fue en el proyecto Manhattan, donde se estudió la difusión aleatoria de neutrones para el desarrollo de la bomba atómica, usando métodos de Montecarlo. 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.

La gran variedad de paradigmas de modelado puede 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 3

. 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 y otros modelos de eventos discretos: 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 sistemas (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 al entorno, los modelos son **autónomos** (no existen entradas) o **no autónomos** (existen entradas). Los autónomos evolucionan en base a la función de tiempo.

1.4. Simulación de DEDS

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 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 4

Un caso típico de un DEDS es una red de computadoras. Se pueden 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, transmisión de un mensaje a otro nodo, fallas de componentes, etc.

Para implementar una simulación de eventos discretos se suele utilizar:

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 $\in \Re$ (el tiempo es continuo).
c) Un reloj global que indica el instante actual de simulación; es la hora de ocurrencia del evento que se está tratando en la actualidad.

Comparado con sistemas de variables continuas, fundamentales en la física, el modelado de DEDS es un fenómeno relativamente reciente. A pesar que pertenece al dominio de la Investigación Operativa (desde el punto de vista que la IO puede pensarse como la técnica de operaciones y eventos de sistemas hechos por el hombre), el desarrollo de DEDS también recibió un gran ímpetu de la teoría de control y sistemas. En particular, los conceptos de dinámica (constantes de tiempo, tiempo de respuesta, frecuencia, controlabilidad) son importantes en el desarrollo de modelos y herramientas de 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 en las que hay un conjunto fijo de estaciones y clientes que se mueven de estación a estación. Las rutas posibles de clientes son fijas (por ejemplo, sistemas de manufactura y redes de comunicación). En sistemas de topología dinámica las interacciones son arbitrarias y dinámicas (Ej.: escenarios de campos de batalla y sistemas de radio móvil). Los componentes de estos sistemas pueden moverse en cualquier dirección e interactuar con los otros componentes. Los sistemas de topología dinámica no suelen ser estrictamente DEDS ya que los componentes se pueden mover continuamente, sin embargo, para el propósito de la simulación, el modelo de movimiento físico en el sistema se discretiza en el tiempo.

1.5. Paradigmas de especificación de modelos DEDS

Existe variedad de paradigmas que pueden utilizarse para especificar formalmente sistemas dinámicos y tienen existencia conceptual independiente de los lenguajes de simulación que pueden usarse para implementar las simulaciones. No puede decirse que ninguno de los formalismos existente sea mejor que otro para representar la variedad de comportamiento de 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. Entre los formalismos desarrollados hasta ahora, mencionamos los siguientes:

. Cadenas de Markov/modelos de autómatas (en este grupo también incluimos Redes de Petri y máquinas de estado extendidas).
. Modelos de olas
. Modelos algebraicos min-max.
. Modelos basados en computadoras (ej.: CSP)
. Modelos de procesos semi-markovianos generalizados (GSMP - dentro de esta categoría incluimos todos los esfuerzos relacionados en lenguajes de simulación de eventos discretos).

	Con tiempo	Sin tiempo
Lógica	Lógica temporal	Máquinas de estados finitos Redes de Petri
Algebraico	Álgebra min-max	Procesos finitamente recursivos CSP
Desempeño	Cadenas de Markov Modelos de colas GSMP Simulación	
	Estocástico ->	<- Determinístico

Figura 3. Clasificación de herramientas de modelado DEDS [Ho89]

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

i) La naturaleza discreta de los eventos. En ocasiones es posible analizar DEDS usando aproximaciones continuas (vía modelos de difusión o manejando exclusivamente tasas en vez de las partes constituyentes), pero como los estados reales de un DEDS son inherentemente discretos, es difícil evitar la naturaleza discreta del problema.

ii) La naturaleza continua de la mayoría de las mediciones de desempeño: las mediciones de desempeño de los DEDS suelen estar formuladas en términos de variables continuas, tales como promedio, rendimiento, tiempo de espera y ganancia.

iii) La importancia de formulación probabilística: en sistemas DEDS ocurren disturbios no planificados y/o caídas. Esto no implica que una aproximación totalmente determinista (por ejemplo, las Redes de Petri o el álgebra min-max) no tenga utilidad, sino que deben hacerse extensiones estocásticas.

iv) La necesidad de análisis jerárquico: las operaciones hechas por el hombre operan en planes jerárquicos. (Ejemplos: una fábrica tiene planes anuales, trimestrales, mensuales, semanales y diarios; los comandos militares se dividen en niveles de teatro/armada/batallón/pelotón). Luego, los detalles necesarios para modelar y controlar deben ser distintos para cada nivel

v) La presencia de dinámica: los modelos deben capturar el comportamiento dinámico y transiente de los DEDS.

vi) Límite computacional: en máquinas de estados finitas o cadenas de Markov sin estructura adicional el número de estados físicos discretos del DEDS modelado explota de forma combinatoria, lo que conduce a requerimientos de cálculo no factibles. Por ende, el paradigma debe estar orientado a desempeño, distinguiendo entre modelos conceptuales y computacionales. El requerimiento de cómputo de un modelo tampoco debe crecer exponencialmente con su tamaño [Ho79].

1.6. Modelos simulables

Para resolver un problema hay que hacer el pasaje de la especificación dinámica a una especificación simulable, y luego interpretar los resultados de la simulación en el sistema dinámico (hacer el pasaje inverso). El pasaje del sistema dinámico a las especificaciones simulables impondrá algunas restricciones sobre los conjuntos de sistemas dinámicos.

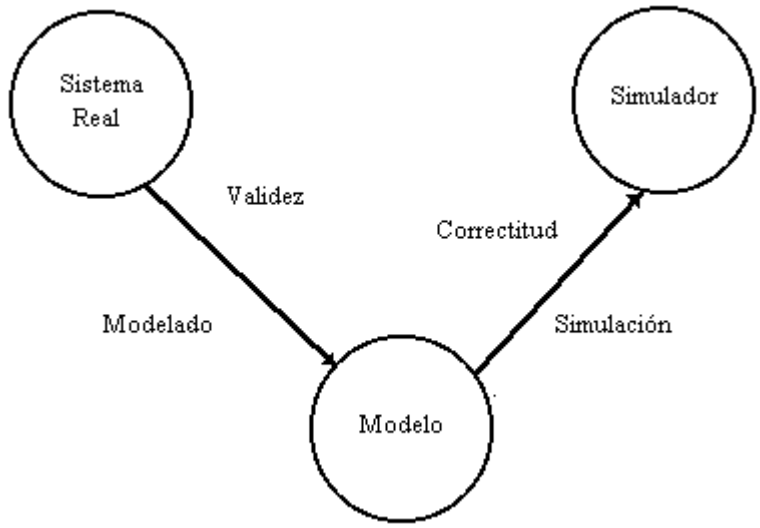

Figura 4 - Relaciones de modelado y simulación [Zei90]

Como vemos, debemos considerar tres objetos básicos que entran en juego:

a. El **sistema real** en existencia, que se ve como una fuente de datos. El sistema real es la parte del mundo real en la que uno se interesa y genera datos de comportamiento. El sistema tiene algunas variables observables y otras no. Al crear el modelo, deben construirse relaciones de entrada/salida de forma tal que hacer una variación de entrada, se pueda observar una variación de salida simular a la que ocurre en el sistema real. Si los elementos de entrada son discretos/continuos, el modelo también debe serlo.

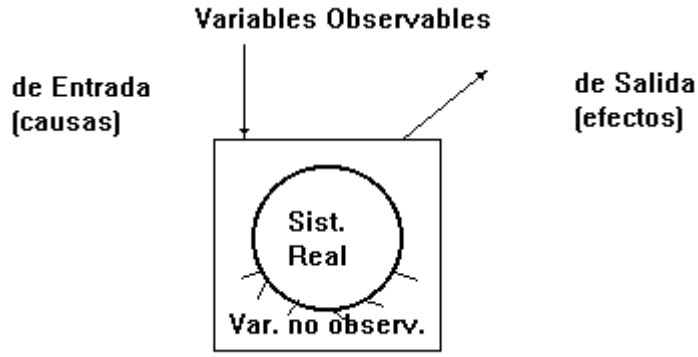

Figura 5 - Variables en un sistema real

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. La definición del modelo debe incluir además las condiciones de funcionamiento del sistema real, y los dominios de aplicación (hay que definir sus condiciones reales de utilización).

b. El **modelo** es un conjunto de instrucciones para generar datos comparables a los observables en el sistema real. La estructura del modelo es su conjunto de instrucciones. El comportamiento del modelo es el conjunto de todos los datos posibles que pueden ser generados ejecutando las instrucciones del modelo. Permite generar estos datos de comportamiento.

El modelo de base especifica el comportamiento de todas las relaciones de entrada/salida que se construyen, y el modelo simplificado es una simplificación (sobre el número de componentes o el nivel del paradigma) para observar el comportamiento .

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

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 cuan 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 debemos, 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. 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). En este pasaje también se pierde información.

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).

Si el modelo sólo permite observación del comportamiento, se lo llama **modelo comportamental** o **caja negra**. En este caso, lo único que se considera son las entradas/salidas del modelo. Cuando consideramos el comportamiento de un sistema, podemos usar representaciones causales y empíricas. Las empíricas son registros reales de datos (historia en el tiempo de las variables) tomadas del sistema real. En cambio las relaciones causales se integran en unidades que pueden generar datos empíricos.

Si el modelo representa la estructura del sistema modelado, se lo llama **modelo estructural** o **caja blanca**. En este caso, se explicitan los componentes del sistema real (que pueden ser modelos comportamentales simples). Aquí entran en juego conceptos de descomposición (como dividir un sistema en componentes) y acoplamiento (como combinar los componentes para reconstituirlo). Una tercera relación fundamental es la taxonomía que concierne las variantes admisibles de un componentes y sus especializaciones.

Ejemplo 5

Consideremos la representación de un modelo de un circuito. Podemos especificar dos clases de modelos. En este caso, el primero es comportamental, y el segundo es estructural. En general, para simulación se usan estos, porque permiten construir un modelo complejo en base a modelos simples. Definiendo los componentes y las uniones se puede construir un modelo complejo en base a componentes simples.

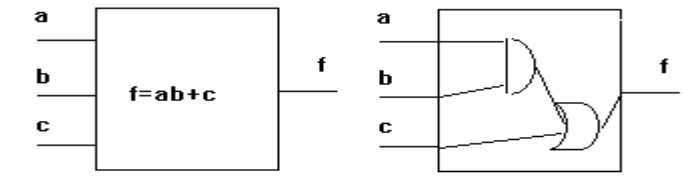

Figura 6. Modelos comportamentales y estructurales.

Como ya fue explicado, en la gran mayoría de los casos, los modelos y su implementación sufren simplificaciones por la complejidad del sistema a modelar. Esta complejidad puede ser de dos tipos:

. Cualitativa: depende de la semántica del paradigma. Toma en cuenta la naturaleza variable de los constituyentes del sistema.

. Cuantitativa: depende del número de constituyentes del modelo. Está ligado al sistema modelado y no al paradigma usado.

Si se desea reducir la complejidad, debe reducirse alguna de ambas, ya sea la del paradigma, o la cantidad de constituyentes en la formulación del modelo.

Ejemplo 6

Volvamos al caso de un modelo de circuitos lógicos. 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 ahora que tengo un circuito de N compuertas lógicas. Si se cambia el paradigma de modelado, y en vez de variables booleaneas se utiliza "transformación de registros", reduzco la complejidad cuantitativa. En este caso, se manipulan enteros, operaciones +, -, *, etc. También aumenta la complejidad cualitativa (en el caso de un registro, se modelan cientos de puertas con un solo operador por medio de un paradigma más complejo). Otra forma de reducir la complejidad cuantitativa sería por medio de descripción jerárquica, usando elementos de base, que se unen formando modelos de mayor nivel.

Por otro lado, si nuevamente consideramos el modelo de un programa secuencial, su programación en assembler tiene complejidad cualitativa muy baja (ya que el conjunto de instrucciones está limitado a unas cuantas operaciones simples), pero cuantitativamente complejo. Lo opuesto ocurre en un lenguaje de alto nivel, que es cualitativamente complejo, y cuantitativamente simple.

2. FORMALISMO DEVS

Muchas de las aproximaciones 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 mecanismo de simulación jerárquica, conocido como DEVS (Discrete EVents dynamic Systems). Esta aproximación propone una teoría de modelado de sistemas a tiempo continuo usando modelado de eventos discretos. Permite descripción **modular** de los fenómenos a modelar (aproximación modular) y ataca la complejidad usando una aproximación **jerárquica**.

La aproximación provee una forma de especificar un objeto matemático llamado sistema. Este se describe como un conjunto consistente de una base de tiempo, entradas, salidas y funciones para calcular los siguientes estados y salidas. Esta aproximación se integró posteriormente con nociones de programación orientada a objetos [Zei84, Zei90, Zei95]. Soporta la construcción de modelos de forma jerárquica y modular. Además de un medio para construir modelos simulables, provee una representación formal para manipular matemáticamente sistemas de eventos discretos.

DEVS es un formalismo universal para modelar y simular DEDS. 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 (atómicos), que se combinan para formar modelos acoplados. Los modelos atómicos son objetos independientes modulares, con variables de estado y parámetros, funciones de transición internas, externas, de salida y avance de tiempo. Existen dos variables de estado estándar: fase y σ. Si no hay eventos externos, el modelo se queda en la fase durante σ. σ es el tiempo restante 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, permitiendo representar el comportamiento de sistemas de tiempo continuo con eventos discretos. Un modelo acoplado especifica cómo se conectan las entradas y salidas de

los componentes. Los nuevos modelos también son modelos modulares, y pueden usarse para armar modelos de mayor nivel.

El principio de la simulación dirigida por eventos consiste en generar un planificador. Este contiene en cada instante la lista cronológica ordenada de los eventos a tratar en el futuro. Contiene los eventos potenciales y representa un futuro posible del modelo. En este caso se especifican modelos básicos a partir de los cuales se construyen otros mas complejos, y cómo acoplar estos modelos de forma jerárquica. La lista de eventos no es única, sino que está restringida a cada submodelo. 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 mejorando la productividad.

2.1. Modelos atómicos

Un modelo atómico es una especificación de un modelo comportamental. Formalmente, esta especificación puede definirse como

$$M = <X, S, Y, \delta int, \delta ext, \lambda, ta>$$

X: conjunto de eventos externos de entrada;
S: conjunto de estados secuenciales;
Y: conjunto de eventos externos generados para salida;
δint: S->S, función de transición interna, que define los cambios de estado por causa de eventos internos;
δext: QxX->S, 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 por el conjunto $Q = \{ (s,e) / s \in S, e \in [0, ta(s)] \}$, donde e representa el tiempo transcurrido desde la última transición de estado con estado s. Por ende, la función de transición externa depende del estado actual y del tiempo transcurrido;
λ: S-> Y, función de salida;
ta: $S->R^+$, función de duración de un estado;
ta(s): es el tiempo que el modelo se queda en el estado s si no hay un evento externo.

Un modelo atómico DEVS transita entre los estados (S) vía sus funciones de transición. Cuando no hay eventos externos, la hora de una función de transición interna está determinada por la función de pasaje de tiempo (ta) aplicada al estado actual. El nuevo estado del modelo se determina aplicando la función de transición interna (δint) al estado antiguo. Antes de cada transición interna, el modelo puede generar un evento de salida que depende del estado previo a la transición. También puede haber una transición de estado cuando hay un evento de entrada externa. La función de transición externa determina el nuevo estado basándose en el estado actual, el tiempo transcurrido en ese estado, y la entrada.

Ejemplo 7

En la figura 7 vemos la secuencia de eventos en un sistema modelado con DEVS: s2=δint(s1); s3=δint(s2), etc. Estas son las transiciones internas, que ocurren cuando no hay eventos externos. Por ejemplo, ta(s2) determina la hora de la próxima transición interna en base al estado s2. Esta hora del próximo evento se calcula en base al evento actual. Aquí y1=λ(s1), ..., son los eventos de salida generados. Estos se ejecutan antes de las transiciones internas. Cuando se recibe el evento de entrada externa x1, la transición externa resultante depende del estado actual, s4, el tiempo transcurrido, e4, y el evento de entrada, x1, para determinar un nuevo estado, s5 = δext(s4, e4, x1).

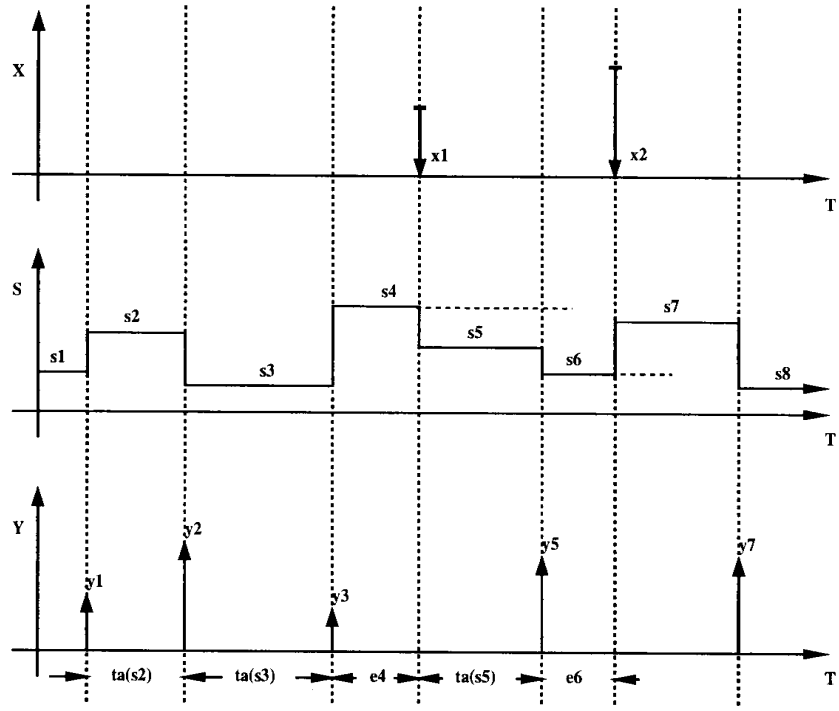

Figura 7 - Trayectorias en un modelo atómico DEVS [Cho95]

Para especificar modelos DEVS es conveniente ver al modelo como con ports de entrada/salida que interactúan con el entorno. 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 (encapsulamiento). Luego, la descripción simulable de un modelo atómico podrá ser guardada en una biblioteca para hacer acoplamiento con otros modelos más complejos.

Por ende, un modelo atómico debe contener la siguiente información:

. El conjunto de ports de entrada a través de los cuales se reciben eventos externos

. El conjunto de ports de salida a través de los cuales se envían eventos externos

. El conjunto de variables de estado y parámetros: como dijimos suelen estar presentes dos variables de estado - fase y σ (tiempo hasta la siguiente transición interna).

. La función de avance de tiempo que controla el timing de las transiciones internas (esta función simplemente el valor de σ).

. La función de transición interna que especifica a que estado pasara el sistema luego del tiempo dado por la función de avance del tiempo.

. La función de transición externa que especifica cómo cambia de estado el sistema cuando se recibe una entrada. El efecto es poner al sistema en una nueva fase y σ, planificándolo para la próxima

transición interna; el siguiente estado se calcula en base al estado actual y el valor del evento externo (que viene por un port de entrada), y el tiempo transcurrido en el estado actual.

. La función de salida que genera una salida externa antes de una transición interna.

Cada port de entrada precisa una especificación de una transición externa, en la forma de "cuando reciba x en el port de entrada p...". La función de transición interna puede especificarse en una descripción procedural con fases y sus transiciones. La función de salida usa frases de la forma "enviar y al port de salida p".

Ejemplo 8

Consideremos un modelo de buffering simple (fig. 8). Hay tres ports de entrada: *entrada*, para recibir datos de entrada, *listo*, para recibir reconocimiento del nivel inferior, y *parar* para control de flujo del proceso de nivel superior. El port *salida* envía ítems hacia abajo. 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).

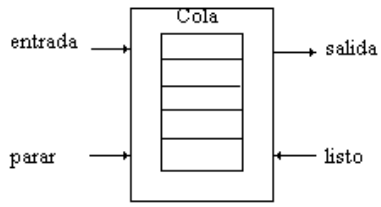

Figura 8- Un modelo de buffering simple

Transición externa

Cuando recibo 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 recibo 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 recibo 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

Transición interna

si fase = ENVIANDO /* Saca el primero de la cola luego del tpo. de preparación */
    Cola = resto(Cola)
    pasivar

Salida

si fase = ENVIANDO
  enviar primero(Cola) al port *salida*

La especificación de transiciones externas tiene tres "cuando", uno para cada port. La primera dice que cuando se recibe un valor en el port *entrada*, se inserta en la cola. Si es el único, el control debe pasar a la fase ENVIANDO (o sea, que debe transmitirse inmediatamente); sino el modelo debe continuar. La transición interna tiene una sola fase, ENVIANDO, en la cual el modelo espera un tiempo, tiempo_de_preparación, que provoca que se planifique la ocurrencia de una transición interna en el instante = hora actual + tiempo_de_preparación. Esto sirve para modelar el tiempo entre que llega el mensaje y se puede retransmitirlo. Al ocurrir dicho evento, se ejecuta la función de transición interna. Previamente se ejecuta la función de salida, que envía el primer valor en su cola al port *salida*, la función de transición interna lo saca de la cola, y se pone en estado pasivo. La fase *pasivar* (en la cual el modelo se pone PASIVA) representa una fase del modelo en la cual espera por eventos externos mientras hace actividad propia. Con esta finalidad, σ, el instante del próximo evento interno, se pone en infinito (inf).

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. Para expresar una interrupción que necesita un cambio en la planificación, reemplazamos la instrucción continuar por las que manipulan σ, como lo muestra la especificación del port de entrada *parar*. Este evento externo, si indica parada, provoca que el modelo abandone la fase ENVIANDO y aborte la transmisión actual; si indica comienzo, la transmisión se reinicia.

El cambio en la planificación se provoca a medida que σ cambia de un valor finito a infinito. Supongamos que el tiempo ya gastado en preparar la salida no necesita ser repetido cuando se reasume la transmisión, almacenamos el tiempo restante (σ - e) en tiempo_restante para restaurar σ antes de entrar a la fase ENVIANDO. Si hay varios trabajos a la vez suspendidos como este, habríamos ahorrado un tiempo de tiempo_restante con cada uno.

Ejemplo 9

Construyamos un modelo para comparar desempeño de sistemas mono/multiprocesador, basado en dos medidas: turnaround y throughput. Para ello comenzamos haciendo un modelo atómico para estudiar la desempeño de arquitecturas simples. Para modelar un monoprocesador, construimos un modelo atómico llamado P, en el que solo representamos el tiempo que toma completar un trabajo.

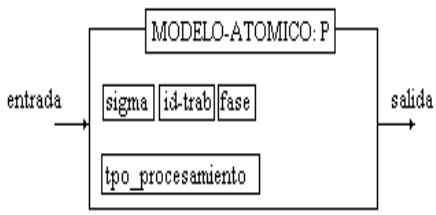

Figura 9. Un modelo atómico de desempeño de un procesador.

MODELO-ATOMICO: P
variables de estado: sigma=inf; fase=pasiva; id-trab;
parámetros: tpo_procesamiento = 5;

Transición externa

según port
  entrada: **según** fase
    pasiva: almacenar id_trab
      mantener ocupado tpo_procesamiento
    ocupado: continuar
  otro: error

Transición interna

según
  ocupado: pasivar
  pasiva: /* nunca ocurre */

Salida
  enviar id_trab al port *salida*

El manejo de trabajos que llegan se modelan con la función de transición externa. Cuando el procesador está libre, cambia la fase y se pone en pasiva. Cuando llega un trabajo por el port *entrada*, se

guarda su id_trab y se simula que el procesador trabaja (mantener ocupado tpo_procesamiento, que pone fase en ocupado, y sigma en tpo_procesamiento). Si llega otro trabajo, lo ignora (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. Lo primero que se ejecuta es la función de salida. Después de la función de salida se llama a la función de pasaje de estado, que contiene la frase "pasivar", que pone el modelo en libre (fase=pasiva, sigma=inf). P tiene dos variables de estado: id_trab y tpo_procesamiento. Tpo_procesamiento está fija: es un parámetro del modelo. 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.

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. Sigma guarda el tiempo restante hasta el próximo evento interno (el valor de avance de tiempo a ser producido por la función de avance de tiempo).

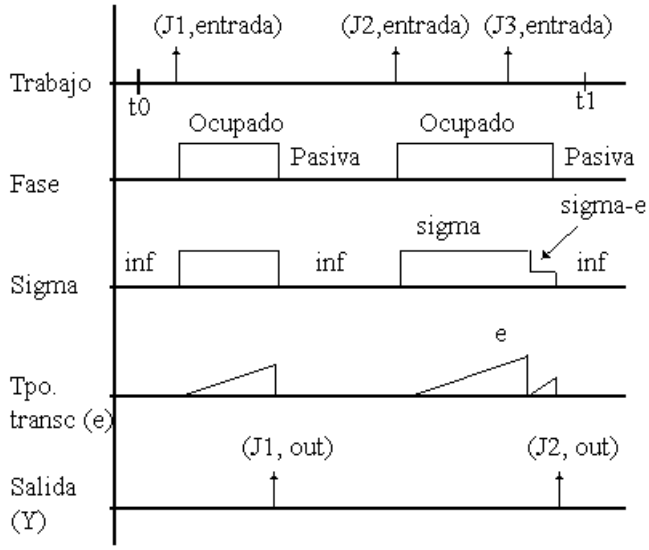

Figura 10. Ejecución del modelo del ejemplo 9

2.2. Modelos acoplados

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

$$DN = <X_{self}, Y_{self}, D, \{Mi\}, \{Ii\}, \{Zij\}, select>$$

D: conjunto de componentes; para cada i en D U {self},
Mi es un modelo componente básico;
Ii es el conjunto de influenciados de i; y para cada j en Ii,
Zij es la función de traducción de salida i a j.
Select es una función, el selector de tie-breaking (función de secuencialización o prioridad).

Esta estructura está sujeta a las restricciones de que para cada $i \in D$,

$$Mi = <Xi, Si, Yi, \delta i, \lambda i, tai>$$

es un modelo atómico;
$Ii \subseteq D \cup \{self\}$; $i \notin Ii$;
$Z_{self,j}$: X_{self} -> Xj, $Z_{i,self}$: Yi -> Y_{self};
Zij: Yi -> Xj.
$select \subseteq D\text{->}D / \forall E \neq \{\varnothing\}, select(E) \in E$.

Para todo j en Ii, Zij es una función, traducción de salida de i a j, Zij: Xi -> Xj. La conexión de i a j está especificada designando j como influenciado por i proveyendo una función de traducción Zij desde el estado de i hasta el conjunto de entradas de j. Cuando un evento interno ocurre en i, envía una señal al componente j en el mismo instante.

Un modelo acoplado dice como acoplar (conectar) varios modelos componentes para formar un nuevo modelo. Este modelo puede ser empleado como componente, permitiendo construcción jerárquica. Un modelo acoplado contiene la siguiente información:

. Conjunto de componentes
. Para cada componente, sus influencias
. El conjunto de ports de entrada a través de los cuales se reciben eventos externos
. El conjunto de ports de salida a través de los cuales se envían eventos externos
. La especificación de acoplamiento, consistente de
- El acoplamiento de entrada externa que conecta los ports de entrada del modelo acoplado a uno o mas de los ports de los componentes (dirige las entradas recibidas por el modelo acoplado a los modelos componentes designados);
- El acoplamiento de salida externa que conecta los ports de salida de componentes a ports de salida del modelo acoplado (cuando una salida es generada por un componente, puede ser enviada a un port de salida designado del modelo acoplado, y transmitirse externamente);
- El acoplamiento interno, que conecta ports de salida de componentes a ports de entrada de otros componentes (cuando una entrada es generada por un componente, puede ser enviada a los ports de entrada de componentes designados, ademas de ser enviados a un port de salida del modelo acoplado).
. La función de selección, que comprende las reglas empleadas para elegir cual de los componentes inminentes (los que tienen menor tiempo al próximo evento) ejecutara su próximo evento.

Ejemplo 10

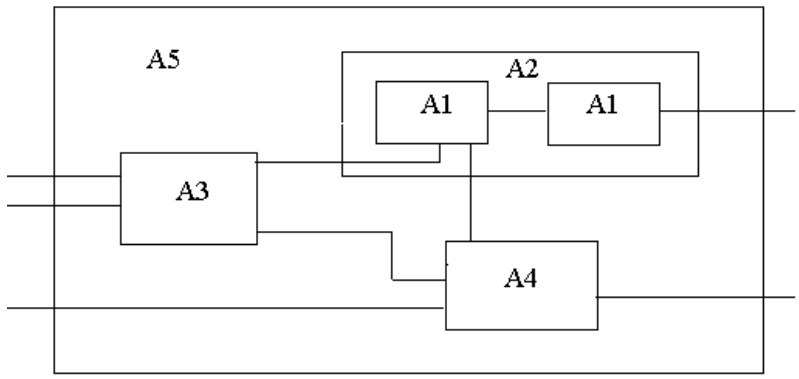

Figura 11 - Un modelo acoplado

Hagamos una descripción jerárquica de un modelo A5 (figura 11). Aquí el modelo A1-A2 está en una forma modular, y puede ponerse en la biblioteca para usarse en descripciones más complejas; el modelo se transforma en un modelo de base. Para construir este modelo seguimos estos pasos: primero se hace una descripción comportamental de los modelos atómicos A1, A3, A4, y se almacenan en biblioteca. A continuación se hace descripción estructural del modelo acoplado A2, almacenamiento en biblioteca, A2 se transforma en un modelo de base, y finalmente la descripción estructural del modelo acoplado A5. Todos los modelos creados se almacenan en biblioteca.

Un modelo multicomponente DN puede ser expresado como un modelo equivalente básico en el formalismo DEVS. Tal modelo básico puede ser empleado en un modelo multicomponente mayor. El formalismo es cerrado bajo acoplamiento, como se requiere para construcción de modelos jerárquicos. El expresar un modelo multicomponente DN como un modelo básico equivalente captura los medios por los cuales los componentes interactúan para lograr el comportamiento general. Un modelo acoplado agrupa varios modelos DEVS en un compuesto que puede verse como otro modelo DEVS, debido a la propiedad de clausura. De esta forma, un modelo puede construirse jerárquicamente a partir de submodelos DEVS.

Un modelo DEVS que no se construye usando un modelo acoplado es un modelo atómico. Un modelo de base es un modelo atómico (descripción jerárquica a dos niveles) o un modelo acoplado (descripción jerárquica a n niveles). Un modelo acoplado almacenado en una biblioteca se transforma en un modelo de base.

Ejemplo 11

Sea un modelo acoplado que consiste de un generador, GENR, y un transductor TRANSDR. El generador genera una secuencia de identificadores de trabajos espaciados en el tiempo. ("mantener activo tpo_entre_llegadas"), que vuelve al modelo a la fase ACTIVE 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, pero podría variar). Antes de que la transición interna ocurra, se produce la salida de un símbolo id_trab. Como GENR es un modelo DEVS, puede testearse 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, para las salidas del generador.

El transductor mide dos índices: el throughput y el tiempo promedio de turnaround. Para calcular medidas de desempeño, el transductor ubica los id_trabs que llegan en su port de entrada *llegada* en su lista_de_llegados junto con sus tiempos de llegada. Cuando el id_trab aparece en el port de entrada *resuelto*, TRANSD lo ubica en la lista_de_resueltos y también calcula el turnaround. TRANSD mantiene su propio reloj local para medir 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, en sigma y e.

MODELO ATOMICO: GENR

variables de estado: sigma = inf;
fase = activa

parámetros: tpo_entre_llegadas = 10

Transición externa

case port *entrada*
parar: pasivar
sino: error

Transición interna

case fase
activa: mantener activa durante
tpo_entre_llegadas
pasiva: /* nunca ocurre */

Salida

case fase
activa: enviar nombre_al_azar al
port *salida*
pasiva: /* nunca ocurre */

MODELO ATOMICO: TRANSD

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

Transición externa

avanzar reloj local h/que coincida con global
case port
llegada: agregar el trabajo a lista_llegada
almacenar tiempo_de_llegada
resuelto: buscar tiempo de llegada del trabajo
total_ta = clock - tiempo de llegada
agregar el trabajo en lista_resueltos
continue

Transición interna

case fase
activa: pasiva /* fin del intervalo de obs. */

Salida

case fase
activa:turnaround_promedio= total_ta/trab_res
throughput = trab_res / clock
otro: no hay salida

La especificación de la interconexión de componentes para construir un nuevo modelo tiene tres partes:

. Ports de entrada, que especifican cómo los ports de entrada del modelo acoplado son identificados informando a sus componentes;

. Ports de salida: especifica cómo los ports de salida del modelo acoplado son identificados informando a sus componentes;

. conexiones internas: especifica cómo los ports de salida de los componentes internos están relacionados con los ports de entrada de los otros.

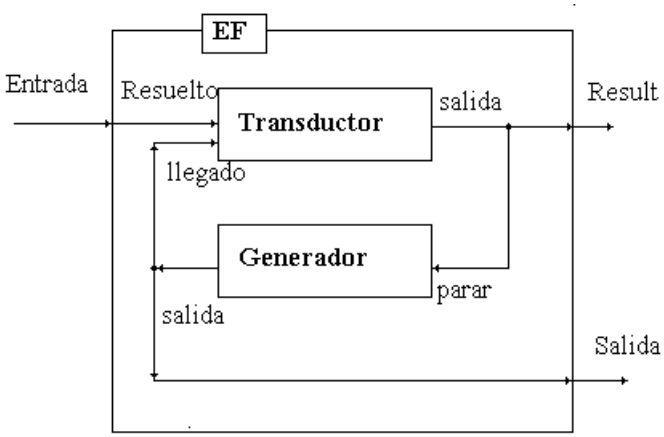

Figura 12 - Modelo acoplado EF

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 -> GERN.parar

Para experimentar con el modelo monoprocesador P, lo acoplamos directamente con el componente EF para formar el digrafo EF-P. Notar que acoplar un marco experimental y un modelo forma un sistema cerrado, libre de entradas. Luego, no hay acoplamiento de entrada externa en EF-P. El acoplamiento de salida externa hace que la salida del transductor este en Result de EF-P. El acoplamiento interno es muy simple: los ports 'salida de cada uno de los componentes se acoplan a los ports 'in de los otros. El marco experimental EF puede acoplarse a cualquier otro modelo de esta forma. Por supuesto, para que tenga sentido, el modelo tiene que usar su port de entrada para recibir identificadores de trabajo, y producir identificadores de trabajo en su port de salida. Esto será verdad para una selección de modelos de arquitectura que discutiremos. Usados de tal forma, el marco experimental generara trabajos para la arquitectura y medirá su throughput y turnaround.

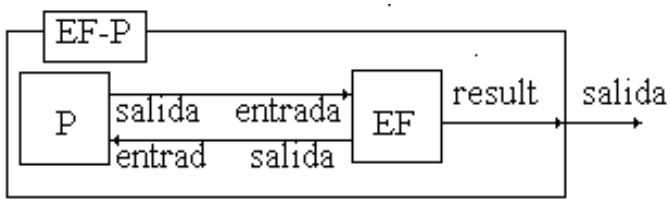

Figura 13 - Modelo acoplado EF-P.

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)

La función select contiene reglas para romper lazos con la planificación. Ej.: en EF-P, si 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 simultáneamente. En particular, si el generador de tpo_entre_llegadas y el tiempo de procesamiento son iguales, todos los segundos trabajos se pierden si el generador ejecuta su próximo evento primero: 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 cuando existe el lazo.

2.3. Simulador abstracto

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. En el concepto de simulador abstracto, la simulación de los modelos atómicos y acoplados es llevada a cabo por distintos procesadores llamados simuladores y coordinadores. Básicamente la simulación es disparada por la recepción de mensajes de simulación 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.

La simulación de un modelo estructural jerárquico se hace con tres clases de procesadores;

. Simulador: trata los modelos atómicos. A cada modelo atómico se asocia un simulador con el rol de mandar la ejecución de distintas funciones del modelo.

. Coordinador: trata los modelos acoplados. A cada modelo acoplado se asocia un coordinador con el rol de comandar los controladores de los modelos que constituyen el acoplado.

. Coordinador raíz: genera la simulación ligada a los coordinadores de mayor nivel.

Cada simulador comunica únicamente con el controlador del nivel superior (padre). Al tope de la jerarquía se encuentra el coordinador raíz que distribuye los mensajes a sus hijos. Un coordinador raíz maneja toda la simulación y esta encadenado con el coordinador del modelo acoplado exterior. La simulación procede por medio de mensajes pasados entre los procesadores que pueden acarrear información con respecto a eventos internos y externos, así como necesidades e sincronización. Los mensajes tienen campos para fuente de origen, hora (acarreando timestamps locales o globales, dependiendo del uso), y un contenido consistente de port/valor, ambos determinados por funciones de salida del modelo atómico. Hay 4 tipos de mensajes: ***, X, Y, listo**.

La simulación se efectúa por pasaje de mensajes entre los distintos procesadores. Un mensaje tiene los siguientes campos: fuente, nombre del emisor, hora (local o global) y contenido (puerto destino y valor asociado, para modelos atómicos).

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

- Un coordinador lo transmite al simulador vía un método de traducción.
- Un simulador llama a la función de transición externa del componente considerada, y responde con un mensaje **listo** que indica que la transición de estado ha terminado y la hora prevista del próximo evento interno.

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

- Si se trata de un coordinador transmite el mismo mensaje a su hijo con el menor tiempo de próximo evento, llamado hijo inminente. Quiere decir que el próximo evento interno será llevado a cabo en su entorno.
- El simulador ejecuta este mensaje provocando el cambio de estado dado por la función de transición interna del modelo atómico, que responderá con un mensaje **Y** y un mensaje **listo** (indica que la transición fue llevada a cabo; acarrea la hora del próximo evento).

Un mensaje **Y** acarrea valores 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 el hijo y sus respectivos ports de entrada al cual hay que mandar el mensaje.

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

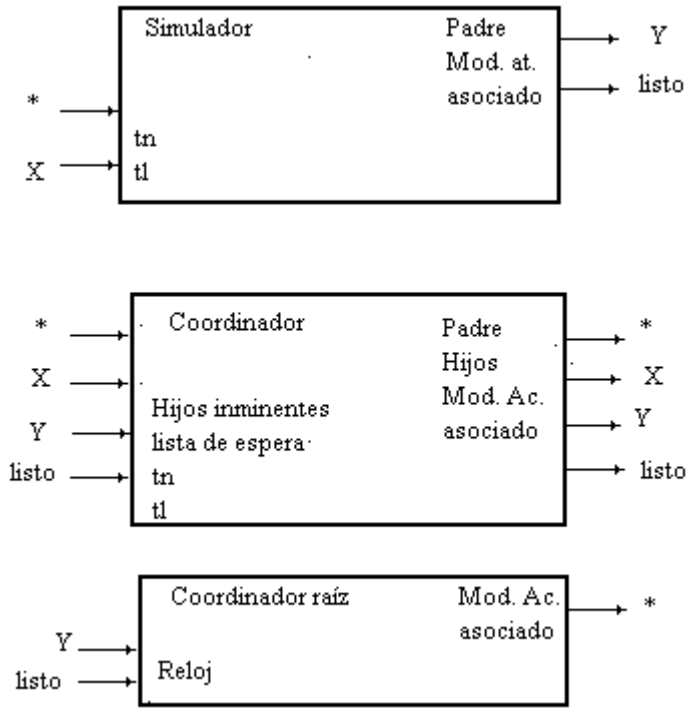

Figura 14 - Procesadores, variables usadas y sus mensajes.

Un modelo atómico es manejado por un simulador que puede recibir dos tipos de mensaje: * (indica que el siguiente evento interno es el de su modelo); y **X** (llegada de un evento externo). Un simulador mantiene varias variables: la hora actual, t; el estado del modelo, s; la hora del último evento tl; la hora del próximo evento interno, tn=tl+ta(s); el tiempo transcurrido desde el último evento, e=t-tl; y el tiempo restante hasta el próximo evento interno, σ = tn- t = ta(s) - e.

Si llega un mensaje *, se chequea si su hora coincide con tn (debe ser así porque un * solo llega al simulador si su modelo es inminente), y aplica la función de salida. Se pone el contenido en un mensaje **Y**, y se lo manda al nivel superior. Se aplica la función de transición interna para actualizar el estado s, y se actualizan las horas de eventos: tn=tl+ta(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.

Un mensaje **X** provoca que se chequee si la hora que lleva el mensaje **X** $\in$ [t1, tn] (debería ser así porque un mensaje **X** sólo debe llegar luego del ultimo evento y antes del próximo interno). Se actualiza el tiempo transcurrido: e = t-tl. Se aplica la función de transición externa con valores actualizados de e y **X;** s = δext(s,e,x), y se actualiza las horas de eventos: tn=tl+ta(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+σ (σ es el mínimo de los tiempos remanentes, tai(si)-ei, entre los componentes i de DN) eligiendo uno usando la función select. Sea i* este componente elegido (inminente). En el momento t+σ, justo antes que

i* cambie de estado, calcula su salida y*=λi*(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 xi*j es Zi*j(y*). El tiempo transcurrido en cualquier componente i en el tiempo t+σ es e'i=ei+σ. Un influenciado j responde al evento externo generado por i* aplicando su función de transición externa, para obtener el siguiente estado s'j = dext(sj, e'j, xi*j) y pone su tiempo transcurrido en 0. Otros componentes que no están en el conjunto de influencia no son afectados por la activación de i* excepto que su reloj de tiempo transcurrido se incrementa en σ. Finalmente, el componente i* ejecuta su transición interna yendo al estado si*'=dint(si*), y poniendo su tiempo transcurrido en 0.

Ejemplo 12

Veamos la asociación del simulador abstracto con el modelo EF-P, que se ve en la figura 15. Aquí los círculos representan los procesadores y los rectángulos los modelos asociados.

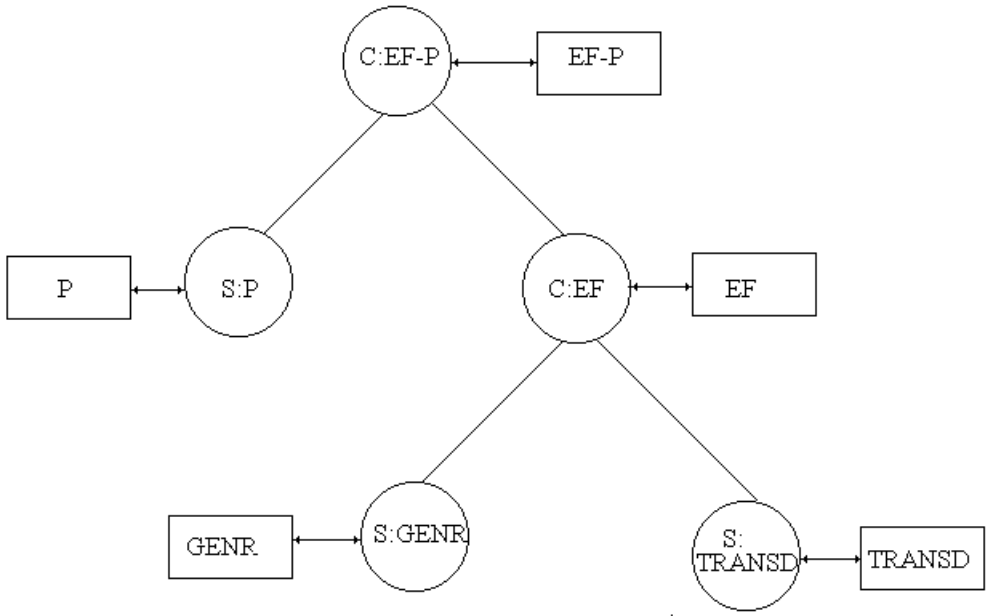

Figura 15 - Simulador abstracto para el modelo EF-P.

El ciclo de simulación comienza por un mensaje * enviado por el coordinador raíz al coordinador del modelo exterior. Aquí C:EF-P recibe el mensaje * y lo transmite a su hijo inminente. Para C:EF-P, el hijo inminente puede ser EF (cuando es hora de generar un nuevo trabajo), o P (cuando un trabajo ha sido procesado). Para C:EF, GENR es el inminente hasta que TRANSD se hace inminente para parar la simulación. Cuando S:GENR recibe un mensaje * manda un mensaje **Y** a su padre C:EF (aplicando su función de salida para producir un contenido con port *salida* y un valor id_trab). S:GENR también provoca la ejecución de la función de transición interna GENR, y manda un mensaje **listo** con el tn de GENR. Para C:EF, cuando GENR es inminente, el mensaje **Y** que se pone en el port *salida* se traduce a un mensaje **Y** en el port *llegada* y se envía a su influenciado, TRANSD.

Para C:EF, cuando GENR es inminente, el mensaje y que pone en el port *salida* aparece como un mensaje **Y** en el port *salida* de su padre EF. 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 libre, se pone en ocupado. Cuando el tiempo de procesamiento ha terminado, S:P genera un mensaje y (un id_trab en el port *salida*) que va a su padre C:EF-P y luego 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, y traduce el port *entrada* a *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 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 son {(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.

El simulador abstracto es genérico, o sea que trabaja con cualquier modelo en la clase o sus subclases. La interfaz entre modelo y su simulador es definida por métodos representando consultas y operaciones que el simulador hace al modelo. Los detalles de la estructura interna del modelo se ocultan en esta interfaz. Cualquier modelo puede simularse por un simulador si tiene los métodos de la interfaz. La eliminación de toda la información relacionada con la simulación de la especificación del modelo, nos permite tratar los modelos como conocimiento, es decir, trabajar con una Base de Modelos.

3. MODELOS CELULARES

Los autómatas celulares son un formalismo para modelado de sistemas dinámicos complejos, de variables y tiempo discretos (el tiempo, espacio y estados del sistema son discretos), creados originalmente por von Neumann y S. Ulam. Un autómata celular es un conjunto infinito n-dimensional de celdas ubicadas geométricamente. Cada punto de la grilla espacial infinita (llamado una celda) puede tener uno de un conjunto de valores finitos de estados, es decir que cada celda tiene un estado elegido de un alfabeto finito. Cada una contiene el mismo aparato de cálculo que las otras y se conectan entre sí de forma uniforme. El estado de las celdas se actualiza simultáneamente y de forma independiente de los demás en pasos de tiempo discreto. Para ello se define la vecindad de una celda, que es un conjunto de celdas cercanas. Esta vecindad es homogénea para todas las celdas

Los estados de las celdas en la grilla se actualizan de acuerdo con una regla local. O sea, 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. Todas las celdas de la grilla se actualizan sincrónicamente. Luego, el estado de toda la celda avanza en pasos de tiempo discreto. Las celdas actualizan sus valores usando una función de transición, que toma como entrada el estado actual de la celda, y un conjunto finito de celdas cercanas, que están a una distancia limitada (la vecindad). La uniformidad en el espacio significa que el sistema posee invariancia espacial y en tiempo.

Pueden usarse para modelar variedad de sistemas discretos naturales, pero también sirven para comprender muchos formalismos comunes que pueden verse como extensiones de concepto básico. Se suelen usar, al igual que las ecuaciones diferenciales para el modelado de sistemas físicos. Existen autómatas para modelar dinámica en fluidos y gases, colonias de animales, modelos ecológicos, entre otros. También tienen aplicación en criptografía. Permiten especificar modelos de sistemas complejos con distintos niveles de descripción, por lo que permiten atacar mayor complejidad que las ecuaciones diferenciales, permitiendo el modelado y estudio de sistemas muy complejos.

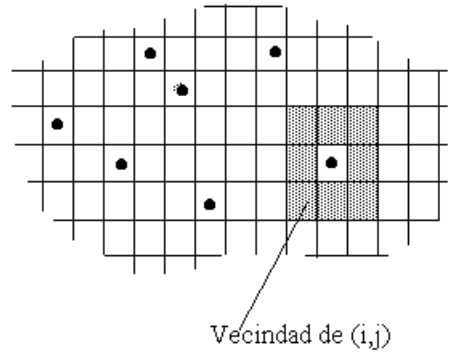

Figura 16 - Un fragmento de autómata celular bidimensional.

Luego, un autómata celular se caracteriza especificando tres parámetros: un conjunto de estados S, una vecindad N, y una función de transición T. Los vecinos de una celda se determinan usando la vecindad N (es un par ordenado finito de enteros, que se suman al par (i,j) para obtener los vecinos). Un estado global en un instante t (una asignación a cada celda del conjunto S), cambia de estado en el instante t+1. Simultáneamente, para cada celda, se aplica la función de transición T a los estados. Los vecinos de una celda son los que influencian su próximo estado (sus influenciadores). Un autómata celular es un sistema discreto autónomo (sin entradas), aunque existen extensiones que permiten entradas. Otras extensiones permiten que cada celda pueda tener distintas reglas, y en estos casos el autómata celular es inhomogéneo. El modelo permite distintas reglas y conexiones (con aplicaciones a la biología teórica).

Formalmente, un autómata celular se define como un conjunto CA=<S,N,T>. Sea $a_{i,j}(t)$ el valor de la celda i,j en el instante t (autómata celular bidimensional). Luego, el autómata evoluciona de acuerdo con la función:

$$a_{ij}(t+1) = T[a_{i-r,j-r}(t),..., a_{i,j}(t),..., a_{i+r,\ j+r}(t)]$$

donde T denota la regla de transformación del autómata, y cada sitio está especificado por un entero aij(t) en el rango [0,k-1] en el instante t. El parámetro de rango r especifica la región afectada por un sitio dado.

El autómata especificado por CA, Mca, se define:

Mca=<Q, d>, donde Q={q / q:IxI ->S} y d:Q->Q. d(q) (i,j)=T(q/N+(i,j)) para cada $q \in Q$ e $(i,j) \in$ IxI.

Notar que el conjunto de estados de Mca es el conjunto de estados globales del sistema. Un estado global q es una asignación a cada una de las celdas de un elemento de S, que se llama un estado local. La función d de Mca se llama la función de transición global (y la T, la local).

Los autómata celulares son discretos, regulares y sincrónicos. Muchas de las características únicas de la dinámica de los autómatas celulares pueden relacionarse con la actualización sincrónica de estados de celdas. El comportamiento colectivo parece ser estable a toda clase de perturbaciones del modelo excepto al abandonar las actualizaciones sincrónicas. La organización de las subunidades en un modelo deben aproximar la organización de las subunidades en el sistema a modelar, y la dinámica del modelo debe ser una buena aproximación de la dinámica en el mundo real.

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.

A pesar de ser una herramienta poderosa, los autómatas celulares tienen algunas restricciones importantes. Por un lado, 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: todas cambian de estado simultáneamente. Otro problema es que el espacio de celdas es infinito. 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: 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.

Por otro lado, el gran número de celdas hace que en general se estén haciendo cálculos innecesarios. Para evitarlo se usa una aproximación que detecta si una celda no puede cambiar de estado. Esto es simple de lograr, ya que una celda no puede cambiar de estado en su transición si ninguno de sus vecinos lo han hecho. Llamemos un evento en una celda un cambio de estado en una celda. Podemos decir que en un autómata celular, los eventos pueden ocurrir en un numero pequeño de ellas (aunque es posible que ocurra en todas simultáneamente), y detectamos si ocurrirá un evento en una celda mirando si hubo eventos en sus vecinos.

Se puede mantener una lista de últimos eventos, que contiene las celdas cuyos eventos ocurrieron en t. En base a la misma, podemos determinar en que celdas ocurrirán eventos en t+1. Luego que simulamos cada una de estas transiciones, chequeamos para ver si hubo un cambio de estado; si es así ponemos la celda en la lista de últimos eventos para t+1.

Para iniciar el procedimiento, hay que tener un estado inicial. Si el modelo es finito con un numero finito de celdas, podemos poner a todas las celdas en la lista de próximos eventos. Una mejor aproximación es detectar estados especiales, llamados estados latente, en el que si un elemento $s \in S$ / si una celda y sus vecinos están en el estado s en el instante t, la celda quedara en el mismo estado en t+1 (o sea, la función T(s,...,s)=s). Entonces es simple iniciar la simulación, ya que solo las celdas que no están en estado latente deben ponerse en la lista de próximos eventos.

Además, si hay solo un conjunto de celdas en estado no latente, se puede simular un modelo con un numero infinito de celdas.

En algunos casos, para modelos finitos, el procesamiento extra de la lista puede no valer la pena. Para un modelo en el cual todas las celdas están llevando a cabo acciones todos los tiempos, no es bueno usar esta estrategia. Para ver la relación costo/beneficio hay que ver el nivel de actividad del modelo (relación entre celdas que llevan a cabo actividad sobre el total de celdas). Este nivel de actividad es una medida del paralelismo del modelo.

Aunque los autómatas celulares representan un modelo de resolución elegante para un gran número de problemas complejos, la investigación cuantitativa de las propiedades básicas de la evolución de tiempo y estado de los autómatas celulares está comenzando. Sin embargo, se han obtenido algunos resultados importantes. Por ejemplo, en [Wol84] pudo mostrarse que la mayoría de los autómatas celulares pueden clasificarse en cuatro clases:

I - La evolución de todos los estados iniciales conduce, luego de un tiempo finito, a un estado homogéneo, donde todos los sitios tienen el mismo valor.

II - La evolución conduce a un conjunto de estructuras separadas estables o periódicas.

III - La evolución conduce a un patrón caótico (ejemplo: sistemas físicos).

IV - La evolución conduce a estructuras complejas localizadas, generalmente metaestables (ejemplo: un modelo para el Juego de la Vida).

3.1. Modelos de autómatas celulares asincrónicos

Como dijimos, un problema de los autómatas celulares es que son modelos a tiempo discreto, lo que en muchos casos reduce la eficiencia de las simulaciones y el grado de paralelismo del modelo. La idea es tener un modelo de autómatas celulares con relojes asincrónicos, y hacer que el modelo sea manejado por eventos. Las transiciones de estado de cada celda es independiente de las otras del autómata, y el autómata puede comportarse de forma muy inhomogénea. Si usamos relojes independientes, el tiempo de iteración para cada celda puede ser diferente. Definiremos un autómata celular asincrónico como:

$$Z = \langle Id, N, V, v0, f, F, T \rangle$$

Id es el conjunto de enteros de d-tuplas, llamado un arreglo del espacio celular. Un elemento de Id representa las coordenadas de una celda o una celda en esa coordenada. El entero positivo **d** se llama la dimensión del espacio celular. **N** es un n-tupla de distintos elementos de Id, llamado el índice de vecindad. Con N=(n1,..., nn) y dada una celda a, un elemento del conjunto {(a+n1),..., (a+nn)} se llama una celda de vecindad de a, y a es la celda central de las de la vecindad. El conjunto consistente de elementos de N se llama la vecindad. El espacio celular es homogéneo, luego, para todas las celdas, **V** es un conjunto finito no vacío, llamado un conjunto de estados. El conjunto V tiene un elemento, **v0**, en el cual la celda está en descanso, llamado un estado latente. La función local **f** es un mapeo de Vn a V y satisface la propiedad que f(v0,..., v0) = v0. El siguiente estado de a estás dado por st(a) = **F**(a, N, f, t),

donde f es instantánea aplicada a la vecindad N de a en el tiempo t. El tiempo de la evaluación de cambio de próximo estado de a se describe por t'=**T**(a, N,t), donde t'> t.

Un autómata celular asincrónico no determinístico puede obtenerse introduciendo un experimento aleatorio y definiendo una variable aleatoria en el espacio de muestra del experimento.

La función local no determinística f se aumenta con S, que es una implementación de la variable aleatoria. La función local se escribe como fS, donde uno puede pensar a fS como una función tabulada dependiendo de S, o si la variable aleatoria es discreta, S puede considerarse como argumento a la función. De la misma forma, la función de incremento de tiempo puede escribirse como TS, que puede ser una función tabulada dependiente de S, o, para variables continuas o discretas aleatorias, S puede considerarse como argumento a la función TS. El uso de S para variables aleatorias continuas es válido ya que el tiempo es continuo en los autómatas celulares asincrónicos.

3.2. Modelos NEVS

Zeigler [Zei90] también definió una extensión a su formalismo DEVS, que permite plantear modelos DEVS celulares. Un espacio de celdas de próximo evento (NEVS) esta especificado por una 4-upla, <S,N,T,SELECT>. Nuevamente, consideramos una grilla infinita de celdas. El estado de cada celda es un par (s,σ), con s en S, σ en R^+. Una celda en (s,σ) se queda en ese estado durante un tiempo σ antes de hacer una transición inducida por ella misma. Sin embargo, en el medio puede llegar un mensaje que altere este estado, como resultado de la acción de otra celda. Luego, s y σ se llaman el estado secuencial y el tiempo restante del estado. Una celda c en este estado en el momento t, se dice que puede ser expresada como que esta en el estado secuencial s y ha sido planificada para una transición en el momento σ (o $t+\sigma$).

Nuevamente los vecinos se calculan agregando sus coordenadas al vecindario prototipo N (otro conjunto de pares finitos). Cuando ocurre una transición de eventos a una celda, ellas y sus vecinos hacen un cambio de estado como lo dicta la función de transición T. Luego, en contraste con el autómata, T toma una lista de pares (s,σ) - todos los estados de la celda y sus vecinos antes del evento - y produce una lista de esos pares, dando los estados totales de estas celdas justo después de la transición.

En contraste con el autómata celular, la actualización de un modelo NEVS tiene lugar en instantes de calculo espaciados irregularmente. Sea ti tal instante, y su sucesor ti+1 calculado así:

Imaginar el estado global en ti como una lista de pares (s,σ) con los estados totales de todas las celdas en este momento. Sea σ^* el mínimo de todos los σ en la lista. Luego, $ti+1=ti+\sigma^*$. El estado global en ti+1 se calcula así: llamar **inminentes** a las celdas cuyos σ es igual a σ^* en el instante ti. Estas celdas están planificadas para un evento en ti+1. Si hay mas de una, aplicar SELECT y elegir una, que llamamos c*. Esta celda lleva a cabo la función de transición como se indica antes, resultando en un nuevo estado global. Los estados totales de los vecinos de c* se calculan usando T; el estado total de las otras se convierte de (s,σ) a $(s,\sigma-\sigma^*)$.

Ejemplo 13

Consideremos el movimiento de un solo cuerpo en un plano. Existe un cuerpo en un plano, que avanza, siempre hacia la derecha. Cada celda del autómata tiene, por ende, dos estados posibles (presencia o ausencia). S={0,1}. Una celda pasiva nunca se pondrá activa por si misma; esta representada por (0,inf). Se mueve hacia la derecha, luego 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 (seudocódigo)
. Poner estado secuencial en 0. Pasivar.
. Prender el estado del vecino (1), y planificar una transición en TIEMPO_DE_MOVIMIENTO.

En este caso no existen conflictos entre submodelos, por ende no es necesario incluir la función Select.

La especificación de autómatas celulares no pone restricciones en su estado global inicial. Sin embargo, hay que hacerlo para que se pueda simular en una computadora. Una forma de hacerlo es pedir un estado local latente, en el cual no estén algunas celdas (numero finito). La idea es que la mayoría del espacio estará latente y se quedara así, y habrá solo un número finito de celdas capaces de abandonar este estado a la vez.

Ejemplo 14

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, tenemos 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 ausencia/presencia de un automóvil. Ponemos en N todas las celdas arriba y abajo del centro, para que el coche se mueva en esas direcciones. Si N esta ordenado en el orden legal de pasar:

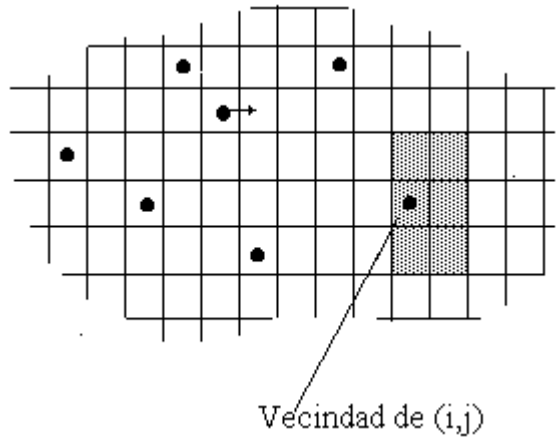

Figura 17 - Vecindad del ejemplo

Un seudocódigo de la función de transición interna es la siguiente:

. 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.

Aquí la función select determina cual de dos autos moviéndose al mismo cuadrado al mismo tiempo lo lograra. Luego, el orden en el cual se eligen las celdas hace una diferencia. 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 coches 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 coches se muevan al azar en vez de a velocidad constante.

Un estado local s es un estado latente cuando los vecinos de una celda están en s en el momento t, luego la celda estará en s en t+1. O sea, s es latente si s=T(s,...,s). Todo estado local del modelo de difusión satisface este criterio, porque el promedio de una constante es la constante. Si inicialmente todas las celdas están en el mismo estado latente, el sistema persistirá en el en el tiempo (estado de equilibrio). Mas interesantes son los estados globales en los cuales hay conjuntos finitos de celdas no latente. Debido a la finitud de la vecindad, estos estados evolucionan hacia estados globales con la misma propiedad. Mas aun, si se conoce el estado latente, la mayoría de las celdas que están en este estado en cualquier momento no tienen que ser examinadas por un simulador. Esto hace que la simulación sea computable.

Luego una celda puede cambiar de estado del momento t al t+1 solo si hay una celda en su vecindario que ha cambiado de t-1 a t. El simulador tiene que saber en cada paso que celdas cambiaron para determinar las celdas que pueden cambiar de estado en el siguiente paso en estas circunstancias. La vecindad inversa, -N, es el conjunto de imágenes espejo de los elementos en N (ej., (i,j) en -N si y solo si (-i, -j) en N).

Lema: una celda (i,j) esta en el vecindario de una celda (k,l) si y solo si (k,l) esta en el vecindario inverso de (i,j). Una celda puede cambiar de estado de t a t+1 si y solo si esta en el vecindario inverso de una celda que cambio de estado de t-1 a t. Luego el conjunto de celdas que pueden cambiar de estado entre t y t+1 es la unión de los vecinos inversos del conjunto que cambiaron de t-1 a t.

Como en el caso de autómatas celulares, los estados iniciales globales deben ser restringidos para computabilidad. Aquí el rol del estado latente es jugado por el estado pasivo. Una celda es pasiva si su tiempo restante es igual a inf. Un estado global inicial puede consistir a lo sumo de un conjunto finito de celdas no pasivas.

4. SIMULACION PARALELA

Hasta ahora hemos estudiado distintos formalismos para sistemas de eventos discretos, con utilidad para hacer simulaciones. En muchos casos, la ejecución de estas simulaciones es poco eficiente. El principal problema de velocidad tiene 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. Para evitar estos problemas, se puede intentar usar simulaciones que no requieran la generación de un número grande de ciclos de simulación. Algunos métodos llamados de reducción de varianza tratan de que el número de replicaciones necesarias de ciclos sea proporcional a la varianza de la estimación aleatoria medida en un ciclo dado. Estos métodos son difíciles de preparar y que la reducción de varianza que proveen no es muy grande (es típica una reducción de un factor de 2 a 5).

Un medio obvio de obtener una simulación mas rápida es dedicar mas recursos. En particular, podemos 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 del 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 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 queremos un procesador con las mismas capacidades que los del sistema multiprocesador, pero estos pueden no tener la suficiente memoria propia para manejar toda la simulación. Luego, el monoprocesador debe tener la misma capacidad que los procesadores en el sistema multiprocesador pero con la suficiente memoria para correr toda la simulación.

Para obtener una medida precisa de la aceleración real el monoprocesador debería ejecutar de la forma más eficiente posible. Sin embargo, en la práctica esto puede ser difícil de determinar. Una alternativa práctica es medir cuánto toma a un solo procesador correr la simulación distribuida, midiendo el tiempo total de ocupado de todos los procesadores durante la simulación distribuida. La medida resultante de aceleración debería ser mayor que la aceleración real obtenible distribuyendo la simulación. También podemos estar interesados en la eficiencia de la simulación, o sea, la aceleración dividida el número de procesadores usados (mide la utilización efectiva de los procesadores).

4.1. Formas de descomposición paralela

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

a) Una aproximación obvia es hacer **replicaciones independientes** de una simulación existente en procesadores separados, y luego recolectar los resultados para análisis estadístico. Esta aproximación es muy eficiente porque no hace falta coordinación entre procesadores. Luego, 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. También podrían hacerse simultáneamente corridas independientes con distintos parámetros. La distribución de experimentos puede no ser posible debido a restricciones de memoria, ya que precisa que todos los procesadores tengan suficiente memoria para contener toda la simulación. Además esta aproximación no permite la descomposición de problemas complejos, que en el caso de simulaciones multiprocesador permite simulaciones más complejas y realistas.

b) Otra forma es aplicar un **compilador paralelizante** a un programa secuencial. Estos tratan de encontrar secuencias de código que se pueden hacer en paralelo y los planifican en distintos procesadores. Es bueno porque es transparente al usuario 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.

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

d) Recordemos que en un modelo discreto sólo se cambia de estado en puntos discretos de tiempo simulado (un evento). Se suele usar una lista de eventos con todos los eventos pendientes planificados, y una variable global reloj para saber el progreso de la simulación. Al usar **eventos distribuidos** se mantiene una lista global de eventos, y cuando hay un procesador disponible, se procesa el próximo evento en tiempo simulado. Hacen falta protocolos 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 distribuidos puede ser razonable cuando hay un número pequeño de procesos y cuando hay un gran número de información global usada por componentes del sistema.

e) La última aproximación es **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 que representan 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 o caminos de los mensajes.

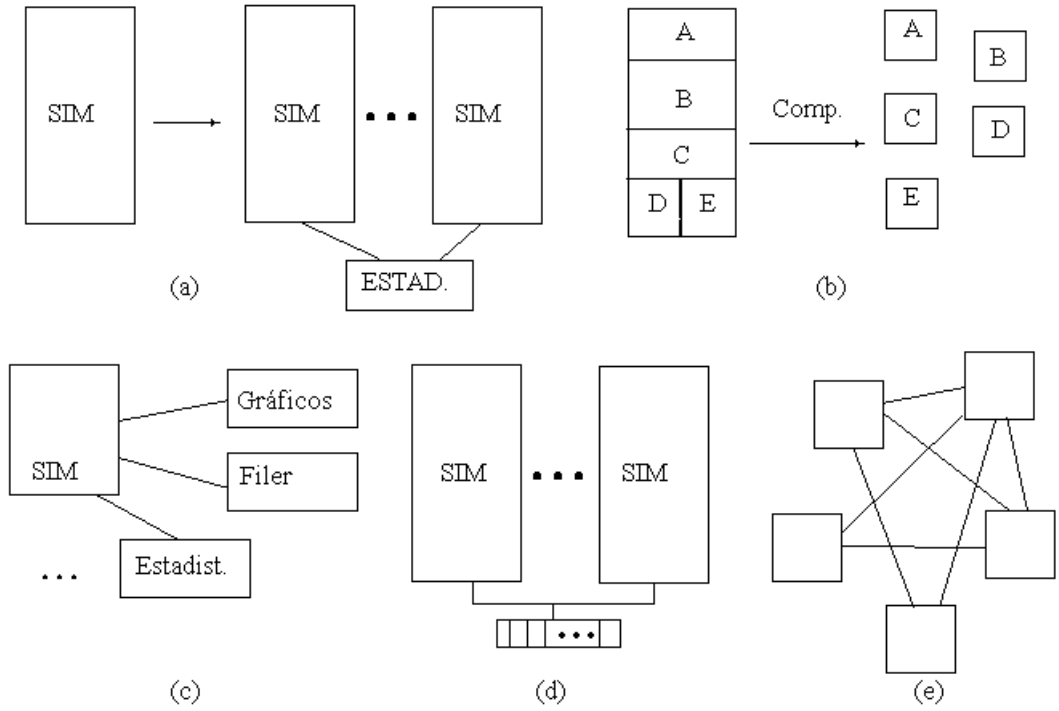

Figura 18 - Mecanismos de descomposición. (a) Replicación independiente; (b) Compilador paralelizante; (c) funciones distribuidas; (d) eventos distribuidos; (e) descomposición del modelo.

4.2. Sincronización en modelos distribuidos

Al distribuir los componentes del modelo, debemos proveer esquemas de sincronización. Las formas para atacar estas cuestiones depende si la simulación es dirigida por tiempo o por eventos, y si la simulación es sincrónica (hay un reloj global y que todos los procesos deben tener el mismo tiempo simulado) o asincrónica (cada proceso tiene su reloj local propio y el tiempo simulado para distintos procesos puede ser distinto). A su vez, la simulación puede ser manejada por tiempo, en la cual el reloj avanza de a un tick y todos los eventos planificados se simulan, o por eventos, donde el reloj se avanza a la hora de la transmisión del mensaje simulado.

a) Simulación dirigida por tiempo

En la simulación dirigida por tiempo, el tiempo simulado avanza en incrementos fijos (ticks), y cada proceso simula su componente sobre cada tick. Los ticks deben ser cortos para garantizar precisión, lo que implica mayor duración de la simulación. La simulación puede ser sincrónica o asincrónica. Si es sincrónica, todos los procesos deben terminar de simular un tick antes que cualquiera pueda comenzar a simular el siguiente tick. Por ende, luego de simular hay una fase de actualización de estado y comunicación. Cuando la simulación es asincrónica un proceso puede empezar a simular el siguiente tick tan pronto como sus predecesores hayan terminado el último tick, y se sincronizan enviando mensajes a sus sucesores. Para implementar un reloj global, se puede usar una aproximación centralizada (un proceso dedicado para actuar como sincronizador) ó de forma distribuida (con un algoritmo de broadcast adecuado).

La simulación asincrónica permite mayor concurrencia, pero a un mayor costo de comunicación (como un procesador no puede simular el siguiente tick hasta que conozca que sus predecesores terminaron de simular el último tick, debe recibir un mensaje de cada uno de sus predecesores para cada tick). Por otro lado, si la simulación es sincrónica, una vez que el reloj global se sincroniza, sólo se precisa transmitir los mensajes señalando interacciones o actualizaciones de estados. Luego, si los estados cambian menos frecuentemente que cada tick, un reloj global puede ser más eficiente.

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 que aún deben simularse. Sin embargo, la

simulación manejada por tiempo evita el overhead extra de sincronización necesario. Luego, puede ser adecuada para sistemas con topología dinámica, y para sistemas en los cuales pasan muchas cosas al mismo tiempo.

b) Simulación dirigida por eventos

Recordemos que en simulaciones dirigidas por eventos, el tiempo se incrementa de un evento al siguiente (evento representa un cambio en el estado), luego puede tener mayor aceleración potencial. También puede ser:

1) Sincrónica: el reloj global se pone en el mínimo tiempo de próximo evento para todos los procesos. Puede haber un único proceso dedicado a sincronizar, o estar distribuido. Hay diversos métodos para mantener el reloj global:

2) Asincrónica: el reloj local para cada proceso se pone el mínimo tiempo de evento para ese proceso. Tiene alta desempeño potencial, ya que los procesos pasan menos tiempo esperando a otros, y los eventos independientes pueden ser simulados simultáneamente, aunque ocurran en distinto tiempo simulado. Estas aproximación ha sido muy difundida debido a que parece ser aquella en la que pueden obtenerse mayores grados de aceleración. En las siguientes secciones estudiaremos con detalle algoritmos para sincronización en esta clase de simulación.

Ejemplo 15

Supongamos el caso de simular eventos en una red como la que aparece en la figura 19. En el caso (a), inicialmente partimos de un reloj global sincronizado (5) y todos los procesos avanzan simultáneamente al siguiente tick de reloj (6). El grado de paralelismo mejora en el caso (b), donde la simulación es asincrónica. Aquí cada proceso puede avanzar un tick si recibe señales de los procesos predecesores, favoreciendo la aceleración en la simulación. La situación mejora aún más en el caso de tener simulación dirigida por evento, ya que el reloj global (GVT) avanza dirigido por eventos y no por reloj, pudiendo acelerar aún mas el procesamiento de eventos. En el caso que la simulación sea sincrónica, los procesos deben sincronizarse entre sí hasta todos tener el mismo tiempo de simulación. El mayor grado de aceleración se logra en el caso (d),donde la simulación es dirigida por eventos y asincrónica, luego las únicas dependencias están dadas por los datos de la simulación en sí.

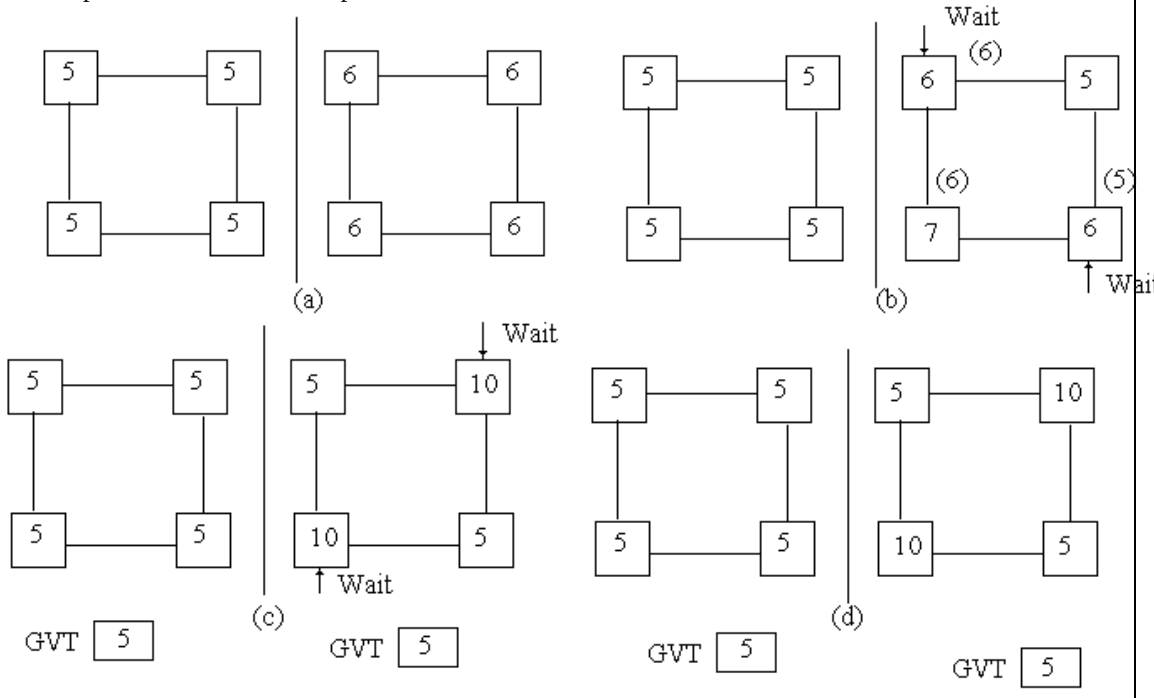

Figura 19 - Simulación paralela dirigida por tiempo: (a) Sincrónica; (b) Asincrónica, y dirigida por eventos: (a) Sincrónica; (b) Asincrónica

4.3. Procesos físicos y lógicos

Volvamos a analizar el funcionamiento de un simulador de eventos discretos: cada evento tiene un timestamp. El simulador saca el evento con menor timestamp de la lista y lo procesa, planificando además nuevos eventos en el futuro simulado. Con este paradigma de ejecución, es crucial que siempre se seleccione el evento con menor timestamp. Si uno eligiera alguno con timestamp mayor, podría modificar variables de estado usadas posteriormente por el evento de menor timestamp, provocando errores de causalidad. En otras palabras, deben satisfacerse ciertas **restricciones de secuencia** para que los cálculos sean correctos.

La mayoría de las estrategias PDES existentes evitan tales secuencias usando una metodología orientada a procesos que prohiba estrictamente que los procesos tengan acceso directo a variables de estado compartidas usando una aproximación asincrónica por eventos. El sistema modelado, conocido como **sistema físico**, se ve como si estuviera compuesto de un conjunto de **procesos físicos** que interactúan en distintos puntos en tiempo simulado. El simulador se construye como un conjunto de **procesos lógicos** P0, P1,..., uno por proceso físico. Todas las interacciones entre procesos físicos se modelan con mensajes enviados entre los procesos lógicos correspondientes.

Asumimos que la simulación consiste de N procesos lógicos P0, ..., PN-1. Clocki se refiere al tiempo simulado hasta el cual ha avanzado Pi. Cuando un evento se procesa, el reloj del proceso se adelanta hasta el timestamp de dicho evento. Si Pi puede mandar un mensaje a Pj durante la simulación, diremos que hay un **enlace** entre Pi y Pj.

Como cada proceso lógico contiene una porción del estado correspondiente al proceso físico que modela, las variables de estado se dividen en un conjunto disjunto, se debe lograr que ningún proceso lógico acceda directamente a más de un estado, lo que es más o menos complejo dependiendo de la aplicación. Por ejemplo, no suele ser difícil para una simulación de una red con colas (un proceso lógico para cada servidor). En cambio, en autómatas celulares la información debe compartirse entre celdas vecinas. En la ausencia de variables de estado compartidas, la aproximación más natural para programar esta simulación es "emular" la memoria compartida construyendo un proceso lógico para cada sector de la grilla, y enviando mensajes de "lectura y escritura" de eventos para acceder a la información compartida. Sin embargo se obtiene baja desempeño porque el overhead del pasaje de mensajes es substancial, y los procesos de la grilla pueden transformarse en cuellos de botella. Una aproximación más eficiente es duplicar la información compartida en los procesos lógicos (celdas) que la necesitan. Como el estado compartido puede ser modificado, se precisa un protocolo para asegurar coherencia entre las diversas copias del estado compartido. Esta aproximación puede ser efectiva para lograr buen desempeño; sin embargo complica significativamente la codificación de la aplicación, haciendo difícil su comprensión y mantenimiento.

Para asegurar que no hay errores de causalidad se debe cumplir la **restricción de causalidad local:** una simulación de eventos discretos consistente de procesos lógicos obedece a la restricción de causalidad local si y solo 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 hay errores de causalidad. Puede no ser necesario, porque dos eventos dentro de un mismo proceso lógico pueden ser independientes, en cuyo caso procesarlos fuera de secuencia no provoca errores de causalidad.

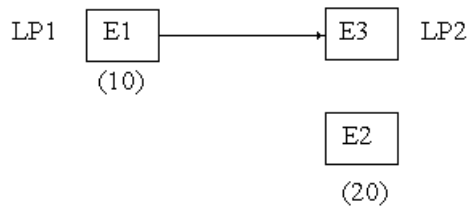

Figura 20 - Error de causalidad por sincronización entre procesos lógicos.

La exclusión de estados compartidos puede provocar errores de causalidad. Consideremos 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 uno no tiene información acerca de qué eventos podrían ser planificados por otros eventos, uno se vería forzado a concluir que el único evento seguro para procesar es el que tiene el menor timestamp, provocando ejecución secuencial.

Desde el punto de vista del sistema físico, la causa siempre debe preceder al efecto. Estas relaciones causa-efecto del sistema físico se transforman en restricciones de secuencia en el simulador (y el simulador puede tener más restricciones que surgen de la forma en que fue programado). La responsabilidad del mecanismo de simulación es asegurar que estas restricciones no sean violadas al ejecutar en paralelo. Podemos decidir si E1 puede ejecutar concurrentemente con E2, pero ¿cómo sabemos si E1 afecta a E2 sin haber ejecutado la simulación de E1? Este es el dilema fundamental en las estrategias de PDES. La secuencia en la cual E1 afecta a E2 puede ser una secuencia compleja de eventos, y depende de los timestamps de los eventos.

Los mecanismos PDES tratan de proveer soluciones a esta clase de problema. En general se usan dos aproximaciones: **conservadoras** y **optimistas**. Las conservadoras **evitan** la posibilidad de errores de causalidad, basándose en alguna estrategia para determinar cuándo es seguro procesar un evento. Por otro lado, las aproximaciones optimistas usan una solución de **detección y recuperación**: los errores se detectan, y se invoca un mecanismo de rollback para recuperarse.

4.4. Mecanismos conservadores

Los primeros mecanismos de simulación distribuida fueron conservadores. El problema básico de estos mecanismos es determinar cuándo es seguro procesar un evento. Más precisamente, si un proceso tiene un evento no procesado con un timestamp (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, porque no habrá una violación de la restricción de causalidad.

En el algoritmo presentado por [Cha79], un proceso con un mensaje no procesado en todos sus enlaces de entrada pone su reloj local en el mínimo de todos los timestamps y procesa todos los mensajes con ese timestamp. Todos los mensajes futuros tendrán timestamps posteriores al reloj local ya que llegan en orden cronológico en cada enlace. El procedimiento se repite mientras haya mensajes no procesados en los enlaces de entrada. Si un enlace con el menor timestamp no tiene mensajes, entonces se bloquea (porque puede recibir un mensaje con timestamp menor a todos los otros).

Estas aproximación puede provocar abrazo mortal si hay un ciclo de procesos bloqueados. En general, si hay relativamente pocos mensajes de 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 [Cha81].

Ejemplo 16

Supongamos una simulación con cuatro procesos lógicos como en la figura 21. 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 fig. 21a. P2 procesa el mensaje (2,m2) y produce (5, m3) que se manda a P4, como se ve en la fig. 21b. En este punto, P2 no puede procesar el mensaje (3,m1) porque P2 está bloqueado por P3. P3 también está bloqueado por P2: por ende hay abrazo mortal.

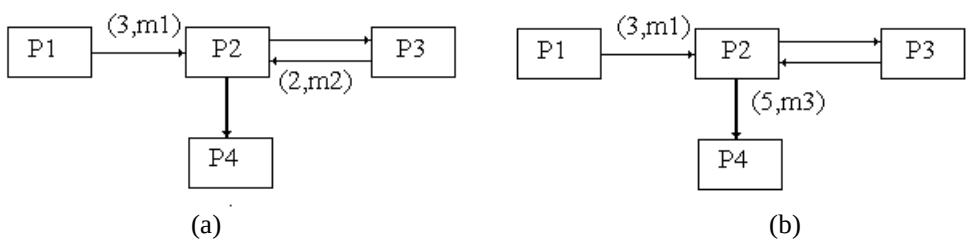

Figura 21. Deadlock en aproximaciones conservadoras

4.4.1. Evitando abrazo mortal

Una forma de resolver el problema de existencia de abrazo mortal trata de evitarlos especificando **estáticamente** enlaces que indican qué procesos pueden comunicarse con otros [Cha79]. Sabemos que para determinar cuándo es seguro procesar un mensaje, la secuencia de timestamps en un enlace debe ser creciente. Por ende, el timestamp del último mensaje en un enlace de entrada es un límite inferior en el timestamp de todo mensaje subsiguiente. La idea es aprovechar esta información y usar mensajes **nulos,** que no corresponden a ninguna actividad en el sistema físico.

Un mensaje nulo de PA a PB es una promesa de que no se enviará un mensaje con timestamp menor que el mensaje nulo. ¿Cómo se determinan los timestamps de los mensajes nulos que se mandan? El reloj de cada enlace de entrada provee un límite inferior en el timestamp del siguiente evento sin procesar. Este límite de entrada puede usarse para determinar un límite inferior en el timestamp del siguiente mensaje de salida. 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 17

Supongamos que estudiamos los tres procesos lógicos de la figura 22a. En este caso, el algoritmo conservador entra en Deadlock. Ahora supongamos que cada proceso lógico progresa en avances de 9 unidades de tiempo. En la figura 22b vemos la emisión de los mensajes nulos para este caso. Luego, el proceso P2 tiene dos enlaces de entrada con tiempo local en 21, por ende, procesa el primer mensaje del enlace elegido y desaparece el abrazo mortal que existía previamente.

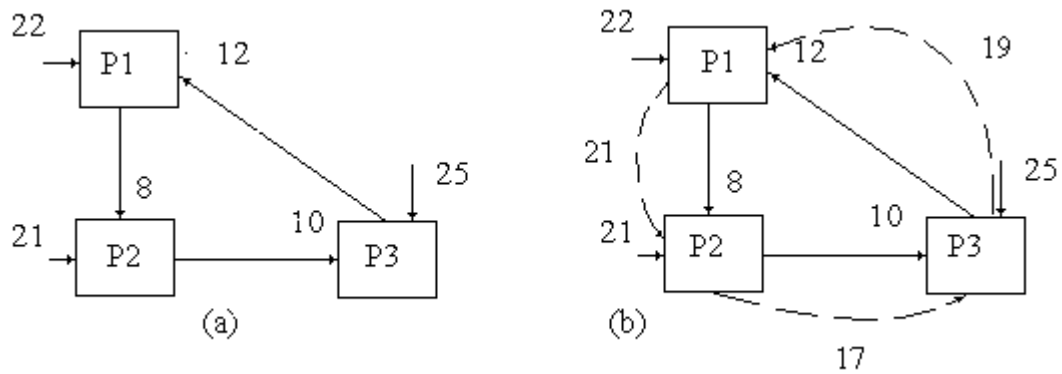

Figura 22 - Evitando Deadlock con mensajes nulos

.

Este mecanismo evita el abrazo mortal mientras que uno no tenga ciclos en los cuales el incremento de timestamp colectivo de un mensaje que atraviesa el ciclo pueda ser cero. Una condición necesaria y suficiente para abrazo mortal usando este esquema es que exista un ciclo de enlaces con el mismo tiempo de enlace (por ejemplo, en la figura 22 si el avance es cero hay abrazo mortal).

Veamos un argumento intuitivo de porqué no hay abrazo mortal. 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, t1 >= t1. Por la presunción de predictibilidad, por lo menos un proceso en el ciclo, digamos Pi, con relojes del enlace de salida estrictamente mayores que su reloj local, por ejemplo, ti>ti+1. Luego, tenemos t1>t1, una contradicción.

Ejemplo 18

Veamos cómo los algoritmos de mensajes nulos evitan abrazo mortal para el ejemplo de la figura 23. El precálculo para los procesadores P2 y P3 se asume como 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.

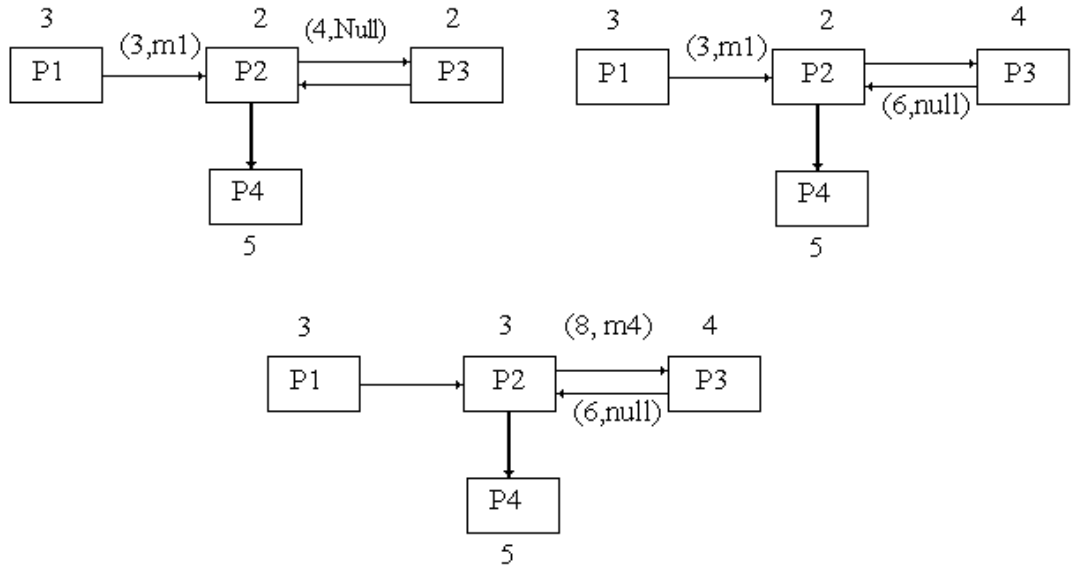

Figura 23 - Uso de mensajes nulos para la figura 21

Una variación en la aproximación de mensaje nulo es mandar mensajes nulos en una base de demanda en lugar de después de cada evento, como hace el protocolo SRADS [Rey82]. Aquí, siempre que un proceso está por bloquearse pide el siguiente mensaje (nulo u otro) al proceso emisor conectado al enlace de entrada. Esta aproximación ayuda a reducir la cantidad de tráfico de mensajes nulos, aunque puede ser necesaria mayor demora para recibir mensajes nulos debido a que son necesarios dos mensajes. SRADS trabaja bien con sistemas sincrónicos con interacciones infrecuentes, ya que a medida que la interacción aumenta, el overhead degrada el desempeño.

Una mejora al esquema de mensajes nulos usa un timeout para transmisión. Si la transmisión de un mensaje nulo se demora un tiempo, podría no necesitar ser transmitido porque será generado un mensaje con mayor timestamp. Esto disminuirá el total de mensajes nulos necesarios, aunque algunos procesos pueden ser demorados al esperar información de timing. Los mensajes nulos posteriores en un enlace podrían eliminar mensajes nulos no procesados previamente en el enlace, ya que son obsoletos. Otra mejora posible del algoritmo es que un proceso mande mensajes nulos en sus enlaces de salida sólo cuando se bloquea.

4.4.2. Detección y recuperación de abrazo mortal

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 el menor timestamp siempre son seguros de procesar. Cuando un proceso se bloquea manda un mensaje (llamado "probe") con timestamp con su reloj local a alguno de sus predecesores para obtener información de sus relojes. El proceso receptor enviará su valor del reloj local al proceso que lo pide si este es posterior al reloj local del proceso que lo pide. Sino, envía probes a sus predecesores. La mayoría de las soluciones precisan que el mensaje contenga el camino que ha atravesado y posiblemente los relojes locales de los procesos en el camino para detectar y corregir un abrazo mortal. Luego, los mensajes crecen en longitud mientras pasan.

Esta aproximación puede modificar para detectar y recuperar abrazos mortales locales (por ejemplo, situaciones donde sólo una porción de la red está en abrazo mortal). Puede hacerse preprocesamiento para identificar todas las subredes que podrían tener abrazo mortal, y aplicar estas técnicas en subredes individuales (el overhead puede ser grande si la topología de la red contiene muchos ciclos).

4.4.3. Precálculo de tiempos de servicio

Llamamos **precálculo** (lookahead) a la habilidad de predecir qué pasará (o, más importante, qué NO ocurrirá) en el futuro simulado. Si un proceso en tiempo simulado Clock puede predecir todos los eventos que generará hasta el tiempo Clock+L, se dice que el proceso tiene precálculo L. El precálculo mejora la habilidad de predecir eventos futuros, que pueden usarse para determinar qué otros eventos son seguros de procesar. Se usa en la aproximación de evitar abrazo mortal para determinar los timestamps que se asignan a mensajes nulos. También se usa para detección y recuperación de abrazo mortal porque siempre que un proceso manda un mensaje con incremento de timestamp de T a otro proceso, está garantizando que ningún otro mensaje seguirá en el enlace que contiene un timestamp menor que Clock + T.

Se dice que hay "predictibilidad" cuando existe un límite inferior de precálculo para cada ciclo. Intuitivamente, los procesos mantienen los relojes de sus enlaces de salida adelante de sus relojes locales mandando mensajes nulos. En el momento de inicialización, todos los relojes se ponen en cero. El reloj local del proceso se pone en el mínimo de todos los relojes de sus enlaces de entrada, y se actualizan los relojes de los enlaces de salida menores al reloj local (ya sea enviando mensajes reales o nulos). Cuando un proceso actualiza los relojes de enlaces de salida, los mueve tan adelante como sea posible, pero no actualiza los relojes de salida que ya están adelante en su reloj local. Luego procesa todo mensaje de entrada con timestamp igual a su tiempo local.

Se puede mejorar la habilidad del precálculo de los procesos haciéndolo por porciones de cálculo de eventos futuros. Por ejemplo, al simular una red de colas FIFO sin remoción, se puede precalcular el tiempo de servicio de los trabajos que aún no han sido recibidos. Si el servidor está libre y su reloj tiene un valor de 100, y el tiempo de servicio del próximo trabajo ha sido precalculado como de 50, el límite inferior en el timestamp del próximo mensaje que mandará es 150 en lugar de 100. El precálculo es posible si se pueden predecir aspectos de los eventos futuros sin conocer mensajes que provocan cálculos. En nuestro ejemplo, si el tiempo de servicio depende de un parámetro en el mensaje el precálculo no será tan simple.

Ejemplo 19

Consideremos la simulación de una red de encolado consistente de dos estaciones conectadas en serie. Cada estación tiene un servidor y una cola con trabajos (clientes) esperando recibir servicio (los trabajos se atienden en orden FIFO). En general, se usan dos tipos de eventos: (1) uno de llegada de un trabajo a una estación; (2) uno de salida de trabajo terminado. Un trabajo J que llega a la primera estación en el momento T pasará, en general Q >= 0 unidades de tiempo en la cola, esperando ser atendido y una cantidad S de tiempo siendo atendido. Este simulador tiene propiedades de precálculo pobres. P1 debe avanzar su tiempo simulado a T+Q+S antes de generar un nuevo evento de llegada con timestamp T+Q+S para P2. Tiene precálculo cero con respecto a generar nuevos eventos de llegada (Fig. 24).

Una aproximación alternativa elimina el evento de salida. Aquí, procesar un evento de llegada en P1 provoca planificar un nuevo evento de llegada en P2 (porque se usan colas FIFO sin remoción). El evento en el momento T puede predecir el evento de llegada en T+Q+S porque S y Q pueden calcularse en tiempo simulado T. En particular, Q es el tiempo de servicio restante para el trabajo atendido en el momento T, mas los tiempos de servicio de todos los trabajos que lo preceden. De forma similar, puede calcularse S en T porque no depende en el estado del proceso en un instante posterior a T. El precálculo usando esta aproximación alternativa es Q + S [Fuj90].

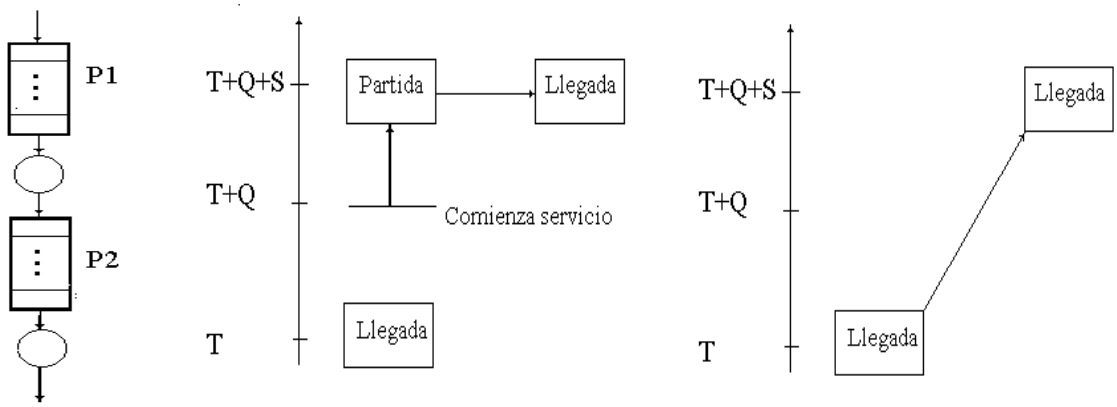

Figura 24 - Precálculo de tiempos de servicio

No siempre se puede tener buen precálculo. Por ejemplo, si la distribución de tiempo de servicio tiene un mínimo de cero y hay remoción, un trabajo de alta prioridad simulado hasta el tiempo T podría afectar (aunque muy raramente) a toda la estación en la red en el momento T, de forma que ninguna estación podría hacer precálculo más allá de T.

4.4.4. Ventanas de tiempo conservadoras

Lubachevsky [Lub89] usa un mecanismo con mensajes nulos y precálculo, pero una ventana móvil de tiempo simulado. Así se trata de reducir el overhead existente para determinar cuándo es seguro procesar un evento. El límite inferior de la ventana se define como el mínimo timestamp de todo evento sin procesar, y sólo los eventos no procesados cuyo timestamp reside en la ventana pueden procesarse.

El propósito de la ventana es reducir el "espacio de búsqueda" que se debe recorrer para determinar si un evento es seguro. Por ejemplo, si la ventana se extiende del tiempo simulado 10 al tiempo simulado 20, y la aplicación es tal que cada evento procesado genera uno nuevo con un incremento de timestamp mínimo de 8 unidades, cada proceso lógico sólo necesita examinar eventos no procesados en los procesos lógicos vecinos para determinar qué eventos son seguros: no hay eventos no procesados a dos o más hops que puedan afectar a alguno en la ventana de 10 a 20 porque tal evento tendría que tener un timestamp anterior al comienzo de la ventana.

Una cuestión importante es el método utilizado para determinar el tamaño de la ventana de tiempo. Si la ventana es muy pequeña, habrá muy pocos eventos disponibles para ejecución concurrente. Por otro lado, si la ventana es muy grande, el mecanismo de simulación se comporta de la misma forma que lo haría si no se usara ninguna ventana de tiempo lo que no justifica el overhead del mecanismo de ventana. Un tamaño adecuado de ventana requiere información específica de la aplicación (que debe obtenerse del programador, del compilador o en tiempo de ejecución).

4.4.5. Desempeño de los mecanismos conservadores

Existen diversos estudios empíricos y analíticos para determinar la desempeño de los diversos esquemas conservadores. Se estudiaron los esquemas de evitar, detectar y recuperar abrazo mortal con resultado diverso. El grado de precálculo juega un rol critico en el desempeño, ya que un proceso con precálculo L puede habilitar a otros procesos para procesar de forma segura mensajes de eventos pendientes.

El programar la simulación explotando el precálculo puede mejorar dramáticamente el desempeño. En la figura 25 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). Hay un número fijo de trabajos (la población de mensajes). Se muestra el número promedio de mensajes procesados entre abrazos mortales, y la aceleración relativa

a una lista de eventos secuencia. Se ve que el uso de precálculo es superior a no hacerlo. Las curvas de aceleración para la aproximación de evitar abrazo mortal usando mensajes nulos fueron similares. Se vio que la aproximación de detectar y recuperar abrazo mortal es generalmente superior, particularmente cuando hay un número pequeño de procesos en el sistema.

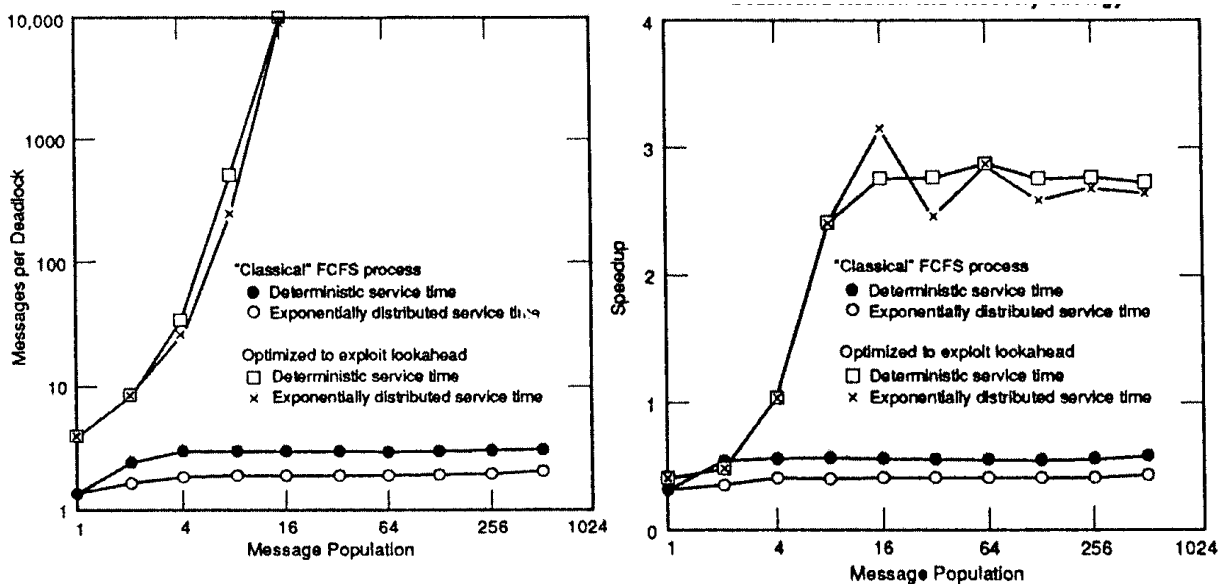

Figura 25 - Desempeño de algoritmos conservadores (a) Eficiencia; (b) Aceleración [Fuj90]

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.

En otro trabajo, Fujimoto usa cargas sintéticas para caracterizar cuantitativamente el precálculo. Para ello usa un parámetro llamado **relación de precálculo**, y presenta datos empíricos para demostrar la importancia de explotar precálculo para lograr buen desempeño. Dependiendo del precálculo, se clasifican las aceleraciones para estas cargas sintéticas desde las más lentas que la ejecución secuencial hasta las casi ideales. Pudo verse que el esquema detección/recuperación y de mensajes nulos puede tener eficiencias tan altas como el 75% con el esquema de mensajes nulos (aceleración relativa a un monoprocesador). El precálculo y la población de mensajes tiene influencia (hay mayor aceleración cuanto mayor precálculo y población de mensajes). Para el esquema de detección/recuperación de abrazo mortal hay un nivel de población crítico: poblaciones inferiores producen desempeño pobre y constante. A medida que la población aumenta por encima de ese nivel, el desempeño mejora dramáticamente. La población necesaria para lograr buena desempeño depende del precálculo (mayor precálculo reduce la población necesaria) [Fuj89].

Quizá la desventaja más obvia de las aproximaciones conservadoras es que no pueden explotar por completo el paralelismo disponible en la aplicación de simulación. Si un evento puede afectar a otro, deben ejecutarse secuencialmente. En general, se fuerza la ejecución secuencial sin ser necesaria. Por ende, los algoritmos dependen del precálculo. Otro problema está relacionado con la robustez: cambios menores en la aplicación pueden tener un efecto catastrófico en el desempeño. Esto es problemático porque los investigadores no suelen tener conocimiento avanzado de todo el rango de experimentos. Otro problema es que las configuraciones son estáticas: no se pueden crear dinámicamente nuevos procesos, y su interconexión también debe definirse estáticamente.

La mayoría de los esquemas conservadores requieren que el programador provea conocimiento con respecto al comportamiento del proceso lógico. Una estimación muy conservadora puede resultar en desempeño muy pobre, y una muy optimista pueden provocar violaciones de restricciones de causalidad.

Tal vez la desventaja más seria con los protocolos conservadores existentes es que el programador debe cuidar los detalles del mecanismo de sincronización para lograr buena desempeño. Los que proponen aproximaciones optimistas dicen que el usuario no debe preocuparse de tal complejidad.

4.5. Mecanismos optimistas

Los métodos optimistas no evitan errores de causalidad sino que los detectan y recuperan. En contraste con los mecanismos conservadores, no precisan determinar cuándo es seguro proceder; sino que detectan un error e invocan un procedimiento de recuperación, explotando paralelismo donde pueden ocurrir errores de causalidad pero de hecho no ocurren.

El mecanismo Time Warp es el protocolo optimista más conocido [Jef85]. Cuando el mensaje de un evento recibido en un proceso lógico tiene timestamp menor al del reloj del proceso, hay un error de causalidad. El evento que provoca el rollback se llama **dispersor**. La recuperación se realiza deshaciendo los efectos de todos los eventos que han sido procesados prematuramente por los procesos que reciben el dispersor (es decir, los eventos procesados con timestamp mayor que el del dispersor).

Los mensajes tienen dos timestamps: tiempo de emisión y de recepción. El reloj local (LVT - Local Virtual Time) de un proceso se pone en el mínimo tiempo de recepción de todos los mensajes no procesados. Los procesos pueden ejecutar eventos mientras tengan alguna entrada, por ende el LVT de un proceso puede adelantarse a los de sus predecesores, y puede 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, puede 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 anular un mensaje transmitido 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 procesado, debe deshacerse el proceso, y eliminar el mensaje positivo. Repitiendo este procedimiento recursivamente se puede cancelar todos los cálculos erróneos.

Para cancelar los efectos de 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.

Para implementar Time Warp cada proceso debe mantener su LVT, su estado actual, una cola de estados con copias de sus estados previos (con por lo menos un estado previo al GVT); una cola de entrada con todos los mensajes recibidos con tiempos de emisión mayores o iguales al GVT (en orden de recepción); y una cola de salida con copia de los mensajes enviados con tiempos de emisión mayores o iguales al GVT (en orden de emisión). Los mensajes de entrada tienen signo positivo; las copias en la cola de salida son antimensajes y tienen signo negativo. Si un mensaje y su antimensaje están en la misma cola, se destruyen.

Al recibir un dispersor (tiempo de recepción menor que el LVT), el proceso hace rollback y lo procesa. El proceso pone el estado actual en el último estado grabado antes del tiempo de recepción del dispersor, y pone su LVT en el tiempo de ese estado. Descarta todos los estados posteriores de su cola de estados, manda todos los antimensajes de su cola de salida con tiempo de emisión mayores que su LVT. Procesa el dispersor y procesa los mensajes en su cola de entrada con tiempos de recepción mayores o iguales al LVT, y avanza en el tiempo de simulación.

Ejemplo 20

Veamos el caso de la figura 26a. Aquí 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.

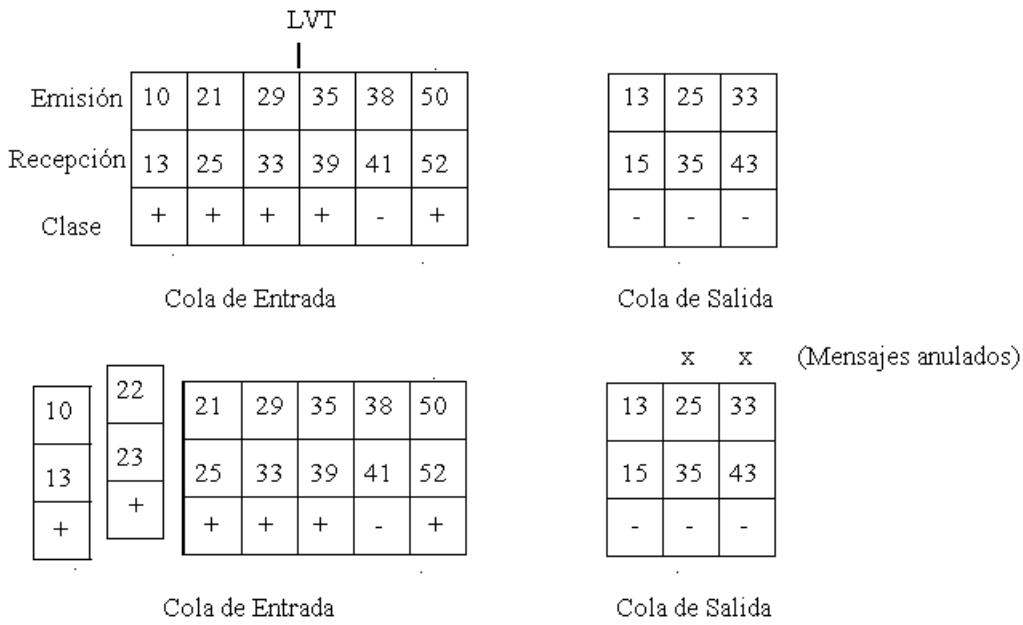

Figura 26 - Llegada de un antimensaje en Time Warp.

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. Una ventaja es que no necesita conocer la topología de las posibles interacciones entre procesos. 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.**

Resumiendo, el algoritmo sigue estos pasos:

. Si un antimensaje llega antes del mensaje correspondiente, se almacena en la cola de entrada (sin importar si su tiempo está antes o después del LVT). Cuando el mensaje positivo llega, ambos se destruyen.

. Si un mensaje positivo llega antes de su antimensaje, y al llegar el antimensaje el LVT es menor que el tiempo de recepción (los mensajes están en futuro simulado), los dos se destruyen.

. Si un mensaje llega antes de su antimensaje y cuando llega el antimensaje el LVT es mayor que su tiempo de recepción, el proceso hace rollback. Se pone el mensaje actual en el último estado grabado con tiempo simulado anterior al tiempo de recepción del mensaje, mensaje y antimensaje se destruyen, y el proceso avanza.

Una desventaja de Time Warp es que requiere mucha memoria. Si se calcula el GVT seguido, se pueden sacar mensajes de la entrada y salida. El GVT puede estimarse con un algoritmo distribuido lineal en el tiempo de hacer broadcast a todos los procesadores en el sistema. Si el GVT se estima frecuentemente, es necesaria menos memoria pero habrá mayor overhead en tiempo. Otra forma de

ahorrar memoria es descartar los estados con timestamps mayores que el LVT que surgen a partir de los rollbacks.

4.5.1. Cancelación Perezosa

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 a un tiempo T, 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.

Dependiendo de la aplicación, la cancelación perezosa puede mejorar o degradar el desempeño, ya que requiere overhead adicional para determinar si ya existe un antimensaje. También permite que los cálculos erróneos se dispersen más que con cancelación agresiva, por ende el desempeño puede degradarse si el simulador fuerza la ejecución de cálculos incorrectos. Por otro lado, si los mensajes no se reproducen, los rollbacks en procesos sucesores serán necesarios bajo ambos mecanismos, y ocurrirán antes con cancelación agresiva.

En la mayoría de los casos la cancelación perezosa es más rápida (hay evidencia empírica) pero precisa más memoria que la agresiva. La cancelación perezosa también permite que los cálculos se ejecute en menos tiempo que el caso de camino crítico, ya que los cálculos con entrada incorrecta (o parcialmente correcta) pueden generar resultados correctos. Supongamos que un proceso calcula el mínimo de A y B, y ejecuta prematuramente usando un valor incorrecto de A. Si ambos valores (correcto e incorrecto) de A son mayores que B, la ejecución incorrecta produce el mismo resultado que la correcta). Esto no es posible usando cancelación agresiva, porque la hacer un rollback los cálculos se descartan inmediatamente.

Estos factores están relacionados con la propiedad de precálculo. Recordemos que en el ejemplo 19 que el evento de llegada en T+Q+S era invariante a otros eventos en [T, T+Q+S]. Luego, P1 hace precálculo y planifica el evento de llegada T+Q+S aunque su reloj local sólo avance hasta T. Ahora consideremos una simulación Time Warp usando cancelación perezosa que usa los eventos de llegada y partida. Asumamos que P1 ha avanzado más allá de T+Q+S, y ha procesado el evento de partida en T+Q+S, y planificado el evento de llegada subsiguiente. Supongamos que un evento dispersor llega con timestamp Tx tal que $T < Tx < T+Q+S$. Si P1 hace rollback, procesa el dispersor, y reejecuta el evento de partida en T+Q+S. Sin embargo, como el evento de llegada en T+Q+S no está afectado por el evento dispersor (debido a que se usan colas FIFO), será recreado el mismo evento de llegada que fue generado en la ejecución original. Debido a que se usa cancelación perezosa, el ejecutivo Time Warp no cancela el tiempo de llegada original en el tiempo T+Q+S. La propiedad de invarianza sobre la cual se basa el precálculo es la misma propiedad que permite que la cancelación perezosa tenga éxito.

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. La desventaja es que su overhead es mayor, ya que el cálculo invariante debe reejecutarse, y son necesarias comparaciones de mensajes para determinar cuándo es aplicable la optimización.

4.5.2. Ventanas de tiempo optimistas

También se han propuesto ventanas de tiempo 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 [Sok88].

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. Los datos empíricos sugieren que las ventanas de tiempo ofrecen poca ventaja en ciertos casos, y mejoras en otros.

4.5.3. Llamadas de Wolf

En [Mad88], 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 (es decir, un mensaje dispersor).

Como en el esquema de ventana de tiempo, la desventaja es que algunos cálculos correctos pueden congelarse innecesariamente. También, 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 límites en estas cantidades puede ser difícil.

4.5.4. Cancelación directa

En Time Warp es importante que uno pueda cancelar un cálculo incorrecto antes que este pueda dispersarse en el sistema. De otra forma puede ocurrir que los cálculos erróneos se dispersen rápidamente a través del sistema, mientras que hay antimensajes tratando frenéticamente de encontrarlos. Tal comportamiento puede evitarse; una forma de prevenirlo es dar 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.

4.5.5. Desempeño de los mecanismos optimistas

Los mecanismos optimistas han logrado aceleraciones substanciales en simulaciones de campos de batalla, redes de comunicaciones, sistemas biológicos, simulación productor/consumidor, de hardware digital y de otros fenómenos físicos. La aceleración típica en un hipercubo tiene un rango de 10 a 20 usando 32 procesadores. Usando 100 procesadores de una Butterfly BBN se obtuvieron aceleraciones de 37, y de 57 en una Butterfly BBN de 64 procesadores. Pudo obtenerse mejora significativa en el desempeño empleando conocimiento especifico de la aplicación para optimizar la ejecución. Otros experimentos obtuvieron aceleraciones mayores a 10 usando 16 procesadores, y eficiencias tan altas como el 90%. El número de procesos por procesador y el balance de la carga tienen impactos significativos en la desempeño. Con 8 procesos en cada uno de los 127 procesadores y 9 procesos en un procesador adicional, la eficiencia es 72% versus una eficiencia del 91% cuando hay 8 procesos pro procesador para 128 procesadores.

La evidencia indica que mientras que el precálculo mejora el desempeño de los algoritmos optimistas, no es un prerrequisito para Time Warp. En la figura 27 se compara Time Warp contra los algoritmos de evitar, detectar y recuperar abrazo mortal. Se designa alguna fracción de los trabajos en el simulador (aquí un 1%) como alta prioridad, mientras que el resto son de baja prioridad. Los trabajos de alta prioridad remueven el servicio de los de baja prioridad. Como se noto antes, esta simulación contiene características de precálculo muy pobres, y no puede optimizarse usando colas FIFO como se hizo antes

para el simulador. Como puede verse, Time Warp puede obtener una aceleración significativa para este problema, mientras que los algoritmos conservadores tienen cierta dificultad.

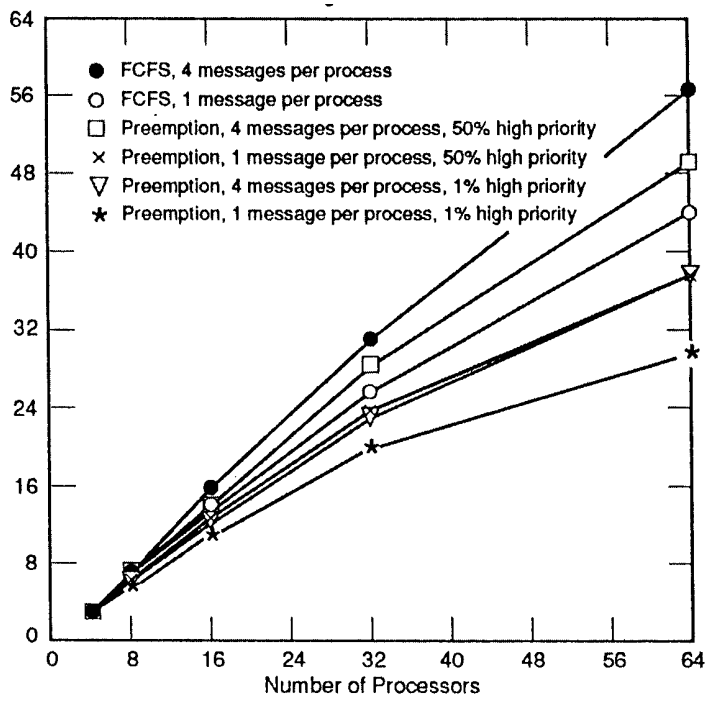

Figura 27 - Aceleración de simulación Time Warp (256 procesos lógicos) [Fuj90]

Las mediciones usaron 256 procesos lógicos. Se muestra el desempeño usando colas FIFO y colas con remoción. 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 test saturadas (mas paralelismo que procesadores) y no saturadas (menos paralelismo que procesadores). Aunque los resultados son alentadores, debemos decir que los overheads de salvado de estado pueden degradar bastante el desempeño (baja cuando el tamaño del estado aumenta).

Si se considera el caso donde el rollback requiere tiempo cero, si los cálculos incorrectos nunca hacen rollback de los correctos. Time Warp usando cancelación agresiva produce desempeño optimo (tiempo de ejecución igual al camino critico de menor limite). Asumiendo un costo fijo para el rollback, se demostró que Time Warp puede mejorar los algoritmos de Chandy-Misra en una cantidad arbitraria (p. ej. 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 [Lip90].

Los modelos analíticos existentes produjeron desempeño desde extremadamente bueno hasta muy pobre, dependiendo del costo de rollback (todo cálculo no presente durante los cálculos normales). El rollback produce tres overheads: restaurar las colas de entrada, de estado, y de salida. Restaurar las colas de entrada y estado tiene costo despreciable porque solo hacen falta unas pocas instrucciones de maquina para poner el puntero en el próximo evento a ser procesado (otros costos tales como insertar el dispersor en la cola de entrada no se consideran costo de rollback porque puede tener lugar durante el progreso normal). El overhead principal en un rollback está en restaurar la cola de salida, lo que implica mandar antimensajes.

El costo de rollback no es proporcional a su longitud. Por un 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 (overhead de

planificación, procesar mensajes de entrada, salvado de estado, y por supuesto, y cálculo de la simulación).

Ejemplo 21

Supongamos que hacer rollback de un cálculo T unidades de tiempo simulado, toma el doble que el progreso hacia adelante. Consideremos el caso de dos procesadores, P1 y P2: supongamos que P2 esta 10 unidades adelante de P1 cuando P1 manda un mensaje a P2, provocando que P2 haga rollback. Asumamos que la transmisión del mensaje toma tiempo cero. Mientras, P2 está haciendo rollback de 10 unidades de tiempo, de acuerdo con nuestra presunción que hacer rollback es mas lenta que el progreso. Ambos procesadores avanzan unas pocas unidades de tiempo simulado, y P2 envía un mensaje que hace rollback de P1. Mientras P1 está haciendo rollback 20 unidades, P2 avanza 40 unidades. Es claro que si estos procesadores siguen haciendo rollback del otro, la distancia de rollback y el tiempo de hacerlo aumenta exponencialmente con la simulación. Luego, la tasa de progreso de los procesadores.

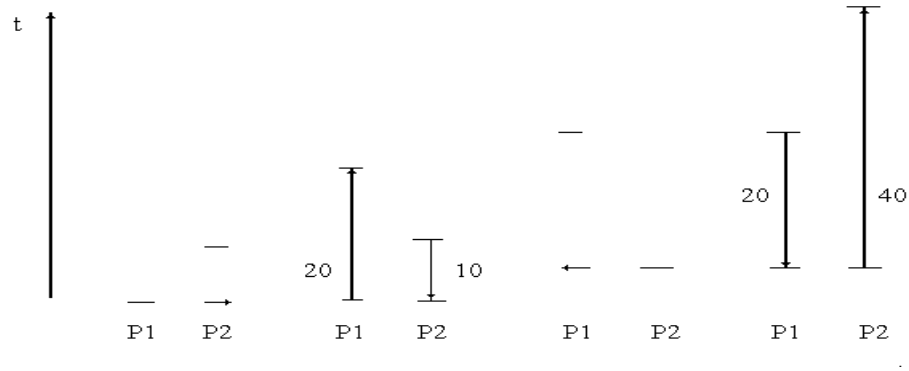

Figura 28 - Comportamiento de procesos con Time Warp

Una cuestión crítica enfrentada por los sistemas optimistas tales como Time Warp es si el sistema exhibirá comportamiento de thrashing donde la mayoría de su tiempo se pasa ejecutando cálculos incorrectos y haciendo rollback. Los cálculos incorrectos serán ejecutados al cuesta de los correctos; mas aun, si la aplicación solo contiene paralelismo limitado relativo al numero de procesadores disponibles, es inevitable un grado significativo de rollback.

Los algoritmos optimistas 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. Mas aún, hacer debugging de las implementaciones de Time Warp consume tiempo porque puede requerir análisis detallado de scenarios de rollback complejos. Por otro lado, este costo de desarrollo sólo debe pagarse una vez cuando se desarrolla el kernel de Time Warp.

5. Conclusiones

En este trabajo hemos presentado diversos mecanismos utilizados en la actualidad para permitir simulación eficiente de sistemas de eventos discretos. Esta actividad requiere de formalismos especiales que permitan su construcción de forma eficiente, segura y a bajo costo.

El formalismo DEVS es uno de estos formalismos, que está cada vez más difundido en actividades de simulación de eventos discretos. Su base formal permite utilizarlo como herramienta matemática, permitiendo estudiar detalladamente el comportamiento de los sistemas desarrollados. Por otro lado, su estructura jerárquica y modular facilita el desarrollo y mantenimiento de las aplicaciones.

Los autómatas celulares, por otro lado, permiten simular determinada clases de sistemas complejos (desde flujo de tráfico hasta sistemas ecológicos complejos) por medio de un mecanismo simple.

En todos los casos es necesario proveer métodos eficientes para la ejecución de las simulaciones. Para ello hemos analizado diversos mecanismos existentes en la actualidad, básicamente aquellos que son asincrónicos y permiten la división de la simulación a realizar (y su lista de eventos), ya que son los que parecen ofrecer mayor grado de aceleración sobre las simulaciones secuenciales. Se puso especial énfasis en distintos métodos, clasificados en optimistas y pesimistas. Los mecanismos pesimistas fueron una primer forma de atacar estos problemas, aunque en la actualidad las aproximaciones optimistas parecen ser las más adecuadas para resolver la mayoría de los problemas.

6. Referencias

[Aya92] AYANI, R.; RAJAEI, H. "Parallel simulation using conservative time windows", Proc. of the Winter Simulation Conference. Dec. 1992. pp. 709-717.

[Cha79] CHANDY, K; MISRA, J. "Distributed simulation: a case study in design and verification of distributed programs", IEEE TOSE, September 1979.

[Cha81] CHANDY, K; MISRA, J. "Asynchronous distributed simulation via a sequence of parallel computations". CACM, (24):4, April 1981. pp 198-206.

[Cho94] CHOW, A.; ZEIGLER, B. "Revised DEVS: a parallel, hierarchical, modular modeling formalism". Proc. Winter Simulation Conf., 1994.

[Cho94b] CHOW, A.; ZEIGLER, B. "Abstract simulator for the parallel DEVS formalism".

[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.

[Cho95b] CHOW, A. "A C++ binding of the parallel devs formalism". Proceedings of the SCS'95. pp. 38. 1995.

[Con88] CONCEPCION, A.; ZEIGLER, B. "DEVS Formalism: a framework for hierarchical model development", IEEE transactions on software engineering. Vol 14, No. 2. Feb 1988. Pp. 228-241.

[Con89] CONCEPCION, A. "A hierarchical computer architecture for distributed simulation", IEEE Transactions con computers, Feb. 1989.

[Eck95] ECKART, J.D. "A Cellular automata 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.

[Fuj89] FUJIMOTO, R. "Performance measuerements of dist5ributed simulation strategies". Transactions of the SCS, 6, 2 (Marzo 1989). pp. 21-26.

[Fuj89b] FUIMOTO, R. "Time Warp on a shared memory multiprocessor". Transactions of the SCS. 6, 3 (Julio 1989). pp. 211-239.

[Fuj90] FUJIMOTO, R. "Parallel Simulation of Discrete Events". Communications of the ACM Vol. 33. No. 10. pp. 30-53. 1990.

[Gaf88] GAFNI, A. "Rollback mechanisms for optimistic distributed simulation systems". Proc. of the SCS M. on DS. 19(3):61-67, July 1988.

[Gar70] GARDNER, M. "The fantastic combinations of John Conway's New Soligaire Game 'Life'.". Scientific American, 23 (4), 1970, pp. 120-123.

[Gar86] GARZIA, R.F.; GARZIA, M.R.; ZEIGLER, B.P. "Discrete Event Simulation", IEEE Spectrum, December 1986. pp. 32-36.

[Gia95] GIAMBIASI, N. "Introduction à la modélisation et à la simulation". Materiales del curso de D.E.A.; Université d'Aix-Marseille III. 1996.

[Gra80] GRAYBEAL, W.; POOCH, U. "Simulation: Principles and Methods". Cambridge, MA, 1980.

[Gut95] GUTOWITZ, H. "Cellular Automata and the sciences of Complexity. Part I-II". To appear in the journal Complexity. November 1995.

[Ho89] HO, Y. "Special issue on discrete event dynamic systems", Proceedings of the IEEE, 77 (1), 1989.

[Hoo89] HOOVER, S.; PERRY, R. "Simulation: A Problem Solving Approach", Addison Wesley, Reading, MA, 1989.

[Jef85] JEFFERSON, D. "Virtual Time". ACM TOPLS, 7(3): 404-425. July 1985.

[Jef87] JEFFERSON, D. "Distributed simulation and the Time Warp Operating System". In 11th. Symposium on OS principles. pp 77-93. November 1987.

[Kim95] KIM, K. et al. "Distributed optimistic simulation of hierchical DEVS models". Proceedings of the SCS'95. pp. 32. 1995.

[Lin95] LIN, Y.; FISHWICK, P. "Asynchronous Parallel Discrete Event Simulation", IEEE Transactions on Systems, Man and Cybernetics. 1995.

[Lip90] LIPTON, R.; MIZELL, D. "Time-Warp vs. Chandy-Misra: a worst-case comparison". Proceedings of the SCS Multiconference on Distributed Simulation. 22, 1 (Enero 1990). pp. 137-143.

[Lub89] LUBACHEVSKY, B. "Efficient distributed event-driven simulations of multiple-loop networks". Communications of the ACM 32. January 1989. pp. 111-123.

[Mad88] MADISETTI, V.; WALRAND, J; MASSERSCHMIT, D. "Wolf: a rollback algorithm for optimistic distributed simulation systems", in Proc. of Winter Simulation Conference, 296-305. 1988.

[Mis86] MISRA, J. "Distributed discrete-event simulations". ACM Computing surveys. Vol. 18, No. 1, 39-65. 1986.

[Nic94] NICOL, D.; FUJIMOTO, R. "Parallel Simulation Today". Annals of Operations reserach, vol. 53, 249-285. Nov 1994.

[Ove92] OVEREINDER, B.; SLOOT, P. "Application of Time-Warp to parallel simulations with asynchronous cellular automata". 1992.

[Ove93] OVEREINDER, B.; SLOOT, P.; HERZBERGER, L. "Time-Warp on a transputer platform: pilot study with asynchronous cellular automata". 1993.

[Pra95] 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.

[Rey82] REYNOLDS, P. Jr. "A shared resource algorithm for distributed simulation". Proceedings of the 9th. Ann. Symp. on Comp. Arch. pp. 259-266. 1982.

[Rig89] RIGHTER, R.; WALRAND, J. "Distributed simulation of discrete event systems". Proceedings of the IEEE. pp. 99. January 1989.

[Sok88] SOKOL, L;BRISCOE, D; WIELAND, A. "MTW: a strategy for scheduling discrete simulation events for concurrent execution", Proc. of SCS conf on distributed simulation, 34-42. 1988.

[Tof94] TOFFOLI, T. "Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata)". Proceedings of ACRI'94. (1994).

[Wol84] WOLFRAM, S. "Universality and complexity in cellular automata", Physica, 10D, 1-35 (1984).

[Zei76] ZEIGLER, B. "Theory of modeling and simulation". Wiley, 1976.

[Zei84] ZEIGLER, B. "Multifaceted Modelling and discrete event simulation". Academic Press, 1984.

[Zei89] ZEIGLER, B. "DEVS representation of dynamical systems: event-based intelligent control". Proceedings of IEEE, Vol. 77 No. 1, pp. 27-80. 1989.

[Zei90] ZEIGLER, B. "Object-oriented simulation with hierarchical modular models". Academic Press, 1990.

[Zei95a] ZEIGLER, B.; KIM, D. "Design of high level modelling / high desempeño 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, Jan. 1996.