Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Implementación de Modelos Cell–DEVS n–dimensionales

Tutor:

Dr. Gabriel A. Wainer

Autor:

Daniel A. Rodríguez

1999

Implementación de modelos Cell–DEVS n–dimensionales

Abstract I

Informe Científico I

I

Informe Técnico I

I

I

Manual del Usuario I

V

Referencias V

Bibliografía Adicional V

I

Artículos Publicados V

I

I

Abstract

El presente trabajo describe las extensiones realizadas a una herramienta utilizada para el estudio, modelado y simulación de modelos celulares n-dimensionales, respetando los paradigmas *DEVS* y *Cell-DEVS*. La misma posibilita, mediante el uso de un lenguaje de especificación de alto nivel, la construcción, en forma sencilla, de sistemas complejos en función de componentes básicos. Esta construcción jerárquica permite la creación de sistemas de fácil entendimiento y mayor confiabilidad debido a que los componentes básicos pueden ser verificados previamente. Los modelos celulares pueden ser incorporados a la simulación como otro tipo de modelo. Además, permite lograr una disminución en los tiempos de desarrollo, chequeo y mantenimiento de los componentes, posibilitando incluso la reusabilidad de los mismos.
La versión original de la herramienta soporta modelos celulares bidimensionales, donde el estado de las celdas pertenece al conjunto { 0, 1, **?** }, y **?** representa al valor indefinido. Las extensiones realizadas están dirigidas hacia este tipo de modelos, y posibilitan la creación de autómatas celulares n–dimensionales, donde las celdas pueden tener valores pertenecientes al conjunto $\Re \cup \{ ? \}$. Esto implicó la expansión del lenguaje utilizado para la definición del comportamiento de las celdas para dar soporte a las nuevas características de la herramienta. Por otra parte, se extendió el manejo de mensajes a través de los puertos de entrada/salida que conectan a modelos celulares con otros modelos DEVS.

Implementación de modelos Cell–DEVS n–dimensionales

Informe Científico

Tabla de Contenidos

Tabla de Contenidos....2
Tabla de Figuras....3
1 Resumen....5
2 Introducción a la Simulación....5
3 Formalismo DEVS....6
3.1 Modelos Atómicos....7
3.2 Modelos Acoplados....7
3.3 Mecanismo de Simulación....8
4 Autómatas Celulares....8
5 Formalismo Cell–DEVS....10
6 Modelos Cell–DEVS Achatados....13
7 CD++....14
8 N–CD++....16
8.1 Extensión del Lenguaje de Especificación de Modelos....17
8.1.1 Incorporación de Funciones Determinísticas....18
8.1.2 Incorporación de Funciones Probabilísticas....19
8.1.3 Optimización en la Evaluación de Reglas....19
8.1.4 Modo de Debug para Validación de Reglas....20
8.2 Redefinición del Comportamiento ante el arribo de un Mensaje....24
8.3 Redefinición del Comportamiento de Salida....26
8.4 Soporte para Modelos n–Dimensionales....27
9 Análisis de Resultados....28
9.1 Juego de la Vida....28
9.2 Juego de la Vida en 3D....33
9.3 Pinball....36
9.4 Difusión del Calor sobre una Superficie....37
9.5 Difusión del Calor en 3D....38
9.6 Juego de la Vida con uso extensivo de puertos de salida....39
9.7 Inversión de Bits....40
9.8 Uso del Quantum para distintos modelos....41
9.9 Mosquitos....44
10 Conclusiones....44
11 Trabajo Futuro....44
Apéndice. Lenguaje de Especificación de N–CD++....45

Tabla de Figuras

Figura 1 – Relaciones de modelado y simulación [Zei90]....5
Figura 2 – Fragmento de un Autómata Celular Bidimensional....9
Figura 3 – Modelo Acoplado Celular Jerárquico vs Achatado [Wai98]....14
Figura 4 – (a) Subclases de Processor. (b) Subclases de Model....14
Figura 5 – Subclases de Message....16
Figura 6 – Reglas para el Juego de la Vida....17
Figura 7 – Comportamiento de los operadores para la lógica trivalente....18
Figura 8 – Comportamiento de los operadores relacionales....18
Figura 9 – Optimización en la evaluación de las operaciones binarias de la lógica trivalente utilizada en N-CD++....20
Figura 10 – Implementación en N-CD++ de una variante del Juego de la Vida (no incluye reglas)....21
Figura 11 – Reglas para la variante del Juego de la Vida cuya evaluación puede resultar siempre falsa....21
Figura 12 – Error generado por la herramienta al no encontrar ninguna regla válida....22
Figura 13 – Reglas para la variante del Juego de la Vida cuya evaluación puede no resultar verdadera para todas las celdas, pero existe alguna que sea indefinida....22
Figura 14 – Alerta generada por la herramienta cuando no existen reglas válidas pero existe alguna regla indefinida....22
Figura 15 – Reglas para la variante del Juego de la Vida para la cual puede existir más de una regla que sea evaluada a Verdadero....23
Figura 16 – Error generado por la herramienta cuando existen dos o más reglas válidas y se encuentra activo el modo de debug....23
Figura 17 – Reglas para la variante del Juego de la Vida para la cual dos reglas pueden evaluar a Verdadero en un modelo Estocástico....23
Figura 18 – Alerta generada por la herramienta cuando existen dos o más reglas válidas en un modelo estocástico y se encuentra activo el modo de debug....24
Figura 19 – Reglas para la variante del Juego de la Vida para la cual ninguna regla podría resultar verdadera en un modelo Estocástico....24
Figura 20 – Alerta generada por la herramienta cuando no existen reglas válidas en un modelo estocástico....24
Figura 21 – Estructura de una Celda Atómica en CD++....24
Figura 22 – Acoplamiento de una Celda en CD++....25
Figura 23 – Extensión realizada sobre un Celda Atómica en N–CD++ para los Puertos de Entrada....25
Figura 24 – Definición del comportamiento de la celda ante la llegada de un mensaje por un puerto In....26
Figura 25 – Estructura de una Celda Atómica en N–CD++....26
Figura 26 – Definición de la zona { (2,2)..(4,6) }....28
Figura 27 – Reglas utilizadas para el estudio del Juego de la Vida....28
Figura 28 – Promedio de los tiempos de ejecución para el Juego de la Vida de 40x40 celdas....29
Figura 29 – Promedio de los tiempos de ejecución para la simulación de 10 transiciones del Juego de la Vida....30
Figura 30 – Tiempos de Inicialización para el Juego de la Vida....30
Figura 31 – Relación entre el tiempo de inicialización frente a 5 minutos de simulación para el Juego de la Vida. (a) Modelos Jerárquicos; (b) Modelos Achatados....31
Figura 32 – Relación entre el tiempo de inicialización frente a 1 hora de simulación para el Juego de la Vida. (a) Modelos Jerárquicos; (b) Modelos Achatados....31
Figura 33 – Porcentaje del tiempo de inicialización a medida que aumenta el tiempo simulado para el Juego de la Vida con 40x40 celdas. (a) Modelos Jerárquicos; (b) Modelos Achatados....32
Figura 34 – Comparación entre CD++ y N-CD++ al evaluar el Juego de la Vida de 30x30 celdas....33
Figura 35 – Reglas utilizadas para el estudio del Juego de la Vida en 3D....33
Figura 36 – Formato del Vecindario para el modelo del Juego de la Vida en 3D....33
Figura 37 – Promedio de los tiempos de ejecución para el Juego de la Vida en 3D con 10x10x16 celdas....34
Figura 38 – Tiempos de Inicialización para el Juego de la Vida en 3D....34
Figura 39 – Relación entre el tiempo de inicialización frente a 30 segundos de simulación para el Juego de la Vida en 3D. (a) Modelos Jerárquicos; (b) Modelos Achatados....35
Figura 40 – Promedio de los tiempos de ejecución para el Juego de la Vida en 3D, en 30 segs de tiempo simulado....36
Figura 41 – Promedio de los tiempos de ejecución para el modelo del Pinball....37
Figura 42 – Promedio de los tiempos de ejecución para el modelo de Difusión del Calor con 10x10 celdas....38
Figura 43 – Promedio de los tiempos de ejecución para el modelo de Difusión del Calor con 40x40 celdas....38

Figura 44 – Promedio de los tiempos de ejecución para el modelo de Difusión del Calor en 3D con 10x10x3 celdas.....39
Figura 45 – Promedio de los tiempos de ejecución para el Juego de la Vida con puertos de salida..................................40
Figura 46 – Promedio de los tiempos de ejecución para el modelo de Inversión de Bits..40
Figura 47 – Número de mensajes involucrados en la ejecución de los ejemplos [WZ99]...42
Figura 48 – Error de las celdas al usar quantum [WZ99]. Ejemplo b: (a) Error Absoluto, (b) Error Relativo (promedio); Ejemplo c: (c) absoluto, (d) relativo; Ejemplo d: (e) absoluto, (f) relativo...43
Figura 49 - Gramática usada para la definición de reglas bajo N-CD++..47

1 Resumen

El presente trabajo explora la definición y características del formalismo DEVS y su extensión a modelos celulares n–dimensionales (Cell-DEVS) con dominio en los reales, para luego mostrar las extensiones realizadas a una herramienta que permite la simulación de sistemas que respeten esta especificación.

El trabajo está organizado de la siguiente manera. El informe científico comienza con una introducción a la modelización y simulación de sistemas para luego introducir los formalismos DEVS, los Autómatas Celulares y Cell–DEVS. Posterior a esto se verá un análisis global de la implementación de la herramienta y las extensiones realizadas sobre la misma, junto con el lenguaje de especificación usado para la definición del comportamiento de las celdas del modelo celular. En las secciones subsiguientes, se muestran los resultados del análisis de la herramienta para la resolución de diversos ejemplos. En el informe técnico se verá en detalle el análisis, diseño e implementación de la herramienta ahondando en sus distintos aspectos y se muestran en detalle los ejemplos escritos para la misma.

2 Introducción a la Simulación

Un **sistema** es una entidad real o artificial que representa una parte de una realidad y está restringida por un entorno. Puede decirse que un sistema es un conjunto ordenado de objetos lógicamente relacionados que atraviesan actividades, interactuando para cumplir los objetivos propuestos [Wai98].

Un **modelo** es una representación inteligible (abstracta y consistente) de un sistema. En muchos casos no es posible la experimentación directamente sobre el sistema a estudiar, o se desea evitar costos, peligro, etc; por lo que se recurre al uso de modelos. Los *modelos analíticos* están basados en el razonamiento deductivo y permiten obtener soluciones generales al problema Un formalismo analítico muy difundido para el modelado de problemas es el uso de ecuaciones diferenciales. Sin embargo, los sistemas del mundo real generalmente son muy complejos, para los cuales se hace muy difícil, y en ciertos casos imposible, hallar soluciones analíticas o heurísticas con resultados satisfactorios. Para el estudio de este tipo de sistemas se ha difundido el uso de metodologías y herramientas de simulación. La **simulación** es el proceso de diseñar un modelo de un sistema real, y realizar experimentos para describir, explicar y predecir el comportamiento del mismo.

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo del sistema, las decisiones a tomar pueden verificarse artificialmente, un mismo modelo puede usarse muchas veces, la experimentación puede ser controlada, permite compresión de tiempo (dado que una simulación puede ser ejecutada en mucho menos tiempo que el sistema real modelado), etc.

Algunos problemas que existen en el uso de simulación son: su tiempo de desarrollo, los resultados pueden tener divergencia con la realidad (dado que precisan validación), y para reproducir el comportamiento del sistema simulado se requiere una colección extensiva de datos. Debido a esto, si el problema es simple, es conveniente el uso de métodos analíticos ya que permiten obtener soluciones generales. En cambio, para modelos complejos, usando simulación se pueden probar distintas condiciones de entrada para los cuales no serían posibles de probar analíticamente, y obtener resultados de salida significativos.

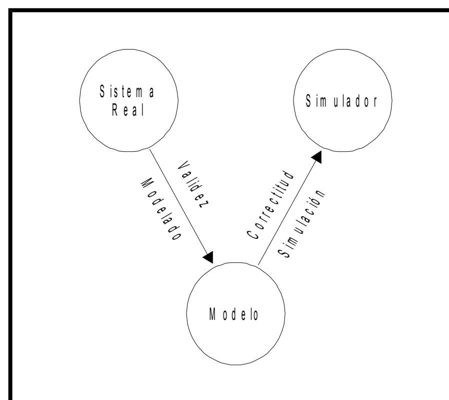


Figura 1 – Relaciones de modelado y simulación [Zei90]

La simulación de un sistema requiere generalmente de los siguientes pasos:

1. Identificación y planteo del problema en función de objetos y actividades.
2. Análisis de los datos de entrada del sistema real y obtención de los mismos.
3. Construcción de un modelo del sistema con los aspectos que se quieren simular.
4. Implementación del modelo para poder ser ejecutado en una computadora.
5. Verificación y validación del modelo.
6. Ejecución de la simulación.
7. Análisis de los datos de salida de la simulación.

Se han desarrollado gran variedad de paradigmas de modelado, que pueden clasificarse de acuerdo a distintos criterios: con respecto a la *base de tiempo*, hay paradigmas a **tiempo continuo**, donde el tiempo evoluciona de forma continua, y a **tiempo discreto**, donde el tiempo avanza por saltos de un valor entero a otro. 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].

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

Por otro lado, un modelo se dice que es **determinístico** si todas las variables tienen certeza completa y están determinadas por sus estados iniciales y entradas. El modelo se dice **probabilístico** si los cambios de estado se producen por medio de leyes aleatorias. Por ejemplo: los datos de entrada del modelo son aleatorios, siendo el modelo determinista, ó el tiempo de llegada de los eventos es aleatorio [Zei96]. Si usa variables aleatorias se dice que el modelo es **estocástico**.

Según el *entorno*, los modelos son **autónomos** (no existen entradas) o no autónomos. Los autónomos evolucionan únicamente en función de tiempo.

3 Formalismo *DEVS*

Actualmente, para gran variedad de aplicaciones complejas se usan modelos y simulación. En algunos de estos sistemas las variables son discretas a tiempo continuo. Tales sistemas 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 [Wai98]. Un paradigma de simulación para DEDS asume que el sistema simulado sólo cambia de estado en puntos discretos del tiempo, ante la ocurrencia de un evento. Un **evento** es un cambio de estado que debe efectuarse a una hora ti $\in \boldsymbol{R}$ (el tiempo es continuo).

Zeigler [Zei76] propuso un formalismo conocido como **DEVS** (*Discrete EVents Systems specifications*) que permite la construcción jerárquica de modelos de eventos discretos con tiempo continuo. Esto favorece la creación de los mismos, ya que pueden ser considerados como nuevos componentes para ser usados en la creación de otros modelos, reduciendo así los tiempos de desarrollo y prueba. Además de un medio para construir modelos simulables, provee una representación formal para manipular matemáticamente sistemas de eventos discretos. El formalismo define cómo generar nuevos valores para las variables y los momentos en los que estos valores deben cambiar. Esta aproximación se integró posteriormente con nociones de programación orientada a objetos [Zei84, Zei90, Zei97].

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 en intervalos, y cuyas transiciones se identifican como eventos discretos. Los intervalos de tiempo entre ocurrencias son variables [Wai98].

Un modelo DEVS se construye sobre la base de un conjunto de modelos básicos llamados **Atómicos**, que se combinan para formar modelos Acoplados, y un conjunto de relaciones que indican como los modelos están conectados en forma jerárquica.

3.1 Modelos Atómicos

Un modelo atómico DEVS puede describirse formalmente como:

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

donde:

X = conjunto de eventos externos de entrada;

Y = conjunto de eventos externos de salida;

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

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

S = conjunto de variables de estados y parámetros. En general se usan las variables *phase* y *sigma* para representar el estado del modelo y el tiempo restante para el próximo cambio de estado;

δ_{INT}: $S \rightarrow S$, es la función de transición interna, que especifica el cambio de estado por causa de eventos internos;

δ_{EXT}: $QxX \rightarrow S$, es la función de transición externa, que especifica los cambios de estado por causa de eventos externos, y posiblemente, una replanificación en su próxima transición interna. Q es el conjunto de estados totales del sistema especificado como $Q = \{ (s,e)\ /\ s \in S,\ e \in [0, D(s)] \}$, donde **e** representa el tiempo transcurrido desde la última transición de estado con estado **s**;

λ: $S \rightarrow Y$, es la función de salida, que genera los resultados en los puertos con anterioridad a la ejecución de la función de transición interna;

D: $S \rightarrow \boldsymbol{R}_0^+ \cup \infty$, función de avance de tiempo, que controla la frecuencia de las transiciones internas. D(s) es el tiempo que el modelo se queda en el estado s si no hay un evento externo.

Los modelos poseen puertos de entrada y salida a través de los cuales interactúan con el entorno. Los eventos determinan los valores que aparecen en esos puertos. Los datos externos son recibidos en un puerto de entrada y la especificación del modelo debe determinar cómo responder a dichos eventos. Los eventos internos que se producen dentro del modelo, modifican su estado y producen eventos destinados a los puertos de salida. Las influencias de los puertos de salida determinarán si estos datos serán enviados a otros componentes como eventos externos.

3.2 Modelos Acoplados

Un modelo atómico puede integrarse con otros modelos DEVS para formar un modelo acoplado. Incluso, varios modelos acoplados pueden formar un nuevo modelo acoplado. Estos se definen como:

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

donde:

X = conjunto de eventos externos de entrada;

Y = conjunto de eventos externos de salida;

I = $< P^X, P^Y>$, representa la interfaz del modelo. En este caso, $\forall\ i \in \{X, Y\}$, P^i es una definición de un puerto (de entrada o salida respectivamente), donde:

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

D = conjunto de nombres de los componentes que lo conforman;

M_i = modelo básico correspondiente al componente i, definido como:

$$M = < X_i, Y_i, I_i, S_i, \delta_{INT,i}, \delta_{EXT,i}, \lambda_i, D_i >$$

$I_i \subseteq D$, es el conjunto de modelos influenciados por el modelo i;

$Z_{ij} = Y_i \rightarrow X_j$, es la función de traducción de la influencia *i* en *j*;

select: $D \rightarrow D$, es la función para determinar prioridades ante eventos simultáneos.

3.3 Mecanismo de Simulación

La simulación comienza con la inicialización de los modelos que la componen, con el objetivo de que cada componente establezca su estado inicial, e informe la planificación de su próxima transición interna. A partir de este momento la simulación comienza a ejecutar considerando los eventos externos y la última planificación de transición interna. El próximo evento a procesar será el que posea la hora más cercana a la hora actual. Esto implica comparar los eventos externos con los internos y seleccionar el menor.

Una vez determinado el próximo evento se le comunica al modelo correspondiente. Si se trata de un modelo acoplado, éste seleccionará la mínima de las planificaciones para todas sus componentes. El modelo que tenga menor tiempo de planificación se lo denomina **inminente**. Si más de un modelo coincide en la hora mínima, la función de selección ***select*** elegirá uno. Posteriormente se reenviará el mensaje recibido al mismo.

Si el modelo receptor es un modelo atómico y el evento recibido es un **evento externo**, el modelo invocará su función de transición externa. El resultado de esta invocación es una posible replanificación de su próximo evento interno. Si en cambio el evento recibido es un **evento interno**, en primer lugar se ejecuta su función de salida, lo que generará un evento como salida que será enviado a las influencias del modelo como una entrada, previamente pasando a través del modelo acoplado, el cual utiliza la función de traducción Z_{ij} para obtener las influencias del modelo que emitió la salida. Posteriormente, la función de transición interna es ejecutada, dando como resultado un cambio de estado y la planificación para su próxima transición interna.

Los comportamientos de las funciones de transición interna y externa dependen del comportamiento del modelo atómico.

4 Autómatas Celulares

Los *Autómatas Celulares* son modelos basados en un formalismo de variables y tiempo discretos, utilizados para modelar sistemas dinámicos complejos, creados originalmente por John von Neumann y Stephan Ulam [Tof94]. Estos permiten definir un conjunto infinito n-dimensional de celdas ubicadas en forma de grilla. Cada celda tiene igual comportamiento y tipo de estado que el resto de los elementos de la grilla. El **vecindario** de una celda es un conjunto de celdas cercanas a la misma que permite actualizar su estado en forma independiente a las demás. Este vecindario es homogéneo para todas las celdas. El estado de las celdas se actualiza en forma simultánea e independiente de los demás pasos de tiempo discreto [Wol86].

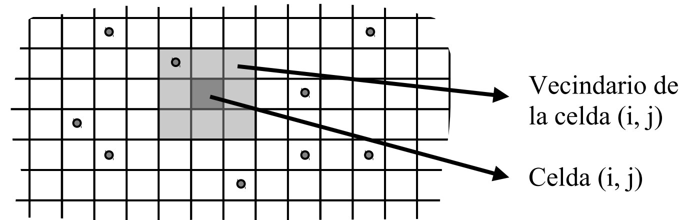


Figura 2 – Fragmento de un Autómata Celular Bidimensional

Los autómatas celulares suelen usarse, al igual que las ecuaciones diferenciales, para el modelado de sistemas físicos. Existen autómatas para modelar dinámica en fluidos y gases, colonias de animales, tránsito vehicular, modelos ecológicos y problemas de criptografía, entre otros. 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.

El formalismo utiliza una base de tiempo discreta, la cual restringe la precisión y eficiencia de los modelos simulados. En el caso particular de autómatas celulares que contienen un número considerable de celdas y reglas de alto grado de complejidad para ser resultas, será necesaria una gran cantidad de tiempo de cómputo para obtener el grado de precisión deseado. Más allá de esto, en muchos casos la mayoría de las celdas del autómata no necesitan actualización. La presencia de celdas inactivas permite la definición de un nuevo paradigma llamado **autómatas celulares asincrónicos**. Se llama un evento en el autómata a un cambio de estado en una celda. Por ende, se puede decir que en un autómata celular los eventos pueden ocurrir en un número reducido de celdas (aunque es posible que ocurra en todas simultáneamente), y que puede detectarse si ocurrirá un evento en una celda mirando si hubo eventos en sus vecinos. En este caso los eventos pueden ocurrir en un instante impredecible de tiempo por lo cual se aproxima a una base de tiempo continuo, lo que permite lograr mayor precisión y evitar la simulación de los períodos de inactividad de las celdas, mejorando así la utilización de los recursos computacionales y obteniendo una simulación con mejor rendimiento. Sin embargo, este enfoque requiere la sincronización explícita de las celdas, incrementando así la complejidad de las rutinas y algoritmos que llevan a cabo la simulación. Por lo tanto se produce una sobrecarga inherente al problema mencionado con anterioridad. En algunos casos esta sobrecarga puede anular las mejoras obtenidas con el enfoque asincrónico.

Para lograr simular estos modelos, se debe mantener una lista de últimos eventos, que contiene las celdas cuyos eventos ocurrieron recientemente, junto con la hora de ocurrencia de los mismos. Luego de simular cada una de las transiciones, se chequea si hubo un cambio de estado, y si es así la celda se agrega en la lista de últimos eventos. Este mecanismo puede causar distintos resultados dependiendo del orden de ejecución de las celdas, especialmente cuando las mismas son evaluadas en paralelo, transformando al autómata en un modelo no determinístico. Para evitar estos problemas, todas las celdas activas que figuren en la lista de eventos deben ejecutar sobre el mismo espacio de celdas de base, antes de hacerle cualquier modificación.

Formalmente un Autómata Celular Asincrónico ejecutable se define como:

$$CA = < A, S, n, \{t_1,...,t_n\}, C, \eta, N, B, Nevs, T, \tau, \boldsymbol{R}_0^+ >$$

donde:

A = $\{A \subseteq \boldsymbol{R} \wedge \#A < \infty\}$: Alfabeto de estados de celdas;

S = $\{S \in A\}$: Conjunto de posibles estados para cada celda;

n $\in \boldsymbol{N}$, es la dimensión del espacio de celdas;

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

C = $\{ C_c$ / c = $(i_1,...,i_n)$, con $0 \leq i_k \leq t_k \ \forall\ k = 1...n$, es la posición de la celda c en el espacio n-dimensional $\wedge\ C_c \in S$ }, es el conjunto de estados de cada una de las celdas del autómata;

η es la cantidad de celdas que integran el vecindario;

N = { $(v_{k1},...,v_{kn})$, $\forall$ k = 1... η } es la definición del vecindario como un desplazamiento;

B es el conjunto de celdas de borde. Se define:
B = {∅} si el espacio de celdas es toroidal; o
B = $\{C_b / C_b \in C$, con $b \in L\}$, siendo L = $\{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \ \forall \ j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \ \forall \ c \notin L$.

Nevs = $\{ (c, t) / c \in C \wedge t \in \boldsymbol{R}_0^+\}$, es una lista de próximos eventos, donde *c* es la posición de una celda en el espacio, y *t* es la hora del evento correspondiente;

T = $\{C \times \boldsymbol{R}_0^+ \rightarrow C\}$: Función de transición global;

$\boldsymbol{\tau}$ = $\{C_i \times N \times \boldsymbol{R}_0^+ \rightarrow C_k\}$: función de cómputo local;

$\boldsymbol{R}_0^+$: Base de tiempo (continua) del autómata celular;

Cada celda en el espacio puede tomar un valor del conjunto *S*. La evolución del autómata se define por la ejecución de la función de transición global (**T**) que actualiza el estado en todo el espacio de celdas (aunque en los autómatas celulares asincrónicos esta función actúa solo sobre las celdas activas). El comportamiento de esta función de transición depende de los resultados de la función de cómputo local ($\boldsymbol{\tau}$), que se ejecuta localmente en cada vecindario perteneciente a una celda (**N**). Conceptualmente, el cómputo de esta función de transición local se realiza en forma sincrónica y paralela en cada celda del modelo.

Para cada autómata celular bidimensional la función de transición local en el instante de tiempo *t* se define como:

$$s_{ij}[t+1] = \tau(s_{i^0, j^0}[t],..., s_{i^\eta, j^\eta}[t])$$

Aquí $s_{i,j}[t]$ representa el estado de la celda (i, j) en un tiempo de simulación *t*, y τ denota la función de cálculo local. Los parámetros $s_{i^0, j^0},..., s_{i^\eta, j^\eta}$ definen el vecindario de la celda. Una vez hecha la definición para dos dimensiones esto puede extenderse a n.

5 Formalismo *Cell–DEVS*

Cell-DEVS [Wai98] es un formalismo que se basa en DEVS, extendiéndolo para poder implementar autómatas celulares. En el paradigma Cell-DEVS cada celda se define como un modelo atómico con un conjunto de estados para los valores del vecindario, un comportamiento y una demora de actualización que puede ser de transporte ó inercial [GM76]. Un modelo acoplado que incluya a un conjunto de estas celdas formará así a un modelo celular.

Los modelos atómicos Cell-DEVS con dominio en los reales, pueden describirse formalmente como:

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

donde para el alfabeto A, con $\#A < \infty \ \wedge \ A \subseteq \ \boldsymbol{R} \cup \{ T, F, ? \}$:

$\boldsymbol{X} \subseteq$ A es el conjunto de eventos externos de entrada;

$\boldsymbol{Y} \subseteq$ A es el conjunto de eventos externos de salida;

I = $< \eta, \mu, P^x, P^y >$ representa la definición de la interfaz del modelo. En este caso,
$\eta \in \boldsymbol{N}, \eta < \infty$ es el tamaño de la vecindad,

$\mu \in \boldsymbol{N}, \eta < \infty$ es la cantidad de puertos de entrada/salida independientes de la vecindad,
P^X es el puerto de entrada que acepta valores de A.
P^Y es el puerto de salida que acepta valores de A.
$\forall\, j \in [1, \eta], i \in \{X, Y\}$ definimos el puerto: $P_j^i = \{ (N_j^i, T_j^i)$ /

$\forall\, j \in [1, \eta+\mu],\ N_j^i \in [i_1, i_{\eta+\mu}]$ (nombre del puerto), y $T_j^i \in I_i$ (Tipo del puerto)},
$I_i = \{ x / x \in X \text{ si } i = X \}$ ó $I_i = \{ x / x \in Y \text{ si } i = Y \}$;

La **demora** puede ser *de transporte* o *inercial*;

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

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

- Si la demora es de transporte:
  $\theta = \{ (s, phase, \sigma_{queue}, \sigma)$ / $s \in S$ es el valor del estado para la celda,
  $phase \in \{activa, pasiva\}$,
  $\sigma_{queue} = \{ ((v_1,\sigma_1),...,(v_m,\sigma_m)) / m \in \boldsymbol{N} \wedge m < \infty)$
  $\wedge \forall (i \in \boldsymbol{N}, i \in [1,m]), v_i \in S \wedge \sigma_i \in \boldsymbol{R}_0^+ \cup \infty\}$,
  $\sigma \in \boldsymbol{R}_0^+ \cup \infty$
  };

- Si la demora es inercial:
  $\theta = \{ (s, phase, \sigma)$ / $s \in S$ es el valor del estado para la celda,
  $phase \in \{activa, pasiva\}$,
  $\sigma \in \boldsymbol{R}_0^+ \cup \infty$
  };

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

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

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

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

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

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

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

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

Aquí **D**$(s, phase, \sigma_{queue}, \sigma, N, d) = t$ representa el tiempo durante el cual, si no hay eventos externos, el modelo atómico conservará el estado actual.

La especificación permite definir una celda como un modelo atómico modular. Un modelo Cell-DEVS acoplado n-dimensional puede definirse como:

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

donde:

Ylist = $\{ (k_1, k_2, ..., k_n) / k_i \in [0,t_i] \; \forall i \in [1,n], i \in \boldsymbol{N} \}$, es la lista de acoplamiento externo;

Xlist = $\{ (k_1, k_2, ..., k_n) / k_i \in [0,t_i] \; \forall i \in [1,n] , i \in \boldsymbol{N} \}$, es la lista de acoplamiento interno;

I = $< P^X, P^Y >$, es la interfaz de comunicación con los modelos externos, donde:
P^x es un port de entrada que acepta valores de $\boldsymbol{R}$.
P^y es un port de salida que acepta valores de $\boldsymbol{R}$.

n $\in N$, $n < \infty$ es la dimensión del espacio de celdas;

$\{t_1,..., t_n\}$, con $t_i \in N$, $1 \leq i \leq n$, es la cantidad de celdas en cada una de las dimensiones;

$\eta \in N$, es el tamaño del vecindario;

$\boldsymbol{X} \subseteq \boldsymbol{R}$, es el conjunto de eventos externos de entrada;

$\boldsymbol{Y} \subseteq \boldsymbol{R}$, es el conjunto de eventos externos de salida;

N = $\{ (i_{1,p}, i_{2,p}, ..., i_{n,p}) / i_{j,p} \in \boldsymbol{Z} \; \forall j \in N, j \in [1,n], \forall p \in N, p \in [1,\eta] \}$. Es la definición del vecindario, expresada como un desplazamiento con respecto a la celda central.

C = $\{C_{k1, k2, ..., kn} / k_i \in [0,t_i] \; \forall i \in N, i \in [1,n]\}$, es el conjunto de celdas atómicas que conforman al modelo acoplado, donde:

$$C_{k1, k2, ..., kn} = < X_{k1, k2, ..., kn}, Y_{k1, k2, ..., kn}, I_{k1, k2, ..., kn}, demora_{k1, k2, ..., kn}, S_{k1, k2, ..., kn}, \theta_{k1, k2, ..., kn}, N_{k1, k2, ..., kn}, d_{k1, k2, ..., kn}, \delta_{int, k1, k2, ..., kn}, \delta_{ext, k1, k2, ..., kn}, \tau_{k1, k2, ..., kn}, \lambda_{k1, k2, ..., kn}, D_{k1, k2, ..., kn} >$$

B es el conjunto de celdas que representan el borde del modelo celular, donde:

- $B = \{\emptyset\}$ si el espacio de celdas es toroidal (Wrapped); o
- $B = \{C_{k1, k2, ..., kn} / (k_1 = 0 \vee k_1 = t_1) \wedge (k_2 = 0 \vee k_2 = t_2) \wedge ... \wedge (k_n = 0 \vee k_n = t_n) \wedge C_{k1, k2, ..., kn} \in C \wedge C_{k1, k2, ..., kn} = < X_{k1, k2, ..., kn}, Y_{k1, k2, ..., kn}, I_{k1, k2, ..., kn}, demora_{k1, k2, ..., kn}, S_{k1, k2, ..., kn}, \theta_{k1, k2, ..., kn}, N_{k1, k2, ..., kn}, d_{k1, k2, ..., kn}, \delta_{int, k1, k2, ..., kn}, \delta_{ext, k1, k2, ..., kn}, \tau_{k1, k2, ..., kn}, \lambda_{k1, k2, ..., kn}, D_{k1, k2, ..., kn} >$ es un componente atómico }, si las celdas atómicas del borde tienen distinto comportamiento que el resto del espacio de celdas;

Z: $I_{k1, k2, ..., kn} \rightarrow I_{w1,w2,...,wn}$, define el acoplamiento interno (conexión de puertos de E/S entre celdas):

$Z(P_{k1, k2, ..., kn}{}^{Y}{}_{q}) = P_{w1,w2,...,wn}{}^{X}{}_{q}$, con $(q \in N, q \in [1,\eta]) \wedge \forall (r_1,r_2,...,r_n) \in N$, $w_1 = (k_1+r_1) \bmod t_1$; $w_2 = (k_2+r_2) \bmod t_2$; ...; $w_n = (k_n+r_n) \bmod t_n$; y

$Z(P_{k1, k2, ..., kn}{}^{X}{}_{q}) = P_{w1,w2,...,wn}{}^{Y}{}_{q}$, con $(q \in N, q \in [1,\eta]) \wedge \forall (s_1,s_2,...,s_n) \in N$, $w_1 = (k_1-s_1) \bmod t_1$; $w_2 = (k_2-s_2) \bmod t_2$; ...; $w_n = (k_n-s_n) \bmod t_n$

select = $\{ (k_1, k_2, ..., k_n) / k_i \in [0,t_i] \; \forall i \in N, i \in [1,n] \}$ es la función de selección de una celda inminente ante eventos simultáneos.

Las celdas con **demora de transporte** usan una cola para mantener los valores de los resultados de los cálculos junto con su hora futura de planificación, ya que durante la demora pueden llegar nuevos eventos externos. Cuando llega un evento por medio de la función de transición externa, la función de cómputo local es invocada, la cual, utilizando los valores de todas las entradas disponibles (el vecindario y los valores ingresados por los puertos) obtiene como resultado un estado al cual la celda debe cambiar, y una demora. Si este estado es distinto al anterior debe encolarse en la cola de

próximos eventos [WFG97]. Debido a que una celda es independiente de la otra, cada una guarda una copia de los estados de los vecinos, así como el estado anterior.
Al arribar un evento interno se invocan primero la función de salida, que envía como resultado al primer valor encolado, y luego la función de transición interna, que elimina el primer elemento de la cola y luego analiza si la cola está vacía. Si existe un elemento significa que el modelo debe reprogramarse en función de lo encolado, en caso contrario debe pasivarse y por lo tanto su próximo cambio de estado será a la hora infinito [BB98].
La celda se mantiene activa mientras haya eventos en la cola, cuando ésta se vacía la celda se pasivará retornando como hora de próximo evento el valor infinito.

Por otro lado existen las celdas con **demora inercial**. Las mismas son útiles cuando se desea representar una semántica con remoción para el comportamiento de una celda. Frente a un arribo de un evento externo la celda ejecuta la función de cómputo local y obtiene como resultado un estado y una demora. Si el nuevo valor obtenido difiere del valor anterior la celda analiza tiempo restante para el próximo evento interno. Si no existe planificación alguna la celda se programa con la demora recién calculada. En caso de existir una programación se analiza el valor de σ (tiempo restante) para comprobar si es mayor que cero, de ser así se descarta la programación anterior y se toma la nueva. De no ser así, continúa con lo planificado. Frente a un evento interno la celda realiza la función de salida con el valor calculado, y en la función de transición interna cambia su estado a pasivo, ya que no tiene más eventos para programar [BB98].

6 Modelos Cell–DEVS Achatados

Un modelo celular convencional está compuesto de tantos modelos atómicos como celdas posea el autómata. Esto produce una gran inversión de tiempo en la creación de los modelos, y una sobrecarga de mensajes por cada evento que arriba al modelo celular. Como todas las celdas comparten la misma estructura, tienen el mismo comportamiento en función de su estado, y es posible obtener sus vecinos aplicando un desplazamiento a su ubicación, toda la estructura necesaria para llevar a cabo la simulación de este tipo de modelo puede simplificarse y verse como una matriz n-dimensional de estados junto a una especificación del lenguaje a utilizar. Esta simplificación permite disminuir drásticamente el tiempo de creación y elimina por completo el pasaje de mensajes dentro del modelo celular. Para tal tarea se usan los modelos celulares achatados [Wai98].

En un modelo celular achatado, frente a un mensaje externo, se resolverá cuáles son las celdas virtuales influenciadas por el mismo, y en lugar de enviarle un mensaje externo a cada una de ellas se invoca directamente a la función de cómputo local y se aplica la demora correspondiente, modificando la lista de próximos eventos de ser necesario. Al finalizar, el coordinador achatado envía un mensaje *done* a su padre con la hora del próximo cambio de estado para la celda virtual inminente.

Un mensaje interno le indica al coordinador achatado que al menos una de las celdas debe realizar su cambio de estado. Para todas las celdas que corresponda se generará su salida y se agregarán todas las celdas influenciadas en la lista de próximos eventos.

Este tratamiento de los eventos elimina por completo los mensajes entre el acoplado celular achatado y sus hijos, ya que los mismos son virtuales y no existen como modelos, y de esta manera los procesadores intermedios no son necesarios. A su vez los mensajes de respuesta se reducen considerablemente debido a que, ante cada mensaje recibido, se efectúan todas las tareas posibles que correspondan al conjunto de celdas. Estos factores reducen el tiempo total de ejecución.

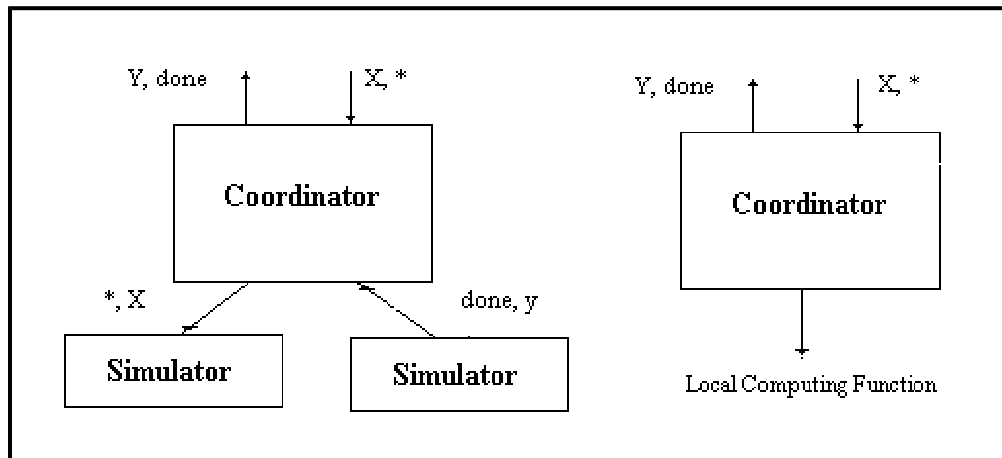


Figura 3 – Modelo Acoplado Celular Jerárquico vs Achatado [Wai98]

7 CD++

CD++ [BBW98b] [Wai98] [RW99a] es una herramienta que permite implementar los conceptos teóricos especificados por los formalismos DEVS y Cell–DEVS. Fue construida usando C++ definiendo una jerarquía de clases para los modelos atómicos que extiende a la definida en ***GAD*** [BBW98a], la cual no incluía modelos celulares. Un lenguaje de especificación permite la creación de modelos acoplados, definir la configuración inicial para los modelos atómicos y la creación de eventos externos para poder ser usados durante la simulación, así como también permite describir el comportamiento de cada celda de un modelo celular.

La herramienta permite la creación de autómatas celulares bidimensionales, donde el estado de una celda tiene un valor perteneciente a una lógica trivalente compuesta por los elementos *Verdadero*, *Falso* e *Indefinido*.

En la jerarquía de clases definida existen dos clases abstractas de las que se derivan las restantes: *Model* y *Processor*. La primera se usa para representar a los distintos modelos ya sean atómicos o acoplados, mientras que la segunda implementa el mecanismo de simulación. La asociación entre modelos y procesadores está dada por los pares *Atomic-Simulator* y *Coupled-Coordinator.* Por cada modelo existente existirá un único procesador asociado y cada procesador administrará un único modelo.

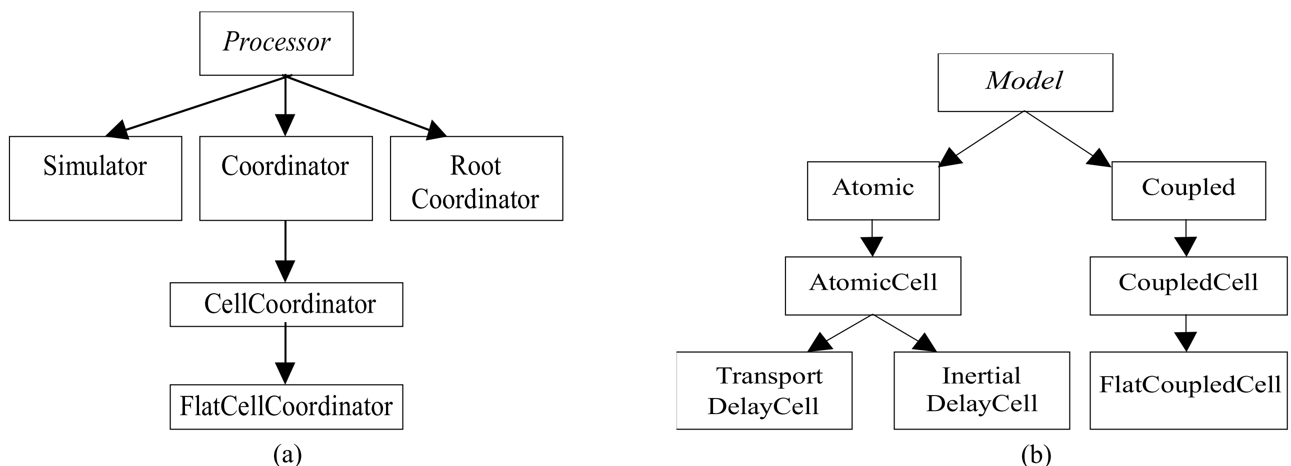


Figura 4 – (a) Subclases de *Processor*. (b) Subclases de *Model*

La clase *Atomic* implementa el comportamiento de los modelos atómicos. La misma dispone de los métodos *int-transfn*, *ext-transfn*, *outputfn* y *time-advancefn* que representan las respectivas funciones de transición externa, interna, salida y avance de tiempo del modelo.

La clase *Coupled-Model* implementa los mecanismos de los modelos acoplados.

Para el caso de un modelo celular, se utiliza un modelo atómico especial para representar a cada celda. En este caso tenemos *AtomicCell* y *CoupledCell* que son subclases de *Atomic* y de *Coupled* respectivamente.

AtomicCell extiende el comportamiento de los modelos atómicos, para definir el funcionamiento de las celdas del espacio, agregando la posibilidad de poseer un estado y un comportamiento (determinado por la función de cómputo local). Esta clase tiene dos especializaciones: *TransportDelayCell* e *InertialDelayCell*, que redefinen su comportamiento según el tipo de demora de actualización (transporte o inercial). Por otra parte, la clase *CoupledCell* permite la administración de un conjunto de celdas atómicas, mientras que la clase *FlatCoupledCell* implementa el mecanismo de los modelos celulares achatados.

La clase *Root-Coordinator* maneja la simulación en su totalidad y se relaciona con el modelo acoplado que tenga el nivel más alto dentro de la jerarquía. Se encarga de mantener el tiempo global, y esta a cargo del comienzo y fin de la simulación. Además, recolecta los resultados de salida. Las clases *Simulator* y *Coordinator* administran la simulación de los modelos atómicos y acoplados respectivamente. La clase *CellCoordinator* es una subclase de *Coordinator*, y esta asociada a un modelo celular acoplado. Ésta, a su vez, tiene como subclase a *FlatCellCoordinator*, que se encarga de la administración de los modelos celulares achatados.

El proceso de simulación se basa en el intercambio de mensajes entre los distintos procesadores, llevando información con respecto a los eventos internos y externos, así como también datos necesarios para la simulación. Cada mensaje contiene información para identificar al emisor del mismo, el tiempo del evento, y el contenido que consiste en un puerto y un valor. Existen cuatro tipos de mensajes: *X* (representa un evento externo), *Y* (representa la salida del modelo), * (representa un evento interno), y *done* (indica que el modelo terminó su procesamiento y una posible replanificación de la próxima transición interna).

La simulación comienza con la inicialización de cada uno de los modelos atómicos. De esta forma se determina la hora de próximo evento. La inicialización se propaga en forma descendente por el árbol y obtiene como resultado una serie de mensajes *done* con la planificación de cada uno de los modelos que componen la simulación. Cuando el *Root-Coordinator* recibe un mensaje *done* de su hijo inminente responde con un mensaje* llevando la hora de su próximo evento. De esta forma se producen series de mensajes hacia las hojas del árbol que resultan en mensajes *Y* y *done* como respuesta hacia sus respectivos padres. El último de los mensajes *done* llegará al simulador raíz y se producirá otra iteración en la simulación.

Cuando un modelo inminente es seleccionado, se envía un mensaje * a su simulador, pasando a través de los coordinadores cuyos modelos acoplados contienen al modelo atómico en cuestión. Cuando este mensaje llega al simulador, se ejecuta la función de transición interna del modelo asociado.

Cuando al simulador asociado a un modelo atómico le llega un mensaje *X* se ejecuta la función de transición externa. Los simuladores responden con mensajes *Y* y/o *done*, los cuales son convertidos como nuevos mensajes *X* y * respectivamente. Cuando un coordinador recibe un mensaje *Y* de su hijo inminente consulta su esquema de acoplamiento de salida externa para decidir si debe transmitirlo a su padre, y también consulta su acoplamiento interno para obtener los hijos a los cuales el mensaje debe ser enviado. El coordinador esperará un mensaje *done* de cada una de las influencias de los hijos. Teniendo todas las respuestas de sus hijos podrá calcular el hijo inminente para la próxima transición interna, devolviendo esta hora en un mensaje *done* a su padre.

Existen otras clases, las cuales son usadas para la administración y puesta en marcha de la simulación. La clase *Message* es una clase abstracta que define los distintos tipos de mensajes que son enviados entre los procesadores durante la simulación. Estos mensajes indican el tipo de evento que debe ser aplicado al modelo que esta asociado al procesador receptor del mismo. *InitMessaage*, *ExternalMessage*, *InternalMessage*, *OutputMessage* y *DoneMessage* son las subclases de *Message*, y representan a los eventos de inicialización, externos, internos, de salida y done respectivamente.

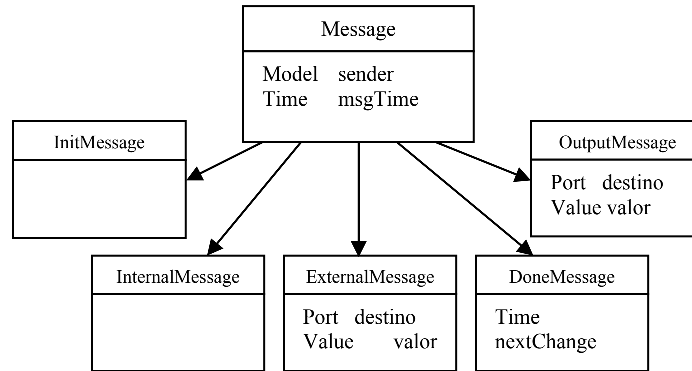


Figura 5 – Subclases de *Message*

Los procesadores tienen como responsabilidad tomar decisiones en función del mensaje que arribe, sin embargo, no son responsables del mecanismo de pasaje de mensajes. Para esto existe un administrador de mensajes (clase *MessageAdm*) que se encarga del intercambio de mensajes entre los módulos. Esta separación favorece la independencia de las clases y permite, por ejemplo, cambiar la política de envío y recepción de mensajes, o realizar una implementación distribuida sin impactar en el código.

La clase *MainSimulator* es la responsable de crear el árbol de modelos/procesadores y los vínculos entre sus puertos a partir de una especificación. La clase *ModelAdm* se encarga de la creación de los modelos, mientras que la clase *SimLoader* es usada para la carga de los eventos externos, los tiempos de comienzo y fin de simulación, y de la obtención de los distintos parámetros que serán utilizados para la configuración inicial de cada modelo.

8 N–CD++

N–CD++ es una extensión de la herramienta *CD++* que permite la creación de modelos Cell–DEVS n–dimensionales, donde el estado de cada celda pertenece al conjunto $\boldsymbol{R} \cup \{ \textbf{?} \}$ y **?** representa al valor *indefinido*. Incluye un lenguaje de especificación que permite describir el comportamiento de cada celda de un modelo celular. Además, permite definir el tamaño del espacio celular y su conexión con otros modelos DEVS (si los hubiese), así como también el tipo de demora, el vecindario, el borde y el estado inicial de cada celda. Para ello se siguen las definiciones teóricas del formalismo *Cell-DEVS*.

La especificación del comportamiento de una celda se logra definiendo un conjunto de reglas de la forma:

VALOR DEMORA { CONDICIÓN }

Cada regla indica que si se satisface la condición, el estado de la celda debe cambiar por el valor designado. Luego, debe demorarse usando el tiempo especificado. Si la condición no es verdadera, entonces se evaluará la siguiente regla (secuencialmente según el orden en que fueron definidas). Este proceso se repetirá hasta que una regla sea satisfecha (considerando sólo la primera que lo haga), o hasta que no haya más reglas. En este último caso, se producirá un error, indicando al modelador tal situación y abortando la simulación. La ocurrencia de este error indica que el modelo ha sido especificado en forma incompleta. Existen otros errores que pueden abortar la ejecución, y generalmente se deben a la falta o a una incorrecta declaración de ciertos parámetros que describen las características del modelo (como la dimensión del espacio celular, el tipo de demora, la definición del vecindario, o los valores iniciales a ser usados por el modelo), o a la definición de un acoplamiento entre modelos que involucre un puerto inexistente.

Cabe considerar que en la definición de una regla, el segundo valor, que se corresponde con la demora de la celda, puede ser un número real, ya sea en forma directa o como resultado de la evaluación de una expresión. Sin embargo, si no es un número entero, este será automáticamente truncado de tal forma que su valor sí lo sea. Por otra parte, si su valor es indefinido (**?**) entonces se informará tal situación y se producirá un error, abortando la simulación.

Un conjunto de reglas permite definir el comportamiento para las celdas de un modelo celular. Consideremos, por ejemplo, el comportamiento del "*Juego de la Vida*" [Gar70], cuyas reglas se muestran en la Figura 6. Esta especificación indica que si la celda esta activa y la cantidad de vecinos activos (incluyendo la celda en cuestión) es 3 ó 4, entonces la

celda seguirá activa (con una demora de transporte de 10 milésimas de segundo). Si la celda esta inactiva, y el vecindario tiene 3 elementos activos, entonces la celda pasa a estar activa. Sino, la celda pasa a estar inactiva.

```
Rule: 1 10 { (0,0) = 1 and ( truecount = 3 or truecount = 4 ) }
Rule: 1 10 { (0,0) = 0 and truecount = 3 }
Rule: 0 10 { t }
```

Figura 6 – Reglas para el *Juego de la Vida*

El formalismo para Autómatas Celulares ha sido extendido por diversidad de autores, permitiendo incluir distintas propiedades. Entre ellas se destacan:

- **Autonomía**: un modelo celular puede tener entradas externas, independientemente de su vecindario. En *N-CD++*, tales entradas se corresponden a puertos que conectan un modelo DEVS con una celda específica del modelo celular.

- **Homogeneidad**: cada celda del modelo puede tener distintas reglas y conexiones, siendo en estos casos un autómata celular inhomogéneo. En *N-CD++* esto se logra mediante la definición de *zonas*.

- **Uniformidad:** otras extensiones permiten que los vecinos no sean las celdas más cercanas, permitiendo el uso de vecindarios más extensos.

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

- **Determinismo:** un autómata celular asincrónico estocástico puede obtenerse definiendo un experimento aleatorio e incluyendo variables aleatorias en las reglas que definen el comportamiento del modelo. Para ello, la función de transición local debe permitir el uso de variables aleatorias.

Todas estas propiedades son soportadas por *N-CD++* y las mismas serán analizadas con mayor profundidad en las siguientes secciones junto a otras extensiones efectuadas sobre *CD++*.

8.1 Extensión del Lenguaje de Especificación de Modelos

Como se mencionó previamente, *CD++* sólo permite que el estado de una celda tenga un valor perteneciente al conjunto {*Verdadero*, *Falso*, **?**}, donde el símbolo **?** representa un valor indefinido, y los valores *Verdadero* y *Falso* pueden ser usados indistintamente en lugar de 1 y 0 respectivamente. Esto restringe los modelos a ser implementados, limitando así el uso de la herramienta en problemas más generales.

Para evitar esta restricción, se decidió la expansión del conjunto de posibles valores para un estado, permitiendo así que el estado de una celda tenga un valor perteneciente al conjunto $\boldsymbol{R} \cup \{ \textbf{?} \}$. De aquí en más al citar el término *número real* estaremos referenciando a un elemento de este conjunto.

La expansión del conjunto de posibles valores para un estado implicó no solo la modificación del estado de una celda, sino también la expansión del lenguaje para poder manipular estos nuevos valores durante la definición de las reglas [RW99a], cuyo BNF es mostrado en el *Apéndice*.

CD++ permite que los valores numéricos usados en la definición de la condición de la regla puedan ser expresiones calculables en tiempo de evaluación de la misma. Las modificaciones realizadas al lenguaje posibilitan que el valor del estado como así también la demora posean la misma cualidad, permitiendo el uso de cualquier función matemática soportada e incluso referenciar el valor de la celda en cuestión o el de una de sus vecinas. Cabe destacar que

si el autómata celular no se define con forma toroidal, entonces la referencia a un vecino de una celda puede producir el acceso a un elemento fuera del espacio celular. En este caso el valor obtenido será indefinido (**?**).

Debido a errores de representación en la computadora, como así también a errores de redondeo, es posible que dos valores numéricos que teóricamente deberían ser iguales, no lo sean. Para solucionar este inconveniente, se permite definir que dos valores *a* y *b* sean iguales si *a* pertenece al intervalo [b-Δ, b+Δ]. El valor de Δ puede ser especificado por el modelador a través de un parámetro en la invocación al simulador, variando así la precisión considerada al efectuar los cálculos.

N-CD++ dispone además de un preprocesador que proporciona ciertas facilidades al lenguaje y que actúa sobre el archivo de definición de modelos, antes de la carga de los mismos. Dicho preprocesador efectúa las tareas de expansión de macros y descarte de comentarios. El uso de macros permite la escritura de funciones de transición, reglas y/o definición de modelos o parte de ellos, en archivos independientes, facilitando su reusabilidad.

8.1.1 Incorporación de Funciones Determinísticas

Al lenguaje para la definición de reglas que dan comportamiento a las celdas, se han incluido las operaciones básicas para manipular los valores de la lógica trivalente, las cuales son mostradas en la Figura 7. Además, esta lógica dispone de las constantes **T** ó **1** para representar al valor *TRUE*, **F** ó **0** para representar al *FALSE*, y **?** para representar al *INDEFINIDO*.

AND	**T**	**F**	**?**
T	T	F	?
F	F	F	F
?	?	F	?

OR	**T**	**F**	**?**
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

XOR	**T**	**F**	**?**
T	F	T	?
F	T	F	?
?	?	?	?

IMP	**T**	**F**	**?**
T	T	F	?
F	T	T	T
?	T	?	?

EQV	**T**	**F**	**?**
T	T	F	F
F	F	T	F
?	F	F	T

Figura 7 – Comportamiento de los operadores para la lógica trivalente

El lenguaje también dispone de los operadores =, **!=**, <, >, <= y >= para la comparación de *números reales*[1], cuyo comportamiento se resume en la Figura 8. Estos operadores devuelven un valor booleano perteneciente a la lógica trivalente anteriormente definida.

=	?	$n \in R$
?	T	?
$m \in R$	?	$m = n$

!=	?	$n \in R$
?	F	?
$m \in R$	?	$m \neq n$

>	?	$n \in R$
?	F	?
$m \in R$	?	$m > n$

<	?	$n \in R$
?	F	?
$m \in R$	?	$m < n$

<=	?	$n \in R$
?	T	?
$m \in R$	?	$m \leq n$

>=	?	$n \in R$
?	T	?
$m \in R$	?	$m \geq n$

Figura 8 – Comportamiento de los operadores relacionales

Considerando las definiciones del comportamiento de estos operadores puede verse que no existe un orden total sobre los elementos pertenecientes a los números reales, debido a que en todos los casos el valor **?** no es comparable con un número real tradicional.

El lenguaje dispone también de las operaciones aritméticas básicas (+, -, * y /). Además, se han agregado distintos tipos de funciones para operar sobre *números reales*, como por ejemplo, funciones trigonométricas, obtención de raíces, potencias, redondeo y truncamiento de valores reales a enteros, generación de números primos, módulo, logaritmos, valor absoluto, factoriales, como así también cálculo de mínimos, máximos, M.C.D. y M.C.M; considerando

[1] Considerando un *número real* a un valor perteneciente al conjunto $\boldsymbol{R} \cup \{ \textbf{?} \}$

que cualquier expresión o función que incluya al valor **?** dará como resultado **?**. Otras funciones existentes permiten chequear si un *número real* es entero, si es par o impar, si es un número primo, ó si es un valor indefinido.

Existen funciones para obtener la cantidad de celdas del vecindario cuyo estado tiene cierto valor. Por ejemplo, *truecount* devuelve la cantidad de celdas en estado 1. También están disponibles las funciones *falsecount*, *undefcount* y *statecount*(n). Esta última es la más genérica y permite especificar el valor del estado a contabilizar. El resto de las funciones de este tipo pueden ser definidas basándose en *statecount*, pero no han sido descartadas del lenguaje para ofrecer compatibilidad con *CD++*.

También se cuenta con la función *cellPos*, que devuelve la i–ésima posición dentro de la tupla que referencia a la celda que esta ejecutando la función de transición, es decir, dada la celda $(x_0,x_1,...,x_n)$, entonces cellPos(i) = x_i.

El lenguaje provee el uso de constantes predefinidas. Entre ellas se encuentran: *pi* y *e*, junto a ciertas constantes de uso frecuente en los dominios de la física y la química (como la constante gravitacional, de aceleración, velocidad de la luz, constante de Planck, etc.). También se definió la constante *INF*, que representa al valor infinito. Esta última se devuelve automáticamente cuando la evaluación de una expresión numérica produce un desborde numérico.

Se agrego la función *Time*, que permite la obtención del tiempo global de simulación expresado en milésimas de segundo.

Se añadieron funciones para obtener valores dependiendo del resultado de la evaluación de cierta condición, tal es el caso de la función IFU(*c*, *t*, *f*, *u*) que devuelve *t* si la condición *c* evalúa a *Verdadero*; *f* si evalúa a *Falso;* y *u* si evalúa a *Indefinido*; y la función IF(*c*, *t*, *f*) que devuelve *t* si *c* evalúa a *Verdadero*, y *f* en otro caso.

Finalmente, se incorporaron funciones para la conversión de valores entre distintas unidades. Se cuentan con las funciones *RadToDeg* y *DegToRad* para la conversión de ángulos expresados en radianes a grados sexagesimales y viceversa, respectivamente. También se cuentan con funciones para la conversión de coordenadas polares a rectangulares y viceversa, y para la conversión de valores que representan temperaturas en grados Centígrados, Fahrenheit o Kelvin.

8.1.2 Incorporación de Funciones Probabilísticas

Se agregaron funciones para la generación de números aleatorios, usando distintas distribuciones (Uniforme, Chi Cuadrado, Beta, Exponencial, F, Gama, Normal, Binomial y Poisson), lo que permite la simulación de modelos estocásticos. También se cuenta con la función *RandomSign* que devuelve un valor al azar, y que representa un signo numérico (+1 ó –1) en forma equiprobable.

Si se utilizan funciones de generación de números aleatorios en la definición de la condición de la regla, es posible que su evaluación dé como resultado el valor *Falso*. Consideremos por ejemplo la regla:

10 100 { random >= 0.4 }

cuya condición es evaluada a *Verdadera* en el 60 % de los casos, y a *Falsa* en el resto. Esto puede provocar que en algunos casos todas las reglas sean evaluadas a *Falso*, mientras que en otros casos exista una o más que no lo hagan. En este caso, por tratarse de un modelo estocástico, la herramienta automáticamente identifica esta situación y si hay más de una regla que es evaluada a *Verdadero*, se considera sólo la primera que lo haga, mientras que si todas las evaluaciones son *Falsas*, se asigna el valor **?** a la celda, y se utiliza el tiempo de demora establecido por defecto en la definición del modelo, informando tal situación al modelador y prosiguiendo con la simulación.

8.1.3 Optimización en la Evaluación de Reglas

El mecanismo de evaluación de las reglas ha sido optimizado con el objetivo de mejorar el tiempo total de ejecución de la herramienta. En *CD++* en el momento de evaluar cualquier tipo de expresión contenida en una regla, se procede a evaluar primero todos los operandos en forma completa, y a los resultados obtenidos se le aplica la operación necesaria. Por ejemplo, para aplicar la operación booleana:

{ (0,0) = 1 AND ((1,1) = 0 OR (2,2) = 3) }

se procede primero a evaluar recursivamente las expresiones:

(0,0) = 1

y

(1,1) = 0 OR (2,2) = 3

y una vez obtenidos los valores necesarios se aplica la operación AND. Debido a que la evaluación se hace en forma recursiva, para evaluar el segundo operando se procede a evaluar sus operandos y luego se le aplica la operación OR.

Sin embargo, este tipo de evaluaciones en forma completa muchas veces es innecesario y produce un overhead en el mecanismo de evaluación. Por ejemplo, si el valor de la celda (0,0) fuese distinto de 1, entonces el primer operando para el AND sería falso y por lo tanto ya es posible afirmar que la evaluación devolverá el valor falso, sin tener que evaluar el segundo operando, optimizando de esta forma el mecanismo de evaluación.

Estas consideraciones se han tenido en cuenta en *N-CD++*, con lo que ciertas operaciones binarias booleanas han sido tenido en cuenta para la optimización, de forma tal que no siempre sea necesario evaluar sus dos operandos para conocer el resultado de la operación. El mecanismo de evaluación de reglas procede a evaluar siempre el primer operando y dependiendo de su resultado y del tipo de operación se procede a evaluar el segundo operando o a devolver directamente un valor, evitando la evaluación del mismo. Esto debería ser tenido en cuenta por el modelador a la hora de escribir las reglas. En la Figura 9 se detallan las optimizaciones realizadas sobre los operadores binarios de *N-CD++* al evaluar las reglas. Ante cualquier combinación entre el resultado de la evaluación del operando izquierdo y una operación no definida en la figura se procede a la evaluación del operando derecho, y posteriormente a la evaluación de la operación con el resultado obtenido para ambos operandos.

Evaluación del Operando Izquierdo	Operación	Resultado de la Operación
FALSO	AND	FALSO
VERDADERO	OR	VERDADERO
INDEFINIDO	XOR	INDEFINIDO
FALSO	IMP	VERDADERO

Figura 9 – Optimización en la evaluación de las operaciones binarias de la lógica trivalente utilizada en *N-CD++*

8.1.4 Modo de Debug para Validación de Reglas

La herramienta dispone de un modo de debug que permite validar, en tiempo de ejecución, la existencia de dos o más reglas cuya condición sea satisfecha y que el valor a ser asignado a la celda tenga distintos resultados o demoras de actualización, evitando así la creación de modelos ambiguos.

Por defecto este modo se encuentra desactivado, por lo que las reglas son evaluadas en el orden en que fueron definidas hasta encontrar alguna que sea satisfecha. Mediante el uso del parámetro **–r** en la invocación al simulador se activa el modo de debug, por lo que todas las reglas son evaluadas, por más que alguna de ellas haya resultado previamente válida.

La Figura 10 muestra la implementación para *N-CD++* de una variante del juego de la vida con tres seres distintos, representados por los valores 0, 1 y 2.

```
[top]
components : life

[new-life]
type : cell
width : 20
height : 2
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
```

```
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue :  0      01121002111200211210
initialrowvalue :  1      02012001211200020020
localtransition : new-life-rule
```

Figura 10 – Implementación en *N-CD++* de una variante del Juego de la Vida (no incluye reglas)

Cuando se evalúan las reglas que definen el comportamiento de un modelo celular, si ninguna de ellas es válida la herramienta genera un error y aborta la simulación. Por ejemplo, en la Figura 11 se muestran un conjunto de reglas para la variante del *Juego de la Vida* previamente definida. Las mismas no están definidas en forma completa, por lo que existen casos en que todas las evaluaciones son falsas, por ejemplo cuando la celda para la cual se están evaluando las reglas tiene como estado el valor 2. Ante esta situación se producirá un error, como se muestra en la Figura 12 y la simulación finaliza de manera anormal. El mensaje mostrado describe el suceso ocurrido y es acompañado de los valores de estado para el vecindario de la celda que estaba ejecutando la función de cómputo local en el momento en que se produjo el error. Si el modelo tiene dimensión mayor a 2, entonces el vecindario será mostrado como una lista de las celdas que lo componen junto a sus respectivos valores de estado. El mensaje mostrado también informa el lugar donde se produjo el error, indicando los números de líneas y los archivos conteniendo el código fuente.

```
[new-life-rule]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 0 }
```

Figura 11 – Reglas para la variante del Juego de la Vida cuya evaluación puede resultar siempre falsa

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
----------------------------------------------------------
Version 2.0-R.43 October-1999
Daniel Rodríguez, Gabriel Wainer, Amir Barylko, Jorge Beyoglonian
Departamento de Computacion. Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires. Argentina.

Loading models from life.ma
Loading events from
Message log: life.log
Output to: /dev/null
Tolerance set to: 1e-08
Configuration to show real numbers: Width = 12 - Precision = 5
Quantum: Not used
Evaluate Debug Mode = OFF
Flat Cell Debug Mode = OFF
Debug Cell Rules Mode = OFF
Temporary File created by Preprocessor = C:\WINDOWS\TEMP\s3vvjps5.
Printing parser information = OFF

Starting simulation. Stop at time: 00:00:05:000

Exception Mexception thrown!
Description: None of the rules evaluate to TRUE!
Thrown in:
        File: synnode.cpp - Method: evaluate - Line: 138
        File: ltranadm.cpp - Method: evaluate - Line: 104
Description:
        Language used is: life-rule
        The state of the Neighbours is:
        +-----------------------------------------+
        |      0.00000       1.00000       2.00000 |
```

```
        |      1.00000       2.00000       1.00000 |
        |      0.00000       1.00000       2.00000 |
        +-----------------------------------------+
Aborting simulation...
```

Figura 12 – Error generado por la herramienta al no encontrar ninguna regla válida

Por otra parte, debido a que la herramienta utiliza una lógica trivalente, al evaluar las condiciones de un conjunto determinado de reglas podría obtenerse un valor *Verdadero*, *Falso* ó *Indefinido*. En caso de que ninguna condición evalúe a *Verdadero*, pero alguna evalúe a *Indefinido* entonces el estado de la celda pasará a tener el valor *Indefinido* y su demora será la establecida por defecto para la declaración del modelo. Ante esta situación se alerta al modelador del acontecimiento de esta situación, pero la simulación no es interrumpida. Por ejemplo, en la Figura 13 se muestra un nuevo conjunto de reglas que pueden ser usadas junto al modelo definido en la Figura 10. Cuando el valor de estado de la celda que esta ejecutando la función de cómputo local es 1 y el valor del vecino (0, 1) no es indefinido, la evaluación de la condición de la primer regla siempre dará como resultado el valor indefinido. Esto se debe a que el resultado de la comparación del valor de la celda (0, 1) con el valor indefinido es indefinido, y la operación AND cuando alguno de sus operandos es indefinido devuelve el valor indefinido. Además, para este caso, la evaluación de las condiciones para el resto de las reglas será falsa. Por lo tanto, ninguna condición fue evaluada a Verdadero, pero existe una que fue evaluada a Indefinido. Ante esta situación se produce un mensaje de alerta como se muestra en la Figura 14 y se prosigue con la simulación, asignándole el valor Indefinido a la correspondiente celda.

```
[new-life-rule]
rule : 2 100 { (0,0) = 1 and (0,1) = ? }
rule : 1 100 { (0,0) = 0 }
rule : 0 100 { (0,0) = 2 }
```

Figura 13 – Reglas para la variante del Juego de la Vida cuya evaluación puede no resultar verdadera para todas las celdas, pero existe alguna que sea indefinida

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
----------------------------------------------------------
...
Starting simulation. Stop at time: 00:00:05:000
...
Warning! - None of the rules evaluate to True, but any evaluate to undefined.
...
Warning! - None of the rules evaluate to True, but any evaluate to undefined.
...
Simulation ended!
```

Figura 14 – Alerta generada por la herramienta cuando no existen reglas válidas pero existe alguna regla indefinida

Cuando el modo de debug se encuentra activo y existen dos o más reglas cuya condición es satisfecha pero sus valores de estado o los tiempos de demora que deben ser asignados a la celda son distintos, entonces se producirá un error, abortando la simulación y evitando la ejecución de modelos no deterministas. En la Figura 15 se muestra un conjunto de reglas que, usadas junto a la variante del juego de la vida previamente descripto, provoca esta situación. Cuando la celda que esté evaluando la función de cómputo local tiene el valor 1, la primera y la segunda reglas son válidas. Sin embargo, la primer regla indica que el estado de la celda debe ser el valor 2, mientras que la segunda indica que debe ser el valor 1. El error generado en este caso se muestra en la Figura 16.

```
[new-life-rule]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 0 or (0,0) = 1 }
rule : 0 100 { (0,0) = 2 }
```

Figura 15 – Reglas para la variante del Juego de la Vida para la cual puede existir más de una regla que sea evaluada a Verdadero

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
----------------------------------------------------------
...
Starting simulation. Stop at time: 00:00:05:000

Exception Mexception thrown!
Description: Two rules evaluate to TRUE and the result is different!
Thrown in:
        File: synnode.cpp – Method: evaluate – Line: 90
        File: ltranadm.cpp – Method: evaluate - Line: 104
Description:
        Language used is: life-rule
        The state of the Neighbours is:
        +-------------------------------------------+
        |      1.00000       0.00000       0.00000 |
        |      1.00000       0.00000       1.00000 |
        |      1.00000       0.00000       0.00000 |
        +-------------------------------------------+
Aborting simulation...
```

Figura 16 – Error generado por la herramienta cuando existen dos o más reglas válidas y se encuentra activo el modo de debug

El uso de funciones para la generación de números aleatorios en la definición de la condición de alguna regla puede producir que su evaluación dé como resultado el valor *Falso*, pudiendo provocar que en algunos casos todas las reglas sean evaluadas a *Falso*, mientras que en otros casos exista una o más que sean válidas. Como se mencionó previamente, si todas las reglas son evaluadas a *Falso* se producirá un error y se interrumpirá la simulación, pero por tratarse de un modelo estocástico se procede de la siguiente manera:

- si hay más de una regla que es evaluada a *Verdadero* en un modelo estocástico y se encuentra activo el modo de debug, se considera sólo la primera regla válida y se informa al usuario de tal situación, pero la simulación no es interrumpida. Por ejemplo, la Figura 17 muestra un conjunto de reglas que, en combinación con la variante del Juego de la Vida previamente descripto, produce este tipo de error. Cuando el valor de la celda que está ejecutando la función de cómputo local es 1 y el número aleatorio generado es menor a 0.8, la primera y la segunda reglas serán Verdaderas, pero los valores que deben ser asignados a la celda son distintos. Para evitar esta ambigüedad se considera la primera de las reglas válidas. El mensaje generado se muestra en la Figura 18.

```
[new-life-rule]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 1 and random < .8 }
rule : 0 100 { (0,0) = 2 or (0,0) = 0 }
```

Figura 17 – Reglas para la variante del Juego de la Vida para la cual dos reglas pueden evaluar a Verdadero en un modelo Estocástico

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
```

```
---------------------------------------------------------
...
Starting simulation. Stop at time: 00:00:05:000
...
WARNING. In the stochastic model, two or more rules evaluate to TRUE.
...
Simulation ended!
```

Figura 18 – Alerta generada por la herramienta cuando existen dos o más reglas válidas en un modelo estocástico y se encuentra activo el modo de debug

- si todas las evaluaciones de las reglas son *Falsas*, se asigna el valor **?** a la celda, y se utiliza el tiempo de demora establecido por defecto en la definición del modelo, informando tal situación al modelador y prosiguiendo con la simulación. Por ejemplo, en la Figura 19 se muestra un conjunto de reglas que producen este tipo de error. Cuando la celda tenga el valor 1 y la función random genere un valor mayor o igual a 0.8, todas las reglas son evaluadas a Falso. El mensaje de error generado se muestra en la Figura 20,

```
[new-life-rule]
rule : 1 100 { (0,0) = 1 and random < .8 }
rule : 0 100 { (0,0) = 2 }
rule : 2 100 { (0,0) = 0 }
```

Figura 19 – Reglas para la variante del Juego de la Vida para la cual ninguna regla podría resultar verdadera en un modelo Estocástico

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
---------------------------------------------------------
...
Warning! - None of the rules evaluate to True in the Stochastic Model.
...
Simulation ended!
```

Figura 20 – Alerta generada por la herramienta cuando no existen reglas válidas en un modelo estocástico

8.2 Redefinición del Comportamiento ante el arribo de un Mensaje

En *CD++*, al crearse una celda se asociaban automáticamente tres puertos: *In*, *Out* y *NeighborChange*, como se muestra en la Figura 21. El puerto de entrada *In* conecta un modelo DEVS externo al espacio de celdas con la celda en cuestión. El puerto de entrada *NeighborChange* define los valores de entrada que llegan desde los vecinos de la celda. Por último, el puerto de salida *Out* conecta a la celda con sus vecinos y posiblemente con otros modelos DEVS.

Cuando un mensaje externo llega a la celda a través del puerto *In*, su valor se encola, y posteriormente se convertirá en el nuevo valor de estado de la celda.

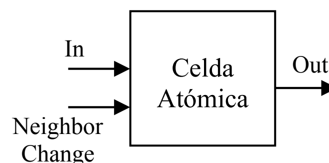


Figura 21 – Estructura de una Celda Atómica en *CD++*

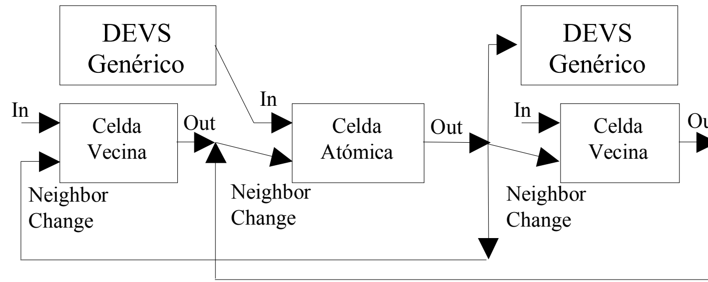


Figura 22 – Acoplamiento de una Celda en *CD++*

En *N–CD++* esta implementación se ha modificado para reflejar fielmente la especificación de los modelos *Cell-DEVS* [Wai98]. Por un lado, los puertos de entrada *In* sólo se crean para aquellas celdas que estén conectadas con un modelo DEVS que no sea su vecino.

Como segunda medida, se modificó la forma de actuar ante la llegada de un mensaje externo por un puerto de entrada. La función de cómputo local τ puede usar todas las entradas disponibles para el cálculo del nuevo valor de la celda, incluyendo los valores enviados por las celdas del vecindario, y los valores de los mensajes externos ingresados por otros puertos de entrada.

Para ello, se extendió el lenguaje de forma tal que permita definir el comportamiento de una celda, indicando qué acción tomar ante la llegada de un mensaje por un puerto de entrada que no provenga de un vecino. Esto implicó, además, la posibilidad de crear distintos puertos de entrada de tipo *In* para una misma celda, ya que es posible que el modelador quiera definir distintos comportamientos de acuerdo al puerto por el cual arribó el mensaje.

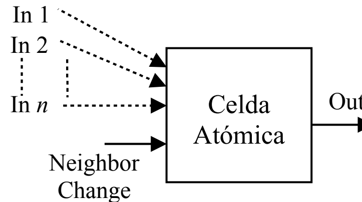


Figura 23 – Extensión realizada sobre un Celda Atómica en *N–CD++* para los Puertos de Entrada

En la Figura 24 se muestra básicamente la forma de describir el comportamiento de la celda ante la llegada de un mensaje por un puerto de tipo *In*. Mediante la directiva *Link* se crea un acoplamiento entre los puertos *portOut* y *portIn*, este último perteneciente a la celda (*x*, *y*) del modelo celular *CellModel*.

Mediante la directiva *portInTransition* se especifica el nombre de la función que será usada para establecer el valor de la celda cuando arribe un mensaje por el puerto especificado.

El lenguaje provee la función *portValue(p)* que permite obtener el valor del último mensaje ingresado por el puerto de entrada *p* de la celda que esta evaluando. Si en el momento de evaluar la función *portValue* aun no arribó un mensaje por el puerto *p* desde el inicio de la simulación, se obtendrá el valor indefinido (**?**). Una vez que arribó un mensaje, al consultarse se obtendrá el último valor ingresado.

Es posible el uso del string "*thisPort"* como parámetro de *portValue*, indicándole al simulador que el valor del puerto que se quiere usar es el puerto por el cual llegó el mensaje, y que produjo la llamada a la función en cuestión.

```
...     ...    ...    ...    ...    ...
Link : portOut  portIn@CellModel (x,y)
...     ...    ...    ...    ...    ...
portInFunction : portIn@CellModel (x,y)     portFunction
...     ...    ...    ...    ...    ...
[portFunction]
rule :. Valor Demora { Condición }
```

```
...      ...    ...    ...    ...    ...
rule :  Valor Demora { Condición }
else :  DefaultFunction

[DefaultFunction]
rule :  Valor Demora { Condición }
...      ...    ...    ...    ...    ...
```

Figura 24 – Definición del comportamiento de la celda ante la llegada de un mensaje por un puerto *In*

Mediante la cláusula *Else* es posible definir el uso de una función que contenga un conjunto de reglas, en caso de que ninguna de las definidas sea verdadera. Esto posibilita agrupar reglas de uso común por varias funciones, evitando así la reescritura de las mismas.

La cláusula *Else* puede llamar a cualquier función que defina el comportamiento de una celda, incluso a otra función que contenga otra cláusula *Else*. Sin embargo, un mal uso de estas podría generar una referencia circular, que provocaría un ciclo, bloqueando al proceso de simulación.

8.3 Redefinición del Comportamiento de Salida

En *CD++*, el puerto de salida *Out* conecta a la celda con sus vecinos y posiblemente con otros modelos DEVS. Cuando el valor de la celda sufre un cambio, dicho valor es enviado a través del puerto *Out* a todos los modelos conectados a ella.

En *N-CD++*, este comportamiento se ha redefinido para cumplir con las definiciones formales descriptas en [Wai98]. La herramienta permite la definición de varios puertos de salida asociados a una celda atómica. Uno de ellos se llama *Out*, y conecta a la celda en cuestión con sus vecinos. Dicho puerto está siempre presente. Cabe destacar que el puerto *Out* también puede ser usado para conectar a otros modelos DEVS, y que estos recibirán un mensaje cada vez que se produzca un cambio en el valor de estado de la celda [RW99c]. El resto de los puertos de salida se crean dinámicamente solo para las celdas que enviarán datos a otros modelos DEVS y su nombre es definido por el usuario.

En la Figura 25 se muestra la estructura de una celda atómica en *N–CD++* considerando los cambios comentados en esta sección para los puertos de salida, y en la sección anterior para los puertos de entrada. Los puertos In_i y Out_i que se conectan a otros modelos DEVS son creados en forma dinámica, mientras que los puertos *NeighborChange* y *Out* son fijos y están presentes para todas las celdas.

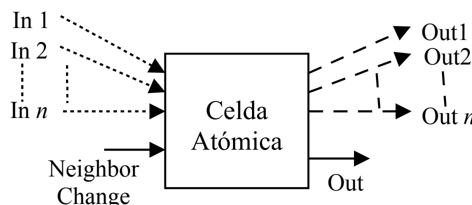


Figura 25 – Estructura de una Celda Atómica en *N–CD++*

Esta posibilidad de tener múltiples puertos de salida conectados con distintos modelos permite que la celda pueda enviar distintos valores a cada uno de ellos. Esto se puede realizar gracias a una nueva ampliación en el lenguaje que permite dar comportamiento a las celdas. Mediante la función **send** se puede especificar un valor a ser enviado por un puerto en cualquier momento. Por ejemplo, en la regla:

{ nuevo_valor + **send**(P, V) } demora { condición }

el comportamiento de la misma el siguiente: Si la *condición* evalúa a verdadero, entonces la celda es demorada de acuerdo al tipo de demora establecida, y el nuevo valor de la celda será el especificado. Además, se enviará el valor V por el puerto P. Dicho valor, puede ser una constante o una expresión compleja que es evaluada en tiempo de ejecución.

La función *send* tiene la siguiente aridad:

send(string, real) → 0

Esto significa que la función siempre devuelve el valor 0. El objetivo es que la función *send* pueda enviar un valor sin que cambie el estado de la celda, como se muestra a continuación:

{ (0,0) + *send*(port1, 15 * log(10)) } 100 { (0,0) > 10 }

En este caso, si el valor actual de la celda es mayor a 10, la celda mantiene su valor y, además, envía el valor resultante de evaluar la expresión 15*log(10) a través del puerto de salida *port1*, demorando la salida durante 100 milisegundos.

Debido a que **send** es una función más del lenguaje, puede ser usada en cualquier lugar donde sea posible, como por ejemplo en la definición de una *condición*. Pero esto no es deseable debido a que una condición puede evaluarse y resultar ser inválida, y por lo tanto se ejecutará el comando **send** enviando un valor por un puerto en forma indiscriminada. En cambio las expresiones que representan el nuevo valor de la celda o la que define el valor de la demora son evaluadas solo cuando la condición es válida.

8.4 Soporte para Modelos n–Dimensionales

Los problemas reales que implican el uso de autómatas celulares generalmente deben representarse usando modelos en dos o tres dimensiones. A esto puede sumársele una serie de problemas teóricos donde es posible el uso de cuatro o más dimensiones para poder dar una respuesta a los mismos. CD++ sólo permite definir autómatas celulares bidimensionales y lineales, aunque estos últimos sean un caso particular de los primeros, ya que se definen mediante la utilización de un autómata bidimensional especificando una dimensión (*d*, 1), donde *d* es el número de celdas. Esto restringe los modelos a ser implementados, limitando así el uso de la herramienta en problemas más generales. Ante esta situación se decidió la expansión de la herramienta para posibilitar la creación de autómatas celulares n-dimensionales [RW99b]. En este caso, la especificación formal definida en [Wai98] fue nuevamente usada para especificar el comportamiento de los modelos.

Como fue visto en las secciones anteriores, cada modelo celular es especificado por medio de un lenguaje que permite definir su dimensión, vecindario y valores iniciales para sus celdas. Dicho lenguaje fue adaptado para incluir referencias a celdas en espacios n–dimensionales.

Según la definición de autómatas celulares, estos pueden ser o no toroidales. Este concepto sigue siendo válido incluso con las extensiones realizadas: si un autómata celular tiene dimensión $(d_1, d_2, ..., d_n)$, entonces la referencia a una celda $(x_1, x_2, ..., x_n)$ de un modelo celular toroidal se corresponde con la celda $(x_1 \bmod d_1, x_2 \bmod d_2, ..., x_n \bmod d_n)$.

La definición de cada modelo celular se realiza mediante un lenguaje de especificación que permite definir la dimensión del mismo, el valor inicial de cada celda y el vecindario utilizado. Además, permite la definición del comportamiento de cada celda utilizando reglas definidas según la gramática mostrada en el *Apéndice*.

En *CD++* el vecindario de una celda puede estar compuesto sólo por celdas del espacio celular adyacentes a la celda en cuestión. De esta forma, y debido a que se utiliza un espacio celular de dos dimensiones, el vecindario de una celda puede tener como máximo 9 elementos (considerando a la celda central una integrante del vecindario). *N-CD++* permite la definición del vecindario de una celda donde los elementos que lo componen no necesariamente deben encontrarse adyacentes a la celda en cuestión. Además, este vecindario puede tener igual o menor dimensión que la usada por el autómata celular.

Por otra parte, *CD++* permite la definición de *zonas* dentro de un espacio celular. Una *zona* es un conjunto de celdas contenidas en el área determinada por el rango $\{celda_1..celda_2\}$, como se muestra en la Figura 26. Cada zona es asociada a un conjunto de reglas diferente. De esta forma, distintas zonas dentro de un mismo modelo celular pueden tener distinto comportamiento. Este concepto se ha mantenido en *N-CD++*, permitiendo definir rangos cuya dimensión puede ser igual e incluso menor a la dimensión del autómata. En este caso, una zona definida por el rango $\{(x_1, x_2, ..., x_n)...(y_1, y_2, ..., y_n)\}$ es el conjunto de celdas $(t_1, t_2, ..., t_n)$ del autómata celular, tales que:

$$\min(x_i, y_i) \le t_i \le \max(x_i, y_i),\ \forall\ i \in [1, n].$$

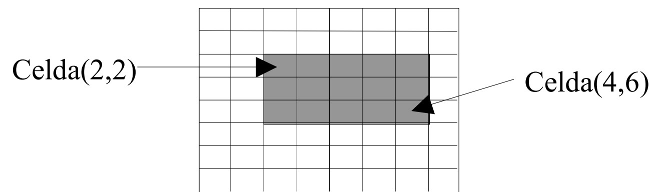


Figura 26 – Definición de la zona { (2,2)..(4,6) }

9 Análisis de Resultados

El objetivo de esta sección es mostrar los resultados obtenidos al comparar los tiempos requeridos por la herramienta para completar la simulación de algunos modelos, con diferentes parámetros y dimensiones.

9.1 Juego de la Vida

El primer caso de estudio es el *Juego de la Vida* [Gar70], que fue descripto en la sección 8. Las reglas que describen el comportamiento del modelo se muestran en la Figura 27. Las mismas son distintas a las mostradas en la Figura 6, sin embargo su evaluación producirá el mismo resultado. El motivo del cambio se debió porque se quiso poner a prueba el simulador activando el modo de debug sobre las reglas. Este modo valida que solo una regla pueda ser satisfecha. Con las reglas de la Figura 6, cuando la primera o la segunda son válidas, la herramienta produce un error cuando el modo de debug se encuentra activo, debido a que la tercer regla es siempre válida.

```
rule : 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) }
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { not ((0,0) = 1 and (truecount = 3 or truecount = 4)) and
               not ((0,0) = 0 and truecount = 3) }
```

Figura 27 – Reglas utilizadas para el estudio del *Juego de la Vida*

Se utilizó un modelo celular de 40x40 celdas, donde su estado inicial fue generado en forma aleatoria, conteniendo un 75 % de las celdas activas. Todas las pruebas fueron efectuadas en una computadora PC con procesador Intel Pentium de 166 MHz, 16 Mb de memoria RAM y sistema operativo Linux. Se realizaron simulaciones exhaustivas con diversas configuraciones de la herramienta, utilizando modelos celulares jerárquicos o achatados, generando o no un archivo de log, y manteniendo activo o no el modo de debug para las reglas. Los resultados obtenidos, variando la duración de la simulación, se muestran en la Figura 28.

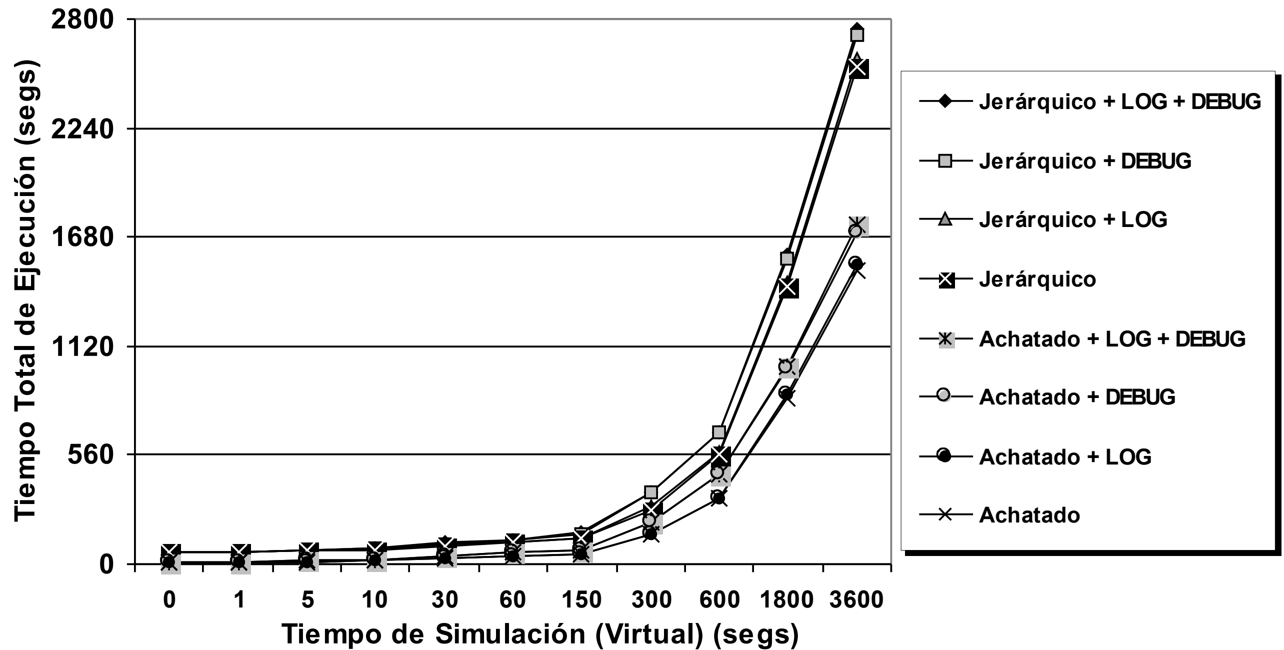
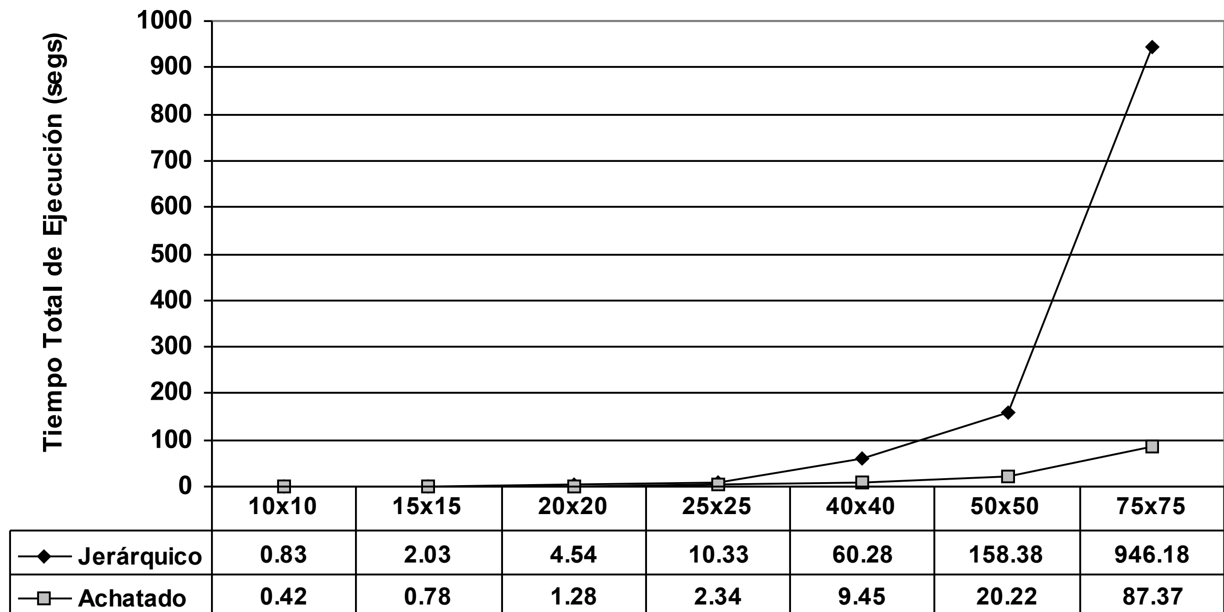


Figura 28 – Promedio de los tiempos de ejecución para el *Juego de la Vida* de 40x40 celdas

Puede verse que los modelos achatados requieren menor tiempo de ejecución que los modelos jerárquicos. Esto se debe a que las celdas virtuales que componen a los modelos achatados no generan mensajes que interactúan dentro del mismo, reduciendo el overhead. Además, cuando se encuentra activo el modo de debug para las reglas, el tiempo de ejecución es entre un 10 % hasta un 20 % mayor a los tiempos cuando el modo se encuentra desactivado. Esto se debe a que la activación del modo produce la evaluación de todas las reglas, con el fin de validar que solo una sea satisfecha, mientras que cuando no se encuentra activo alcanza con evaluar las reglas hasta encontrar la primera válida. Por otra parte, no se aprecian grandes diferencias de tiempo al generar el archivo de log, con respecto a cuando el mismo no es generado. Esto causó gran inquietud, por lo que se procedió a analizar con mayor profundidad este caso. Se realizaron ejecuciones con un modelo celular análogo al utilizado, durante 10 minutos de tiempo de simulación. Con los modelos jerárquicos y generando el archivo de log la ejecución demoró 680.13 segundos en promedio, mientras que cuando los datos del log fueron redireccionandos al dispositivo NULL, la simulación demoró 674.09 segundos. De esto se concluye que el tiempo de creación del archivo de log fue de tan solo 6.04 segundos, mientras que el tamaño del mismo fue de 25 Mb en promedio. Se procedió a crear un programa que escriba 25 Mb de información a través de la salida estándar, donde se dispone de un ciclo y en cada iteración se escriben 100 bytes. El objetivo es comprobar cuanto tardaría en generarse el archivo de log fuera del proceso de simulación. La ejecución de este programa redireccionando su salida a un archivo demoró 7.05 segundos, lo que incluyó la ejecución del comando *sync* de Unix para volcar toda la información contenida en los buffers al disco. La ejecución del mismo programa pero redireccionando su salida al dispositivo NULL demoró 1.2 segundos. Por lo tanto hay una diferencia de 5.85 segundos y es razonable pensar que en la simulación con o sin la generación del archivo de log haya en promedio 6 segundos de diferencia.

En la Figura 29 se muestra el promedio de los tiempos de ejecución para el *Juego de la Vida* variando el tamaño de los modelos. Para todos los casos se ejecutaron 10 transiciones en la simulación, efectuadas una cada 100 milésimas de segundo de tiempo simulado, comenzando con un estado inicial generado aleatoriamente, con el 75 % de las celdas activas. En todos los casos los modelos achatados tienen tiempos de ejecución menores a los modelos jerárquicos. En el peor caso (para los modelos de 10x10 celdas) los modelos achatados son el doble de rápidos que los jerárquicos. En el mejor caso (modelos de 75x75 celdas) los achatados son casi 11 veces mejores que los jerárquicos.



	10x10	15x15	20x20	25x25	40x40	50x50	75x75
Jerárquico	0.83	2.03	4.54	10.33	60.28	158.38	946.18
Achatado	0.42	0.78	1.28	2.34	9.45	20.22	87.37

Figura 29 – Promedio de los tiempos de ejecución para la simulación de 10 transiciones del *Juego de la Vida*

La Figura 30 muestra los tiempos de inicialización para modelos del *Juego de la Vida* de distintos tamaños. Los modelos achatados requieren menor tiempo de inicialización que los modelos jerárquicos. Para los pequeños modelos (de 25 a 625 celdas) esto se da en una relación de 5 a 1, mientras que para los grandes modelos (de 1600 a 5625 celdas) la relación es de hasta 10 a 1.

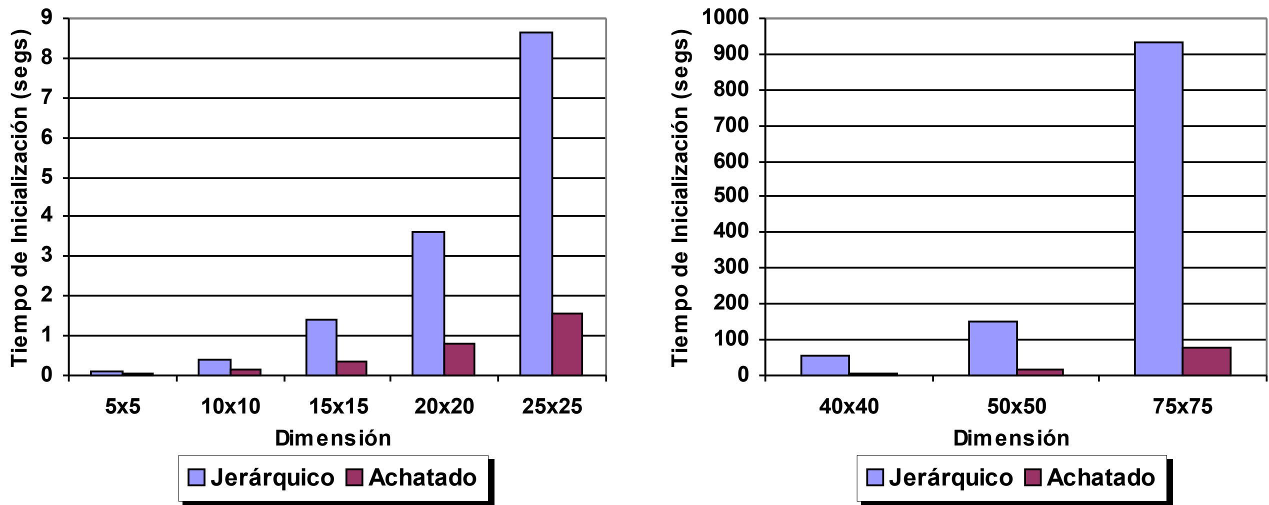


Figura 30 – Tiempos de Inicialización para el *Juego de la Vida*

En la Figura 31 se muestra el porcentaje de tiempo que requiere la inicialización con respecto al tiempo total de ejecución para 5 minutos de tiempo simulado. A medida que crece el tamaño del modelo, mayor es el porcentaje que se utiliza para inicialización, lo que implica que el tiempo de inicialización crece más rápido que el tiempo neto de ejecución (considerado como el tiempo total de ejecución menos el tiempo de inicialización). Para los modelos achatados ocurre lo mismo, pero los porcentajes son siempre menores a los valores de los respectivos modelos jerárquicos.

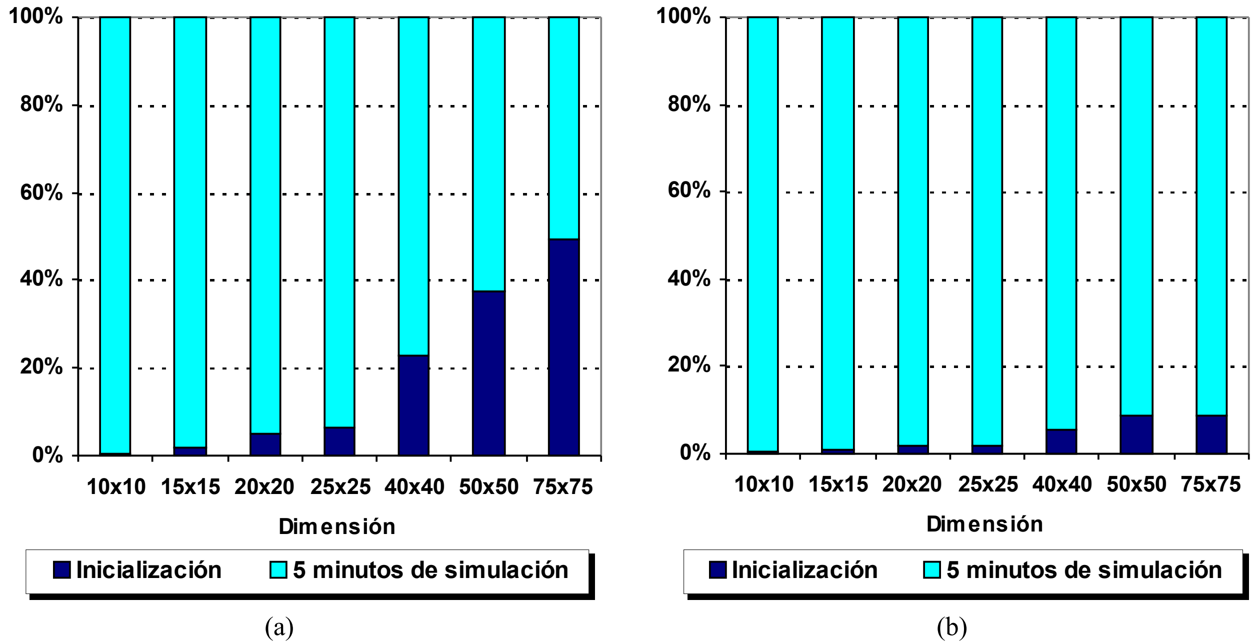


Figura 31 – Relación entre el tiempo de inicialización frente a 5 minutos de simulación para el *Juego de la Vida*. (a) Modelos Jerárquicos; (b) Modelos Achatados

Se repitieron las pruebas para un tiempo de simulación de una hora. Los resultados obtenidos se muestran en la Figura 32. Al igual que en el caso anterior, a medida que crece el tamaño del modelo, mayor es el porcentaje de tiempo requerido en la inicialización. Sin embargo, estos porcentajes son mucho menores a los mostrados en la Figura 31, siendo despreciables en la mayor parte de los casos.

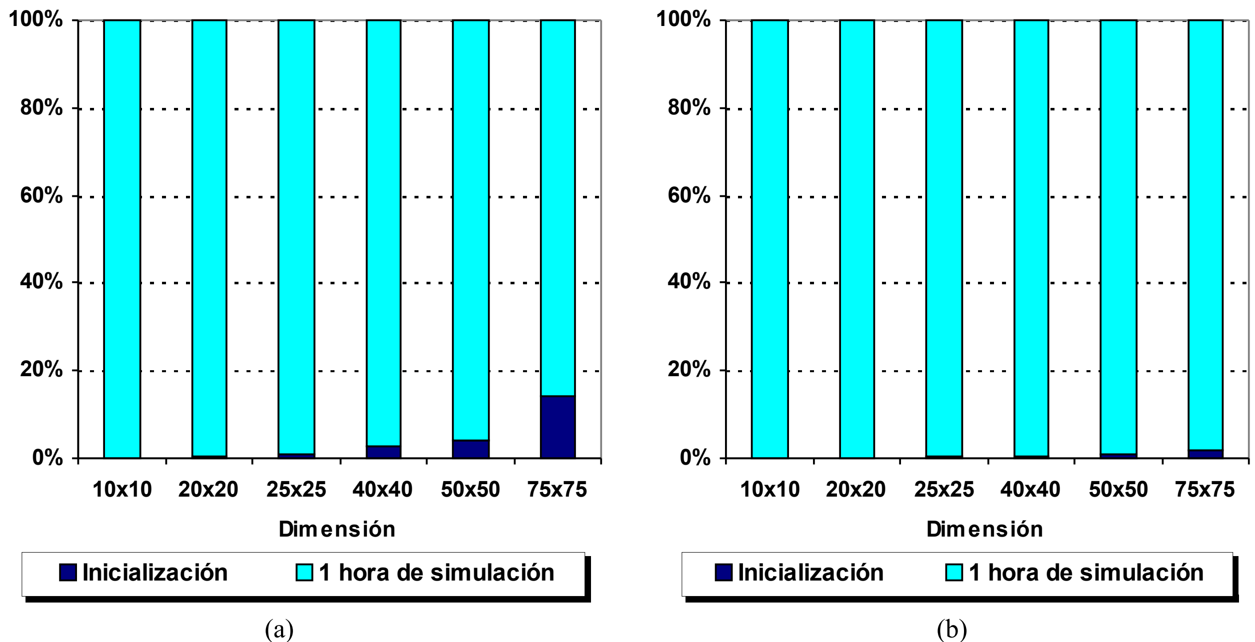


Figura 32 – Relación entre el tiempo de inicialización frente a 1 hora de simulación para el *Juego de la Vida*. (a) Modelos Jerárquicos; (b) Modelos Achatados

En la Figura 33 se muestran los porcentajes de tiempo de inicialización a medida que aumenta el tiempo simulado, para el *Juego de la Vida* con 40x40 celdas.

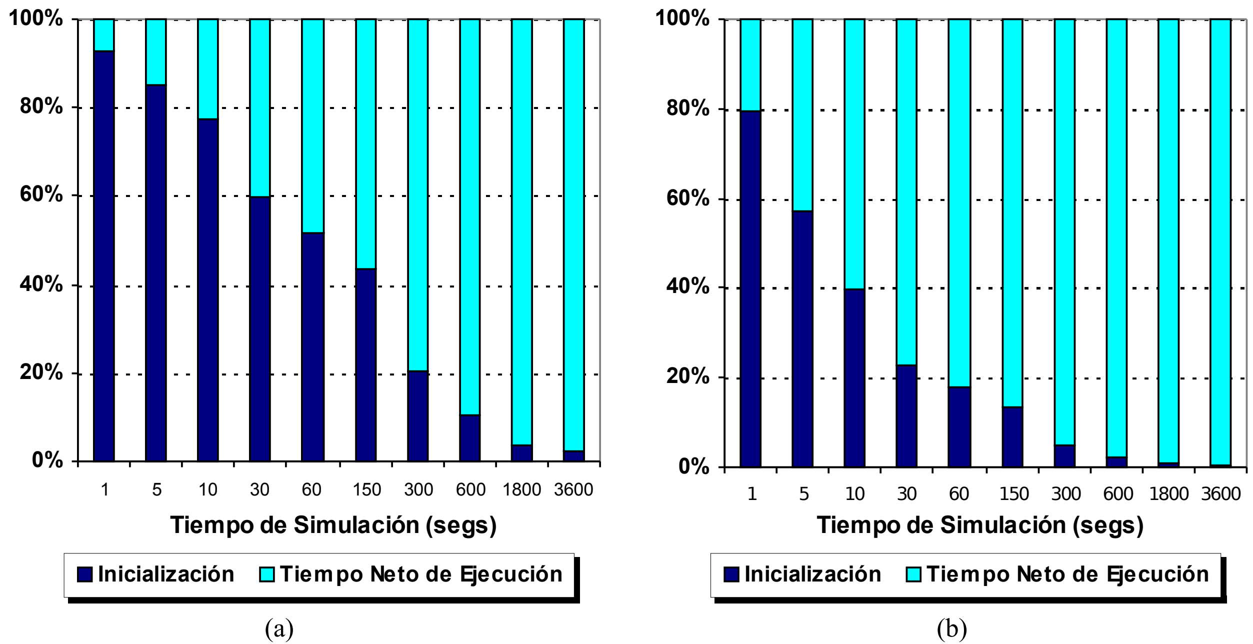


Figura 33 – Porcentaje del tiempo de inicialización a medida que aumenta el tiempo simulado para el *Juego de la Vida* con 40x40 celdas. (a) Modelos Jerárquicos; (b) Modelos Achatados

Para concluir con el análisis del *Juego de la Vida*, se realizó una comparación entre los tiempos de ejecución de los modelos bajo *CD++* y *N-CD++*. La comparación se realizó con modelos de 30x30 celdas, variando el tiempo de simulación. La Figura 34 muestra los resultados obtenidos. Aquí puede verse que los modelos jerárquicos de *N-CD++* requieren menos tiempo de ejecución e inicialización que los mismos modelos en *CD++*. Existen varios motivos para que suceda esto: el principal es que las estructuras de datos de *N-CD++* han sido mejoradas, reduciendo los tiempos de inicialización y ejecución. Para esto se optimizó el tiempo de creación de los objetos, de acceso y actualización a los datos, y fundamentalmente debido a la optimización de las operaciones booleanas (descriptas en la sección 8.1.3) que permite que *N-CD++* no siempre evalúe las reglas en forma completa, como lo hace *CD++*.

Por otra parte, para los modelos achatados *CD++* es mejor y supera ampliamente a *N-CD++* mientras mayor sea el tiempo de simulación. Esto se debe a que, por más que se hayan optimizado varias estructuras de datos y el mecanismo de evaluación de reglas, también se produce un overhead extra debido al tratamiento de valores reales y el soporte para modelos n-dimensionales que debe ser considerado en *N-CD++* por más que el modelo solo incluya valores 0 y 1, e implique el uso de un autómata de dos dimensiones. Los cambios introducidos para dar soporte a las nuevas características de *N-CD++* parecen haber sido compensados para los modelos jerárquicos con las optimizaciones realizadas, dejando un saldo favorable en el tiempo total de ejecución. Sin embargo, para los modelos achatados esto parece no ser así, y el manejo de las estructuras de datos internas de los mismos parece requerir mayor tiempo de ejecución, superando los tiempos ganados con las optimizaciones realizadas.

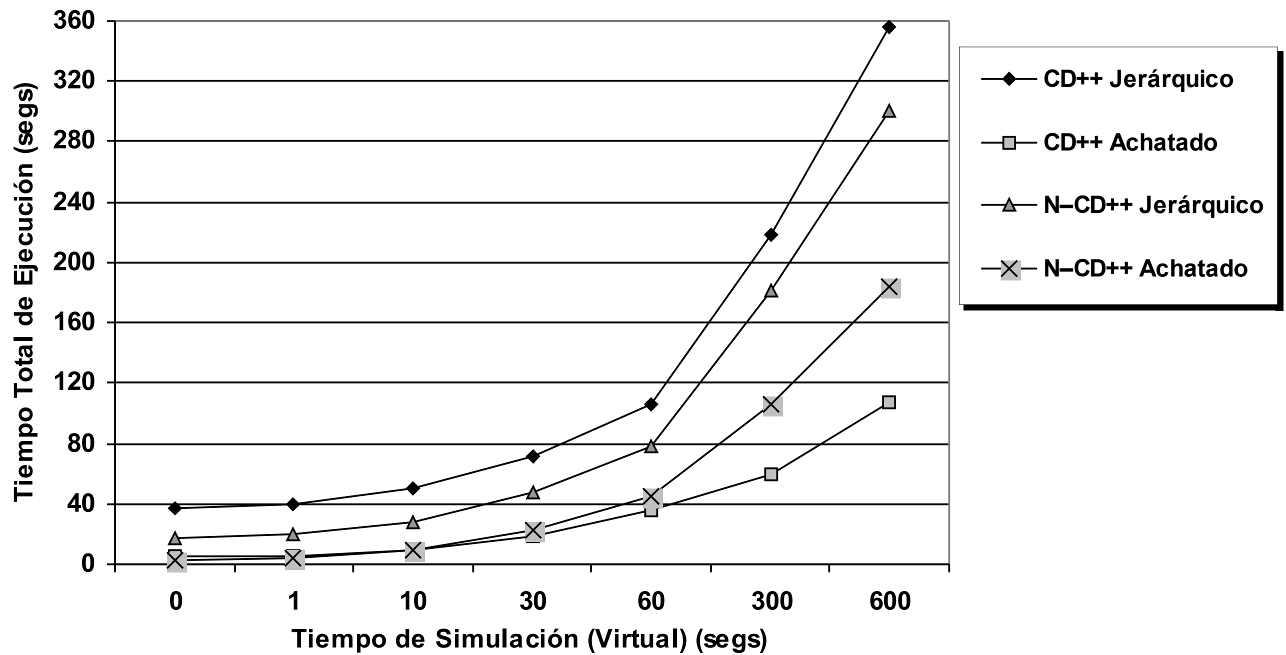


Figura 34 – Comparación entre CD++ y N-CD++ al evaluar el *Juego de la Vida* de 30x30 celdas

9.2 Juego de la Vida en 3D

El siguiente caso de estudio es una extensión del *Juego de la Vida* a tres dimensiones. Las reglas utilizadas por el modelo se muestran en la Figura 35, mientras que la definición del vecindario tiene la forma mostrada en la Figura 36.

```
rule : 1 100 { (0,0,0) = 1 and truecount >= 15 }
rule : 1 100 { (0,0,0) = 0 and truecount <= 13 and truecount > 6 }
rule : 0 100 { ((0,0,0) != 1 or truecount < 15) and ((0,0,0) != 0 or
               truecount > 13 or truecount <= 6) }
```

Figura 35 – Reglas utilizadas para el estudio del *Juego de la Vida* en 3D

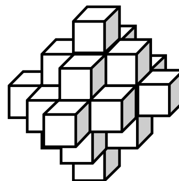


Figura 36 – Formato del Vecindario para el modelo del *Juego de la Vida* en 3D

Se utilizó un modelo celular de 10x10x16 celdas, donde inicialmente el 75 % de ellas contienen el valor 1, elegidas aleatoriamente, y el resto tiene el valor 0. Se realizaron simulaciones con diversas configuraciones de la herramienta, utilizando modelos celulares jerárquicos o achatados y manteniendo activo o no el modo de debug de las reglas. Los resultados obtenidos, variando la duración de la simulación, se muestran en la Figura 37. Los modelos jerárquicos requieren mayor cantidad de tiempo de ejecución que los respectivos modelos achatados. Además, cuando se encuentra activo el modo de debug para las reglas, la simulación requiere cerca de un tercio más de tiempo que cuando este modo se encuentra desactivado.

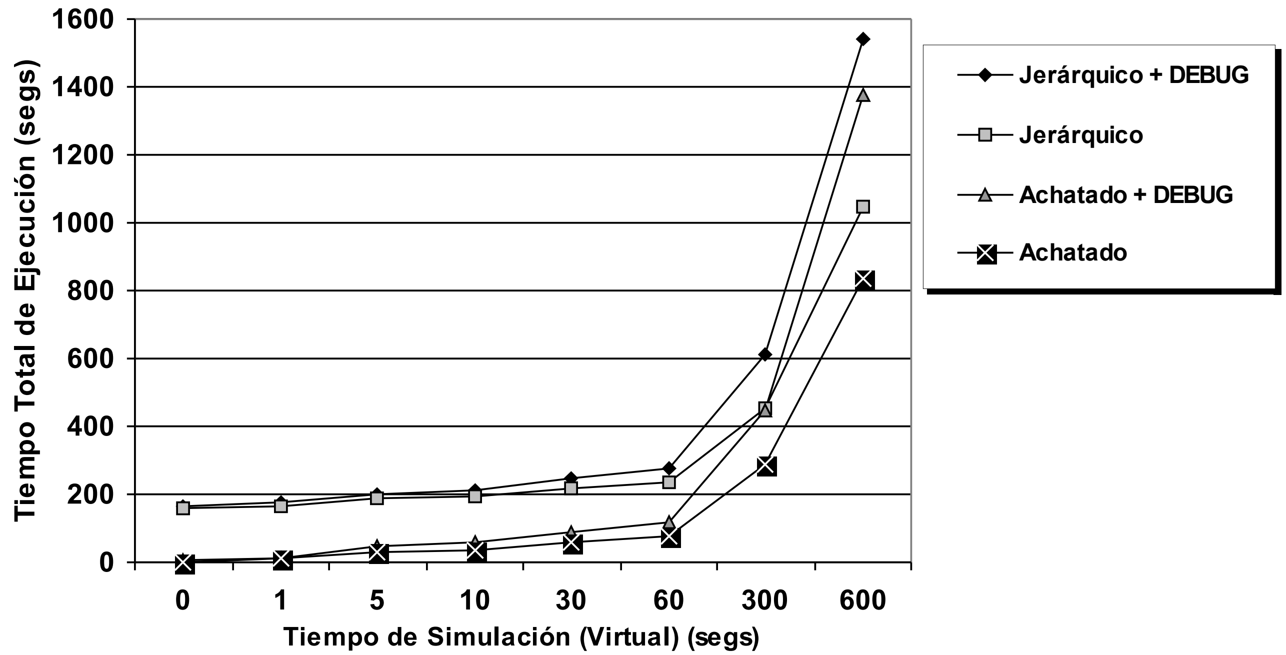


Figura 37 – Promedio de los tiempos de ejecución para el *Juego de la Vida* en 3D con 10x10x16 celdas

La Figura 38 muestra los tiempos de inicialización para modelos del *Juego de la Vida* en 3D, con distintos tamaños. Los modelos jerárquicos requieren mayor tiempo de inicialización que los modelos achatados, en orden exponencial. Para los pequeños modelos (de 27 a 125 celdas) esto se da en una relación que va desde 2 a 1 hasta 6 a 1, mientras que para los grandes modelos (de 1000 a 2000 celdas) la relación es de hasta 100 a 1. Nótese que cuando los modelos tienen igual cantidad de celdas, sus tiempos de inicialización no difieren demasiado.

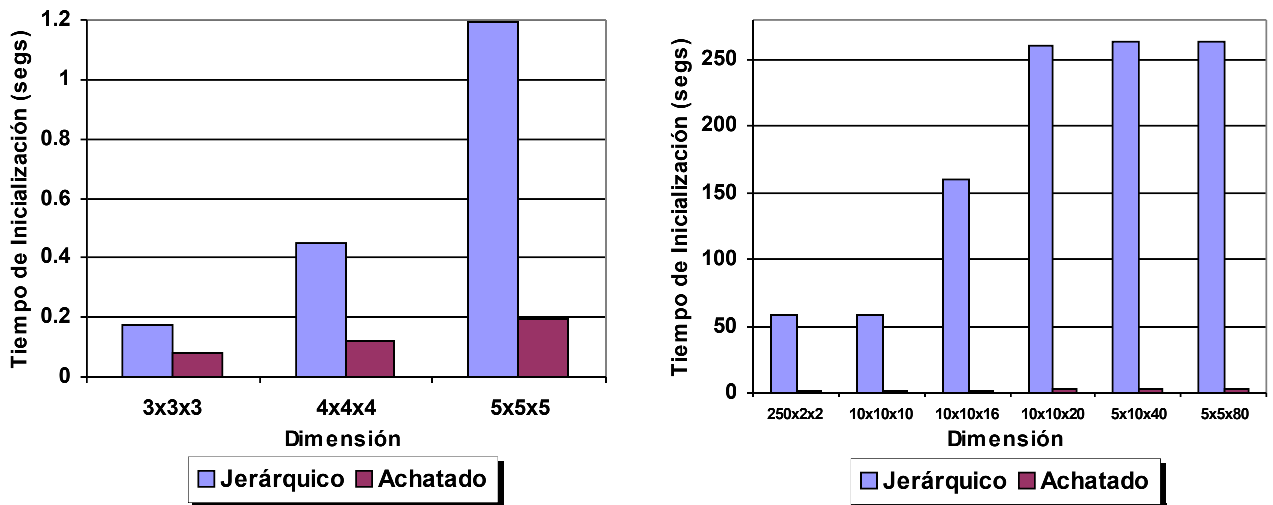


Figura 38 – Tiempos de Inicialización para el *Juego de la Vida* en 3D

En la Figura 39 se muestra el porcentaje de tiempo que requiere la inicialización con respecto al tiempo total de ejecución para simulaciones de 30 segundos.

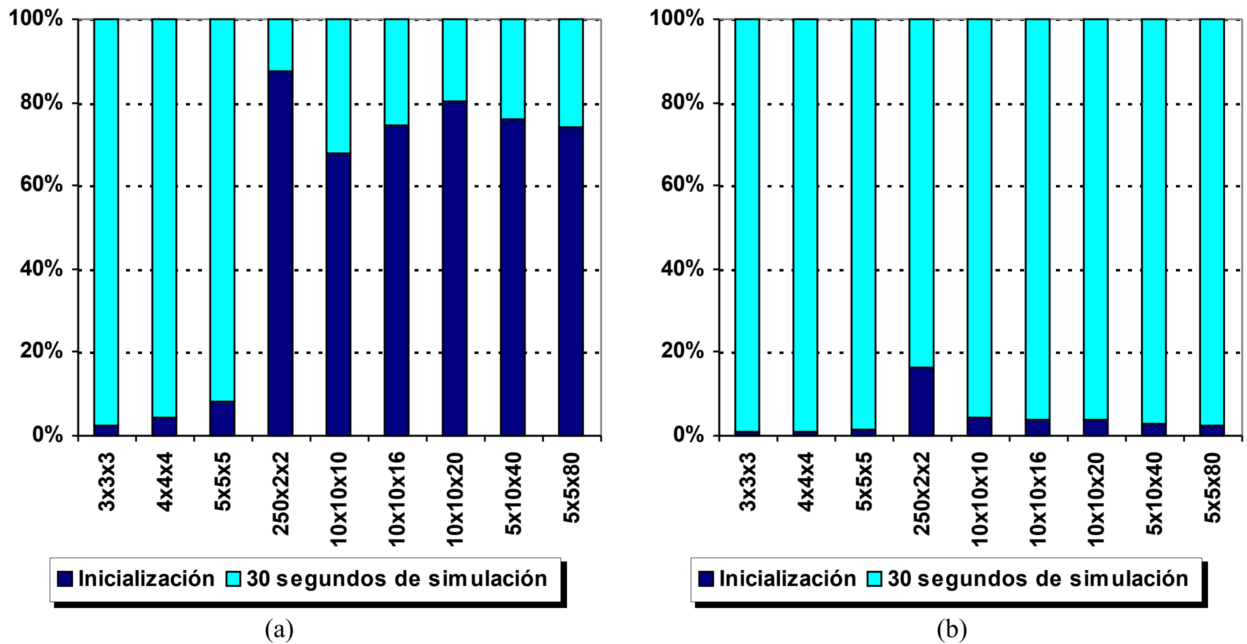


Figura 39 – Relación entre el tiempo de inicialización frente a 30 segundos de simulación para el *Juego de la Vida* en 3D. (a) Modelos Jerárquicos; (b) Modelos Achatados.

Para los pequeños modelos jerárquicos el tiempo de inicialización representa menos del 10 % del tiempo de ejecución de la simulación durante 30 segundos, pero para los grandes modelos jerárquicos la inicialización constituye entre un 70 a un 85 % de su tiempo total de ejecución. Esto no es así para los modelos achatados, donde los tiempos de inicialización no influyen en el tiempo total de simulación, salvo en el caso del modelo con tamaño 250x2x2. Este modelo tiene igual cantidad de celdas que otro de los modelos evaluados (con tamaño 10x10x10), sin embargo, su tiempo de inicialización tiene mayor influencia sobre el tiempo total de simulación. Esto se debe a que, como los modelos creados para su evaluación fueron todos toroidales, cuando se crean los puertos que vinculan a una celda con sus vecinos para este modelo en particular sucede que una misma celda cumple la función de "vecino múltiple". Para estos modelos en 3D se utilizó un vecindario de 27 elementos, compuesto por la celda origen y sus 26 vecinos adyacentes a la misma. Para el modelo de tamaño 250x2x2, si se considera a la celda (x, y, z), entonces sus vecinos $(x, y, z+1)$ y $(x, y, z-1)$ serán la misma celda, debido a que la tercera coordenada de tupla debe ser un valor comprendido entre 0 a 1, y por lo tanto:

$$z+1 \equiv z-1 \quad (2)$$

Lo mismo ocurre para otros pares de celdas, como $(x, y+1, z)$ y $(x, y-1, z)$; $(x, y+1, z-1)$ y $(x, y-1, z+1)$; y $(x, y+1, z+1)$ y $(x, y-1, z-1)$. Esto provoca, en combinación con las reglas definidas para el modelo, que el número de celdas activas disminuya rápidamente en los primeros pasos de simulación, y luego se mantenga hasta finalizar la ejecución. Debido a que el número de celdas activas es menor que en otros ejemplos de igual tamaño, se efectúan menos evaluaciones sobre las celdas, y el tiempo total de simulación es mucho menor, sin embargo, el tiempo de inicialización es semejante al de los otros modelos de igual tamaño. Esto provoca que el tiempo de inicialización tenga mayor influencia sobre el tiempo total de ejecución, que es menor al de otros modelos semejantes.

Para el caso del modelo jerárquico de 250x2x2 ocurre lo mismo que con en el modelo achatado, pero debido a que el tiempo de inicialización de todos los modelos jerárquicos es de mayor magnitud, esta diferencia no lo destaca del resto. Sin embargo, éste modelo es el que requiere mayor porcentaje de inicialización dentro de los modelos de su tipo.

En la Figura 40 se muestran el promedio de los tiempos de ejecución para modelos con distintos tamaños. Para todos los casos se ejecutó el modelo durante 30 segundos de tiempo simulado, comenzando con un estado inicial aleatorio con el 75 % de las celdas activas. En todos los casos los modelos achatados tienen tiempos de ejecución menores a los modelos jerárquicos. El modelo de tamaño 250x2x2, que tiene igual cantidad de celdas que el de 10x10x10, tiene un tiempo de ejecución mucho menor a éste, incluso, para el caso de los modelos achatados, menor al

modelo de tamaño 5x5x5. A pesar de que la simulación comienza con el 75 % de las celdas activas, el comportamiento para este modelo en particular, debido a la existencia de celdas que actúan como "vecinos múltiples", hace que la mayor parte de éstas muera en los primeros pasos de la simulación. Por lo tanto la ejecución prosigue hasta que la simulación alcanza los 30 segundos con pocas celdas activas.

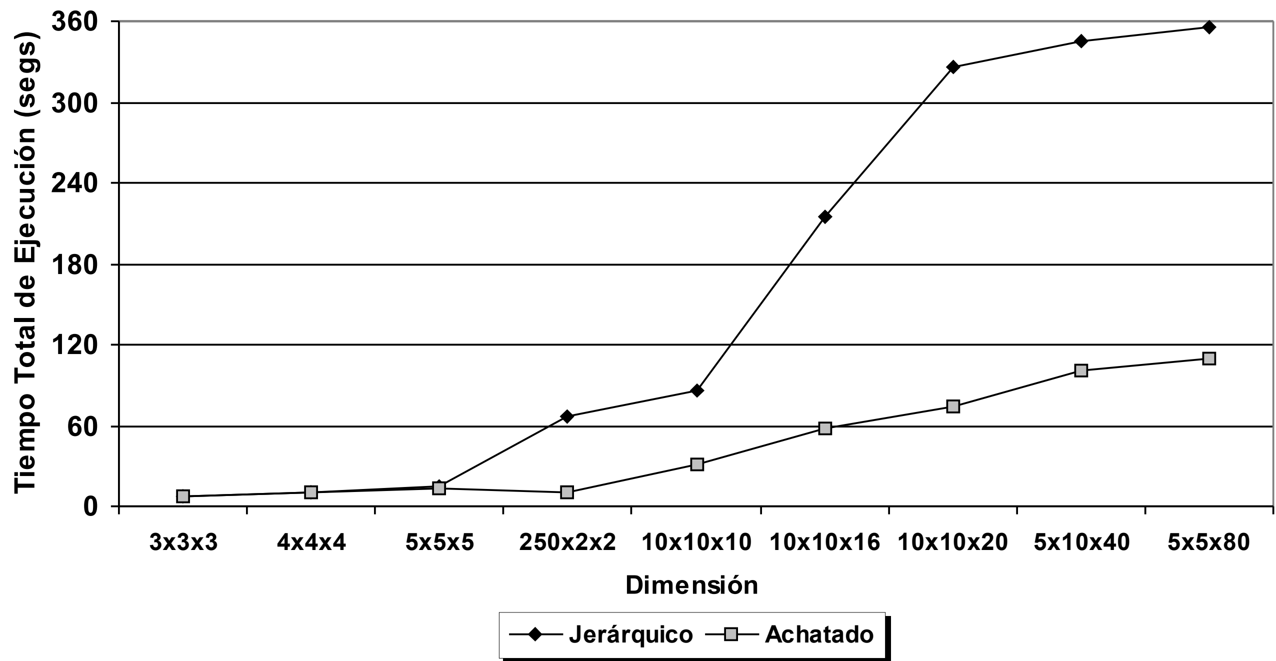


Figura 40 – Promedio de los tiempos de ejecución para el *Juego de la Vida* en 3D, en 30 segs de tiempo simulado

9.3 Pinball

Este caso consiste de un modelo celular de dos dimensiones, que representa un pinball. El modelo cuenta con una única pelotita, que se mueve por el tablero en ocho direcciones posibles, cambiando de dirección al chocar contra una pared. Los posibles valores de estado para las celdas son:

0: Indica que la celda se encuentra vacía.
1: Indica que la celda contiene a la pelotita, y que la misma se dirige en dirección noreste.
2: La celda contiene a la pelotita y se dirige en dirección norte.
3: La celda contiene a la pelotita y se dirige en dirección noroeste.
4: La celda contiene a la pelotita y se dirige en dirección oeste.
5: La celda contiene a la pelotita y se dirige en dirección sudoeste.
6: La celda contiene a la pelotita y se dirige en dirección sur.
7: La celda contiene a la pelotita y se dirige en dirección sudeste.
8: La celda contiene a la pelotita y se dirige en dirección este.
9: Indica que la celda contiene una pared.

Se utilizó un modelo celular de 20x20 celdas, donde inicialmente se ubica una pelotita en una posición aleatoria dentro del autómata, con dirección también aleatoria. Se establecen 50 celdas con valor 9, representando paredes, las cuales son elegidas al azar. Los resultados obtenidos variando la duración de la simulación se muestran en la Figura 41. Los modelos achatados requieren menor cantidad de tiempo para ejecutar la simulación, y a medida que mayor es el tiempo a simular, mayor es la diferencia en relación con respecto al tiempo de ejecución de los modelos jerárquicos. Sin embargo, esta diferencia no es tan destacada como en otros ejemplos, debido a la existencia de pocas celdas activas, dado que en cada instante de tiempo solo se encuentran activas todas las celdas del vecindario perteneciente a la celda donde se encuentra ubicada la pelotita, que esta compuesto por 9 elementos.

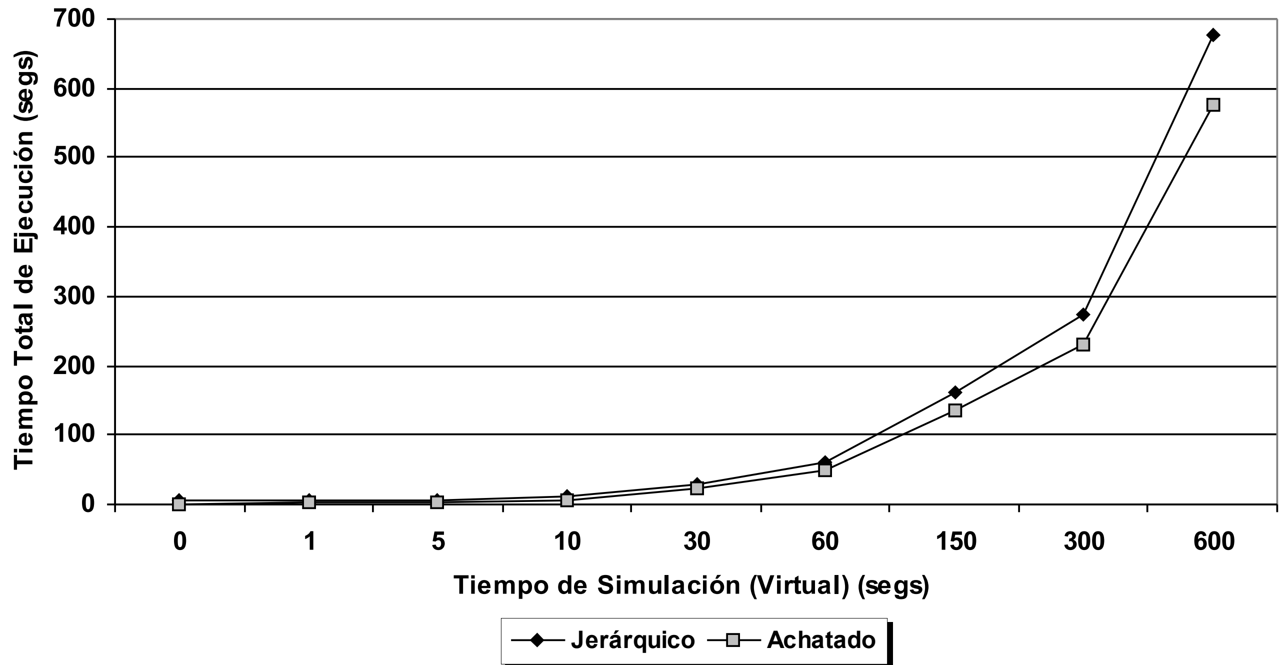


Figura 41 – Promedio de los tiempos de ejecución para el modelo del *Pinball*

9.4 Difusión del Calor sobre una Superficie

El modelo de difusión del calor consiste de un ambiente representado por un autómata celular, donde cada celda contiene una temperatura. En cada etapa de la simulación la temperatura de la celda se calcula como el promedio de los valores de la vecindad. El modelo a utilizar dispone de dos puertos de entrada conectados con el modelo celular. Estos permiten el ingreso de valores para el frío y el calor. Los datos para las temperaturas generadas son valores almacenados en un archivo e ingresados al sistema como eventos externos. Este archivo es generado aleatoriamente y es utilizado para la simulación de un modelo achatado y uno jerárquico, de forma tal que la simulación este controlada y que en ambos se produzcan las mismas acciones.

Los modelos utilizados en la simulación tienen un tamaño de 10x10 celdas e inicialmente todas las cedas tienen 24°C. La Figura 42 muestra los resultados obtenidos al ejecutar los modelos. Como era de esperar, los modelos achatados requieren menor tiempo de ejecución que los modelos jerárquicos.

En la Figura 43 se muestran los resultados de la evaluación del mismo modelo pero con un tamaño de 40x40 celdas. En este caso, salvo hasta los primeros 15 segundos, el modelo achatado requiere mayor tiempo de ejecución que los respectivos modelos jerárquicos. Esto se debe a que en el modelo existe gran número de celdas activas (las cuales se incrementan a medida que el tiempo avanza). En *N-CD++*, los modelos achatados utilizan una lista para almacenar los eventos ordenados por su tiempo de activación. El mantenimiento de esta lista, cuando hay gran cantidad de eventos pendientes, hace que sea crítico para el proceso de simulación. En futuras versiones de la herramienta se prevé una mejora sobre la estructura de esta lista con el objetivo de reducir los tiempos de ejecución.

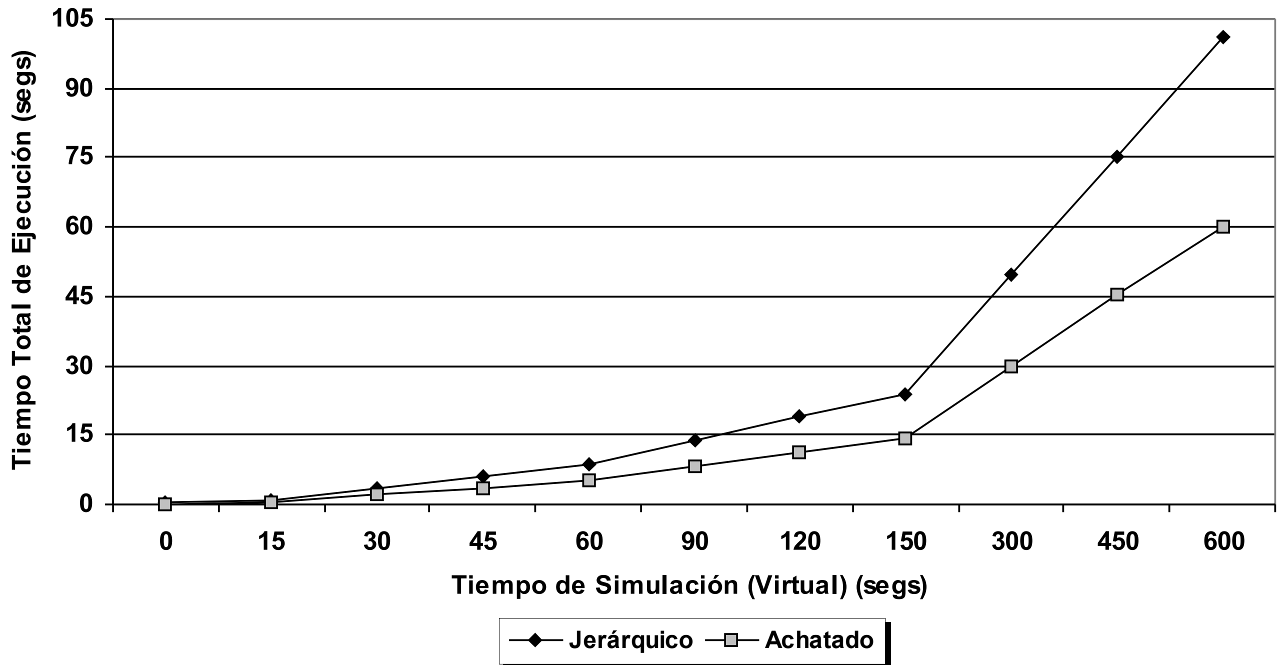


Figura 42 – Promedio de los tiempos de ejecución para el modelo de *Difusión del Calor* con 10x10 celdas

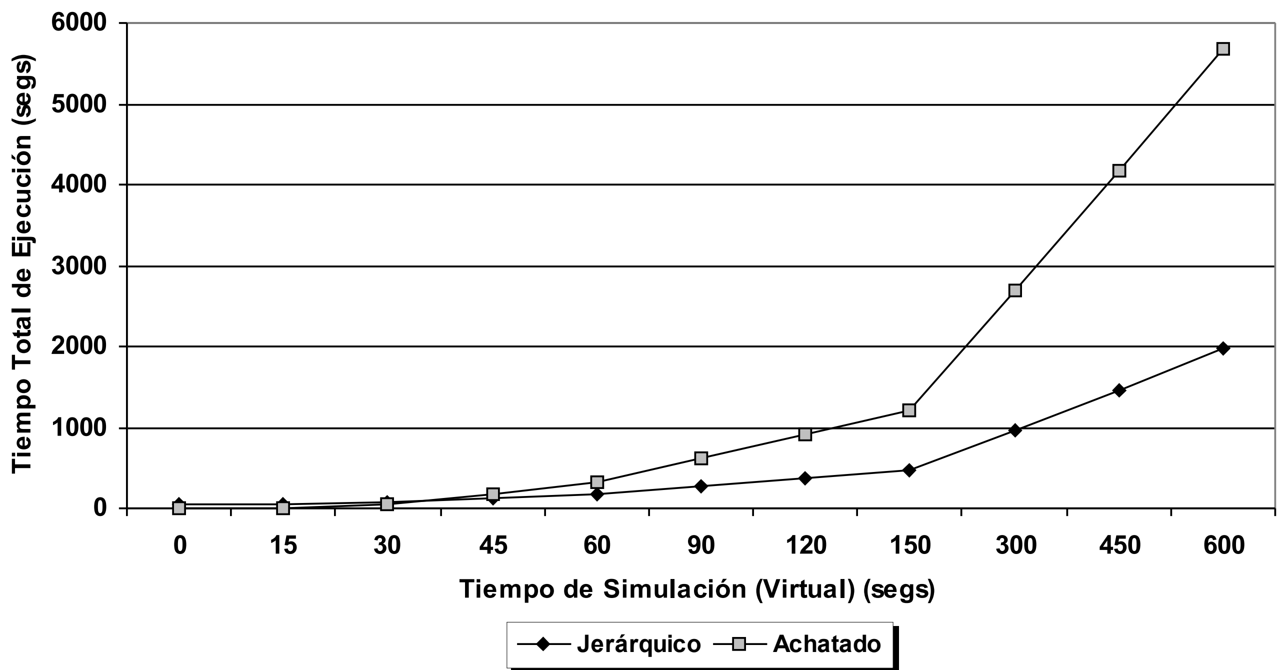


Figura 43 – Promedio de los tiempos de ejecución para el modelo de *Difusión del Calor* con 40x40 celdas

9.5 Difusión del Calor en 3D

Este caso es una extensión del modelo anterior a tres dimensiones. Los modelos simulados tienen un tamaño de 10x10x3 celdas, inicialmente todas las cedas tienen 24° C y, además, se utiliza un vecindario compuesto por una celda y sus 26 celdas adyacentes en tres dimensiones. Los resultados obtenidos al ejecutar los modelos se muestran en la Figura

44. En este caso, y por igual motivo que en el ejemplo anterior, los modelos achatados requieren mayor tiempo de ejecución que los modelos jerárquicos cuando el tiempo simulado supera los 30 segundos.

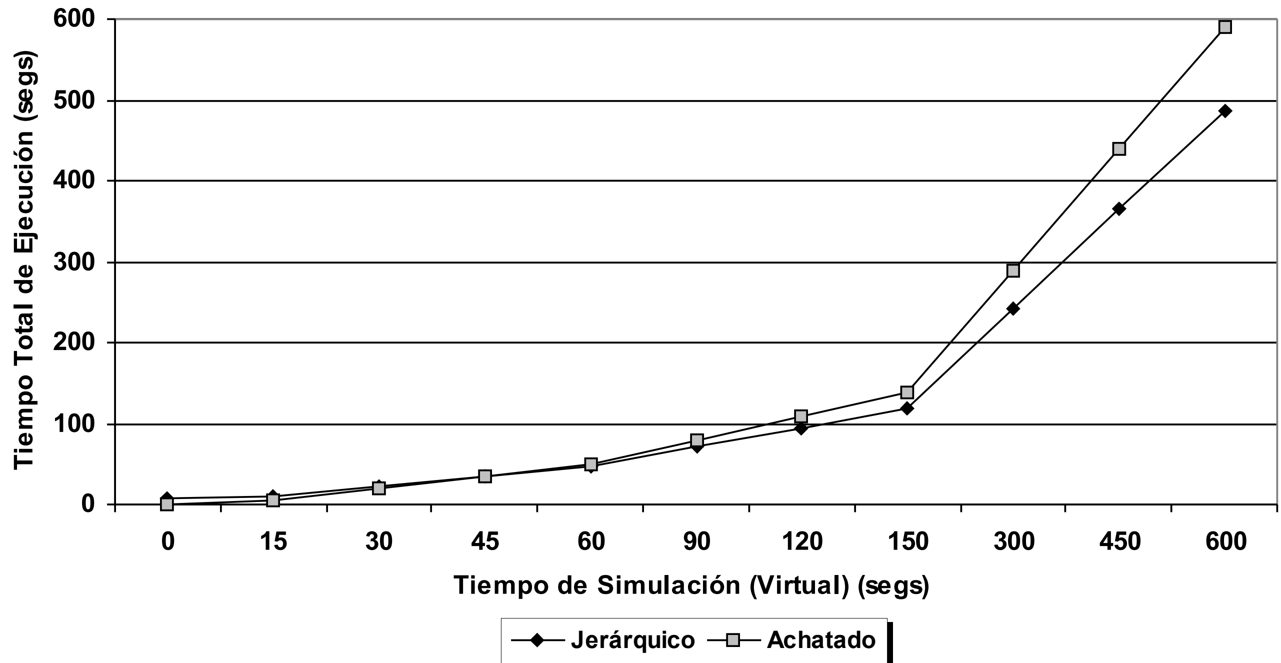


Figura 44 – Promedio de los tiempos de ejecución para el modelo de *Difusión del Calor* en 3D con 10x10x3 celdas

9.6 Juego de la Vida con uso extensivo de puertos de salida

Este ejemplo es usado para evaluar el tiempo de ejecución de un modelo, como el Juego de la Vida, que dispone de varios puertos de salida. Se supone que los modelos achatados son teóricamente mejores que los modelos jerárquicos, y en la práctica esto sucede salvo en el caso cuando hay un excesivo número de celdas activas. Sin embargo, los modelos achatados tienen una desventaja: al no intercambiarse mensajes dentro del modelo acoplado celular, los mismos no se detallan en el archivo de log, y por lo tanto estos modelos aportan menos información que los modelos jerárquicos. Cuando se ejecuta la herramienta con el modo de debug para los modelos celulares achatados, es posible redireccionar a pantalla o a un archivo los valores para las celdas del modelo en cada instante de tiempo, pero estos datos no quedan registrados en el archivo de log. El objetivo es evaluar que sucede cuando todas las celdas de un modelo achatado tienen puertos de salida que las conectan con la salida del modelo acoplado de mayor nivel, de tal forma que todos los cambios de estados producidos en la celda sean almacenados como eventos externos, y disponer así de información detallada sobre los cambios producidos. La idea es ver si los modelos achatados donde todas sus celdas tienen puertos de salida requieren menor tiempo de ejecución que los modelos jerárquicos.

Se creo un modelo celular bidimensional con las características previamente descriptas, de tamaño 10x40 celdas. La Figura 45 muestra el promedio de los tiempos de ejecución para los modelos achatados donde todas sus celdas se conectan a puertos de salida, comparado contra el mismo modelo sin puertos, y contra los modelos jerárquicos con y sin puertos de salida. En este caso, los modelos achatados son siempre requieren menor tiempo de ejecución que los modelos jerárquicos, independientemente de que si estos tienen o no puertos de salida. Esto implica que los modelos achatados con puertos de salida son mejores a los modelos jerárquicos, y por lo tanto ambos pueden generar información detallada del estado de las celdas, pero los modelos achatados con puertos de salida lo harían en menor tiempo.

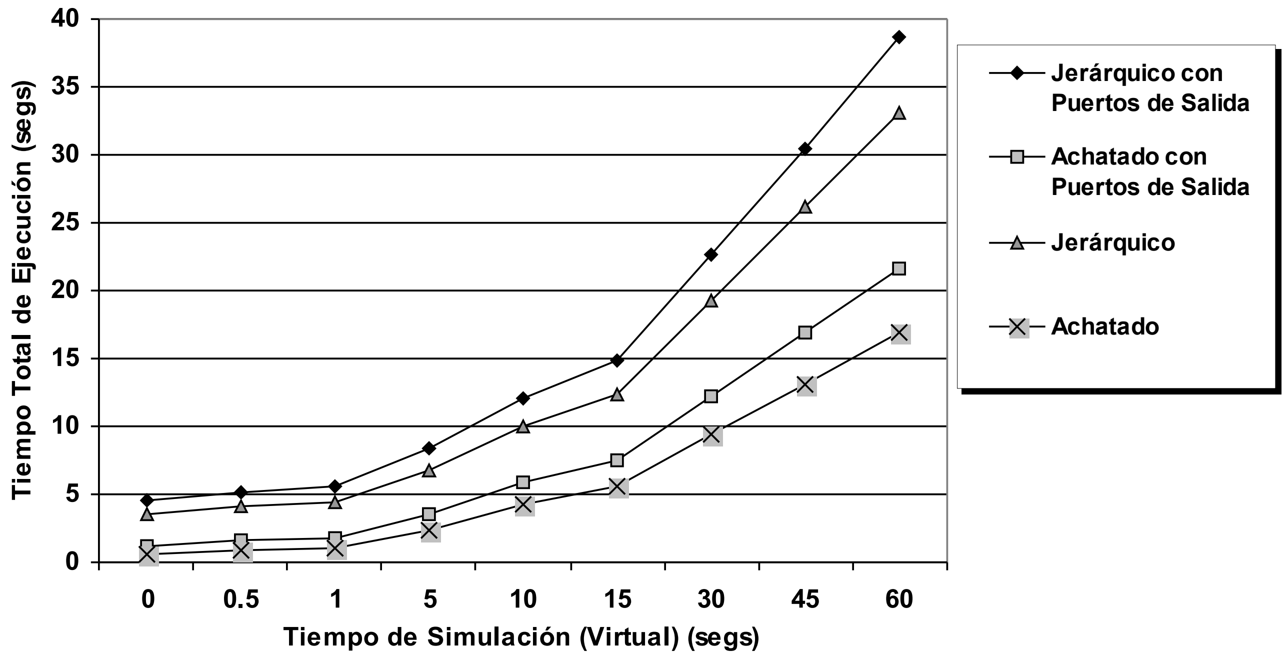


Figura 45 – Promedio de los tiempos de ejecución para el *Juego de la Vida* con puertos de salida

9.7 Inversión de Bits

El siguiente ejemplo consiste de un modelo celular bidimensional con un tamaño de 20x20 celdas, tal que las mismas tienen valores 0 y 1 distribuidos aleatoriamente. Las reglas definidas para el modelo realizan una inversión de bits, poniendo un 1 en la celda que tiene un 0 y viceversa. Claramente este modelo tiene siempre el 100 % de sus celdas activas y por lo tanto no es aconsejable la modelización a través de un autómata celular asincrónico. Sin embargo, el objetivo es verificar el comportamiento de la herramienta ante este tipo de situación, y compararlo con *CD++*. En la Figura 46 se muestran los resultados obtenidos.

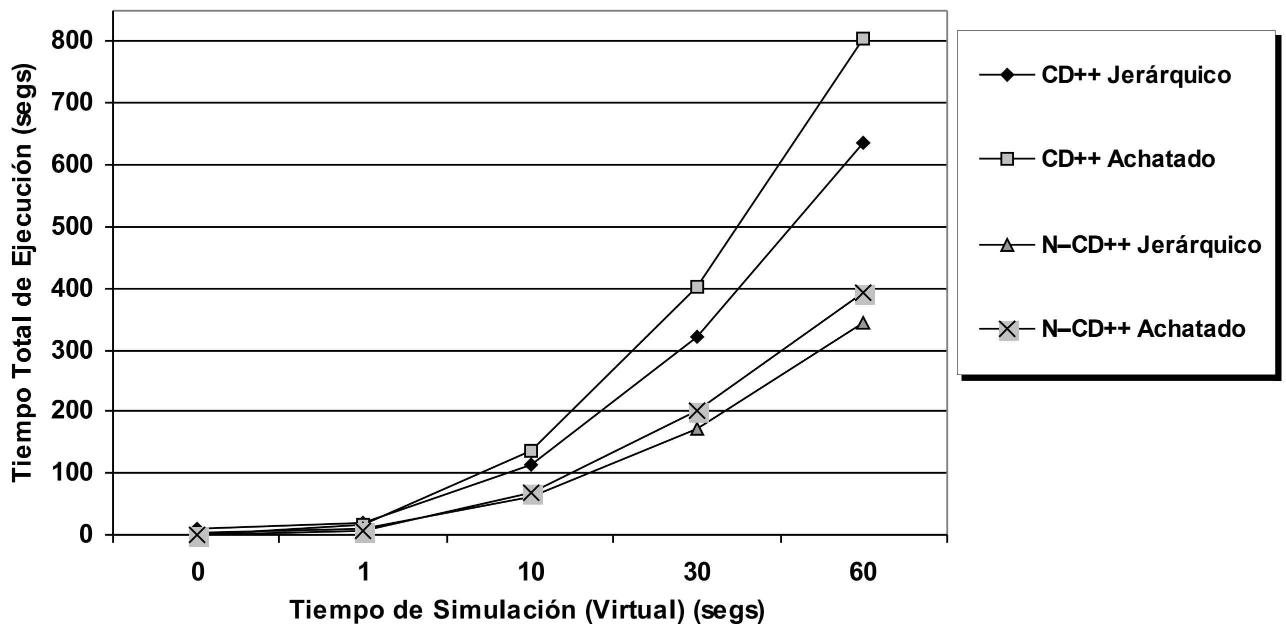


Figura 46 – Promedio de los tiempos de ejecución para el modelo de *Inversión de Bits*

Los tiempos de ejecución en *N-CD++* son inferiores en todos los casos a los de *CD++*. Sin embargo, los modelos achatados tienen mayor tiempo de ejecución que los modelos jerárquicos como era de esperar para los modelos con gran porcentaje de celdas activas. Esto sigue siendo válido incluso para los modelos achatados de *CD++*, donde el manejo de la lista de eventos pendientes se administra igual que en *N-CD++*.

9.8 Uso del Quantum para distintos modelos

N-CD++ permite especificar un valor de *quantum*, por medio de un parámetro en su invocación, con el objetivo de cuantificar el resultado obtenido por la función de cómputo local que se ejecuta en cada celda del modelo. De esta forma, todos los valores antes de ser asignados como nuevo estado de una celda son redondeados hacia el quantum inferior. Por ejemplo: si el quantum es 0.01 y el valor devuelto por la función de computo local es 0.2371, la celda obtendrá como nuevo estado el valor 0.23.

Como las celdas envían mensajes a sus vecinos solo cuando ocurre un cambio en su estado, mediante el uso de un quantum estos cambios serán menos frecuentes, y por lo tanto se reducirá el número de mensajes transmitidos en la simulación. Sin embargo, esta ventaja implica tener un error en los valores de estado de las celdas.

En [WZ99] se realizaron varios experimentos para analizar el comportamiento de los modelos Cell-DEVS quantizados. Los modelos analizados fueron los siguientes:

a) Modelo de difusión de calor sobre una superficie de 10x10 celdas, análogo al modelo descripto en la sección 9.4, pero los valores de las temperaturas, en vez de ser ingresados a través de eventos externos, son producidos por un generador de calor que esta conectado a una única celda del modelo. Se ejecutó el modelo durante 1 minuto de tiempo simulado, y se utilizó un vecindario compuesto de 5 celdas: la celda central y las 4 celdas adyacentes ubicadas arriba, abajo, a la derecha y a la izquierda de esta.
b) El mismo modelo que en a) pero el generador de calor está conectado a 4 celdas del modelo.
c) Modelo de difusión de calor en 3D, con generadores de frío y calor que se conectan a 4 celdas del modelo.
d) Variante del Juego de la Vida en 3D, con 10x10x10 celdas, y utilizando un vecindario con algunas celdas no adyacentes a la celda origen. Las reglas para este modelo fueron modificadas: los valores actuales cambian de acuerdo a los valores de los vecinos, promediándolos en algunos casos, y multiplicándolos en otros. El objetivo es la utilización de un vecindario complejo y valores reales para los estados de las celdas. El modelo fue ejecutado durante 20 segundos de tiempo simulado.
e) Extensión del modelo anterior a cuatro dimensiones. Se ejecutó el modelo durante 20 segundos de tiempo simulado.
f) Modelo buscador de calor. Este modelo tridimensional consiste de dos planos adyacentes. Uno de ellos representa una superficie, y representa el modelo de difusión del calor. El otro plano modela un dispositivo buscador de calor, que se mueve en búsqueda de la celda con máxima temperatura (en un máximo local). El modelo fue ejecutado durante 3 segundos de tiempo simulado.

El número de mensajes participantes durante la ejecución de un modelo Cell-DEVS puede expresarse como:

$$m_i = \sum_{j=1}^{i} n_j \cdot \mu \qquad (1)$$

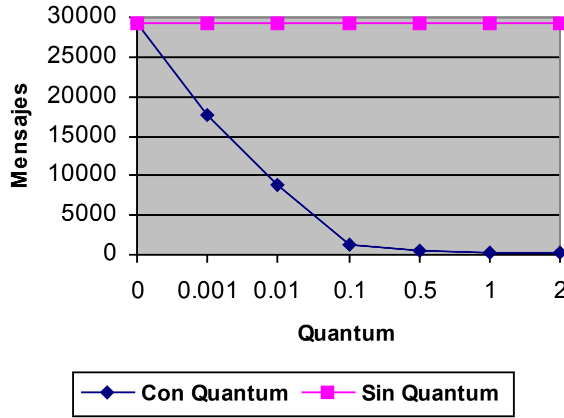
donde:

m_i: número de mensajes desde el comienzo hasta el i-ésimo paso de simulación;
n_j: número de celdas activas en el j-ésimo paso de simulación;
μ: cantidad de celdas del vecindario.

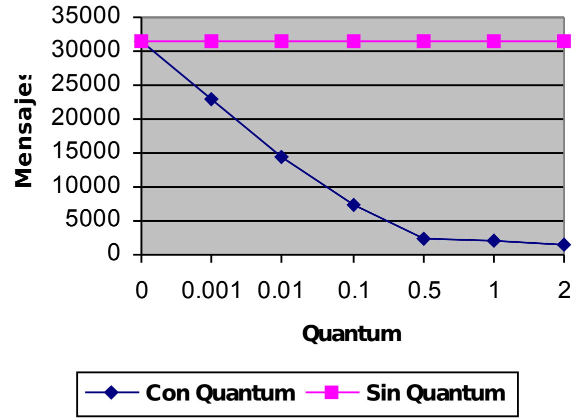
En la Figura 47 se muestra el número de mensajes involucrados en la ejecución de los distintos ejemplos. Puede verse que los resultados obtenidos muestran una reducción exponencial en el número de mensajes participantes de la simulación cuanto se utiliza un valor de quantum. Analizando la ecuación (1), puede verse que n_j se ha reducido (el tamaño del vecindario es fijo). Además, el uso de un valor de quantum hace que se generen menos pasos en la simulación, reduciendo el valor de *i* en la ecuación.

El comportamiento del modelo (b) es levemente peor que el de (a), debido a que el número de celdas inicialmente activas es mayor, por lo que en los primeros pasos de la simulación los valores de n_j es mayor a los respectivos valores del modelo (a).

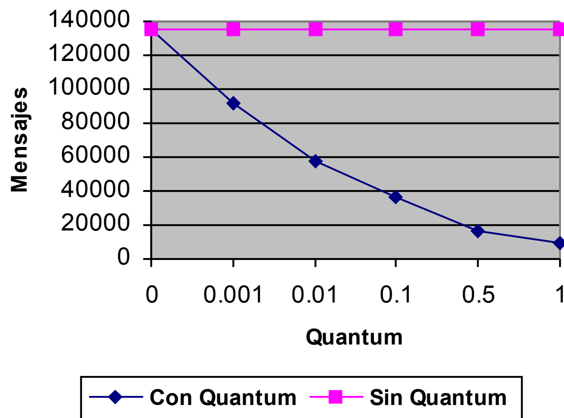
Para los casos (c), (d) y (e) el número de mensajes se ha incrementado significativamente debido al aumento en el tamaño del vecindario y de la cantidad de celdas participantes en la simulación. El comportamiento para el modelo (f) se debe a características propias del mismo. En este caso, el número de celdas activas entre 0 y 0.001 cambia entre un 1 a un 5 % debido a que los valores iniciales para las temperaturas tienen una precisión de 0.1, y por lo tanto el promedio de temperaturas variará en 0.01 ó 0.001 solo luego de varios pasos de ejecución, pero el tiempo de simulación fue de solo 3 segundos.



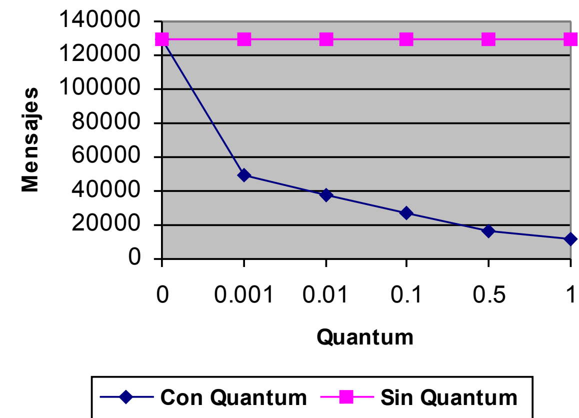
(a)



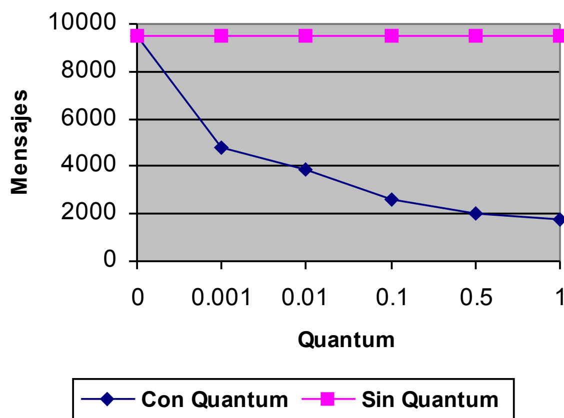
(b)



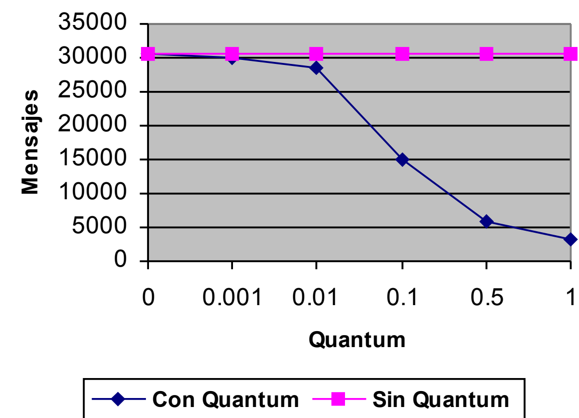
(c)



(d)



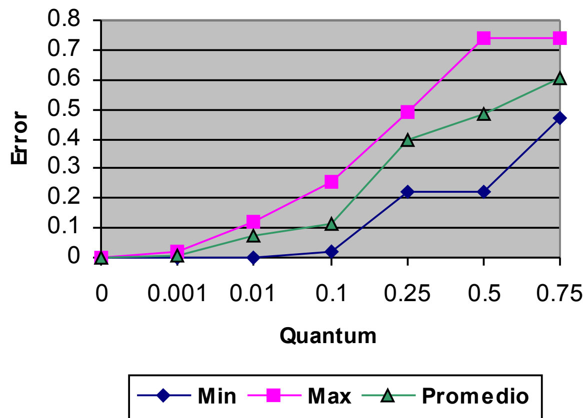
(e)



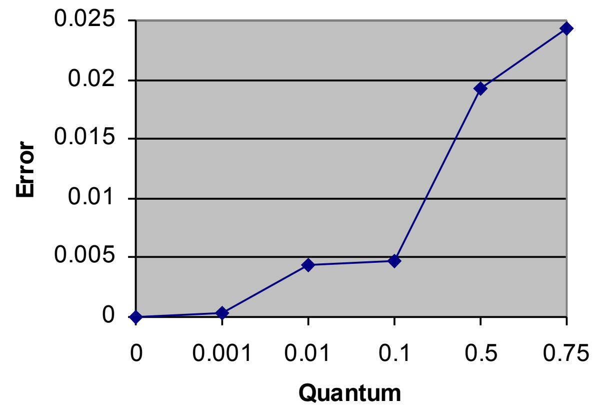
(f)

Figura 47 – Número de mensajes involucrados en la ejecución de los ejemplos [WZ99]

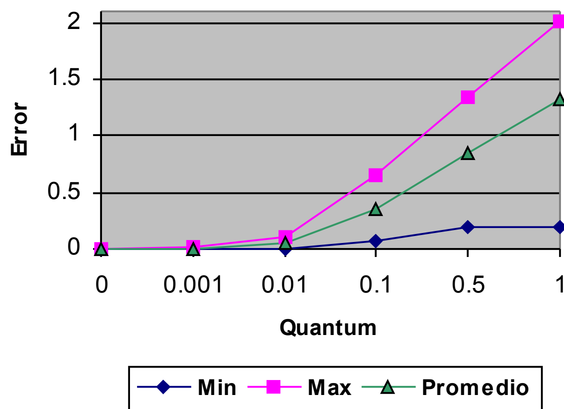
Mientras mayor sea el valor del quantum, menor será el número de mensajes intervinientes, reduciendo así el número de pasos de la simulación, pero mayor será el error en el estado de la celda. En la Figura 48 se muestra el error absoluto y relativo de algunas celdas observadas pertenecientes a algunos de los modelos previamente definidos.



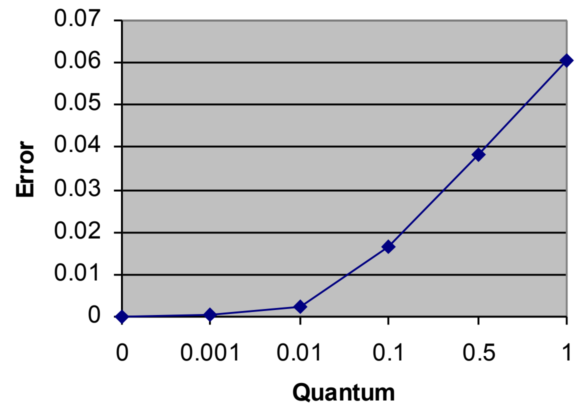
(a)



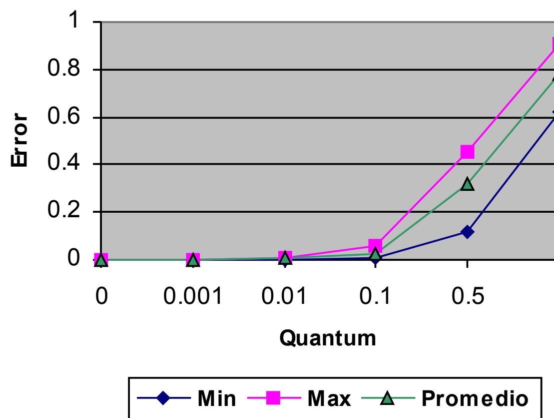
(b)



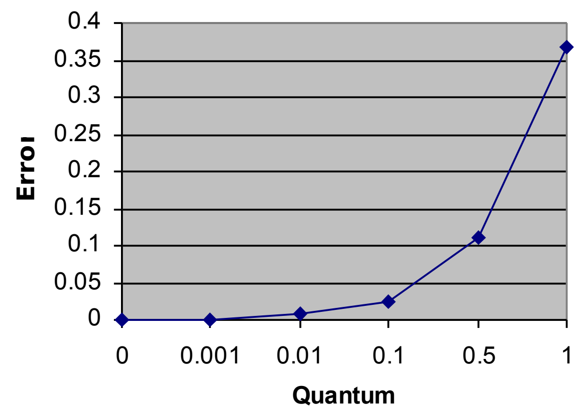
(c)



(d)



(e)



(f)

Figura 48 – Error de las celdas al usar quantum [WZ99]. ***Ejemplo b*****: (a) Error Absoluto, (b) Error Relativo (promedio);** ***Ejemplo c*****: (c) absoluto, (d) relativo;** ***Ejemplo d*****: (e) absoluto, (f) relativo.**

9.9 Mosquitos

El siguiente problema intenta modelar un ambiente de tres dimensiones. Para tal motivo se utiliza un autómata celular. El mismo posee celdas conectadas a un puerto por el cual ingresan datos que producirán valores para la presión atmosférica. La presión para el resto de las celdas se calcula como el promedio del vecindario. A partir del valor de la presión, es posible calcular la temperatura para la celda. En el ambiente hay mosquitos que vuelan desde una celda (x, y, z) a la celda (x, y, z+1). Si la temperatura de una celda es menor a 16 grados centígrados, el mosquito que ocupa la celda muere, de lo contrario sigue volando.

Para modelar un ambiente de tamaño 4x4x4 se usa un autómata celular de 4 dimensiones y tamaño (3, 4, 4, 4). Cada uno de los 3-hipercubos (a, x, y, z), con a = 1, 2, 3, permiten almacenar una variable distinta del modelo. La presión atmosférica de la posición (x, y, z) del ambiente es almacenada en la celda (0, x, y, z) y su temperatura se almacena en (1, x, y, x). La celda (2, x, y, z) es utilizada para indicar la presencia de un mosquito, pudiendo tener un valor 0 indicando que la celda (x, y, z) esta vacía, ó 1 indicando la existencia de un mosquito en la misma. Una celda puede estar ocupada por un solo mosquito.

Para dar comportamiento a cada 3-hipercubo, correspondientes a la temperatura y los mosquitos, se definen zonas, donde se asocia una función de cómputo local distinta para cada una de ellas. Para la zona que abarca las celdas que representan la temperatura se asocia una función que calcula el valor de la temperatura a partir de la presión; mientras que en la zona que modela a los mosquitos, se utiliza otra función para representar su desplazamiento, y hacer que estos mueran si la posición a la cual pertenecen en el modelo tiene una temperatura menor a 16 grados centígrados. La función de cómputo local utilizada por las celdas no incluidas en ninguna zona permite el cálculo de la presión atmosférica como el promedio de los valores de estado del vecindario.

Como puede verse, si bien la solución es válida, se complica la implementación del modelo. Sería de gran importancia, que además de que la cada celda permita almacenar un valor real, posibilite almacenar un arreglo de reales (o una estructura semejante), de tal forma que cada valor represente una variable para la celda. Esto evitaría la necesidad de utilizar más dimensiones que las necesarias, como ocurrió con este ejemplo.

10 Conclusiones

En este trabajo se presentó una extensión a la herramienta *CD++* utilizada para el modelado y simulación de modelos DEVS permitiendo, además, la creación de modelos celulares. Este formalismo posibilita la construcción de modelos jerárquicos basándose en modelos más simples y de menor tamaño, lo que facilita las etapas de desarrollo, chequeo y mantenimiento de los mismos, y posibilitando la reusabilidad de sus componentes.

Las extensiones desarrolladas a la herramienta permiten, mediante la utilización de un lenguaje sencillo y conciso, representar nuevos modelos los cuales, debido a las limitaciones de la versión anterior, eran imposibles de simular. Además, ofrece la posibilidad de usar funciones para la generación de números aleatorios con distintas distribuciones, lo que permite la creación de modelos estocásticos.

Las nuevas definiciones sobre puertos de Entrada/Salida permiten la múltiple interconexión entre modelos DEVS y Cell–DEVS. Los principales cambios fueron dirigidos para la inclusión de modelos n–dimensionales, convirtiendo a la herramienta en un ambiente para la definición y simulación de modelos genéricos más complejos.

Para concluir, cabe considerar que como la herramienta fue construida usando una versión estándar de C++ esto favoreció la portabilidad de la misma a distintas plataformas sin costo adicional. Hasta el momento *N-CD++* fue testeado en forma exitosa en Windows 95 / 98 / NT, Linux, AIX, HP-UX y Sun, lo que creó un incentivo en los usuarios a realizar simulaciones en su ámbito habitual para luego poder realizar estudios más profundos con máquinas con mayor poder de cómputo.

11 Trabajo Futuro

En la actualidad se están realizando diversos trabajos en relación con el presente, que serán resumidos a continuación:

- Para simular sistemas complejos tales como los espacios Cell-DEVS, suele ser necesario ejecutar un alto número de cálculos. Futuras implementaciones de la herramienta prevén el mapeo del mecanismo de simulación en un multiprocesador, con el objetivo de lograr paralelismo en la ejecución de modelos atómicos en forma asincrónica. Por otra parte, la ejecución lógica de los procesos asociados a cada procesador deberá ser sincronizada usando un mecanismo optimista o pesimista [WGR99].

- Extensión del formalismo Cell-DEVS considerando el formalismo R-DEVS [ChZ94], que ha sido definido para permitir ejecución paralela de los modelos. En este caso se propone una forma alternativa de especificar las prioridades para tratar eventos simultáneos. Aquí se tratará de clasificar las celdas inminentes en el espacio de celdas, usando la función de transición confluente δ_{con}.

- Soporte de modelos celulares hexagonales, lo que permitirá incluir nuevos modelos en la simulación, como por ejemplo los descriptos en [EDPWJ89] para el modelado de una colonia de hormigas, y en [GLNW94] para el modelado de una red de telefonía celular.

- Incorporación de autómatas celulares no homogéneos, permitiendo una definición de vecindario distinta para cada celda. Incluso, el vecindario podría cambiar durante la ejecución de la simulación.

- Creación de una interfaz gráfica, reemplazando la tarea efectuada por *DrawLog* que permite ver los resultados de la simulación de los modelos celulares en modo texto.

- Con respecto al lenguaje de definición de reglas, se prevé la inclusión de funciones para derivación e integración numérica, con el objetivo de la representación y simulación de modelos más complejos.

- Posibilidad de definir funciones o macros parametrizables, lo que reduciría el tamaño de las reglas, y permitiría una mejor reutilización del código.

- Nueva extensión del estado de la celda. En vez de que cada celda pueda almacenar un valor real, sería importante la posibilidad de que también pueda almacenar un arreglo de reales (o una estructura semejante), de tal forma que cada valor represente una variable para la celda, evitando la necesidad de utilizar más dimensiones que las necesarias, como ocurrió en el ejemplo de los mosquitos donde tres hipercubos almacenan los valores para los mosquitos, la temperatura y la presión de las celdas.

- Mejoras sobre las estructuras de datos de los modelos achatados, especialmente la que administra la lista de eventos pendientes, con el objetivo de reducir los tiempos de ejecución que hacen que dichos modelos sean peores que los respectivos modelos jerárquicos cuando la simulación implica gran cantidad de celdas activas.

Apéndice. Lenguaje de Especificación de N–CD++

La sintaxis del lenguaje usado por *N-CD++* para la especificación del comportamiento de los modelos celulares atómicos puede definirse con la siguiente BNF, donde las palabras escritas con letras minúsculas y en negrita representan terminales, mientras que las escritas en mayúsculas representan no terminales:

```
RULELIST        = RULE
                  |  RULE RULELIST

RULE            = RESULT RESULT { BOOLEXP }

RESULT          = CONSTANT
                  |  { REALEXP }

BOOLEXP         = BOOL
                  |  ( BOOLEXP )
                  |  REALRELEXP
                  |  not BOOLEXP
```

```
                  |  BOOLEXP OP_BOOL BOOLEXP

OP_BOOL          = and  |  or  |  xor  |  imp  |  eqv

REALRELEXP       = REALEXP OP_REL REALEXP
                  |  COND_REAL_FUNC(REALEXP)

REALEXP          = IDREF
                  |  ( REALEXP )
                  |  REALEXP OPER REALEXP

IDREF            = CELLREF
                  |  CONSTANT
                  |  FUNCTION
                  |  portValue(PORTNAME)
                  |  send(PORTNAME, REALEXP)
                  |  cellPos(REALEXP)

CONSTANT         = INT
                  |  REAL
                  |  CONSTFUNC
                  |  ?

FUNCTION         = UNARY_FUNC(REALEXP)
                  |  WITHOUT_PARAM_FUNC
                  |  BINARY_FUNC(REALEXP, REALEXP)
                  |  if(BOOLEXP, REALEXP, REALEXP)
                  |  ifu(BOOLEXP, REALEXP, REALEXP, REALEXP)

CELLREF          = (INT, INT RESTO_TUPLA

RESTO_TUPLA      = , INT RESTO_TUPLA
                  |  )

BOOL             = t  |  f  |  ?

OP_REL           = !=  |  =  |  >  |  <  |  >=  |  <=

OPER             = +  |  -  |  *  |  /

INT              = [SIGN] DIGIT {DIGIT}

REAL             = INT  |  [SIGN] {DIGIT}.DIGIT {DIGIT}

SIGN             = +  |  -

DIGIT            = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

PORTNAME         = thisPort  |  STRING

STRING           = LETTER {LETTER}

LETTER           = a | b | c |...| z | A | B | C |...| Z

CONSTFUNC        = pi | e | inf | grav | accel | light | planck | avogadro |
                   faraday | rydberg | euler_gamma | bohr_radius | boltzmann |
                   bohr_magneton | golden | catalan | amu | electron_charge |
                   ideal_gas | stefan_boltzmann | proton_mass | electron_mass |
                   neutron_mass  | pem

WITHOUT_PARAM_FUNC = truecount | falsecount | undefcount | time | random |
                     randomSign

UNARY_FUNC       = abs | acos | acosh | asin | asinh | atan | atanh | cos |
```

```
                  sec | sech | exp | cosh | fact | fractional | ln | log |
                  round | cotan | cosec | cosech | sign | sin | sinh |
                  statecount | sqrt | tan | tanh | trunc | truncUpper |
                  poisson | exponential | randInt | chi | asec | acotan |
                  asech | acosech | nextPrime | radToDeg | degToRad |
                  nth_prime | acotanh | CtoF | CtoK | KtoC | KtoF | FtoC |
                  FtoK

BINARY_FUNC    = comb | logn | max | min | power | remainder | root | beta |
                  gamma | lcm | gcd | normal | f | uniform | binomial |
                  rectToPolar_r | rectToPolar_angle | polarToRect_x | hip |
                  polarToRect_y

COND_REAL_FUNC = even | odd | isInt | isPrime | isUndefined
```

Figura 49 - Gramática usada para la definición de reglas bajo *N-CD++*

Implementación de modelos Cell–DEVS n–dimensionales

Informe Técnico

Tabla de Contenidos

Tabla de Contenidos....2
Tabla de Figuras....4
1 Introducción....5
2 Jerarquía de Modelos....5
2.1 Model....6
2.2 Atomic....7
2.3 Coupled....8
2.4 AtomicCell....8
2.5 TransportDelayCell....10
2.6 InertialDelayCell....11
2.7 CoupledCell....11
2.8 FlatCoupledCell....13
3 Jerarquía de Procesadores....14
3.1 Processor....15
3.2 Simulator....15
3.3 Coordinator....16
3.4 Root....17
3.5 CellCoordinator....18
3.6 FlatCoordinator....19
4 Administración de Modelos y Procesadores....19
4.1 ProcessorAdmin....19
4.2 ModelAdmin....20
5 Intercambio de Mensajes....21
5.1 Message....21
5.2 InitMessage....22
5.3 InternalMessage....22
5.4 ExternalMessage....22
5.5 DoneMessage....22
5.6 OutputMessage....23
5.7 MessageAdm....23
6 Jerarquía de Cargadores....23
6.1 SimLoader....24
6.2 StandAloneLoader....25
6.3 Network Loader....26
6.4 CommChannel....26
6.5 BSDChannel....26
7 Lenguaje de Especificación de Reglas para los Modelos Celulares....26
7.1 Real....26
7.2 TvalBool....28
7.3 SyntaxNode....30
7.4 SpecNode....31
7.5 RuleNode....32
7.6 VarNode....32
7.7 ConstantNode....33
7.8 PortDefNode....33
7.9 StringNode....34
7.10 SendPortNode....34
7.11 TimeNode....35
7.12 AbsCellPosNode....35
7.13 CountNode....35
7.14 OpNode....36
7.15 UnaryOpNode....36
7.16 BinaryOpNode....37

7.17 ThreeOpNode....38
7.18 FourOpNode....38
7.19 Nodos para la definición de Funciones y Operadores del Lenguaje....39
7.20 Parser....39
7.21 LocalTransAdmin....40
8 Soporte para modelos n–dimensionales....42
8.1 nTupla....43
8.2 CellState....46
8.3 NeighborhoodValue....47
9 Clases para el Inicio de la Simulación....48
9.1 Clase MainSimulator....48
9.2 Módulo Main....50
10 Clases Auxiliares....50
10.1 MacroExpansion....50
10.2 ClsEvalParam....51
10.3 Impresion....51
10.4 Zone....52
10.5 RealPrecision....52
10.6 MyList....52
10.7 Otras Clases....52
11 Simulación en N–CD++....53
12 Definición de modelos DEVS y Cell-DEVS en N-CD++....54
13 DrawLog....57
14 Ejemplos 57
14.1 Variante del Juego de la Vida....57
14.2 Juego de la Vida en 3D....58
14.3 Ordenamiento de Números Reales....60
14.4 Sistema de Trafico de Vehículos....62
14.5 Difusión del Calor sobre una Superficie....64
14.6 Difusión del Calor en 3D....66
14.7 Fluido de Gases....69
14.8 Uso de las extensiones para los puertos de Entrada y Salida....71
14.9 Sistema de Clasificación de Materias Primas....72
14.10 Pinball....76
14.11 Pista de Baile....78
14.12 Mosquitos....81
Apéndice A – Nodos para las Operaciones y Funciones del Lenguaje....84
Apéndice B – Funciones para Manipulación de Valores Reales....86

Tabla de Figuras

Figura 1 – Jerarquía de Modelos....6
Figura 2 – Jerarquía de Procesadores....14
Figura 3 – Definición recursiva para el orden de celdas....18
Figura 4 – Jerarquía de Mensajes....22
Figura 5 – Jerarquía de Cargadores....24
Figura 6 – Comportamiento de los operadores relacionales en la clase Real....28
Figura 7 – Comportamiento de los operadores para la lógica trivalente....30
Figura 8 – Jerarquía de Clases para los Nodos del Árbol que representa las Reglas....30
Figura 9 – Optimización en la evaluación de las operaciones binarias de la lógica trivalente....37
Figura 10 – Reglas para el Juego de la Vida....40
Figura 11 – Estructura de Arbol que Representa la Primer Regla del Juego de la Vida....41
Figura 12 – Recorrido sobre todas las celdas en CD++....45
Figura 13 – Recorrido sobre todas las celdas en N-CD++....46
Figura 14 – Ejemplo de acoplamiento de una celda atómica con sus vecinos....54
Figura 15 – Implementación de la Variante del Juego de la Vida....58
Figura 16 – Resultados obtenidos para la variante del Juego de la Vida....58
Figura 17 – Vecindario para el Juego de la Vida en 3D....59
Figura 18 – Implementación del Juego de la Vida en 3D....59
Figura 19 – Resumen de los resultados obtenidos durante la simulación del Juego de la Vida en 3D....60
Figura 20 – Esquema de Configuración Inicial para el Autómata de Ordenamiento Numérico....60
Figura 21 – Ejemplo de la configuración inicial del Autómata de Ordenamiento Numérico....60
Figura 22 – Ejemplo del Autómata de Ordenamiento Numérico luego del Primer Paso....61
Figura 23 – Ejemplo del Autómata Celular luego del Segundo Paso de Ordenamiento....61
Figura 24 – Implementación del Autómata para la Ordenamiento Numérico....62
Figura 25 – Resultados obtenidos para el Autómata de Ordenamiento Numérico....62
Figura 26 – Esquema de acoplamiento para el modelo de tráfico vehicular....62
Figura 27 – Implementación del modelo de Tráfico Vehicular....63
Figura 28 – Resultados obtenidos para la simulación del modelo de Tráfico Vehicular....64
Figura 29 – Esquema de acoplamiento para el modelo de Difusión del Calor sobre una superficie....65
Figura 30 – Implementación del Modelo de Difusión del Calor sobre una Superficie....66
Figura 31 – Resumen de la salida generada para el modelo de difusión del calor en una superficie....66
Figura 32 – Vecindario para el ejemplo de Difusión del Calor en 3D....67
Figura 33 – Esquema de acoplamiento para el modelo de Difusión del Calor....67
Figura 34 – Definición del modelo de Difusión del Calor en 3D....68
Figura 35 – Resumen de los resultados obtenidos durante la simulación de difusión del calor....69
Figura 36 – Implementación del modelo de Difusión de Partículas de Gas....70
Figura 37 – Resultados para el Modelo de Difusión de Partículas de Gas....71
Figura 38 – Esquema de acoplamiento para el ejemplo que muestra las extensiones del manejo de puertos....71
Figura 39 – Ejemplo del uso de las extensiones realizadas sobre los puertos de Entrada y Salida....72
Figura 40 – Modelo Acoplado de la Clasificadora de Materias Primas....73
Figura 41 – Implementación del Sistema de Embalaje y Clasificación de Materias Primas....74
Figura 42 - Resultados para el Sistema de Embalaje y Clasificación de Materias Primas....75
Figura 43 – Resultados de la Simulación sobre Embalaje y Clasificación de Sustancias....76
Figura 44 – Implementación del modelo del Pinball....77
Figura 45 – Resumen de los resultados arrojados para la simulación del Pinball....78
Figura 46 – Implementación del Modelo de la Pista de Baile....80
Figura 47 – Contenido del archivo dance.val definiendo el estado inicial para el modelo de la Pista de Baile....80
Figura 48 – Resumen de los resultados generados por la herramienta para la simulación de la Pista de Baile....81
Figura 49 – Implementación del modelo de los Mosquitos....83
Figura 50 – Contenido del archivo Eventos.ev detallando los eventos externos para el problema de los mosquitos....83
Figura 51 – Contenido del archivo mosquito.val detallando el estado inicial para el problema de los mosquitos....83
Figura 52 – Resumen de los resultados generados para el problema de los mosquitos....84

1 Introducción

Existen sistemas para los cuales resulta difícil, y en ciertos casos imposible, hallar soluciones analíticas con resultados satisfactorios. Para estos sistemas se ha difundido el uso de metodologías y herramientas de simulación. El uso de la simulación tiene muchas ventajas, entre ellas: puede reducirse el tiempo de desarrollo del sistema, las decisiones pueden verificarse artificialmente, un mismo modelo puede usarse muchas veces, la experimentación puede ser controlada, etc.

La simulación se ha convertido en una metodología muy utilizada en estos tiempos. Debido a esto, el principal objetivo es el desarrollo de herramientas genéricas que faciliten su empleo, ampliando las aplicaciones sobre las cuales puede ser usada. Es conveniente que un simulador permita la construcción de modelos en forma jerárquica, mediante la combinación de modelos simples, simplificando la creación y prueba de los mismos.

La herramienta *N–CD++* fue implementada en C++. La versión utilizada es la ANSI/ISO C++ draft standard, publicada en Diciembre 96 (XJ316) por el ANSI/ISO, e incluye las características recientemente incorporadas al lenguaje como template function, C++ style cast y STL (Standard Template Library). El desarrollo sobre un lenguaje estándar posibilitó la portabilidad del mismo a múltiples plataformas sin ningún tipo de modificación del código fuente, desde una simple PC con sistema operativo Windows, hasta en grandes equipos multiusuarios con sistemas operativos que cumplan las normas POSIX. Esto posibilita a los usuarios el desarrollo de modelos en su ámbito habitual para luego poder realizar estudios más profundos con máquinas con mayor poder de cómputo.

El compilador a utilizar debe cumplir los requerimientos antes descriptos. Se ha seleccionado el compilador de C++ **gcc** v.2.8.x de GNU, debido a su carácter de "distribución libre" (G.P.L., General Public License) y su adaptación al estándar.

El código fuente del simulador se divide en dos partes: la librería de simulación, y el código fuente principal que incluye los modelos atómicos. Las rutinas de la librería llevan a cabo por completo todos los pasos de la simulación y se complementan con el código fuente principal y los archivos de configuración para cargar el modelo a simular y configurar aspectos particulares de la plataforma en la cual el simulador correrá.

El desarrollo del presente informe incluye la especificación de las clases participantes del desarrollo, explicando sus objetivos y principales métodos. Posteriormente se detalla el proceso de simulación en *N–CD++* y se ejemplifica el uso de la herramienta para la resolución de diversos problemas.

2 Jerarquía de Modelos

Los modelos son los encargados de describir el comportamiento deseado dentro de la simulación. Los modelos atómicos proveen la base para que los nuevos modelos puedan redefinir los criterios para la programación de los eventos internos y cambios de estado. Los modelos acoplados incluyen otros modelos y permiten que estos se conecten entre sí. Todos los modelos mantienen características en común. Esto se refleja en la clase *Model* que es la clase base abstracta y principal en la jerarquía de modelos. Esta clase provee solamente la interfaz para otras clases y nunca será instanciada.

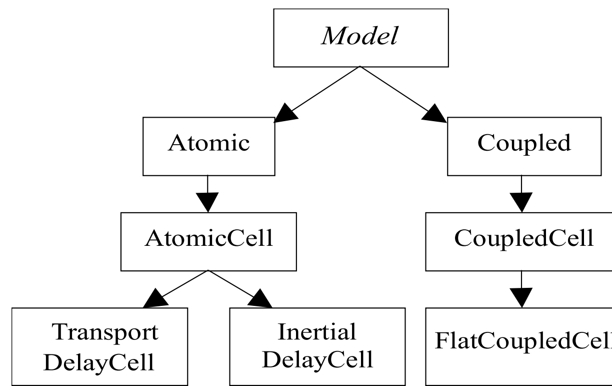


Figura 1 – Jerarquía de Modelos

2.1 Model

Es la clase base abstracta para todos los modelos. Es responsable de administrar los puertos de entrada y salida, conocer la hora del próximo y anterior evento, el identificador del modelo y al padre del mismo.

Métodos principales:

addInputPort

Signatura: `Port &addInputPort( const string &portName )`

Descripción: Agrega un nuevo puerto de entrada a la lista de puertos de entrada del modelo. El parámetro representa el nombre del puerto a crear. Si ya existe un puerto con ese nombre para el modelo, se producirá un **InvalidPortException**. Si la operación se realiza con éxito retorna el puerto creado.

addOutputPort

Signatura: `Port &addOutputPort( const string &portName )`

Descripción: Agrega un nuevo puerto de salida a la lista de puertos de salida del modelo. Este método tiene el mismo comportamiento que *addInputPort*.

nextChange

Signatura: `const Time &nextChange() const`

Signatura: `protected Model &nextChange( const Time& )`

Descripción: Permite obtener o modificar el tiempo restante para la próxima transición interna.

lastChange

Signatura: `const Time &lastChange() const`

Signatura: `protected Model &lastChange( const Time& )`

Descripción: Permite obtener o modificar la hora en la que se produjo la última transición interna.

id

Signatura: `const ModelId &id() const`

Descripción: Devuelve el identificador del modelo.

parent

Signatura: `const Model &parent() const`

Descripción: Devuelve el modelo padre. Si un modelo no tiene padre al invocar este método se producirá un **InvalidModelIdException**.

sendOutput

Signatura: `Model &sendOutput(const Time &t, const Port &p, const Value &v)`

Descripción: Envía un mensaje con valor y hora especificada, a través de un puerto del modelo. Este método solamente puede ser invocado por las subclases de *Model*.

processor

Signatura: `Processor &processor()`
Descripción: Devuelve el procesador asociado al modelo. En caso de no tener procesador asociado se producirá un **InvalidProcessorIdException**.

inputPorts

Signatura: `const PortList &inputPorts() const`
Descripción: Devuelve una lista conteniendo los puertos de entrada del modelo.

outputPorts

Signatura: `const PortList &outputPorts() const`
Descripción: Devuelve una lista conteniendo los puertos de salida del modelo.

port

Signatura: `Port &port( const string &portName )`
Descripción: Devuelve un puerto especifico del modelo. Si el puerto no existe se producirá un error.

2.2 Atomic

La clase abstracta *Atomic* representa la interfaz de un modelo atómico. Es una especialización de la clase *Model* y hereda las facilidades para manejo de puertos, consulta del estado interno y planificación del próximo evento. Además, permite cambiar la programación y estado del modelo y provee una interfaz para la inicialización del mismo y para las funciones de transición interna, externa y de salida.

Métodos principales:

initFunction

Signatura: `virtual Model &initFunction() = 0`
Descripción: Este método es invocado cuando se produce una inicialización del modelo. El mismo debe ser obligatoriamente implementado en las subclases de *Atomic*.

externalFunction

Signatura: `virtual Model &externalFunction( const ExternalMessage & ) = 0`
Descripción: Este método es invocado cuando arriba un evento externo al modelo. El mismo debe ser obligatoriamente implementado en las subclases de *Atomic*.

internalFunction

Signatura: `virtual Model &internalFunction( const InternalMessage & ) = 0`
Descripción: Este método es invocado cuando se deba producir un evento interno en el modelo. El mismo debe ser obligatoriamente implementado en las subclases de *Atomic*.

outputFunction

Signatura: `virtual Model &outputFunction( const InternalMessage & ) = 0`
Descripción: Este método es invocado cuando se deba producirse la salida del modelo. El mismo debe ser obligatoriamente implementado en las subclases de *Atomic*.

holdIn

Signatura: `Model &holdIn( const State &, const Time & )`
Descripción: Cambia el estado del modelo al estado especificado como parámetro, y programa el próximo evento para la hora indicada.

passivate

Signatura: `Model &passivate()`

Descripción: Cambia el estado del modelo a *Pasivo* y la hora del próximo evento a infinito.

state

Signatura: `Model &state( const State &s )`
Signatura: `const State &state() const`
Descripción: Permite obtener o modificar el estado del modelo.

2.3 Coupled

La clase *Coupled* es una especialización de la clase *Model* y representa un modelo acoplado, permitiendo la creación y manipulación de grupos de modelos que a su vez contienen otros modelos. Un modelo acoplado dispone de una lista para almacenar los modelos básicos que lo componen.

Métodos principales:

addModel

Signatura: `Model &addModel( Model & )`
Descripción: Agrega un modelo básico al modelo acoplado, insertándolo en la lista de modelos del mismo.

addInfluence

Signatura:
```
virtual Model &addInfluence( const string &sourceName,
                const string &sourcePort, const string &destName,
                const string &destPort)
```
Descripción: Agrega una influencia entre dos componentes pertenecientes al modelo acoplado. Los parámetros indican el nombre del modelo origen y del modelo destino, junto a los puertos de cada uno de estos. Si uno de los modelos no pertenece a los hijos del acoplado esto producirá un **InvalidChildException**.
El método obtiene el puerto correspondiente del modelo origen, e invoca al método *addInfluence* (de la clase *Port*) con el puerto correspondiente al modelo destino, creando así un enlace entre los puertos.

children

Signatura: `const ModelList &children() const`
Descripción: Devuelve una lista de todos los modelos básicos que integran al modelo acoplado.

2.4 AtomicCell

La clase abstracta *AtomiCell* es una especialización de la clase *Atomic* y provee la interfaz para las celdas atómicas de un modelo celular. Es responsable de conocer la función de cómputo local asociada, el vecindario, los distintos puertos y el valor de la celda. Además, se encarga de la inicialización de la celda, enviando el estado de ésta a sus vecinos. El puerto de entrada *neighborChange* y el de salida *Out* se crean automáticamente al construir una nueva instancia de esta clase. El resto de los puertos de entrada y salida se crean en forma dinámica, según sean requeridos. Se dispone de dos listas, llamadas *in* y *output*, para almacenar estos puertos creados dinámicamente. Para cada puerto de entrada se registra, además, la función de cómputo local a utilizar, permitiéndole a la celda un comportamiento alternativo ante la llegada de un mensaje a través de alguno de sus puertos de entrada. También, para cada puerto de entrada se mantiene el valor del último mensaje arribado, el cual puede ser consultado mediante la función *portValue* del lenguaje de definición de reglas.

Métodos principales:

AtomicCell

Signatura:
```
AtomicCell( const string & = cellClassName,
Const LocalTransAdmin::Function &id = LocalTransAdmin::InvalidFn )
```
Descripción: Construye una nueva celda atómica. Sus parámetros indican el nombre de la misma, y la función de cómputo local asociada. Además, se crea el puerto de salida *Out*, y de entrada *neighborChange*, que son los puertos que obligatoriamente tienen todas las celdas atómicas.

value

Signatura: `const Real &value() const`
Signatura: `AtomicCell &value( const Real &val )`
Descripción: Permite obtener o modificar el valor de la celda atómica.

localFunction

Signatura: `const LocalTransAdmin::Function &localFunction() const`
Descripción: Devuelve el identificador de la función de cómputo local.

neighborhood

Signatura: `const NeighborhoodValue &neighborhood() const`
Signatura: `AtomicCell &neighborhood( NeighborhoodValue * )`
Descripción: Permite obtener o modificar el vecindario de la celda atómica.

outputPort

Signatura: `const Port &outputPort() const`
Descripción: Devuelve el puerto de salida *Out*. Este puerto es utilizado para comunicar a la celda con sus vecinos. Además, es posible usarlo para conectar a otros modelos DEVS, aunque para tal motivo *N-CD++* permite el uso de puertos de salida adicionales.

getOutPorts

Signatura: `void getOutPorts(VirtualPortList *vpl)`
Descripción: Devuelve una lista de puertos de salida. Esta lista no contiene al puerto *Out*, sino que solo contiene a los puertos que se crean en forma dinámica.

inputPort

Signatura: `PortList &inputPort()`
Descripción: Devuelve la lista de puertos de entrada de la celda. Esta lista contiene a todos los puertos de entrada que se crean en forma dinámica, por lo tanto no contiene al puerto *neighborChange.*

neighborPort

Signatura: `const Port &neighborPort() const`
Descripción: Devuelve el puerto de entrada *neighborChange* de la celda. Este puerto permite que los vecinos de la celda se conecten a la misma.

addInPort

Signatura: `bool addInPort( string portName )`
Descripción: Agrega un puerto de entrada a la correspondiente lista de puertos de entrada. Si este ya existe devuelve *False*, sino devuelve *True*. Además, asocia la función de cómputo local especificada en la creación de la celda atómica, al puerto agregado. Esta asociación puede ser cambiada mediante el método *setPortInFunction*. Por último, establece el valor del último mensaje arribado a la celda como el valor indefinido.

addOutPort

Signatura: `bool addOutPort( string portName )`
Descripción: Agrega un puerto de salida a la correspondiente lista de puertos de salida. Si este ya existe devuelve *False*, sino devuelve *True*.

setPortInFunction

Signatura:
```
void setPortInFunction( const string portName,
                        const string functionName)
```
Descripción: Permite asociar una función de transición a un puerto de entrada. Esta asociación será de utilidad cuando llegue un mensaje a través del puerto especificado, por lo que se deberá ejecutar la función de transición asociada al puerto para obtener el nuevo valor de la celda.

Este método tiene una relación directa con la cláusula *PortInFunction* del lenguaje para la escritura de reglas de *N-CD++*.

setPortValue

Signatura: `void setPortValue(const string portName, const Real portValue)`

Descripción: Establece el valor del último mensaje arribado por el puerto especificado. Este valor es obtenido por la función *PortValue* del lenguaje de definición de reglas de *N-CD++*.

outputFunction

Signatura: `virtual Model &outputFunction( const InternalMessage & )`

Descripción: Representa la función de salida de la celda atómica, la cual envía un mensaje con la hora del mensaje pasado como parámetro, con el valor que tiene en ese momento la celda, a través del puerto *Out*.

initFunction

Signatura: `Model &initFunction()`

Descripción: Inicializa la celda atómica. Para esto ejecuta la función de transición externa donde su parámetro es un mensaje conteniendo la hora cero, el nombre de la celda, y el puerto *neighborChange*. Esto provocará que se envíen mensajes a todos los vecinos, por lo que al fin de la inicialización, cada celda conocerá el estado de su vecindario.

2.5 TransportDelayCell

La clase *TransportDelayCell* es una especialización de la clase *AtomicCell*. Representa a las celdas que utilizan una demora de transporte, redefiniendo el comportamiento para las funciones de transición interna, externa y de salida. La demora de transporte aplica una política FIFO para los eventos. A medida que estos llegan si hay alguna programación pendiente el nuevo evento será encolado para ser ejecutado posteriormente.

Métodos principales:

internalFunction

Signatura: `Model &internalFunction( const InternalMessage & )`

Descripción: Este método es invocado cuando arriba un mensaje correspondiente a un evento interno. En este caso, como la celda utiliza una demora de transporte, se elimina el evento que fue encolado y se reprograma la celda en caso de ser necesario.

externalFunction

Signatura: `Model &externalFunction( const ExternalMessage & )`

Descripción: Este método es invocado cuando arriba un mensaje correspondiente a un evento externo. Si dicho mensaje provino del puerto *neighborChange*, indica que el valor de estado de un vecino ha cambiado, por lo que se debe evaluar la celda en cuestión para obtener su nuevo estado. Si el mensaje provino de otro puerto de entrada, se obtiene la función de cómputo local asociada al puerto. En caso de que no haya sido asociada ninguna función al mismo, el valor proveniente en el mensaje es encolado con su demora. Sino, se evalúa la correspondiente función de cómputo local y su resultado será el valor encolado junto a la demora correspondiente.

outputFuncttion

Signatura: `Model &outputFunction( const InternalMessage & )`

Descripción: Este método es invocado antes de atender un evento interno. En este caso, por tratarse de una celda con demora de transporte, se toma el primer valor de la cola, que se corresponde al valor actual de la celda, y se envía por el puerto de salida.

2.6 *InertialDelayCell*

La clase *InertialDelayCell* es una especialización de la clase *AtomicCell*. Representa a las celdas que utilizan una demora inercial, redefiniendo el comportamiento para las funciones de transición interna, externa y de salida. Cuando un nuevo evento externo llega la celda evalúa la función de cálculo local obteniendo un valor y una demora. Luego analiza la cantidad de tiempo faltante hasta el próximo evento. Si es mayor a cero hace remoción del valor y programa el próximo evento con la nueva demora.

Métodos principales:

internalFunction

Signatura: `Model &internalFunction( const InternalMessage & )`

Descripción: Este método es invocado cuando arriba un mensaje correspondiente a un evento interno. Al arribar un evento la función de salida ya ha sido invocada y por lo tanto la celda se pasiva hasta la llegada de un nuevo evento externo.

externalFunction

Signatura: `Model &externalFunction( const ExternalMessage & )`

Descripción: Este método es invocado cuando arriba un mensaje correspondiente a un evento externo. Si el mensaje provino del puerto *neighborChange*, indicando que el valor de estado de un vecino ha cambiado, se evalúa la función de cómputo local, obteniendo así un nuevo valor para el estado y una nueva demora.
Si el mensaje provino de otro puerto de entrada, se obtiene la función de cómputo local asociada al puerto. En caso de que no haya sido asociada ninguna función al mismo, el valor proveniente en el mensaje es encolado con su demora. Sino, se evalúa la correspondiente función de cómputo local.
Si el nuevo valor de estado es distinto al anterior, se analiza el tiempo restante para el próximo evento; si es mayor a cero, se remueve el último valor y se reprograma con el nuevo.

outputFunction

Signatura: `Model &outputFunction( const InternalMessage & )`

Descripción: Envía el valor actual de la celda a sus vecinos, a través del puerto *Out*. El tiempo del mensaje es el tiempo actual de simulación.

2.7 *CoupledCell*

La clase *CoupledCell* es una especialización de la clase *Coupled*, y representa a los modelos celulares acoplados. Es responsable de conocer la grilla de celdas, la dimensión, el tipo de demora a utilizar y su tiempo por defecto, el tipo de borde, el valor inicial para cada celda, la función de cómputo local por defecto para las celdas y las zonas definidas con comportamientos alternativos. Además, permite la creación de un conjunto de celdas y los vínculos entre ellas.

Métodos principales:

borderWrapped

Signatura: `CoupledCell &borderWrapped( bool )`

Signatura: `bool borderWrapped() const`

Descripción: Permite obtener o modificar la propiedad del modelo acoplado celular que indica si la grilla es toroidal o no.

inertialDelay

Signatura: `CoupledCell &inertialDelay( bool )`

Signatura: `bool inertialDelay() const`

Descripción: Permite obtener o modificar la propiedad del modelo acoplado celular que indica si el tipo de demora utilizada es inercial o de transporte.

initialCellValue

Signatura: `CoupledCell &initialCellValue( const Real & )`
Signatura: `const Real &initialCellValue() const`
Descripción: Permite obtener o modificar la propiedad que indica el valor inicial para todo el modelo acoplado celular en el momento de ser creado.

defaultDelay

Signatura: `CoupledCell &defaultDelay( const Time & )`
Signatura: `const Time &defaultDelay() const`
Descripción: Permite obtener o modificar la propiedad que indica la demora por defecto con la que las celdas deben ser creadas.

localTransition

Signatura: `CoupledCell &localTransition(const LocalTransAdmin::Function &)`
Signatura: `const LocalTransAdmin::Function &localTrans() const`
Descripción: Permite obtener o modificar la función de cómputo local con la que las celdas deben ser creadas.

setLocalTransition

Signatura:
```
virtual CoupledCell &setLocalTransition( const CellPosition &,
                                         const LocalTransAdmin::Function & )
```
Descripción: Establece la función de cómputo local para una celda en particular.

dim

Signatura: `CoupledCell &dim( nTupla &nt )`
Signatura: `CoupledCell &dim( unsigned int Dim, unsigned int Width)`
Signatura: `const nTupla &dimension() const`
Descripción: Obtiene o establece la dimensión del modelo acoplado celular.

createCells

Signatura:
```
virtual CoupledCell &createCells( const CellPositionList
           &neighbors, const CellPositionList &selectList )
```
Descripción: Crea la grilla de celdas, en el momento de la creación del modelo acoplado celular. Los parámetros representan la lista de desplazamientos de celdas que representan el vecindario, y la lista de posiciones de celdas que se corresponde a la función **select** del formalismo *Cell-DEVS*, que representa el orden de creación de las mismas.
La creación de la grilla de celdas implica la creación de cada una de las celdas que componen al modelo acoplado celular, y el establecimiento de sus valores iniciales. También se crean los enlaces entre puertos de las celdas que integran la grilla, de acuerdo al vecindario.

cellState

Signatura: `CellState *cellState()`
Signatura: `CoupledCell &cellState( CellState * )`
Descripción: Establece o modifica la grilla de celdas del modelo acoplado celular.

setCellValue

Signatura: `CoupledCell &CoupledCell::setCellValue(const ModelId &, Value)`
Signatura:
```
CoupledCell &CoupledCell::setCellValue(const CellPosition &,
                          const Real &)
```
Signatura:
```
CoupledCell &CoupledCell::setCellValue(const string &sCellPos,
                          const Real &v )
```
Descripción: Establece un nuevo valor para una celda del modelo celular acoplado.

2.8 FlatCoupledCell

La clase *FlatCoupledCell* es una especialización de la clase *CoupledCell*. Los modelos celulares achatados no crean una grilla de modelos atómicos, sino que usan una matriz con los valores representando cada celda. Debido a que estos modelos no tienen modelos hijos, se evita la sobrecarga del pasaje de mensajes entre los coordinadores. Por esto, la clase debe proveer métodos para administrar al conjunto de celdas virtuales. Esta estrategia permite que la utilización de una grilla de celdas virtuales sea completamente transparente para el resto de los modelos externos.

Métodos principales:

createCells

Signatura:
```
CoupledCell &createCells( const CellPositionList &neighbors,
                          const CellPositionList &selectList )
```
Descripción: Crea una estructura para contener los valores que representan el estado de las celdas. Dicha estructura es de tipo *CellState*.

setLocalTransition

Signatura:
```
CoupledCell &setLocalTransition( const CellPosition &,
                                 const LocalTransAdmin::Function & )
```
Descripción: Registra la función de cómputo local para una celda específica.

addInfluence

Signatura:
```
Model &addInfluence( const string &sourceName,
            const string &sourcePort, const string &destName,
            const string &destPort)
```
Descripción: Registra una conexión entre dos celdas, al igual que en su clase base. Como las celdas son virtuales, es necesario registrar las influencias de entrada y salida en las listas *Xlist* e *Ylist* respectivamente. Además, se establece en indefinido el valor del último mensaje arribado por el puerto de entrada.

initFunction

Signatura:
```
Model &initFunction()
```
Descripción: Inicializa los valores que representan el estado de las celdas. Aquí se ejecuta la función de transición externa para cada celda virtual, con tiempo cero.

externalFunction

Signatura:
```
Model &externalFunction( const Time &, const CellPosition &,
      bool bExternal = false, Real eventValue = Real::tundef,
      const string &portIn = "" )
```
Descripción: Sus parámetros indican la hora y el valor del evento, la celda destinataria del mismo, y si proviene de un modelo externo o de un vecino. En caso de provenir de un modelo externo, *portIn* contiene el puerto por el cual arribó dicho evento.
Si el evento externo proviene a través del puerto *neighborChange*, indicando que hubo un cambio de estado en alguno de sus vecinos, se invoca a la función de cómputo local correspondiente a la celda, pasándole como parámetro el vecindario y la lista de últimos valores de los mensajes arribados a través de los puertos de entrada, para obtener así una demora y un nuevo valor de estado. Las celdas pueden tener distintas funciones de cómputo local asociadas según la definición de zonas realizadas en la especificación de modelos.
Si el evento externo proviene de un puerto de entrada distinto del puerto *neighborChange*, primero se establece el valor del evento como el nuevo valor del último mensaje arribado a través del puerto. Posteriormente se verifica si el puerto de la celda tiene asociada una función de cómputo a ser usada cuando arribe un mensaje por este, según fue definida con la cláusula *portInFunction* en *N-CD++*. Si hay una función asociada, la utiliza para calcular la nueva demora y valor de estado de la celda, usando el vecindario y la lista de valores de los últimos mensajes arribados por cada puerto de entrada. Sino, el valor del evento será el futuro valor para la celda.

Para finalizar, como las celdas no existen, es necesario registrar en una lista todos los eventos programados para poder elegir al inminente.

internalFunction

Signatura: `Model &internalFunction( const Time & )`

Descripción: Al producirse un evento interno, se generan las salidas correspondientes y se actualiza la lista de celdas influenciadas que deben ser planificadas, enviando un mensaje de transición externa a cada una de ellas, indicando el cambio de estado en el vecindario.

getOutputPorts

Signatura: `void getOutputPorts(VirtualPortList **vpl,const CellPosition&)`

Descripción: Obtiene una lista de los puertos de salida de la celda especificada.

getInputPortsValues

Signatura:
```
void getInputPortValues( PortValues *pv,
      const CellPosition &cellPos, const string &portIn )
```

Descripción: Obtiene a partir de la lista de valores de los puertos de todas las celdas del modelo acoplado, una lista que contiene solo los puertos referentes a una celda específica.

setCellValue

Signatura: `CoupledCell &setCellValue( const CellPosition &, const Real &)`

Descripción: Establece el valor de estado para una celda perteneciente al modelo acoplado celular.

setPortInFunction

Signatura:
```
void setPortInFunction( const CellPosition &cellPos,
      const string &sourcePort, const string &functionName )
```

Descripción: Asocia una función de cómputo local al puerto de entrada específico de una celda.

setLastValuePortIn

Signatura:
```
void setLastValuePortIn( const CellPosition &cellPos,
      const string &portIn, const Real &value)
```

Descripción: Establece el valor del último mensaje arribado a través de un puerto de entrada de la celda especificada.

3 Jerarquía de Procesadores

Los procesadores tienen como principal responsabilidad proveer el mecanismo de simulación necesario para que los modelos puedan llevar a cabo su comportamiento. La clase abstracta *Processor* es la responsable de administrar la recepción de mensajes y tomar las acciones correspondientes.

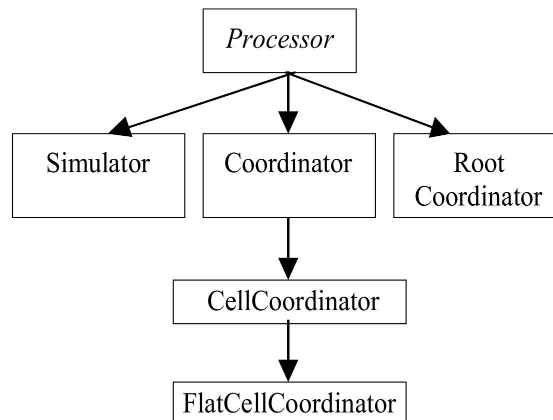


Figura 2 – Jerarquía de Procesadores

La clase *RootCoordiator* representa a la raíz del árbol de componentes en una simulación. *Simulator* y *Processor* son los procesadores responsables de simular a los modelos atómicos y acoplados respectivamente. Para los modelos celulares se agregan *CellCoordinator*, que está asociado a un modelo celular acoplado, mientras que *FlatCellCoordinator* esta asociado a los modelos celulares achatados.

La asociación entre modelos y procesadores esta dada por los pares *Atomic–Simulator*, *Coupled–Coordinator*, *CellCoupled–CellCoordinator* y *FlatCoupled–FlatCoordinator.* Por cada modelo existente, existirá un único procesador asociado y cada procesador administrará a un único modelo.

3.1 Processor

La clase *Processor* es la clase base abstracta para toda la jerarquía de procesadores, y representa el mecanismo de simulación utilizado. Define los métodos abstractos para la recepción de los distintos tipos de mensajes. Además, se encarga de conocer a su modelo asociado y a su procesador padre, y de enviar mensajes de salida hacia su procesador padre.

Métodos principales:

receive

Signatura: `virtual Processor &receive( const InitMessage & )`
Signatura: `virtual Processor &receive( const InternalMessage & )`
Signatura: `virtual Processor &receive( const OutputMessage & )`
Signatura: `virtual Processor &receive( const ExternalMessage &)`
Signatura: `virtual Processor &receive( const DoneMessage &)`
Descripción: Genera una excepción de tipo **InvalidMessageException** debido a que esta clase abstracta no puede recibir mensajes, sino que esta responsabilidad recae en las subclases de *Processor*.

id

Signatura: `const ProcId &id() const`
Descripción: Devuelve el identificador del procesador.

model

Signatura: `Model &model()`
Descripción: Devuelve el modelo asociado al procesador.

sendOutput

Signatura:
```
virtual Processor &sendOutput( const Time &, const Port &,
                        const Value &)
```
Descripción: Envía un mensaje de salida al padre, con la hora, valor y puertos especificados.

3.2 Simulator

La clase *Simulator* es una especialización de la clase *Processor.* Es usada para representan el mecanismo de simulación para los modelos atómicos. Se encarga de recibir los mensajes y derivarlos al modelo atómico asociado y de enviar mensajes hacia su procesador padre indicando la hora del próximo evento de su modelo asociado.

Métodos principales:

receive

Signatura: `Processor &receive( const InitMessage & )`
Descripción: Invoca a la función de inicialización de su modelo atómico asociado. Posteriormente envía un mensaje *done* a su procesador padre, indicando la hora de su primer evento interno.

receive

Signatura: `Processor &receive( const InternalMessage & )`

Descripción: Representa el arribo de un mensaje representando un evento interno. Se procede a ejecutar la función de salida del modelo asociado, y posteriormente la función de transición interna del mismo. Luego se envía un mensaje *done* al procesador padre, indicando una posible replanificación.

receive

Signatura: `Processor &receive( const ExternalMessage & )`

Descripción: Representa el arribo de un mensaje representando un evento externo. Se procede a ejecutar la función de transición externa del modelo asociado, enviando luego un mensaje *done* a su padre, con la nueva planificación para su modelo asociado.

3.3 Coordinator

La clase *Coordinator* es una especialización de la clase *Processor*. Representa el mecanismo de simulación para los modelos acoplados. Es responsable de redireccionar los mensajes de inicialización hacia todos sus hijos, los mensajes externos y de salida hacia los correspondientes influenciados, y los mensajes internos hacia el procesador inminente.

Métodos principales:

receive

Signatura: `Processor &receive( const InitMessage & )`

Descripción: Representa la recepción de un mensaje de inicialización enviado por su procesador padre. El método se encarga del envío de un mensaje de inicialización a todos los procesadores que componen el modelo acoplado asociado, y registra en la variable de instancia *doneCount* la cantidad de mensajes enviados para saber la cuantos mensajes *done* debe esperar en respuesta de cada uno de sus hijos.

receive

Signatura: `Processor &receive( const InternalMessage & )`

Descripción: Representa la recepción de un mensaje interno. El método se encarga del envío de un mensaje de transición interna (tipo *) al procesador correspondiente del modelo inminente, y registra en la variable de instancia *doneCount* el valor 1, indicando que espera un mensaje *done* en respuesta del mensaje enviado.

receive

Signatura: `Processor &receive( const ExternalMessage & )`

Descripción: Representa la recepción de un mensaje externo. El método se encarga del envío de un mensaje de transición externa (tipo X) a los procesadores de los modelos que son influenciados por el puerto por donde ingreso el mensaje. Además, registra en la variable de instancia *doneCount* la cantidad de mensajes enviados, indicando que espera igual cantidad de mensajes *done* en respuesta. Cuando recibe todos los mensajes *done* que espera, envía un mensaje *done* a su padre, con la hora de su hijo inminente.

receive

Signatura: `Processor &receive( const OutputMessage & )`

Descripción: Envía un mensaje externo (tipo X) a todos los procesadores de los modelos influenciados por el puerto por donde se recibió el mensaje de salida. Si alguno de los puertos influenciados es un puerto de salida del modelo acoplado, se envía un mensaje de salida (tipo Y) al procesador padre. Además, se registra en la variable de instancia *doneCount* la cantidad de mensajes enviados a los procesadores hijos, indicando que espera igual cantidad de mensajes *done* en respuesta.

receive

Signatura: `Processor &receive( const DoneMessage & )`

Descripción: Actualiza la variable de instancia *doneCount*, decrementando su valor en 1. Si luego de la actualización *doneCount* tiene valor cero, significa que recibió todos los mensajes *done* que

esperaba, por lo que recalcula la hora e identidad del hijo inminente, y envía a su procesador padre la hora del próximo evento interno en un mensaje *done*.

recalcInminentChild

Signatura: `Coordinator &recalcInminentChild()`

Descripción: Recalcula el hijo inminente y la hora del próximo evento entre todos los modelos que componen al modelo acoplado asociado. Este método encapsula el uso de la función **select** del formalismo, ya que frente a dos modelos con la misma hora de próximo evento se decide utilizando el orden propuesto durante la especificación de los modelos.

3.4 Root

La clase *Root* es una especialización de la clase *Processor*. Representa la raíz del árbol de procesadores a partir de la cual comienza el mecanismo de simulación. Es el único procesador que no tiene modelo asociado. Existe una sola instancia de esta clase en toda la simulación y debe ser conocida públicamente. Es responsable de la administración de los eventos externos, avanzar la hora de simulación, generar la salida a partir de los eventos de salida recibidos, y de iniciar y detener la simulación.

Métodos principales:

instance

Signatura: `static Root &Instance()`

Descripción: Devuelve o genera la única instancia de la clase *Root*.

initialize

Signatura: `Root &initialize()`

Descripción: Establece la hora del próximo y anterior evento en cero, limpia la lista de eventos externos y registra la hora de finalización de la simulación, almacenada en una variable de instancia, con valor infinito.

stopTime

Signatura: `Root &stopTime( const Time & )`

Signatura: `const Time &stopTime() const`

Descripción: Obtiene o establece la hora de finalización de la simulación.

addExternalEvent

Signatura: `Root &addExternalEvent(const Time&, const Port&, const Real &)`

Descripción: Agrega un evento a la lista de eventos externos. Los parámetros indican la hora, el valor y el puerto de entrada del evento.

top

Signatura: Coupled &top()

Descripción: Devuelve el modelo acoplado de mayor nivel en la jerarquía.

receive

Signatura: `Processor &receive( const OutputMessage & )`

Descripción: Envía por el dispositivo de salida especificado en la única instancia de la clase de *MainSimulator*, un string conteniendo la hora, valor y puerto del mensaje especificado.

receive

Signatura: `Processor &receive( const DoneMessage & )`

Descripción: Se encarga del avance del tiempo, haciendo que continúe el proceso de simulación. En caso de haber llegado a la hora de finalización de la simulación, o de haber consumido todos los eventos externos y que todos los modelos que componen la jerarquía se encuentran en estado *pasivo* (es decir, la hora de próximo evento es infinito), la simulación se da por finalizada; de

no ser así, se analiza el próximo evento a ser tratado y se envía un mensaje al coordinador de mayor nivel, avanzando de esta forma la hora de simulación.

stop

Signatura: `Root &stop()`

Descripción: Detiene la simulación. Para ello, ejecuta el método *stop* de *SingleMsgAdm*.

simulate

Signatura: `Root &simulate()`

Descripción: Inicia la simulación. Para ello, ordena la lista de eventos externos de acuerdo a la hora de ocurrencia de los mismos, y envía un mensaje de inicialización al modelo acoplado de mayor nivel en la jerarquía. Posteriormente ejecuta el método *run* de la clase *MessageAdmin*, comenzando así el ciclo de envío/recepción de mensajes.

3.5 CellCoordinator

La clase *CellCoordinator* es una especialización de la clase *Coordinator*. Representa el mecanismo de simulación para los modelos celulares. Es responsable de selección del hijo inminente y el tratamiento de los mensajes de salida para evitar mensajes repetidos enviados a las celdas, debido a que estos solo indican la necesidad de recálculo del estado, pero no un valor externo específico como en el resto de los modelos DEVS.

Métodos principales:

receive

Signatura: `Processor &receive( const InternalMessage & )`

Descripción: Envía un mensaje que representa a un evento interno (tipo *) a todos los procesos asociados de los modelos inminentes, es decir, cuya hora de próximo cambio de estado es igual a la hora del mensaje * recibido. Además, actualiza la variable de instancia *doneCount* con la cantidad de mensajes enviados, para registrar así el número de mensajes *done* que espera recibir.
El orden en que se envían los mensajes representa la función **select** del formalismo, que por defecto, si no se define en la especificación de los modelos, se aplica el orden mostrado en la Figura 3, definido recursivamente.

$$(x_1, x_2) < (y_1, y_2) \Leftrightarrow x_1 < y_1 \vee (x_1 = y_1 \wedge x_2 < y_2)$$

$$(x_1, x_2, x_3, ..., x_n) < (y_1, y_2, y_3, ..., y_n) \Leftrightarrow (x_1 < x_2) \vee (x_1 = y_1 \wedge (x_2, x_3, ..., x_n) < (y_2, y_3, ..., y_n))$$

Figura 3 – Definición recursiva para el orden de celdas

receive

Signatura: `Processor &receive( const OutputMessage & )`

Descripción: Para cada influencia a un puerto de salida, se envía un mensaje de salida (tipo Y) hacia su procesador padre. Para el resto de las influencias del puerto por el cual se recibió el mensaje, si el modelo que tiene el puerto no se encuentra registrado en la lista *influenced*, se envía un mensaje externo (tipo X) y se lo registra en la lista. Posteriormente se actualiza la variable de instancia *doneCount*, con la cantidad de mensajes externos enviados, indicando la espera de igual cantidad de mensajes *done*.

receive

Signatura: `Processor &receive( const DoneMessage & )`

Descripción: Al recibir un mensaje *done*, decrementa la variable *doneCount* en 1. Si luego de la actualización, la misma tiene valor 0, significa que recibió todos los mensajes *done* que esperaba, por lo que recalcula la hora e identidad de sus hijos inminentes y los almacena en *inminentsChild*. Posteriormente envía un mensaje done al procesador padre, indicándole la hora del próximo evento interno. Para finalizar, limpia la lista *influenced*.

3.6 FlatCoordinator

La clase *FlatCoordinator* es una especialización de la clase *Coordinator*. Representa el mecanismo de simulación para los modelos celulares achatados. Se encarga de recibir los mensajes y redireccionarlos hacia el modelo acoplado chato asociado. Para este tipo de coordinador, que no posee hijos asociados, los métodos para recibir mensajes de *Output* y *Done* nunca serán invocados.

Métodos principales:

receive

Signatura: `Processor &receive( const InitMessage & )`

Descripción: Ejecuta el método de inicialización del modelo celular acoplado chato asociado, establece los valores iniciales para la hora del próximo y ultimo evento, y luego envía un mensaje *done* a su procesador padre, indicando la hora de su próximo cambio de estado.

receive

Signatura: `Processor &receive( const InternalMessage & )`

Descripción: Ejecuta el método de transición interna del modelo celular acoplado chato asociado, actualiza los valores para la hora del próximo y ultimo evento, y luego envía un mensaje *done* a su procesador padre, indicando la hora de su próximo cambio de estado.

receive

Signatura: `Processor &receive( const ExternalMessage & )`

Descripción: Ejecuta el método de transición externa del modelo celular acoplado chato asociado, actualiza los valores para la hora del próximo y ultimo evento, y luego envía un mensaje *done* a su procesador padre, indicando la hora de su próximo cambio de estado.

4 Administración de Modelos y Procesadores

Los modelos y procesadores utilizados en la simulación son creados durante la interpretación del archivo de especificación de modelos y eliminados al finalizar la ejecución del simulador. Durante el transcurso de la simulación se hace referencia a modelos padres e hijos. Debido a esto, debe proveerse un mecanismo para identificar y obtener estos modelos y procesadores.

4.1 ProcessorAdmin

La clase *ProcessorAdmin* administra todos los procesadores participantes en la simulación. Esta clase debe ser conocida por todos los componentes de la simulación, por lo que existe una única instancia llamada *SingleProcessorAdmin*. Su responsabilidades son la creación del coordinador raíz, los simuladores, los coordinadores, los coordinadores celulares y los celulares achatados. También dispone de métodos para la búsqueda de un procesador a partir de su identificador.

Métodos principales:

generateRoot

Signatura: `Processor &generateRoot()`

Descripción: Crea la única instancia de la clase *Root*.

generateProcessor

Signatura: `Processor &generateProcessor( Atomic * )`

Descripción: Crea una nueva instancia de la clase *Simulator*, relacionado al modelo atómico indicado, y lo agrega a la base de procesadores.

generateProcessor

Signatura: `Processor &generateProcessor( Coupled * )`

Descripción: Crea una nueva instancia de la clase *Coordinator*, relacionado al modelo acoplado indicado, y lo agrega a la base de procesadores.

generateProcessor

Signatura: `Processor &generateProcessor( CoupledCell * )`

Descripción: Crea una nueva instancia de la clase *CellCoordinator*, relacionado al modelo celular acoplado indicado, y lo agrega a la base de procesadores.

generateProcessor

Signatura: `Processor &generateProcessor( FlatCoupledCell * )`

Descripción: Crea una nueva instancia de la clase *FlatCellCoordinator*, relacionado al modelo celular acoplado chato indicado, y lo agrega a la base de procesadores.

processor

Signatura: `Processor &processor( const ProcId & )`

Descripción: Devuelve el procesador cuyo identificador coincida con el especificado como parámetro. En caso de no existir, ocurre un **InvalidModelIdException**.

processor

Signatura: `Processor &processor( const string & )`

Descripción: Devuelve el procesador cuyo nombre coincida con el especificado como parámetro. En caso de no existir, ocurre un **InvalidModelIdException**.

4.2 ModelAdmin

La clase *ModelAdmin* permite la creación de los distintos tipos de modelos (atómicos, celdas atómicas, acoplados, celulares acoplados y celulares acoplados achatados). Debido a que todos los modelos tienen un procesador asociado, la creación es simultánea. Como es la única clase capaz de realizar esta tarea, existe una única instancia, llamada *SingleModelAdm*.

Métodos principales:

registerAtomic

Signatura: `AtomicType registerAtomic(const NewFunction &, const string &)`

Descripción: Registra un nuevo tipo de modelo atómico. Los parámetros indican la función de creación a utilizar y el nombre del modelo. Devuelve un identificador para el modelo atómico registrado.

newAtomic

Signatura: `Atomic &newAtomic(const AtomicType &, const string &modelName)`

Descripción: Crea una nueva instancia del modelo atómico solicitado.

newAtomicCell

Signatura:

```
AtomicCell &newAtomicCell( bool inertial = false,
                           const string &modelName = "AtomicCell" )
```

Descripción: Crea una nueva instancia del modelo atómico celular solicitado, que puede ser con demora inercial o de transporte.

newCoupled

Signatura: `Coupled &newCoupled( const string &modelName )`

Descripción: Crea una nueva instancia de la clase *Coupled*.

newCoupledCell

Signatura: `CoupledCell &newCoupledCell( const string &modelName )`

Descripción: Crea una nueva instancia de la clase *CoupledCell*.

newFlatCoupledCell

Signatura: `FlatCoupledCell &newFlatCoupledCell( const string &modelName )`
Descripción: Crea una nueva instancia de la clase *FlatCoupledCell*.

5 Intercambio de Mensajes

El intercambio de mensajes entre los coordinadores representa el mecanismo del motor de simulación. Deben existir tantos mensajes distintos como eventos posibles dentro del formalismo. A su vez, cada mensaje podrá tener información particular asociada al tipo de evento que representa.

El uso de una entidad para el envío de los mensajes entre los procesadores permite encapsular el mecanismo de pasaje de mensajes, permitiendo cambiar la política de intercambio utilizada solamente modificando el método de distribución de mensajes, sin tener mayor impacto en el resto del código. En esta implementación se usa una política FIFO, haciendo que la simulación sea secuencial, dado que el envío del próximo mensaje se producirá cuando el modelo finalice el procesamiento del mensaje anterior.

En esta sección se detallan los distintos tipos de mensajes y la entidad encargada del intercambio de los mismos entre los procesadores.

5.1 Message

La clase *Message* es una clase abstracta de los mensajes. Dispone de métodos para conocer el emisor y la hora del mensaje.

Métodos principales:

time

Signatura: `const Time &time() const`
Signatura: `Message &time( const Time &time )`
Descripción: Devuelve o modifica la hora del mensaje.

procId

Signatura: `const ProcId &procId() const`
Signatura: `Message &procId( const ProcId &p )`
Descripción: Devuelve o modifica el identificador del procesador que envío el mensaje.

sendTo

Signatura: `virtual Message &sendTo( Processor & ) = 0`
Descripción: Este mensaje nunca es ejecutado en la clase abstracta, pero si en sus subclases. El objetivo en todos los casos es invocar al método *receive* del procesador indicado como parámetro.

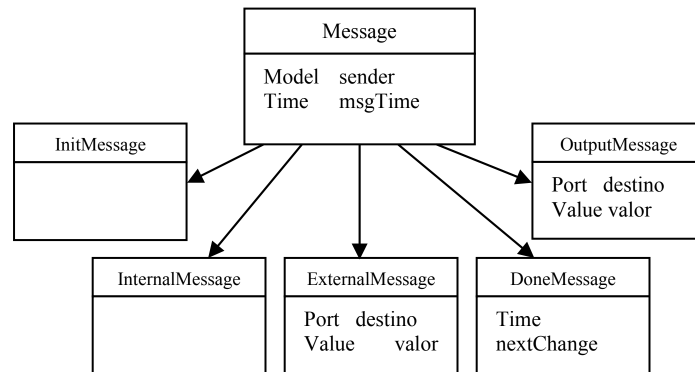


Figura 4 – Jerarquía de Mensajes

5.2 InitMessage

La clase *InitMessage* es una especialización de la clase *Message*. Representa el mensaje de inicialización que reciben los procesadores al comenzar la simulación.

5.3 InternalMessage

La clase *InternalMessage* es una especialización de la clase *Message*. Representa el arribo de un evento interno a un procesador, correspondiente al mensaje ***** del formalismo DEVS.

5.4 ExternalMessage

La clase *ExternalMessage* es una especialización de la clase *Message*. Representa el arribo de un evento externo, correspondiente al mensaje ***X*** del formalismo DEVS. Además de heredar los métodos de la clase *Message*, tiene métodos para identificar el puerto por el cual arribó el mensaje y el valor del mismo.

Métodos principales:

value

Signatura: `const Value &value() const`
Signatura: `Message &value( const Value &val )`
Descripción: Devuelve o modifica el valor contenido en el mensaje.

port

Signatura: `const Port &port() const`
Signatura: `Message &port( const Port &port )`
Descripción: Devuelve o modifica el puerto por el cual arribó el mensaje.

5.5 DoneMessage

La clase *DoneMessage* es una especialización de la clase *Message*. Representa el mensaje que recibe un procesador proveniente de alguno de sus hijos, indicando la hora en la cuál tendrá su próximo cambio de estado. Se corresponde con el mensaje ***Done*** del formalismo DEVS.

Métodos principales:

nextChange

Signatura: `const Time &nextChange() const`
Signatura: `Message &nextChange( const Time & )`
Descripción: Devuelve o modifica la hora del próximo cambio de estado.

5.6 OutputMessage

La clase *DoneMessage* es una especialización de la clase *Message*. Representa a los mensajes de salida. Se corresponde al mensaje ***Y*** del formalismo DEVS. Además de los métodos heredados de Message, dispone de métodos para conocer el emisor y la hora de envío del mensaje.

Métodos principales:

value

Signatura: `const Value &value() const`
Signatura: `Message &value( const Value &val )`
Descripción: Devuelve o modifica el valor de salida que contiene el mensaje.

port

Signatura: `const Port &port() const`
Signatura: `Message &port( const Port &port )`
Descripción: Devuelve o modifica el puerto por el cual el mensaje debe ser enviado.

5.7 MessageAdm

La clase *MessageAdm* administra los pedidos de envío de mensajes entre los procesadores. Como es la única clase capaz de realizar esta tarea existe una única instancia llamada *SingleMessageAdm*, la cual es conocida por todos los procesadores.

Métodos principales:

run

Signatura: `MessageAdmin &run()`
Descripción: Inicia el ciclo de pasaje de mensajes. Este método entra en un ciclo, enviando todos los mensajes que tenga encolados al correspondiente procesador, mientras que haya mensajes por enviar y mientras no se detenga el ciclo mediante la invocación del método *stop*. Además, se encarga de registrar en el archivo de log el intercambio de mensajes entre los procesadores.

stop

Signatura: `MessageAdmin &stop()`
Descripción: Finaliza el ciclo de pasaje de mensajes.

send

Signatura: `MessageAdmin &send( const Message &, const ModelId & )`
Descripción: Envía un mensaje al procesador indicado. Se encarga de encolar el mensaje en una estructura con política FIFO, de tal forma que el mismo es tomado dentro del ciclo del método *run* y enviado al procesador que corresponda.

6 Jerarquía de Cargadores

La herramienta permite indicar a través de qué medio se desea la carga de la especificación de los modelos, los eventos externos, la hora de finalización de la simulación, y el destino de los valores de salida. Existen dos formas de cargar estos valores: a través de la línea de comandos, ó ejecutando el simulador como servidor de simulación a la espera de clientes que especifiquen a través de la conexión la configuración deseada y obtengan los resultados por este medio.

La carga del sistema a simular incluye la lectura e interpretación de los archivos que contienen la especificación de los modelos a simular y la posterior generación del entorno a simular. Una vez cargados todos estos datos se comienza la simulación, obteniendo resultados para la misma, y direccionándolos hacia la salida correspondiente que pueden ser archivos locales ó el canal de comunicación.

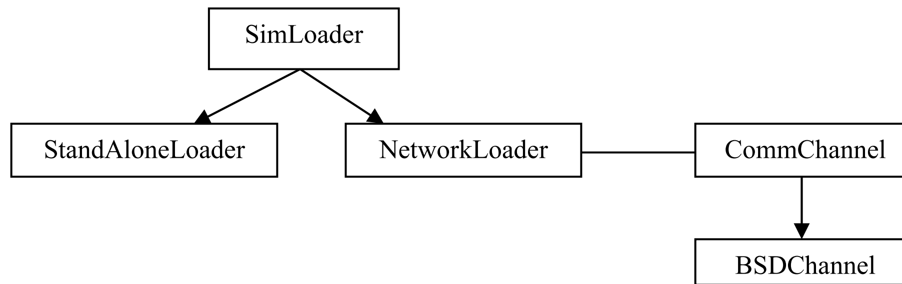


Figura 5 – Jerarquía de Cargadores

6.1 SimLoader

La clase abstracta *SimLoader* provee la interfaz para la carga de la configuración del simulador. Sus clases derivadas implementan la obtención de los parámetros por distintos medios. *StandAloneLoader* es una subclase de *SimLoader* y es responsable de la carga de parámetros a través de la línea de comandos, mientras que *NetworkLoader*, también subclase de *SimLoader*, es responsable de la obtención de los parámetros a través de la red. Esta última solo esta disponible en plataformas *Unix*.

Métodos principales:

modelsStream
Signatura: `virtual istream &modelsStream()`
Descripción: Devuelve el stream usado para el ingreso de la descripción de los modelos.

eventsStream
Signatura: `virtual istream &eventsStream()`
Descripción: Devuelve el stream asociado para el ingreso de los eventos externos.

logStream
Signatura: `virtual ostream &logStream()`
Descripción: Devuelve la salida usada para el almacenamiento de la descripción del intercambio de mensajes entre los procesadores. Este método puede ser sobrecargado por las subclases de *SimLoader*.

outputStream
Signatura: `virtual ostream &outputStream()`
Descripción: Devuelve la salida usada para el almacenamiento de los mensajes de salida generados en la simulación. Este método puede ser sobrecargado por las subclases de *SimLoader*.

printParserInfo
Signatura: `bool printParserInfo() const`
Descripción: Devuelve *true* si se desea imprimir la información detallada de los tokens identificados durante el parseo de las reglas.

stopTime
Signatura: `const Time &stopTime() const`
Signatura: `SimLoader &stopTime( const Time &t )`
Descripción: Devuelve o modifica la hora máxima hasta la cual puede avanzar la simulación.

loadData
Signatura: `virtual SimLoader &loadData() = 0`
Descripción: Define la interfaz para la carga de los parámetros.

writeResults

Signatura: `virtual SimLoader &writeResults() = 0`

Descripción: Define la interfaz para la devolución de los resultados de la simulación, como los eventos de salida y la información inicial del simulador.

6.2 StandAloneLoader

La clase S*tandAloneLoader* es una especialización de la clase *SimLoader* y se encarga de la obtención de los parámetros a través de la línea de comandos. Además, se encarga del manejo de los archivos intervinientes para los datos de entrada y salida, y de establecer los valores por defecto para algunos parámetros.

Métodos principales:

StandAloneLoader

Signatura: `StandAloneLoader(int argc, const char *argv[])`

Descripción: Constructor de la clase. Recibe los parámetros pasados por el usuario por la línea de comandos en la invocación del simulador. Su responsabilidad es analizar dichos parámetros, e inicializar algunas variables del simulador con valores por defecto. Entre las variables inicializadas figuran:

- *iniName*: nombre del archivo que contiene la definición de los modelos a simular. Si no se especificó a través del parámetro –**m**, por defecto se asume el valor "*model.ma*".
- *outName*: indica la salida usada por el simulador donde se enviaran los eventos externos. Si no se especificó a través del parámetro **–o** se asume el valor "*/dev/null*".
- *logName*: indica el dispositivo que será usado para registrar los mensajes del log. Si no se especificó a través del parámetro **–l** se asume el valor "*/dev/null*".
- *evName*: indica la entrada usada para obtener los eventos externos del modelo de mayor nivel en la jerarquía, que serán usados en la simulación. Si no se especificó a través del parámetro **–e** no se asume ningún valor (""), indicando la ausencia de eventos externos como entrada.
- *evalDebugName*: indica la salida usada para registrar los datos generados al evaluar las reglas de los modelos celulares, siempre que el modo de debug correspondiente se encuentre activado. Si no se especificó a través del parámetro **–v** se asume el valor "*/dev/null*".
- *flatDebugName*: indica la salida usada para registrar los datos generados cuando se encuentra activo el modo de debug para modelos celulares achatados. Si no se especificó a través del parámetro **–f** se asume el valor "*/dev/null*".
- *parserFileName*: indica la salida usada para registrar los datos generados cuando el modo de debug del parser se encuentra activo. Si no se especificó a través del parámetro **–p** se asume el valor "*stdout*", enviando los datos por la salida estándar.
- *printParserInfo*: indica si se encuentra activo el modo de debug del parser. Si se especificó el parámetro **–p** tiene el valor *True*.
- *stopTime*: contiene el tiempo de finalización de la simulación. Si no se especificó el parámetro **–t** se asume el valor infinito.
- *preprocessor*: indica si se usará o no el preprocesador. Inicialmente contiene el valor 1, indicando que el preprocesador se encuentra activo, al menos que se lo desactive mediante el parámetro **-b**.

Por otro lado, se inicializa la semilla del generador de números pseudoaleatorios con el valor tomado del reloj de la computadora.

También se establecen en falso el método *Active* de *DebugCellRules* que deshabilita la validación sobre la evaluación de las reglas, y de *UseQuantum* que indica que no se hará uso de un valor de quantum. También se inicializa el ancho y la precisión para los valores reales mostrados por el simulador en sus diferentes salidas, con los valores 12 y 5 respectivamente.

loadData

Signatura: `SimLoader &loadData()`

Descripción: Muestra el mensaje de versión y los valores para los parámetros de la herramienta, y abre los archivos que serán usados para la entrada y la salida de datos. Para los datos de entrada, se lee su contenido y se lo almacena en el espacio reservado definido por *SimLoader*. Además, si el preprocesador se encuentra activo, realiza los procesos de macroexpasión y descarte de comentarios.

writeResults

Signatura: `SimLoader &writeResults()`

Descripción: Cierra los archivos abiertos en el método *loadData*.

6.3 Network Loader

La clase *NetworkLoader* es una especialización de la clase *SimLoader* y se encarga de la obtención de los parámetros y retorno de información a través de un canal de comunicaciones. Para esto utiliza alguna instancia de *CommChannel*, que ofrece primitivas para realizar las comunicaciones a través de la red.

6.4 CommChannel

La clase *CommChannel* es la clase abstracta que especifica la interfaz que deberán redefinir las clases derivadas para implementar el comportamiento de un canal de comunicaciones concreto. Tiene métodos para la apertura y cierre del canal, como así también para la lectura y escritura de datos al mismo.

6.5 BSDChannel

La clase *BSDChannel* es una subclase de *CommChanel*, y se encarga de ofrecer primitivas para el acceso de datos en una red a través de una conexión TCP/IP usando BSD sockets.

7 Lenguaje de Especificación de Reglas para los Modelos Celulares

El comportamiento de los autómatas celulares se describe utilizando un lenguaje cuya BNF se detalla en el *Manual del Usuario*. Este lenguaje permite escribir reglas, mediante el uso de constantes, funciones, referencias a celdas dentro del vecindario, y valores provenientes de los puertos de entrada asociados a la celda.

Cada celda puede tener un valor perteneciente al conjunto $\Re \cup \{ \textbf{?} \}$, donde **?** representa un valor indefinido. La clase *Real* representa los valores posibles para la celda, junto a ciertas operaciones sobre los mismos. El uso de valores indefinidos en el lenguaje hace que el resultado de la evaluación de una operación o función que use dicho valor como argumento sea indefinido. En particular, el uso de los operadores lógicos sobre valores indefinidos generará un resultado indefinido, por ejemplo si se compara una referencia a una celda con el valor indefinido. Debido a esto debe usarse una lógica trivalente, compuesta por los valores *Verdadero*, *Falso* e *Indefinido*. La clase *TvalBool* representa lo valores de esta lógica trivalente.

En esta sección se analizarán las clases las utilizadas en el parseo del lenguaje y las intervinientes en el árbol que representa las reglas, como así también en la interpretación de las mismas durante la ejecución de la simulación.

7.1 Real

La clase *Real* permite representar un valor perteneciente al conjunto $R \cup \{ \textbf{?} \}$ y dispone de operaciones para la manipulación de los mismos. Para representar un valor de este conjunto se utiliza el tipo de datos ***Value*** (que representa un *double* provisto por el lenguaje *C*). Sin embargo, el valor **?** debe representar un valor distinto a cualquier número real, por lo que se utilizó la constante **NAN** (*Not A Number*) definida en el ANSI C para representar dicho valor, la cual puede ser almacenada en una variable de tipo *double*.

Métodos principales:

Real

Signatura: `Real(Value valor)`

Descripción: Crea un nuevo valor perteneciente a los reales, cuyo valor será el indicado como parámetro.

Real

Signatura: `Real()`

Descripción: Crea un nuevo valor perteneciente a los reales, cuyo valor será indefinido (**?**).

value

Signatura: `const Value value() const`

Descripción: Devuelve el valor almacenado.

value

Signatura: `const Value value(Value valor)`

Descripción: Permite establecer un nuevo valor para un *Real*.

ValueUndef

Signatura: `const Value ValueUndef()`

Descripción: Establecer el valor indefinido (**?**) como nuevo valor.

IsUndefined

Signatura: `bool IsUndefined() const`

Descripción: Devuelve *True* si el *Real* tiene un valor indefinido. En caso contrario devuelve *False*.

asString

Signatura:

```
string asString(int width = Impresion::Default.Width(),
      int precision = Impresion::Default.Precision() ) const
```

Descripción: Devuelve un string con el valor del *Real*. Si no se pasan parámetros, se utilizan los valores por defecto para el formato del mismo. Si el valor es indefinido, se devuelve el string "**?**".

operator =

Signatura: `Real &operator = ( const Real & )`

Descripción: Permite la asignación del valor de un *Real*, a partir de otro *Real*.

operator ==

Signatura: `TValBool operator == ( const Real & ) const`

Descripción: Permite verificar si dos números reales son iguales. Su comportamiento se muestra en la Figura 6. Para el caso en que ambos valores no sean indefinidos, la comparación de los mismos utiliza un valor de tolerancia, que puede ser definido como parámetro en la llamada al simulador. De esta forma, los valores no indefinidos a y b son iguales si $a \in [\ b + \Delta, b - \Delta\ ]$, donde Δ representa la tolerancia utilizada.

operator !=

Signatura: `TValBool operator != ( const Real & ) const`

Descripción: Permite verificar si dos reales son distintos. Su comportamiento se muestra en la Figura 6 y se utiliza el mismo criterio de comparación, usando la tolerancia al igual que con el operador ==.

operator >

Signatura: `TValBool operator > ( const Real & ) const`

Descripción: Permite verificar si un real es mayor a otro. Su comportamiento se muestra en la Figura 6.

operator <

Signatura: `TValBool operator < ( const Real & ) const`

Descripción: Permite verificar si un real es menor a otro. Su comportamiento se muestra en la Figura 6.

operator >=

Signatura: `TValBool operator >= ( const Real & ) const`

Descripción: Permite verificar si un real es mayor o igual a otro. Su comportamiento se muestra en la Figura 6.

operator <=

Signatura: `TValBool operator <= ( const Real & ) const`

Descripción: Permite verificar si un real es menor o igual a otro. Su comportamiento se muestra en la Figura 6.

operator +

Signatura: `const Real operator + (const Real &) const`

Descripción: Devuelve una nueva instancia de *Real*, con el valor de la suma del parámetro más el valor de la instancia. Si alguno de los valores a sumar es indefinido, devuelve el valor indefinido.

operator -

Signatura: `const Real operator - (const Real &) const`

Descripción: Devuelve una nueva instancia de *Real*, cuyo valor es la resta del valor de la instancia menos el valor del parámetro. Si alguno de los valores participantes en la resta es indefinido, devuelve el valor indefinido.

operator *

Signatura: `const Real operator * (const Real &) const`

Descripción: Devuelve una nueva instancia de *Real*, con el valor del producto del parámetro por el valor de la instancia. Si alguno de los valores a multiplicar es indefinido, devuelve el valor indefinido.

operator /

Signatura: `const Real operator / (const Real &) const`

Descripción: Devuelve una nueva instancia de *Real*, cuyo valor es la división del valor de la instancia dividido el valor del parámetro. Si el dividendo o el divisor son indefinidos, o si el divisor es cero, devuelve el valor indefinido.

operator TValBool

Signatura: `operator TValBool() const`

Descripción: Realiza un casting de un valor real a un booleano de la lógica trivalente, siempre y cuando el valor del real sea **0**, **1** ó **?**, en caso contrario se produce un error.

==	?	$n \in R$
?	T	?
$m \in R$	?	$m = n$

!=	?	$n \in R$
?	F	?
$m \in R$	?	$m \neq n$

>	?	$n \in R$
?	F	?
$m \in R$	?	$m > n$

<	?	$n \in R$
?	F	?
$m \in R$	?	$m < n$

<=	?	$n \in R$
?	T	?
$m \in R$	?	$m \leq n$

>=	?	$n \in R$
?	T	?
$m \in R$	?	$m \geq n$

Figura 6 – Comportamiento de los operadores relacionales en la clase *Real*

Además de los métodos definidos, se cuenta con distintas funciones que operan sobre números reales, las cuales son invocadas durante la evaluación de una regla, y tienen una asociación directa con alguna función del lenguaje que permite dar comportamiento a las celdas. En todos los casos, si estas funciones reciben como parámetro un valor indefinido, devolverán automáticamente el valor indefinido. Estas funciones se detallan en el *Apéndice B*.

7.2 TvalBool

La clase *TValBool* representa los valores y operaciones para la lógica trivalente utilizadas en el lenguaje. Las constantes de clase son:

TValBool::ttrue Representa al valor *Verdadero.*
TValBool::tfalse Representa al valor *Falso*.
TValBool::tundef Representa al valor *Indefinido*.

Internamente, el valor de cada constante booleana se almacena como un entero, siendo:
0 cuando es *Falso*.
1 cuando es *Verdadero*.
-1 cuando es *Indefinido*.

Métodos principales:

TValBool
Signatura: `TValBool()`
Descripción: Constructor. Devuelve un booleano cuyo valor es indefinido.

TValBool
Signatura: `TValBool(int)`
Descripción: Constructor. Devuelve un booleano de acuerdo al valor entero pasado como parámetro (0 = Falso, 1 = Verdadero, -1 = Indefinido).

TValBool
Signatura: `TValBool(char)`
Descripción: Constructor. Devuelve un booleano de acuerdo al carácter pasado como parámetro. Si el carácter pertenece al conjunto { 'T', 't', '1' } su valor será verdadero. Si el carácter pertenece al conjunto { 'F', 'f', '0' } su valor será falso. Sino será indefinido.

operator ==
Signatura: `bool operator == ( const TValBool & ) const`
Descripción: Devuelve True si los valores booleanos son iguales. En otro caso devuelve False.

xor
Signatura: `TValBool xor( const TValBool & ) const`
Descripción: Realiza una operación XOR (O Exclusivo) entre valores booleanos de la lógica trivalente. El comportamiento de esta operación se muestra en la Figura 7.

imp
Signatura: `TValBool imp( const TValBool & ) const`
Descripción: Realiza una operación IMP (Implicación Lógica) entre valores booleanos de la lógica trivalente. El comportamiento de esta operación se muestra en la Figura 7.

eqv
Signatura: `TValBool eqv( const TValBool & ) const`
Descripción: Realiza una operación EQV (Equivalencia) entre valores booleanos de la lógica trivalente. El comportamiento de esta operación se muestra en la Figura 7.

operator &&
Signatura: `TValBool operator && ( const TValBool & ) const`
Descripción: Aplica el operador lógico AND, cuyo comportamiento se muestra en la Figura 7.

operator ||
Signatura: `TValBool operator || ( const TValBool & ) const`
Descripción: Aplica el operador lógico OR, cuyo comportamiento se muestra en la Figura 7.

operator !
Signatura: `TValBool operator !() const`
Descripción: Aplica el operador lógico NOT, cuyo comportamiento se muestra en la Figura 7.

asChar

Signatura: `char asChar() const`

Descripción: Devuelve un carácter, dependiendo del valor del booleano. Si su valor es verdadero, devuelve '**T**', si es falso devuelve '**F**' y si es indefinido devuelve '**?**'.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

XOR	T	F	?
T	F	T	?
F	T	F	?
?	?	?	?

IMP	T	F	?
T	T	F	?
F	T	T	T
?	T	?	?

EQV	T	F	?
T	T	F	F
F	F	T	F
?	F	F	T

Figura 7 – Comportamiento de los operadores para la lógica trivalente

7.3 SyntaxNode

La clase *SyntaxNode* es la clase base abstracta que permite describir un nodo del árbol de evaluación de una regla. El chequeo de tipos se realiza utilizando el mismo árbol, ya que cada construcción conoce el tipo que deben tener sus parámetros. La construcción del árbol está implícita dentro del código de la función *yyparse* a partir de la especificación de la gramática con la sintaxis del **yacc**. En la sección 7.21 se detalla el mecanismo de evaluación para las reglas a partir del árbol de evaluación construido.

SyntaxNode
SpecNode
RuleNode
ConstantNode
VarNode
AbsCellPosNode
TimeNode
CountNode
OpNode
PortRefNode
StringNode
SendPortNode
UnaryOpNode
BinaryOpNode
ThreeOpNode
FourOpNode

Figura 8 – Jerarquía de Clases para los Nodos del Árbol que representa las Reglas

Métodos principales:

evaluate

Signatura: `virtual Real evaluate() = 0`

Descripción: Devuelve el valor de la evaluación del nodo, expresado como un *Real*. Incluso los valores de la lógica trivalente son devuelto como reales, ya que el valor ***True*** puede expresarse como 1, *False* como 0, y **?** (indefinido) como –1. Este método se encuentra sobrecargado por las subclases de *SyntaxNode*.

checkType

Signatura: `virtual bool checkType() const = 0`

Descripción: Realiza un chequeo de tipos para el nodo. Este método se encuentra sobrecargado por las subclases de *SyntaxNode*.

name

Signatura: `virtual const string name() = 0`

Descripción: Devuelve el nombre del nodo. Este método se encuentra sobrecargado por las subclases de *SyntaxNode*.

print

Signatura: `virtual ostream &print( ostream & ) = 0`

Descripción: Muestra información del nodo por el stream indicado. Este método se encuentra sobrecargado por las subclases de *SyntaxNode*.

7.4 SpecNode

La clase *SpecNode* extiende a la clase *SyntaxNode.* Representa una lista de reglas. Contiene métodos para la incorporación de reglas y para la búsqueda de una regla válida. Esta clase implementa a la función de cómputo local.

Métodos principales:

evaluate

Signatura: `Real evaluate(bool anyUndefined, bool anyStochast)`

Descripción: Itera sobre todas las reglas de la lista, en búsqueda de alguna válida. Dependiendo del modo de ejecución, el simulador busca la primer regla que evalúa a verdadero ó, además, si el modelo no es estocástico, valida que no exista otra que pueda ser satisfecha. Si se produce un error de este tipo se genera una **InvalidEvaluation**. Si dos o más reglas son satisfechas y el modelo es estocástico, se produce un warning, pero la simulación no es interrumpida.
En caso de que la lista represente una función de cómputo local definida mediante la cláusula *portInTransition* de *N-CD++* y que no tenga reglas satisfacibles, si para la misma se definió una función alternativa mediante la cláusula *else* se procede a la evaluación de la misma en búsqueda de alguna regla válida.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*SpecNode*".

addRule

Signatura: `SpecNode &addRule( SyntaxNode *rule, int StochasticCondition )`

Descripción: Agrega una regla al final de la lista, junto a un valor que indica si la regla contiene o no alguna función aleatoria.

value

Signatura: `Real value() const`

Descripción: Devuelve el valor asociado a la regla que fue satisfecha.

delay

Signatura: `Real delay() const`

Descripción: Devuelve la demora asociada a la regla que fue satisfecha.

elseFunction

Signatura: `const string elseFunction() const`

Signatura: `void elseFunction( const string ef )`

Descripción: Asigna o devuelve el nombre de la función de cómputo local que será utilizada en caso de que ninguna de las reglas de la lista sean satisfechas. La asignación de esta función se corresponde al uso de la cláusula *else* en *N-CD++*, la cual solo esta permitida para las funciones definidas con *portInTransition*.

7.5 RuleNode

La clase *RuleNode* es una subclase de *SyntaxNode.* Representa una regla de la especificación. Dentro de sus componentes se encuentran las expresiones correspondientes al valor y la demora que debe tomar la celda, y la expresión lógica trivalente asociada que representa la condición de la misma.

Métodos principales:

RuleNode

Signatura: `RuleNode( SyntaxNode *value = NULL, SyntaxNode *delay = NULL, SyntaxNode *condition = NULL )`

Descripción: Constructor de la clase. Genera un nuevo nodo, conteniendo un subárbol con la expresión para el valor de la regla, otro subárbol con la expresión para la demora, y un tercer subárbol con la condición de la regla.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve la evaluación del subárbol correspondiente a la condición de la regla. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

value

Signatura: `Real value()`

Descripción: Devuelve la evaluación del subárbol correspondiente al valor de la regla, que será usada como valor de la celda.

delay

Signatura: `Real delay()`

Descripción: Devuelve la evaluación del subárbol correspondiente a la demora de la regla, que será usada como demora de la celda.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*RuleNode*".

7.6 VarNode

La clase *VarNode* es una especialización de la clase *SyntaxNode* y permite almacenar una referencia a una celda vecina, definida como un desplazamiento con respecto a la celda origen del vecindario. *VarNode* contiene una variable de tipo *nTupla* para almacenar la referencia a un vecino.

El nombre de esta clase se desprende de *CD++*, donde las referencias a las celdas del vecindario eran las únicas variables posibles en la especificación de las reglas. Si bien en *N–CD++* los valores obtenidos por los puertos de entrada también son considerados como variables en el lenguaje, el nombre de la clase no ha sido alterado.

Métodos principales:

VarNode

Signatura: `VarNode( nTupla nt )`

Descripción: Constructor de la clase. Genera un nuevo nodo, conteniendo la tupla indicada.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve el valor de la celda vecina correspondiente. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*VarRef*".

7.7 ConstantNode

La clase *ConstantNode* es una especialización de la clase *SyntaxNode* y permite almacenar una constante numérica cuyo dominio pertenece a los reales, los enteros o es un valor booleano perteneciente a la lógica trivalente usada. Para el caso en que se referencia a una constante simbólica definida en *N-CD++* (*T*, *F*, ***?***, *Pi*, *Inf*, *e*, *Planck*, etc.) se agrega al árbol que conforma la representación de una regla un nodo de tipo *ConstantNode* con el valor numérico de dicha variable.

Métodos principales:

ConstantNode

Signatura: `ConstantNode( Real n, const TypeValue &t )`

Descripción: Constructor de la clase. Genera un nuevo nodo, conteniendo un valor constante del tipo indicado. El tipo indica puede ser real, booleano de la lógica trivalente, o entero. Dicho valor siempre se codifica como un *Real*.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve el valor de la constante, expresado como un *Real*. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*Constant*".

7.8 PortDefNode

La clase *PortDefNode* es una especialización de la clase *SyntaxNode* y permite definir una referencia a un puerto de entrada de la celda, devolviendo el valor del último mensaje arribado por el mismo.

Métodos principales:

PortRefNode

Signatura: `PortRefNode( SyntaxNode *x = NULL )`

Descripción: Constructor de la clase. Genera un nuevo nodo, conteniendo un subnodo con el nombre del puerto.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve el valor del último mensaje arribado al puerto asociado. Este puerto asociado es de la clase *StringNode*. Si el nombre del puerto es "*thisport*", se utilizará el puerto por el cual arribó el mensaje. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*PortRef*".

7.9 StringNode

La clase *StringNode* es una especialización de la clase *SyntaxNode* y permite almacenar una cadena de caracteres que representa el nombre de un puerto de la celda, que será usado al evaluarse las funciones *PortValue* y *send* del lenguaje de especificación de reglas.

Métodos principales:

StringNode

Signatura: `StringNode( string n )`

Descripción: Constructor de la clase. Genera un nuevo nodo, conteniendo un string que representa el nombre de un puerto.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve el valor 0 como un *Real*.

getString

Signatura: `string getString()`

Descripción: Devuelve el string que almacena el nodo. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el string en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*String*".

7.10 SendPortNode

La clase *SendPortNode* es una especialización de la clase *SyntaxNode* y permite el envío de un valor, proveniente de la evaluación de una expresión, a través de un puerto de salida de la celda.

Métodos principales:

SendPortNode

Signatura: `SendPortNode( SyntaxNode *x = NULL, SyntaxNode *y = NULL )`

Descripción: Constructor de la clase. Genera un nuevo nodo conteniendo dos subnodos: el primero representa el nombre del puerto, y el segundo la expresión de la que saldrá el valor a ser enviado.

evaluate

Signatura: `Real evaluate()`

Descripción: Evalúa el subnodo que representa al nombre del puerto, para obtener el mismo, y luego evalúa el nodo que representa la expresión. Posteriormente envía el valor de la evaluación de la expresión, a través del puerto correspondiente de la celda, con la hora de simulación igual a la hora actual. Esta función devuelve el valor 0 como un *Real*, de acuerdo al comportamiento de la instrucción **send** del lenguaje de definición de reglas de *N-CD++*. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*SendPort*".

7.11 TimeNode

La clase *TimeNode* es una especialización de la clase *SyntaxNode* y permite obtener el tiempo de simulación, utilizado por la función *Time* del lenguaje de especificación de reglas.

Métodos principales:

TimeNode

Signatura: `TimeNode()`

Descripción: Constructor de la clase. Genera un nuevo nodo que será utilizado para obtener la hora actual de simulación.

evaluate

Signatura: `Real evaluate()`

Descripción: Devuelve la hora actual de simulación. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*TimeNode*".

7.12 AbsCellPosNode

La clase *AbsCellPosNode* es una especialización de la clase *SyntaxNode* y permite obtener parte de la posición de la celda que esta ejecutando la función de cómputo local.

Métodos principales:

AbsCellPosNode

Signatura: `AbsCellPosNode(SyntaxNode *x)`

Descripción: Constructor de la clase. Genera un nuevo nodo que contiene un subnodo con una constante que tendrá la posición dentro de la tupla usada para referenciar a una celda y que será devuelta.

evaluate

Signatura: `Real evaluate()`

Descripción: Evalúa el subnodo para obtener el índice que será usado para acceder al elemento dentro de la tupla que referencia a la celda en el modelo acoplado. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*AbsCellPosNode*".

7.13 CountNode

La clase *CountNode* es una especialización de la clase *SyntaxNode* y permite obtener la cantidad de celdas del vecindario que poseen determinado valor, la cual es utilizada por las funciones *TrueCount*, *FalseCount, UndefCount* y *StateCount*.

Métodos principales:

CountNode

Signatura: `CountNode( const Real &v )`

Signatura: `CountNode( SyntaxNode * &s )`
Descripción: Constructores de la clase. Generan un nuevo nodo que será utilizado para obtener la cantidad de vecinos con el estado indicado. Es posible indicar un valor real, o una expresión la cual será evaluada para obtener un valor.

evaluate
Signatura: `Real evaluate()`
Descripción: Devuelve la cantidad de vecinos que poseen determinado valor de estado. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name
Signatura: `const string name()`
Descripción: Devuelve el string "*CountNode*".

7.14 OpNode

La clase *OpNode* es una clase abstracta derivada de la clase *SyntaxNode*. Es usada para representar a las funciones con uno o más parámetros. Tiene como subclases a: *UnaryOpNode*, *BinaryOpNode*, *ThreeOpNode* y *FourOpNode*, que representan a las funciones con uno, dos, tres o cuatro parámetros respectivamente.

Métodos principales:

name
Signatura: `const string name()`
Descripción: Devuelve el string "*Operation*".

7.15 UnaryOpNode

La clase *UnaryOpNode* es una especialización de la clase *OpNode*. Esta clase representa la clase base abstracta paramétrica para los nodos que representan funciones y operaciones unarias. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>. La clase dispone de las variables de instancia *node* para almacenar un puntero al subárbol que contiene la expresión que representa el parámetro y *op* para almacenar la operación.

Métodos principales:

UnaryOpNode
Signatura: `UnaryOpNode( SyntaxNode *n = NULL )`
Descripción: Constructor de la clase. Genera un nuevo nodo que contiene un nodo con el subárbol que representa la expresión que al ser evaluada será utilizada como parámetro de la operación unaria.

evaluate
Signatura: `Real evaluate()`
Descripción: Evalúa la expresión asociada, y al valor obtenido le aplica la operación especificada. Devuelve el resultado de la operación. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name
Signatura: `const string name()`
Descripción: Devuelve el string "*UnaryOp*".

child
Signatura: `UnaryOpNode &child( SyntaxNode *n )`
Signatura: `SyntaxNode *child()`

Descripción: Devuelve o establece el subárbol que representa el parámetro.

7.16 BinaryOpNode

La clase *BinaryOpNode* es una especialización de la clase *OpNode*. Esta clase representa la clase base abstracta paramétrica para los nodos que representan funciones y operaciones binarias. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>. La clase dispone de las variables de instancia *l* y *r* (nombres provenientes de *left* y *right*) para almacenar un puntero a los subárboles que contienen las expresiones que representan los parámetros y *op* para almacenar la operación.

Métodos principales:

BinaryOpNode

Signatura: `BinaryOpNode(SyntaxNode *left = NULL, SyntaxNode *right =NULL)`

Descripción: Constructor de la clase. Genera un nuevo nodo que contiene dos nodos con los subárboles que representan las expresiones que al ser evaluadas serán utilizadas como parámetros de la operación binaria.

evaluate

Signatura: `Real evaluate()`

Descripción: Evalúa las expresiones asociadas, y a los valores obtenidos le aplica la operación especificada. Devuelve el resultado de la operación. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

La evaluación de los nodos que representan los parámetros de la operación se realiza primero para el nodo de la izquierda y luego para el de la derecha (según estén almacenados en la variable de instancia *l* ó *r*). Para optimizar los tiempos de la evaluación de las reglas y mejorar así la performance del simulador, N-CD++ incorpora una mejora sobre su predecesor, CD++. La misma consiste en optimizar la evaluación de ciertas operaciones binarias booleanas, de forma tal que no siempre sea necesario evaluar los dos nodos para conocer el resultado de la operación. Esto implicó la creación de una función de evaluación que conozca información adicional sobre las operaciones de la lógica booleana trivalente. De esta forma, en algunos casos es posible conocer el resultado que tendrá la operación sólo con evaluar el nodo de la izquierda, evitando la evaluación del nodo derecho. Si de esa forma no es posible determinar el resultado de la operación, se procede a evaluar el nodo derecho y a ejecutar la operación con los valores obtenidos las respectivas evaluaciones.

En la Figura 9 se detallan las optimizaciones que realiza *N-CD++*, podando el árbol de evaluación y mejorando los tiempos de ejecución.

Sin embargo, cuando se encuentra activo el modo de debug para la evaluación de las reglas, como el objetivo es mostrar la evaluación completa de todo el árbol para ser útil al usuario a modo de debugging, no se aplican las optimizaciones.

Evaluación del Nodo Izquierdo	Operación	Resultado de la Operación
FALSO	AND	FALSO
VERDADERO	OR	VERDADERO
INDEFINIDO	XOR	INDEFINIDO
FALSO	IMP	VERDADERO

Figura 9 – Optimización en la evaluación de las operaciones binarias de la lógica trivalente

name

Signatura: `const string name()`

Descripción: Devuelve el string "*BinaryOp*".

left

Signatura: `BinaryOpNode &left( SyntaxNode *n )`

Signatura: `SyntaxNode *left()`
Descripción: Devuelve o establece el subárbol que representa el primer parámetro de la operación.

right
Signatura: `BinaryOpNode &right( SyntaxNode *n )`
Signatura: `SyntaxNode *right()`
Descripción: Devuelve o establece el subárbol que representa el segundo parámetro de la operación.

7.17 ThreeOpNode

La clase *ThreeOpNode* es una especialización de la clase *OpNode*. Esta clase representa la clase base abstracta paramétrica para los nodos que representan funciones ternarias. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>. La clase dispone de las variables de instancia *c1*, *c2* y *c3* para almacenar un puntero a los subárboles que contienen las expresiones que representan los parámetros y *op* para almacenar la operación.

Métodos principales:

ThreeOpNode
Signatura:
```
ThreeOpNode(SyntaxNode *child1 = NULL,
        SyntaxNode *child2 = NULL, SyntaxNode *child3 = NULL)
```
Descripción: Constructor de la clase. Genera un nuevo nodo que contiene tres nodos con los subárboles que representan las expresiones que al ser evaluadas serán utilizadas como parámetros de la operación ternaria.

evaluate
Signatura: `Real evaluate()`
Descripción: Evalúa las expresiones almacenadas en las variables de instancia *c1*, *c2* y *c3*, y a los valores obtenidos le aplica la operación especificada. Devuelve el resultado de la operación. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name
Signatura: `const string name()`
Descripción: Devuelve el string "*ThreeOp*".

child1
child2
child3
Signatura: `ThreeOpNode &child1( SyntaxNode *n )`
Signatura: `ThreeOpNode &child2( SyntaxNode *n )`
Signatura: `ThreeOpNode &child3( SyntaxNode *n )`
Signatura: `SyntaxNode *child1()`
Signatura: `SyntaxNode *child2()`
Signatura: `SyntaxNode *child3()`
Descripción: Devuelve o establece el subárbol que representa el primer, segundo o tercer parámetro de la operación.

7.18 FourOpNode

La clase *FourOpNode* es una especialización de la clase *OpNode*. Esta clase representa la clase base abstracta paramétrica para los nodos que representan funciones que tienen cuatro parámetros. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>. La clase dispone de las variables de instancia *c1*, *c2*, *c3* y *c4* para almacenar un puntero a los subárboles que contienen las expresiones que representan los parámetros y *op* para almacenar la operación.

Métodos principales:

FourOpNode

Signatura: `FourOpNode(SyntaxNode *child1=NULL, SyntaxNode *child2 = NULL, SyntaxNode *child3 = NULL, SyntaxNode *child4 = NULL )`

Descripción: Constructor de la clase. Genera un nuevo nodo que contiene cuatro nodos con los subárboles que representan las expresiones que al ser evaluadas serán utilizadas como parámetros de la operación.

evaluate

Signatura: `Real evaluate()`

Descripción: Evalúa las expresiones almacenadas en las variables de instancia *c1*, *c2*, *c3* y *c4*, y a los valores obtenidos le aplica la operación especificada. Devuelve el resultado de la operación. Si se encuentra activo el modo de debug para la evaluación de las reglas, imprime el resultado de dicha evaluación en el stream asociado al modo.

name

Signatura: `const string name()`

Descripción: Devuelve el string "*FourOp*".

child1
child2
child3
child4

Signatura: `FourOpNode &child1( SyntaxNode *n )`
Signatura: `FourOpNode &child2( SyntaxNode *n )`
Signatura: `FourOpNode &child3( SyntaxNode *n )`
Signatura: `FourOpNode &child4( SyntaxNode *n )`
Signatura: `SyntaxNode *child1()`
Signatura: `SyntaxNode *child2()`
Signatura: `SyntaxNode *child3()`
Signatura: `SyntaxNode *child4()`

Descripción: Devuelve o establece el subárbol que representa el primer, segundo, tercer o cuarto parámetro de la operación.

7.19 Nodos para la definición de Funciones y Operadores del Lenguaje

Las operaciones y funciones aprovechan las distintas clases paramétricas para su definición. Los parámetros de estas clases indican:

- el nombre de la función paramétrica que implementa su comportamiento.
- el tipo de valores tomado como parámetro.
- el tipo de valor devuelto por la función u operación.

El listado de este tipo de nodos se describe en el *Apéndice A*.

7.20 Parser

La clase *Parser* permite generar una estructura capaz de ser evaluada en tiempo de ejecución a partir de una especificación. Dicha clase utiliza el árbol de evaluación generado por la herramienta **yacc**, quien realiza una análisis léxico de las reglas y lo traduce a una estructura capaz de ser analizada fácilmente por el simulador. Como es la única clase capaz de realizar esta tarea existe una única instancia llamada *SingleParser*.

Mediante el uso del parámetro **–p** en la invocación al simulador se muestra información adicional obtenida durante la etapa del parseo de las reglas de los modelos celulares. Dicho parámetro permite que la salida generada pueda ser enviada a un archivo o la salida estándar, resultando de utilidad para hallar errores sintácticos en la escritura de reglas muy complejas.

Funciones principales:

yyparse

Signatura: `int yyparse()`

Descripción: Función provista por el **yacc** para ejecutar el parseo.

Métodos principales:

parse

Signatura:
```
Parser &parse( istream &source, bool printParserInformation,
       bool parseForPortIn, const string &elseFunction = "" )
```

Descripción: Toma la cadena pasada como parámetro bajo el nombre de *source*, y llama a la función **yyparse** para generar una lista de árboles que representan a las reglas. El resto de los parámetros indican: si se desea imprimir información adicional durante la etapa del parseo (establecido mediante un parámetro en la invocación al simulador) y si se está parseando una regla perteneciente a una función definida con la cláusula *portInTransition*, para la cual puede existir una función alternativa definida mediante la cláusula *else*.

specification

Signatura: `SpecNode *specification()`

Descripción: Devuelve el código interpretable de la última especificación analizada.

nextToken

Signatura: `int nextToken()`

Descripción: Devuelve el próximo valor léxico, el cual es usado por la función *yylex* del **yacc**.

7.21 LocalTransAdmin

Para evaluar las reglas que definen el comportamiento de una celda, la especificación de las mismas debe traducirse a una especificación ejecutable. Como la mayoría de las celdas poseen la misma función de cálculo local, el código se asocia a un identificador de función, y cada celda registra el identificador que le corresponde. Para esto se dispone de una tabla que contiene los pares (Identificador, Lista de Reglas). Durante la etapa en que se carga la descripción de los modelos a simular, al leerse la definición de una función de transición que no esta registrada en la tabla previamente descripta, se crea una nueva entrada en la misma, donde el nombre de la función actúa como identificador, y cada una de las reglas se representa con una tupla (*valor*, *demora*, *condición*), donde cada elemento de esta tupla tiene una estructura con forma de árbol.

De esta forma es posible representar mediante una estructura de árbol a cada una de las reglas que definen el comportamiento de una celda. Por ejemplo, la representación de la primer regla del *Juego de la Vida* (Figura 10) puede verse en la Figura 11.

```
Rule: 1 10 { (0,0) = 1 and ( truecount = 3 or truecount = 4 ) }
Rule: 1 10 { (0,0) = 0 and truecount = 3 }
Rule: 0 10 { t }
```

Figura 10 – Reglas para el *Juego de la Vida*

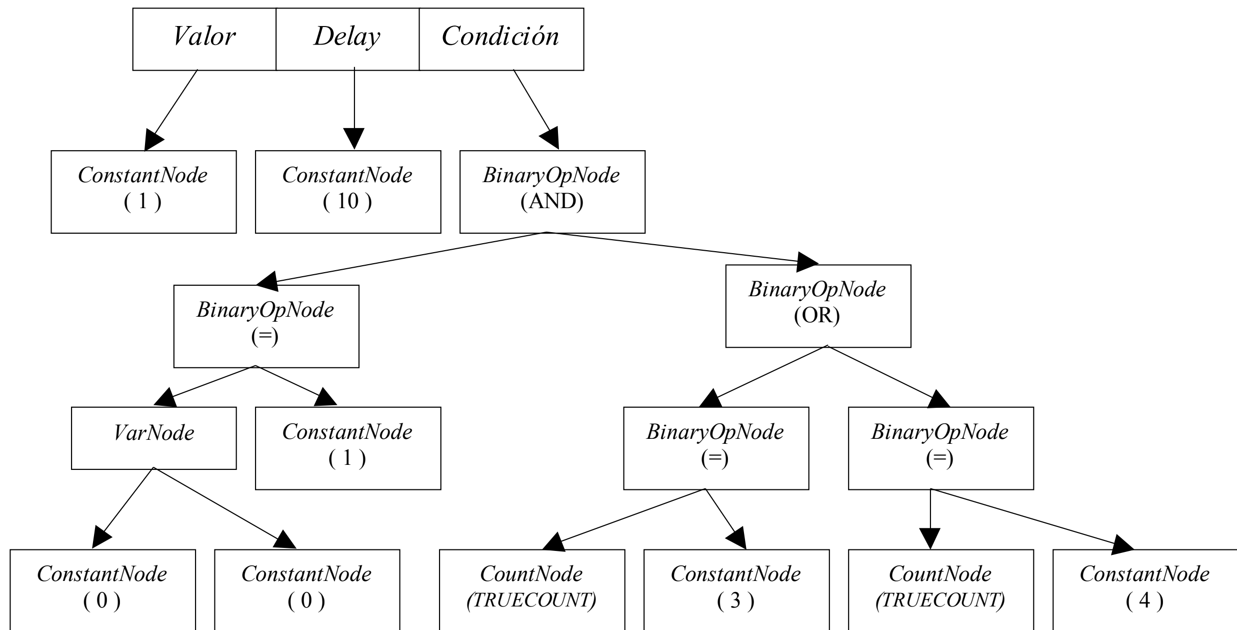


Figura 11 – Estructura de Arbol que Representa la Primer Regla del *Juego de la Vida*

De esta forma, para evaluar una regla, primero se evalúa en forma recursiva el árbol que representa la condición de la misma. Si el resultado de la evaluación es favorable, se procede a evaluar los árboles correspondientes al valor y a la demora, y los resultados de tales evaluaciones serán los valores tomados en cuenta por la celda.

La clase *LocalTransAdmin* es la encargada de registrar este conjunto de reglas. Para ello utiliza la clase *Parser* para obtener una tupla (*valor*, *demora*, *condición*), representando una regla capaz de ser evaluada posteriormente. Para almacenar los valores de esta tupla se utiliza la clase *RuleNode*. Cuando llega un evento externo a la celda debe utilizarse la función local para obtener el valor y la demora asociados a la regla que es satisfecha. La evaluación de las reglas y la devolución de los resultados también es efectuada por la clase *LocalTransAdmin*. Como es la única clase capaz de realizar esta tarea existe una única instancia llamada *SingleLocalTransitionAdmin*.

Métodos principales:

evaluate

Signatura:

```
Real evaluate( const Function &, const NeighborhoodValue &,
      PortValues *, Time &delay , Time &actualtime,
      VirtualPortList *outPorts, Model *actualCell,
      string portSource = "" )
```

Descripción: Evalúa la función de cómputo local especificada como parámetro, utilizando todos los datos que influyen sobre la celda, los cuales están compuestos por el vecindario y el valor de los mensajes arribados por los distintos puertos de entrada de la celda.
Devuelve el valor que pasará a ser el nuevo estado de la celda, y en la variable *delay* devuelve la demora para la misma.
Debido a que las funciones de cómputo local son genéricas a modo de ser utilizadas por varias celdas, al método *evaluate* debe indicársele otros valores particulares de la celda para la cual esta siendo evaluada, con el fin de disponer de los datos necesarios para la ejecución de ciertas funciones provistas por el lenguaje de especificación de reglas. En el momento de la invocación del método *evaluate*, el parámetro *actualTime* posee la hora actual de simulación. La misma solo es usada en el caso de que se requiera obtener el tiempo de simulación mediante la función *time*.
El parámetro *outPorts* es una lista de los puertos de salida de la celda que esta evaluando la función de cómputo local. Solo es utilizada en caso de encontrarse una función *send* en el lenguaje de especificación de reglas. En ese caso se debe enviar un valor por un puerto de la celda, usando el método *sendOutput* de la misma. Para ello se hace uso del parámetro

actualCell, que es un puntero al modelo atómico (celda atómica o atómica achatada) que será usada usará para la invocación del método *sendOutput*.
Si el parámetro *portSource* es un string vacío, significa que la función de cómputo local que se desea evaluar es una función definida mediante la cláusula *portInTransition*, y por lo tanto el valor de *portSource* contiene el nombre del puerto por el cual arribó el mensaje.

Mediante el uso del parámetro **–v** en la invocación del simulador, toda evaluación de las reglas de los modelos celulares mostrará paso a paso los resultados de las evaluaciones de las funciones y operadores que la componen, sin utilizar la optimización de los mismos. Dicho parámetro permite especificar el lugar donde se mostrarán los resultados, pudiendo ser un archivo o a la salida estándar.

registerTransition

Signatura:

```
LocalTransAdmin &registerTransition(
      const LocalTransAdmin::Function &transName,
      istream &transText, bool printParserInfo,
      bool parseForPortIn, const string &elseFunction = "" )
```

Descripción: Registra una nueva función de cómputo local. El primer parámetro indica el nombre de la función y el segundo de donde debe tomarse la especificación para realizar su parseo.
Si el parámetro booleano *printParserInfo* tiene el valor verdadero, el parseo de la función generará información detallada del mismo.
El parámetro *parseForPortIn* indica si se desea parsear para la creación de una función definida mediante la cláusula *portInTransition*, realizando validaciones adicionales, como por ejemplo no permitir el uso de la función *portValue* si el valor de *parseForPortIn* es falso.
El parámetro *elseFunction* permite asociar una función alternativa en caso de que ninguna de las reglas de una función definida mediante la cláusula *portInTransition* sea válida.
Si el nombre de la función ya existe se produce una **InvalidTransId**. Si existe algún error sintáctico al parsear una regla se genera una **ErrorParsingException**.

cellValue

Signatura: `const Real &cellValue( const NeighborPosition & )`

Descripción: Permite obtener el valor de una celda dentro del vecindario especificado.

portValue

Signatura: `const Real &portValue(const string portName)`

Descripción: Devuelve el valor asociado a un puerto de entrada de la celda.

8 Soporte para modelos n–dimensionales

CD++ almacena internamente el espacio celular de dimensión (d_1, d_2) sobre un vector de $d_1 \cdot d_2$ elementos, donde el elemento (x_1, x_2), con $x_i \in [0, d_i -1]$, ocupa la posición $x_1 + x_2 \cdot d_1$. Análogamente, *N-CD++* usa un arreglo de $\prod_{i=1...n} d_i$ elementos para almacenar el autómata celular de dimensión (d_1, d_2, ..., d_n), y en este caso el elemento (x_1, x_2, ..., x_n) ocupa la posición:

$$\sum_{i=1...n} x_i \cdot \left(\prod_{k=1...i-1} d_k \right)$$

En *CD++*, cada referencia a una celda o a alguno de sus vecinos se almacena como un par ordenado (fila, columna). En la nueva versión, y debido a que la dimensión del espacio celular depende del modelo que se esté interpretando, se debió crear un tipo de dato, al cual se lo llamó *nTupla*, que almacena valores numéricos enteros en un orden arbitrario. De esta forma es posible almacenar una posición dentro del espacio celular de dimensión *n*, usando una *nTupla* que almacene exactamente *n* valores.

En *CD++* el vecindario de una celda se almacena en un vector que siempre tiene 9 posiciones, dado que esa es la máxima cantidad de elementos que puede tener un vecindario bidimensional con celdas adyacentes a la celda de origen.

En *N–CD++*, debido a que la cantidad de celdas pertenecientes al vecindario depende de la dimensión con la cual se esté trabajando y que, además, es posible definir un vecino de una celda que no necesariamente sea adyacente a la misma, no es posible determinar de antemano la cantidad de celdas del vecindario. Debido a esto se modificó la estructura que almacena el esquema del vecindario, reemplazando el vector por una lista dinámica de celdas.

8.1 nTupla

La clase *nTupla* es la encargada de implementar la referencia a las celdas de un modelo celular n-dimensional, junto a todas las operaciones sobre las mismas. Esta compuesta de un vector de *n* posiciones, donde en cada ubicación se almacena un valor entero.

En el resto de los módulos, una referencia a la celda se hace a través de un *CellPosition*, el cual es un tipo de datos análogo a la *nTupla*, ya que fue creado mediante la directiva *typedef* de *C*, por lo que dispone de los mismos métodos que ésta.

Métodos principales:

nTupla
- **Signatura**: `nTupla()`
- **Descripción**: Construye una nueva tupla, de dimensión 0.

nTupla
- **Signatura**: `nTupla(const nTupla &pos)`
- **Descripción**: Construye una nueva tupla, con igual dimensión y valores que la indicada como parámetro.

nTupla
- **Signatura**: `nTupla(const string &str)`
- **Descripción**: Construye una nueva tupla, parseando los valores de un string con el formato: “$(x_1, x_2, ..., x_n)$”.

nTupla
- **Signatura**: `nTupla(unsigned dim, int value)`
- **Descripción**: Construye una nueva tupla conteniendo *dim* elementos con el valor indicado.

dimension
- **Signatura**: `unsigned dimension() const`
- **Descripción**: Devuelve la cantidad de elementos de la tupla.

operator ==
- **Signatura**: `bool operator == ( const nTupla & ) const`
- **Descripción**: Devuelve *true* si dos tuplas de igual dimensión son iguales coordenada a coordenada, en otro caso devuelve *false*.

operator <
- **Signatura**: `bool operator <  ( const nTupla & ) const`
- **Descripción**: Devuelve *true* si una tupla es menor a otra y ambas tienen igual dimensión. En otro caso devuelve *false*. El orden de las tuplas esta dado por la definición recursiva mostrada en la Figura 3, aunque la implementación del operador < no sea en forma recursiva.

operator =
- **Signatura**: `nTupla  &operator = (const nTupla & )`
- **Signatura**: `nTupla  &operator = (const string &cellStr)`
- **Descripción**: Asigna los valores y la dimensión de otra tupla (o de un string) a la celda en cuestión.

operator +=
- **Signatura**: `nTupla  *operator += ( const nTupla & )`

Descripción: Suma de tuplas de igual dimensión coordenada a coordenada.

operator -=

Signatura: `nTupla  *operator -= ( const nTupla & )`

Descripción: Resta de tuplas de igual dimensión coordenada a coordenada.

add

Signatura: `void add(int n)`

Descripción: Agrega un elemento a la tupla, incrementando la dimensión de la misma en 1, y ubicando al valor *n* en la última ubicación.

addFirst

Signatura: `nTupla *addFirst(int n)`

Descripción: Agrega un elemento a la tupla, incrementando la dimensión de la misma en 1, y ubicando al valor *n* en la primer ubicación, desplazando el resto de los valores un lugar.

setElement

Signatura: `void setElement(unsigned pos, int n)`

Descripción: Establece el valor *n* en la posición indicada dentro de la tupla, agregando elementos con valor 0 en caso de que dicha posición sea mayor a la dimensión.

set

Signatura: `void set(unsigned n, int value)`

Descripción: Agrega a la tupla *n* elementos con el valor indicado.

fill

Signatura: `void fill(unsigned n, int value)`

Descripción: Establece los primeros *n* elementos de la tupla con el valor indicado.

get

Signatura: `int get(unsigned n) const`

Descripción: Devuelve el elemento de la tupla ubicado en la posición *n*, considerando que el primer elemento se encuentra en la posición 0.

contains

Signatura: `bool contains( int elem )`

Descripción: Devuelve *true* si la tupla contiene el elemento indicado, sino devuelve *false*.

canonizar

Signatura: `nTupla &canonizar( nTupla &dim )`

Descripción: Aplica módulos a todos los elementos de la tupla, considerando los valores de la dimensión. Este método es de gran utilidad para la referencia a celdas de los modelos celulares toroidales.

print

Signatura: `string print() const`

Descripción: Devuelve un string con el formato “(x_1, x_2, ..., x_n)” con los valores de la tupla.

includes

Signatura: `bool includes( const nTupla &p ) const`

Descripción: Devuelve *true* si la tupla que invocó al método contiene a la tupla *p*. Para esto se analizan los valores de ambas tuplas coordenada a coordenada: si el valor de la posición *i* de la tupla *p* es mayor al valor de la posición *i* de la tupla que invocó el método, o si el valor de la posición *i* de la tupla *p* es menor a 0, se dice que la tupla *p* no esta contenida en la otra tupla.

calculateIndex

Signatura: `long calculateIndex( const nTupla &dim,`

`bool validateIncludes = true ) const`

Descripción: Permite calcular la posición correspondiente a la tupla que invocó al método, dentro de un vector capaz de almacenar los valores para todas las celdas de un modelo con dimensión *dim*, aplicando para esto la fórmula definida en la sección 8.

totalElements

Signatura: `long totalElements() const`

Descripción: Devuelve la cantidad de elementos que debe tener un vector para almacenar los valores para todas las celdas de un modelo n-dimensional cuya dimensión es la tupla que invocó al método.

minCoordToCood

Signatura: `void minCoordToCoord( nTupla &t1, nTupla &t2 )`

Descripción: Calcula el mínimo coordenada a coordenada entre dos tuplas, y deja el resultado en la tupla que invocó al método.

maxCoordToCood

Signatura: `void maxCoordToCoord( nTupla &t1, nTupla &t2 )`

Descripción: Calcula el máximo coordenada a coordenada entre dos tuplas, y deja el resultado en la tupla que invocó al método.

next

Signatura: `bool next( const nTupla &dim )`

Descripción: Calcula la siguiente tupla, incrementando en 1 la ultima posición y considerando la dimensión para hacer los acarreos necesarios. Devuelve *true* si la tupla calculada supera a la dimensión, o *false* en caso en que la suma pudo realizarse con éxito.
En CD++, como la dimensión de los modelos celulares siempre era 2, varios métodos implementaban dos ciclos for anidados para poder generar todas las posiciones de las celdas, como se muestra en la Figura 12.

```
int     row, col;

for (row = 0; row < height; row++)
    for (col = 0; col < width; col++)
    {
        ...
        Realiza cierta operación con la posición (row, col)
        ..
    }
```

Figura 12 – Recorrido sobre todas las celdas en *CD++*

En *N-CD++,* como la dimensión del modelo no es fija, no es posible escribir una cantidad determinada de ciclos for anidados. Para poder recorrer todas las posiciones de un espacio celular se creó el método *next*, que permite obtener la próxima posición de celda e indica cuando se alcanzó la ultima posición accesible de acuerdo a la dimensión del modelo. En la Figura 13 se ejemplifica su uso.

```
nTupla counter( dim, 0);    // Crea la posición inicial: (0, 0, ..., 0)
bool   overflow = false;    // Overflow indica si se terminó de recorrer el espacio
                            // celular.
while (!overflow)
{
    ...
    Realiza cierta operación con la posición counter
    ...
    ...
```

```
    overflow = counter.next( dimension );   // Incrementa la posición en 1 para
                                            // la siguiente iteración.
                                            // Aquí la variable dimension es una
                                            // nTupla que contiene la dimensión
                                            // del modelo celular.
}
```

Figura 13 – Recorrido sobre todas las celdas en *N-CD++*

Por ejemplo, si se tiene un autómata celular de tamaño (2, 3, 4), para recorrer todas sus celdas se establece inicialmente una *nTupla*, que llamaremos *counter*, con valor (0, 0, 0). Al aplicarle el método *next* a *counter* (usando la dimensión del autómata celular como parámetro) se obtiene la próxima celda: (0, 0, 1) que consiste en sumar 1 a su última coordenada. Las siguientes celdas obtenidas son (0, 0, 2) y (0, 0, 3). Pero luego, *next* genera la celda (0, 1, 0) debido a que se produce un acarreo, pues la última coordenada de la dimensión es 4, y por lo tanto la última coordenada de las celdas debe estar comprendida entre 0 a 3. Las próximas celdas obtenidas son (0, 1, 1), (0, 1, 2), (0, 1, 3), (0, 2, 0), (0, 2, 1), (0, 2, 2), (0, 2, 3). Considerando el último valor obtenido, cuando se aplique el método next, incrementando en 1 el valor de la última coordenada, se produce un doble acarreo y la celda obtenida es (1, 0, 0). De esta forma se pueden seguir obteniendo nuevas celdas hasta llegar a (1, 2, 3), la última celda generada. Si se aplica el método next a esta última celda, se devuelve el valor *True*, indicando que se produjo un desborde y que todas las referencias a celdas han sido generadas.

8.2 CellState

La clase *CellState* es la encargada de almacenar y administrar el estado de las celdas en un autómata celular n-dimensional, el cual puede ser toroidal o no.

Métodos principales:

CellState

Signatura: `CellState( nTupla &dim, bool wrapped = false )`

Descripción: Construye un nuevo estado de celdas, según la dimensión dada. Si el parámetro *wrapped* es *true*, se indica que el espacio de celdas debe ser toroidal.

operator []

Signatura: `Real &operator[]( CellPosition & )`

Descripción: Devuelve el elemento indicado por la posición dada como parámetro. Si el espacio de celdas no es toroidal y la posición esta fuera de dimensión del espacio, se devuelve el valor indefinido.

includes

Signatura: `bool includes( const CellPosition &pos ) const`

Descripción: Devuelve *true* si la posición indicada se encuentra dentro del espacio de celdas. En caso de tratarse de un estado de celdas toroidal siempre se devolverá *true*.

isWrapped

Signatura: `bool isWrapped() const`

Descripción: Devuelve *true* si el espacio celular es toroidal, sino devuelve *false*.

dimension

Signatura: `nTupla &dimension()`

Descripción: Devuelve la tupla que contiene la dimensión del espacio celular.

calcRealPos

Signatura: `void calcRealPos( CellPosition &pos )`

Descripción: Si el espacio es toroidal, modifica la posición de la celda pasada como parámetro, de tal forma que la misma siempre este incluida en la dimensión del espacio. Para eso, se invoca al método *canonizar* de la posición indicada, con la dimensión del espacio celular como parámetro.

print

Signatura: `void CellState::print( ostream &os, char undefChar ='?' ) const`

Descripción: Muestra el espacio celular por el stream indicado, usando el carácter *undefChar* para representar al valor indefinido, que por default es '**?**'. Este método solo debe ser usado cuando la dimensión del autómata celular es 2 ó 3.

printFormatedList

Signatura: `void printFormatedList( ostream & ) const`

Descripción: Análogo a print, pero solo debe ser usado cuando la dimensión es mayor a 3. En este caso la salida generada consiste de una lista con todas las posiciones de las celdas contenidas en el espacio celular y su respectivo valor.

8.3 NeighborhoodValue

La clase *NeighborhoodValue* permite representar el vecindario de una celda. Sus responsabilidades son la administración del vecindario y la actualización del valor de estado de su celda origen. Un vecindario se define como una lista de desplazamientos de celdas con respecto a la celda de origen. Cada desplazamiento tiene asociado un puntero a un valor del espacio de celdas (*CellState*). De esta forma, actualizando el valor de la celda origen del vecindario, todas las celdas que tengan en su vecindario a esa celda actualizada obtendrán los cambios automáticamente.

En *CD++*, el vecindario se almacenaba en un arreglo de 9 posiciones, que representa una matriz de 3x3. Esta es la máxima cantidad de celdas que puede tener un vecindario que solo puede estar compuesto de celdas adyacentes a la celda de origen en un modelo celular bidimensional. En *N-CD++*, el vecindario no puede ser almacenado en un arreglo de tamaño fijo debido a que al variar la dimensión del modelo varia el tamaño del arreglo y, además, el vecindario puede ser definido con celdas no necesariamente adyacentes a la celda de origen. Debido a esto se optó por hacer que el vecindario sea una lista de posiciones a celdas, referenciando a cada vecino.

Métodos principales:

NeighborhoodValue

Signatura:
```
NeighborhoodValue( CellState &mat,
                   const CellPositionList &neighbors,
                   const CellPosition &center)
```

Descripción: Construye un nuevo vecindario. Sus parámetros indican la lista de desplazamientos que define al vecindario, la posición de la celda que será el origen del mismo, y el estado de celdas que será utilizado para que cada vecino apunte a un valor del mismo.

set

Signatura: `NeighborhoodValue & set( const Real &v )`

Descripción: Establece un nuevo valor para el estado de la celda origen del vecindario.

get

Signatura: `const Real &get() const`

Descripción: Devuelve el valor de la celda origen del vecindario.

get

Signatura: `const Real &get( const NeighborPosition &n )  const`

Descripción: Devuelve el valor de una celda dentro del vecindario determinada por un desplazamiento indicado por parámetro.

9 Clases para el Inicio de la Simulación

9.1 Clase MainSimulator

La clase *MainSimulator* es responsable de la creación, a partir de una especificación, del árbol de modelos y procesadores junto a los vínculos entre sus puertos. Para realizar esta tarea toma la especificación obtenida por el cargador y utiliza la clase *Ini* para realizar el análisis de la especificación de los modelos. Una vez finalizada la construcción del entorno de simulación, la instancia de *MainSimulator* informa a la instancia de *RootCoordinator* los eventos externos especificados y la hora a la cual debe detener la simulación, que por defecto tiene valor infinito. Luego se invoca el método *run()* de *RootCoordinator* para que comience la simulación. La misma finaliza cuando llega la hora de terminación o todos los modelos que componen la simulación se encuentran en estado pasivo y no se esperan nuevos eventos externos.

Métodos principales:

run

Signatura: `MainSimulator &run()`

Descripción: Lleva a cabo todo el ciclo de simulación. Se encarga de la carga y creación de los modelos, los eventos externos y de establecer la hora de finalización de la simulación; posteriormente da comienzo a la simulación invocando al método *simulate* de la clase *Root*. Luego solicita al cargador que envíe los resultados.

loadModels

Signatura: `MainSimulator &loadModels( istream&, bool printParserInfo )`

Descripción: Invoca al método *parse* de la clase *Ini* con los datos obtenidos del cargador. Luego invoca al método *loadModel*, para crear en forma jerárquica todos los modelos y procesadores que integran la simulación, comenzando por *Root*.
Mediante sus parámetros es posible indicar el stream de donde obtener la definición de los modelos, pudiendo indicar si se desea mostrar información adicional durante la interpretación de los mismos.

loadModel

Signatura:
```
MainSimulator &loadModel( Coupled &parent, Ini &ini,
                          bool printParserInfo )
```

Descripción: Invoca al método *loadPorts* para la creación de puertos del modelo acoplado indicado. Luego, dependiendo del tipo de modelo, se invoca a *loadCells* o *loadComponents*. *PrintParserInfo* indica si se desea mostrar información adicional durante la interpretación de las reglas de los modelos celulares. Por último se invoca al método *loadLinks* para la creación de las influencias internas y externas.
La obtención de todos los datos necesarios se hace a través de la instancia de la clase *Ini* pasada como parámetro.

loadPorts

Signatura: `MainSimulator &loadPorts( Coupled &parent, Ini &ini )`

Descripción: Crea los puertos definidos en el archivo de definición de modelos, para el modelo acoplado indicado.

loadComponents

Signatura:
```
MainSimulator &loadComponents( Coupled &parent, Ini &ini,
                               bool printParserInfo)
```

Descripción: Crea los modelos que integran al modelo acoplado indicado. Para los modelos acoplados hace una llamada recursiva a *loadModel*.

loadCells

Signatura: `MainSimulator &loadCells( CoupledCell &parent, Ini &ini,`

```
bool printParserInfo)
```

Descripción: Obtiene de *ini* los datos para la configuración del modelo celular, como la dimensión, el tipo de demora (inercial o de transporte), si el modelo es toroidal o no, la demora por defecto, el vecindario y la función de cómputo local. Luego invoca al método *createCell* del parámetro *parent* (que debe pertenecer a la clase *CoupledCell* o *FlatCoupledCell*), con el fin de crear todas las celdas. Posteriormente invoca al método *loadInitialCellValues* para establecer el valor inicial para las celdas y luego invoca a *loadLocalZones* para establecer las funciones de cómputo alternativas para determinadas celdas.

loadLinks

Signatura:
```
MainSimulator &loadLinks( Coupled &parent, Ini &ini )
```
Descripción: Crea los enlaces internos entre puertos de los modelos que integran al modelo acoplado especificado como parámetro.

loadInitialCellValues

Signatura:
```
MainSimulator &loadInitialCellValues( CoupledCell &, Ini & )
```
Descripción: Establece los valores iniciales para las celdas del modelo celular especificado como parámetro, según fue determinado en el archivo de definición de modelos.

loadInitialCellValuesFromFile

Signatura:
```
MainSimulator &loadInitialCellValuesFromFile(
                CoupledCell &parent, const string &fileName )
```
Descripción: Establece los valores iniciales para las celdas del modelo celular pero a diferencia de *loadInitialCellValues*, los valores son obtenidos de un archivo de tipo .VAL. El formato para este tipo de archivos puede encontrarse en el Manual del Usuario.

loadInitialCellValuesFromMapFile

Signatura:
```
MainSimulator &loadInitialCellValuesFromMapFile(
                CoupledCell &parent, const string &fileName )
```
Descripción: Establece los valores iniciales para las celdas del modelo celular pero a diferencia de *loadInitialCellValues* y *loadInitialCellValuesFromFile*, los valores son obtenidos de un archivo de tipo .MAP que contiene un mapa de valores para un modelo celular. El formato para este tipo de archivos puede encontrarse en el Manual del Usuario.

loadLocalZones

Signatura:
```
MainSimulator &loadLocalZones( CoupledCell &parent,
                    Ini &init, bool printParserInfo )
```
Descripción: Registra para cada celda perteneciente a la zona definida en el archivo de configuración, la función de cómputo local especificada para dicha zona. Si *printParserInfo* tiene valor true, se mostrará información adicional durante el parseo de las funciones de cómputo local.

loadExternalEvents

Signatura:
```
MainSimulator &loadExternalEvents( istream& )
```
Descripción: Carga los eventos externos de la entrada especificada y se los agrega a la clase *Root*, quien se encarga de la administración de los mismos durante la simulación.

loadDefaultTransitions

Signatura:
```
MainSimulator &loadDefaultTransitions( CoupledCell &parent,
                    Ini &ini, bool printParserInfo )
```
Descripción: Obtiene el nombre de la función de cómputo local para el modelo acoplar especificado como parámetro y llama al método *registerTransition*.

loadPortInTransitions

Signatura:
```
MainSimulator &loadPortInTransitions( CoupledCell &parent,
                    Ini &ini, bool printParserInfo )
```
Descripción: Obtiene el nombre de las funciones de cómputo local especificadas con la cláusula *portInTransition* y las registra invocando al método *registerTransitionPortIn*.

registerTransition

Signatura:
```
MainSimulator &registerTransition(
        const LocalTransAdmin::Function &functionName,
        Ini &ini, bool printParserInfo )
```
Descripción: Dada un nombre de una función de cómputo local, se obtiene su definición del archivo de especificación y se invoca al método *registerTransition* de la clase *SingleLocalTransAdmin*, que se encarga de interpretarlo y almacenarlo en su base de funciones de cómputo local. Si *printParserInfo* tiene valor true, se mostrará información adicional durante el parseo de la función.

registerTransitionPortIn

Signatura:
```
MainSimulator & registerTransitionPortIn(
        const LocalTransAdmin::Function &fName, Ini &ini,
        bool printParserInfo, const string &elseFunction )
```
Descripción: Registra una función de cómputo local asociada al puerto de entrada de una celda.

9.2 Módulo Main

El módulo *Main* contiene la función *main* de C, que es la primer función en ser ejecutada al iniciarse cualquier programa escrito en este lenguaje. Primero se encarga de establecer un cargador. Si no se indica ningún parámetro por la línea de comandos, se activa el modo de servidor de simulación, por lo tanto el cargador a utilizar es *NetworkLoader*, de otra forma se utiliza el cargador *StandAloneLoader*. Luego se invoca al método *run* de la única instancia de la clase *MainSimulator*.

10 Clases Auxiliares

10.1 MacroExpansion

La clase *macroExpansion* permite realizar las expansiones de las macros y descarte de comentarios contenidos en el archivo de definición de modelos. Como es la única clase capaz de realizar esta tarea existe una única instancia llamada *instanceMacroExpansion*. La invocación de los métodos de esta clase es realizada por *StandAloneLoader* antes de inicializar la carga e interpretación de los modelos. Si en la invocación al simulador se utiliza el parámetro **–b**, se evita el uso del preprocesador y por lo tanto nunca se utiliza el módulo de macro expansiones.

Métodos principales:

macroExpansion

Signatura: `macroExpansion(string fileName)`

Descripción: Construye una nueva instancia del módulo de macro expansiones, que actuará sobre el archivo indicado.

expand

Signatura: `string expand()`

Descripción: Realiza la macro expansión del archivo especificado en el constructor de la clase. Este método genera un archivo temporario que será utilizado para contener los datos del archivo original, pero donde se expanden todas las invocaciones a macros por su definición, que pueden estar contenidos en otros archivos especificados por la cláusula *#Include*. Además, ignora los comentarios que el archivo original pueda tener. Posteriormente cierra el archivo temporario y el original, pero no borra ninguno de ellos.

tempFileName

Signatura: `string tempFileName()`

Descripción: Devuelve el nombre del archivo temporario creado por *expand* al realizar la expansión de macros y descarte de comentarios. Este archivo será utilizado para la carga de los modelos,

con lo que de aquí en adelante el resto de los módulos desconoce de la existencia de macros y comentarios existentes en el archivo original de definición de modelos.

~macroExpansion

Signatura: `~macroExpansion()`

Descripción: Destructor de la clase. Se encarga de borrar el archivo temporario creado por *expand*.

10.2 ClsEvalParam

La clase *clsEvalParam* permite identificar algunas de las opciones que pueden ser activadas mediante parámetros en la invocación al simulador, y en caso de que una opción se encuentre activa es posible asociarle un stream de salida. Las instancias existentes de esta clase son:

- *evalDebugInstance*: Indica si se encuentra activo el modo de debug que permite ver en detalle información sobre la evaluación de las reglas que definen el comportamiento de los modelos celulares.
- *flatDebugInstance*: Indica si se encuentra activo el modo de debug que permite ver el estado de los autómatas celulares achatados luego de cada avance de tiempo.
- *rulesDebugInstance*: Indica si se encuentra activo el modo de debug que valida la existencia de una única regla válida durante la ejecución de la función de cómputo local.
- *parserDebugInstance*: Indica si se encuentra activo el modo de debug que muestra información adicional en la etapa de parseo de las reglas que definen el comportamiento de un modelo celular.
- *showVirtualTimeDebugInstance*: Indica si se desea mostrar la hora de fin de simulación a través de stderr, una vez finalizada la misma.
- *useQuantumDebugInstance*: Indica si se desea usar un quantum durante la evaluación de la función de cómputo local.

Métodos principales:

stream

Signatura: `ostream &Stream()`

Signatura: `void Stream( ostream *evalParamStr )`

Descripción: Devuelve o establece el stream asociado.

Active

Signatura: `const bool Active()`

Signatura: `void Active( bool mode )`

Descripción: Devuelve o establece el estado de la opción.

Value

Signatura: `double Value( void )`

Signatura: `void Value( double valor )`

Descripción: Devuelve o establece un valor asociado a la opción.

10.3 Impresion

La clase *Impresion* permite la representación de valores reales formateados con distintas opciones. Esta clase cuenta con la variable *Default*, utilizada para la consulta de los valores numéricos generados por el resto de las clases.

Métodos principales:

Impresion

Signatura:
```
Impresion(int precision = 3, int width = 10,
            bool notCient = false, bool printZero = true)
```

Descripción: Constructor de la clase. Establece la precisión y el ancho que tendrá cada valor numérico, junto a condiciones que indican si los valores numéricos serán expresados usando notación

científica y si se desea imprimir el valor 0 (en caso de que esta última sea falsa, se genera el string vacío en el momento de producirse una salida).

Precision

Signatura: `int Precision()`
Signatura: `int Precision(int p)`
Descripción: Devuelve o establece la cantidad de caracteres establecidas para la precisión, usados para la representación de los valores numéricos.

Width

Signatura: `int Width()`
Signatura: `int Width(int p)`
Descripción: Devuelve o establece la cantidad de caracteres establecidos como ancho para la representación de los valores numéricos.

PrintZero

Signatura: `bool PrintZero()`
Signatura: `bool PrintZero(bool pz)`
Descripción: Devuelve o establece el valor de la condición que indica si se mostrará o no el valor cero en la representación de los valores numéricos.

10.4 Zone

La clase *Zone* permite representar un rango n-dimensional de celdas y es utilizada para dar un comportamiento particular a ciertas celdas de un modelo celular, pudiéndole asociar una función de cómputo local distinta que al resto de las celdas. Dispone de métodos para la obtención de las celdas pertenecientes al rango definido.

10.5 RealPrecision

La clase *RealPrecision* administra el valor de tolerancia utilizado al realizar las comparaciones de valores de la clase *Real*. Dispone de métodos para la obtención y modificación de dicho valor. Por defecto, si se construye una instancia de esta nueva clase sin especificar un valor de tolerancia, se asume que la misma tiene el valor $1x10^{-8}$.

10.6 MyList

La clase *MyList* representa una lista dinámica encadenada que contiene los pares (*CellPosition*, *Real*). Posee métodos para limpiar la lista, agregar un elemento al final de la misma, recorrer la lista mediante el uso de cursores, así como para buscar y obtener valores almacenados en ella.

10.7 Otras Clases

Existen otras clases utilizadas en *N-CD++* que fueron heredadas de su predecesor, *CD++* y que no han sufrido cambios. Estas clases se detallan a continuación y la descripción de sus principales métodos puede encontrarse en [BB98]:

- **Time**: Representa una hora, la cual esta compuesta por horas, minutos, segundos y milésimas de segundos. Tiene constantes para representar las horas cero e infinito. Además, dispone de métodos para comparación, suma y resta de horas.
- **Ini**: Representa un archivo de texto separado en secciones, donde cada sección a su vez puede contener definición de nombres de variables y valores. Esta clase es utilizada para la lectura del archivo de especificación del modelo a simular. Dispone de métodos para realizar un análisis sintáctico de un archivo de este tipo, y para obtener los valores para determinado grupo y definición.
- **Distribution**: Clase abstracta utilizada para representar las distintas distribuciones aleatorias.
- **ChiDistribution, NormalDistribution, ConstantDistribution, PoissonDistribution, ExponentialDistribution**: Subclases de *Distribution* que implementan distintas distribuciones aleatorias.
- **MException**: Clase abstracta que implementa el mecanismo de excepciones.

- **AssertException, InvalidMessageException, InvalidModelIdException, InvalidProcessorIdException, InvalidAtomicTypeException, InvalidPortRequest, ZoneException**: subclases de *MException* que representan distintas excepciones.
- **Port**: Representa un puerto de un modelo, junto a operaciones para operar sobre el mismo. Contiene una lista de modelos influenciados, un identificador, un nombre y el identificador del modelo al cual pertenece.

También existen las siguientes estructuras y tipos de datos:

- **Event**: Es una estructura que representa a un evento. Almacena la hora, el puerto y el valor del mismo; y dispone de operaciones de comparación.
- **ModelId**: Es un tipo de dato compuesto por un número entero. Representa el identificador de un modelo.
- **PortId**: Es otro tipo de dato, también compuesto por un número entero. Representa el identificador de un puerto.

11 Simulación en N–CD++

Esta sección describe el proceso de simulación realizado por *N–CD++* en sus distintas etapas. La primer fase del simulador consiste en la carga de los modelos. Según el modo en que se esté invocando al simulador (modo servidor o pasaje de parámetros a través de la línea de comandos) se instancia la correspondiente subclase de *SimLoader* y se le indica a la única instancia de *MainSimulator* que cargador utilizar. Éste, mediante el uso del cargador, obtiene la configuración del simulador y crea la jerarquía de modelos y procesadores y los eventos externos, dejando listo el entorno para dar comienzo a la simulación. Posteriormente le indica al coordinador raíz la hora de finalización y da comienzo a la simulación invocando el método *simulate*().

La simulación comienza con el envío de un mensaje de inicialización al coordinador del modelo acoplado de mayor nivel (*TOP*). Este modelo reenvía el mensaje a todos sus hijos y espera un mensaje *done* de cada uno de ellos con la hora de su próximo evento. Una vez recibidos todos los mensajes, calcula cual de todos sus hijos es el inminente, es decir el que tenga menor hora de próximo evento, y envía un mensaje *done* a su padre indicando su hora de próximo evento interno. Cuando el coordinador raíz recibe un mensaje *done* toma la hora del mensaje como la hora del próximo cambio de estado del modelo acoplado de mayor nivel, y la compara con el próximo evento externo que tiene programado, quedándose con el menor entre ambos. Si se trata de un evento externo enviará un *ExternalMessage* y si es interno enviará un *InternalMessage*, actualizando la hora global del simulador.

Si la planificación del hijo inminente propone como hora de próximo cambio infinito, significa que todos los modelos se han pasivado. En este caso, si no existen eventos externos se finaliza la simulación. La otra posibilidad de finalización ocurre cuando la hora del próximo evento elegido es mayor o igual a la hora de finalización especificada.

Cuando un coordinador recibe un mensaje *ExternalMessage* conteniendo el puerto por el cual fue originado, analiza el mapeo de las influencias sobre el puerto destino del mensaje utilizando para esto la lista de influencias que posee su modelo acoplado asociado. Por cada influencia se genera y envía un nuevo mensaje externo con el valor del mensaje original. Posteriormente, el coordinador registra la cantidad de mensajes *done* que debe esperar para enviar un nuevo mensaje *done* a su coordinador padre con la hora del próximo cambio de estado.

La llegada de un *InternalMessage* al coordinador del modelo acoplado de mayor nivel proveniente del coordinador raíz indica el arribo de un evento interno. Esto representa un cambio de estado programado para alguno de sus hijos. El coordinador del modelo acoplado de mayor nivel lo redirecciona hacia su hijo inminente. Este hijo es potencialmente cualquier especialización de *Processor*: *Simulator, Coordinator, CellCoordinator* o *FlatCoordinator*.
Cuando una instancia de la clase *Simulator* recibe un mensaje interno invoca a los métodos *outputFunction()* e *internalFunction()* de su modelo atómico asociado. Al retornar de este último método genera un *DoneMessage* indicando la hora de su próximo cambio de estado y lo envía a su padre. Si el modelo atómico genera una salida, ésta será enviada por su simulador asociado a su coordinador padre.
Cuando una instancia de la clase *Coordinator* o alguna de sus derivadas recibe un evento interno lo redirecciona hacia su hijo inminente. Cuando reciba el *DoneMessage* calculará la hora de su hijo inminente y la enviará en un mensaje *done* hacia su procesador padre.

Cuando un coordinador recibe un OutputMessage, representando un mensaje de salida, debe analizar las influencias del puerto por el cual se produjo la salida. Si la influencia tiene destino en un modelo externo al coordinador debe reenviarse el mensaje de salida hacia el coordinador padre. Si en cambio, el destino para la influencia es un modelo

dentro del coordinador, el mensaje de salida es traducido a un *ExternalMessage* y enviado por el puerto que indique la influencia. Por cada mensaje externo se espera su mensaje *done* asociado. Cuando arriben todos, el procesador retornará un mensaje *done* a su coordinador padre con la hora del hijo inminente.

Los modelos celulares acoplados manejan a las celdas como modelos hijos. En la creación de las celdas se le asigna el comportamiento especificado a cada una y se la vincula con los modelos según su vecindario, los cuales se conectan mediante los puertos *Out–NeighborChange*, como se muestra en el ejemplo de la Figura 14. La clase *CellCoordinator* da un comportamiento especial al tratamiento de los eventos internos y de salida. Frente a un mensaje interno se enviarán tantos *InternalMessage* como hijos inminentes existan, forzando así el orden en que los mensajes externos disparados por las salidas de las celdas serán procesados, evitando que una celda se active más de una vez en el mismo instante de tiempo. Para lograr esto el coordinador celular sobrecarga la recepción de los mensajes de salida guardando temporalmente las celdas ya afectadas para evitar los mensajes externos duplicados, que en los modelos celulares implican un recálculo del estado de la celda. Los mensajes externos varían la semántica cuando arriban por el puerto *NeighborChange*, provenientes de un vecino, ya que indican un aviso de recálculo y no un valor externo.

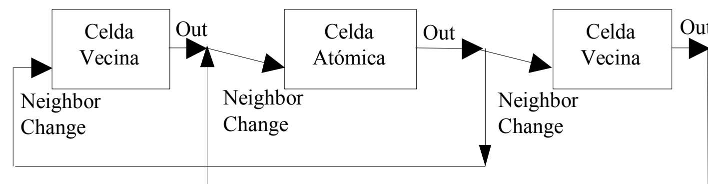


Figura 14 – Ejemplo de acoplamiento de una celda atómica con sus vecinos

Un modelo celular convencional está compuesto de tantos modelos atómicos como celdas posea el autómata. Esto produce una gran inversión de tiempo en la creación de los modelos y una sobrecarga de mensajes por cada evento que arriba al modelo celular. Como todas las celdas comparten la misma estructura, tienen el mismo comportamiento en función de su estado, y es posible obtener sus vecinos aplicando un desplazamiento sobre su ubicación, toda la estructura necesaria para llevar a cabo la simulación de este tipo de modelo puede simplificarse ser vista como una matriz de estados junto a una especificación del lenguaje a utilizar. Esta simplificación permite disminuir drásticamente el tiempo de creación y elimina por completo el pasaje de mensajes dentro del modelo celular. Para tal tarea se usan los modelos celulares achatados. Cuando una instancia de *FlatCoordinator* recibe un mensaje externo invoca al método *externalFunction* del acoplado celular achatado asociado para todas las celdas virtuales influenciadas por el puerto donde fue recibido el mensaje. El acoplado asociado evaluará la función de transición local para la posición indicada y aplicará la demora correspondiente, modificando la lista de próximos eventos en caso de ser necesario. Al finalizar se envía un mensaje *Done* al coordinador padre con la hora del hijo inminente.
En caso de recibir un *internalMessage*, el *FlatCoordinator* invoca al método *internalFunction* del acoplado celular achatado asociado, el cual itera sobre todas las celdas con hora de próximo evento igual a la hora actual, generando su salida y agregando todas las celdas influenciadas por la salida a la lista de próximos eventos. Para concluir, se ejecuta el método *externalFunction* para todas las celdas virtuales que figuren en esta lista.

12 Definición de modelos DEVS y Cell-DEVS en *N-CD*++

El archivo que permite definir los modelos usados por *N-CD++* esta compuesto por grupos de definiciones de modelos acoplados y configuración de modelos atómicos, esta última opcional. Cada definición indica el nombre del modelo (entre [ ]) y sus atributos. El grupo del modelo **[top]** es obligatorio y define el modelo acoplado de mayor nivel.

Uno de los objetivos de *N-CD++* fue mantener la compatibilidad con su predecesor, *CD++*, por lo que cualquier modelo definido para este último puede ser simulado en la nueva versión de la herramienta sin realizar adaptación alguna.

Para los modelos acoplados, es posible el uso de cuatro cláusulas para la definición de los mismos:

- **Components**: describe los modelos que integran el modelo acoplado. Su formato es:

nombre_de_modelo@nombre_de_clase

El orden en que se especifican los modelos determina la prioridad utilizada para el envío de mensajes, representando así a la función **select** del formalismo. El nombre de clase puede hacer referencia tanto a modelos atómicos como a modelos acoplados. Estos últimos deben estar descriptos en el mismo archivo de configuración como un nuevo grupo.

- **Out**: enumera los nombres de los puertos de salida del modelo acoplado. El uso de esta cláusula es opcional ya que un modelo puede no tener puertos de este tipo.

- **In**: enumera los nombres de los puertos de entrada del modelo acoplado. El uso de esta cláusula es opcional ya que un modelo puede no tener puertos de este tipo.

- **Link**: permite definir el esquema de acoplamiento entre los modelos que integran el modelo acoplado, y su posible conexión con los puertos de entrada y salida del mismo. El formato esta dado por el par:

puerto_origen[@*modelo*] *puerto_destino*[@*modelo*]

donde el nombre del modelo es opcional, ya que si no se indica se considera un puerto correspondiente al modelo acoplado en cuestión.

Los modelos atómicos deben definirse en lenguaje C++, y ser incluidos en la herramienta. Para el uso de los mismos, en el mismo archivo de especificación de modelos, debe configurarse las características que dependerán de la implementación de cada modelo, según haya sido especificado por el desarrollador. Cada instancia de un tipo de modelos atómico podrá ser configurada independientemente de otras instancias del mismo tipo.

Para la definición de modelos celulares, es posible el uso de las siguientes cláusulas:

- **Type**: [CELL | FLAT]
  Indica si el modelo celular será achatado o no. Si no se especifica se asume que es un modelo no achatado (*CELL*).

- **Width :** entero
  Permite definir la cantidad de columnas del modelo celular. Solo puede ser usada cuando se define un modelo celular unidimensional o bidimensional. El uso de *Width* implica necesariamente el uso de la cláusula *Height* para completar la definición de la dimensión del modelo, y prohibe el uso de la cláusula *Dim* en la definición del mismo.

- **Height :** entero
  Permite definir la cantidad de filas del modelo celular. Sólo puede ser usada cuando se define un modelo celular unidimensional o bidimensional, conteniendo el valor 1 para el primer caso. El uso de *Height* implica necesariamente el uso de la cláusula *Width* para completar la definición de la dimensión del modelo, y prohibe el uso de la cláusula *Dim* en la definición del mismo.

- **Dim :** $(x_0, x_1, ..., x_n)$
  Permite definir la dimensión de cualquier modelo celular.

- **Border :** [**WRAPPED** | **NOWRAPPED**]
  Indica si el modelo es o no toroidal. Por defecto toma el valor NOWRAPPED. Al usar un borde no toroidal, la referencia a una celda fuera del espacio celular retornará el valor indefinido (**?**).

- **Delay :** [**TRASPORT** | **INERTIAL**]
  Especifica el tipo de demora usada en cada celda. Por defecto toma el valor TRANSPORT.

- **DefaultDelayTime :** entero
  Demora por defecto para los eventos externos, expresada en milisegundos.

- **Neighbors :** *nombreCelular*($x_{1,1}, x_{2,1},...,x_{n,1}$)... *nombreCelular*($x_{1,m}, x_{2,m},...,x_{n.m}$)
  Define el vecindario para todas las celdas del modelo, donde cada celda ($x_{1,i}, x_{2,i},...,x_{n.i}$) representa un desplazamiento con respecto a la celda de origen del vecindario.

- **In :** puertos de entrada
  Análogo a la cláusula usada para los modelos acoplados. Esta cláusula es optativa.

- **Out :** puertos de salida
  Análogo a la cláusula usada para los modelos acoplados. Esta cláusula es optativa.

- **Link :** *puerto_origen*[*@modelo*] *puerto_destino*[*@modelo*]
  Igual que en los modelos acoplados, pero para hacer referencia a una celda debe usarse el nombre del modelo acoplado junto a la referencia a la celda, de la forma.
  Ejemplos: *Link* puertoSalida puertoEntrada@nombreCelular($x_1,x_2,...,x_n$)
  Link puertoSalida@nombreCelular($x_1,x_2,...,x_n$) puertoEntrada

- **Select :** *nombreCelular*($x_{1,1}, x_{2,1},...,x_{n,1}$)... *nombreCelular*($x_{1,m}, y_{2,m},...,k_{n.m}$)
  Representa la función ***select*** del formalismo, indicando las celdas que poseen prioridad sobre el resto. Las celdas no especificadas poseen la prioridad dictada por el orden de pares según su posición.

- **Initialvalue :** [*Real* | **?**]
  Representa el valor inicial para todo el espacio de celdas.

- **InitialRowValue :** $fila_i$ $valor_1$...$valor_{width}$
  Permite definir una lista de valores los cuales serán establecidos como iniciales para una fila de un modelo celular bidimensional. El valor definido en la posición j será usado para establecer el estado inicial de la celda (i, j) del modelo celular. Los valores se indican uno al lado del otro sin ningún carácter separador y deben pertenecer al conjunto {**?**, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

- **InitialRow :** $fila_i$ $valor_1$ ... $valor_{width}$
  Esta cláusula es semejante a **InitialRowValue**, solo que los valores se indican uno al lado del otro separados por un espacio en blanco, permitiendo el uso de cualquier valor perteneciente al conjunto $\Re \cup$ {**?**}

- **InitialCellsValue** : *fileName*
  Especifica el nombre de un archivo que contiene los valores iniciales de las celdas de un modelo celular n-dimensional.

- **InitialMapValue** : *fileName*
  Especifica el nombre de un archivo que contiene un mapa de valores que serán usados como estado inicial para un modelo celular.

- **LocalTransition :** *transitionFunctionName*
  Indica el nombre del grupo que contiene las reglas a utilizar para la función de transición local para todas las celdas.

- **PortInTransition :** *portName@ nombreCelular*($x_1, x_2,...,x_n$) *transitionFunctionName*
  Permite definir un comportamiento alternativo cuando arriba un mensaje externo por el puerto de entrada indicado de la celda ($x_1, x_2,...,x_n$) del modelo celular.

- **Zone :** *transitionFunctionName* { $rango_1$[..$rango_n$] }
  Permite definir un comportamiento alternativo para el conjunto de celdas comprendidas dentro del rango especificado. En el momento de calcular el nuevo estado para una celda, si dicha celda pertenece a algún rango se usará la función definida para tal, sino se utilizará la función de transición definida en la cláusula **LocalTransition**.

En la sección 14 se encuentran ejemplos de la definición de modelos DEVS y Cell-DEVS, junto al uso de las cláusulas previamente descriptas.

13 DrawLog

Mediante la herramienta *DrawLog* es posible representar gráficamente la actividad del simulador en cada instante de tiempo para los modelos celulares, utilizando para ello los datos registrados en el archivo de *log*. Este archivo es generado por *N-CD++*, y registra el flujo de mensajes entre los modelos que participan en la simulación.

DrawLog es capaz de generar tres tipos de salida, dependiendo de la dimensión del modelo celular a representar. Para los modelos de dos dimensiones, se generará una matriz bidimensional con los valores de estado en cada instante del tiempo simulado. Para los modelos tridimensionales, la representación consiste en una serie de matrices bidimensionales para cada instante de tiempo, donde la primer matriz representa el estado para todas las celdas de la forma (x, y, 0), la segunda representa el estado para las celdas de la forma (x, y, 1), y así hasta que se muestran todos los slices del modelo. Cuando los modelos tienen 4 o más dimensiones, se generará, para cada instante de tiempo, una representación consistente en un listado detallado de cada celda (representada por su posición dentro del modelo celular) y su respectivo valor.

El uso de modelos celulares achatados evita el envío de gran cantidad de mensajes dentro del mismo, y por lo tanto no es posible registrar en el archivo de log el envío de mensajes entre celdas. En este caso el *DrawLog* será incapaz de mostrar dichos resultados. Sin embargo, mediante el uso del parámetro **–f** en la invocación al simulador, es posible ver el estado de los modelos celulares achatados en cada instante de tiempo simulado, permitiendo indicar el destino donde serán mostrados dichos datos.

14 Ejemplos

El objetivo de esta sección es mostrar el uso de la herramienta a través de distintos ejemplos de aplicación, poniendo énfasis en las características propias de *N–CD++*.

14.1 Variante del Juego de la Vida

En este caso se muestra una variación del *Juego de la Vida* [Gar70], donde inicialmente se cuenta con una población compuesta por seres de dos tipos, representados por los valores 1 y 2. Los seres están dispersos en un área de 9 x 7 celdas, de las cuales algunas contienen el valor 0 indicando la ausencia de todo tipo de ser. Las reglas para la supervivencia y muerte de un individuo son análogas a las reglas del modelo original. El nacimiento de un nuevo ser se produce cuando la celda tiene tres seres vivos de cualquier tipo como vecinos. En ese caso el ser generado será una composición de los tres vecinos, y el valor que lo representará será el promedio de los valores de los vecinos.

En la Figura 15 se muestra la descripción del modelo en el lenguaje provisto por la herramienta.

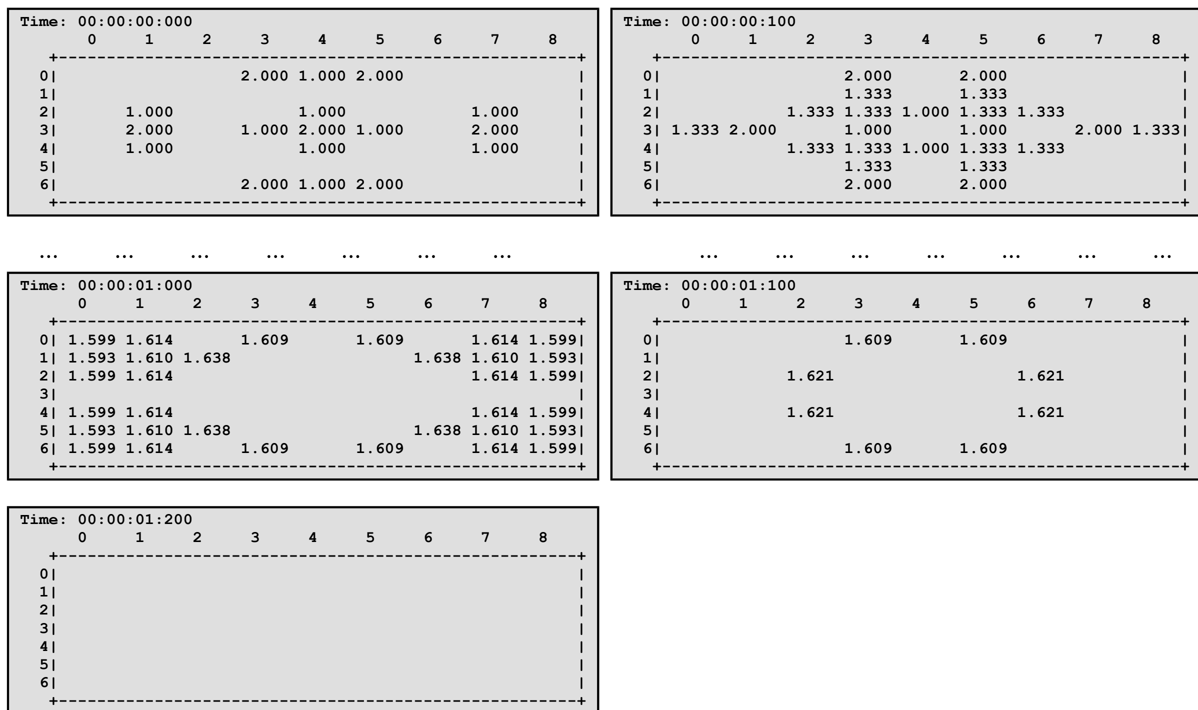
```
01      [top]
02      components : lifeExt
03
04      [lifeExt]
05      type : cell
06      width : 9
07      height : 7
08      delay : transport
09      defaultDelayTime  : 100
10      border : wrapped
11      neighbors : life(-1,-1) life(-1,0) life(-1,1)
12      neighbors : life(0,-1)  life(0,0)  life(0,1)
13      neighbors : life(1,-1)  life(1,0)  life(1,1)
14      initialvalue : 0
15      initialrowvalue :  0      000212000
16      initialrowvalue :  1      000000000
17      initialrowvalue :  2      010010010
```

```
18      initialrowvalue :  3      020121020
19      initialrowvalue :  4      010010010
20      initialrowvalue :  5      000000000
21      initialrowvalue :  6      000212000
22      localtransition : lifeExt-rule
23
24      [lifeExt-rule]
25      rule : {(0,0)} 100 {(0,0) != 0 and (9-falsecount = 3 or 9-falsecount = 4)}
26      Rule : { ((-1,-1) + (-1,0) + (-1,1) + (0,-1) + (0,1) + (1,-1) + (1,0) +
                 (1,1)) / 3} 100 { (0,0) = 0 and 9 - falsecount = 3 }
27      Rule : 0 100 { t }
```

Figura 15 – Implementación de la *Variante del Juego de la Vida*

Entre las líneas 1 y 3 se define el modelo acoplado de mayor nivel, compuesto en este caso por el modelo *lifeExt*. El resto de las líneas define los parámetros de este modelo. Entre las líneas 5 y 21 se define el tamaño del espacio de celdas, el tipo de demora, la forma del vecindario, y los valores iniciales para las celdas. En la línea 22 se especifica el nombre de la función de transición usada para calcular el valor de las celdas en cada etapa de la simulación. Esta función se define en las líneas 24 a 27.

El resumen de los resultados arrojados por la herramienta, redondeando los valores reales a 3 dígitos decimales, es el siguiente:



```
Time: 00:00:00:000
       0     1     2     3     4     5     6     7     8
   +------------------------------------------------------+
  0|                   2.000 1.000 2.000                  |
  1|                                                      |
  2|       1.000             1.000             1.000      |
  3|       2.000       1.000 2.000 1.000       2.000      |
  4|       1.000             1.000             1.000      |
  5|                                                      |
  6|                   2.000 1.000 2.000                  |
   +------------------------------------------------------+
```

```
Time: 00:00:00:100
       0     1     2     3     4     5     6     7     8
   +------------------------------------------------------+
  0|                   2.000       2.000                  |
  1|                   1.333       1.333                  |
  2|             1.333 1.333 1.000 1.333 1.333            |
  3| 1.333 2.000       1.000       1.000       2.000 1.333|
  4|             1.333 1.333 1.000 1.333 1.333            |
  5|                   1.333       1.333                  |
  6|                   2.000       2.000                  |
   +------------------------------------------------------+
```

... ...

```
Time: 00:00:01:000
       0     1     2     3     4     5     6     7     8
   +------------------------------------------------------+
  0| 1.599 1.614       1.609       1.609       1.614 1.599|
  1| 1.593 1.610 1.638                   1.638 1.610 1.593|
  2| 1.599 1.614                               1.614 1.599|
  3|                                                      |
  4| 1.599 1.614                               1.614 1.599|
  5| 1.593 1.610 1.638                   1.638 1.610 1.593|
  6| 1.599 1.614       1.609       1.609       1.614 1.599|
   +------------------------------------------------------+
```

```
Time: 00:00:01:100
       0     1     2     3     4     5     6     7     8
   +------------------------------------------------------+
  0|                   1.609       1.609                  |
  1|                                                      |
  2|             1.621                   1.621            |
  3|                                                      |
  4|             1.621                   1.621            |
  5|                                                      |
  6|                   1.609       1.609                  |
   +------------------------------------------------------+
```

```
Time: 00:00:01:200
       0     1     2     3     4     5     6     7     8
   +------------------------------------------------------+
  0|                                                      |
  1|                                                      |
  2|                                                      |
  3|                                                      |
  4|                                                      |
  5|                                                      |
  6|                                                      |
   +------------------------------------------------------+
```

Figura 16 – Resultados obtenidos para la variante del *Juego de la Vida*

Aquí puede apreciarse que inicialmente sólo existen seres de tipo 1 y 2, pero en los estados subsiguientes se produce la aparición de nuevos individuos, resultantes de la fusión de rasgos de los vecinos que posibilitaron su creación. La distribución inicial de los seres en el espacio de celdas hace que la población no sea estable, tendiendo a disminuir en cantidad, lo que finalmente sucede en el tiempo 00:00:01:200 con la extinción de los mismos.

14.2 Juego de la Vida en 3D

En este caso se muestra una variación del *Juego de la Vida* [Gar70], donde inicialmente se cuenta con una población compuesta por seres representados por el valor 1, dispersos en un área de 7 x 7 x 3 celdas, de las cuales

algunas contienen el valor 0 indicando la ausencia de cualquier tipo de ser. El nacimiento de un nuevo ser se produce cuando la celda tiene diez o más seres vivos como vecinos. Por otra parte, un ser permanecerá vivo mientras que su vecindario contenga 8 ó 10 seres vivos. En cualquier otro caso la celda contendrá el valor 0, debido a que ya estaba vacía o a que el ser que la ocupaba ha muerto.

Para este ejemplo se usará un vecindario de dimensión 3 x 3 x 3, según se muestra en la siguiente figura:

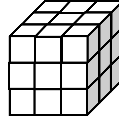


Figura 17 – Vecindario para el *Juego de la Vida* en 3D

La implementación del modelo en el lenguaje provisto por la herramienta se muestra en la Figura 18.

```
01      [top]
02      components : 3dl
03
04      [3dl]
05      type : cell
06      dim : (7,7,3)
07      delay : transport
08      defaultDelayTime  : 100
09      border : wrapped
10      neighbors: 3dl(-1,-1,-1) 3dl(-1,0,-1) 3dl(-1,1,-1)
11      neighbors: 3dl(0,-1,-1)  3dl(0,0,-1)  3dl(0,1,-1)
12      neighbors: 3dl(1,-1,-1)  3dl(1,0,-1)  3dl(1,1,-1)
13      neighbors: 3dl(-1,-1,0)  3dl(-1,0,0)  3dl(-1,1,0)
14      neighbors: 3dl(0,-1,0)   3dl(0,0,0)   3dl(0,1,0)
15      neighbors: 3dl(1,-1,0)   3dl(1,0,0)   3dl(1,1,0)
16      neighbors: 3dl(-1,-1,1)  3dl(-1,0,1)  3dl(-1,1,1)
17      neighbors: 3dl(0,-1,1)   3dl(0,0,1)   3dl(0,1,1)
18      neighbors: 3dl(1,-1,1)   3dl(1,0,1)   3dl(1,1,1)
19      initialvalue: 0
20      initialCellsValue: 3dl.val
21      localtransition: 3dl-rule
22
23      [3dl-rule]
24      rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
25      rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
23      rule : 0 100 { t }
```

Figura 18 – Implementación del *Juego de la Vida en 3D*

Las líneas 1 y 2 definen el modelo acoplado de mayor nivel, que en este caso esta compuesto por el autómata celular *3dl* (Three dimensional Life). Entre las líneas 4 y 19 se definen los parámetros para dicho autómata, como su dimensión, tipo de demora, y vecindario. La línea 20 define el archivo que contiene los valores iniciales para el modelo. La línea 21 contiene el nombre de la función de cómputo local que será usada para actualizar los valores de las celdas durante la simulación. Dicha función es definida entre las líneas 23 a 26.

Los resultados obtenidos en la ejecución del modelo son mostrados en la Figura 19. Cuando el modelo tiene tres dimensiones, donde su tamaño es especificado por la tupla (x, y, z), la herramienta automáticamente genera una salida que consta de una división del espacio en z planos de dimensión (x, y). Para este caso, la simulación comienza con un gran número de celdas activas, pero la distribución inicial hace que la población no sea estable, y el número de seres vivientes se reduzca hasta extinguirse, lo que sucede en el tiempo de simulación 00:00:01:000.

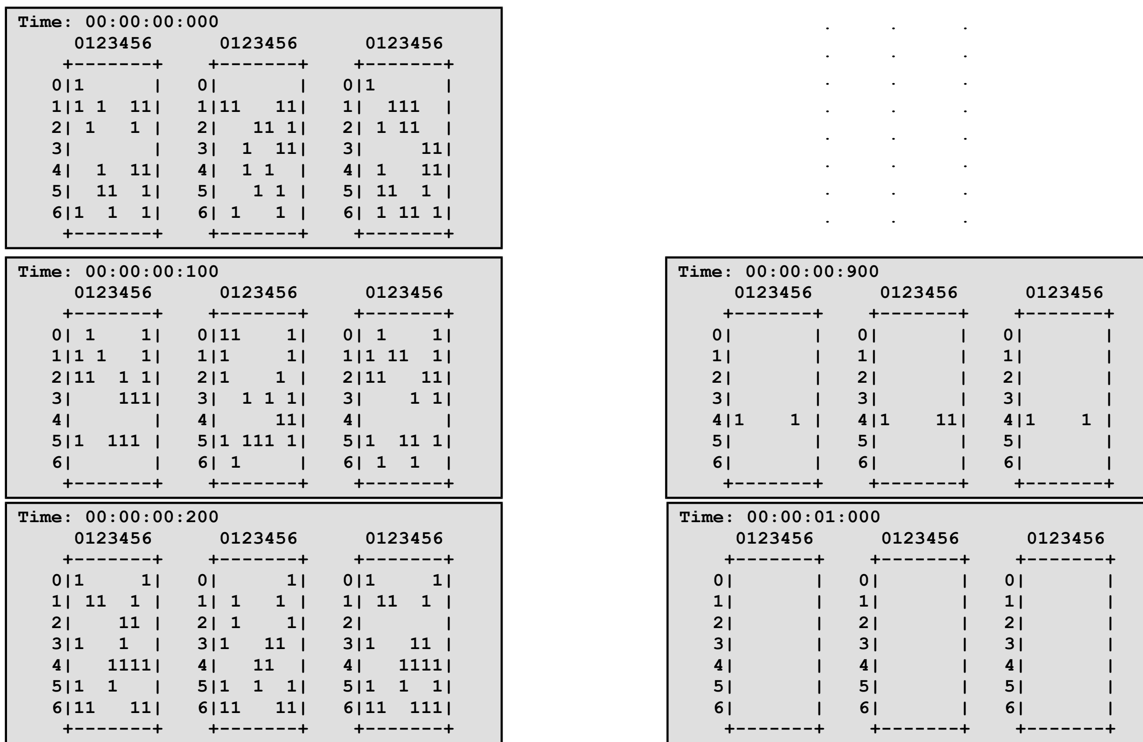


Figura 19 – Resumen de los resultados obtenidos durante la simulación del *Juego de la Vida en 3D*

14.3 Ordenamiento de Números Reales

Se mostrará un mecanismo de ordenamiento numérico sobre autómatas celulares, que actúa en forma semejante al algoritmo *bubble sort*. Se trabajará con un modelo celular no toroidal, por lo que la referencia a una celda fuera del autómata dará como resultado un valor indefinido (?), lo que servirá para determinar el límite de los valores a ordenar.

Inicialmente el autómata celular tiene los valores de acuerdo al esquema mostrado en la Figura 20.

X1	X2	X3	X4	...	Xn
1	?	1	?	...	s

Donde:

Xi es un valor numérico.
? es el valor indefinido.
s es 1 ó ? dependiendo de n, la cantidad de elementos a ordenar. Si n es par, entonces s = ?, sino s = 1.

Figura 20 – Esquema de Configuración Inicial para el Autómata de Ordenamiento Numérico

Por ejemplo, si se desea ordenar los 7 números: 4.1, 7.5, 14.3, -4.22, 2, 0.33 y 3.25 debe inicializarse un autómata celular de 2x7 celdas como se muestra en la Figura 21.

4.1	7.5	14.3	-4.22	2	0.33	3.25
1	?	1	?	1	?	1

Figura 21 – Ejemplo de la configuración inicial del Autómata de Ordenamiento Numérico

Los valores de la segunda fila indican contra quien se realizaran las comparaciones. Si debajo de uno de los valores a ordenar se encuentra un 1 significa que será comparado y ordenado con respecto al valor ubicado a su derecha. Por otra parte, si el valor es **?** significa que será ordenado con respecto al valor ubicado a su izquierda.

Por lo tanto, en el primer paso se ordenaran los pares (4.1, 7.5), (14.3, -4.22) y (2, 0.33). Como el último elemento (3.25) no tiene un valor a su derecha, no será ordenado en este paso.

Los pares ordenados son (4.1, 7.5), (-4.22, 14.3) y (0.33, 2). El autómata resultante luego del primer paso se muestra en la Figura 22.

4.1	7.5	-4.22	14.3	0.33	2	3.25
?	1	?	1	?	1	?

Figura 22 – Ejemplo del Autómata de Ordenamiento Numérico luego del Primer Paso

Nótese que los valores de la segunda fila han cambiado (los que eran **?** ahora son 1 y viceversa). Esto significa que en el próximo paso se crearan nuevos pares para las respectivas comparaciones. Estos pares son: (7.5, -4.22), (14.3, 0.33) y (2, 3.25). El primer elemento, como tiene debajo un **?**, deberá compararse con el elemento a su izquierda, pero como este es un **?** (pues se esta referenciando a una celda no definida por el autómata) significa que no deberá realizarse una ordenamiento para el elemento en este paso.

Los pares ordenados son (-4.22, 7.5), (0.33, 14.3) y (2, 3.25). El autómata resultante luego del segundo paso de ordenamiento se muestra en la Figura 23.

4.1	-4.22	7.5	0.33	14.3	2	3.25
1	?	1	?	1	?	1

Figura 23 – Ejemplo del Autómata Celular luego del Segundo Paso de Ordenamiento

Como puede verse, los elementos de la segunda fila debajo de un valor numérico han cambiado nuevamente, para armar nuevos pares en el próximo paso. El autómata sigue realizando pasos en forma infinita, debido a que por más que no se realicen cambios en los pares al estar ordenados correctamente, los elementos de la segunda fila siguen cambiando de valor en cada paso. Por esto, para que finalice la ejecución debe asignarse un tiempo máximo de simulación, el cual dependerá de la cantidad y del orden inicial de los valores a ordenar.

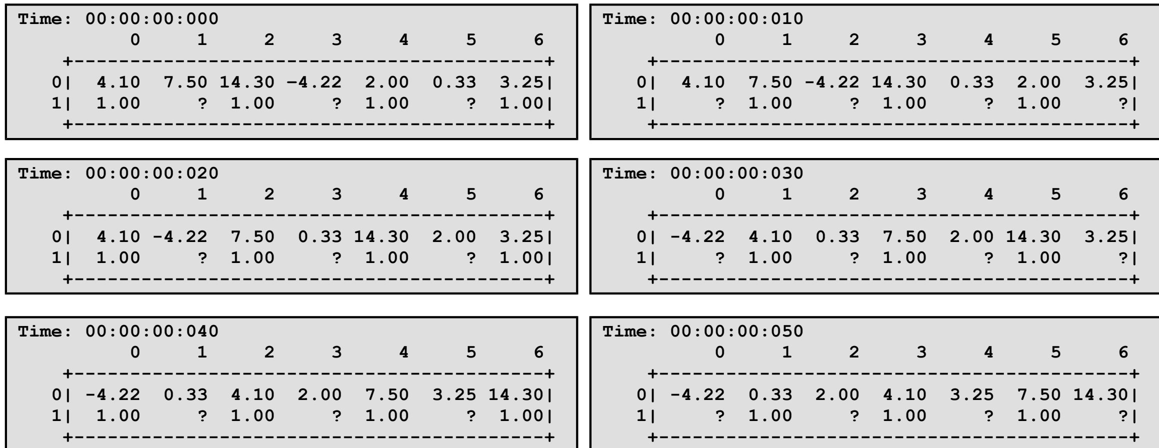
En la Figura 24 se detalla la definición del modelo en el lenguaje provisto por la herramienta.

```
01      [top]
02      components : sort
03
04      [sort]
05      type : cell
06      width : 7
07      height : 2
08      delay : transport
09      defaultDelayTime  : 10
10      border : nowrapped
11      neighbors : sort(-1,-1) sort(-1,0) sort(-1,1)
12      neighbors : sort(0,-1)  sort(0,0)  sort(0,1)
13      neighbors : sort(1,-1)  sort(1,0)  sort(1,1)
14      initialvalue : 0
15      initialrow :  0    4.1   7.5  14.3  -4.22    2   0.33   3.25
16      initialrow :  1      1     ?     1      ?    1      ?      1
17      localtransition : sort-rule
18
19      [sort-rule]
20      rule : ?        10 { (0,0) = 1 and ( (0,1) = ? or (0,-1) = ? ) }
21      rule : 1        10 { (0,0) = ? }
22      rule : {(0,1)}  10 { (1,0) = 1 and (0,0) > (0,1) }
```

```
24      rule : {(0,-1)} 10 { (1,0) = ? and (0,-1) > (0,0) }
24      rule : {(0,0)}  10 { t }
```

Figura 24 – Implementación del Autómata para la Ordenamiento Numérico

El resumen de los resultados generados por la herramienta se muestra en la Figura 25. En el tiempo 00:00:00:050 los valores deseados ya se encuentran ordenados, sin embargo, la simulación proseguirá hasta el tiempo definido, pues los valores de la segunda fila son intercambiados constantemente.



```
Time: 00:00:00:000
        0     1     2     3     4     5     6
   +------------------------------------------+
  0|  4.10  7.50 14.30 -4.22  2.00  0.33  3.25|
  1|  1.00     ?  1.00     ?  1.00     ?  1.00|
   +------------------------------------------+

Time: 00:00:00:010
        0     1     2     3     4     5     6
   +------------------------------------------+
  0|  4.10  7.50 -4.22 14.30  0.33  2.00  3.25|
  1|     ?  1.00     ?  1.00     ?  1.00     ?|
   +------------------------------------------+

Time: 00:00:00:020
        0     1     2     3     4     5     6
   +------------------------------------------+
  0|  4.10 -4.22  7.50  0.33 14.30  2.00  3.25|
  1|  1.00     ?  1.00     ?  1.00     ?  1.00|
   +------------------------------------------+

Time: 00:00:00:030
        0     1     2     3     4     5     6
   +------------------------------------------+
  0| -4.22  4.10  0.33  7.50  2.00 14.30  3.25|
  1|     ?  1.00     ?  1.00     ?  1.00     ?|
   +------------------------------------------+

Time: 00:00:00:040
        0     1     2     3     4     5     6
   +------------------------------------------+
  0| -4.22  0.33  4.10  2.00  7.50  3.25 14.30|
  1|  1.00     ?  1.00     ?  1.00     ?  1.00|
   +------------------------------------------+

Time: 00:00:00:050
        0     1     2     3     4     5     6
   +------------------------------------------+
  0| -4.22  0.33  2.00  4.10  3.25  7.50 14.30|
  1|     ?  1.00     ?  1.00     ?  1.00     ?|
   +------------------------------------------+
```

Figura 25 – Resultados obtenidos para el Autómata de Ordenamiento Numérico

14.4 Sistema de Trafico de Vehículos

El siguiente ejemplo permite simular un cruce de dos avenidas de un solo sentido, donde no existe un semáforo. Existen automóviles que se dirigen de oeste a este (los cuales serán representados usando el valor 1) y automóviles que van desde el sur al norte (representados por el valor 2). Una casilla sin automóviles esta representada por el valor 0. En la parte inferior izquierda del modelo existe un empalme con otra calle de un solo sentido, con dirección hacia el cruce, por la cual circulan vehículos representados por el valor 1, los que al llegar al cruce pueden dirigirse aleatoriamente hacia el norte o al este, con probabilidad 0.6 y 0.4 respectivamente.

Para la modelización de este ejemplo se usarán dos autómatas celulares, uno para representar el cruce, y otro para representar la calle que empalmará con el cruce. El acoplamiento de los modelos *Cell-DEVS* se muestra en la Figura 26.

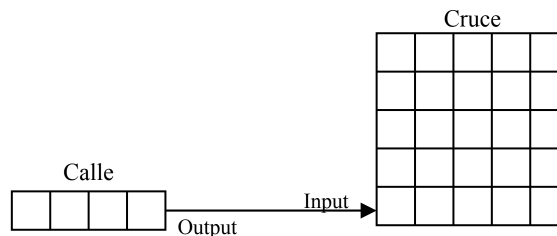


Figura 26 – Esquema de acoplamiento para el modelo de tráfico vehicular

La implementación del modelo en el lenguaje provisto por la herramienta se muestra en la Figura 27.

```
01   [top]
02   components : cruce calle
03   link : output@calle input@cruce
04
05   [calle]
06   type : cell
07   width : 4
```

```
08   height : 1
09   delay : transport
10   defaultDelayTime  : 1000
11   border : nowrapped
12   neighbors : calle(0,-1) calle(0,0) calle(0,1)
13   initialvalue : 0
14   initialrowvalue :  0      1100
15   out : output
16   link : out@calle(0,3) output
17   localtransition : movim-calle
18
19   [movim-calle]
20   rule : 1 1000 { (0,0) = 1 and (0,1) = 1 }
21   rule : 1 1000 { (0,0) = 0 and (0,-1) = 1 }
22   rule : 0 1000 { t }
23
24   [cruce]
25   type : cell
26   width : 5
27   height : 5
28   delay : transport
29   defaultDelayTime  : 1000
30   border : nowrapped
31   neighbors:             cruce(-1,0)
32   neighbors: cruce(0,-1) cruce(0,0) cruce(0,1)
33   neighbors:             cruce(1,0) cruce(1,1)
34   initialvalue : 0
35   initialrowvalue :  0      01011
36   initialrowvalue :  1      00210
37   initialrowvalue :  2      20102
38   initialrowvalue :  3      00200
39   initialrowvalue :  4      00120
40   in : input
41   link : input in@cruce(4,0)
42   portInTransition : in@cruce(4,0)
43   ingresoAutomovil
44   localtransition : traffic-rule
45
46   [traffic-rule]
47   rule : 2  1000  { (0,0) = 2 and ((-1,0) = 1 or (-1,0) = 2) }
48   rule : 2  1000  { (0,0) = 0 and (1,0) = 2 }
49   rule : 1  1000  { (0,0) = 1 and (0,1) = 0 and (1,1) = 2 }
50   rule : 1  1000  { (0,0) = 1 and ((0,1) = 1 or (0,1) = 2) }
51   rule : 1  1000  { (0,0) = 0 and (0,-1) = 1 }
52   rule : 0  1000  { (0,0) = 0 }
53
54   [ingresoAutomovil]
55   rule : 1 1000 { portValue(thisPort) != 0 and random > 0.6 }
56   rule : 2 1000 { portValue(thisPort) != 0 }
57   rule : 0 1000 { t }
```

Figura 27 – Implementación del modelo de Tráfico Vehicular

Entre las líneas 1 a 3 se define el modelo acoplado de mayor nivel. Aquí se indican sus componentes y como se acoplan entre sí.

Entre las líneas 5 a 17 se define el modelo acoplado celular para representar la Calle que empalmará con el Cruce. Se especifica su dimensión (1x4), su demora, vecindario y valores iniciales para las celdas, como así también el puerto de salida. En la línea 16 se indica que los valores provenientes de la salida de la celda (0,3) vayan al puerto *Output* del modelo celular. Finalmente en la línea 17 se define la función que contiene el comportamiento para las celdas. Este comportamiento esta definido en las líneas 19 a 22 mediante una serie de reglas que describen el desplazamiento de un vehículo de oeste a este.

Entre las líneas 24 a 44 se define el autómata celular de 5x5 que modela el Cruce. Entre las líneas 34 a 39 se definen los valores que contendrá inicialmente el modelo. En la línea 44 se establece la función que describirá el comportamiento las celdas, mientras que en la línea 41 se indica que la entrada proveniente del puerto *Input* del modelo acoplado celular vaya al puerto *In* de la celda (4,0). En la línea 42 se define la función que dictará el comportamiento de la celda (4,0) cuando ingrese un mensaje externo por el puerto de entrada *In*. Esta función esta definida entre las líneas 54 a 57.

El resumen de los resultados arrojados por la herramienta se muestra en la Figura 28. Para cada instante de tiempo se muestran los estados de los dos autómatas. En el tiempo de simulación 00:00:03:000 el automóvil que circulaba por la calle llega al cruce y decide dirigirse hacia el norte, mientras que en el tiempo 00:00:05:000 el segundo automóvil que transitaba la calle al llegar al cruce decide dirigirse hacia el este.

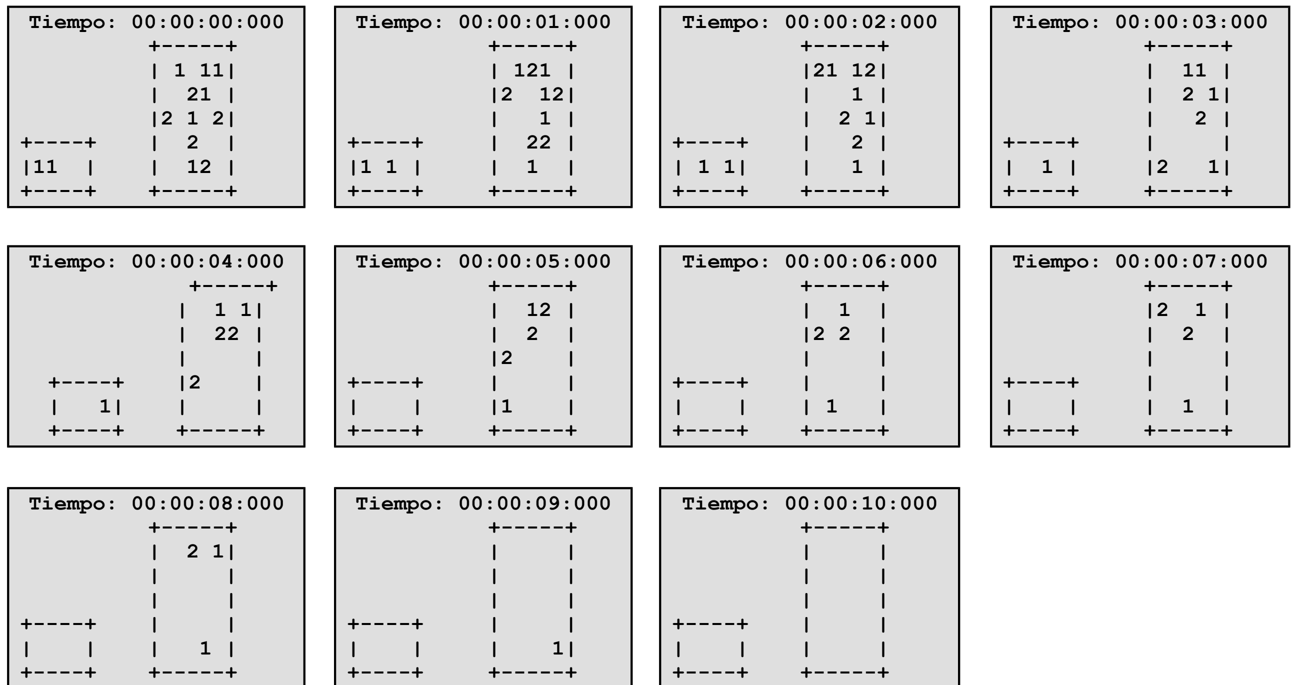


Figura 28 – Resultados obtenidos para la simulación del modelo de Tráfico Vehicular

14.5 Difusión del Calor sobre una Superficie

Este modelo consiste de una superficie, representada por un autómata celular, donde cada celda contiene una temperatura. En cada etapa de la simulación, la temperatura de la celda es calculada como el promedio de los valores del vecindario. El modelo cuenta con un generador de calor conectado a las celdas (2, 2) y (5, 5), que permite la creación de temperaturas en el rango [24, 40] con distribución uniforme. Por otro lado, se cuenta con un generador de frío, que permite crear valores en el rango [10, 15] también con distribución uniforme, y que esta conectado a las celdas (2, 8) y (8, 8). Los dos generadores crean valores luego de *x* segundos, donde *x* tiene una distribución exponencial con media 50 segundos. El esquema de acoplamiento del modelo es mostrado en la Figura 29.

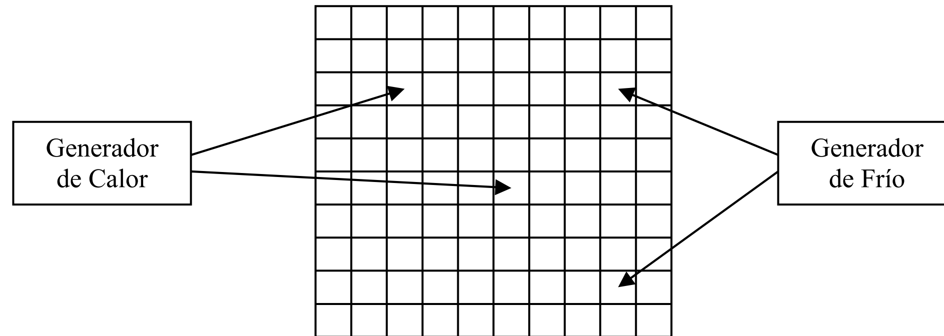


Figura 29 – Esquema de acoplamiento para el modelo de Difusión del Calor sobre una superficie

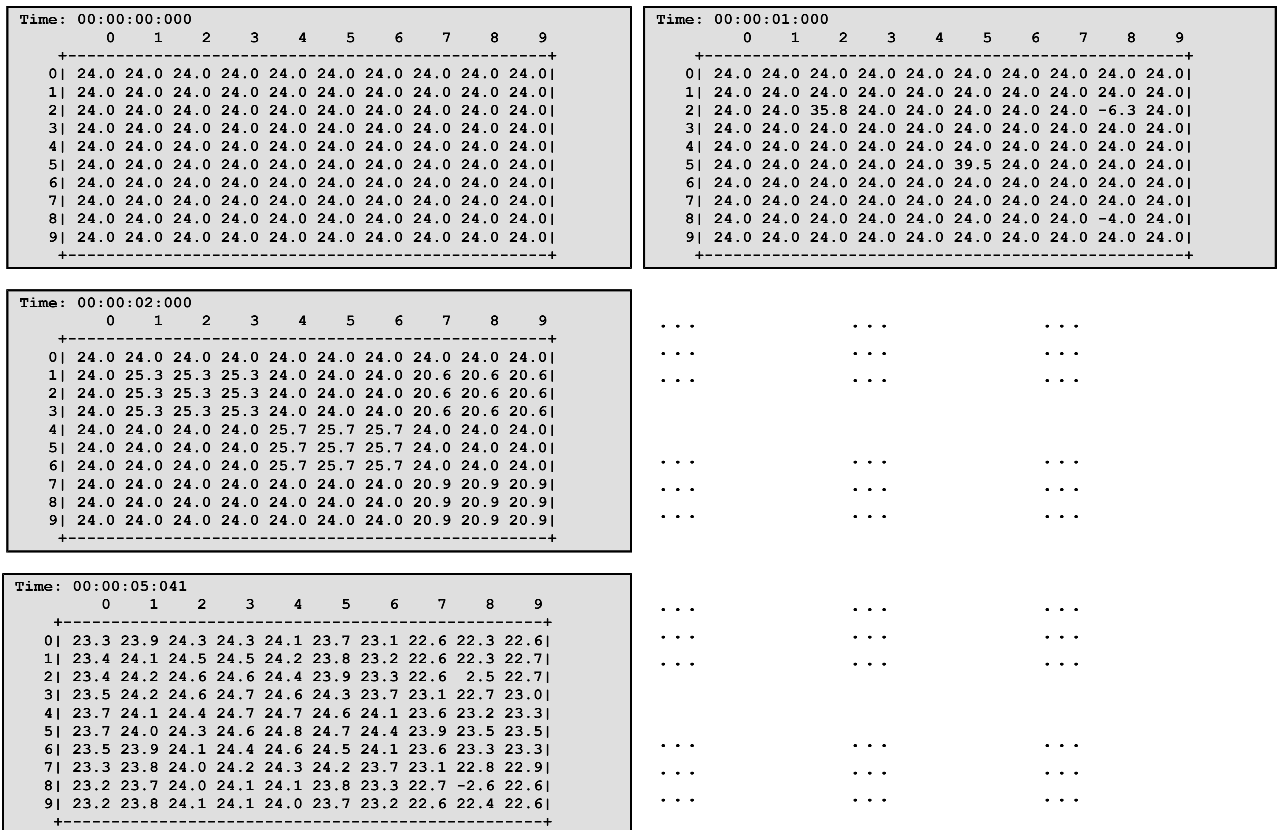
La definición del modelo usando el lenguaje provisto por *N–CD++* es mostrada en la Figura 30. El modelo acoplado de mayor nivel es definido entre las líneas 1 y 4. Entre las líneas 6 y 26, se define el modelo que representa a la superficie. El mismo esta compuesto por un autómata celular de 10x10 celdas, donde cada celda tiene una temperatura inicial de 24° C. En las líneas 28 y 29 se define la función de cómputo local que da comportamiento a dicho autómata. En las líneas 31 y 32 se define la función para crear temperaturas en el rango [24, 40] con distribución uniforme. De manera semejante, las líneas 34 y 35 definen la función para crear temperaturas en el rango [10, 15] también con distribución uniforme. Finalmente, entre las líneas 37 y 47 se definen los generadores de frío y calor.

```
01   [top]
02   components : surface generatorHeat@Generator generatorCold@Generator
03   link : out@generatorHeat  inputHeat@surface
04   link : out@generatorCold  inputCold@surface
05
06   [surface]
07   type : cell
08   width : 10
09   height : 10
10   delay : transport
11   defaultDelayTime  : 100
12   border : wrapped
13   neighbors : surface(-1,-1) surface(-1,0)  surface(-1,1)
14   neighbors : surface(0,-1)  surface(0,0)   surface(0,1)
15   neighbors : surface(1,-1)  surface(1,0)   surface(1,1)
16   initialvalue : 24
17   in : inputHeat inputCold
18   link : inputHeat in@surface(5,5)
19   link : inputHeat in@surface(2,2)
20   link : inputCold in@surface(8,8)
21   link : inputCold in@surface(2,8)
22   localtransition : heat-rule
23   portInTransition : in@surface(5,5)  setHeat
24   portInTransition : in@surface(2,2)  setHeat
25   portInTransition : in@surface(8,8)  setCold
26   portInTransition : in@surface(2,8)  setCold
27
28   [heat-rule]
29   rule : { ((0,0) + (-1,-1) + (-1,0) + (-1,1) + (0,-1) + (0,1) + (1,-1) +
              (1,0) + (1,1)) / 9 } 10000 { t }
30
31   [setHeat]
32   rule : { uniform(24,40) } 1000 { t }
33
34   [setCold]
35   rule : { uniform(-10,15) } 1000 { t }
36
37   [generatorHeat]
38   distribution : exponential
```

```
39   mean : 50
40   initial : 1
41   increment : 0
42
43   [generatorCold]
44   distribution : exponential
45   mean : 50
46   initial : 1
47   increment : 0
```

Figura 30 – Implementación del Modelo de *Difusión del Calor sobre una Superficie*

El resumen de la salida generada por la herramienta es mostrada en la Figura 31. En el tiempo de simulación 00:00:01:000 los generadores de frío y calor producen cambios en las celdas a las cuales se conectan. Esto produce que los estados de los vecinos de estas celdas cambien en 00:00:02:000, y que en etapas sucesivas, se produzca un cambio en el resto de las celdas. En el tiempo 00:00:05:041, el generador de frío produce un cambio en el estado de las celdas (2, 8) y (8, 8), estableciendo los valores 2.5 y -2.6 respectivamente, lo que producirá futuros cambios en los estados de sus vecinos.



```
Time: 00:00:00:000
       0    1    2    3    4    5    6    7    8    9
   +--------------------------------------------------+
  0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  2| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  5| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   +--------------------------------------------------+
```

```
Time: 00:00:01:000
       0    1    2    3    4    5    6    7    8    9
   +--------------------------------------------------+
  0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  2| 24.0 24.0 35.8 24.0 24.0 24.0 24.0 24.0 -6.3 24.0|
  3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  5| 24.0 24.0 24.0 24.0 24.0 39.5 24.0 24.0 24.0 24.0|
  6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 -4.0 24.0|
  9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   +--------------------------------------------------+
```

```
Time: 00:00:02:000
       0    1    2    3    4    5    6    7    8    9
   +--------------------------------------------------+
  0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
  1| 24.0 25.3 25.3 25.3 24.0 24.0 24.0 20.6 20.6 20.6|
  2| 24.0 25.3 25.3 25.3 24.0 24.0 24.0 20.6 20.6 20.6|
  3| 24.0 25.3 25.3 25.3 24.0 24.0 24.0 20.6 20.6 20.6|
  4| 24.0 24.0 24.0 24.0 25.7 25.7 25.7 24.0 24.0 24.0|
  5| 24.0 24.0 24.0 24.0 25.7 25.7 25.7 24.0 24.0 24.0|
  6| 24.0 24.0 24.0 24.0 25.7 25.7 25.7 24.0 24.0 24.0|
  7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 20.9 20.9 20.9|
  8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 20.9 20.9 20.9|
  9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 20.9 20.9 20.9|
   +--------------------------------------------------+
```

```
...          ...          ...
...          ...          ...
...          ...          ...

...          ...          ...
...          ...          ...
...          ...          ...
```

```
Time: 00:00:05:041
       0    1    2    3    4    5    6    7    8    9
   +--------------------------------------------------+
  0| 23.3 23.9 24.3 24.3 24.1 23.7 23.1 22.6 22.3 22.6|
  1| 23.4 24.1 24.5 24.5 24.2 23.8 23.2 22.6 22.3 22.7|
  2| 23.4 24.2 24.6 24.6 24.4 23.9 23.3 22.6  2.5 22.7|
  3| 23.5 24.2 24.6 24.7 24.6 24.3 23.7 23.1 22.7 23.0|
  4| 23.7 24.1 24.4 24.7 24.7 24.6 24.1 23.6 23.2 23.3|
  5| 23.7 24.0 24.3 24.6 24.8 24.7 24.4 23.9 23.5 23.5|
  6| 23.5 23.9 24.1 24.4 24.6 24.5 24.1 23.6 23.3 23.3|
  7| 23.3 23.8 24.0 24.2 24.3 24.2 23.7 23.1 22.8 22.9|
  8| 23.2 23.7 24.0 24.1 24.1 23.8 23.3 22.7 -2.6 22.6|
  9| 23.2 23.8 24.1 24.1 24.0 23.7 23.2 22.6 22.4 22.6|
   +--------------------------------------------------+
```

```
...          ...          ...
...          ...          ...
...          ...          ...

...          ...          ...
...          ...          ...
...          ...          ...
```

Figura 31 – Resumen de la salida generada para el modelo de difusión del calor en una superficie

14.6 Difusión del Calor en 3D

En este ejemplo se cuenta con un ambiente representado por un autómata celular, donde cada celda contiene una temperatura. En cada etapa de la simulación, la temperatura de la celda se calcula como el promedio de los valores de la vecindad, cuya forma se muestra en la Figura 32. Además, existe un generador de calor que se conecta a las celdas (2,2,1) y (3,3,0), el cual permite la creación de temperaturas en el rango [24, 80] grados centígrados, con distribución uniforme. Por otra parte, se cuenta con un generador de frío, que permite crear valores en el rango [-45, 10] también con distribución uniforme, y se conecta a las celdas (1,3,3) y (3,3,2). Ambos generadores crean valores luego de

transcurridos *x* segundos, donde *x* sigue una distribución exponencial con media 40 segundos. El esquema del acoplamiento de los modelos se muestra en la Figura 33.

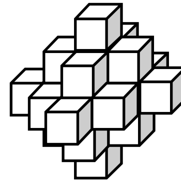


Figura 32 – Vecindario para el ejemplo de *Difusión del Calor* en 3D

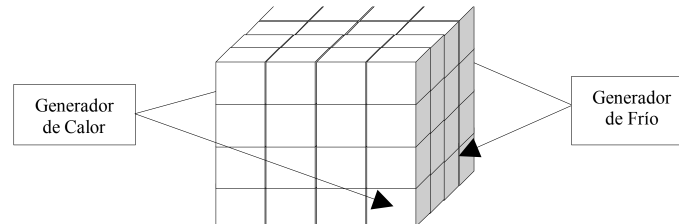


Figura 33 – Esquema de acoplamiento para el modelo de *Difusión del Calor*

La definición del modelo usando el lenguaje provisto por la herramienta se muestra en la Figura 34. Entre las líneas 1 y 4 se define el modelo con mayor nivel de abstracción, junto a los modelos que los integran. Además, se muestra el acoplamiento entre los mismos. Entre las líneas 6 y 32 se define el modelo que representa el ambiente, compuesto por un autómata celular de 10 x 10 x 4 celdas, donde inicialmente todas tienen una temperatura de 24° C (según lo definido en la línea 22). Entre las líneas 34 y 35 se define la función que calcula el valor de estado de las celdas como el promedio de los valores del vecindario. En las líneas 37 y 38 se define la función que crea una temperatura en el rango [24, 80] con distribución uniforme, al ingresar un mensaje por el puerto In de las celdas (2,2,1) y (3,3,0) como se determinó en las líneas 24 y 25. Análogamente, en las líneas 40 y 41 se define la función para crear temperaturas en el rango [-45, 10] con distribución uniforme, al ingresar un mensaje por el puerto In de las celdas (1,3,3) y (3,3,2). Finalmente, entre las líneas 43 y 53 se definen los generadores de frío y calor, que generan valores cada *x* segundos, donde *x* sigue una distribución exponencial con media 40 segundos.

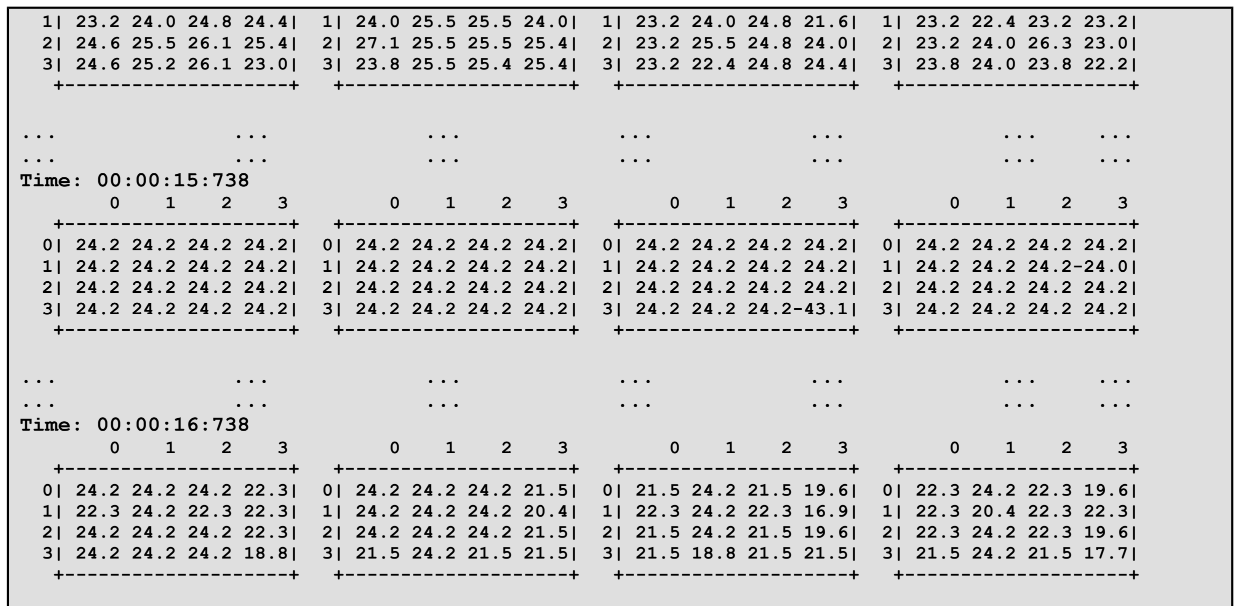
```
01      [top]
02      components : superficie generadorCalor@Generator generadorFrio@Generator
03      link : out@generadorCalor inputCalor@superficie
04      link : out@generadorFrio  inputFrio@superficie
05
06      [superficie]
07      type : cell
08      dim : (4, 4, 4)
09      delay : transport
10      defaultDelayTime  : 100
11      border : wrapped
12      neighbors :                     superficie(-1,0,-1)
13      neighbors : superficie(0,-1,-1)  superficie(0,0,-1)  superficie(0,1,-1)
14      neighbors :                     superficie(1,0,-1)
15      neighbors : superficie(-1,-1,0)  superficie(-1,0,0)  superficie(-1,1,0)
16      neighbors : superficie(0,-1,0)   superficie(0,0,0)   superficie(0,1,0)
17      neighbors : superficie(1,-1,0)   superficie(1,0,0)   superficie(1,1,0)
18      neighbors :                     superficie(-1,0,1)
19      neighbors : superficie(0,-1,1)   superficie(0,0,1)   superficie(0,1,1)
20      neighbors :                     superficie(1,0,1)
21      neighbors : superficie(0,0,-2)   superficie(0,0,2)   superficie(0,2,0)
21      neighbors : superficie(0,-2,0)   superficie(2,0,0)   superficie(-2,0,0)
22      initialvalue : 24
23      in : inputCalor inputFrio
24      link : inputCalor in@superficie(3,3,0)
```

```
25      link : inputCalor in@superficie(2,2,1)
26      link : inputFrio  in@superficie(3,3,2)
27      link : inputFrio  in@superficie(1,3,3)
28      localtransition : calor-rule
29      portInTransition : in@superficie(3,3,0)  setCalor
30      portInTransition : in@superficie(2,2,1)  setCalor
31      portInTransition : in@superficie(3,3,2)  setFrio
32      portInTransition : in@superficie(1,3,3)  setFrio
33
34      [calor-rule]
35      rule : { ( (-1,0,-1) + (0,-1,-1) + (0,0,-1) + (0,1,-1) + (1,0,-1) +
                   (-1,-1,0) + (-1,0,0) + (-1,1,0) + (0,-1,0) + (0,0,0) + (0,1,0) +
                   (1,-1,0) + (1,0,0) + (1,1,0) + (-1,0,1) + (0,-1,1) + (0,0,1) +
                   (0,1,1) + (1,0,1) + (0,0,-2) + (0,0,2) + (0,2,0) + (0,-2,0) +
                   (2,0,0) + (-2,0,0) ) / 25 } 1000 { t }
36
37      [setCalor]
38      rule : { uniform(24,80) } 1000 { t }
39
40      [setFrio]
41      rule : { uniform(-45,10) } 1000 { t }
42
43      [generadorCalor]
44      distribution : exponential
45      mean : 10
46      initial : 1
47      increment : 0
48
49      [generadorFrio]
50      distribution : exponential
51      mean : 10
52      initial : 1
53      increment : 0
```

Figura 34 – Definición del modelo de *Difusión del Calor* en 3D

El resumen de los resultados de salida generados por la herramienta se muestra en la Figura 35. En el tiempo 00:00:01:000 los generadores de frío y calor producen cambios en las celdas a las cuales están conectadas. Esto conduce a que los estados de los vecinos de estas celdas cambien en 00:00:02:000, y en las sucesivas etapas se producirá un cambio en el resto de las celdas. En el tiempo 00:00:15:738, el generador de frío producirá un cambio en el estado de las celdas (1,3,3) y (3,3,2), estableciéndose los valores -24 y –43.1 respectivamente, lo que producirá futuros cambios de estados en los vecinos de las mismas.

```
Time: 00:00:00:000
       0    1    2    3        0    1    2    3        0    1    2    3        0    1    2    3
   +--------------------+  +--------------------+  +--------------------+  +--------------------+
  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|
  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0 24.0|
  2| 24.0 24.0 24.0 24.0|  2| 24.0 24.0 24.0 24.0|  2| 24.0 24.0 24.0 24.0|  2| 24.0 24.0 24.0 24.0|
  3| 24.0 24.0 24.0 24.0|  3| 24.0 24.0 24.0 24.0|  3| 24.0 24.0 24.0 24.0|  3| 24.0 24.0 24.0 24.0|
   +--------------------+  +--------------------+  +--------------------+  +--------------------+

Time: 00:00:01:000
       0    1    2    3        0    1    2    3        0    1    2    3        0    1    2    3
   +--------------------+  +--------------------+  +--------------------+  +--------------------+
  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|  0| 24.0 24.0 24.0 24.0|
  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0 24.0|  1| 24.0 24.0 24.0  4.5|
  2| 24.0 24.0 24.0 24.0|  2| 24.0 24.0 62.7 24.0|  2| 24.0 24.0 24.0 24.0|  2| 24.0 24.0 24.0 24.0|
  3| 24.0 24.0 24.0 38.8|  3| 24.0 24.0 24.0 24.0|  3| 24.0 24.0 24.0  4.2|  3| 24.0 24.0 24.0 24.0|
   +--------------------+  +--------------------+  +--------------------+  +--------------------+

Time: 00:00:02:000
       0    1    2    3        0    1    2    3        0    1    2    3        0    1    2    3
   +--------------------+  +--------------------+  +--------------------+  +--------------------+
  0| 24.6 24.0 24.6 23.8|  0| 24.0 24.0 27.1 23.8|  0| 23.2 24.0 23.2 22.4|  0| 23.2 24.0 23.2 23.0|
```



```
  1| 23.2 24.0 24.8 24.4|  1| 24.0 25.5 25.5 24.0|  1| 23.2 24.0 24.8 21.6|  1| 23.2 22.4 23.2 23.2|
  2| 24.6 25.5 26.1 25.4|  2| 27.1 25.5 25.5 25.4|  2| 23.2 25.5 24.8 24.0|  2| 23.2 24.0 26.3 23.0|
  3| 24.6 25.2 26.1 23.0|  3| 23.8 25.5 25.4 25.4|  3| 23.2 22.4 24.8 24.4|  3| 23.8 24.0 23.8 22.2|
   +-------------------+   +-------------------+   +-------------------+   +-------------------+

...              ...            ...            ...            ...            ...     ...
...              ...            ...            ...            ...            ...     ...
Time: 00:00:15:738
       0    1    2    3        0    1    2    3        0    1    2    3        0    1    2    3
   +-------------------+   +-------------------+   +-------------------+   +-------------------+
  0| 24.2 24.2 24.2 24.2|  0| 24.2 24.2 24.2 24.2|  0| 24.2 24.2 24.2 24.2|  0| 24.2 24.2 24.2 24.2|
  1| 24.2 24.2 24.2 24.2|  1| 24.2 24.2 24.2 24.2|  1| 24.2 24.2 24.2 24.2|  1| 24.2 24.2 24.2-24.0|
  2| 24.2 24.2 24.2 24.2|  2| 24.2 24.2 24.2 24.2|  2| 24.2 24.2 24.2 24.2|  2| 24.2 24.2 24.2 24.2|
  3| 24.2 24.2 24.2 24.2|  3| 24.2 24.2 24.2 24.2|  3| 24.2 24.2 24.2-43.1|  3| 24.2 24.2 24.2 24.2|
   +-------------------+   +-------------------+   +-------------------+   +-------------------+

...              ...            ...            ...            ...            ...     ...
...              ...            ...            ...            ...            ...     ...
Time: 00:00:16:738
       0    1    2    3        0    1    2    3        0    1    2    3        0    1    2    3
   +-------------------+   +-------------------+   +-------------------+   +-------------------+
  0| 24.2 24.2 24.2 22.3|  0| 24.2 24.2 24.2 21.5|  0| 21.5 24.2 21.5 19.6|  0| 22.3 24.2 22.3 19.6|
  1| 22.3 24.2 22.3 22.3|  1| 24.2 24.2 24.2 20.4|  1| 22.3 24.2 22.3 16.9|  1| 22.3 20.4 22.3 22.3|
  2| 24.2 24.2 24.2 22.3|  2| 24.2 24.2 24.2 21.5|  2| 21.5 24.2 21.5 19.6|  2| 22.3 24.2 22.3 19.6|
  3| 24.2 24.2 24.2 18.8|  3| 21.5 24.2 21.5 21.5|  3| 21.5 18.8 21.5 21.5|  3| 21.5 24.2 21.5 17.7|
   +-------------------+   +-------------------+   +-------------------+   +-------------------+
```

Figura 35 – Resumen de los resultados obtenidos durante la simulación de difusión del calor

14.7 Fluido de Gases

En este caso se muestra la difusión de un grupo de partículas de cierto gas. Cada partícula esta representada por un número real. La parte entera de su valor simboliza la dirección en la que se desplaza la partícula, pudiendo ser 1 (hacia la derecha), 2 (hacia arriba), 3 (hacia la izquierda) o 4 (hacia abajo); mientras que la parte decimal representa el grado pureza de la misma, perteneciente al intervalo [0, 1).

Las partículas aisladas se mueven en líneas rectas. Cuando dos partículas provenientes de distintas direcciones, chocan formando un ángulo de 180 grados, el par se destruye, y se crea un nuevo par, viajando en un nuevo ángulo de 90 grados con respecto al anterior, donde la pureza de ambas partículas será el promedio de la pureza del par que chocó. Por el contrario, si chocan formando un ángulo de 90 grados, entonces las partículas se fusionan formando solo una partícula. El grado de pureza se promedia como en el caso anterior, y la dirección corresponderá a la de la dirección de la partícula que tenga mayor pureza.

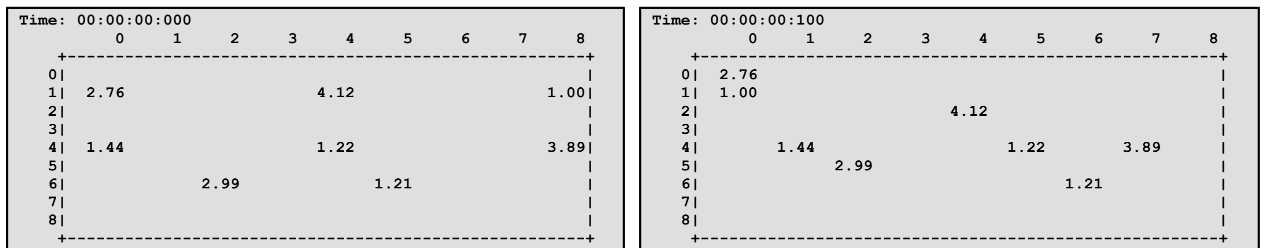
La descripción del modelo en el lenguaje provisto por *N–CD++* se muestra en la Figura 36.

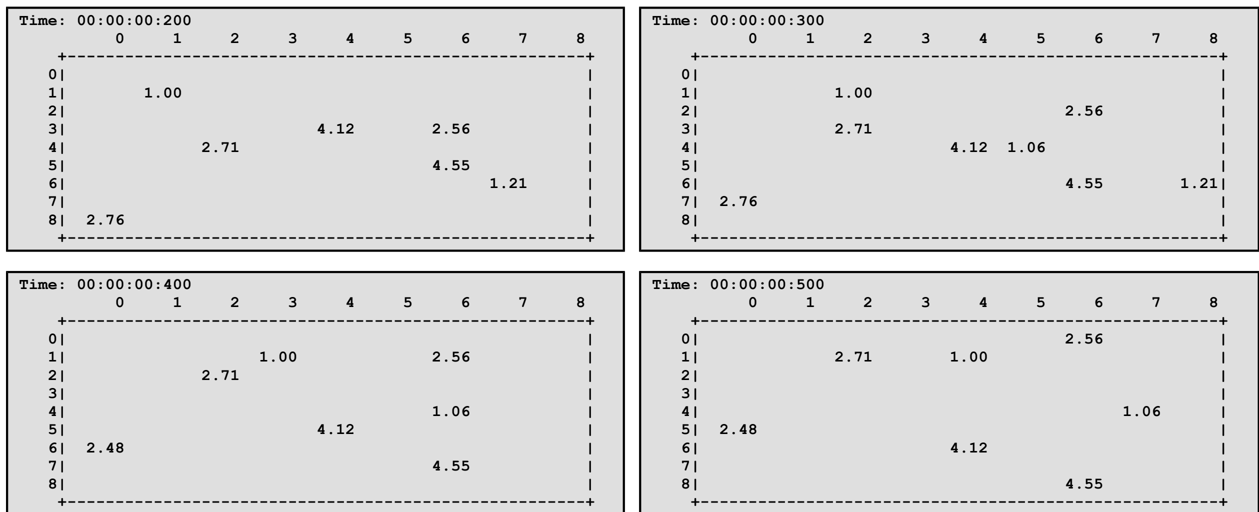
```
01      [top]
02      components : particulas
03
04      [particulas]
05      type : cell
06      width : 9
07      height : 9
08      delay : transport
09      defaultDelayTime  : 100
10      border : wrapped
11      neighbors : particulas(-1,-1) particulas(-1,0) particulas(-1,1)
12      neighbors : particulas(0,-1)  particulas(0,0)  particulas(0,1)
13      neighbors : particulas(1,-1)  particulas(1,0)  particulas(1,1)
14      initialvalue : 0
15      initialrow : 1     2.76 0.00 0.00 0.00 4.12 0.00 0.00 0.00 1.00
16      initialrow : 4     1.44 0.00 0.00 0.00 1.32 0.00 0.00 0.00 3.89
17      initialrow : 6     0.00 0.00 2.99 0.00 0.00 1.21 0.00 0.00 0.00
18      localtransition : particulas-rule
19
```

```
20      [particulas-rule]
21      rule : { 2 + ( fractional((1,0)) + fractional((1,1)) ) / 2 }    100
                                          { trunc((1,0)) = 1 and trunc((1,1)) = 3 }
22      rule : { 4 + ( fractional((-1,-1)) + fractional((-1,0)) ) / 2 } 100
                                       { trunc((-1,-1)) = 1 and trunc((-1,0)) = 3 }
23      rule : { 2 + ( fractional((1,-1)) + fractional((1,1)) ) / 2 }   100
                                         { trunc((1,-1)) = 1 and trunc((1,1)) = 3 }
25      Rule : { 4 + ( fractional((-1,-1)) + fractional((-1,1)) ) / 2 } 100
                                       { trunc((-1,-1)) = 1 and trunc((-1,1)) = 3 }
26      Rule : { 1 + ( fractional((-1,-1)) + fractional((0,-1)) ) / 2 } 100
                                       { trunc((0,-1)) = 2 and trunc((-1,-1)) = 4 }
26      rule : { 3 + ( fractional((1,1)) + fractional((0,1)) ) / 2 }    100
                                          { trunc((0,1)) = 4 and trunc((1,1)) = 2 }
27      rule : { 1 + ( fractional((-1,-1)) + fractional((1,-1)) ) / 2 } 100
                                       { trunc((-1,-1)) = 4 and trunc((-1,1)) = 2 }
28      rule : { if( fractional((0,1)) > fractional((1,0)), 3, 2) +
                             (fractional((0,1)) + fractional((1,0)) ) / 2 } 100
                             { trunc((0,1)) = 3 and trunc((1,0)) = 2 }
29      rule : { if( fractional((0,1)) > fractional((-1,0)), 3, 4) +
                             (fractional((0,1)) + fractional((-1,0)) ) / 2 } 100
                             { trunc((0,1)) = 3 and trunc((-1,0)) = 4 }
30      rule : { if( fractional((0,-1)) > fractional((1,0)), 1, 2) +
                             (fractional((0,-1)) + fractional((1,0)) ) / 2 } 100
                             { trunc((0,-1)) = 1 and trunc((1,0)) = 2 }
31      rule : { if( fractional((0,-1)) > fractional((-1,0)), 1, 4) +
                             (fractional((0,-1)) + fractional((-1,0)) ) / 2 } 100
                             { trunc((0,-1)) = 1 and trunc((-1,0)) = 4 }
32      rule : 0          100 { trunc((0,0)) = 1 and (0,1) = 0 }
33      rule : { (0,-1) } 100 { trunc((0,-1)) = 1 and trunc((0,1)) != 3 }
34      rule : 0          100 { trunc((0,0)) = 3 and (0,-1) = 0 }
35      rule : { (0,1) }  100 { trunc((0,1)) = 3 and trunc((0,-1)) != 1 }
36      rule : 0          100 { trunc((0,0)) = 2 and (-1,0) = 0 }
37      rule : { (1,0) }  100 { trunc((1,0)) = 2 and trunc((-1,0)) != 4 }
38      rule : 0          100 { trunc((0,0)) = 4 and (1,0) = 0 }
39      rule : { (-1,0) } 100 { trunc((-1,0)) = 4 and trunc((1,0)) != 2 }
40      rule : 0          100 { t }
```

Figura 36 – Implementación del modelo de *Difusión de Partículas de Gas*

La salida generada por la herramienta se muestra en la Figura 37. En el tiempo 00:00:00:0200 se producirá un choque de las partículas con valores 1.44 y 2.99, fusionándose para formar la partícula con valor 2.71 en la celda (4, 2). También se producirá un choque entre las partículas con valores 1.22 y 3.89, por lo que las mismas cambiaran su rumbo en 90 grados y sus purezas serán promediadas. En el tiempo 00:00:00:0400 se produce otro choque entre las partículas con valores 1.21 y 2.76, generando la partícula con valor 2.48.





```
Time: 00:00:00:200
       0    1    2    3    4    5    6    7    8
   +-----------------------------------------------------+
  0|                                                     |
  1|       1.00                                          |
  2|                                                     |
  3|                             4.12       2.56         |
  4|            2.71                                     |
  5|                                        4.55         |
  6|                                             1.21    |
  7|                                                     |
  8|  2.76                                               |
   +-----------------------------------------------------+

Time: 00:00:00:300
       0    1    2    3    4    5    6    7    8
   +-----------------------------------------------------+
  0|                                                     |
  1|            1.00                                     |
  2|                                        2.56         |
  3|            2.71                                     |
  4|                        4.12  1.06                   |
  5|                                                     |
  6|                                        4.55     1.21|
  7|  2.76                                               |
  8|                                                     |
   +-----------------------------------------------------+

Time: 00:00:00:400
       0    1    2    3    4    5    6    7    8
   +-----------------------------------------------------+
  0|                                                     |
  1|                  1.00            2.56               |
  2|            2.71                                     |
  3|                                                     |
  4|                                  1.06               |
  5|                        4.12                         |
  6|  2.48                                               |
  7|                                  4.55               |
  8|                                                     |
   +-----------------------------------------------------+

Time: 00:00:00:500
       0    1    2    3    4    5    6    7    8
   +-----------------------------------------------------+
  0|                                        2.56         |
  1|            2.71       1.00                          |
  2|                                                     |
  3|                                                     |
  4|                                             1.06    |
  5|  2.48                                               |
  6|                        4.12                         |
  7|                                                     |
  8|                                        4.55         |
   +-----------------------------------------------------+
```

Figura 37 – Resultados para el Modelo de Difusión de Partículas de Gas

14.8 Uso de las extensiones para los puertos de Entrada y Salida

El siguiente ejemplo es usado para mostrar algunas de las nuevas características de N-CD++ para el manejo de los puertos.

El modelo definido usa el esquema de interconexión mostrado en la siguiente figura, donde los puertos usados para la conexión de los vecinos que integran al modelo celular han sido ocultados para lograr mayor claridad.

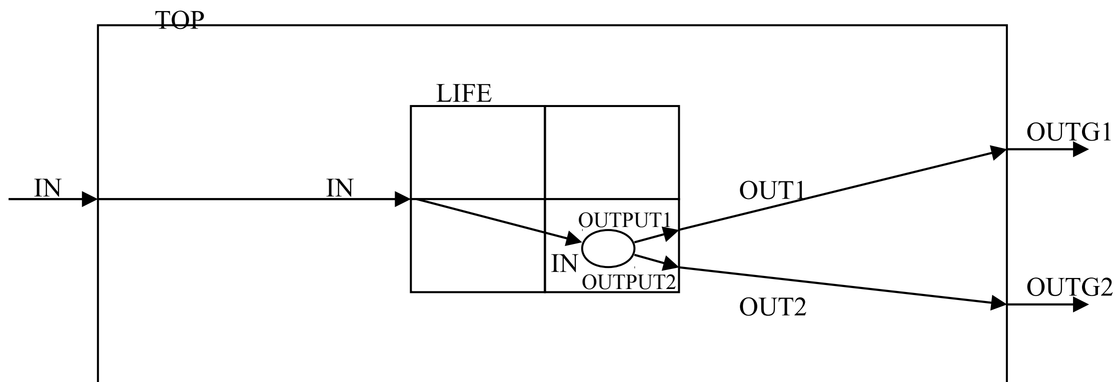


Figura 38 – Esquema de acoplamiento para el ejemplo que muestra las extensiones del manejo de puertos

En la Figura 39 se muestra la implementación del ejemplo para el lenguaje provisto por la herramienta. El modelo acoplado de mayor nivel (*TOP*) usa un puerto de entrada y dos de salida, y solo incluye un componente (*LIFE*), que es un modelo celular de tamaño 2x2 celdas. La cláusula *Link* es usada para crear el acoplamiento entre los puertos que interconectan a los modelos. Posteriormente se definen otros parámetros del modelo, como la dimensión, la demora, el vecindario, y el acoplamiento externo.

La cláusula *portInTransition* permite definir el nombre de una función a ser usada cuando llegue un mensaje a través de un puerto específico. En este caso, se ejecutará *specialRule*. La función *portValue*(*x*) permite obtener el valor del último mensaje que llegó por el puerto *x* de la celda. El uso de *thisPort* como parámetro permite indicar el uso del puerto por el cual arribó el mensaje.

La función de cómputo local, llamada *nothingRule*, no realiza ninguna tarea. Existe una zona especial (*generateOut*) usada para definir el envío de valores del modelo celular hacia sus puertos de salida.

```
[top]
```

```
components : life
in : in
out : outG1 outG2
link : out1@life outG1
link : out2@life outG2
link : in in@life

[life]
type : cell
width : 2
height : 2
delay : transport
defaultDelayTime : 1
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
in : in
out : out1 out2
link : in in@life(1,1)
link : output1@life(1,1) out1
link : output2@life(1,1) out2
portInTransition : in@life(1,1)          specialRule
localtransition : nothing-rule
zone : generateOut { (1,1) }

[nothing-rule]
rule : { (0,0) } 1 { t }

[specialRule]
rule : { portValue(thisPort) } 1 { t }

[generateOut]
rule : { (0,0) + send(output1, 9.9999) } 1 { (0,0) >= 10 }
rule : { (0,0) + send(output2, 3.3333) } 1 { (0,0) < 10 }
```

Figura 39 – Ejemplo del uso de las extensiones realizadas sobre los puertos de *Entrada* y *Salida*

14.9 Sistema de Clasificación de Materias Primas

El objetivo de este ejemplo es mostrar el uso de la redefinición efectuada en el comportamiento de las celdas ante la llegada de un evento externo. Se cuenta con un modelo que representa el embalaje y clasificación de cierto tipo de materia prima, que contiene aproximadamente un 30 % de carbono. Además, se dispone de una máquina que ubica fracciones de 100 gramos de esa sustancia en una cinta transportadora. Esta las almacena temporalmente hasta que sean procesadas por una embaladora, que toma dichas fracciones hasta alcanzar el kilogramo de peso, y las envasa. Posteriormente, se clasifica la sustancia ya envasada. Si cada envase contiene 30 ± 1 % de carbono, entonces es catalogado como de primera calidad; sino se lo cataloga como de segunda calidad.

Se usará un modelo atómico *DEVS*, llamado *Generator*, que devuelve un valor inicial luego de transcurridos *x* segundos. El generador puede ser configurado para que este parámetro siga una función de distribución probabilística a partir de un valor inicial junto a un incremento en el mismo. Luego de haberse retornado el valor inicial, éste es incrementado según el valor definido, y su resultado será el siguiente valor devuelto. En el ejemplo se genera siempre el valor 1, que indica, al arribar a la cinta transportadora, que se genere una de las fracciones de la sustancia. Se contará con un modelo celular para representar la cinta transportadora que almacena temporalmente las fracciones y otro modelo celular realizará las tareas de embalaje (agrupamiento de 10 fracciones) y selección.

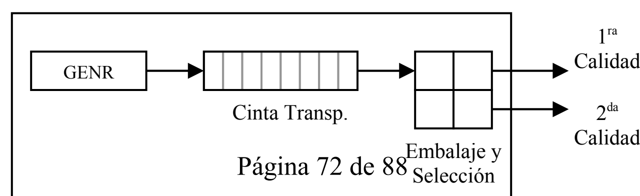


Figura 40 – Modelo Acoplado de la Clasificadora de Materias Primas

La descripción del modelo en el lenguaje provisto por la herramienta esta definida en la Figura 41.

```
01      [top]
02      components: genSustancias@Generator cola embalaje
03      out : outPrimeraSeleccion outSegundaSeleccion
04      link : out@genSustancias in@cola
05      link : out@cola in@embalaje
06      link : out1@embalaje outPrimeraSeleccion
07      link : out2@embalaje outSegundaSeleccion
08
09      [genSustancias]
10      distribution : exponential
11      mean : 3
12      initial : 1
13      increment : 0
14
15      [cola]
16      type : cell
17      width : 6
18      height : 1
19      delay : transport
20      defaultDelayTime  : 1
21      border : nowrapped
22      neighbors : cola(0,-1) cola(0,0) cola(0,1)
23      initialvalue : 0
24      in : in
25      out : out
26      link : in in@cola(0,0)
27      link : out@cola(0,5) out
28      localtransition : cola-rule
29      portInTransition: in@cola(0,0) establecerSustancia
30
31      [cola-rule]
32      rule: 0        1 {(0,0) != 0 and (0,1) = 0 }
33      rule: {(0,-1)} 1 {(0,0) = 0 and (0,-1) != 0 and not isUndefined((0,-1))}
34      rule: 0     3000 {(0,0) != 0 and isUndefined((0,1))}
35      rule: {(0,0)}  1 { t }
36
37      [establecerSustancia]
38      rule : { 30 + normal(0,2) } 1000 { t }
39
40      [embalaje]
41      type : cell
42      width : 2
43      height : 2
44      delay : transport
45      defaultDelayTime  : 1000
46      border : nowrapped
47      neighbors : embalaje(-1,-1) embalaje(-1,0) embalaje(-1,1)
48      neighbors : embalaje(0,-1)  embalaje(0,0)  embalaje(0,1)
49      neighbors : embalaje(1,-1)  embalaje(1,0)  embalaje(1,1)
50      in : in
51      out : out1 out2
52      initialvalue : 0
53      link : in in@embalaje(0,0)
54      link : in in@embalaje(1,0)
55      link : out@embalaje(0,1) out1
56      link : out@embalaje(1,1) out2
57      localtransition : embalaje-rule
58      portInTransition: in@embalaje(0,0) acumular-rule
59      portInTransition: in@embalaje(1,0) incCant-rule
```

```
60
61      [embalaje-rule]
62      rule: 0  1000 { isUndefined((1,0)) and isUndefined((0,-1)) and (0,0) = 10 }
63      rule: 0  1000 { isUndefined((-1,0)) and isUndefined((0,-1)) and (1,0) = 10}
64      rule: { (0,-1) / (1,-1) } 1000 { isUndefined((-1,0)) and isUndefined((0,1))
                  and (1,-1) = 10 and abs( (0,-1)/(1,-1) - 30 ) <= 1 }
65      rule: { (-1,-1) / (0,-1) } 1000 { isUndefined((1,0)) and isUndefined((0,1))
                  and (0,-1) = 10 and abs( (-1,-1)/(0,-1) - 30 ) > 1 }
66      rule: { (0,0) } 1000 { t }
67
68      [acumular-rule]
69      rule: { portValue(thisPort) + (0,0) } 1000 { t }
70
71      [incCant-rule]
72      rule: {1+(0,0)} 1000 {portValue(thisPort)!=0 }
73      rule: { (0,0) } 1000 { t }
```

Figura 41 – Implementación del Sistema de Embalaje y Clasificación de Materias Primas

Entre las líneas 1 y 7 se define el modelo acoplado de mayor nivel, indicando sus componentes y como se acoplan entre sí. El modelo cuenta con un generador de sustancias, una cinta transportadora y un mecanismo de embalaje y clasificación según se mostró en la Figura 40. La salida del generador de sustancias es enviada a la cinta transportadora. Esta a su vez envía las materias al sistema de embalaje, el cual dispone de dos salidas, una para las sustancias clasificadas como de primera calidad y otra para las de segunda calidad.

Entre las líneas 9 y 13 se describen las características que tendrá el generador de sustancias. Aquí se establece la generación del valor 1 con una distribución exponencial con una media de 3 segundos.

Entre las líneas 15 y 38 se describe la cinta transportadora como un autómata celular de 1x6 celdas. En las líneas 31 a 35 se define el comportamiento para las celdas, que comprende el transporte de las sustancias sobre la cinta, modelado como el desplazamiento de los valores de las celdas hacia la derecha mientras la celda destino este libre y no sea el final de la cinta; y en caso de que llegue al final de la misma el valor será enviado a la embaladora (a través del puerto de salida *Out*). Como el autómata celular que representa a la cinta transportadora es no toroidal, se debe especificar el comportamiento que tendran las celdas pertenecientes al borde del modelo celular. Esto se realiza mediante el uso de la función *isUndefined*. La regla de la línea 33 permite que un elemento avance sobre la cinta transportadora de izquierda a derecha. Para cada celda que evalué la regla, si la misma se encuentra vacía (representado por el valor 0), tomará el valor de la celda ubicada a su izquierda. Claramente, esta función debe ser realizada por todas las celdas excepto para la celda (0, 0), debido a que por ser la celda ubicada más a la izquierda dentro del modelo, no tiene una celda a su izquierda. Esto se logra verificando que el valor del vecino (0, -1) no sea indefinido. Para la celda (0, 0) del modelo celular el valor del vecino (0, -1) es indefinido debido a que el autómata celular es no toroidal y se está referenciando a una celda fuera del espacio celular.
En la regla de la línea 34 también se utiliza la función isUndefined, en este caso para identificar a la celda ubicada más a la derecha dentro del autómata celular y poder darle un comporamiento distinto que al resto de las celdas. Cuando esta celda tenga un elemento (es decir, tenga un valor distinto a 0), su estado pasará a ser 0 y tendrá una demora mayor que para el resto de los casos, representando así el envío de un elemento a la embaladora y clasificadora.

Para los mensajes externos provenientes del generador de sustancias, que ingresen en la celda (0,0) de la cinta transportadora, se les da un comportamiento especial (líneas 37 y 38), indicando que por cada mensaje obtenido, que siempre será con valor 1, se genere una sustancia cuyo porcentaje de carbono será de 30 % más un error establecido según una distribución normal con media 0 y varianza 2. Este porcentaje de carbono será el valor almacenado en la celda (0,0).

Entre las líneas 40 y 73 se define el comportamiento para la embaladora y clasificadora de sustancias, que esta representada por un modelo celular compuesto por 4 celdas. En las líneas 68 a 73 se definen los comportamientos para las celdas (0,0) y (1,0) cuando llega una sustancia proveniente de la cinta transportadora, las cuales almacenan el total de carbono de las sustancias (en gramos) y la cantidad de fracciones obtenidas, respectivamente. En las líneas 61 a 66 se define el comportamiento genérico para todas las celdas. Aquí, cuando la cantidad de fracciones llega a 10 unidades, se

verifica si el porcentaje de carbono de las mismas está en el intervalo [29, 31]. Si es así, su valor es almacenado en la celda (0,1) la cual se comunica con el puerto de salida *outPrimeraSeleccion*; mientras que si no lo está se envía a la celda (1,1) que se comunica con el puerto *outSegundaSeleccion*.

Claramente, cada una de las 4 celdas que componen al modelo que representa la embaladora y clasificadora de sustancias tienen un comportamiento específico. Para poder identificar a cada una de estas celdas, y darle el comportamiento adecuado, se hace uso de la función *isUndefined*. En la línea 62, la condición de la regla verifica si los vecinos (1,0) y (0,-1) de la celda son indefinidos. Esta situación sólo es válida cuando se evalúa la función de cómputo local para la celda (1,0) del autómata celular, debido a que la referencia a un vecino fuera del espacio celular en caso de que el autómata celular sea no toroidal, devuelve el valor indefinido. De forma análoga, en la línea 63 se identifica a la celda (0,0) del modelo celular, validando que los vecinos (-1,0) y (0,-1) sean indefinidos. En la línea 64 se verifica si los vecinos (-1,0) y (0,1) son indefinidos para poder identificar a la celda (0,1) del autómata celular, mientras que en la línea 65 se verifica que los vecinos (1,0) y (0,1) sean indefinidos para identificar a la celda (1,1).

El resumen de los resultados generados por la herramienta, luego de 10 minutos de tiempo simulado, se muestra en la Figura 42.

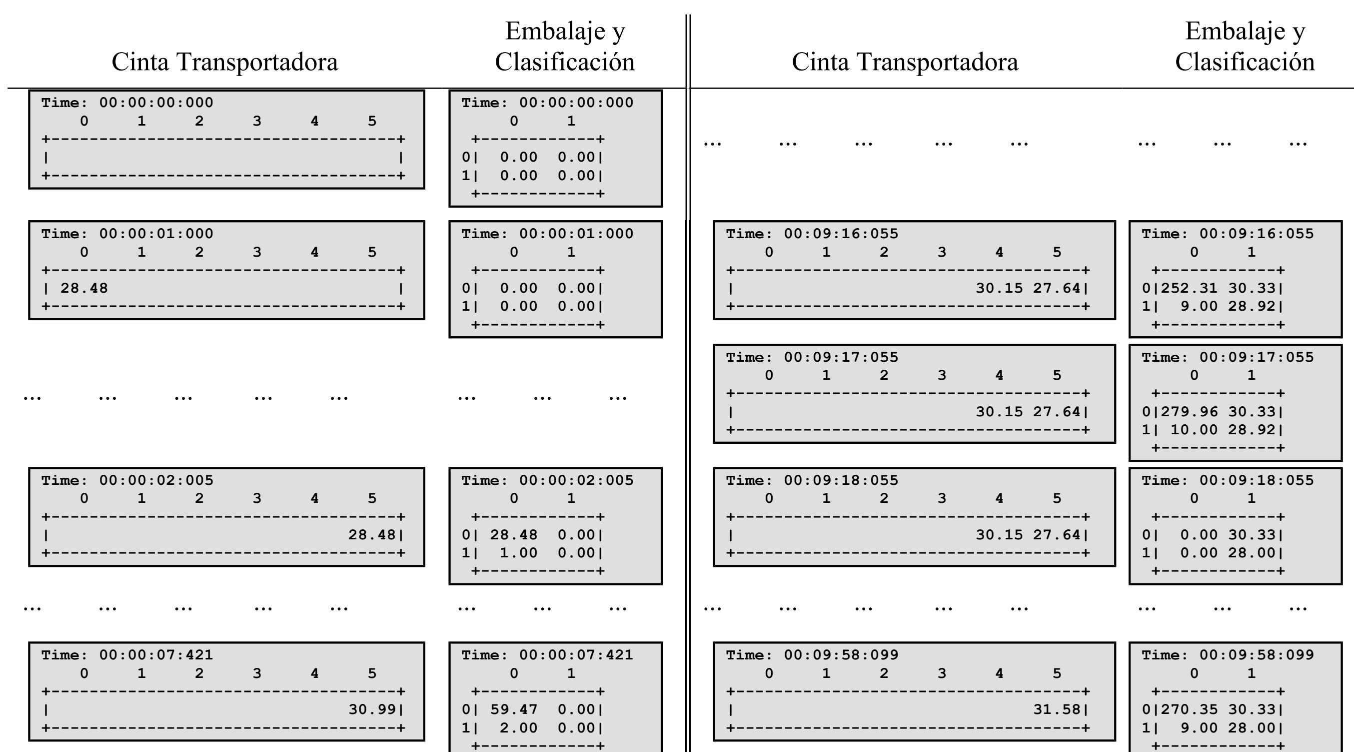


Figura 42 - Resultados para el Sistema de Embalaje y Clasificación de Materias Primas

Puede apreciarse que en la hora de simulación 00:00:02:005, la primer sustancia se envía a la embaladora, que registra la cantidad de carbono que posee, así como la cantidad de fracciones recibidas. En 00:00:07:421 recibe la segunda sustancia, acumulando la cantidad de carbono para las 2 fracciones recibidas.

n 00:09:17:055 puede observarse un caso donde la fracción recibida es la última para completar el kilogramo. En ese caso se actualizan los valores correspondientes y un segundo más tarde, luego de comprobar que la cantidad de carbono corresponde al 27.996 % de la sustancia, se coloca su valor en la celda (1,1) la cual se comunica con el puerto de salida para la segunda calidad, generando una salida en el modelo acoplado de mayor nivel.

Finalmente, cuando el tiempo alcanza los 10 minutos, la simulación se detiene. Los valores enviados por los puertos *outPrimeraSelección* y *outSegundaSelección* son mostrados en la Figura 43

Tiempo	Puerto de Salida	% de Carbono
00:00:35:056	OutPrimeraSelección	29.5468

00:01:11:041	OutPrimeraSelección	30.3376
00:01:41:051	OutPrimeraSelección	30.3913
00:02:11:061	OutPrimeraSelección	29.8409
00:02:43:656	OutPrimeraSelección	29.7743
00:03:27:317	OutPrimeraSelección	29.7819
00:03:57:974	OutPrimeraSelección	29.8887
00:04:35:148	OutPrimeraSelección	30.0995
00:05:05:158	OutSegundaSelección	31.5097
00:05:35:168	OutPrimeraSelección	29.7525
00:06:05:178	OutPrimeraSelección	29.0363
00:06:43:939	OutSegundaSelección	28.9098
00:07:15:934	OutSegundaSelección	28.9176
00:07:47:991	OutPrimeraSelección	30.3644
00:08:18:001	OutPrimeraSelección	29.3838
00:08:48:045	OutPrimeraSelección	30.3320
00:09:18:055	OutSegundaSelección	27.9956

Figura 43 – Resultados de la Simulación sobre Embalaje y Clasificación de Sustancias

14.10 Pinball

Este ejemplo consiste de un modelo celular de dos dimensiones que representa un pinball. El modelo cuenta con una única pelotita, que se mueve por el tablero en ocho direcciones posibles, cambiando de dirección al chocar contra una pared. Los posibles valores de estado para las celdas son:

0: Indica que la celda se encuentra vacía.
1: Indica que la celda contiene a la pelotita, y que la misma se dirige en dirección noreste.
2: La celda contiene a la pelotita y se dirige en dirección norte.
3: La celda contiene a la pelotita y se dirige en dirección noroeste.
4: La celda contiene a la pelotita y se dirige en dirección oeste.
5: La celda contiene a la pelotita y se dirige en dirección sudoeste.
6: La celda contiene a la pelotita y se dirige en dirección sur.
7: La celda contiene a la pelotita y se dirige en dirección sudeste.
8: La celda contiene a la pelotita y se dirige en dirección este.
9: Indica que la celda contiene una pared.

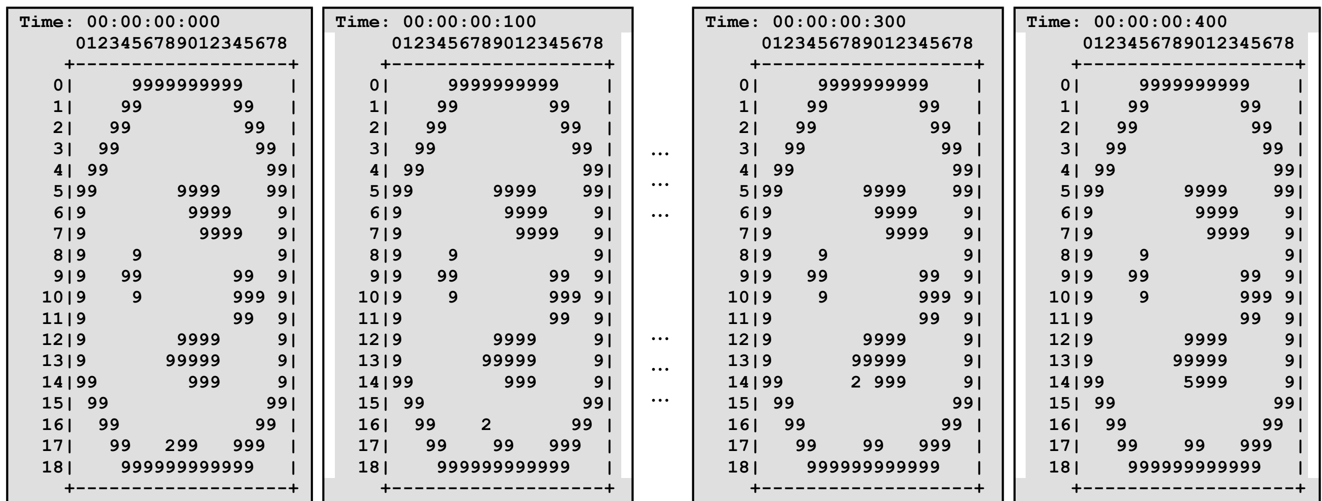
La implementación del modelo en el lenguaje provisto por la herramienta se muestra en la Figura 44. Las reglas consideran todos los casos posibles que pudieran suceder al colisionar con una pared desde distintas posiciones y con diferentes direcciones.

```
[top]
components : pinball

[Pinball]
type : cell
width : 19
height : 19
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue :  0      0000099999999990000
initialrowvalue :  1      0000990000000099000
initialrowvalue :  2      0009900000000009900
initialrowvalue :  3      0099000000000000990
initialrowvalue :  4      0990000000000000099
initialrowvalue :  5      9900000009999000099
initialrowvalue :  6      9000000000999900009
```

```
initialrowvalue :  7      9000000000099990009
initialrowvalue :  8      9000090000000000009
initialrowvalue :  9      9000990000000099009
initialrowvalue :  10     9000090000000099909
initialrowvalue :  11     9000000000000099009
initialrowvalue :  12     9000000009999000009
initialrowvalue :  13     9000000099999000009
initialrowvalue :  14     9900000000999000009
initialrowvalue :  15     0990000000000000099
initialrowvalue :  16     0099000000000000990
initialrowvalue :  17     0009900029900099900
initialrowvalue :  18     0000999999999999000
localtransition : pinball-rule

[pinball-rule]
rule : 9 100 { (0,0) = 9 }

rule : 1 100 { (0,0) = 0 and (1,1) = 1 }
rule : 4 100 { (0,0) = 0 and (0,-1) = 9 and (1,0) = 1 }
rule : 7 100 { (0,0) = 0 and (-1,0) = 9 and (-1,1) = 9 and (0,1) = 1 }
rule : 5 100 { (0,0) = 1 and (-1,0) = 9 and (0,-1) = 9 }

rule : 2 100 { (0,0) = 0 and (1,0) = 2 }
rule : 5 100 { (0,0) = 0 and (0,-1) = 2 and (-1,-1) = 9 }
rule : 7 100 { (0,0) = 2 and (-1,0) = 9 and (0,1) = 9 }

rule : 3 100 { (1,-1) = 3 and (0,0) = 0 }
rule : 6 100 { (0,0) = 0 and (-1,0) = 9 and (0,-1) = 3 }
rule : 1 100 { (0,0) = 0 and (1,0) = 3 and (0,1) = 9 and (1,1) = 9 }
rule : 7 100 { (0,0) = 3 and (-1,0) = 9 and (0,1) = 9 }

rule : 4 100 { (0,0) = 0 and (0,-1) = 4 }
rule : 7 100 { (0,0) = 0 and (-1,0) = 4 and (-1,1) = 9 }
rule : 1 100 { (0,0) = 4 and (0,1) = 9 and (1,0) = 9 }

rule : 5 100 { (0,0) = 0 and (-1,-1) = 5 }
rule : 8 100 { (0,0) = 0 and (-1,0) = 5 and (0,1) = 9 }
rule : 3 100 { (0,0) = 0 and (0,-1) = 5 and (1,-1) = 9 and (1,0) = 9 }
rule : 1 100 { (0,0) = 5 and (1,0) = 9 and (0,1) = 9 }

rule : 6 100 { (0,0) = 0 and (-1,0) = 6 }
rule : 1 100 { (0,0) = 0 and (0,1) = 6 and (1,1) = 9 }
rule : 3 100 { (0,0) = 6 and (0,-1) = 9 and (1,0) = 9 }

rule : 7 100 { (0,0) = 0 and (-1,1) = 7 }
rule : 1 100 { (0,0) = 0 and (1,0) = 9 and (0,1) = 7 }
rule : 5 100 { (0,0) = 0 and (-1,0) = 7 and (0,-1) = 9 and (-1,-1) = 9 }
rule : 3 100 { (0,0) = 7 and (0,-1) = 9 and (1,0) = 9 }

rule : 8 100 { (0,0) = 0 and (0,1) = 8 }
rule : 3 100 { (0,0) = 0 and (1,-1) = 9 and (1,0) = 8 }
rule : 5 100 { (0,0) = 8 and (0,-1) = 9 and (-1,0) = 9 }

rule : 0 100 { t }
```

Figura 44 – Implementación del modelo del *Pinball*

El resumen de los resultados generados por la herramienta puede verse en la Figura 45.



```
Time: 00:00:00:000
     0123456789012345678
    +-------------------+
   0|     9999999999    |
   1|    99        99   |
   2|   99          99  |
   3|  99            99 |
   4| 99              99|
   5|99       9999    99|
   6|9         9999    9|
   7|9          9999   9|
   8|9    9            9|
   9|9   99        99  9|
  10|9    9        999 9|
  11|9             99  9|
  12|9        9999     9|
  13|9       99999     9|
  14|99        999     9|
  15| 99              99|
  16|  99            99 |
  17|   99   299   999  |
  18|    999999999999   |
    +-------------------+
```

```
Time: 00:00:00:100
     0123456789012345678
    +-------------------+
   0|     9999999999    |
   1|    99        99   |
   2|   99          99  |
   3|  99            99 |
   4| 99              99|
   5|99       9999    99|
   6|9         9999    9|
   7|9          9999   9|
   8|9    9            9|
   9|9   99        99  9|
  10|9    9        999 9|
  11|9             99  9|
  12|9        9999     9|
  13|9       99999     9|
  14|99        999     9|
  15| 99              99|
  16|  99    2       99 |
  17|   99    99   999  |
  18|    999999999999   |
    +-------------------+
```

…

```
Time: 00:00:00:300
     0123456789012345678
    +-------------------+
   0|     9999999999    |
   1|    99        99   |
   2|   99          99  |
   3|  99            99 |
   4| 99              99|
   5|99       9999    99|
   6|9         9999    9|
   7|9          9999   9|
   8|9    9            9|
   9|9   99        99  9|
  10|9    9        999 9|
  11|9             99  9|
  12|9        9999     9|
  13|9       99999     9|
  14|99      2 999     9|
  15| 99              99|
  16|  99            99 |
  17|   99    99   999  |
  18|    999999999999   |
    +-------------------+
```

```
Time: 00:00:00:400
     0123456789012345678
    +-------------------+
   0|     9999999999    |
   1|    99        99   |
   2|   99          99  |
   3|  99            99 |
   4| 99              99|
   5|99       9999    99|
   6|9         9999    9|
   7|9          9999   9|
   8|9    9            9|
   9|9   99        99  9|
  10|9    9        999 9|
  11|9             99  9|
  12|9        9999     9|
  13|9       99999     9|
  14|99       5999     9|
  15| 99              99|
  16|  99            99 |
  17|   99    99   999  |
  18|    999999999999   |
    +-------------------+
```

Figura 45 – Resumen de los resultados arrojados para la simulación del *Pinball*

Inicialmente la pelotita se encuentra en con dirección hacia el norte (2). En el tiempo de simulación 00:00:00:100 la pelotita avanza hacia la celda ubicada al norte y sigue manteniendo su misma dirección debido a que no colisionó con ninguna pared. La pelotita sigue avanzando hasta que en el tiempo 00:00:00:400 se produce una colisión con una de las paredes,. Inmediatamente se cambia el rumbo de la misma y se la dirige hacia el sudoeste.

14.11 Pista de Baile

El siguiente ejemplo representa una pista de baile, donde una persona comienza a bailar recorriendo la pista, y al pasar cerca de otras personas lo siguen, generando un "trencito".

La pista de baile será representada por un autómata celular bidimensional de 5x5 celdas. Las celdas conteniendo el valor 0 indican la ausencia de una persona en esa ubicación. Para representar a las personas en movimiento se utiliza un valor real codificado de la siguiente forma: la parte entera del valor identifica a la persona, mientras que la parte decimal representa la dirección a la cual se dirige. La persona identificada con el valor 1 representa a quien inicia y encabeza el trencito. La persona identificada con 2 sigue a la identificada con 1, mientras que la identificada con 3 sigue a la 2, y la 4 sigue a la 3. De esta forma se genera el trencito. Por otra parte, los valores decimales que representan el movimiento de las personas son:

0 = indica que la persona esta inmóvil.
1 = indica que la persona se dirige hacia el sur.
2 = la persona se dirige hacia el norte.
3 = la persona se dirige hacia el este.
4 = la persona se dirige hacia el oeste.

De esta forma, con un número real puede identificarse a una persona y la dirección a la cual se dirige. Por ejemplo: el valor 2.3 indica que la persona identificada con 2 se dirige hacia el este (valor 3).

En la Figura 46 se muestra la implementación en *N-CD++* del modelo previamente descripto, mientras que en la Figura 47 se muestra el contenido del archivo *dance.val* definiendo el estado inicial del autómata celular.

```
[top]
components : dance

[dance]
type : cell
width : 5
height : 5
delay : transport
defaultDelayTime  : 100
```

```
border : wrapped
neighbors : dance(-2,-2) dance(-2,-1) dance(-2,0) dance(-2,1) dance(-2,2)
neighbors : dance(-1,-2) dance(-1,-1) dance(-1,0) dance(-1,1) dance(-1,2)
neighbors : dance(0,-2)  dance(0,-1)  dance(0,0)  dance(0,1)  dance(0,2)
neighbors : dance(1,-2)  dance(1,-1)  dance(1,0)  dance(1,1)  dance(1,2)
neighbors : dance(2,-2)  dance(2,-1)  dance(2,0)  dance(2,1)  dance(2,2)
initialvalue : 0
initialCellsValue : dance.val
localtransition : dance-rule

[dance-rule]
rule : { 1 }                     100 { (-1,0) != 0 and (0,0) = 1.2 }
rule : { 1 }                     100 { (1,0)  != 0 and (0,0) = 1.1 }
rule : { 1 }                     100 { (0,-1) != 0 and (0,0) = 1.4 }
rule : { 1 }                     100 { (0,1)  != 0 and (0,0) = 1.3 }
rule : { 1 }                     100 { (-1,0) = 1.1 and (0,0) = 0 }
rule : { 1 }                     100 { (1,0)  = 1.2 and (0,0) = 0 }
rule : { 1 }                     100 { (0,-1) = 1.3 and (0,0) = 0 }
rule : { 1 }                     100 { (0,1)  = 1.4 and (0,0) = 0 }
rule : { 1.1 + randInt(3) / 10 } 100 { (0,0) = 1 }

rule : { 2 }   100 { (-1,0) = 2.1 and (0,0) = 0 }
rule : { 2.1 } 100 { (0,0) = 2.1 and ( (2,0) = 1.2 or (1,0) != 0 or (1,1) = 1.4 or
                     (1,-1) = 1.3 ) }
rule : { 2.1 } 100 { (0,0) = 2 and ( trunc((1,0)) = 1 or trunc((1,-1)) = 1 or
                      trunc((1,1)) = 1 or trunc((2,-2)) = 1 or trunc((2,-1)) = 1 or
                      trunc((2,0)) = 1 or trunc((2,1)) = 1 or trunc((2,2)) = 1 ) }

rule : { 2 }   100 { (1,0) = 2.2 and (0,0) = 0 }
rule : { 2.2 } 100 { (0,0) = 2.2 and ( (-2,0) = 1.1 or (-1,0) != 0 or (-1,1) = 1.4
                     or (-1,-1) = 1.3 ) }
rule : { 2.2 } 100 { (0,0) = 2 and ( trunc((-1,0)) = 1 or trunc((-1,-1)) = 1 or
                     trunc((-1,1)) = 1 or trunc((-2,-2)) = 1 or trunc((-2,-1)) = 1
                     or trunc((-2,0))= 1 or trunc((-2,1))= 1 or trunc((-2,2))= 1 )}

rule : { 2 }   100 { (0,1) = 2.3 and (0,0) = 0 }
rule : { 2.3 } 100 { (0,0) = 2.3 and ( (0,-2) = 1.4 or (0,-1) != 0 or (-1,-1) = 1.1
                     or (1,-1) = 1.2 ) }
rule : { 2.3 } 100 { (0,0) = 2 and ( trunc((0,-1)) = 1 or trunc((-1,-2)) = 1 or
                     trunc((0,-2)) = 1 or trunc((1,-2)) = 1 ) }

rule : { 2 }   100 { (0,-1) = 2.4 and (0,0) = 0 }
rule : { 2.4 } 100 { (0,0) = 2.4 and ( (0,2) = 1.3 or (0,1) != 0 or (-1,1) = 1.1 or
                     (1,1) = 1.2 ) }
rule : { 2.4 } 100 { (0,0) = 2 and ( trunc((0,1)) = 1 or trunc((-1,2)) = 1 or
                     trunc((0,2)) = 1 or trunc((1,2)) = 1 ) }

rule : { 3 }   100 { (-1,0) = 3.1 and (0,0) = 0 }
rule : { 3.1 } 100 { (0,0) = 3.1 and ( (2,0) = 1.2 or (2,0) = 2.2 or (1,0) != 0 or
                     (1,1) = 1.4 or (1,1) = 2.4 or (1,-1) = 1.3 or (1,-1) = 2.3 ) }
rule : { 3.1 } 100 { (0,0) = 3 and ( trunc((1,0)) = 2 or trunc((1,-1)) = 2 or
                     trunc((1,1)) = 2 or trunc((2,-2)) = 2 or trunc((2,-1)) = 2 or
                     trunc((2,0)) = 2 or trunc((2,1)) = 2 or trunc((2,2)) = 2 ) }

rule : { 3 }   100 { (1,0) = 3.2 and (0,0) = 0 }
rule : { 3.2 } 100 { (0,0) = 3.2 and ( (-2,0) = 1.1 or (-2,0) = 2.1 or (-1,0) != 0
                     or (-1,1) = 1.4 or (-1,1) = 2.4 or (-1,-1) = 1.3 or
                     (-1,-1) = 2.3 ) }
rule : { 3.2 } 100 { (0,0) = 3 and ( trunc((-1,0)) = 2 or trunc((-1,-1)) = 2 or
                     trunc((-1,1)) = 2 or trunc((-2,-2)) = 2 or trunc((-2,-1)) = 2
                     or trunc((-2,0))= 2 or trunc((-2,1))= 2 or trunc((-2,2))= 2 )}

rule : { 3 }   100 { (0,1) = 3.3 and (0,0) = 0 }
rule : { 3.3 } 100 { (0,0) = 3.3 and ( (0,-2) = 1.4 or (0,-2) = 2.4 or (0,-1) != 0
```

```
                     or (-1,-1) = 1.1 or (-1,-1) = 2.1 or (1,-1) = 1.2 or
                     (1,-1) = 2.2 ) }
rule : { 3.3 } 100 { (0,0) = 3 and ( trunc((0,-1)) = 2 or trunc((-1,-2)) = 2 or
                     trunc((0,-2)) = 2 or trunc((1,-2)) = 2 ) }

rule : { 3 }   100 { (0,-1) = 3.4 and (0,0) = 0 }
rule : { 3.4 } 100 { (0,0) = 3.4 and ( (0,2) = 1.3 or (0,2) = 2.3 or (0,1) != 0 or
                     (-1,1) = 1.1 or (-1,1) = 2.1 or (1,1) = 1.2 or (1,1) = 2.2 ) }
rule : { 3.4 } 100 { (0,0) = 3 and ( trunc((0,1)) = 2 or trunc((-1,2)) = 2 or
                     trunc((0,2)) = 2 or trunc((1,2)) = 2 ) }

rule : { 4 }   100 { (-1,0) = 4.1 and (0,0) = 0 }
rule : { 4.1 } 100 { (0,0) = 4.1 and ( (2,0) = 1.2 or (2,0) = 2.2 or (2,0) = 3.2 or
                     (1,0) != 0 or (1,1) = 1.4 or (1,1) = 2.4 or (1,1) = 3.4 or
                     (1,-1) = 1.3 or (1,-1) = 2.3 or (1,-1) = 3.3 ) }
rule : { 4.1 } 100 { (0,0) = 4 and ( trunc((1,0)) = 3 or trunc((1,-1)) = 3 or
                     trunc((1,1)) = 3 or trunc((2,-2)) = 3 or trunc((2,-1)) = 3 or
                     trunc((2,0)) = 3 or trunc((2,1)) = 3 or trunc((2,2)) = 3 ) }

rule : { 4 }   100 { (1,0) = 4.2 and (0,0) = 0 }
rule : { 4.2 } 100 { (0,0) = 4.2 and ( (-2,0) = 1.1 or (-2,0) = 2.1 or (-2,0) = 3.1
                     or (-1,0) != 0 or (-1,1) = 1.4 or (-1,1) = 2.4 or (-1,1) = 3.4
                     or (-1,-1) = 1.3 or (-1,-1) = 2.3 or (-1,-1) = 3.3 ) }
rule : { 4.2 } 100 { (0,0) = 4 and ( trunc((-1,0)) = 3 or trunc((-1,-1)) = 3 or
                     trunc((-1,1)) = 3 or trunc((-2,-2)) = 3 or trunc((-2,-1)) = 3
                     or trunc((-2,0))= 3 or trunc((-2,1))= 3 or trunc((-2,2))= 3 )}

rule : { 4 }   100 { (0,1) = 4.3 and (0,0) = 0 }
rule : { 4.3 } 100 { (0,0) = 4.3 and ( (0,-2) = 1.4 or (0,-2) = 2.4 or (0,-2) = 3.4
                     or (0,-1) != 0 or (-1,-1) = 1.1 or (-1,-1) = 2.1 or
                     (-1,-1) = 3.1 or (1,-1) = 1.2 or (1,-1) = 2.2 or (1,-1)= 3.2)}
rule : { 4.3 } 100 { (0,0) = 4 and ( trunc((0,-1)) = 3 or trunc((-1,-2)) = 3 or
                     trunc((0,-2)) = 3 or trunc((1,-2)) = 3 ) }

rule : { 4 }   100 { (0,-1) = 4.4 and (0,0) = 0 }
rule : { 4.4 } 100 { (0,0) = 4.4 and ( (0,2) = 1.3 or (0,2) = 2.3 or (0,2) = 3.3 or
                     (0,1) != 0 or (-1,1) = 1.1 or (-1,1) = 2.1 or (-1,1) = 3.1 or
                     (1,1) = 1.2 or (1,1) = 2.2 or (1,1) = 3.2 ) }
rule : { 4.4 } 100 { (0,0) = 4 and ( trunc((0,1)) = 3 or trunc((-1,2)) = 3 or
                     trunc((0,2)) = 3 or trunc((1,2)) = 3 ) }

rule : { (0,0) } 100 { (0,0) = trunc((0,0)) }
rule : 0         100 { t }
```

Figura 46 – Implementación del Modelo de la Pista de Baile

```
(4,2)  = 1.3
(1,4) = 4
(3,2) = 3
(1,1)  = 2
```

Figura 47 – Contenido del archivo *dance.val* definiendo el estado inicial para el modelo de la Pista de Baile

La salida generada por la herramienta se muestra en la Figura 48. Inicialmente la persona identificada con el valor 1 se encuentra en movimiento, dirigiéndose hacia el este, mientras que el resto de las personas están detenidas en distintos puntos de la pista de baile. En el tiempo 00:00:00:100 puede verse que el individuo 2 cambia su estado para dirigirse hacia el norte y acercarse así al individuo 1. Por su parte, el individuo 3 también se dirige hacia el norte, mientras que el 4 al desear acercarse al 3 y se dirige al sur. Luego del primer segundo de simulación puede apreciarse como se va formando el trencito, donde el individuo 1 encabeza el mismo.

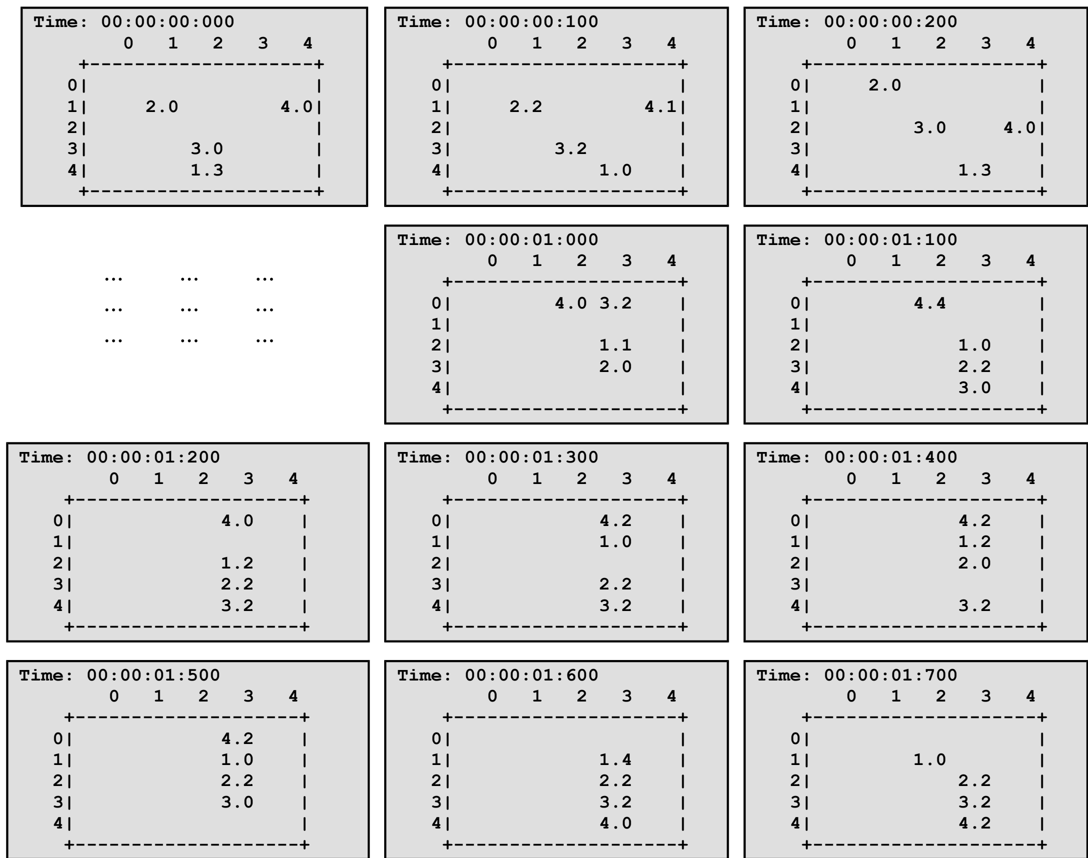


Figura 48 – Resumen de los resultados generados por la herramienta para la simulación de la Pista de Baile

14.12 Mosquitos

Ser intenta modelar un ambiente de tres dimensiones, representado por un autómata celular donde existen celdas que conectadas a un puerto por el cual ingresan datos que producirán valores para la presión atmosférica. La presión para el resto de las celdas se calcula como el promedio del vecindario. A partir del valor de la presión, es posible calcular la temperatura para la celda. Además, en el ambiente hay mosquitos que vuelan desde una celda (x, y, z) a la celda (x, y, z+1). Si la temperatura de una celda es menor a 16 grados centígrados, el mosquito que ocupa la celda muere, de lo contrario sigue volando.

La modelización consiste en un autómata celular de 4 dimensiones, para representar un ambiente de tamaño 4x4x4. El autómata celular tiene tamaño (3, 4, 4, 4), y cada uno de los 3-hipercubos (a, x, y, z), con a = 1, 2, 3, son utilizados para almacenar una variable distinta del modelo. La presión atmosférica de la posición (x, y, z) del ambiente se almacena en (0, x, y, z) y su temperatura se almacena en (1, x, y, x). La celda (2, x, y, z) se utiliza para indicar la presencia de un mosquito en la posición (x, y, z) del ambiente, y puede tener un valor 0 indicando que la celda esta vacía, ó 1 indicando la existencia de un mosquito en la misma. Una celda puede estar ocupada por un solo mosquito.

En la Figura 49 se muestra la implementación escrita para *N-CD++*. El puerto de entrada *inPort* permite el ingreso de datos que generarán los valores para la presión atmosférica de la celda (0,1,1,1). Cuando un evento llegue por este puerto se ejecutará la función *setPresion*, que establecerá un valor aleatorio con distribución uniforme perteneciente al intervalo [0.1, 1.3]. Los eventos externos se definen en el archivo *eventos.ev*, definido en la Figura 50.

Para dar comportamiento a cada 3-hipercubo, correspondientes a la temperatura y los mosquitos, se definen zonas, donde se asocia una función de cómputo local distinta para cada una de ellas. Para la zona que abarca las celdas que representan la temperatura se asocia la función *temp-rule*, que calcula el valor de la temperatura a partir de la presión; mientras que en la zona que modela a los mosquitos, se utiliza *mosquito-rule* para representar su desplazamiento, y hacer que mueran si la posición a la cual pertenecen en el modelo tiene una temperatura menor a 16

grados centígrados. La función de cómputo local utilizada por las celdas no incluidas en ninguna zona permite el cálculo de la presión atmosférica como el promedio de los valores de estado del vecindario, mediante la función *presion-rule*.

La Figura 51 define el estado inicial para el autómata celular. Se considera que inicialmente la presión atmosférica para todas las celdas es de 1 atmósfera. Esto implica que todas las celdas (0, x, y, z) ∀ x, y, z tengan valor 1. Además, la temperatura para todas las posiciones del ambiente es de 20 grados centígrados y hay 12 mosquitos distribuidos en distintas posiciones.

```
[top]
components : ambiente
in : inPort
link : inPort inputPresion@ambiente

[ambiente]
type : cell
dim : (3, 4, 4, 4)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : ambiente(0,-1,-1,-1) ambiente(0,-1,-1,0) ambiente(0,-1,-1,1)
neighbors : ambiente(0,-1,0,-1)  ambiente(0,-1,0,0)  ambiente(0,-1,0,1)
neighbors : ambiente(0,-1,1,-1)  ambiente(0,-1,1,0)  ambiente(0,-1,1,1)

neighbors : ambiente(0,0,-1,-1)  ambiente(0,0,-1,0)  ambiente(0,0,-1,1)
neighbors : ambiente(0,0,0,-1)   ambiente(0,0,0,0)   ambiente(0,0,0,1)
neighbors : ambiente(0,0,1,-1)   ambiente(0,0,1,0)   ambiente(0,0,1,1)

neighbors : ambiente(0,1,-1,-1)  ambiente(0,1,-1,0)  ambiente(0,1,-1,1)
neighbors : ambiente(0,1,0,-1)   ambiente(0,1,0,0)   ambiente(0,1,0,1)
neighbors : ambiente(0,1,1,-1)   ambiente(0,1,1,0)   ambiente(0,1,1,1)

neighbors : ambiente(-1,0,0,0)
neighbors : ambiente(1,0,0,0)

initialvalue : 0
in : inputPresion
link : inputPresion in@ambiente(0,1,1,1)
localtransition : presion-rule
initialCellsValue : mosquito.val
zone : temp-rule { (1,0,0,0)..(1,3,3,3) }
zone : mosquito-rule { (2,0,0,0)..(2,3,3,3) }
portInTransition : in@ambiente(0,1,1,1)  setPresion

[setPresion]
rule : { uniform(0.1,1.3) } 0 { t }

[presion-rule]
rule : { ( (0,-1,-1,-1) + (0,-1,-1,0) + (0,-1,-1,1) + (0,-1,0,-1) + (0,-1,0,0) +
           (0,-1,0,1) + (0,-1,1,-1) + (0,-1,1,0) + (0,-1,1,1) + (0,0,-1,-1) +
           (0,0,-1,0) + (0,0,-1,1) + (0,0,0,-1) + (0,0,0,0) + (0,0,0,1) +
           (0,0,1,-1) + (0,0,1,0) + (0,0,1,1) + (0,1,-1,-1) + (0,1,-1,0) +
           (0,1,-1,1) + (0,1,0,-1) + (0,1,0,0) + (0,1,0,1) + (0,1,1,-1) +
           (0,1,1,0) + (0,1,1,1) ) / 27 } 1000 { t }

[temp-rule]
rule : { 273 * ( (-1,0,0,0) - 1 ) + 20 } 1000 { t }

[mosquito-rule]
rule : 1 1000 { (0,0,0,-1) = 1 AND (-1,0,0,0) >= 16 }
rule : 0 1000 { t }
```

Figura 49 – Implementación del modelo de los Mosquitos

```
00:00:01:000 inPort 1
00:00:05:000 inPort 1
00:00:11:000 inPort 1
00:00:14:000 inPort 1
00:00:17:000 inPort 1
00:00:21:000 inPort 1
00:00:25:000 inPort 1
00:00:27:000 inPort 1
00:00:29:000 inPort 1
```

Figura 50 – Contenido del archivo *Eventos.ev* detallando los eventos externos para el problema de los mosquitos

```
(0,0,0,0) = 1
(0,0,0,1) = 1
(0,0,0,2) = 1
(0,0,0,3) = 1
(0,0,1,0) = 1
(0,0,1,1) = 1
(0,0,1,2) = 1
(0,0,1,3) = 1
(0,0,2,0) = 1
(0,0,2,1) = 1
(0,0,2,2) = 1
(0,0,2,3) = 1
(0,0,3,0) = 1
(0,0,3,1) = 1
(0,0,3,2) = 1
(0,0,3,3) = 1
(0,1,0,0) = 1
(0,1,0,1) = 1
(0,1,0,2) = 1
(0,1,0,3) = 1
(0,1,1,0) = 1
(0,1,1,1) = 1
(0,1,1,2) = 1
(0,1,1,3) = 1
(0,1,2,0) = 1
(0,1,2,1) = 1
(0,1,2,2) = 1
(0,1,2,3) = 1
```

```
(0,1,3,0) = 1
(0,1,3,1) = 1
(0,1,3,2) = 1
(0,1,3,3) = 1
(0,2,0,0) = 1
(0,2,0,1) = 1
(0,2,0,2) = 1
(0,2,0,3) = 1
(0,2,1,0) = 1
(0,2,1,1) = 1
(0,2,1,2) = 1
(0,2,1,3) = 1
(0,2,2,0) = 1
(0,2,2,1) = 1
(0,2,2,2) = 1
(0,2,2,3) = 1
(0,2,3,0) = 1
(0,2,3,1) = 1
(0,2,3,2) = 1
(0,2,3,3) = 1
(0,3,0,0) = 1
(0,3,0,1) = 1
(0,3,0,2) = 1
(0,3,0,3) = 1
(0,3,1,0) = 1
(0,3,1,1) = 1
(0,3,1,2) = 1
(0,3,1,3) = 1
```

```
(0,3,2,0) = 1
(0,3,2,1) = 1
(0,3,2,2) = 1
(0,3,2,3) = 1
(0,3,3,0) = 1
(0,3,3,1) = 1
(0,3,3,2) = 1
(0,3,3,3) = 1
(1,0,0,0) = 20
(1,0,0,1) = 20
(1,0,0,2) = 20
(1,0,0,3) = 20
(1,0,1,0) = 20
(1,0,1,1) = 20
(1,0,1,2) = 20
(1,0,1,3) = 20
(1,0,2,0) = 20
(1,0,2,1) = 20
(1,0,2,2) = 20
(1,0,2,3) = 20
(1,0,3,0) = 20
(1,0,3,1) = 20
(1,0,3,2) = 20
(1,0,3,3) = 20
(1,1,0,0) = 20
(1,1,0,1) = 20
(1,1,0,2) = 20
(1,1,0,3) = 20
```

```
(1,1,1,0) = 20
(1,1,1,1) = 20
(1,1,1,2) = 20
(1,1,1,3) = 20
(1,1,2,0) = 20
(1,1,2,1) = 20
(1,1,2,2) = 20
(1,1,2,3) = 20
(1,1,3,0) = 20
(1,1,3,1) = 20
(1,1,3,2) = 20
(1,1,3,3) = 20
(1,2,0,0) = 20
(1,2,0,1) = 20
(1,2,0,2) = 20
(1,2,0,3) = 20
(1,2,1,0) = 20
(1,2,1,1) = 20
(1,2,1,2) = 20
(1,2,1,3) = 20
(1,2,2,0) = 20
(1,2,2,1) = 20
(1,2,2,2) = 20
(1,2,2,3) = 20
(1,2,3,0) = 20
(1,2,3,1) = 20
(1,2,3,2) = 20
(1,2,3,3) = 20
```

```
(1,3,0,0) = 20
(1,3,0,1) = 20
(1,3,0,2) = 20
(1,3,0,3) = 20
(1,3,1,0) = 20
(1,3,1,1) = 20
(1,3,1,2) = 20
(1,3,1,3) = 20
(1,3,2,0) = 20
(1,3,2,1) = 20
(1,3,2,2) = 20
(1,3,2,3) = 20
(1,3,3,0) = 20
(1,3,3,1) = 20
(1,3,3,2) = 20
(1,3,3,3) = 20
(2,0,0,0) = 0
(2,0,0,1) = 0
(2,0,0,2) = 0
(2,0,0,3) = 1
(2,0,1,0) = 0
(2,0,1,1) = 0
(2,0,1,2) = 0
(2,0,1,3) = 0
(2,0,2,0) = 0
(2,0,2,1) = 0
(2,0,2,2) = 1
(2,0,2,3) = 0
```

```
(2,0,3,0) = 0
(2,0,3,1) = 0
(2,0,3,2) = 0
(2,0,3,3) = 1
(2,1,0,0) = 0
(2,1,0,1) = 0
(2,1,0,2) = 0
(2,1,0,3) = 0
(2,1,1,0) = 1
(2,1,1,1) = 0
(2,1,1,2) = 0
(2,1,1,3) = 0
(2,1,2,0) = 0
(2,1,2,1) = 0
(2,1,2,2) = 0
(2,1,2,3) = 0
(2,1,3,0) = 1
(2,1,3,1) = 0
(2,1,3,2) = 1
(2,1,3,3) = 0
(2,2,0,0) = 0
(2,2,0,1) = 0
(2,2,0,2) = 0
(2,2,0,3) = 1
(2,2,1,0) = 0
(2,2,1,1) = 0
(2,2,1,2) = 0
(2,2,1,3) = 1
```

```
(2,2,2,0) = 0
(2,2,2,1) = 0
(2,2,2,2) = 1
(2,2,2,3) = 0
(2,2,3,0) = 0
(2,2,3,1) = 0
(2,2,3,2) = 0
(2,2,3,3) = 0
(2,3,0,0) = 0
(2,3,0,1) = 1
(2,3,0,2) = 0
(2,3,0,3) = 0
(2,3,1,0) = 0
(2,3,1,1) = 1
(2,3,1,2) = 0
(2,3,1,3) = 0
(2,3,2,0) = 0
(2,3,2,1) = 0
(2,3,2,2) = 0
(2,3,2,3) = 0
(2,3,3,0) = 1
(2,3,3,1) = 0
(2,3,3,2) = 0
(2,3,3,3) = 0
```

Figura 51 – Contenido del archivo *mosquito.val* detallando el estado inicial para el problema de los mosquitos

El resumen de los resultados generados por la herramienta se muestra en la Figura 52. En el tiempo 00:00:01.000 el ingreso de un valor por el puerto de entrada *inPort* produce la generación de un valor aleatorio para la presión atmosférica en la celda (0, 1, 1, 1) del modelo celular, lo que producirá una alteración en la temperatura de la celda asociada. A su vez, puede apreciarse el desplazamiento del mosquito que se encuentra en la celda (2, 1, 1, 1). En el tiempo 00:00:02.000 se promedian los valores del vecindario de las celdas que representan la presión atmosférica, y se obtienen así nuevos valores para ellas. Al mismo tiempo, el valor de la temperatura, en la celda (1, 1, 1, 1) es recalculado, debido al cambio en la celda (0, 1, 1, 1) en el instante anterior. Llegando al tiempo 00:00:07.000 puede apreciarse que la temperatura en la celda (1, 1, 1, 0) es menor a 16 ° C. Esto hace que el mosquito que está en la posición (2, 1, 1, 3) muera al querer desplazarse a la posición (2, 1, 1, 0) en el tiempo 00:00:08.000.

```
Time:
00:00:00:000
...
(0,1,0,3) = 1
(0,1,1,0) = 1
(0,1,1,1) = 1
(0,1,1,2) = 1
...
(1,1,0,3) = 20
(1,1,1,0) = 20
(1,1,1,1) = 20
(1,1,1,2) = 20
...
(2,1,1,0) = 1
(2,1,1,1) = 0
(2,1,1,2) = 0
(2,1,1,3) = 0
...
```

```
Time:
00:00:01:000
...
(0,1,0,3) = 1
(0,1,1,0) = 1
(0,1,1,1) = 0.876
(0,1,1,2) = 1
...
(1,1,0,3) = 20
(1,1,1,0) = 20
(1,1,1,1) = 20
(1,1,1,2) = 20
...
(2,1,1,0) = 0
(2,1,1,1) = 1
(2,1,1,2) = 0
(2,1,1,3) = 0
...
```

```
Time:
00:00:02:000
...
(0,1,0,3) = 1
(0,1,1,0) = 0.995
(0,1,1,1) = 0.995
(0,1,1,2) = 0.995
...
(1,1,0,3) = 20
(1,1,1,0) = 20
(1,1,1,1) = -13.6
(1,1,1,2) = 20
...
(2,1,1,0) = 0
(2,1,1,1) = 0
(2,1,1,2) = 1
(2,1,1,3) = 0
...
```

...

```
Time:
00:00:07:000
...
(0,1,0,3) = 0.984
(0,1,1,0) = 0.978
(0,1,1,1) = 0.968
(0,1,1,2) = 0.978
...
(1,1,0,3) = 19.47
(1,1,1,0) = 11.37
(1,1,1,1) = 11.37
(1,1,1,2) = 11.37
...
(2,1,1,0) = 0
(2,1,1,1) = 0
(2,1,1,2) = 0
(2,1,1,3) = 1
...
```

```
Time:
00:00:08:000
...
(0,1,0,3) = 0.986
(0,1,1,0) = 0.984
(0,1,1,1) = 0.984
(0,1,1,2) = 0.984
...
(1,1,0,3) = 15.87
(1,1,1,0) = 14.07
(1,1,1,1) = 11.37
(1,1,1,2) = 14.07
...
(2,1,1,0) = 0
(2,1,1,1) = 0
(2,1,1,2) = 0
(2,1,1,3) = 0
...
```

Figura 52 – Resumen de los resultados generados para el problema de los mosquitos

Apéndice A – Nodos para las Operaciones y Funciones del Lenguaje

A continuación se detalla el listado de las distintas especializaciones de nodos que representan a las funciones y operadores del lenguaje de especificación de reglas de *N-CD++*.

Nodos para los Operadores Relacionales

typedef BinaryOpNode< REAL_Equal_to, BoolType, RealType > EqualNode ;
typedef BinaryOpNode< REAL_Not_equal_to, BoolType, RealType > NotEqualNode ;
typedef BinaryOpNode< REAL_Less, BoolType, RealType > LessNode ;
typedef BinaryOpNode< REAL_Greater, BoolType, RealType > GreaterNode ;
typedef BinaryOpNode< REAL_Less_equal, BoolType, RealType > LessEqualNode ;
typedef BinaryOpNode< REAL_Greater_equal, BoolType, RealType > GreaterEqualNode ;

Nodos para los Operadores Booleanos de la Lógica Trivalente

typedef UnaryOpNode< TVB_not, BoolType, BoolType > NOTNode ;
typedef BinaryOpNode< TVB_and, BoolType, BoolType > ANDNode ;
typedef BinaryOpNode< TVB_or, BoolType, BoolType > ORNode ;
typedef BinaryOpNode< TVB_xor, BoolType, BoolType > XORNode ;
typedef BinaryOpNode< TVB_imp, BoolType, BoolType > IMPNode ;
typedef BinaryOpNode< TVB_eqv, BoolType, BoolType > EQVNode ;

Nodos para las Funciones que devuelvan Valores Booleanos

typedef UnaryOpNode< REAL_Even, BoolType, RealType > FuncEVEN ;
typedef UnaryOpNode< REAL_Odd, BoolType, RealType > FuncODD ;
typedef UnaryOpNode< REAL_IsInt, BoolType, RealType > FuncISINT ;
typedef UnaryOpNode< REAL_IsPrime, BoolType, RealType > FuncISPRIME ;
typedef UnaryOpNode< REAL_IsUndefined, BoolType, RealType > FuncISUNDEFINED ;

Nodos para las Funciones que devuelvan Valores Reales

typedef BinaryOpNode< REAL_Plus, RealType, RealType > PlusNode ;
typedef BinaryOpNode< REAL_Minus, RealType, RealType > MinusNode ;
typedef BinaryOpNode< REAL_Divides, RealType, RealType > DividesNode ;
typedef BinaryOpNode< REAL_Multiplies, RealType, RealType > MultipliesNode ;
typedef UnaryOpNode< REAL_Tan, RealType, RealType > FuncTAN ;
typedef UnaryOpNode< REAL_Tanh, RealType, RealType > FuncTANH ;

```
typedef UnaryOpNode< REAL_Sqrt, RealType, RealType > FuncSQRT ;
typedef UnaryOpNode< REAL_Sinh, RealType, RealType > FuncSINH ;
typedef UnaryOpNode< REAL_Sin, RealType, RealType > FuncSIN ;
typedef UnaryOpNode< REAL_Round, RealType, RealType > FuncROUND ;
typedef UnaryOpNode< REAL_Fractional, RealType, RealType > FuncFRACTIONAL ;
typedef BinaryOpNode< REAL_Remainder, RealType, RealType > FuncREMAINDER ;
typedef BinaryOpNode< REAL_Power, RealType, RealType > FuncPOWER ;
typedef UnaryOpNode< REAL_Abs, RealType, RealType > FuncABS ;
typedef UnaryOpNode< REAL_Exp, RealType, RealType > FuncEXP ;
typedef UnaryOpNode< REAL_Ln, RealType, RealType > FuncLN ;
typedef UnaryOpNode< REAL_Log, RealType, RealType > FuncLOG ;
typedef UnaryOpNode< REAL_Cosh, RealType, RealType > FuncCOSH ;
typedef UnaryOpNode< REAL_Cos, RealType, RealType > FuncCOS ;
typedef UnaryOpNode< REAL_Sec, RealType, RealType > FuncSEC ;
typedef UnaryOpNode< REAL_Sech, RealType, RealType > FuncSECH ;
typedef UnaryOpNode< REAL_Trunc, RealType, RealType > FuncTRUNC ;
typedef UnaryOpNode< REAL_TruncUpper, RealType, RealType > FuncTRUNCUPPER ;
typedef UnaryOpNode< REAL_Atanh, RealType, RealType > FuncATANH ;
typedef UnaryOpNode< REAL_Atan, RealType, RealType > FuncATAN ;
typedef UnaryOpNode< REAL_Asinh, RealType, RealType > FuncASINH ;
typedef UnaryOpNode< REAL_Asin, RealType, RealType > FuncASIN ;
typedef UnaryOpNode< REAL_Acosh, RealType, RealType > FuncACOSH ;
typedef UnaryOpNode< REAL_Acos, RealType, RealType > FuncACOS ;
typedef BinaryOpNode< REAL_Max, RealType, RealType > FuncMAX ;
typedef BinaryOpNode< REAL_Min, RealType, RealType > FuncMIN ;
typedef UnaryOpNode< REAL_Sign, RealType, RealType > FuncSIGN ;
typedef UnaryOpNode< REAL_Fact, RealType, RealType > FuncFACT ;
typedef BinaryOpNode< REAL_Logn, RealType, RealType > FuncLOGN ;
typedef BinaryOpNode< REAL_Root, RealType, RealType > FuncROOT ;
typedef BinaryOpNode< REAL_Comb, RealType, RealType > FuncCOMB ;
typedef UnaryOpNode< REAL_Random, RealType, RealType > FuncRANDOM ;
typedef BinaryOpNode< REAL_Beta, RealType, RealType > FuncBETA ;
typedef UnaryOpNode< REAL_Chi, RealType, RealType > FuncCHI ;
typedef UnaryOpNode< REAL_RandInt, RealType, RealType > FuncRANDINT ;
typedef BinaryOpNode< REAL_F, RealType, RealType > FuncF ;
typedef UnaryOpNode< REAL_Exponential, RealType, RealType > FuncEXPONENTIAL ;
typedef BinaryOpNode< REAL_GAMMA, RealType, RealType > FuncGAMMA ;
typedef BinaryOpNode< REAL_NORMAL, RealType, RealType > FuncNORMAL ;
typedef BinaryOpNode< REAL_UNIFORM, RealType, RealType > FuncUNIFORM ;
typedef BinaryOpNode< REAL_Binomial, RealType, RealType > FuncBINOMIAL ;
typedef UnaryOpNode< REAL_Poisson, RealType, RealType > FuncPOISSON ;
typedef BinaryOpNode< REAL_MCM, RealType, RealType > FuncMCM ;
typedef BinaryOpNode< REAL_GCD, RealType, RealType > FuncGCD ;
typedef BinaryOpNode< REAL_HIP, RealType, RealType > FuncHIP ;
typedef BinaryOpNode< REAL_RECTTOPOLAR_R, RealType, RealType > FuncRECTTOPOLAR_R ;
typedef BinaryOpNode< REAL_RECTTOPOLAR_ANGLE, RealType, RealType > FuncRECTTOPOLAR_ANGLE ;
typedef BinaryOpNode< REAL_POLARTORECT_X, RealType, RealType > FuncPOLARTORECT_X ;
typedef BinaryOpNode< REAL_POLARTORECT_Y, RealType, RealType > FuncPOLARTORECT_Y ;
typedef UnaryOpNode< REAL_Cotan, RealType, RealType > FuncCOTAN ;
typedef UnaryOpNode< REAL_Cosec, RealType, RealType > FuncCOSEC ;
typedef UnaryOpNode< REAL_Cosech, RealType, RealType > FuncCOSECH ;
typedef UnaryOpNode< REAL_Asec, RealType, RealType > FuncASEC ;
typedef UnaryOpNode< REAL_Acotan, RealType, RealType > FuncACOTAN ;
typedef UnaryOpNode< REAL_Asech, RealType, RealType > FuncASECH ;
typedef UnaryOpNode< REAL_Acosech, RealType, RealType > FuncACOSECH ;
typedef UnaryOpNode< REAL_Acotanh, RealType, RealType > FuncACOTANH ;
typedef UnaryOpNode< REAL_CtoF, RealType, RealType > FuncCTOF ;
```

```
typedef UnaryOpNode< REAL_CtoK, RealType, RealType > FuncCTOK ;
typedef UnaryOpNode< REAL_FtoC, RealType, RealType > FuncFTOC ;
typedef UnaryOpNode< REAL_FtoK, RealType, RealType > FuncFTOK ;
typedef UnaryOpNode< REAL_KtoF, RealType, RealType > FuncKTOF ;
typedef UnaryOpNode< REAL_KtoC, RealType, RealType > FuncKTOC ;
typedef UnaryOpNode< REAL_NextPrime, RealType, RealType > FuncNEXTPRIME ;
typedef UnaryOpNode< REAL_RadToDeg, RealType, RealType > FuncRADTODEG ;
typedef UnaryOpNode< REAL_DegToRad, RealType, RealType > FuncDEGTORAD ;
typedef UnaryOpNode< REAL_Nth_Prime, RealType, RealType > FuncNTH_PRIME ;
typedef UnaryOpNode< REAL_RandomSign, RealType, RealType > FuncRANDOMSIGN ;
typedef ThreeOpNode< REAL_IF, RealType, RealType > IFNode ;
typedef FourOpNode< REAL_IFU, RealType, RealType > IFUNode ;
```

Apéndice B – Funciones para Manipulación de Valores Reales

A continuación se detalla una lista de las funciones adicionales que operan sobre números reales. Estas funciones son invocadas durante la evaluación de una regla, y tienen una asociación directa con alguna función del lenguaje que permite dar comportamiento a las celdas. Para obtener más detalle sobre el funcionamiento de estas funciones consultar el Manual del Usuario.

Función	**Descripción**	*Asociación con una función del lenguaje usado por N-CD++.*
Real abs(const Real &r)	Devuelve el valor absoluto de x.	*ABS*
Real acos(const Real &r)	Devuelve el arco coseno de x.	*ACOS*
Real acosech(const Real &r)	Calcula el arco cosecante hiperbólico de x.	*ACOSECH*
Real acosh(const Real &r)	Devuelve el coseno hiperbólico inverso de x.	*ACOSH*
Real acotan(const Real &r)	Devuelve el arco cotangente de x.	*ACOTAN*
Real acotanh(const Real &r)	Calcula el arco cotangente hiperbólico de x.	*ACOTANH*
Real asec(const Real &r)	Devuelve el arco secante de x.	*ASEC*
Real asech(const Real &r)	Calcula el arco secante hiperbólico de x.	*ASECH*
Real asin(const Real &r)	Devuelve el arco seno de x.	*ASIN*
Real asinh(const Real &r)	Devuelve el seno hiperbólico inverso de x.	*ASINH*
Real atan(const Real &r)	Devuelve el arco tangente de x.	*ATAN*
Real atanh(const Real &r)	Devuelve la tangente hiperbólica inversa de x.	*ATANH*
Real beta(const Real &a, const Real &b)	Devuelve un número pseudoaleatorio co distribución beta.	*BETA*
Real binomial(const Real &n, const Real &p)	Devuelve un número pseudoaleatorio con distribución binomial (n, p).	*BINOMIAL*
Real chi(const Real &r)	Devuelve un número pseudoaleatorio con distribución chi cuadrado con r grados de libertad.	*CHI*
Real comb(const Real &m, const Real &k)	Calcula el valor del combinatorio (m, k).	*COMB*
Real cos(const Real &r)	Devuelve el coseno de x.	*COS*
Real cosec(const Real &r)	Calcula la cosecante de x.	*COSEC*
Real cosech(const Real &r)	Calcula la cosecante hiperbólica de x.	*COSECH*
Real cosh(const Real &r)	Devuelve el coseno hiperbólico de x.	*COSH*
Real cotan(const Real &r)	Calcula la cotangente de x.	*COTAN*
Real CtoF(const Real &r)	Convierte de grados Centígrados a Fahrenheit.	*CTOF*
Real CtoK(const Real &r)	Convierte de grados Centígrados a Kelvin.	*CTOK*
Real degToRad(const Real &r)	Convierte de grados a radianes.	*DEGTORAD*
Real exp(const Real &r)	Devuelve el valor de e ^ r.	*EXP*
Real exponential(const Real &r)	Devuelve un número pseudoaleatorio con distribución exponencial, con media x.	*EXPONENTIAL*
Real f(const Real &a, const Real &b)	Devuelve un número pseudoaleatorio con distribución F, con a y b los grados de libertad.	*F*
Real fact(const Real &r)	Devuelve el factorial de x.	*FACT*

Función	**Descripción**	*Asociación con una función del lenguaje usado por N-CD++.*
Real fractional(const Real &r)	Devuelve la parte decimal de *x*.	*FRACTIONAL*
Real FtoC(const Real &r)	Convierte de grados Fahrenheit a Centígrados.	*FTOC*
Real FtoK(const Real &r)	Convierte de grados Fahrenheit a Kelvin.	*FTOK*
Real gamma(const Real &a, const Real &b)	Devuelve un número pseudoaleatorio con distribución Gamma.	*GAMMA*
Real gcd(const Real &r1, const Real &r2)	Calcula el Máximo Común Divisor entre r1 y r2.	*GCD*
Real hip(const Real &c1, const Real &c2)	Obtiene la hipotenusa del triángulo.	*HIP*
Real KtoC(const Real &r)	Convierte de grados Kelvin a Centígrados.	*KTOC*
Real KtoF(const Real &r)	Convierte de grados Kelvin a Fahrenheit.	*KTOF*
Real ln(const Real &r)	Devuelve el logaritmo natural de *x*.	*LN*
Real log(const Real &r)	Devuelve el valor del logaritmo (base 10) de *x*.	*LOG*
Real logn(const Real &v, const Real &n)	Calcula el logaritmo *n*-ésimo de *v*.	*LOGN*
Real max(const Real &r1, const Real &r2)	Calcula el máximo entre r1 y r2.	*MAX*
Real mcm(const Real &r1, const Real &r2)	Calcula el Mínimo Común Múltiplo entre r1 y r2.	*MCM*
Real min(const Real &r1, const Real &r2)	Calcula el mínimo entre r1 y r2.	*MIN*
Real nextPrime(const Real &r)	Devuelve el siguiente primo mayor a *x*.	*NEXTPRIME*
Real normal(const Real &a, const Real &b)	Devuelve un número pseudoaleatorio con distribución normal con media *a* y desvío *b*.	*NORMAL*
Real nth_Prime(const Real &r)	Devuelve el n-ésimo primo a partir del valor 2.	*NTH_PRIME*
Real poisson(const Real &m)	Devuelve un número pseudoaleatorio con distribución Poisson con media *m*.	*POISSON*
Real polarToRect_x(const Real &r, const Real &angle)	Convierte a coordenadas cartesianas.	*POLARTORECT_X*
Real polarToRect_y(const Real &r, const Real &angle)	Convierte a coordenadas cartesianas.	*POLARTORECT_Y*
Real power(const Real &r1, const Real &r2)	Calcula r1 ^ r2.	*POWER*
Real radToDeg(const Real &r)	Convierte de radianes a grados.	*RADTODEG*
Real randInt(const Real &r)	Devuelve un número pseudoaleatorio entero perteneciente a [0, *x*] con distribución uniforme.	*RANDINT*
Real random(const Real &r)	Devuelve un número pseudoaleatorio con distribución uniforme [0, 1]. Si *x* es indefinido devuelve indefinido, sino genera el valor.	*RANDOM*
Real randomSign(const Real &r)	Calcula un signo en forma pseudoaleatoria. Si *x* es indefinido devuelve indefinido, sino genera un signo.	*RANDOMSIGN*
Real realIf(const TvalBool &c, const Real &t, const Real &f)	Si *c* es verdadero devuelve *t*, sino devuelve *f*.	*IF*
Real realIfu(const TvalBool &c, const Real &t, const Real &f, const Real &u)	Si *c* es verdadero devuelve *t*, si es falso devuelve *f*, sino (si es indefinido) devuelve *u*.	*IFU*
Real rectToPolar_angle(const Real &x, const Real &y)	Convierte a coordenadas polares. Devuelve el ángulo.	*RECTTOPOLAR_ANGLE*
Real rectToPolar_r(const Real &x, const Real &y)	Convierte a coordenadas polares. Devuelve el radio.	*RECTTOPOLAR_R*
Real remainder(const Real &r1, const Real &r2)	Calcula el resto de r1/r2.	*REMAINDER*
Real root(const Real &v, const Real &n)	Calcula la raíz *n*-ésima de *v*.	*ROOT*
Real round(const Real &r)	Redondea el valor de *x* al entero más cercano.	*ROUND*
Real sec(const Real &r)	Devuelve la secante de *x*.	*SEC*
Real sech(const Real &r)	Devuelve la secante hiperbólica de *x*.	*SECH*
Real sign(const Real &r)	Devuelve el signo de *x*.	*SIGN*
Real sin(const Real &r)	Devuelve el seno de *x*.	*SIN*
Real sinh(const Real &r)	Devuelve el seno hiperbólico de *x*.	*SINH*

Función	**Descripción**	*Asociación con una función del lenguaje usado por N-CD++.*
Real sqrt(const Real &r)	Devuelve la raíz cuadrada de *x*	*SQRT*
Real tan(const Real &r)	Devuelve la tangente de *x*.	*TAN*
Real tanh(const Real &r)	Devuelve la tangente hiperbólica de *x*.	*TANH*
Real trunc(const Real &r)	Trunca el valor de x a un entero.	*TRUNC*
Real truncUpper(const Real &r)	Trunca el mayor entero el valor de *x*.	*TRUNCUPPER*
Real uniform(const Real &a, const Real &b)	Devuelve un número pseudoaleatorio con distribución uniforma [a, b].	*UNIFORM*
Real valueWithQuantum(Real &r, Real q)	Devuelve el valor de *r* según el quantum *q*.	-
TValBool even(const Real &r)	Dice si el número *x* es entero y par.	*EVEN*
TValBool isInt(const Real &r)	Dice si el número *x* es entero.	*ISINT*
TValBool isPrime(const Real &r)	Dice si el número *x* es entero y primo.	*ISPRIME*
TValBool isUndefinedReal(const Real &r)	Dice si el valor *x* es indefinido.	*ISUNDEFINED*
TValBool odd(const Real &r)	Dice si el número *x* es entero e impar.	*ODD*

N-CD++

Implementación de modelos Cell–DEVS n–dimensionales

Manual del Usuario

Contenido

1 Invocación del Simulador....8
1.1 Modo Standalone....8
1.2 Servidor de Simulación....10
2 Definición de Modelos....10
2.1 Modelos Acoplados....10
2.2 Modelos Atómicos....12
2.3 Modelos Celulares....12
3 Lenguaje de Especificación de Reglas....16
3.1 Gramática del Lenguaje....16
3.2 Orden de Precedencia y Asociatividad de los Operadores....18
3.3 Funciones y Constantes usadas por el lenguaje....18
3.3.1 Tratamiento de Valores Booleanos....18
3.3.1.1 Constantes Booleanas de la Lógica Trivalente....18
3.3.1.2 Operadores Booleanos....19
3.3.1.2.1 Operador AND....19
3.3.1.2.2 Operador OR....19
3.3.1.2.3 Operador NOT....19
3.3.1.2.4 Operador XOR....19
3.3.1.2.5 Operador IMP....19
3.3.1.2.6 Operador EQV....20
3.3.2 Funciones y Operaciones para el Tratamiento de Números Reales....20
3.3.2.1 Operadores Relacionales....20
3.3.2.1.1 Operador =....20
3.3.2.1.2 Operador !=....20
3.3.2.1.3 Operador >....20
3.3.2.1.4 Operador <....21
3.3.2.1.5 Operador <=....21
3.3.2.1.6 Operador >=....21
3.3.2.2 Operadores Aritméticos....21
3.3.2.3 Funciones sobre Números Reales....22
3.3.2.3.1 Funciones para la Verificación de Propiedades sobre Números Reales....22
Función Even....22
Función Odd....22
Función isInt....22
Función isPrime....22
Función isUndefined....22
3.3.2.3.2 Funciones Matemáticas....22
3.3.2.3.2.1 Funciones Trigonométricas....23
Función tan....23
Función sin....23
Función cos....23
Función sec....23
Función cotan....23
Función cosec....23
Función atan....23
Función asin....23
Función acos....23
Función asec....24
Función acotan....24
Función sinh....24
Función cosh....24
Función tanh....24
Función sech....24

Función cosech....24
Función atanh....24
Función asinh....24
Función acosh....25
Función asech....25
Función acosech....25
Función acotanh....25
Función hip....25
3.3.2.3.2.2 Funciones para el Cálculo de Raíces, Potencias y Logaritmos....25
Función sqrt....25
Función exp....25
Función ln....25
Función log....26
Función logn....26
Función power....26
Función root....26
3.3.2.3.2.3 Funciones para el cálculo del MCM, MCD y Resto de la División Numérica....26
Función LCM....26
Función GCD....27
Función remainder....27
3.3.2.3.3 Funciones para la Conversión de Valores Reales a Enteros....27
Función round....27
Función trunc....27
Función truncUpper....27
Función fractional....28
3.3.2.3.4 Funciones para el Tratamiento del Signo de Valores Numéricos....28
Función abs....28
Función sign....28
Función randomSign....28
3.3.2.3.5 Funciones para el Manejo de Números Primos....28
Función isPrime....28
Función nextPrime....28
Función nth_Prime....28
3.3.2.3.6 Funciones para la obtención de Mínimos y Máximos....29
Función min....29
Función max....29
3.3.2.3.7 Funciones Condicionales....29
Función if....29
Función ifu....29
3.3.2.3.8 Funciones Probabilísticas....29
Función randomSign....29
Función random....29
Función chi....30
Función beta....30
Función exponential....30
Función f....30
Función gamma....30
Función normal....30
Función uniform....30
Función binomial....30
Función poisson....30
Función randInt....31
3.3.2.3.9 Funciones para cálculo de Factoriales y Combinatorios....31
Función fact....31
Función comb....31
3.3.2.4 Funciones que actúan sobre una Celda y su Vecindario....31

Función stateCount....31
Función trueCount....31
Función falseCount....31
Función undefCount....32
Función cellPos....32
3.3.2.5 Funciones para la Obtención del Tiempo de Simulación....32
Función Time....32
3.3.2.6 Funciones para la Conversión de Valores entre distintas Unidades....32
3.3.2.6.1 Funciones para la Conversión de Grados a Radianes....32
Función radToDeg....32
Función degToRad....32
3.3.2.6.2 Funciones para la Conversión de Coordenadas Rectangulares a Polares....32
Función rectToPolar_r....32
Función rectToPolar_angle....32
Función polarToRect_x....33
Función polarToRect_y....33
3.3.2.6.3 Funciones para la Conversión de Temperaturas en distintas unidades....33
Función CtoF....33
Función CtoK....33
Función KtoC....33
Función KtoF....33
Función FtoC....33
Función FtoK....33
3.3.2.7 Funciones para la manipulación de Valores de los Puertos de Entrada y Salida....33
Función portValue....33
Función send....34
3.3.3 Constantes Predefinidas por el Lenguaje....35
Constante Pi....35
Constante e....35
Constante INF....35
Constante electron_mass....35
Constante proton_mass....35
Constante neutron_mass....36
Constante Catalan....36
Constante Rydberg....36
Constante grav....36
Constante bohr_radius....36
Constante bohr_magneton....36
Constante Boltzmann....36
Constante accel....36
Constante light....36
Constante electron_charge....36
Constante Planck....36
Constante Avogadro....36
Constante amu....36
Constante pem....36
Constante ideal_gas....36
Constante Faraday....36
Constante Stefan_boltzmann....36
Constante golden....37
Constante euler_gamma....37
3.4 Mecanismos para Evitar la Reescritura de Reglas....37
3.4.1 Cláusula Else....37
3.4.2 Preprocesador – Uso de Macros....38
4 Archivo para la definición de los Valores Iniciales del Modelo....39
5 Archivo de Mapa de Valores Iniciales del Modelo....40

6 Formato del Archivo para la definición de Eventos Externos....41
7 Formato de la Salida de Eventos....41
8 Formato del Archivo de Log....41
9 Salida generada al activarse el modo de Debug del Parser....42
10 Salida del modo de Debug para la Evaluación de las Reglas....43
11 Representación de los Resultados – DrawLog....44
11.1 Representación del DrawLog para modelos Bidimensionales....46
11.2 Representación del DrawLog para modelos Tridimensionales....47
11.3 Representación del DrawLog para modelos de 4 ó más dimensiones....47
12 Generación Aleatoria de Valores Iniciales – MakeRand....48
13 Conversión de Archivos de Valores a Mapa de Valores – ToMap....49
14 Conversión de Archivos de Valores para uso en CD++ – ToCDPP....50
15 Incorporación de Nuevos Modelos Atómicos....51
15.1 Ejemplo. Construcción de una Cola....51
16 Apéndice A – Ejemplos de la Definición de Modelos....53
16.1 Juego de la Vida....53
16.2 Simulación del Rebote de un Objeto....54
16.3 Clasificación de Materias Primas....55
16.4 Juego de la Vida en 3D....57
16.5 Uso de Macros....58
17 Apéndice B – El Preprocesador y los Archivos Temporarios....59

Indice de Figuras

Figura 1 – Ayuda del Simulador para la línea de comandos....8
Figura 2 – Ejemplo de la creación de un modelo DEVS acoplado....11
Figura 3 – Formato genérico para el establecimiento de parámetros a un modelo atómico DEVS....12
Figura 4 – Ejemplo del establecimiento de parámetros para modelos atómicos DEVS....12
Figura 5 – Gramática usada para la definición de reglas bajo N-CD++....17
Figura 6 – Orden de Precedencia y asociatividad usadas por N-CD++....18
Figura 7 – Comportamiento del operador booleano AND....19
Figura 8 – Comportamiento del operador booleano OR....19
Figura 9 – Comportamiento del operador booleano NOT....19
Figura 10 – Comportamiento del operador booleano XOR....19
Figura 11 – Comportamiento del operador booleano IMP....19
Figura 12 – Comportamiento del operador booleano EQV....20
Figura 13 – Comportamiento del operador relacional =....20
Figura 14 – Comportamiento del operador relacional !=....20
Figura 15 – Comportamiento del operador relacional >....21
Figura 16 – Comportamiento del operador relacional <....21
Figura 17 – Comportamiento del operador relacional <=....21
Figura 18 – Comportamiento del operador relacional >=....21
Figura 19 – Operadores Aritméticos....21
Figura 20 – Ejemplo de uso de la función portValue....34
Figura 21 – Ejemplo de uso de la función portValue junto a thisPort....34
Figura 22 – Ejemplo del uso de la cláusula Else....37
Figura 23 – Ejemplo de una referencia circular debido al mal uso de la cláusula Else....38
Figura 24 – Ejemplo de una referencia circular directa detectada por el simulador....38
Figura 25 – Formato de la definición de una macro....38
Figura 26 – Ejemplo del uso de Comentarios....39
Figura 27 – Formato del Archivo para la Definición de los valores iniciales del modelo celular....39
Figura 28 – Ejemplo de un archivo para la definición de valores iniciales para un Modelo Celular....40
Figura 29 – Formato del Archivo de Mapa de Valores de un modelo celular....40
Figura 30 – Ejemplo de un archivo para la definición de Eventos Externos....41
Figura 31 – Ejemplo de un archivo de Salida....41
Figura 32 – Fragmento de un archivo de log....42
Figura 33 – Salida generada por el modo de debug del parser para el Juego de la Vida....43
Figura 34 – Fragmento de la salida generada por el modo de debug para la Evaluación de Reglas....44
Figura 35 – Ayuda del DrawLog para la línea de comandos....45
Figura 36 – Ejemplo de llamadas al DrawLog....46
Figura 37 – Fragmento de la salida generada por el DrawLog para un modelo bidimensional....46
Figura 38 – Fragmento de la salida generada por el DrawLog para un modelo tridimensional....47
Figura 39 – Fragmento de la salida generada por el DrawLog para un modelo de dimensión 4....48
Figura 40 – Ayuda del MakeRand para la línea de comandos....48
Figura 41 – Ayuda del ToMap para la línea de comandos....49
Figura 42 – Ayuda del toCDPP para la línea de comandos....50
Figura 43 – Esquema de una Cola....52
Figura 44 – Método para la inicialización de la Cola....52
Figura 45 – Método para la definición de la función de Transición Externa de la Cola....53
Figura 46 – Métodos para las funciones de Salida y de Transición Interna de la Cola....53
Figura 47 – Implementación del Juego de la Vida....54
Figura 48 – Implementación del Modelo de Rebote de un Objeto....55
Figura 49 – Esquema de Acoplamiento de la Clasificadora de Materias Primas....56
Figura 50 – Implementación del Sistema de Clasificación de Materias Primas....57
Figura 51 – Implementación del Juego de la Vida en 3D....58
Figura 52 – Archivo life.val conteniendo los valores iniciales para el Juego de la Vida en 3D....58
Figura 53 – Implementación del Juego de la Vida en 4D con el uso de Macros....59

Figura 54 – Archivo life.val conteniendo los valores iniciales para el Juego de la Vida en 4D..59
Figura 55 – Archivo life.inc conteniendo algunas macros usadas en el Juego de la Vida – 4D..59
Figura 56 – Archivo life–1.inc conteniendo el resto de las macros usadas en el Juego de la Vida – 4D............................59

N-CD++

Manual del Usuario

1 Invocación del Simulador

Existen dos formas de invocar al simulador:

- *Modo Standalone.*
- *Servidor de Simulación* (vía conexión por red).

1.1 Modo Standalone

Para configurar la ejecución del simulador, son posibles los siguientes parámetros:

–h: muestra la siguiente ayuda

```
simu [-ehlmotdpvbfrsqw]
        e: events file (default: none)
        h: show this help
        l: message log file (default: /dev/null)
        m: model file (default : model.ma)
        o: output (default: /dev/null)
        t: stop time (default: Infinity)
        d: set tolerance used to compare real numbers
        p: print extra info when the parsing occurs (only for cells models)
        v: evaluate debug mode (only for cells models)
        b: bypass the preprocessor (macros are ignored)
        f: flat debug mode (only for flat cells models)
        r: debug cell rules mode (only for cells models)
        s: show the virtual time when the simulation ends (on stderr)
        q: use quantum to calculate cells values
        w: sets the width and precision (with form xx-yy) to show numbers
```

Figura 1 – Ayuda del Simulador para la línea de comandos

–**e**: Nombre de archivo del cual se cargarán los eventos externos. Si se omite este parámetro el simulador no utilizará ningún evento externo.
La descripción del formato utilizado para el ingreso de los eventos externos se encuentra en la sección 5.

–**l**: Nombre de archivo en el cual se almacenarán los mensajes que reciben y emiten los modelos a lo largo de la simulación. Si se omite este parámetro el simulador no generará el log de actividad. Si se desea obtener el log por el standard output no debe especificarse ningún parámetro (–l).
La descripción del formato del log se encuentra en la sección 8.

–**m**: Nombre de archivo del cual se cargará el modelo a simular. Si se omite este parámetro el simulador cargará los modelos del archivo *model.ma*.

–**o**: nombre de archivo en el cual se grabará la salida generada por el simulador. Si se omite este parámetro el simulador no generará ninguna salida. Si se desea obtener los resultados por la salida estándar no debe especificarse ningún parámetro (–o).
La descripción del formato de la salida generada se encuentra en la sección 7.

–t: Hora de finalización de la simulación. Si se omite este parámetro el simulador solo se detendrá ante la falta de eventos (internos o externos). El formato de la hora debe ser HH:MM:SS:MS, donde:

HH: cantidad de horas

MM: minutos (de 0 a 59)

SS: segundos (de 0 a 59)

MS: milésimas de segundo (de 0 a 999)

–d: Define el valor de la tolerancia utilizada para realizar las comparaciones de números reales. El parámetro pasado será un número el cual será establecido como la nueva tolerancia.
Si no se establece este parámetro en la invocación del simulador el valor usado por defecto para la tolerancia es 10^{-8}.

–p: Muestra información adicional durante la etapa del parseo de las reglas utilizadas para la definición del comportamiento de los modelos celulares. El parámetro usado debe ser un nombre de archivo donde se almacenará el respectivo detalle. Este seteo resulta de utilidad para hallar errores sintácticos en la escritura de reglas muy complejas.
El formato de la salida generada cuando este modo esta activado es mostrado en la sección 9.

–v: Establece el modo de debug para la evaluación de las reglas de un modelo celular. Si este modo esta activo, entonces toda regla al ser evaluada mostrará paso a paso los resultados de las evaluaciones de las funciones y operadores que la componen. Este modo, además, evalúa las reglas en forma completa, es decir, no utiliza la optimización de operadores. El parámetro requerido debe ser el nombre de un archivo donde se dejaran las evaluaciones respectivas.
El formato de la salida generada cuando este modo esta activado es mostrado en la sección 10.

–b: Evita el uso del preprocesador de macros.

–f: Activa el modo de debug para los modelos celulares achatados. Este modo posibilita ver el estado del modelo en cada instante de tiempo para los modelos achatados, debido a que cuando se utiliza este mecanismo de simulación se evita el envío de gran cantidad de mensajes dentro del modelo celular acoplado y por lo tanto no es posible registrar en el archivo de log el envío de mensajes entre celdas, con lo que el *DrawLog* será incapaz de mostrar dichos resultados.
El parámetro se corresponde al nombre del archivo donde se almacenarán los resultados. Si no se pasa ningún nombre de archivo, los datos serán mostrados a través de la salida estándar (–f).

–r: Activa el modo de debug para la validación de las reglas que definen el comportamiento de los modelos celulares. Cuando este modo esta activo el simulador verifica en tiempo de ejecución si existe una única regla que pueda ser satisfecha. Si dicha condición no se cumple, la simulación será abortada. Esta posibilidad de chequeo de integridad y consistencia del lenguaje está disponible solamente en modo standalone.
Existen tratamientos especiales ante determinados casos: si el modelo es estocástico (es decir, si utiliza funciones de generación de números aleatorios) es posible que varias reglas sean válidas, o que ninguna de ellas lo sea. Para ambos casos se informa al usuario mediante la generación de un mensaje de alerta por la salida estándar y la simulación no es abortada. Para el primer caso se toma la primer regla válida. Para el segundo caso, se asume que el valor de la celda es indefinido (**?**) y se utiliza el tiempo de demora establecido por defecto en la definición del modelo.
Si no se establece este parámetro en la invocación del simulador, el modo queda desactivado y solo se considera la primer regla que sea satisfecha.

–s: Muestra la hora de finalización de la simulación por stdErr.

–q: Indica que se usará un valor de *quantum*, con el objetivo de cuantificar el resultado obtenido por la función de cómputo local que se ejecuta en cada celda del modelo. De esta forma, todos los valores antes de ser

asignados como nuevo estado de una celda, son redondeados hacia abajo a la escala del quantum, por lo que todos los valores usados serán múltiplos del quantum. De esta forma se reduce el número de mensajes transmitidos en la simulación a costa de tener un error en los valores de estado de las celdas.
Por ejemplo: si el quantum es 0.01 y el valor devuelto por la función de computo local es 0.2371, la celda obtendrá como nuevo estado el valor 0.23.
El valor que se usará como quantum debe ser indicado a continuación del parámetro –q, por ejemplo: para indicar el valor de quantum 0.01 deberá usarse –q0.001.
Si el valor del quantum es 0 ó no se indica el parámetro **–q** en la invocación al simulador, se deshabilita el uso de quantum, con lo que los valores devueltos por la función de cómputo local serán directamente los nuevos valores de estado de las celdas.

–w: Permite establecer un ancho para la representación de valores numéricos y su precisión, los cuales serán usados para mostrar valores en las salidas generadas por el simulador (archivo de log, de eventos externos, de resultados de evaluación de reglas, etc.).
Por defecto se asume un ancho de 12 caracteres y una precisión de 5 dígitos. Para este caso, de los 12 caracteres de ancho, 5 serán para la precisión, 1 para el punto decimal, y el resto serán usados para la parte entera, que incluirá un carácter para el signo en caso de que el valor sea negativo.
Para indicar nuevos valores para el ancho y la precisión, se deberá escribir el parámetro **–w** seguido de la cantidad de caracteres para el ancho, un guión, y la cantidad de caracteres usados para la precisión. Por ejemplo, para usar 10 caracteres de ancho y 3 de precisión se deberá escribir: **–w10–3**.
Cualquier valor numérico que deba ser mostrado por el simulador se ajustará a los valores del ancho y la precisión, siendo redondeado en caso de ser necesario. Así, por ejemplo, si una celda tiene como estado el valor 7.0007 y se establece –w10–3, el valor mostrado para la celda en cualquier salida generada será 7.001, sin embargo, el valor almacenado internamente por la celda que es usado en la simulación no se ve afectado por los valores de ancho y precisión.

1.2 Servidor de Simulación

La invocación del simulador sin especificar parámetros indica al simulador que debe cargarse en modo servidor de simulación. En esta implementación, la comunicación con el servidor se realizará utilizando los servicios TCP/IP y solo esta disponible en las versiones bajo Unix. El simulador esperará en un puerto por la especificación de una simulación, la correrá y retornará los resultados por la misma vía.

La especificación se compone de tres partes separadas por una línea con un punto en la primer posición. El orden de la especificación es el siguiente:

Descripción del modelo.

Lista de eventos externos.

Hora de Finalización.

2 Definición de Modelos

El archivo que permite definir los modelos esta compuesto por grupos de definiciones de modelos acoplados y configuración de modelos atómicos, esta última opcional. Cada definición indica el nombre del modelo (entre [ ]) y sus atributos. El grupo del modelo **[top]** es obligatorio y define el modelo acoplado de mayor nivel.

En la sección 16 se encuentran diversos ejemplos que detallan la definición de modelos, los cuales serán analizados en las próximas secciones.

2.1 Modelos Acoplados

Existen cuatro posibles propiedades a configurar: componentes (*components*), puertos de salida (*out*), puertos de entrada (*in*) y conexiones entre modelos (*link*). El nombre con el cual se define cada una de las partes del modelo es reservado y cualquier otra palabra será ignorada. La sintaxis es la siguiente:

Components: Describe los modelos que integraran el modelo acoplado. El formato es:

nombre_de_modelo@nombre_de_clase

El orden en que se especifican los modelos determina la prioridad utilizada para el envío de mensajes. Esto representa la función **select** del formalismo.

El nombre de modelo es necesario ya que es posible construir un modelo acoplado con más de una instancia del mismo modelo atómico. Por ejemplo un acoplado que posee dos colas llamadas *cola1* y *cola2*.

El nombre de clase puede hacer referencia tanto a modelos atómicos como a modelos acoplados. Estos últimos deben estar descriptos en el mismo archivo de configuración como un nuevo grupo.

En caso de no especificarse este atributo se producirá un error indicando la falta del mismo.

Out: Enumera los nombres de los puertos de salida. Este atributo es opcional ya que un modelo puede no tener puertos de este tipo.
Ejemplo: *Out* puerto1 puerto2 puerto3

In: Enumera los nombres de los puertos de entrada. Este atributo es opcional ya que un modelo puede no tener puertos de este tipo.
Ejemplo: *In* puerto1 puerto2 puerto3

Link: Describe el esquema de acoplamiento interno, externo de entrada y externo de salida. El formato esta dado por el par:

puerto_origen[@*modelo*] *puerto_destino*[@*modelo*]

El nombre del modelo es opcional ya que si no se indica se toma como un puerto correspondiente al acoplado en cuestión.

Ejemplo:

```
[top]
components : transducer@Transducer generator@Generator Consumidor
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumidor
Link : out@Consumidor solved@transducer
Link : out@transducer out

[Consumidor]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

Figura 2 – Ejemplo de la creación de un modelo DEVS acoplado

2.2 Modelos Atómicos

En esta definición se configuran los modelos atómicos participantes de la simulación. En caso de no figurar ninguna referencia se tomarán los valores por defecto que halla programado el desarrollador de dicha clase (para más detalle ver la sección 12).
La configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
.
.
.
nombre_variablen : valor_varn
```

Figura 3 – Formato genérico para el establecimiento de parámetros a un modelo atómico DEVS

Los nombre de las variables dependen del desarrollador de la clase que se desea configurar y deben estar documentados junto con el código fuente de la misma.
Cada instancia de un tipo de modelos atómico podrá ser configurada independientemente de otras instancias del mismo tipo.

En el siguiente ejemplo se ve dos instancias de la clase *Processor* (derivada de *Atomic*) con distinta configuración.

```
[top]
components : Queue@queue Processor1@processor Processor2@processor
.
.
.

[processor]
distribution : exponential
mean : 10

[processor2]
distribution : poisson
mean : 50

[queue]
preparation : 0:0:0:0
```

Figura 4 – Ejemplo del establecimiento de parámetros para modelos atómicos DEVS

2.3 Modelos Celulares

Los modelos celulares son una variante de los modelos acoplados, y debido a esto se utiliza la misma especificación con el agregado de ciertos parámetros inherentes a las características de los mismos. Dichos parámetros son:

Type : [**CELL** | **FLAT**]

Indica si el modelo celular será achatado o no. Si no se especifica se asume que es un modelo no achatado (*CELL*).

Width : entero

Permite definir la cantidad de columnas sólo para modelos celulares unidimensionales y bidimensionales.
El uso de *Width* implica necesariamente el uso de la cláusula *Height* para completar la definición de la dimensión del modelo.
Si se utiliza *Width*, la invocación de la cláusula *Dim* en la definición del mismo modelo producirá un error.

Height : entero

Permite definir la cantidad de filas sólo para un modelo celular bidimensional.
Si se desea modelar un autómata celular unidimensional, debe establecerse el valor 1 para la cláusula *Height*.
El uso de *Height* implica necesariamente el uso de la cláusula *Width* para completar la definición de la dimensión del modelo.
Si se utiliza *Height*, la invocación de la cláusula *Dim* en la definición del mismo modelo producirá un error.

Dim : $(x_0, x_1, ..., x_n)$

Permite definir la dimensión de cualquier modelo celular.
Todos los valores x_i deben ser enteros.
Si se utiliza *Dim*, la invocación de las cláusulas *Width* ó *Height* en la definición del mismo modelo producirá un error.
La tupla que define la dimensión del modelo celular debe contener al menos dos elementos. Esto implica que si se quiere modelar un autómata celular unidimensional, deberá ser tratarse como si un modelo bidimensional de tamaño $(x_0, 1)$.

Cabe destacar que todas las referencias a celdas deberán ser de la forma:

$$(y_0, y_1, ..., y_n) \quad \text{donde: } 0 \leq y_i < x_i \quad \forall\, i = 0, .., n$$

con y_i un valor entero

Select : *nombreCelular*($x_{1,1}, x_{2,1},...,x_{n,1}$)... *nombreCelular*($x_{1,m}, y_{2,m},...,k_{n.m}$)

Con: $0 \leq x_{1,i} < dim_1 \;\vee\; 0 \leq x_{1,i} < Width \quad \forall\, i = 1, ..., m$
$0 \leq x_{2,i} < dim_2 \;\vee\; 0 \leq x_{2,i} < Height \quad \forall\, i = 1, ..., m$
$0 \leq x_{k,i} < dim_k \quad \forall\, i = 1, ..., m \,;\, \forall\, k = 3, ..., n$

Representa la función ***select*** del formalismo indicando las celdas que poseen prioridad sobre el resto. Las celdas no especificadas poseen la prioridad dictada por el orden de pares según su posición.

In : Igual que en los modelos acoplados. Esta cláusula puede no estar definida, con lo que no existirán puertos de entrada para el modelo celular.

Out : Igual que en los modelos acoplados. Esta cláusula puede no estar definida, con lo que no existirán puertos de salida para el modelo celular.

Link : Igual que en los modelos acoplados pero para hacer referencia a una celda se debe usar el nombre del acoplado más $(x_1,x_2,...,x_n)$ sin dejar espacios.

Ejemplos: *Link* puertoSalida puertoEntrada@nombreCelular($x_1,x_2,...,x_n$)
Link puertoSalida@nombreCelular($x_1,x_2,...,x_n$) puertoEntrada

Border : [**WRAPPED** | **NOWRAPPED**]

Indica si el modelo es o no toroidal. Por defecto toma el valor NOWRAPPED.
Si se utiliza un borde no toroidal, cualquier referencia a una celda fuera del espacio celular retornará el valor indefinido (**?**).

Delay : [**TRASPORT** | **INERTIAL**]

Especifica el tipo de demora usada en cada celda. Por defecto toma el valor TRANSPORT.

DefaultDelayTime : entero

Demora por defecto para los eventos externos (medida en milisegundos).

Neighbors : *nombreCelular*($x_{1,1}, x_{2,1},...,x_{n,1}$)... *nombreCelular*($x_{1,m}, x_{2,m},...,x_{n.m}$)

Define el vecindario para todas las celdas del modelo. Cada celda ($x_{1,i}, x_{2,i},...,x_{n.i}$) representa un desplazamiento con respecto a la celda de origen del vecindario.
N-CD++ no impone restricciones sobre la creación del vecindario, permitiendo que las celdas que lo componen puedan no estar adyacentes a la celda de origen.
Es posible utilizar más de una sentencia **neighbors** para declarar el vecindario para un modelo celular.

Initialvalue : [*Real* | **?**]

Representa el valor inicial para todo el espacio de celdas. **?** representa al valor indefinido.

InitialRowValue : $fila_i$ $valor_1...valor_{width}$

Con $0 \leq fila_i <$ *Height* (donde *Height* es el segundo elemento de la dimensión definida con **Dim**, o el valor definido con **Height**).
Permite definir una lista de valores los cuales serán establecidos como iniciales para una fila para un modelo celular bidimensional. El valor definido en la posición j será usado para establecer el estado inicial de la celda (i, j) del modelo celular.
Los valores se indican uno al lado del otro sin ningún carácter separador y deben pertenecer al conjunto {**?**, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
Si se utiliza esta cláusula para la descripción de un modelo con más de 2 dimensiones, se producirá un error.
Para ver un ejemplo de su uso vea la sección 16.1.

InitialRow : $fila_i$ $valor_1 ... valor_{width}$

Con $0 \leq fila_i <$ *Height* (donde *Height* es el segundo elemento de la dimensión definida con **Dim**, o el valor definido con **Height**).
Permite definir una lista de valores los cuales serán establecidos como iniciales para una fila del modelo celular bidimensional. El valor definido en la posición j será usado para establecer el estado inicial de la celda (i, j) del modelo celular.

Los valores se indican uno al lado del otro separados por un espacio en blanco. A diferencia de **InitialRowValue**, mediante esta cláusula es posible usar cualquier valor perteneciente al conjunto $\Re \cup \{\mathbf{?}\}$.
Si se utiliza esta cláusula para la descripción de un modelo con más de 2 dimensiones, se producirá un error.

InitialCellsValue : *fileName*

Especifica el nombre del archivo a utilizar para definir los valores iniciales de las celdas de un modelo celular. El formato de dicho archivo se define en la sección 4.
InitialCellsValue puede ser usada con cualquier tipo de modelos celulares, incluso con modelos bidimensionales. En cambio *InitialRowValue* e *InitialRow* no podrán ser usados cuando la dimensión del modelo sea mayor a 2. Si la dimensión es 2, puede usarse cualquiera de ellas, e incluso una combinación de las mismas, pero en este caso los valores leídos del archivo especificado en *InitialCellsValue* reemplazarán a los valores de las mismas celdas definidas por *InitialRowValue* o *InitialRow*.

InitialMapValue : *fileName*

Especifica el nombre del archivo a utilizar, y que contiene un mapa de valores que serán usados como estado inicial para un modelo celular. El formato de dicho archivo se define en la sección 5.

LocalTransition : *transitionFunctionName*

Indica el nombre del grupo que contiene las reglas a utilizar para la función de transición local para todas las celdas.

PortInTransition : *portName@ nombreCelular*($x_1, x_2,...,x_n$) *transitionFunctionName*

Permite definir un comportamiento alternativo cuando arriba un mensaje externo por el puerto de entrada indicado de la celda ($x_1, x_2,...,x_n$) del modelo celular.
Si no se especifica una función asociada al puerto de la celda, al arribar un mensaje externo por el mismo, el valor de dicho mensaje será asignado a la celda usando la demora especificada por defecto en la definición del modelo.
En la sección 16.3 se ejemplifica el uso de dicha cláusula.

Zone : *transitionFunctionName* { rango_1[..rango_n] }

Permite definir un comportamiento alternativo para el conjunto de celdas comprendidas dentro del rango especificado. Cada rango es algo de la forma ($x_1,x_2,...,x_n$) describiendo una celda, ($x_1,x_2,...,x_n$)..(y_1,y_2,y_n) describiendo una zona de celdas, o una lista que puede combinar una cantidad arbitraria de ambas (separando a cada elemento de la misma por un espacio en blanco).
Por ejemplo: *zone :* bache { (10,10)..(13,13) (1,3) }

En el momento de calcular el nuevo estado para una celda, si dicha celda pertenece a algún rango se usará la función definida para tal, sino se utilizará la función de transición definida en la cláusula **LocalTransition**.
Para ver un ejemplo de su uso vea la sección 16.2.

3 Lenguaje de Especificación de Reglas

La definición de las reglas que describen un cierto comportamiento se hace en forma independiente a los modelos celulares que la utilizan. Esto permite que más de un modelo celular utilice la misma especificación como así también que varias zonas dentro de un espacio celular lo utilicen sin necesidad de redefinirla.

El lenguaje se define como un nuevo grupo dentro de la especificación, donde cada componente del grupo es una regla con la siguiente sintaxis:

rule : resultado demora { condición }

Cada regla esta compuesta por tres elementos: una *condición*, una *demora* y un *resultado*. Para calcular el nuevo estado de una celda, se toma cada una de las reglas (en el orden en que fueron definidas) y si la condición de la misma es evaluada a verdadero, entonces se evalúan su resultado y su demora, y estos valores serán los utilizados por la celda. Si la evaluación de la condición de la regla es falsa, entonces se toma la siguiente regla. Si se evalúan todas las reglas sin haber encontrado alguna válida, entonces la simulación se cancelará y se informará al usuario de tal situación. Si existe más de una regla válida se toma la primera de ellas.

Cabe considerar que si al evaluar la demora se obtiene como resultado el valor indefinido, entonces la simulación será automáticamente cancelada.

3.1 Gramática del Lenguaje

La sintaxis del lenguaje usado por *N-CD++* para la especificación del comportamiento de los modelos celulares atómicos puede definirse con la BNF mostrada en la Figura 5, donde las palabras escritas con letras minúsculas y en negrita representan terminales, mientras que las escritas en mayúsculas representan no terminales.

```
RULELIST        = RULE
                  |  RULE RULELIST

RULE            = RESULT RESULT { BOOLEXP }

RESULT          = CONSTANT
                  |  { REALEXP }

BOOLEXP         = BOOL
                  |  ( BOOLEXP )
                  |  REALRELEXP
                  |  not BOOLEXP
                  |  BOOLEXP OP_BOOL BOOLEXP

OP_BOOL         = and  |  or  |  xor  |  imp  |  eqv

REALRELEXP      = REALEXP OP_REL REALEXP
                  |  COND_REAL_FUNC(REALEXP)

REALEXP         = IDREF
                  |  ( REALEXP )
                  |  REALEXP OPER REALEXP

IDREF           = CELLREF
                  |  CONSTANT
                  |  FUNCTION
                  |  portValue(PORTNAME)
                  |  send(PORTNAME, REALEXP)
                  |  cellPos(REALEXP)

CONSTANT        = INT
                  |  REAL
```

```
                  |  CONSTFUNC
                  |  ?

FUNCTION        = UNARY_FUNC(REALEXP)
                  |  WITHOUT_PARAM_FUNC
                  |  BINARY_FUNC(REALEXP, REALEXP)
                  |  if(BOOLEXP, REALEXP, REALEXP)
                  |  ifu(BOOLEXP, REALEXP, REALEXP, REALEXP)

CELLREF         = (INT, INT RESTO_TUPLA

RESTO_TUPLA     = , INT RESTO_TUPLA
                  |  )

BOOL            = t  |  f  |  ?

OP_REL          = !=  |  =  |  >  |  <  |  >=  |  <=

OPER            = +  |  -  |  *  |  /

INT             = [SIGN] DIGIT {DIGIT}

REAL            = INT  |  [SIGN] {DIGIT}.DIGIT {DIGIT}

SIGN            = +  |  -

DIGIT           = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

PORTNAME        = thisPort  |  STRING

STRING          = LETTER {LETTER}

LETTER          = a | b | c |...| z | A | B | C |...| Z

CONSTFUNC       = pi | e | inf | grav | accel | light | planck | avogadro |
                  faraday | rydberg | euler_gamma | bohr_radius | boltzmann |
                  bohr_magneton | golden | catalan | amu | electron_charge |
                  ideal_gas | stefan_boltzmann | proton_mass | electron_mass |
                  neutron_mass  | pem

WITHOUT_PARAM_FUNC = truecount | falsecount | undefcount | time | random |
                     randomSign

UNARY_FUNC      = abs | acos | acosh | asin | asinh | atan | atanh | cos |
                  sec | sech | exp | cosh | fact | fractional | ln | log |
                  round | cotan | cosec | cosech | sign | sin | sinh |
                  statecount | sqrt | tan | tanh | trunc | truncUpper |
                  poisson | exponential | randInt | chi | asec | acotan |
                  asech | acosech | nextPrime | radToDeg | degToRad |
                  nth_prime | acotanh | CtoF | CtoK | KtoC | KtoF | FtoC |
                  FtoK

BINARY_FUNC     = comb | logn | max | min | power | remainder | root | beta |
                  gamma | lcm | gcd | normal | f | uniform | binomial |
                  rectToPolar_r | rectToPolar_angle | polarToRect_x | hip |
                  polarToRect_y

COND_REAL_FUNC = even | odd | isInt | isPrime | isUndefined
```

Figura 5 – Gramática usada para la definición de reglas bajo *N-CD++*

Cabe considerar que en la definición de una regla, el segundo valor, que se corresponde con la demora de la celda, puede ser un número real, ya sea en forma directa o como resultado de la evaluación de una expresión. Sin

embargo, si no es un número entero, este será automáticamente truncado de tal forma que su valor sí lo sea. Por otra parte, si su valor es indefinido (**?**) entonces se informará tal situación y se producirá un error, abortando la simulación.

3.2 Orden de Precedencia y Asociatividad de los Operadores

La precedencia indica que operación se resolverá primero. Por ejemplo si tenemos:

$$C + B * A$$

donde * y + son las operaciones habituales sobre números reales; y *A*, *B* y *C* son números reales. En este caso, como * tiene mayor precedencia que + entonces primero se resolverá *B* * *A*; por lo tanto, será equivalente a resolver *C* + (*B* * *A*).

La asociatividad indica que función se resolverá ante dos operaciones de igual precedencia. Por ejemplo: la asociatividad a izquierda de los operadores *AND* y *OR* indica que si se interpreta la línea

C **and** *B* **or** *D*

como **AND** y **OR** tienen igual precedencia, se recurre a la regla de asociatividad para elegir a alguno. Como estos son asociativos a izquierda se elige resolver primero el **AND**.

Las operaciones que no tienen asociatividad es porque no pueden combinarse en forma simultánea sin usar otro operador de distinta precedencia. Por ejemplo dos números reales no tienen asociatividad, ya que no pueden estar en la forma *REAL REAL*, sino que debe haber una operación que los vincule, como por ejemplo un operador aritmético.

La tabla de órdenes de precedencia y asociatividad usada para la interpretación en el lenguaje usado por *N-CD++* se muestra en la siguiente tabla:

Orden	Código	Asociatividad
Menor Precedencia	**AND OR XOR IMP EQV**	Izquierda
	NOT	Derecha
	= **!=** > < >= <=	
	+ **-**	Izquierda
	* /	Izquierda
	FUNCTION	
Mayor Precedencia	**REAL INT BOOL** COUNT **? STRING** CONSTFUNC	
	( )	

Figura 6 – Orden de Precedencia y asociatividad usadas por *N-CD++*

3.3 Funciones y Constantes usadas por el lenguaje

3.3.1 Tratamiento de Valores Booleanos

Esta sección describe las constantes que representan los valores booleanos de la lógica trivalente utilizada por *N-CD++* y se muestran los operadores aplicables sobre las mismas.

3.3.1.1 Constantes Booleanas de la Lógica Trivalente

La lógica trivalente usa los valores **T** ó **1** para representar al valor *TRUE*, **F** ó **0** para representar al *FALSE*, y **?** para representar al *INDEFINIDO*; este último permite representar un estado cuyo valor no puede determinarse.

3.3.1.2 Operadores Booleanos

3.3.1.2.1 Operador *AND*

El comportamiento del operador ***AND*** se define con la siguiente tabla de verdad:

AND	**T**	**F**	**?**
T	T	F	?
F	F	F	F
?	?	F	?

Figura 7 – Comportamiento del operador booleano *AND*

3.3.1.2.2 Operador *OR*

El comportamiento del operador ***OR*** se define con la siguiente tabla de verdad:

OR	**T**	**F**	**?**
T	T	T	T
F	T	F	?
?	T	?	?

Figura 8 – Comportamiento del operador booleano *OR*

3.3.1.2.3 Operador *NOT*

El comportamiento del operador booleano ***NOT*** se define con la siguiente tabla:

NOT	
T	F
F	T
?	?

Figura 9 – Comportamiento del operador booleano *NOT*

3.3.1.2.4 Operador *XOR*

El comportamiento del operador ***XOR*** se define con la siguiente tabla de verdad:

XOR	**T**	**F**	**?**
T	F	T	?
F	T	F	?
?	?	?	?

Figura 10 – Comportamiento del operador booleano *XOR*

3.3.1.2.5 Operador *IMP*

IMP representa la operación de implicación lógica, y su comportamiento se define con la siguiente tabla de verdad:

IMP	**T**	**F**	**?**
T	T	F	?
F	T	T	T
?	T	?	?

Figura 11 – Comportamiento del operador booleano *IMP*

3.3.1.2.6 Operador *EQV*

EQV representa la operación de equivalencia entre valores de la lógica trivalente, y su comportamiento se define con la siguiente tabla de verdad:

EQV	**T**	**F**	**?**
T	T	F	F
F	F	T	F
?	F	F	T

Figura 12 – Comportamiento del operador booleano *EQV*

3.3.2 Funciones y Operaciones para el Tratamiento de Números Reales

3.3.2.1 Operadores Relacionales

Los operadores relacionales trabajan sobre números reales[1] y devuelven un valor booleano perteneciente a la lógica trivalente anteriormente definida. El lenguaje usado por N–CD++ dispone de los operadores ==, !=, >, <, >=, <=, cuyo comportamiento se describe a continuación.

Considerando las definiciones del comportamiento de estos operadores puede verse que no existe un orden total sobre los elementos pertenecientes a los números reales, debido a que en todos los casos el valor **?** no es comparable con un número real tradicional.

3.3.2.1.1 Operador =

El operador = se utiliza para saber si dos números reales son iguales. Su comportamiento se define a continuación:

=	**?**	**n° real**
?	T	?
n° real	?	= de nros. reales

Figura 13 – Comportamiento del operador relacional =

3.3.2.1.2 Operador !=

El operador **!=** se utiliza para saber si dos números reales son distintos. Su comportamiento se define a continuación:

!=	**?**	**n° real**
?	F	?
n° real	?	≠ de nros. reales

Figura 14 – Comportamiento del operador relacional !=

3.3.2.1.3 Operador >

El operador **>** se utiliza para saber si un número real es estrictamente mayor a otro. Su comportamiento se define a continuación:

>	**?**	**n° real**
?	F	?
n° real	?	> de nros. reales

[1] De aquí en más, al referirse al termino *Número Real* se estará considerando a un valor perteneciente al conjunto $\boldsymbol{R} \cup \{ \textbf{?} \}$

Figura 15 – Comportamiento del operador relacional >

3.3.2.1.4 Operador <

El operador < se utiliza para saber si un número real es estrictamente menor a otro. Su comportamiento se define a continuación:

<	?	n° real
?	F	?
n° real	?	< de nros. reales

Figura 16 – Comportamiento del operador relacional <

3.3.2.1.5 Operador <=

El operador <= se utiliza para saber si un número real es menor o igual a otro. Su comportamiento se define a continuación:

<=	?	n° real
?	T	?
n° real	?	≤ de nros. reales

Figura 17 – Comportamiento del operador relacional <=

3.3.2.1.6 Operador >=

El operador >= se utiliza para saber si un número real es mayor o igual a otro. Su comportamiento se define a continuación:

>=	?	n° real
?	T	?
n° real	?	≥ de nros. reales

Figura 18 – Comportamiento del operador relacional >=

3.3.2.2 Operadores Aritméticos

El lenguaje cuenta con operadores para realizar las operaciones más usuales sobre números reales. Cabe considerar que si uno de los operandos es el valor indefinido, entonces el resultado de dicha operación será indefinido. Esto también es válido cuando se usa cualquier tipo de función, y alguno de sus parámetros es indefinido.

Los operadores usados son:

op1 + op2 que devuelve la suma de los operandos.
op1 – op2 devuelve la diferencia entre los operandos.
op1 / op2 devuelve el operando 1 dividido el operando 2.
op1 * op2 devuelve el producto de los operandos.

Figura 19 – Operadores Aritméticos

Para el caso en que se produzca una división por cero, se devolverá el valor indefinido.

3.3.2.3 Funciones sobre Números Reales

3.3.2.3.1 Funciones para la Verificación de Propiedades sobre Números Reales

En esta sección se detallan las funciones que permiten comprobar si un número real cumple ciertas propiedades, como ser un número entero, indefinido, par, impar o primo.

Función Even

Signatura: **even** : *Real* → *Bool*

Descripción: Devuelve *True* si el valor es entero y par. Si el valor es indefinido, devuelve *Indefinido*. En otro caso devuelve *False*.

Ejemplos:
even(?) = F
even(3.14) = F
even(3) = F
even(2) = T

Función Odd

Signatura: **odd** : *Real* → *Bool*

Descripción: Devuelve *True* si el valor es entero e impar. Si el valor es indefinido, devuelve *Indefinido*. En otro caso devuelve *False*.

Ejemplos:
odd(?) = F
odd(3.14) = F
odd(3) = T
odd(2) = F

Función isInt

Signatura: **isInt** : *Real* → *Bool*

Descripción: Devuelve *True* si el valor es entero y no es indefinido. En otro caso devuelve *False*.

Ejemplos:
isInt(?) = F
isInt(3.14) = F
isInt(3) = T

Función isPrime

Signatura: **isPrime** : *Real* → *Bool*

Descripción: Devuelve *True* si el valor es un número primo. En otro caso devuelve *False*.

Ejemplos:
isPrime(?) = F
isPrime(3.14) = F
isPrime(6) = F
isPrime(5) = T

Función isUndefined

Signatura: **isUndefined** : *Real* → *Bool*

Descripción: Devuelve *True* si el valor es indefinido, sino devuelve *False*.

Ejemplos:
isUndefined(?) = T
isUndefined(4) = F

3.3.2.3.2 Funciones Matemáticas

Esta sección describe distintos tipos de funciones usadas habitualmente en trigonometría, así como también para el cálculo de raíces, potencias y logaritmos. Además, se incluyen funciones para obtener el resto y el módulo de la división de números enteros.

3.3.2.3.2.1 Funciones Trigonométricas

Función tan

Signatura: **tan** : *Real a* → *Real*
Descripción: Devuelve la tangente de *a* medida en radianes.
Para los valores cercanos a π/2 radianes devuelve la constante *INF*.
Si *a* es indefinida, devuelve indefinido.
Ejemplos: tan(*PI* / 2) = *INF*
tan(?) = ?
tan(*PI*) = 0

Función sin

Signatura: **sin** : *Real a* → *Real*
Descripción: Devuelve el seno de *a* medida en radianes.
Si *a* tiene el valor **?**, devuelve **?**.

Función cos

Signatura: **cos** : *Real a* → *Real*
Descripción: Devuelve el coseno de *a* medida en radianes.
Si *a* tiene el valor **?**, devuelve **?**.

Función sec

Signatura: **sec** : *Real a* → *Real*
Descripción: Devuelve la secante de *a* medida en radianes.
Si *a* tiene el valor **?**, devuelve **?**.
Si el ángulo es de la forma π/2 + *x*.π, con *x* un número entero, devuelve la constante *INF*.

Función cotan

Signatura: **cotan** : *Real a* → *Real*
Descripción: Calcula la cotangente de *a*.
Si *a* tiene el valor **?**, devuelve **?**.
Si *a* es cero o múltiplo de π, devuelve *INF*.

Función cosec

Signatura: **cosec** : *Real a* → *Real*
Descripción: Calcula la cosecante de *a*.
Si *a* tiene el valor **?**, devuelve **?**.
Si *a* es cero o múltiplo de π, devuelve *INF*.

Función atan

Signatura: **atan** : *Real a* → *Real*
Descripción: Devuelve el arco tangente de *a* medida en radianes, que se define como el valor *b* tal que tan(*b*) = *a*.
Si *a* tiene el valor **?**, devuelve **?**.

Función asin

Signatura: **asin** : *Real a* → *Real*
Descripción: Devuelve el arco seno de *a* medido en radianes, que se define como el valor *b* tal que: sin(*b*) = *a*.
Si *a* tiene el valor **?**, ó si *a* ∉ [-1, 1], entonces devuelve **?**.

Función acos

Signatura: **acos** : *Real a* → *Real*

Descripción: Devuelve el arco coseno de *a* medido en radianes, que se define como el valor *b* tal que $\cos(b) = a$.
Si *a* tiene el valor **?**, ó si $a \notin [-1, 1]$, entonces devuelve **?**.

Función asec

Signatura: **asec** : *Real a* → *Real*

Descripción: Devuelve el arco secante de *a* medido en radianes, que se define como el valor *b* tal que $\sec(b) = a$.
Si *a* es indefinida (**?**) ó si $|a| < 1$, entonces devuelve **?**.

Función acotan

Signatura: **acotan** : *Real a* → *Real*

Descripción: Devuelve el arco cotangente de *a* medido en radianes, que se define como el valor *b* tal que $\mathrm{cotan}(b) = a$.
Si *a* es indefinida (**?**), devuelve **?**.

Función sinh

Signatura: **sinh** : *Real a* → *Real*

Descripción: Devuelve el seno hiperbólico de *a* medido en radianes.
Si *a* tiene el valor **?**, devuelve **?**.

Función cosh

Signatura: **cosh** : *Real a* → *Real*

Descripción: Devuelve el coseno hiperbólico de *a* medido en radianes, definido como:
$\cosh(x) = (e^{x} + e^{-x}) / 2$.
Si *a* tiene el valor **?**, devuelve **?**.

Función tanh

Signatura: **tanh** : *Real a* → *Real*

Descripción: Devuelve la tangente hiperbólica de *a* medida en radianes, que se define como:
sinh(a) / cosh(a).
Si *a* tiene el valor **?**, devuelve **?**.

Función sech

Signatura: **sech** : *Real a* → *Real*

Descripción: Devuelve la secante hiperbólica de *a* medida en radianes, definida como $1 / \cosh(a)$
Si *a* tiene el valor **?**, devuelve **?**.

Función cosech

Signatura: **cosech** : *Real a* → *Real*

Descripción: Devuelve la cosecante hiperbólica de *a* medida en radianes.
Si *a* tiene el valor **?**, devuelve **?**.

Función atanh

Signatura: **atanh** : *Real a* → *Real*

Descripción: Devuelve el arco tangente hiperbólica de *a* medida en radianes, que se define como el valor *b* tal que $\tanh(b) = a$.
Si *a* tiene el valor **?**, ó si su valor absoluto es mayor a 1 (es decir, $a \notin [-1, 1]$), entonces devuelve **?**.

Función asinh

Signatura: **asinh** : *Real a* → *Real*

Descripción: Devuelve el arco seno hiperbólico de *a* medido en radianes, que se define como el valor *b* tal que $\sinh(b) = a$.

Si *a* tiene el valor **?**, devuelve **?**.

Función acosh

Signatura: **acosh** : *Real a* → *Real*

Descripción: Devuelve el arco coseno hiperbólico de *a* medido en radianes, que se define como el valor *b* tal que cosh(*b*) = *a*.
Si *a* tiene el valor **?** ó es menor a 1, entonces devuelve **?**.

Función asech

Signatura: **asech** : *Real a* → *Real*

Descripción: Devuelve el arco secante hiperbólico de *a* medido en radianes, que se define como el valor *b* tal que sech(*b*) = *a*.
Si *a* tiene el valor indefinido, devuelve **?**. Si es cero devuelve la constante *INF*.

Función acosech

Signatura: **acosech** : *Real a* → *Real*

Descripción: Devuelve el arco cosecante hiperbólico de *a* medido en radianes, que se define como el valor *b* tal que cosech(*b*) = *a*.
Si *a* tiene el valor indefinido, devuelve **?**. Si es cero devuelve la constante *INF*.

Función acotanh

Signatura: **acotanh** : *Real a* → *Real*

Descripción: Devuelve el arco cotangente hiperbólico de *a* medido en radianes, que se define como el valor *b* tal que cotanh(*b*) = *a*.
Si *a* tiene el valor indefinido, devuelve **?**. Si es 1 devuelve la constante *INF*.

Función hip

Signatura: **hip** : *Real c1* x *Real c2* → *Real*

Descripción: Calcula la hipotenusa del triángulo formado por los catetos *c1* y *c2*.
Si *c1* ó *c2* son indefinidos o negativos, devuelve **?**.

3.3.2.3.2.2 Funciones para el Cálculo de Raíces, Potencias y Logaritmos.

Función sqrt

Signatura: **sqrt** : *Real a* → *Real*

Descripción: Devuelve la raíz cuadrada de *a*.
Si *a* tiene el valor **?** o es negativo, devuelve **?**.

Ejemplos:
sqrt(4) = 2
sqrt(2) = 1.41421
sqrt(0) = 0
sqrt(-2) = ?
sqrt(?) = ?

Nota: sqrt(*x*) es equivalente a **root**(*x*, 2) $\forall x$

Función exp

Signatura: **exp** : *Real x* → *Real*

Descripción: Devuelve el valor de e^x.
Si *x* tiene el valor **?**, devuelve **?**.

Ejemplos:
exp(?) = ?
exp(–2) = 0.135335
exp(1) = 2.71828
exp(0) = 1

Función ln

Signatura: **ln** : *Real a* → *Real*

Descripción: Devuelve el logaritmo natural de a.
Si a tiene el valor **?** o es menor o igual a cero, devuelve **?**.
Ejemplos: ln(–2) = ?
ln(0) = ?
ln(1) = 0
ln(?) = ?
Nota: ln(x) es equivalente a **logn**(x, e) $\forall x$

Función log

Signatura: **log** : *Real a* → *Real*
Descripción: Devuelve el logaritmo en base 10 de a.
Si a tiene el valor **?** o es menor o igual a cero, devuelve **?**.
Ejemplos: log(3) = 0.477121
log(–2) = ?
log(?) = ?
log(0) = ?
Nota: log(x) es equivalente a **logn**(x, 10) $\forall x$

Función logn

Signatura: **logn** : *Real a* x *Real n* → *Real*
Descripción: Devuelve el logaritmo en base n del valor a.
Si a ó n son indefinidos, negativos o cero, devuelve **?**.
Notas: logn(x, e) es equivalente a **ln**(x) $\forall x$
logn(x, 10) es equivalente a **log**(x) $\forall x$

Función power

Signatura: **power** : *Real a* x *Real b* → *Real*
Descripción: Devuelve a^b.
Si a ó b tienen el valor **?** ó b no es entero, devuelve **?**.

Función root

Signatura: **root** : *Real a* x *Real n* → *Real*
Descripción: Devuelve la raíz n-ésima de a.
Si a ó n son indefinidos, devuelve **?**. También devuelve ese valor en el caso en que a sea negativa o n sea cero.
Ejemplos: root(27, 3) = 3
root(8, 2) = 3
root(4, 2) = 2
root(2, ?) = ?
root(3, 0.5) = 9
root(–2, 2) = ?
root(0, 4) = 0
root(1, 3) = 1
root(4, 3) = 1.5874
Nota: root(x, 2) es equivalente a **sqrt**(x) $\forall x$

3.3.2.3.2.3 Funciones para el cálculo del MCM, MCD y Resto de la División Numérica

Función LCM

Signatura: **lcm** : *Real a* x *Real b* → *Real*
Descripción: Calcula el *Mínimo Común Múltiplo* (Less Common Multiplier) entre a y b.
Si a ó b tienen el valor **?** ó no son enteros, devuelve **?**.
El número devuelto siempre es entero.

Función GCD

Signatura: **gcd** : *Real a* x *Real b* → *Real*

Descripción: Calcula el *Máximo Común Divisor* (Greater Common Divisor) entre *a* y *b*.
Si *a* ó *b* tienen el valor **?** ó no son enteros, devuelve **?**.
El número devuelto siempre es entero.

Función remainder

Signatura: **remainder** : *Real a* x *Real b* → *Real*

Descripción: Calcula el resto de la división entre *a* y *b*. El valor devuelto es *a* – *n* * *b*, donde *n* es el cociente *a*/*b* redondeado como un número entero.
Si *a* ó *b* tienen el valor **?**, devuelve **?**.

Ejemplos:
remainder(12, 3) = 0
remainder(14, 3) = 2
remainder(4, 2) = 0
remainder(0, y) = 0 ∀ y ≠ **?**
remainder(x, 0) = x ∀ x
remainder(1.25, 0.3) = 0.05
remainder(1.25, 0.25) = 0
remainder(?, 3) = ?
remainder(5, ?) = ?

3.3.2.3.3 Funciones para la Conversión de Valores Reales a Enteros

En esta sección se detallan funciones para convertir valores reales a enteros mediante las técnicas de redondeo y truncamiento. También existen funciones para obtener la parte fraccionaria de un valor real.

Función round

Signatura: **round** : *Real a* → *Real*

Descripción: Redondea el valor *a* al entero más cercano.
Si *a* tiene el valor **?**, devuelve **?**.

Ejemplos:
round(4) = 4
round(?) = ?
round(4.1) = 4
round(4.7) = 5
round(–3.6) = –4

Función trunc

Signatura: **trunc**: *Real x* → *Real*

Descripción: Devuelve el mayor número entero menor o igual a *x*.
Si *x* tiene el valor **?**, devuelve **?**.

Ejemplos:
trunc(4) = 4
trunc(?) = ?
trunc(4.1) = 4
trunc(4.7) = 4

Función truncUpper

Signatura: **truncUpper**: *Real x* → *Real*

Descripción: Devuelve el menor número entero mayor o igual a *x*.
Si *x* tiene el valor **?**, devuelve **?**.

Ejemplos:
truncUpper(4) = 4
truncUpper(?) = ?
truncUpper(4.1) = 5
truncUpper(4.7) = 5

Función fractional

Signatura: **fractional** : *Real a → Real*
Descripción: Devuelve la parte fraccionaria del valor real *a*, incluyendo el signo.
Si *a* tiene el valor **?**, devuelve **?**.
Ejemplos: fractional(4.15) = 0.15
fractional(?) = ?
fractional(-3.6) = -0.6

3.3.2.3.4 Funciones para el Tratamiento del Signo de Valores Numéricos

Función abs

Signatura: **abs** : *Real a → Real*
Descripción: Devuelve el valor absoluto de *a*.
Si *a* tiene el valor **?**, devuelve **?**.
Ejemplos: abs(4.15) = 4.15
abs(?) = ?
abs(-3.6) = 3.6
abs(0) = 0

Función sign

Signatura: **sign** : *Real a → Real*
Descripción: Devuelve el signo de *a* en la siguiente forma:
Si *a* > 0, devuelve 1.
Si *a* < 0, devuelve –1.
Si *a* = 0, devuelve 0.
Si *a* = **?**, devuelve **?**.

Función randomSign

Ver la sección 3.3.2.3.8.

3.3.2.3.5 Funciones para el Manejo de Números Primos

Si bien el lenguaje permite el manejo de números primos, todas estas instrucciones son muy complejas, lo que puede incrementar considerablemente el tiempo de simulación.

Función isPrime

Ver la sección 3.3.2.3.1.

Función nextPrime

Signatura: **nextPrime** : Real *r → Real*
Descripción: Devuelve el próximo número primo mayor a *r*.
Si *r* es **?**, devuelve **?**.
Es probable que ocurra un desborde numérico al calcular el próximo primo. En ese caso se devuelve el valor *INF*.

Función nth_Prime

Signatura: **nth_Prime** : Real *n → Real*
Descripción: Devuelve el *n*-ésimo primo, considerando como primer número primo al 2.
Si *n* es **?** o no es entero, devuelve **?**.
Es probable que ocurra un desborde numérico al calcular el número primo. En ese caso se devuelve el valor *INF*.

3.3.2.3.6 Funciones para la obtención de Mínimos y Máximos

Función min

Signatura: **min** : *Real a* x *Real b* → *Real*

Descripción: Devuelve el mínimo entre *a* y *b*.
Si *a* ó *b* son indefinidos, devuelve **?**.

Función max

Signatura: **max** : *Real a* x *Real b* → *Real*

Descripción: Devuelve el máximo entre *a* y *b*.
Si *a* ó *b* son indefinidos, devuelve **?**.

3.3.2.3.7 Funciones Condicionales

Las funciones descriptas en esta sección permiten devolver determinados valores reales dependiendo de la evaluación de una condición lógica especificada.

Función if

Signatura: **if** : Bool *c* x Real *t* x Real *f* → *Real*

Descripción: Si la condición *c* evalúa a *TRUE*, devuelve la evaluación de *t*. Sino devuelve la evaluación de *f*.
Cabe destacar que tanto los valores de *t* y *f* pueden provenir de la evaluación de cualquier expresión que devuelva un valor real, incluso otra sentencia *if*.

Ejemplos: Se desea devolver el valor 1.5 cuando el logaritmo natural de la celda (0, 0) es nulo o si la celda es negativa, ó 2 en otro caso. En este caso deberá escribirse:

if (ln((0, 0)) = 0 or (0, 0) < 0, 1.5, 2)

Si se desea devolver el valor de las celdas (1, 1) + (2, 2) cuando la celda (0, 0) no es nula ó la raíz cuadrada de (3, 3) en otro caso, deberá escribirse:

if((0, 0) != 0, (1, 1) + (2, 2), sqrt((3, 3)))

También puede usarse para el tratamiento de desbordes numéricos. Por ejemplo, si el factorial de la celda (0, 1) produce un desborde numérico devolver –1, sino devolver el resultado obtenido. En este caso deberá escribirse:

if(fact((0, 1)) = *INF*, –1, fact((0, 1)))

Función ifu

Signatura: **ifu** : Bool *c* x Real *t* x Real *f* x Real *u* → *Real*

Descripción: Si la condición *c* evalúa a *TRUE*, devuelve la evaluación de *t*. Si evalúa a *FALSE*, devuelve la evaluación de *f*. Sino, en el caso en que evaluó a indefinido, devuelve la evaluación de *u*.

Ejemplos: Se desea devolver el valor de la celda (0, 0) si dicha celda no es cero ni indefinida, 1 si el valor de la celda es 0, y π si la celda tiene el valor indefinido. En este caso deberá invocarse:

ifu((0, 0) != 0, (0, 0), 1, *PI*)

3.3.2.3.8 Funciones Probabilísticas

Función randomSign

Signatura: **randomSign** : → *Real*

Descripción: Devuelve un valor numérico que representa un signo (+1 ó –1) en forma aleatoria, con igual probabilidad para ambos valores.

Función random

Signatura: **random** : → *Real*

Descripción: Devuelve un número real pseudo-aleatorio perteneciente al intervalo (0, 1), con distribución uniforme.

Nota: random es equivalente a ejecutar *uniform(0,1)*.

Función chi

Signatura: **chi** : *Real df* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución Chi-Cuadrada con *df* grados de libertad.
Si *df* es indefinido, negativo o cero, devuelve **?**.

Función beta

Signatura: **beta** : *Real a* x *Real b* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución Beta con parámetros *a* y *b*.
Si *a* o *b* son indefinidos ó menores a 10^{-37}, devuelve **?**.

Función exponential

Signatura: **exponential** : *Real av* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución Exponencial con media *av*.
Si *av es* indefinida ó negativa, devuelve **?**.

Función f

Signatura: **f** : *Real dfn* x *Real dfd* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución F, con *dfn* grados de libertad para el numerador, y *dfd* para el denominador.
Si *dfn* ó *dfd* son indefinidos, negativos o cero, devuelve **?**.

Función gamma

Signatura: **gamma** : *Real a* x *Real b* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución Gamma con parámetros (*a*, *b*).
Si *a* ó *b* son indefinidos, negativos o cero, devuelve **?**.

Función normal

Signatura: **normal** : *Real* μ x *Real* σ → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución Normal (μ, σ), donde μ es la media, y σ es el desvío estándar.
Si μ *ó* σ son indefinidos, ó σ es negativo, devuelve **?**.

Función uniform

Signatura: **uniform** : *Real a* x *Real b* → *Real*

Descripción: Devuelve un número real pseudo-aleatorio con distribución uniforme, perteneciente al intervalo (*a*, *b*).
Si *a* ó *b* son indefinidos, ó *a* > *b*, devuelve **?**.

Nota: *uniform(0, 1)* es equivalente a ejecutar la función *random*.

Función binomial

Signatura: **binomial** : *Real n* x *Real p* → *Real*

Descripción: Devuelve un número pseudo-aleatorio con distribución Binomial, donde *n* es el número de intentos, y *p* es la probabilidad de éxito de un evento.
Si *n* ó *p* son indefinidos, *n* no es entero ó negativo, ó *p* no pertenece al intervalo [0, 1], entonces devuelve **?**.
El número devuelto siempre es entero.

Función poisson

Signatura: **poisson** : *Real n* → *Real*

Descripción: Devuelve un número pseudo-aleatorio con distribución Poisson, con media *n*.
Si *n* es indefinida ó negativa, devuelve **?**.

El número devuelto siempre es entero.

Función randInt

Signatura: **randInt** : *Real n* → *Real*

Descripción: Devuelve un número pseudo-aleatorio entero perteneciente al intervalo [0, *n*], con distribución uniforme.
Si *n* es indefinida, devuelve **?**.

Nota: *randInt*(*n*) es equivalente a *round*(*uniform*(*0*, *n*))

3.3.2.3.9 Funciones para cálculo de Factoriales y Combinatorios

Función fact

Signatura: **fact** : *Real a* → *Real*

Descripción: Devuelve el factorial de *a*.
Si *a* es indefinido, negativo ó no es entero, entonces devuelve **?**.
Si ocurre un desborde numérico durante el cálculo del valor a retornar, devuelve la constante *INF*.

Ejemplos:
fact(3) = 6
fact(0) = 1
fact(5) = 120
fact(13) = 1.93205e+09
fact(43) = *INF*

Función comb

Signatura: **comb** : *Real a* x *Real b* → *Real*

Descripción: Devuelve el combinatorio $\binom{a}{b}$
Si *a* ó *b* son indefinidos, son negativos o cero, ó no son enteros, entonces devuelve **?**. Este mismo valor es retornado si a < b.
Si ocurre un desborde numérico durante el cálculo del valor a retornar, devuelve la constante *INF*.

3.3.2.4 Funciones que actúan sobre una Celda y su Vecindario

En esta sección se detallan las funciones que permiten contar la cantidad de celdas pertenecientes al vecindario cuyo estado tiene determinado valor, como así también la función *cellPos* que permite proyectar un elemento de la tupla que referencia a la celda.

Función stateCount

Signatura: **stateCount** : *Real a* → *Real*

Descripción: Devuelve el número de vecinos de la celda cuyo estado sea igual al valor *a*.

Función trueCount

Signatura: **trueCount** : → *Real*

Descripción: Devuelve el número de vecinos de la celda con estado en 1.
Esta función es equivalente a *stateCount*(1) y se mantiene en el lenguaje para ofrecer compatibilidad con la versión original de la herramienta (*CD++*).

Función falseCount

Signatura: **falseCount** : → *Real*

Descripción: Devuelve el número de vecinos de la celda con estado en 0.
Esta función es equivalente a *stateCount*(0) y se mantiene en el lenguaje para ofrecer compatibilidad con la versión original de la herramienta.

Función undefCount

Signatura: **undefCount** : → *Real*

Descripción: Devuelve el número de vecinos de la celda con estado indefinido (**?**).
Esta función es equivalente a *stateCount*(?) y se mantiene en el lenguaje para ofrecer compatibilidad con la versión original de la herramienta.

Función cellPos

Signatura: **cellPos** : *Real i* → *Real*

Descripción: Devuelve la i–ésima posición dentro de la tupla que referencia a la celda que esta ejecutando la función de transición, es decir, dada la celda $(x_0,x_1,...,x_n)$, entonces cellPos(i) = x_i.
Si el valor de *i* no es entero, entonces será automáticamente truncado a tal al ser usado por *cellPos*.
Si $i \notin [0, n+1)$, donde n es la dimensión del modelo, se producirá un error y se abortara la simulación.
El valor devuelto siempre será entero.

Ejemplos: Dada la celda (4, 3, 10, 2):
cellPos(0) = 4
cellPos(3.99) = cellPos(3) = 2
cellPos(1.5) = cellPos(1) = 3
cellPos(–1) y cellPos(4) generarán un error.

3.3.2.5 Funciones para la Obtención del Tiempo de Simulación

Función Time

Signatura: **time** : → *Real*

Descripción: Devuelve la hora actual de simulación, en el momento en que la regla esta siendo evaluada, expresada en milisegundos.

3.3.2.6 Funciones para la Conversión de Valores entre distintas Unidades

3.3.2.6.1 Funciones para la Conversión de Grados a Radianes

Función radToDeg

Signatura: **radToDeg** : Real *r* → *Real*

Descripción: Convierte el valor *r* de radianes a grados sexagesimales.
Si *r* es **?**, devuelve **?**.

Función degToRad

Signatura: **degToRad** : Real *r* → *Real*

Descripción: Convierte el valor *r* de grados sexagesimales a radianes.
Si *r* es **?**, devuelve **?**.

3.3.2.6.2 Funciones para la Conversión de Coordenadas Rectangulares a Polares

Función rectToPolar_r

Signatura: **rectToPolar_r** : Real *x* x Real *y* → *Real*

Descripción: Convierte la coordenada cartesiana (*x*, *y*) a la forma polar (*r*, θ), y devuelve *r*.
Si *x ó y* son indefinidos, devuelve **?**.

Función rectToPolar_angle

Signatura: **rectToPolar_angle** : Real *x* x Real *y* → *Real*

Descripción: Convierte la coordenada cartesiana (*x*, *y*) a la forma polar (*r*, θ), y devuelve θ.

Si x *ó* y son indefinidos, devuelve **?**.

Función polarToRect_x

Signatura: **polarToRect_x** : Real r x Real $\theta \rightarrow$ *Real*

Descripción: Convierte la coordenada polar (r, θ) a la forma cartesiana (x, y), y devuelve x. Si r *ó* θ son indefinidos, ó r es negativo, devuelve **?**.

Función polarToRect_y

Signatura: **polarToRect_y** : Real r x Real $\theta \rightarrow$ *Real*

Descripción: Convierte la coordenada polar (r, θ) a la forma cartesiana (x, y), y devuelve y. Si r *ó* θ son indefinidos, ó r es negativo, devuelve **?**.

3.3.2.6.3 Funciones para la Conversión de Temperaturas en distintas unidades

Función CtoF

Signatura: **CtoF** : Real → *Real*

Descripción: Convierte un valor expresado en grados Centígrados a grados Fahrenheit. Si el valor es indefinido, devuelve **?**.

Función CtoK

Signatura: **CtoK** : Real → *Real*

Descripción: Convierte un valor expresado en grados Centígrados a grados Kelvin. Si el valor es indefinido, devuelve **?**.

Función KtoC

Signatura: **KtoC** : Real → *Real*

Descripción: Convierte un valor expresado en grados Kelvin a grados Centígrados. Si el valor es indefinido, devuelve **?**.

Función KtoF

Signatura: **KtoF** : Real → *Real*

Descripción: Convierte un valor expresado en grados Kelvin a grados Fahrenheit. Si el valor es indefinido, devuelve **?**.

Función FtoC

Signatura: **FtoC** : Real → *Real*

Descripción: Convierte un valor expresado en grados Fahrenheit a grados Centígrados. Si el valor es indefinido, devuelve **?**.

Función FtoK

Signatura: **FtoK** : Real → *Real*

Descripción: Convierte un valor expresado en grados Fahrenheit a grados Kelvin. Si el valor es indefinido, devuelve **?**.

3.3.2.7 Funciones para la manipulación de Valores de los Puertos de Entrada y Salida

Función portValue

Signatura: **portValue** : String $p \rightarrow$ *Real*

Descripción: Devuelve el último valor arribado por el puerto de entrada p de la celda que se está evaluando. Esta función sólo podrá ser usada cuando se definan funciones de transición en la cláusula ***PortInTransition*** (ver sección 2.3) la cual permite dar comportamiento a la celda ante el arribo de un mensaje por un puerto de entrada. Si se utiliza en una función de

transición no definida con *PortInTransition* se generará un error en la interpretación de la regla.

Si en el momento de evaluar la función *portValue* aun no arribó un mensaje por el puerto *p* desde el inicio de la simulación, se obtendrá el valor indefinido (**?**). Una vez que arribó un mensaje, al consultarse se obtendrá el último valor ingresado.

Mediante del uso del string "***thisPort***" pasado como parámetro a *portValue*, es posible indicar al simulador que el valor del puerto que se quiere obtener es el puerto por el cual llegó el mensaje. A continuación se ejemplifica su uso:

Supóngase que una celda tiene asociado el puerto de entrada *A*, y otra celda tiene asociado el puerto *B*. Entonces es posible definir funciones para calcular el valor de la celda al arribar un mensaje por los mismos. En este caso se definen las funciones:

```
PortInTransition: portA@celda(0,0)     functionA
PortInTransition: portB@celda(1,1)     functionB

[functionA]
rule: 10     100    { portValue(portA) > 10 }
rule: 0      100    { t }

[functionB]
rule: 10     100    { portValue(portB) > 10 }
rule: 0      100    { t }
```

Figura 20 – Ejemplo de uso de la función *portValue*

En el ejemplo, se creó una función para cada puerto. El comportamiento de ambas funciones es el mismo, pero debido a que los nombres de los puertos son distintos, no es posible unificar ambas funciones. Una posible solución es hacer que los puertos de las celdas tengan igual nombre, por ejemplo *portN*, y al referenciar al valor del puerto se realiza un *portValue*(*portN*). La otra solución es trabajar con ***thisPort*** como se muestra en la Figura 21.

```
PortInTransition: portA@celda(0,0)     functionA
PortInTransition: portB@celda(1,1)     functionA

[functionA]
rule: 10     100    { portValue(thisPort) > 10 }
rule: 0      100    { t }
```

Figura 21 – Ejemplo de uso de la función *portValue* junto a *thisPort*

De esta forma, se unifica el comportamiento, evitando la reescritura de una función por más que los puertos tengan distintos nombres.

En la sección 16.3 se ejemplifica el uso de la función *portValue* en la implementación de un modelo para la clasificación de materias primas.

Función send

Signatura: **send** : String *p* x Real $x \rightarrow 0$

Descripción: Envía el valor *x* por el puerto de salida *p*.
Si la celda no tiene asociado al puerto p entonces se producirá un error al evaluar la función y se abortará la simulación.

Cada vez que se produce un cambio en una celda, *N–CD++* envía dicho valor por el puerto *Out* de la misma. Pero en ciertos casos es deseable enviar determinado valor (que no necesariamente debe ser el estado de la misma) a alguna celda o modelo DEVS en particular. Para estos casos se utiliza la función *send*.

Se recomienda el uso de la función de la siguiente forma:

{ nuevo_valor + **send**(P, V) }   demora { condición }

En ese caso, si la *condición* evalúa a verdadero, entonces el nuevo valor de la celda será el especificado y, además, se enviará el valor V por el puerto P.

La función *send* siempre devuelve el valor 0, debido a que fue creada con la idea de enviar un valor por un puerto sin necesidad de alterar el valor de la celda, como se ejemplifica en el siguiente caso:

{ (0,0) + **send**(puerto1, 15 * log(10)) } 100 { (0,0) > 10 }

Nota: **Send** es una función más del lenguaje, por lo que puede ser usada en cualquier lugar donde sea posible, como por ejemplo en la definición de una *condición*. Pero esto no es deseable debido a que una condición puede evaluarse y resultar ser inválida, y por lo tanto se ejecutará el comando **send** enviando un valor por un puerto en forma indiscriminada. En cambio las expresiones que representan el nuevo valor de la celda o la que define el valor de la demora son evaluadas solo cuando la condición es válida.

3.3.3 Constantes Predefinidas por el Lenguaje

El lenguaje usado por *N–CD++* permite el uso de constantes predefinidas utilizadas frecuentemente en los dominios de la física y la química.

Las constantes pueden verse como funciones que no reciben parámetros y que devuelven siempre el mismo valor real.

Constante Pi

Devuelve 3.14159265358979323846, que representa el valor de π, la relación entre la circunferencia y el radio de un círculo.

Constante e

Devuelve 2.7182818284590452353, el valor que representa a la base de los logaritmos naturales.

Constante INF

Esta constante representa al valor infinito, aunque en realidad devuelve el máximo valor representable en un *Double* (en procesadores 80x86 de Intel, este número es 1.79769×10^{308}).
Cabe destacar que si, por ejemplo, hacemos *x + INF – INF*, donde x es cualquier valor real, dará como resultado 0, pues el operador + es asociativo a izquierda, por lo que se resolverá:

$(x + INF) - INF = INF - INF = 0$.

Nota: Al generarse un desborde numérico producido por cualquier operación, se devuelve *INF* ó *–INF*. Por ejemplo: power(12333333, 78134577) = *INF*

Constante electron_mass

Devuelve la masa de un electrón, que es $9.1093898 \times 10^{-28}$ gramos.

Constante proton_mass

Devuelve la masa de un protón, que es $1.6726231 \times 10^{-24}$ gramos.

Constante neutron_mass

Devuelve la masa de un neutrón, que es 1.6749286 x 10^{-24} gramos.

Constante Catalan

Devuelve la constante de Catalan, definida como $\sum_{k=0}^{\infty}(-1)^{k}.(2^{k}+1)^{-2}$, que es aproximadamente 0.9159655941772.

Constante Rydberg

Devuelve la constante de Rydberg, definida como 10.973.731,534 / m.

Constante grav

Devuelve la constante gravitacional, definida como 6,67259 x 10^{-11} m^3 / (kg . s^2)

Constante bohr_radius

Devuelve el radio de Bohr, definido como 0,529177249 x 10^{-10} m.

Constante bohr_magneton

Devuelve el valor del magnetón de Bohr, definido como 9,2740154 x 10^{-24} joule / tesla.

Constante Boltzmann

Devuelve el valor de la constante de Boltzmann, definida como 1,380658 x 10^{-23} joule / °K.

Constante accel

Devuelve la constante de aceleración estándar, definida como 9,80665 m / seg^2.

Constante light

Devuelve la constante que representa la velocidad de la luz en el vacío, definida como 299.792.458 m / seg.

Constante electron_charge

Devuelve el valor de la carga de un electrón, definido como 1,60217733 x 10^{-19} coulomb.

Constante Planck

Devuelve la constante de Planck, definida como 6,6260755 x 10^{-34} joule . seg

Constante Avogadro

Devuelve la constante de Avogadro, definida como 6,0221367 x 10^{23} moles.

Constante amu

Devuelve el valor de la unidad de masa atómica (Atomic Mass Unit), definida como 1,6605402 x 10^{-27} kg.

Constante pem

Devuelve la relación entre la masa del protón y la del electrón (Proton-Electron Mass), definida como 1836,152701.

Constante ideal_gas

Devuelve la constante del gas ideal, definida como 22,41410 litros / moles.

Constante Faraday

Devuelve la constante de Faraday, definida como 96485,309 coulomb / mol.

Constante Stefan_boltzmann

Devuelve la constante de Stefan-Boltzmann, definida como 5,67051 x 10^{-8} Watt / (m^2 . $°K^4$)

Constante golden

Devuelve el *Golden Ratio*, definido como $\frac{1+\sqrt{5}}{2}$.

Constante euler_gamma

Devuelve el valor Gama de Euler, definido como 0.5772156649015.

3.4 Mecanismos para Evitar la Reescritura de Reglas

En esta sección se describen distintas técnicas que permiten evitar la reescritura de reglas, permitiendo la reutilización de las mismas en otros modelos, y facilitando la lectura y mantenimiento del modelo por parte del usuario.

3.4.1 Cláusula Else

Cuando se utiliza la cláusula **portInTransition** (ver sección 2.3) para la descripción de la función a usar en caso de que arribe un evento externo a través de un puerto de entrada de una celda, es posible emplear la cláusula **else** para permitir darle a la misma un comportamiento alternativo en caso de que no se cumplan ninguna de las reglas de la función declarada, y evitar así la reescritura de código.

En la Figura 22 se puede observar, mediante un ejemplo, el formato para el uso de la cláusula *Else*. Las celdas del modelo ejemplificado usan la función *default_rule* para el cálculo de su nuevo estado, y la celda (13,13) usa la función *another_rule* cuando arriba un evento externo por el puerto *In* de dicha celda. Esta función esta compuesta por una serie de reglas. Si al evaluar las condiciones de todas estas reglas ninguna es válida, la cláusula *else* determina que se use la función *default_rule* para el cálculo del estado de la celda.

```
[demoModel]
type: cell
...
link: in  in@demoModel(13,13)
localTransition: default_rule
portInTransition: in@demoModel(13,13)    another_rule

[default_rule]
rule: ...
...
rule: ...

[another_rule]
rule: 1 1000 { portValue(thisPort) = 0 }
...
else: default_rule
```

Figura 22 – Ejemplo del uso de la cláusula *Else*

La cláusula *Else* puede llamar a cualquier función que defina el comportamiento de una celda, incluso a otra función que contenga otra cláusula *Else* y que describa la conducta ante el arribo de un evento por un puerto de otra celda. Sin embargo, un mal uso de estas podría generar una referencia circular, las cuales no son detectadas por el simulador, y que provocaría un ciclo sin fin que bloquearía al proceso de simulación, como se muestra en la Figura 23.

```
[another_rule1]
rule: 1    1000 { portValue(thisPort) = 0 }
rule: 1.5  1000 { (0,0) = 5 }
rule: 3    1500 { (1,1) + (0,0) >= 1 }
else: another_rule2

[another_rule2]
rule: 1 1000 { (0,0) + portValue(thisPort) > 3 }
```

```
else: another_rule1
```

Figura 23 – Ejemplo de una referencia circular debido al mal uso de la cláusula *Else*

Estas referencias circulares también pueden darse en forma menos directa a la citada anteriormente, donde podrían estar implicadas *n* funciones, donde la primer función referencia por medio de un *else* a la segunda, la segunda referencia a la tercera, y así hasta que la función *n*–1 referencia a la *n*–ésima, y la *n*–ésima referencia a la primera.

Cuando desde la cláusula *else* se referencia a la misma función donde esta siendo es usada, como se muestra en la Figura 24, la herramienta detectará esta situación y producirá un error durante la etapa de parseo de las reglas, informando tal situación al usuario y finalizando su tarea.

```
[another_rule]
rule: ...
rule: ...
else: another_rule
```

Figura 24 – Ejemplo de una referencia circular directa detectada por el simulador

3.4.2 Preprocesador – Uso de Macros

La herramienta proporciona ciertas facilidades al lenguaje por medio de un preprocesador, que actúa sobre el archivo de definición de modelos, antes de la carga de los mismos. Dicho preprocesador puede ser desactivado mediante el parámetro **–b** durante la invocación del simulador, acelerando la carga de los modelos, pero no pudiendo aprovechar las ventajas que brinda el mismo.

La cláusula **#include** permite incluir el contenido de un archivo. Su formato es:

#include(*fileName*)

donde *fileName* es el nombre del archivo que contiene la definición de las macros. Dicho archivo debe encontrarse en el mismo directorio donde se encuentra el archivo de definición de modelos.

La cláusula *#include* debe estar contenida sólo en los archivos de definición de modelos, y puede existir más de una inclusión de distintos archivos dentro de la definición de modelos.

Las cláusulas **#BeginMacro** y **#EndMacro** permiten dar comienzo y fin a la definición de una macro.
Una definición de macro tiene la forma:

```
#BeginMacro(nombreMacro)
...
...contenido de la macro...
...
#EndMacro
```

Figura 25 – Formato de la definición de una macro

El contenido de la macro es arbitrario, pudiendo abarcar cualquier cantidad de líneas. Las definiciones de macros no pueden estar contenidas en el mismo archivo donde son invocadas.

La cláusula **#Macro** permite el uso de una macro previamente definida, reemplazando el texto que la invoca por el contenido de dicha macro. Su formato es:

#Macro(*nombreMacro*)

El archivo de macros puede contener cualquier cantidad de macros, por más que estas nunca sean usadas en el modelo.

El texto que figura fuera de la definición de una macro es ignorado, permitiendo de esta forma incluir comentarios sobre la funcionalidad del mismo con solo escribir dicho texto antes o después de la definición.

Si una macro requerida no es encontrada en ninguno de los archivos incluidos con la cláusula *#include*, se generará un error y la herramienta finalizará su ejecución.

Los *#include* pueden estar definidos en cualquier lugar del archivo, pero siempre deber estar antes de la cláusula *#Macro* que utilice una macro cuya descripción este contenida en el archivo referenciado por el #include.

Dentro de la definición de una macro no puede realizarse una invocación a otra macro.

El preprocesador permite también el uso de comentarios en cualquier parte de un archivo **.MA**. Los comentarios empiezan con el carácter '**%**', y cuando el preprocesador encuentra un comentario, ignora el string que se encuentren comprendido entre el carácter '**%**' hasta el fin de la línea.

```
% Comienzo de la definición de reglas
Rule : 1  100  { truecount > 1 or (0,0,1) = 2 }    % Valida la presencia
                                                    % de otro individuo.
```

Figura 26 – Ejemplo del uso de Comentarios

Si un archivo contiene invocación a macros y/o usa comentarios, y al ejecutar el simulador se le pasa el parámetro **–b** para desactivar al preprocesador, esto generará un incorrecto parseo de los modelos, que quizás no genere un error que aborte a la simulación, pero pudiendo no llegar a interpretar correctamente todos los modelos y logrando resultados no deseados.

En la sección 16.5 puede observarse la implementación de una variante del Juego de la Vida para 4 dimensiones, donde se aprovecha el uso del preprocesador para trabajar con macros y comentarios.

Para detalles de donde se crean los archivos temporarios generados por el preprocesador consulte el *Apéndice B*.

4 Archivo para la definición de los Valores Iniciales del Modelo

Para detallar los valores iniciales que tomará un modelo a simular se utiliza la cláusula *InitialCellValue* en la descripción del mismo, como se comentó en la sección 2.3. Esta cláusula permite especificar el nombre de un archivo que contendrá los valores que serán asignados para algunas o todas las celdas del modelo antes de dar comienzo a la simulación. El formato de dicho archivo se muestra en la Figura 27.

```
(x0,x1,...,xn) = value_1
...        ...       ...
(y0,y1,...,yn) = value_m
```

Figura 27 – Formato del Archivo para la Definición de los valores iniciales del modelo celular

Este archivo debe consistir de una serie de líneas, donde cada una contenga la expresión:

tupla = valor_real

Por convención, se utiliza la extensión **.VAL** en el nombre de este tipo de archivos.

La dimensión de la tupla debe coincidir con la definida para el modelo en cuestión y las mismas deben estar contenidas en el espacio especificado por dicha dimensión.

Para la definición de los valores iniciales de un modelo celular debe utilizarse un solo archivo, y cada archivo no podrá contener los valores iniciales de dos o más modelos.

No es necesario que se definan valores para todas las celdas del modelo. Aquellas celdas que no tengan valor asociado dentro del archivo serán inicializadas con el valor establecido por la cláusula **Initialvalue**.

La interpretación de las líneas del archivo se realiza en orden secuencial, por lo que si se define un valor para una celda y posteriormente se establece un nuevo valor para la misma, el valor asignado será el más reciente.

Ejemplo: En la Figura 28 se observa un archivo que describe los valores iniciales de algunas celdas de un modelo de 4 dimensiones.

```
(0,0,0,0) = ?
(1,0,0,0) = 25
(0,0,1,0) = -21
(0,1,2,2) = 28
(1, 4, 1,2) = 17
(1, 3, 2,1) = 15.44
(0,2,1,1) = -11.5
(1,1,1,1) = 12.33
(1,4,1,0) = 33
(1,4,0,1) = 0.14
```

Figura 28 – Ejemplo de un archivo para la definición de valores iniciales para un Modelo Celular

5 Archivo de Mapa de Valores Iniciales del Modelo

Para indicar los valores iniciales que tomará un modelo es posible el uso la cláusula *InitialMapValue*, como se comentó en la sección 2.3. Esta cláusula permite especificar el nombre de un archivo que contendrá un mapa de valores que será asignado a las celdas del modelo antes de dar comienzo a la simulación. El formato de dicho archivo consta de una serie líneas, donde cada una contiene un valor real, como se muestra en la Figura 29.

```
value_1
...   ...
value_m
```

Figura 29 – Formato del Archivo de Mapa de Valores de un modelo celular

Cada valor del mapa definido será asignado a una celda del modelo según el orden que se muestra en el siguiente ejemplo:

Supóngase que se dispone de un modelo celular tridimensional de tamaño (2, 3, 2). Entonces, el primer valor del mapa será asignado a la celda (0, 0, 0), el segundo valor a la celda (0, 0, 1), el tercero a la celda (0, 1, 0), el cuarto a la celda (0, 1, 1), y así sucesivamente hasta que todas las celdas del modelo tengan asignado un valor.

Si el archivo que contiene el mapa de valores no dispone de suficientes datos para ser asignados a todas las celdas del modelo, se producirá un error y la simulación será abortada. Por el contrario, si un mapa contiene más valores de lo necesario, se asignarán los valores iniciales hasta cubrir los requerimientos del modelo, y el resto será ignorado.

Por convención, se utiliza la extensión **.MAP** en el nombre de este tipo de archivos.

Mediante la herramienta *ToMap* (ver sección 13) es posible la conversión de un archivo que contiene la descripción de una lista de valores según el formato descripto en la sección 4 a un archivo de Mapa de Valores.

6 Formato del Archivo para la definición de Eventos Externos

Los eventos externos se definen en forma separada a la descripción de los modelos. El archivo consiste de una secuencia de líneas, donde cada línea describe un evento con el siguiente formato:

HH:MM:SS:MS PUERTO VALOR

donde:

HH:MM:SS:MS	indica la hora en que ocurrirá el evento.
Puerto	indica el nombre del puerto por el cual se lanzará el evento.
Valor	Valor numérico asociado al evento. Puede ser un número real ó el valor indefinido (**?**).

Ejemplo:

```
00:00:10:00 in 1
00:00:15:00 done 1.5
00:00:30:00 in .271
00:00:31:00 in -4.5
00:00:33:10 inPort ?
```

Figura 30 – Ejemplo de un archivo para la definición de Eventos Externos

7 Formato de la Salida de Eventos

La salida generada por el simulador tiene el formato semejante al archivo de definición de los eventos externos:

HH:MM:SS:MS PUERTO VALOR

Ejemplo:

```
00:00:01:00 out 0.000
00:00:02:00 out 1.000
00:00:03:50 outPort ?
00:00:07:31 outPort 5.143
```

Figura 31 – Ejemplo de un archivo de Salida

8 Formato del Archivo de Log

El archivo de *log* registra el flujo de mensajes entre los modelos que participan en la simulación. Cada línea del archivo muestra el tipo de mensaje, la hora a la que se produjo, quien lo emitió y el destinatario. Esta información es común a todos los mensajes. En caso de ser un mensaje de tipo ***X*** ó ***Y*** aparecerá, además, el puerto y el valor. Para los mensajes de tipo ***D*** se agrega la hora del próximo evento, ó '…' en caso de que la hora sea infinito.
Los números que figuran junto al nombre del simulador asociado a cada modelo son a solo efecto de información para el desarrollador.

Ejemplo:

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para life(02)
Mensaje I / 00:00:00:000 / life(02) para life(0,0,0)(03)
Mensaje I / 00:00:00:000 / life(02) para life(0,0,1)(04)
Mensaje D / 00:00:00:000 / life(0,0,0)(03) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,1)(04) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,2)(05) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,1,0)(06) / ... para life(02)
Mensaje * / 00:00:00:100 / Root(00) para top(01)
Mensaje * / 00:00:00:100 / top(01) para life(02)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,0)(03)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,1)(04)
Mensaje Y / 00:00:00:100 / life(0,0,0)(03) / out /      0.000 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,0)(03) / ... para life(02)
Mensaje Y / 00:00:00:100 / life(0,0,1)(04) / out /     10.500 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,1)(04) / ... para life(02)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,0,0)(03)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,1,0)(06)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,2,0)(09)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,9,0)(30)
```

Figura 32 – Fragmento de un archivo de log

9 Salida generada al activarse el modo de Debug del Parser

Cuando se invoca al simulador con la opción **–p** se activa el modo de debug del parser, en el cual se muestra información adicional durante la interpretación de las reglas que definen el comportamiento de los modelos celulares. La salida generada constará de una secuencia de caracteres mostrando el contenido del buffer, donde se ubican las reglas que serán procesadas por el parser, y a continuación una descripción detallada de cada token que es identificado dentro del buffer. De esta forma, si se produce un error gramatical en la escritura de una regla, mediante el modo de debug es posible identificar en que ubicación se produjo ese error, ya que la salida mostrará todos los tokens interpretados correctamente y la primer aparición de un valor desconocido o erróneo será informada.

En la Figura 33 se muestra la salida generada cuando se encuentra activo el modo de debug del parser para el *Juego de la Vida* implementado en la sección 16.1.

```
********* BUFFER ********
 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) } 1 100 { (0,0) = 0 and
truecount = 3 } 0 100 { t } 0 100 { t }
Number 1 analyzed
Number 100 analyzed
Number 0 analyzed
Number 0 analyzed
OP_REL parsed (=)
Number 1 analyzed
AND parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 3 analyzed
OR parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 4 analyzed
Number 1 analyzed
Number 100 analyzed
Number 0 analyzed
Number 0 analyzed
OP_REL parsed (=)
```

```
Number 0 analyzed
AND parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 3 analyzed
Number 0 analyzed
Number 100 analyzed
BOOL parsed (t)
Number 0 analyzed
Number 100 analyzed
BOOL parsed (t)
```

Figura 33 – Salida generada por el modo de debug del parser para el *Juego de la Vida*

10 Salida del modo de Debug para la Evaluación de las Reglas

Mediante el parámetro **–v** en la invocación al simulador, es posible activar el modo de debug para la evaluación de las reglas de un modelo celular. De esta forma, toda regla al ser evaluada mostrará paso a paso los resultados de las evaluaciones de las funciones y operadores que la componen.

En la Figura 34 se muestra un fragmento de la salida generada para el Juego de la Vida implementado en la sección 16.1 cuando se encuentra activado el modo de debug para la evaluación de las reglas. Los números que se muestran al comienzo de cada línea no son generados por la salida, sino que han sido agregados especialmente para poder hacer referencia a determinadas partes del texto.

La salida comienza con una línea divisoria y una leyenda diciendo "New Evaluation" (líneas 0 y 1), indicando que una nueva celda ejecutará la función de transición. A continuación se muestra en detalle la evaluación de cada regla hasta que alguna de ellas sea válida.

En la línea 2 comienza la evaluación de la primer regla para la celda. Aquí puede observarse que el valor de la celda (0,0) es 0. En la línea 3 se obtiene la constante 1, la que posteriormente es comparada, en la línea 4, contra el valor obtenido en la línea 2. El rótulo "BinaryOp" indica que se está evaluando una función binaria, que en este caso recibe como parámetros los valores 0 y 1, y el nombre de dicha función esta indicado entre paréntesis, en este caso se utiliza la comparación (=). Luego del nombre de la función se encuentra el resultado de la evaluación, en este caso 0 (indicando que la comparación dio como resultado falso). En la salida generada, los valores True son representados con un 1 y los False con un 0.
En la línea 5 se utiliza la operación *CountNode* con parámetro 1 y su evaluación se compara con la constante 3 en la línea 7.
En la línea 11 se evalúa la operación OR entre los valores 0 y 0 (es decir Falso y Falso). Su resultado es Falso.
En la línea 13 se indica el resultado final para la condición de la regla, que en este caso es falsa. Debido a esto se toma la siguiente regla (a partir de la línea 15) y se procede a evaluarla. Finalmente, en la línea 24 se obtiene la ultima regla y la evaluación de esta es válida. Por lo tanto se evalúan la demora de esta regla (en la línea 27) que en este caso solo esta compuesta por la constante 100, pero que podría ser una expresión más compleja con lo que se mostraría con detalle su evaluación. Posteriormente, en la línea 28, se calcula el valor que obtendrá la celda, en este caso la constante 0, pero análogamente a la demora de la regla, esta puede ser una expresión de mayor complejidad.
Los puntos suspensivos de las líneas 30 a 33 no son generados por la salida, sino que han sido agregados para indicar que existen otras evaluaciones.

```
00 +--------------------------------------------------------------------------+
01 New Evaluation:
02 Evaluate: Cell Reference(0,0) = 0
03 Evaluate: Constant = 1
04 Evaluate: BinaryOp(0, 1) =  (=) 0
05 Evaluate: CountNode(1) = 1
06 Evaluate: Constant = 3
07 Evaluate: BinaryOp(1, 3) =  (=) 0
```

```
08 Evaluate: CountNode(1) = 1
09 Evaluate: Constant = 4
10 Evaluate: BinaryOp(1, 4) =  (=) 0
11 Evaluate: BinaryOp(0, 0) =  (or) 0
12 Evaluate: BinaryOp(0, 0) =  (and) 0
13 Evaluate: Rule  = False
14
15 Evaluate: Cell Reference(0,0) = 0
16 Evaluate: Constant = 0
17 Evaluate: BinaryOp(0, 0) =  (=) 1
18 Evaluate: CountNode(1) = 1
19 Evaluate: Constant = 3
20 Evaluate: BinaryOp(1, 3) =  (=) 0
21 Evaluate: BinaryOp(1, 0) =  (and) 0
22 Evaluate: Rule  = False
23
24 Evaluate: Constant = 1
25 Evaluate: Rule  = True
26
27 Evaluate: Constant = 100
28 Evaluate: Constant = 0
29 +-------------------------------------------------------------------------+
30 ...
31 ...
32 ...
33 ...
34 +-------------------------------------------------------------------------+
35 New Evaluation:
36 Evaluate: Cell Reference(0,0) = 1
37 Evaluate: Constant = 1
38 Evaluate: BinaryOp(1, 1) =  (=) 1
39 Evaluate: CountNode(1) = 4
40 Evaluate: Constant = 3
41 Evaluate: BinaryOp(4, 3) =  (=) 0
42 Evaluate: CountNode(1) = 4
43 Evaluate: Constant = 4
44 Evaluate: BinaryOp(4, 4) =  (=) 1
45 Evaluate: BinaryOp(0, 1) =  (or) 1
46 Evaluate: BinaryOp(1, 1) =  (and) 1
47 Evaluate: Rule  = True
48
49 Evaluate: Constant = 100
50 Evaluate: Constant = 1
51 +-------------------------------------------------------------------------+
52 ...
53 ...
54 ...
55 ...
```

Figura 34 – Fragmento de la salida generada por el modo de debug para la Evaluación de Reglas

11 Representación de los Resultados – *DrawLog*

Mediante la herramienta *DrawLog* es posible representar gráficamente la actividad del simulador en cada instante de tiempo para los modelos celulares, utilizando para ello los datos registrados en el archivo de *log*. Los parámetros posibles son:

–h: muestra la siguiente ayuda:

```
drawlog –[?hmtclwp0]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containing the model (.ma)
        t       Initial time
        c       Specify the coupled model to draw
        l       Log file containing the output generated by SIMU
        w       Width (in characters) used to represent numeric values
        p       Precision used to represent numeric values (in characters)
        0       Don't print the zero value
```

Figura 35 – Ayuda del *DrawLog* para la línea de comandos

–?: análogo a –**h**.

–**m**: Especifica el nombre del archivo del cual se cargara el modelo a representar. Este parámetro es obligatorio.

–**t**: Hora a partir de la cual se desean graficar los datos. Si no se define se comenzará a mostrar desde el tiempo de simulación 00:00:00:000.

–**c**: Nombre del modelo celular que se desea representar. Este parámetro es obligatorio debido a que no siempre alcanza con definir mediante **–m** el archivo que contiene la descripción de los modelos, ya que el mismo podría llegar a albergar a más de un modelo celular.

–**l**: Nombre de archivo de *log*, el cual tiene registrado la actividad del simulador. Si se omite este parámetro se tomarán los datos de la entrada estándar.

–**w**: Permite definir el ancho (*Width*), en caracteres, de los valores numéricos que se mostraran en la representación. Este valor debe contemplar todos los dígitos del número, más el punto y el signo del mismo (en caso de que este último sea negativo). Por ejemplo, –**w7** define un tamaño fijo para cada valor de 7 posiciones, y en caso de que estos valores no cubran dicho espacio su representación será completada con espacios en blanco.
Si no se especifica un valor para este parámetro se asume por defecto que el ancho para la representación de valores numéricos es de 10 caracteres.
Para una correcta representación se recomienda usar un ancho que sea mayor o igual a la precisión (definida con el parámetro **–p**) + 3.

–**p**: Permite definir la precisión, en caracteres, de los valores numéricos que se mostraran en la representación. Si se define **–p0** entonces todos los valores reales serán truncados a valores enteros y no se mostraran dígitos decimales en su representación. Generalmente es usada en combinación con la opción **–w**. Por ejemplo: –**w6 –p2** define que todos los valores a mostrar tengan 6 posiciones, de las cuales 2 serán decimales, una será para el punto decimal, y las 3 posiciones restantes serán usadas para la parte entera del valor (incluyendo el signo en caso de que dicho valor sea negativo).
Si no se especifica un valor para este parámetro se asume por defecto que la precisión de los valores numéricos a ser representados es de 3 caracteres.

–**0**: Mediante esta opción los números cuyos valores sean 0 no serán mostrados en la representación, y en su lugar se mostrarán espacios en blanco. Esto puede resultar de utilidad para poder apreciar en una forma más adecuada ciertos modelos donde gran parte de sus celdas tiene valor 0 y sus contenidos no cambian frecuentemente.
Si este parámetro no es usado en la invocación del *DrawLog*, entonces por defecto todos los valores 0 serán mostrados según el ancho y la precisión establecidos.

Ejemplo:

```
drawlog –mlife.ma –clife –llife.log –w7 –p2 –0

ó

simu –mlife.ma –l- | drawlog –mlife.ma –clife -w7 –p2 -0
```

Figura 36 – Ejemplo de llamadas al *DrawLog*

Nota: Si se ejecuta el *N-CD++* para un modelo celular achatado (*Flat*) el *Drawlog* no será de utilidad en este caso, debido a que el intercambio de mensajes dentro del modelo acoplado achatado no será registrado en el archivo de *log*. Para este caso se debe activar el modo de debug para modelos achatados mediante el parámetro **–f** del *N-CD++*.

DrawLog tiene tres modos para representar los resultados en cada instante de tiempo para los modelos celulares dependiendo de su dimensión:

- Salida para modelos celulares bidimensionales.
- Salida para modelos celulares tridimensionales.
- Salida para modelos celulares de 4 ó más dimensiones.

11.1 Representación del DrawLog para modelos Bidimensionales

Cuando el modelo a ser representado tiene dimensión 2, *DrawLog* generará una representación que consiste en un esquema para el estado del autómata en cada instante del tiempo simulado.

En la Figura 37 se muestra un fragmento de la salida generada por el *DrawLog* para un modelo bidimensional de dimensión (10, 10), donde se han utilizado los parámetros **–w5 –p1** para formatear los valores numéricos.

```
Line : 238 - Time: 00:00:00:000
         0    1    2    3    4    5    6    7    8    9
    +--------------------------------------------------+
   0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   2| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   5| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
    +--------------------------------------------------+

Line : 358 - Time: 00:00:01:000
         0    1    2    3    4    5    6    7    8    9
    +--------------------------------------------------+
   0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   2| 24.0 24.0 35.8 24.0 24.0 24.0 24.0 24.0 -6.3 24.0|
   3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   5| 24.0 24.0 24.0 24.0 24.0 39.5 24.0 24.0 24.0 24.0|
   6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 -4.0 24.0|
   9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
    +--------------------------------------------------+
```

Figura 37 – Fragmento de la salida generada por el *DrawLog* para un modelo bidimensional

11.2 Representación del DrawLog para modelos Tridimensionales

Cuando el modelo a ser representado tiene dimensión 3, *DrawLog* generará una representación que consiste en una serie de esquemas para el estado del autómata en cada instante del tiempo simulado. El primer esquema representa el estado para todas las celdas de la forma (x, y, 0), el segundo representa el estado para las celdas de la forma (x, y, 1), y así hasta que se muestren todos los slices del modelo.

En la Figura 38 se muestra un fragmento de la salida generada por el *DrawLog* para un modelo tridimensional de dimensión (5, 5, 4), donde se han utilizado los parámetros **–w1 –p0 –0** para formatear los valores numéricos. Para cada instante de tiempo se muestran 4 gráficos correspondientes a los slices (x, y, 0), (x, y, 1), (x, y, 2) y (x, y, 3) respectivamente, donde $0 \leq x, y \leq 4$.

```
Line : 247 - Time: 00:00:00:000
     01234      01234      01234      01234
    +-----+    +-----+    +-----+    +-----+
   0|1    |   0|     |   0|1    |   0|     |
   1|1 1  |   1|11  1|   1|  111|   1|   11|
   2| 1   |   2|   11|   2| 1 11|   2|    1|
   3|     |   3|  1  |   3|    1|   3|    1|
   4|  1 1|   4|  1 1|   4| 1  1|   4|    1|
    +-----+    +-----+    +-----+    +-----+

Line : 557 - Time: 00:00:00:100
     01234      01234      01234      01234
    +-----+    +-----+    +-----+    +-----+
   0|     |   0|11 11|   0|1  11|   0|   11|
   1|     |   1|     |   1|1    |   1|    1|
   2|     |   2|1   1|   2|1    |   2|   11|
   3|   1 |   3|  11 |   3|1  11|   3|1   1|
   4|     |   4|     |   4|     |   4|     |
    +-----+    +-----+    +-----+    +-----+

Line : 829 - Time: 00:00:00:200
     01234      01234      01234      01234
    +-----+    +-----+    +-----+    +-----+
   0|     |   0|     |   0|1   1|   0|     |
   1|    1|   1|    1|   1|   11|   1|    1|
   2|     |   2|     |   2|1   1|   2|     |
   3|     |   3|     |   3|1  1 |   3|     |
   4|     |   4|    1|   4|1  11|   4|    1|
    +-----+    +-----+    +-----+    +-----+
```

Figura 38 – Fragmento de la salida generada por el *DrawLog* para un modelo tridimensional

11.3 Representación del DrawLog para modelos de 4 ó más dimensiones

Cuando los modelos a ser representados tienen 4 ó más dimensiones, *DrawLog* generará una representación que consiste en un listado detallado de la referencia de la celda y su respectivo valor para cada instante del tiempo simulado. Para este modo los parámetros **–w**, **–p** y **–0** del *DrawLog* no cumplen ninguna funcionalidad.

En la Figura 39 se muestra un fragmento de la salida generada por el *DrawLog* para un modelo de dimensión 4, con tamaño (2, 10, 3, 4).

```
Line : 506 - Time: 00:00:00:000
  (0,0,0,0) = ?
  (0,0,0,1) = 0
  (0,0,0,2) = 9
```

```
  (0,0,0,3) = 0
  (0,0,1,0) = 21
  ...    ...    ...
  ...    ...    ...
  (1,9,1,0) = 0
  (1,9,1,1) = 4.333
  (1,9,1,2) = 0
  (1,9,1,3) = -2
  (1,9,2,0) = 6
  (1,9,2,1) = 0
  (1,9,2,2) = 7
  (1,9,2,3) = 0


Line : 789 - Time: 00:00:00:100
  (0,0,0,0) = 0
  (0,0,0,1) = 0
  (0,0,0,2) = 13.33
  (0,0,0,3) = 0
  (0,0,1,0) = 5.75
  ...    ...    ...
  ...    ...    ...
  (1,9,1,0) = 6.165
  (1,9,1,1) = 2
  (1,9,1,2) = 0
  (1,9,1,3) = 1.14
  (1,9,2,0) = 0
  (1,9,2,1) = 0
  (1,9,2,2) = 5.25
  (1,9,2,3) = 0
```

Figura 39 – Fragmento de la salida generada por el *DrawLog* para un modelo de dimensión 4

12 Generación Aleatoria de Valores Iniciales – *MakeRand*

Mediante la herramienta *MakeRand* es posible crear estados iniciales aleatorios, los cuales pueden ser usados para simulaciones de distintos modelos. Los parámetros posibles son:

–h: muestra la siguiente ayuda:

```
makerand -[?hmcir]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containig the model (.ma)
        c       Specify the Cell model within the .ma file
        i       Generate integer numbers randomly. The format is:
                        -iquantity[+|-]minNumber[+|-]maxNumber
                For example:    -i100-10+5      will sets 100 cells with
                integer values in the interval [-10,5] with uniform distribution

        r       Generate real numbers randomly. The format is:
                        -rquantity[+|-]minNumber[+|-]maxNumber
                and its behaviour is similar to the -i parameter.
```

Figura 40 – Ayuda del *MakeRand* para la línea de comandos

–?: análogo a –**h**.

–**m**: Especifica el nombre del archivo que contiene la definición del modelo para el cual se creará el estado inicial. Este parámetro es obligatorio.

–c: Nombre del modelo celular. Este parámetro es obligatorio y será fundamental para conocer la dimensión del modelo para el cual se creará el estado inicial.

–i: Especifica la cantidad de celdas a las cuales se les establecerá un valor aleatorio entero perteneciente al rango indicado. Por ejemplo, el parámetro **–i100-10+5** indica que 100 celdas serán elegidas aleatoriamente, y su valor será un número entero seleccionado al azar con distribución uniforme perteneciente al intervalo [-10, 5]. El resto de las celdas contendrá el valor establecido por defecto en la definición del modelo.
Si la cantidad de celdas especificadas es mayor a la cantidad de celdas del modelo, entonces se producirá un error, se avisará al usuario de tal situación, y no se generará ningún estado inicial.

–r: Este parámetro es análogo a **–i**, pero indica que los valores generados serán números reales.

Los datos creados siempre serán almacenados en un archivo con el formato definido en la sección 4 y el nombre del mismo será generado a partir del nombre del archivo que contiene la descripción del modelo (indicado por el parámetro **–m**) pero con la extensión .VAL.

13 Conversión de Archivos de Valores a Mapa de Valores – *ToMap*

N-CD++ permite definir el estado inicial para los modelos celulares mediante el uso de archivos que describen los valores para las celdas que los componen. Existen dos formatos distintos de archivos aceptados por la herramienta. Los archivos con formato .VAL contienen una lista de sentencias con la forma *CELDA = VALOR*, y permiten definir el estado inicial para algunas o todas las celdas del modelo. Por otra parte, los archivos con formato .MAP tienen una secuencia de valores, los cuales son asignados en orden a cada una de las celdas del modelo. Este último tipo de archivo define el estado inicial para todas las celdas del autómata celular.

La herramienta *ToMap* permite la conversión de archivos con formato .VAL al formato .MAP, respetando los valores iniciales que se definan para las celdas. Como el archivo de entrada puede no contener un valor para todas las celdas del modelo, para las celdas que no estén definidas se les asigna el valor especificado por la cláusula *InitialValue* incluida en la definición del modelo celular. El archivo generado tendrá el mismo nombre que el archivo de entrada y su extensión será .MAP.

Los parámetros aceptados por la herramienta son:

–h: muestra la siguiente ayuda:

```
toMap -[?hmci]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containig the model (.ma)
        c       Specify the Cell model within the .ma file
        i       Specify the input .VAL file
```

Figura 41 – Ayuda del *ToMap* para la línea de comandos.

–?: análogo a –**h**.

–**m**: Especifica el nombre del archivo que contiene la definición del modelo celular. Este parámetro es obligatorio.

–c: Nombre del modelo celular. Este parámetro es obligatorio y será fundamental para conocer la dimensión del modelo para poder generar el archivo con el mapa de valores.

–i: Especifica el nombre del archivo .VAL que contiene la lista celdas con sus respectivos valores iniciales.

14 Conversión de Archivos de Valores para uso en CD++ – *ToCDPP*

La herramienta *ToCDPP* toma como entrada un archivo conteniendo la definición de los modelos, y un archivo conteniendo el estado inicial para un modelo celular con formato .VAL, y genera un nuevo archivo resultante de la combinación de ambos. El archivo generado es análogo al archivo que define los modelos, indicado como entrada, pero se reemplaza la cláusula *InitialCellsValue,* que hace referencia al archivo que contiene el estado inicial del modelo, por una secuencia de cláusulas *InitialRowValue* conteniendo los valores iniciales para las celdas que hayan sido definidas en él.

Esta herramienta resulta de utilidad para poder utilizar en *CD++* modelos simples definidos en *N-CD++*, debido a que la versión original del simulador no utiliza archivos externos para la definición del estado inicial de los modelos celulares. Para ello, el estado inicial del modelo utilizado por *N-CD++* solo debe contener los valores **0**, **1** e **?**, dado que son los únicos valores soportados en *CD++*, y el archivo de especificación de modelos no debe hacer uso de características propias de *N-CD++*.

Los parámetros aceptados por la herramienta son:

–h: muestra la siguiente ayuda:

```
toCDPP -[?hmcio]

where:
        ?       Show this message
        h       Show this message
        m       Specify the input file containig the model (.ma)
        c       Specify the Cell model within the .ma file
        i       Specify the input .VAL file
        o       Specify the output .MA file
```

Figura 42 – Ayuda del *toCDPP* para la línea de comandos.

–?: análogo a –**h**.

–**m**: Especifica el nombre del archivo .MA que contiene la definición de los modelos. Este parámetro es obligatorio.

–c: Nombre del modelo celular. Este parámetro es obligatorio y será fundamental para establecer el modelo al cual se el establecerán los valores iniciales.

–i: Especifica el nombre del archivo .VAL que contiene el estado inicial del modelo celular.

–**o**: Especifica el nombre del archivo .MA que será generado por la herramienta.

15 Incorporación de Nuevos Modelos Atómicos

Esta sección describe el mecanismo para definir e incorporar nuevos modelos atómicos a la herramienta. Sin embargo, dichos modelos no podrán ser usados para crear un modelo acoplado celular, sino para interactuar directamente con otros modelos ó para formar parte un modelo acoplado *DEVS* más general. Este capítulo esta orientado a usuarios con conocimientos de programación en lenguaje *C++* y su contenido puede no ser de utilidad para aquellas personas a las que sólo le interesa usar la herramienta con el fin de crear modelos celulares y/o modelos acoplados que utilicen los ya definidos en *N-CD++*.

Para generar un nuevo modelo atómico, se debe comenzar por diseñar una nueva clase que sea derivada de la clase *Atomic* y se debe agregar al método *MainSimulator.registerNewAtomics*() el nuevo tipo de modelo atómico. Luego se deben sobrecargar obligatoriamente los siguientes métodos:

- ***initFunction***: este método es invocado por el simulador una única vez al comenzar la simulación en el tiempo cero. El objetivo es permitir la inicialización que el modelo considere necesaria. Antes de invocar al método, *sigma* vale infinito y el estado es *pasivo*.

- ***externalFunction***: este método es invocado cuando arriba un evento externo por alguno de los puertos del modelo.

- ***internalFunction***: antes de invocar a este método, *sigma* vale cero, ya que se ha cumplido el intervalo para la transición interna.

- ***outputFunction***: antes de invocar al método *sigma* vale cero, ya que se ha cumplido el intervalo para la transición interna.

- ***className***: nombre de la clase.

Estos métodos pueden invocar ciertas primitivas predefinidas, que permiten interactuar con el simulador abstracto:

- ***holdIn***(estado, tiempo): indica al simulador que el modelo debe mantenerse en el estado durante un tiempo, y que, luego de transcurrido, provocará una transición interna.

- ***passivate***(): indica al simulador que el modelo entra en modo *pasivo* y que únicamente deberá ser reactivado ante la llegada de un evento externo.

- ***sendOutput***(hora, port, valor): envía un mensaje de salida por el puerto indicado.

- ***nextChange***(): este método permite obtener el tiempo restante para su próximo cambio de estado (*sigma*).

- ***lastChange***(): este método permite obtener la hora en que se produjo el último cambio de estado.

- ***state***(): este método obtiene la fase en la que se encuentra el modelo.

- ***getParameter***(nombreModelo, nombreParámetro): este método permite acceder a los parámetros de configuración de la clase.

Para utilizar e inicializar un modelo atómico vea la sección 2.2.

15.1 Ejemplo. Construcción de una Cola

Una cola es un dispositivo de almacenamiento temporal con política *FIFO* (First In First Out). Para implementarlo en *N-CD++* se debe crear una nueva clase (que en este caso llamaremos *Queue*) que extienda a *Atomic*.

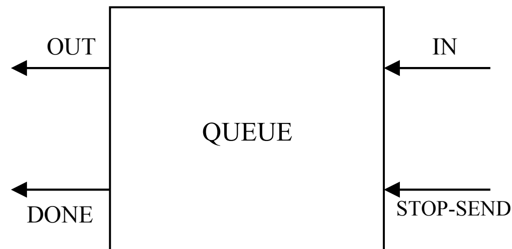


Figura 43 – Esquema de una Cola

Una cola debe poseer un puerto de entrada por el cual el resto de los modelos ingresen los elementos a ser almacenados, y un puerto de salida por donde se envían los mismos cuando llegue el momento adecuado. El tiempo de demora entre la llegada del elemento y su salida (tiempo de preparación para ser enviado) es configurable por medio del archivo de carga del simulador. Para cumplir con estos requerimientos la cola define dos puertos, un puerto *done* que indica la recepción del elemento enviado por el puerto de salida y un puerto regulador de flujo llamado *stop-send*.

El modelo *Queue* sobrecarga los métodos de inicialización, función de transición interna, externa y salida. En la inicialización las variables del modelo toman el valor inicial y se eliminan todos los valores de la cola interna.

```
Model &Queue::initFunction()
{
   this->elements.erase( elements.begin(), elements.end() );
   return *this;
}
```

Figura 44 – Método para la inicialización de la *Cola*

Frente a un evento externo por el puerto de entrada se ingresa el valor en la cola interna y luego se verifica si el estado de la cola permite programarse para realizar un nuevo envío por el puerto de salida. Si el mensaje arribó por el puerto *done* el último elemento enviado puede ser eliminado de la cola interna y prepara el próximo si lo hay. Si el mensaje proviene del puerto *stop* debe analizarse el contenido para interpretar el pedido como "detener" o "reanudar" la expedición de datos. Si se detiene el procesamiento de salida se registra el tiempo restante para finalizar la iteración para tomarlo en cuenta al reanudar las tareas.

```
Model &Queue::externalFunction( const ExternalMessage &msg )
{
   if( msg.port() == in ) {
      elements.push_back( msg.value() );
      if( elements.size() == 1 )
         this->holdIn( active, preparationTime );
   }

   if( msg.port() == done )
   {
      elements.pop_front();
      if( !elements.empty() )
         this->holdIn( active, preparationTime );
   }

   if( msg.port() == stop )
      if( this->state() == active && msg.value() )
      {
         timeLeft = msg.time() - this->lastChange();
         this->passivate();
      }
      else
         if( this->state() == passive && !msg.value() )
```

```
            this->holdIn( active, timeLeft );

   return *this;
}
```

Figura 45 – Método para la definición de la función de Transición Externa de la *Cola*

La función de salida indica que ha finalizado el tiempo de preparación para el primer elemento de la cola y se envía este por el puerto *out*. Luego es ejecutada la función de transición interna indicando que se ha terminado de enviar el valor, por lo tanto el modelo se *pasiva*. El ciclo continuará con el próximo mensaje externo.

```
Model &Queue::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, elements.front() );
   return *this;
}

Model &Queue::internalFunction( const InternalMessage & )
{
   this->passivate();
   return *this;
}
```

Figura 46 – Métodos para las funciones de Salida y de Transición Interna de la *Cola*

16 Apéndice A – Ejemplos de la Definición de Modelos

16.1 Juego de la Vida

En el caso del *Juego de la Vida*, las reglas especifican lo siguiente:

- Una celda activa permanecerá en este estado si tiene dos o tres vecinos activos.
- Una celda inactiva pasará a estado activo si tiene exactamente dos vecinos activos.
- De otra forma la celda pasará a estado inactivo.

La implementación de este modelo en *N-CD++* es la siguiente:

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue :  1  00010001111000000000
initialrowvalue :  2  00110111100010111100
initialrowvalue :  3  00110000011110000010
initialrowvalue :  4  00101111000111100011
initialrowvalue : 10  01111000111100011110
```

```
initialrowvalue : 11  0001000111000000000
localtransition : life-rule

[life-rule]
rule : 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) }
rule : 1 100 { (0,0) = 0 and truecount = 2 }
rule : 0 100 { t }
```

Figura 47 – Implementación del *Juego de la Vida*

16.2 Simulación del Rebote de un Objeto

La siguiente es la especificación de un modelo que representa un objeto en movimiento que rebota al chocar contra los bordes de un ambiente. Este ejemplo es ideal para ilustrar el uso de un autómata celular no toroidal, donde las celdas del borde tienen distinto comportamiento al resto de las celdas.

Para la representación del problema, se utilizan 5 valores distintos para los estados de cada celda, estos son:
0 = representa un casillero vacío.
1 = representa el objeto moviéndose hacia la celda inferior–derecha con respecto a la posición actual.
2 = representa el objeto moviéndose hacia la celda superior–derecha.
3 = representa el objeto moviéndose hacia la celda inferior–izquierda.
4 = representa el objeto moviéndose hacia la celda superior–izquierda.

La siguiente es la especificación del modelo:

```
[top]
components : rebota

[rebota]
type : cell
width : 20
height : 15
delay : transport
defaultDelayTime  : 100
border : nowrapped
neighbors : rebota(-1,-1)             rebota(-1,1)
neighbors :              rebota(0,0)
neighbors : rebota(1,-1)              rebota(1,1)
initialvalue : 0
initialrowvalue : 13     00000000000000000010
localtransition : mover-rule
zone : esquinaUL-rule { (0,0) }
zone : esquinaUR-rule { (0,19) }
zone : esquinaDL-rule { (14,0) }
zone : esquinaDR-rule { (14,19) }
zone : top-rule { (0,1)..(0,18) }
zone : bottom-rule { (14,1)..(14,18) }
zone : left-rule { (1,0)..(13,0) }
zone : right-rule { (1,19)..(13,19) }

[mover-rule]
rule : 1 100 { (-1,-1) = 1 }
rule : 2 100 { (1,-1) = 2 }
rule : 3 100 { (-1,1) = 3 }
rule : 4 100 { (1,1) = 4 }
rule : 0 100 { t }

[top-rule]
rule : 3 100 { (1,1) = 4 }
```

```
rule : 1 100 { (1,-1) = 2 }
rule : 0 100 { t }

[bottom-rule]
rule : 4 100 { (-1,1) = 3 }
rule : 2 100 { (-1,-1) = 1 }
rule : 0 100 { t }

[left-rule]
rule : 1 100 { (-1,1) = 3 }
rule : 2 100 { (1,1) = 4 }
rule : 0 100 { t }

[right-rule]
rule : 3 100 { (-1,-1) = 1 }
rule : 4 100 { (1,-1) = 2 }
rule : 0 100 { t }

[esquinaUL-rule]
rule : 1 100 { (1,1) = 4 }
rule : 0 100 { t }

[esquinaUR-rule]
rule : 3 100 { (1,-1) = 2 }
rule : 0 100 { t }

[esquinaDL-rule]
rule : 2 100 { (-1,1) = 3 }
rule : 0 100 { t }

[esquinaDR-rule]
rule : 4 100 { (-1,-1) = 1 }
rule : 0 100 { t }
```

Figura 48 – Implementación del Modelo de Rebote de un Objeto

16.3 Clasificación de Materias Primas

El objetivo del próximo ejemplo será mostrar el uso comportamiento especial que se le puede dar a una celda cuando arriba un evento externo a través de un puerto de entrada. Se cuenta con un modelo que representa el embalaje y clasificación de cierto tipo de materia prima, que contiene aproximadamente un 30 % de carbono. Además, se dispone de una máquina que ubica fracciones de 100 gramos de esa sustancia en una cinta transportadora. Esta las almacena temporalmente hasta que sean procesadas por una embaladora, que toma dichas fracciones hasta alcanzar el kilogramo de peso, y las envasa. Posteriormente, se clasifica la sustancia ya envasada. Si cada envase contiene 30 ± 1 % de carbono, entonces es catalogado como de primera calidad; sino se lo cataloga como de segunda calidad.

Se cuenta con un modelo atómico *Generator* que genera valores (en este caso siempre el valor 1) cada x segundos (donde x tiene distribución Exponencial con media 3). Estos valores son pasados a la cinta transportadora, representado por un modelo celular, la cual genera una de las fracciones de la sustancia. Otro modelo celular obtiene las fracciones de la sustancia de la cinta transportadora y realizará las tareas de embalaje (agrupamiento de 10 fracciones) y selección.

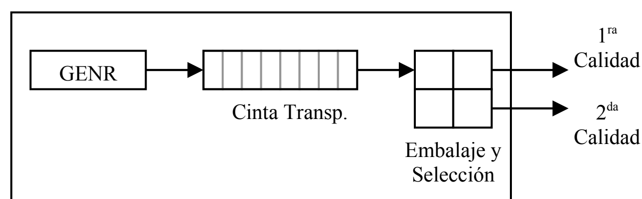


Figura 49 – Esquema de Acoplamiento de la Clasificadora de Materias Primas

La siguiente es la especificación del modelo de clasificación de materias primas:

```
[top]
components : genSustancias@Generator cola embalaje
out : outPrimeraSeleccion outSegundaSeleccion
link : out@genSustancias in@cola
link : out@cola in@embalaje
link : out1@embalaje outPrimeraSeleccion
link : out2@embalaje outSegundaSeleccion

[genSustancias]
distribution : exponential
mean : 3
initial : 1
increment : 0

[cola]
type : cell
width : 6
height : 1
delay : transport
defaultDelayTime  : 1
border : nowrapped
neighbors : cola(0,-1) cola(0,0) cola(0,1)
initialvalue : 0
in : in
out : out
link : in in@cola(0,0)
link : out@cola(0,5) out
localtransition : cola-rule
portInTransition : in@cola(0,0)  establecerSustancia

[cola-rule]
rule : 0           1 { (0,0) != 0 and (0,1) = 0 }
rule : { (0,-1) }  1 { (0,0) = 0 and (0,-1) != 0 and not isUndefined((0,-1)) }
rule : 0        3000 { (0,0) != 0 and isUndefined((0,1)) }
rule : { (0,0) }   1 { t }

[establecerSustancia]
rule : { 30 + normal(0,2) } 1000 { t }

[embalaje]
type : cell
width : 2
height : 2
delay : transport
defaultDelayTime  : 1000
border : nowrapped
neighbors : embalaje(-1,-1) embalaje(-1,0) embalaje(-1,1)
neighbors : embalaje(0,-1)  embalaje(0,0)  embalaje(0,1)
neighbors : embalaje(1,-1)  embalaje(1,0)  embalaje(1,1)
in : in
out : out1 out2
initialvalue : 0
initialrowvalue :  0      00
initialrowvalue :  1      00
link : in in@embalaje(0,0)
link : in in@embalaje(1,0)
```

```
link : out@embalaje(0,1) out1
link : out@embalaje(1,1) out2
localtransition : embalaje-rule
portInTransition : in@embalaje(0,0) acumular-rule
portInTransition : in@embalaje(1,0) incCant-rule

[embalaje-rule]
rule : 0   1000 { isUndefined((1,0)) and isUndefined((0,-1)) and (0,0) = 10 }
rule : 0   1000 { isUndefined((-1,0)) and isUndefined((0,-1)) and (1,0) = 10 }
rule : { (0,-1) / (1,-1) }  1000 { isUndefined((-1,0)) and isUndefined((0,1))
                        and (1,-1) = 10 and abs( (0,-1) / (1,-1) - 30 ) <= 1 }
rule : { (-1,-1) / (0,-1) } 1000 { isUndefined((1,0)) and isUndefined((0,1))
                        and (0,-1) = 10 and abs( (-1,-1) / (0,-1) - 30 ) > 1 }
rule : { (0,0) } 1000 { t }

[acumular-rule]
rule : { portValue(thisPort) + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }                       1000 { t }

[incCant-rule]
rule : { 1 + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }     1000 { t }
```

Figura 50 – Implementación del Sistema de Clasificación de Materias Primas

Dentro de la definición del modelo *cola* que representa a la cinta transportadora puede apreciarse que se da un comportamiento especial para los mensajes externos que ingresen en la celda (0,0) provenientes del generador de sustancias, mediante la cláusula **portInTransition**. También dentro de la definición del modelo *embalaje* se utiliza dicha cláusula para especificar los nombres de las funciones que describen los comportamientos para las celdas (0,0) y (1,0) cuando llega una sustancia proveniente de la cinta transportadora.

16.4 Juego de la Vida en 3D

El siguiente ejemplo es una adaptación del *Juego de la Vida* modelado con un autómata celular de 3 dimensiones. Se han realizado modificaciones sobre las reglas y sobre el vecindario utilizado, el cual consta de un cubo de tamaño 3x3x3 celdas.

En la Figura 51 se muestra la descripción del modelo en el lenguaje provisto por la herramienta, mientras que en la Figura 52 se muestra el archivo "*life.val*" que contiene los valores iniciales para las celdas del autómata.

```
[top]
components : 3d-life

[3d-life]
type : cell
dim : (7,7,3)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : 3d-life(-1,-1,-1) 3d-life(-1,0,-1) 3d-life(-1,1,-1)
neighbors : 3d-life(0,-1,-1)  3d-life(0,0,-1)  3d-life(0,1,-1)
neighbors : 3d-life(1,-1,-1)  3d-life(1,0,-1)  3d-life(1,1,-1)
neighbors : 3d-life(-1,-1,0)  3d-life(-1,0,0)  3d-life(-1,1,0)
neighbors : 3d-life(0,-1,0)   3d-life(0,0,0)   3d-life(0,1,0)
neighbors : 3d-life(1,-1,0)   3d-life(1,0,0)   3d-life(1,1,0)
neighbors : 3d-life(-1,-1,1)  3d-life(-1,0,1)  3d-life(-1,1,1)
neighbors : 3d-life(0,-1,1)   3d-life(0,0,1)   3d-life(0,1,1)
neighbors : 3d-life(1,-1,1)   3d-life(1,0,1)   3d-life(1,1,1)
initialvalue : 0
initialCellsValue : 3d-life.val
```

```
localtransition : 3d-life-rule

[3d-life-rule]
rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
rule : 0 100 { t }
```

Figura 51 – Implementación del *Juego de la Vida* en 3D

```
(0,0,0) = 1
(0,0,2) = 1
(1,0,0) = 1
(1,0,1) = 1
(1,1,1) = 1
(1,2,0) = 1
(1,2,2) = 1
(1,3,2) = 1
(1,4,2) = 1
(1,5,0) = 1
(1,5,1) = 1
(1,6,0) = 1
(1,6,1) = 1
(2,1,2) = 1
(2,1,0) = 1
(2,3,1) = 1
(2,3,2) = 1
```

```
(2,4,1) = 1
(2,4,2) = 1
(2,5,0) = 1
(2,6,1) = 1
(3,2,1) = 1
(3,5,1) = 1
(3,5,2) = 1
(3,6,1) = 1
(3,6,2) = 1
(4,1,2) = 1
(4,2,0) = 1
(4,2,1) = 1
(4,4,1) = 1
(4,5,0) = 1
(4,5,2) = 1
(4,6,0) = 1
(4,6,2) = 1
```

```
(5,1,2) = 1
(5,2,0) = 1
(5,2,2) = 1
(5,3,0) = 1
(5,3,1) = 1
(5,5,1) = 1
(5,5,2) = 1
(5,6,0) = 1
(6,0,0) = 1
(6,1,1) = 1
(6,1,2) = 1
(6,3,0) = 1
(6,3,2) = 1
(6,4,2) = 1
(6,5,1) = 1
(6,6,0) = 1
(6,6,2) = 1
```

Figura 52 – Archivo *life.val* conteniendo los valores iniciales para el *Juego de la Vida* en 3D

16.5 Uso de Macros

El siguiente ejemplo muestra el uso de macros para el modelado d una versión del *Juego de la Vida* en 4 dimensiones.

En la Figura 55 se muestra el contenido del archivo *LIFE.INC*. Dicho archivo contiene la definición de una de las macros usadas en esta variante del *Juego de la Vida.* Este tipo de archivo puede contener la definición de varias macros. Como puede verse, es posible la inclusión de comentarios con solo escribir un texto fuera de la definición de la macro. Todo el texto no contenido entre las directivas *#BeginMacro* y *#EndMacro* es ignorado.

```
#include(life.inc)
#include(life-1.inc)

[top]
components : life

[life]
type : cell
dim : (2,10,3,4)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors :                life(-1,-1,0,0) life(-1,0,0,0) life(-1,1,0,0)
neighbors : life(0,-8,0,0) life(0,-1,0,0)  life(0,0,0,0)  life(0,1,0,0)
neighbors :                life(1,-1,0,0)  life(1,0,0,0)  life(1,1,0,0)
initialvalue : 0
initialCellsValue :  life.val
localtransition : life-rule
```

```
[life-rule]
% Comentario: Aquí comienza la definición de las reglas
rule : 1                 100 { #macro(HaceCalor) or #macro(Llueve) }
rule : 0                 100 { (0,0,0,0) = ? OR (0,0,0,0) = 2 }
#macro(regla1)   % Otro comentario: Invocación de una macro
rule : 1                 100 { (0,0,0,0) = (1,0,0,0) AND (0,0,0,0) > 1 }
#macro(regla2)
```

Figura 53 – Implementación del *Juego de la Vida* en 4D con el uso de Macros

```
(0,0,0,0) = ?
(1,0,0,0) = 25
(0,0,1,0) = 21
(0,1,2,2) = 28
(1, 4, 1,2) = 17
(1, 3, 2,1) = 15.44
```

Figura 54 – Archivo *life.val* conteniendo los valores iniciales para el Juego de la Vida en 4D

```
Esto es un comentario: La macro Regla3 asigna el valor 0 si la celda contiene el
valor 3 y el valor 4 cuando el estado de la celda es negativo.

#BeginMacro(Regla3)
rule : 0 100 { (0,0,0,0) = 3 }
rule : 4 100 { (0,0,0,0) < 0 }
#EndMacro


#BeginMacro(Regla1)
rule : 0 100 { (0,0,0,0) + (1,0,0,0) + (1,1,0,0) + (0,-8,0,0) = 11 }
#EndMacro

#BeginMacro(HaceCalor)
(0,0,0,0) > 30
#EndMacro
```

Figura 55 – Archivo *life.inc* conteniendo algunas macros usadas en el *Juego de la Vida* – 4D

```
#BeginMacro(Regla2)
rule : 0 100 { (0,0,0,0) = 7 }
rule : { (0,0,0,0) + 2 } 100 { t }
#EndMacro

#BeginMacro(Llueve)
(0,-8,0,0) > 25
#EndMacro
```

Figura 56 – Archivo *life–1.inc* conteniendo el resto de las macros usadas en el *Juego de la Vida* – 4D

17 Apéndice B – El Preprocesador y los Archivos Temporarios

Al usar el preprocesador se generará automáticamente un archivo temporario, que contendrá la definición de los modelos donde previamente se reemplazan todas las invocaciones a macros por el contenido de las mismas (si es que

existen), y se eliminan todos los comentarios. Este archivo temporario es pasado al simulador para su interpretación. Debido a esto, si el archivo que contiene la definición de modelos incluye invocaciones a macros o comentarios, y en la invocación del simulador se usa el parámetro **–b** para ignorar al preprocesador, el simulador usará directamente el archivo que contiene este código sin que se hayan realizado las macro–expansiones y con comentarios, lo que generará una incorrecta interpretación de los modelos.

El nombre del archivo temporario se corresponde con el valor devuelto por la instrucción *tmpnam* del *GCC*. Para la selección del directorio donde se ubicaran los archivos temporarios se utiliza la siguiente política:

1. Al compilarse *N–CD++*, se incluye dentro del código ejecutable una referencia al directorio establecido por la variable *P_tmpdir* ubicada en <*stdio.h*>. Si este directorio no es el directorio raíz, el mismo será usado para almacenar el archivo temporario.
   En *Linux* esta variable tiene usualmente el valor: "**/TMP**", mientras que en la versión del *GCC* 2.8.1 para *Windows–32 bits*, esta variable referencia al directorio raíz de la unidad de disco que se esta usando.
2. En caso de que en el paso anterior se obtenga una referencia al directorio raíz, se procede a leer el contenido de la variable de entorno **TEMP**. Si esta variable esta definida, su valor será considerado como el directorio a utilizar para almacenar los archivos temporarios.
   Si la variable de entorno **TEMP** no se encuentra definida, se recurre a consultar la variable de entorno **TMP**. Si esta variable esta definida, su valor será considerado como el directorio a utilizar para almacenar los archivos temporarios.
3. Si la variable de entorno **TMP** tampoco se encuentra definida, se usará el directorio actual donde se encuentra el archivo ejecutable del simulador.

Referencias

[BB98] BARYLKO, A.; BEYOGLONIAN, J.; "*CD++. Una herramienta de implementación de modelos Cell–DEVS binarios*". Tesis de Licenciatura. FCEyN. UBA. Argentina, 1998.

[BBW98a] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "*GAD: A General Application Tool for DEVS Modeling and Simulation".* En Proceedings of IASTED International Conference on Applied Modelling and Simulation, Hawai, USA, 1998.

[BBW98b] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "*CD++: Una Herramienta de Implementación de Modelos Cell-DEVS Binarios*". Memorias de la XXIV Conferencia Latinoamericana de Informática. Quito, Ecuador, 1998.

[ChZ94] CHOW, A.; ZEIGLER, B. "*Revised DEVS: a parallel, hierarchical, modular modeling formalism*". Proceedings of the SCS Winter Simulation Conference. 1994.

[EDPWJ89] EBLING, M.; DI LORETO, J.; PRESLEY, B.; WIELAND, J. JEFFERSON, D. "*An ant foraging model implemented on the time warp operating system*". Distributed Simulation 1989. pp. 21-30. 1989.

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

[Gia96] GIAMBIASI, N. "*Introduction à la modélisation et à la simulation*". Material del curso de D.E.A.; Univerité d'Aix-Marseille III. 1996.

[GLNW94] GREENBERG, A.; LUBACHEVSKY, B.; NICOL, D.; WRIGHT, P. "*Efficient Massively Parallel Simulation of Dynamic Channel Assignment Schemes for Wireless Cellular Comunications*". Proceedings of the 8th. Workshop on Parallel and Distributed Simulation. pp. 187-194. 1994.

[GM76] GIAMBIASI, N.; MIARA, A. "*SILOG: A practical tool for digital logic circuit simulation*". In *Proceedings of the 16th. D.A.C.*, San Diego, U.S.A. 1976.

[RW99a] RODRÍGUEZ, D.; WAINER G. "*Redefinition of a specification language for Cell-DEVS models*". In Proceedings of Information Systems Analysis and Synthesis, ISAS'99. Orlando, USA. Agosto de 1999.

[RW99b] RODRIGUEZ, D.; WAINER. G. "*Aplicación del formalismo Cell-DEVS usando la herramienta N-CD++*". Conferencia Latinoamericana de Informática – CLEI'99. Asunción, Paraguay. 1999.

[RW99c] RODRIGUEZ, D.; WAINER. G. "*New Extensions to the CD++ tool*". Summer Computer Conference (SCSC'99). Chicago, Illinois, EE.UU. Julio de 1999.

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

[Wai98] WAINER, G. "*Contribución a la Modelización y Simulación de Sistemas de Eventos Discretos*". Tesis de Doctorado. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina; y Universite' d'Aix-Marseille III. Francia. 1998.

[WFG97] WAINER, G. A.; FRYDMAN, C.; GIAMBIASI, N., "*An environment for simulation of cellular DEVS models*". En Proceedings of the 1997 SCS European Multiconference on Computer Simulation. Estambul, Turquía. 1997.

[WGR99] WAINER, G.; GIAMBIASI, N.; RODRÍGUEZ, D.; "*Simulating Cell–DEVS models in parallel*". Proceeding of the 15th ISPE/FAC. International Conference on CAD/CAM, Robotics and Factories of the Future. Aguas de Lindoia, San Pablo, Brasil. 1999.

[Wol86] WOLFRAM, S. "*Theory and Applications of Cellular Automata*". Vol. 1, Advances Series on Complex Systems. World Scientific, Singapore, 1986.

[WZ99] WAINER, G.; ZEIGLER, B. "*Experimental Results of Timed Cell-DEVS quantization*". Informe interno del Depto. de Computación. UBA. Enviado para su publicación. 1999.

[Zei76] ZEIGLER, B. "*Theory of Modelling and Simulation*". Wiley, N.Y. 1976.

[Zei84] ZEIGLER, B. "*Multifaced Modeling and Discrete Event Simulation*". Academic Press, 1984.

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

[Zei97] ZEIGLER, B. "*Objects and systems: principled design with Java/C++ Implementation*". *Springer-Verlag*. New-York, 1997.

Bibliografía Adicional

Aho, A; Sethi, R.; Ullman, J., "*Compiladores, principios técnicas y herramientas*", Addison Wesley, 1990.

Barylko, A.; Beyoglonian J. "*CD++. Una Herramienta de implementación de modelos Cell-DEVS Binarios*". Tesis de Licenciatura. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

Barylko, A.; Beyoglonian J.; Wainer, G. "*A General Application DEVS Environment*". Informe Técnico No. 98-005. Departamento de Computación. FCEyN. UBA. Publicación on-line del DEVS Group, Austria (http://www.cast.uni-linz.ac.at/devs-archive).

Barylko, A.; Beyoglonian, J.; Wainer, G. "Implementation of a Cell-DEVS modeling environment". Informe Técnico No. 98-006. Departamento de Computación. FCEyN. UBA. Publicación on-line del DEVS Group, Austria (http://www.cast.uni-linz.ac.at/devs-archive).

Beuret, O.; Tomassini, M.; "*Behaviour of Múltiple Generalized Langton's Ants*". Logic Systems Laboratory. Swiss Federal Institute of Technology. Lausanne, Suiza.

Chandy, K.M.; Misra, J.; "*Distributed Simulation: A case of study in design and verification of distributed systems*". IEEE Transactions on Software Engineering 5. pp.440-452. September 1979.

Chandy, K.M.; Misra, J.; "*Asynchroneous distributed simulation via a sequence of parallel computations*". Communications of the ACM, (24), pp. 198-206. Abril 1981.

Cho, H.J.; Cho, Y.K.; "*DEVS-Java. Reference Guide*". Artificial Intelligence and Simulation Research Group. Dept. of Electrical and Computer Engineering. University of Arizona. 1997.

Chow, A., "*A C++ binding of the parallel devs formalism*". Proceedings of the SCS Multiconference on PDES. pp. 38. 1995.

Christensen, E.R.; Zeigler, B.P.; "*Hierarchical, distributed, object oriented and knowledge base simulation*". In 8th Military Operations Research Society Symposiums, Annapolis, MD. US Naval Academy. 1990.

Eckart, J.D.; "*Cellang: Language Reference Manual*". Radford University. 1997.

Espericueta, R.; "*Cellular Automata Dynamics*". Math Dept. Bakersfield College. 1997.

Fujimoto, R.M.; "*Parallel Discrete Event Simulation*". Communications of the ACM, vol 33, No. 10, pp. 30-53, November 1990.

Kim, T.G.; Park, S.; "*The DEVS formalism: Hierarchical modular system specifications in C++*". In Proc. 1992 European Simulation Mult-Conf. Pages152-156. York. UK. 1992.

Kim, T.G.; Hong, G.P.; "*The DEVS Formalism: A Framework for Logical Analysis and Performance Evaluation for Discrete Event Systems*". Dept. of Electrical Engineering. Korea Advanced Institute os Science and Technology. 1994.

Kim, T.G.; Song, H.S.; "*The DEVS Framework for Discrete Event Systms Control*". Dept. of Electrical Engineering. Korea Advanced Institute os Science and Technology. 1994.

Kim, T.G.; Lee, K.H.; "*Performance Modeling and Analysis of Distributed Access Network System Using DEVSim++*". 1994.

Levine, R. ; Mason, T. ; Brown, D., "*lex & yacc*", O'reilly & Associates, Inc. 1992.

Lindgren, K.; "*Evolutionary Dynamics in Game-Theoretic Models*". Talk presented at the Workshop: "The economy as an evolving complex system II". Santa Fe Institurte, Agosto-1995.

Mitchell, M.; "*Computation in Cellular Autómata: A Selected Review*". Santa Fe Institute. 1996.

Moore, C.; Crutchfield, J.; "*Quantum Automata and Quantum Grammars*". Santa Fe Institute. USA.

Otero, R.; Lorenzo, D.; Cabalar, P.; "*Automatic induction of DEVS structures*". Dept. Computación. Fac. Informática. University of La Coruña. España.

Praehofer, H.; "*Distributed Discrete Event Simulation*". Technical Report TR-93-1. Dept of Systems Theory. Johannes Kepler University. Linz. Austria. 1993.

Praehofer, H.; "*An Environment for DEVS-based Multi-Formalism Modeling and Simulation in C++*". Dept of Systems Theory. Johannes Kepler University. Linz. Austria. 1993.

Praehofer, H.; Reisinger, G.; "*Distributed Simulation of DEVS-Based Multiformalism Models*". *Proceedings of AI Simulation and Planning in High-Autonomy systems*, pp. 150-156. 1994.

Praehofer, H.; Reisinger, G.; "*Object Orieted Realization of a Parallel Discrete Event Simulator*". *Technical Report Johannes Kepler University, Department of System Theory and Information Engineering*. Linz, Austria. 1995.

Praehofer, H.; Pree, D.; "*Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs*". In Winter Simulation Conference. 1993. Pag. 595-603. Los Angeles, California.

Press, W.; Teukolsky, S.; Vetterling, W.; Flannery, B.; "*Numerical Recipes in C. The Art of Scientific Computing*". Cambridge University Press. Second Edition. 1992.

Sipper, M.; "*Non-Uniform Cellular Automata: Evolution in Rule Space and Formation of Complex Structures*". Artificial Life IV. Pages 394-399. The MIT Press. 1994.

Sipper, M.; "Studying Artificial Life Using a Simple, General Cellular Model". Artificial Life Journal. Vol. 2, No. 1, Pages 1-35. The MIT Press. 1995.

Sipper, M.; Tomassini, M.; "*Generating Parallel Random Number Generators by Cellular Programming*". International Journal of Modern Physics C, Vol 7, Nro 2. Pags 181-190. 1996.

Stroustrup, Bjarne "*The C++ Programming Language*", Addison Wesley, 1997.

Tadaki, S.; "*Two-dimensional cellular automaton model of traffic flow with open boundaries*". Technical Report. Dept. of Information Science. Saga University, Japan. 1996.

Thomas, C.; "*Interface-Oriented Classification of DEVS Models*". Research and Technology Dept. Daimler-Benz AG. Berlín, Alemania. 1994.

Uhrmacher, A.M.; Arnold, R.; "*Distributing and Maintaining Knowledge: Agents in Variable Structure Environments*". Dept. of Computer Science. Institute of Artificial Intelligence. Ulm, Alemania. 1994.

Wainer, G. A. "*Introducción a la Simulación de Sistemas de Eventos Discretos*". Reporte Técnico 96-005. Departamento de Computación – FCEyN. Universidad de Buenos Aires. 1996.

Wainer, G.; Giambiasi, N. "*Specification, Modelling and Simulation of Timed Cell-DEVS Models*". Technical Report TR-97-006, Departamento de Computación, FCEyN - UBA. Enviado para su publicación a ACM Transactions on Modelling and Simulation. 1998.

Wainer, G.; Giambiasi, N. "*CELL-DEVS models with transport and inertial delays*". En Proceedings of the SCS European Simulation Symposium on Industrial Simulation, Passau, Alemania. 1997.

Wainer, G.; Frydman, C.; Giambiasi, N. "*An Environment for Simulation of Cellular DEVS Models*". En Proceedings of the 1997 SCS European Multiconference on Computer Simulation. Estambul, Turquía. 1997.

Wainer, G.; Rodríguez, D.; "*Parallelizing CD++*". IFAC/CARs & FOF. Campinas, Brasil. Agosto de 1999.

Zeigler, B.; Song, H.S.; Kim, T.G.; Praheofer, H.; "*DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems*".

Zeigler, B.; Barros, F.; Mendes, M.; "*Variable DEVS – Variable Structure Modeling Formalism: An Adaptive Computer Architecture Application*". 1994.

Zeigler, B.; Praehofer, H.; "*On the Expresibility of Discrete Event Specified Systems*".

Zeigler, B.; Chow, A.; "*Abstract Simulator for the Parallel DEVS Formalism*". *Technical Report. University of Arizona.* 1994.

Zeigler, B.; Vahie, S.; "*DEVS Formalism and Methodology: Unity of Conception/Diversity of a Application*". AI & Simularion Research Group. Dept. of Electrical and Computer Eng. University of Arizona.

Zeigler, B.; Kim, J.; "*Extending the DEVS-scheme Knowledge-based Simulation Environment for Real-Time Event-Based Control*". AI Simulation Group. Dept. of Electrical and Computer Eng. University of Arizona.

Zeigler, B.; Moon, Y.; Kim, D.; Kim, J.G.; "DEVS-C++: A High Performance Modelling and Simulation Environment". Technical Report, Department of Electrical and Computer Engineering, University of Arizona . In Proceedings of 29th. Hawaii International Conference on System Sciences, January 1996.

Artículos Publicados

New Extensions to the CD++ Tool

Proceedings of SCSC'99
Summer Computer Simulation Conference
Chicago, Illinois, EE.UU.
Julio de 1999

Redefinition of a Specification Language for Cell-DEVS Models

Proceedings of ISAS'99
Fifth International Conference on Information Systems, Analisys and Synthesis
Orlando, Florida, EE.UU.
Agosto de 1999

Parallelizing CD++

Proceedings of IFAC/CARs & FOF
Campinas, Brasil
Agosto de 1999