CD++

Implementación de modelos Cell–DEVS n–dimensionales

Manual del Usuario

Contenido

Indice de Figuras

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 6.

–**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 9.

–**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 8.

–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 10.

–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 11.

–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 13).
La configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
.
.
.
nombre_variable_n_ : valor_var_n_
```

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:

$(y_0, y_1, ..., y_n)$ donde: $0 \le y_i < x_i$ $\forall$ 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: $0 \le x_{1,i} < dim_1 \ \vee \ 0 \le x_{1,i} < Width \ \forall$ i = 1, ..., m
$0 \le x_{2,i} < dim_2 \ \vee \ 0 \le x_{2,i} < Height \ \forall$ i = 1, ..., m
$0 \le x_{k,i} < dim_k$ $\forall$ 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.
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 $\Re \cup \{?\}$.
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 5.
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 6.

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 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 *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.

3.1 Ejemplo. Construcción de una Cola

Una cola es un dispositivo de almacenamiento temporal con política *FIFO* (First In First Out). Para implementarlo en *CD++* se debe crear una nueva clase (que en este caso llamaremos *Queue*) que extienda a *Atomic*.

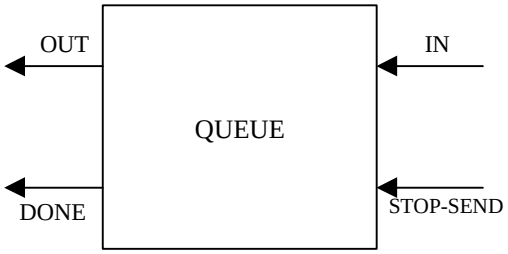

Figura 5 – Esquema de una Cola

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 6 – 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 7 – 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 8 – Métodos para las funciones de Salida y de Transición Interna de la *Cola*

4 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.

4.1 Gramática del Lenguaje

La sintaxis del lenguaje usado por *CD++* para la especificación del comportamiento de los modelos celulares atómicos puede definirse con la BNF mostrada en la Figura 9, 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 9 – Gramática usada para la definición de reglas bajo *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.

4.2 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 *CD++* se muestra en la siguiente tabla:

Orden	Código	Asociatividad
Menor Precedencia	**AND OR XOR IMP EQV**	Izquierda
	NOT	Derecha

Mayor

= != > < >= <=	
+ -	Izquierda
* /	Izquierda
FUNCTION	
REAL INT BOOL COUNT **?** **STRING** CONSTFUNC	
( )	

Figura 10 – Orden de Precedencia y asociatividad usadas por *CD++*

4.3 Funciones y Constantes usadas por el lenguaje

4.3.1 Tratamiento de Valores Booleanos

Esta sección describe las constantes que representan los valores booleanos de la lógica trivalente utilizada por *CD++* y se muestran los operadores aplicables sobre las mismas.

4.3.1.1 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.

4.3.1.2 Operadores Booleanos

4.3.1.2.1 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 11 – Comportamiento del operador booleano *AND*

4.3.1.2.2 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 12 – Comportamiento del operador booleano *OR*

4.3.1.2.3 Operador *NOT*

El comportamiento del operador booleano ***NOT*** se define con la siguiente tabla:

NOT	
T	F
F	T
?	?

Figura 13 – Comportamiento del operador booleano *NOT*

4.3.1.2.4 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 14 – Comportamiento del operador booleano *XOR*

4.3.1.2.5 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 15 – Comportamiento del operador booleano *IMP*

4.3.1.2.6 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 16 – Comportamiento del operador booleano *EQV*

4.3.2 Funciones y Operaciones para el Tratamiento de Números Reales

4.3.2.1 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.

4.3.2.1.1 Operador =

El operador = se utiliza para saber si dos números reales son iguales. Su comportamiento se define a continuación:

=	**?**	**n° real**
?	T	?

[1] De aquí en más, al referirse al termino *Número Real* se estará considerando a un valor perteneciente al conjunto $\boldsymbol{R} \cup \{ \textbf{?} \}$

n° real	?	= de nros. reales

Figura 17 – Comportamiento del operador relacional =

4.3.2.1.2 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 18 – Comportamiento del operador relacional !=

4.3.2.1.3 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

Figura 19 – Comportamiento del operador relacional >

4.3.2.1.4 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 20 – Comportamiento del operador relacional <

4.3.2.1.5 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 21 – Comportamiento del operador relacional <=

4.3.2.1.6 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 22 – Comportamiento del operador relacional >=

4.3.2.2 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 23 – Operadores Aritméticos

Para el caso en que se produzca una división por cero, se devolverá el valor indefinido.

4.3.2.3 Funciones sobre Números Reales

4.3.2.3.1 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

4.3.2.3.2 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.

4.3.2.3.2.1 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* ∉ [-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 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 **?**.

4.3.2.3.2.2 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*) ∀*x*

4.3.2.3.2.3 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, ?) = ?

4.3.2.3.3 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

4.3.2.3.4 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 4.3.2.3.8.

4.3.2.3.5 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 4.3.2.3.1.

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 $n \rightarrow$ *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*.

4.3.2.3.6 Funciones para la obtención de Mínimos y Máximos

Función min

Signatura: **min** : *Real a* x *Real b* $\rightarrow$ *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* $\rightarrow$ *Real*
Descripción: Devuelve el máximo entre *a* y *b*.
Si *a* ó *b* son indefinidos, devuelve **?**.

4.3.2.3.7 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* $\rightarrow$ *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* $\rightarrow$ *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)
```

4.3.2.3.8 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* m x *Real* s → *Real*
Descripción: Devuelve un número real pseudo-aleatorio con distribución Normal (m, s), donde m es la media, y s es el desvío estándar.
Si m *ó* s son indefinidos, ó s 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*))

4.3.2.3.9 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 $\left(\frac{a}{b}\right)$
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*.

4.3.2.4 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.

4.3.2.5 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.

4.3.2.6 Funciones para la Conversión de Valores entre distintas Unidades

4.3.2.6.1 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 **?**.

4.3.2.6.2 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*, *q*), 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*, *q*), y devuelve *q*. Si *x ó y* son indefinidos, devuelve **?**.

Función polarToRect_x

Signatura: **polarToRect_x** : Real *r* x Real *q* → *Real*
Descripción: Convierte la coordenada polar (*r*, *q*) a la forma cartesiana (*x*, *y*), y devuelve *x*. Si *r ó q* son indefinidos, ó *r* es negativo, devuelve **?**.

Función polarToRect_y

Signatura: **polarToRect_y** : Real *r* x Real *q* → *Real*
Descripción: Convierte la coordenada polar (*r*, *q*) a la forma cartesiana (*x*, *y*), y devuelve *y*. Si *r ó q* son indefinidos, ó *r* es negativo, devuelve **?**.

4.3.2.6.3 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 **?**.

4.3.2.7 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 24 – 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 25.

```
PortInTransition: portA@celda(0,0)     functionA
PortInTransition: portB@celda(1,1)     functionA

[functionA]
rule: 10     100    { portValue(thisPort) > 10 }
rule: 0      100    { t }
```

Figura 25 – 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.

4.3.3 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 x 10^{-28} gramos.

Constante proton_mass
Devuelve la masa de un protón, que es 1.6726231 x 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.

4.4 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.

4.4.1 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 26 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 26 – 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 27.

```
[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 27 – 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 28, 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 28 – Ejemplo de una referencia circular directa detectada por el simulador

4.4.2 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 29 – 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 30 – 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*.

5 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 31.

```
(x0,x1,...,xn) = value_1
...       ...      ...
(y0,y1,...,yn) = value_m
```

Figura 31 – 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 32 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 32 – Ejemplo de un archivo para la definición de valores iniciales para un Modelo Celular

6 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 33.

```
value_1
...  ...
value_m
```

Figura 33 – 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 14) es posible la conversión de un archivo que contiene la descripción de una lista de valores según el formato descripto en la sección 5 a un archivo de Mapa de Valores.

7 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 34 – Ejemplo de un archivo para la definición de Eventos Externos

8 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 35 – Ejemplo de un archivo de Salida

9 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 36 – Fragmento de un archivo de log

10 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 37 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 37 – Salida generada por el modo de debug del parser para el *Juego de la Vida*

11 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 38 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 38 – Fragmento de la salida generada por el modo de debug para la Evaluación de Reglas

12 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 39 – 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 40 – Ejemplo de llamadas al *DrawLog*

Nota: Si se ejecuta el *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 *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.

12.1 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 41 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 41 – Fragmento de la salida generada por el *DrawLog* para un modelo bidimensional

12.2 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 42 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 42 – Fragmento de la salida generada por el *DrawLog* para un modelo tridimensional

12.3 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 43 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 43 – Fragmento de la salida generada por el *DrawLog* para un modelo de dimensión 4

13 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 44 – 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 5 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.

14 Conversión de Archivos de Valores a Mapa de Valores – *ToMap*

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 45 – 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.

15 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 *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 *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 *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 46 – 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.

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 *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  00010001111000000000
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.

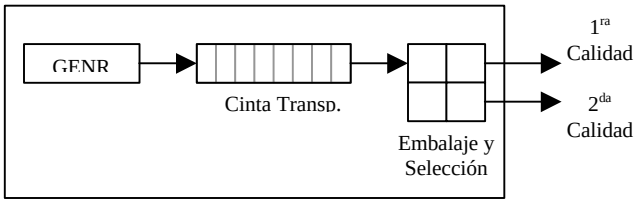

Figura 49 – Esquema de Acoplamiento de la Clasificadora de Materias Primas

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
```

```
(2,4,1) = 1
(2,4,2) = 1
(2,5,0) = 1
(2,6,1) = 1
(3,2,1) = 1
```

```
(5,1,2) = 1
(5,2,0) = 1
(5,2,2) = 1
(5,3,0) = 1
(5,3,1) = 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
```

```
(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
```

```
(5,5,1) = 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.