# Informe Científico

# 1 Introducción a la simulación

Existen diversidad de sistemas complejos para los cuales se hace muy dificil, y en casos es imposible, hallar soluciones analiticas o heurísticas con resultado satisfactorio. Para el estudio de este tipo de sistemas se ha difundido el uso de metodologías y herramientas de simulación.

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo del sistema, las decisiones pueden verificarse artificialmente, un mismo modelo puede usarse muchas veces, etc. La simulación es de empleo más simple que ciertas técnicas analíticas y precisa menos simplificaciones.

## *1.1 Sistemas reales*

Un sistema es una entidad real o artificial, representa parte de una realidad, restringida por un entorno. Está compuesto por entidades que experimentan efectos espacio-tiempo y relaciones mutuas.

Desde otro punto de vista, podemos describir a un sistema como un conjunto ordenado de objetos lógicamente relacionados que atraviesan actividades, interactuando para cumplir los objetivos propuestos.

Distinguimos dos interpretaciones de la palabra **sistema**:

Un **sistema real** es una combinación de elementos con relaciones estructurales que se influencian mutuamente.

Un **sistema dinámico** es una construcción formal que describe conceptos generales de modelización para distintas clases de disciplinas [Gia96].

Dado un sistema, un modelo es una representación inteligible (abstracta y consistente) del mismo.

El objeto de la construcción de un modelo es el desarrollo de la representación simplificada y observable del comportamiento y/o estructura del sistema real. Un modelo es simplificador, un filtro de la realidad.

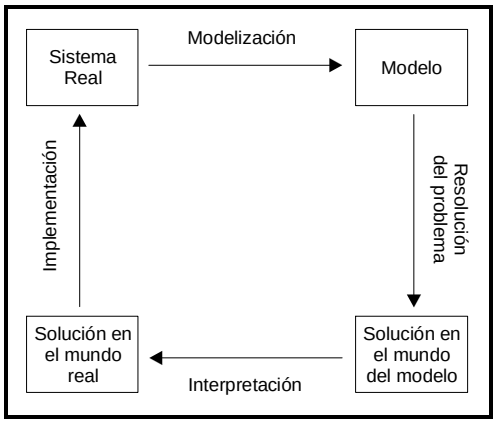



**Ilustración 1 - Pasos en el estudio de un sistema por medio de modelización [Gia96]**

## 1.2 Modelos y Simulación

En muchos casos no es posible la experimentación directamente sobre el sistema a estudiar por lo que se recurre al uso de modelos. Se distinguen dos grandes grupos de métodos para modelar sistemas complejos:

**Analíticos**: los modelos están basados en razonamiento. Suelen ser simbólicos, y permiten obtener soluciones generales al problema.

Un formalismo analítico muy difundido son las ecuaciones diferenciales. Sin embargo para sistemas complejos, con pocas excepciones, serán analíticamente intratables y numéricamente prohibitivos de evaluar, por ende, para poder usar estos métodos para el analisis de los los problemas que existen en el mundo real, se debe simplificar el modelo a un nivel tal que las soluciones obtenidas pueden alejarse de la realidad. Frente a esta situación, la simulación ofrece otra aproximación de resolución de problemas que permite tratar cierta complejidad.

**Basados en simulación**: en ellos no existen soluciones generales. Buscan soluciones particulares para el problema.

Si el problema es simple, es conveniente el uso de métodos analíticos ya que nos permiten obtener soluciones generales. En cambio, si es complejo, usando simulación se pueden probar distintas condiciones de entrada que no serían posibles de probar y obtener resultados de salida significativos.

El uso de simulación permite experimentación controlada, compresión de tiempo (una simulación se realiza en mucho menos tiempo que el sistema real que modela), y análisis de sensibilidad.

Algunos problemas que existen en el uso de simulación son su tiempo de desarrollo, los resultados pueden tener divergencia con la realidad (precisan validación), y para reproducir el comportamiento del sistema simulado se precisa una colección extensiva de datos. Por ende, la simulación es el proceso de diseñar un modelo de un sistema real, y conducir experimentos basados en computadoras para describir, explicar y predecir el comportamiento del sistema real [Gia96].

En general, para hacer un simulador se siguen los siguientes pasos:

**Planteo del problema:** En primer lugar se identifica el problema a resolver y se describe su operación en términos de objetos y actividades dentro de un marco físico**.** Luego se identifican las variables de entrada y salida del sistema. Finalmente se construye una estructura mas detallada del modelo, identificando todos los objetivos con sus atributos e interfaces.

**Recolección y análisis de los datos de entrada**: Se estudia el sistema real para obtener datos de entrada vía observación. El objetivo es la obtención de una muestra estadísticamente válida. Finalmente se decide qué datos serán tratados como aleatorios y cuáles se asumirán como determinísticos.

**Modelización:** En esta fase se construye un modelo del sistema con los aspectos que se quieren simular.

**Implementación:** En esta etapa, en base al lenguaje seleccionado, se construye una simulación del modelo que pueda ejecutarse en una computadora.

**Verificación y validación del modelo**: La validación enfoca la correspondencia entre el modelo y la realidad, pudiéndose realizar modificaciones al modelo u/o a la implementación de ser necesario.

**Experimento de simulación y optimización:** En esta fase se hace evaluación estadística de las salidas del simulador para determinar algún nivel de precisión de las medidas de desempeño.

**Análisis de datos de salida:** Por último se analizan las salidas de la simulación para comprender el comportamiento del sistema. Estas salidas se emplean para obtener respuesta al comportamiento del sistema original.

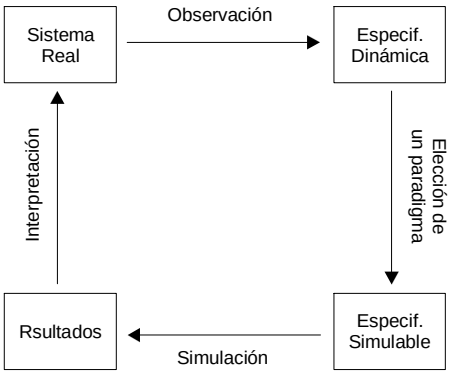



**Ilustración 2 - Pasos en la especificacion de modelos**

Giambiasi [Gia96] define simulación como: "*La reproducción del comportamiento dinámico de un sistema real en base a un sistema con el fin de llegar a conclusiones aplicables al mundo real*".

Por ende, simulación es el proceso de diseñar un modelo de un sistema real, y conducir experimentos para describir, explicar y predecir el comportamiento del mismo.

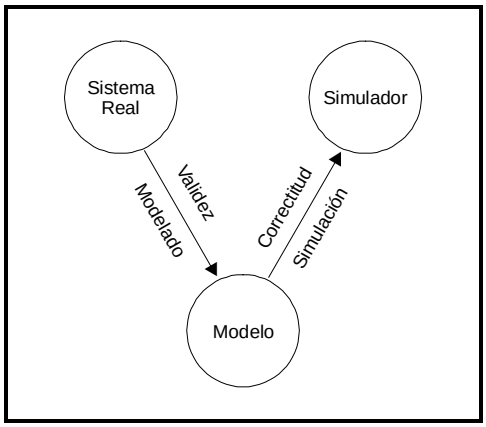



**Ilustración 3 - Relaciones de modelado y simulación [Zei90]**

## *1.3 Paradigmas de Modelado*

A través de la tecnología moderna, el hombre ha creado sistemas dinámicos que no pueden ser descriptos fácilmente por medio de ecuaciones diferenciales ordinarias o parciales. Como ejemplos de tales sistemas pueden nombrarse lineas de producción/ensamblado, control de tráfico, etc.

En estos sistemas, la evolución en el tiempo depende de interacciones complejas de varios eventos discretos y de su temporalidad, tales como la llegada o partida de un trabajo, y la iniciación o finalización de una tarea, etc.

*"El **estado** de tales sistemas sólo cambia en instantes discretos de tiempo en lugar de continuamente"* [Ho89] .

En conjunción al avance de la tecnología de computadoras, la simulación se ha convertido en la alternativa para el estudio de este tipo de sistemas complejos, y se han desarrollado una gran variedad de paradigmas de modelado, que pueden clasificarse de acuerdo a distintos criterios: con respecto a la base de tiempo, hay paradigmas a **tiempo continuo**, donde se supone que el tiempo evoluciona de forma continua (es un numero real), y a **tiempo discreto**, donde el tiempo avanza por saltos de un valor entero a otro (el tiempo es un entero). Con respecto a los conjuntos de valores de las variables descriptivas del modelo, hay paradigmas de estados o **eventos discretos** (las variables toman sus valores en un conjunto discreto), **continuos** (las variables son números reales), y **mixtos** (ambas posibilidades) [Gia96].

En cuanto a la caracterización del problema a modelar, los modelos pueden ser **prescriptivos** si formulan y optimizan el problema (en general son métodos analíticos) o **descriptivos** si describen el comportamiento del sistema (suelen ser métodos numéricos). Por otro lado, un modelo se dice que es determinístico si todas las variables tienen certeza completa y están determinadas por sus estados iniciales y entradas.

El modelo se dice **probabilístico** en el caso que una respuesta pueda tomar un rango de valores dado el estado inicial y sus entradas (si usa variables aleatorias se dice que el modelo es estocástico). En un modelo **probabilístico**, los cambios de estado del modelo se producen por medio de leyes aleatorias: las entradas al modelo son aleatorias (siendo el modelo determinista), o el tiempo de llegada de los eventos es aleatorio [Zei96].

De acuerdo al entorno, los modelos son **autónomos** (no existen entradas) o no autónomos (existen entradas). Los autónomos evolucionan únicamente en base a la función de tiempo.

## *1.4 DEDS (Discrete Events Dynamic Systems)*

Los sistemas donde las variables son discretas a tiempo continuo reciben el nombre de **Sistemas Dinámicos de Eventos Discretos** (DEDS - Discrete Events Dynamic Systems) en oposición a los **Sistemas Dinámicos de Variables Continuas** (CVDS - Continuous Variable Dynamic Systems) que se describen por ecuaciones diferenciales .

Un paradigma de simulación para DEDS asume que el sistema simulado sólo cambia de estados en puntos discretos en tiempo simulado, o sea que el modelo cambia de estado ante ocurrencia de un evento. Para implementar una simulación de eventos discretos se suele utilizar:

Un conjunto de variables de estado discretas.

Un planificador que contiene una lista cronológica de eventos (o mensajes) a tratar. Un evento es un cambio de estado que debe efectuarse a una hora $t_i \in \Re$ (el tiempo es continuo).

Un reloj global que indica el instante actual de simulación; es la hora de ocurrencia del evento que se está tratando en la actualidad. Comparado con sistemas de variables continuas, fundamentales en la física, el modelado de DEDS es un fenómeno relativamente reciente.

A pesar que pertenece al dominio de la Investigación Operativa (desde el punto de vista que la IO puede pensarse como la técnica de operaciones y eventos de sistemas hechos por el hombre), el desarrollo de DEDS también recibió un gran impulso de la teoría de control y sistemas. En particular, los conceptos de dinámica (constantes de tiempo, tiempo de respuesta, frecuencia, controlabilidad) son importantes en el desarrollo de modelos y herramientas de DEDS [Ho89].

## *1.5 Modelos Simulables*

Los modelos dinámicos descriptos hasta aqui no son directamente simulables, para ello debemos realizar una conversión de la especificación a una nueva especificación simulable, y luego interpretar los resultados de la simulación en el sistema dinámico (hacer el pasaje inverso). Para esto se debe considerar tres objetivos básicos:

El sistema real en existencia debe observarse como una fuente de datos, analizar sus variables, relaciones de entrada/salida y tipología de las mismas (discretos/continuos). El objetivo de la construcción de un modelo es el desarrollo de una representación simplificada y observable del comportamiento.

El modelo es un conjunto de instrucciones para generar datos comparables a los observables en el sistema real.

El simulador ejecuta las instrucciones del modelo para generar su comportamiento. Estos objetos básicos están ligados por dos relaciones:

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

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

# 2 Formalismo DEVS (Discrete EVents Systems)

Muchas de las aproximaciones existentes para modelar sistemas de eventos discretos tratan de describir el fenómeno combinando diversas técnicas, haciendo muy difícil el desarrollo de las simulaciones. Para evitar estos problemas, Zeigler [Zei76] propuso un mecanismo de simulación jerárquica conocido como DEVS. Esta aproximación propone una teoría de modelado de sistemas a tiempo continuo (el tiempo evoluciona en forma continua) usando modelado de eventos discretos.

La aproximación provee una forma de especificar un objeto matemático llamado sistema. Este se describe como un conjunto consistente de una base de tiempo, entradas, salidas y funciones para calcular los siguientes estados y valores de salida. Soporta la construcción de modelos de forma jerárquica y modular. Además de un medio para construir modelos simulables, provee una representación formal para manipular matemáticamente sistemas de eventos discretos. El formalismo define cómo generar nuevos valores para las variables y los momentos en los que estos valores deben cambiar.

DEVS es un formalismo universal para modelar y simular DEDS. Puede verse como una forma de especificar sistemas cuyas entradas, estados y salidas son constantes en intervalos, y cuyas transiciones se identifican como eventos discretos. Los intervalos de tiempo entre ocurrencias son variables, lo que trae ciertas ventajas frente a los formalismos con una granularidad única donde es difícil describir los modelos debido a que hay muchos procesos operando en distintas escalas de tiempo, y la simulación no es eficiente ya que los estados deben actualizarse en el momento con menor incremento de tiempo, lo cual desperdicia tiempo cuando se aplica a los procesos mas lentos.

La base de la modelización y simulación del formalismo DEVS concierne tres componentes básicos:

El **sistema real** (origen de datos).

El **modelo** representa el conjunto de instrucciones para generar datos comparables a los que se obtienen del sistema real. El comportamiento del modelo es el conjunto de todos los posibles datos que pueden ser generados por la ejecución de las instrucciones del modelo.

El **simulador** ejecuta las instrucciones del modelo creando su comportamiento.

Los componentes están unidos por las relaciones **modela** y **simula**.

La relación **modela** vincula al sistema real y al modelo, definiendo cuán bien el modelo representa el sistema o entidad a ser modelada. En términos generales un modelo puede ser considerado válido si los datos generados por el modelo concuerdan con los datos producidos por el sistema real. La relación **simula** vincula al modelo y al simulador. Representa cuán fiel el simulador lleva a cabo las instrucciones del modelo.

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

## 2.1 Modelos Básicos

Los modelos básicos se definen por la tupla $M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$ donde:

X: conjunto de eventos externos.

S: conjunto de estado secuencial.

Y: conjunto de eventos externos generados como salida.

$\delta_{int}$: función de transición interna.

$\delta_{ext}$: función de transición externa.

$\lambda$: función de salida.

ta: función de avance de tiempo (time advance).

Los modelos poseen ports de entrada y salida a través de los cuales interactúan con el entorno. Los eventos determinan los valores que aparecen en esos ports.

Los eventos externos son recibidos en un port de entrada y la especificación del modelo debe determinar cómo responder a dichos eventos. Los eventos internos que se producen dentro del modelo, modifican su estado y producen eventos destinados a los ports de salida. Las influencias de los ports de salida determinarán si estos datos serán enviados a otros componentes como eventos externos.

Un modelo básico contiene la siguiente información:

Conjunto de ports de entrada a través de los cuales son recibidos los eventos externos.

Conjunto de ports de salida a través de los cuales son enviados los eventos.

Conjunto de variables de estado y parámetros. En general se usan las variables phase y sigma para representar el estado del modelo y el tiempo restante para el próximo cambio de estado.

Función de avance de tiempo controla la frecuencia de las transiciones internas.

Función de transición interna especifica como el cambio de estado del modelo una vez transcurrido el intervalo determinado por la función de avance de tiempo.

Función de transición externa específica como el modelo cambia su estado cuando recibe un evento en alguno de sus ports. Este cambio produce una nueva planificación para la próxima transición interna.

Función de salida genera los resultados en los ports de salida previamente la ejecución de la función de transición interna.

## 2.2 Modelos Acoplados

Los modelos básicos pueden unirse en el formalismo DEVS para componer un modelo que se define por la tupla

$$DN = <D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select>$$

donde:

D: conjunto de nombres de los componentes.

$M_i$: modelo básico correspondiente al componente i.

$I_i$: conjunto de influencias del componente i.

$Z_{i,j}$: es la función de traducción de la influencia i.

Select: función utilizada para determinar prioridades.

Los componentes de un modelo acoplado son otros modelos. Esto permite que un modelo acoplado puede ser usado como modelo básico dentro de otro modelo acoplado.

Un modelo acoplado contiene la siguiente información:

Conjunto de componentes que lo forman.

Las influencias para cada componente.

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

Conjunto de ports de salida a través de los cuales se envían los resultados.

Especificación del acoplamiento:

Acoplamiento en función de la entrada externa relaciona los ports de entrada del acoplado con los ports de entrada de los componentes.

Acoplamiento en función de las salidas relaciona los ports de salida de los componentes con los ports de salida del acoplado.

Acoplamiento interno relaciona los ports de salida de los componentes con ports de entrada de otros componentes.

La función **select** determina prioridades frente a dos eventos en la misma hora y la elección del hijo inminente para recibir el próximo evento.

## 2.3 Mecanismo de Simulación

La simulación comienza con la inicialización de los modelos que la componen. La inicialización se produce solamente al comienzo de la simulación y es al sólo efecto que cada modelo que la compone inicialice su estado e informe la planificación de su próxima transición interna.

A partir de este momento la simulación comienza a ejecutar tomando en cuenta los eventos externos y la última planificación de transición interna. El próximo evento a procesar será el que posea la hora más cercana a la hora actual. Esto implica comparar los eventos externos con los internos y seleccionar el menor.

Una vez determinado el próximo evento se le comunica al modelo correspondiente. En caso de ser un evento externo el modelo invocará su función de transición externa, y si se trata de un evento interno invocará su función de transición interna.

Un modelo básico puede recibir dos eventos, un evento **interno** o un evento **externo**.

Al recibir un evento externo e $\in$ X (incluye el port por el cual ingresó y su valor) el modelo ejecutará su función de transición externa $\delta_{ext}$. El resultado de esta invocación es una posible re-planificación de su próximo evento interno.

Al recibir un evento interno se le indica al modelo que la transición interna debe llevarse a cabo, dado que la hora de arribo responde a la planificación hecha previamente. El comportamiento dentro de la transición interna depende de cada tipo de modelo básico. En primer lugar enviará un evento Y $\in$ Y como salida. Luego ejecutará su función de transición interna $\delta_{int}$ dando como resultado un cambio de estado y la planificación para su próxima transición interna. Si el modelo se encontraba en un estado $s_i \in S$, pasará al estado $s_{i+1} \in S$.

Para el tratamiento de eventos un modelo acoplado puede verse como un modelo básico de la siguiente manera: en un instante t cada componente i está en un estado $S_i$, y ha permanecido en ese estado durante $e_i$. El próximo evento en el sistema ocurrirá a la hora que se corresponda con el mínimo de las planificaciones para todos los componentes i en DN. Si más de un modelo coincide en la hora mínima, la función de selección **Select** elegirá uno. Sea i* el elegido (componente inminente), el modelo acoplado calculará la salida del componente (y*) aplicando $\lambda_i$*($s_i$*). Esta salida es enviada para cada un de las influencias de i* en la forma de una entrada traducida. Para cada influencia j la entrada $x_{i*,j}$ será equivalente a $Z_{i*,j}$(y*). Cada influencia j responde al evento externo generado por i* invocando su función de transición externa. Finalmente i* ejecutará su función de transición interna $\delta_{int}$.

## 2.4 Implementación DEVS-C++ para los modelos DEVS

Está basado en el formalismo DEVS y provee un ambiente para construir modelos de eventos discretos jerárquicos. La arquitectura del sistema de simulación deriva de los conceptos abstractos del simulador asociados con el formalismo DEVS.

Zeigler propone una jerarquía de clases responsable de administrar la recepción de mensajes y tomar las acciones correspondientes. La clase *Processor* es la clase base asbtracta y las clases *Simulator*, *Coordinator* y *Root-Coordinator* son las clases concretas. La asociación entre modelos y procesadores está dada por los pares Atomic-Simulator y Coupled-Coordinator. Por cada modelo existente existirá un único procesador asociado y cada procesador administrará un único modelo.

Todas las clases dentro de DEVS-C++ son subclases de la clase base *Entity* la cual provee métodos para identificación de entidades. *Models* y *Processors* son las subclases principales de entidad, proveen los constructores básicos necesarios para el modelado y simulación. La clase *Models* es la clase base de *Atomic* y *Coupled* las cuales son especializadas en algunos casos más específicos.

### 2.4.1 Model

*Model* tiene la variable de instancia *processor* que identifica al procesador asociado, *parent* indica cuál es el modelo acoplado que lo contiene. La clase *Atomic* representa los modelos básicos del formalismo subyacente, sus variables se corresponden con cada una de las partes del mismo.

Los métodos *int-transfn, ext-transfn, outputfn* y *time-advancefn* representan las respectivas funciones de transición externa, interna, salida y avance de tiempo del modelo básico.

La clase *Coupled-Model* es la que representa las construcciones juerárquicas del formalismo DEVS. Un modelo acoplado se define especificando los modelos que lo componen, llamados *children,* y las relaciones de acoplamiento que se establecen por medio de las influencias.

### 2.4.2 Processor

Los *Simulators, Coordinators* y *Root-Coordinators* son especializaciones de *Processors.* Tienen como responsabilidad llevar a cabo la simulación de los modelos DEVS implementando los principios abstractos de simulación desarrollados como parte de la teoría DEVS. En escencia un simulador abstracto es una descripción algorítmica de cómo llevar a cabo las instrucciones implícitas de los modelos DEVS para generar su comportamiento.

Los *Simulators* y *Coordinators* están diseñados para manejar modelos atómicos y acoplados en una relación uno a uno respectivamente. El par modelo - procesador es guardado en la variable *processor* y *devs-component* del modelo y procesador respectivamente.

Un *Root-Coordinator* maneja la simulación en su totalidad y está relacionado con el modelo acoplado que tenga el nivel más alto dentro de la jerarquía. La simulación se desarrolla en términos de mensajes que viajan entre los procesadores, llevando información con respecto a los eventos internos y externos, así como también datos necesarios para la simulación. Los mensajes tienen campos para conocer el origen del mensaje, la hora a la cual se produce y un contenido que consiste del nombre de un port y el valor.

Existen cuatro tipos de mensajes:

*: indica un cambio de estado, producto de un evento interno.

X: indica el arribo de un evento externo, trae consigo el port y el valor.

Y: salida del modelo, indican el port y el valor.

Done: indica el fin del procesamiento y un posible cambio en la planificacion de la próxima transición interna.

Un procesador recibe y manda varios tipos de mensajes.

Un coordinador transmite un mensaje X a los procesadores de los componentes DEVS a los cuales está dirigido. Cuando un simulador recibe un mensaje X invoca la función de transición interna de su componente DEVS y responde con un mensaje Done. Este último indica al padre que la transición ha sido llevada a cabo y lleva consigo la hora de su próximo evento interno.

Un mensaje * que llega a un procesador indica que debe llevarse a cabo un evento interno. Un coordinador responde a un mensaje de este tipo transmitiéndoselo a su hijo inminente (el hijo con la menor hora de próximo evento o que halla sido escogido por la función de selección en caso de un empate).

Cuando un coordinador recibe un mensaje Y de su hijo inminente, consulta su esquema de acoplamiento de salida externa para decidir si lo debe transmitir a su padre, así como también consulta su esquema de acoplamiento interno para obtener los hijos a los cuales el mensaje debe ser enviado. El coordinador esperará el mensaje Done de cada una de las influencias de los hijos. Teniendo la respuesta de todos sus hijos calculará el hijo inminente para la próxima transición interna, devolviendo esta hora en un mensaje Done dirigido al padre.

### 2.4.3 Comienzo de la Simulación

La simulación comienza con la inicialización de cada uno de los modelos atómicos. De esta forma se determina la hora de próximo evento. La inicialización se propaga en forma descendente por el árbol y obtiene como resultado una catarata de mensajes *done* acarreando la planificación de cada uno de los modelos que componen la simulación.

Cuando un *Root-Coordinator* recibe un mensaje Done de su hijo inminente responde con un mensaje * llevando la hora de su próximo evento. De esta forma se producen olas de mensajes hacia las hojas del árbol que resultan en mensajes Y y Done como respuesta hacia sus respectivos padres. El último de los mensajes Done llegará al simulador raíz y se producirá otra iteración en la simulación.

# 3 Autómatas Celulares

Los Autómatas Celulares son un formalismo para modelado de sistemas dinámicos complejos, de variables y tiempos discretos (el tiempo, espacio y estados del sistema son discretos), creados originalmente por Von Neumann y S. Ulam. Estos se definen como un conjunto infinito n-dimensional de celdas ubicadas geométricamente que conforma una grilla o malla de celdas. Cada punto de la grilla puede tener uno tener un estado elegido de un alfabeto finito. Cada una contiene el mismo aparato de cálculo que las otras y se conectan entre sí de forma uniforme. El estado de las celdas se actualiza simultáneamente y de forma independiente de los demás en pasos de tiempo discreto. Para ello se define la vecindad de una celda que es un grupo de celdas cercanas. Esta vecindad es homogénea para todas las celdas.

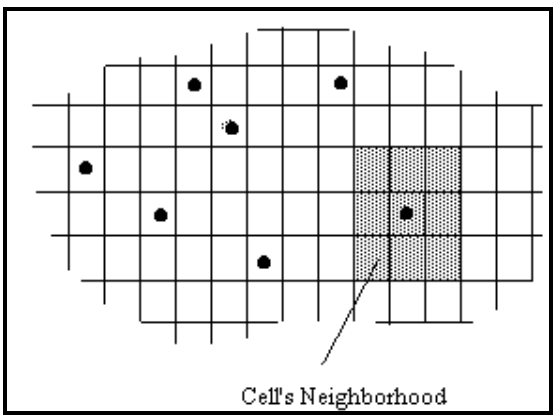


**Ilustración 4 - Autómata Celular**

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

Formalmente un Autómata Celular se define como:

$$CA = < A, S, C, N, T, t, c.\boldsymbol{Z}_0^+>$$

donde:

$A=\{A \subseteq Z \wedge \#A < \infty\}$: Alfabeto de estados de celdas.
$S=\{S \in A\}$: Conjunto de posibles estados para cada celda.
$C=\{c_{ij} / i,j \in Z \wedge c_{ij} \in S) \}$: Espacio celular.
$N=\{(i,j) / i \in [-\eta, \eta] \wedge j \in [-\eta, \eta]\}$: Definición del vecindario como un desplazamiento ($\eta \in Z$ es la dimensión).
$T=\{C \rightarrow C\}$: Función de transición global.
$\tau=\{N \rightarrow N\}$: función de cómputo local.
$c.Z_0+$: Base de tiempo discreta para el autómata celular.

El espacio celular está compuesto por celdas individuales que evolucionan en iteraciones de tiempo discreto. Cada celda en el espacio puede tomar un valor del conjunto *S*.

La evolución del autómata se define por la ejecución de la función de transición global que actualiza el estado global. El comportamiento de esta función de transición depende de los resultados de la función de cómputo local, que se ejecuta localmente en cada vecindario perteneciente a una celda. Conceptualmente el cómputo de esta función de transición local se realiza en forma sincrónica y potencialmente en paralelo en cada celda del espacio celular. Para cada autómata celular bidimensional la función de transición local se define como:

$$s_{ij}[t+1] = \tau(s_{i0,j0}[t],..., s_{i\eta, j\eta}[t])$$

Aqui $s_{ij}$ representa el estado de la celda (i,j) en un tiempo de simulación t y $\tau$ denota la función de cálculo local. Los parámetros $s_{i0,j0},..., s_{i\eta, j\eta}$ definen el vecindario de la celda. Una vez hecha la definición para dos dimensiones esto puede externderse a n.

El formalismo utiliza una base de tiempo discreta, la cual restringe la precisión y eficiencia de los modelos simulados. En el caso particular de autómatas celulares complejos será necesaria una gran cantidad de tiempo de cómputo para obtener el grado de precisión deseado. Más allá de esto, en muchos casos la mayoría de las celdas del autómata no necesitan actualización. La presencia de celdas comatosas (inactivas) permite la definición de un nuevo paradigma de bricolaje llamado autómatas celulares asincrónicos. En este caso los eventos pueden ocurrir en un instante impredecible de tiempo por lo cual se aproxima a una base de tiempo contínuo. El uso de una base de tiempo contínuo permite lograr mayor precisión y evitar la simulación de los períodos de inactividad de las celdas mejorando así la utilización de los recursos computacionales y obteniendo una simulación más performante. Sin embargo este enfoque requiere la sincronización explícita de las celdas, incrementando asi la complejidad de las rutinas y algoritmos que llevan a cabo la simulación. Por lo tanto se produce una sobrecarga inherente al problema mencionado con anterioridad. En algunos casos esta sobrecarga puede anular las mejoras obtenidas con el enfoque asincrónico.

# 4 El formalismo CELL-DEVS

El formalismo CELL-DEVS está basado en el formalismo DEVS extendiendo las especificaciones para implementar un autómata celular. El uso de formalismo DEVS se ve reflejado ya que se aprovecha la idea básica de modelos acoplados que aúnan a modelos atómicos permitiendo asi conformar el espacio de celdas. A su vez éstos son controlados por los coordinadores asociados.

En este formalismo aparecerán dos nuevos modelos, los modelos *AtomicCell* y los modelos *CoupledCell*. El primero extiende a los modelos atómicos agregando la posibilidad de poseer un estado, un comportamiento (determinado por la función de transición local) y una demora de actualización (transporte o inercial). El segundo extiende a los modelos acoplados permitiendo manejar un conjunto de celdas.

Los coordinadores celulares aprovechan el mecanismo de pasaje de mensajes de DEVS para llevar a cabo la simulación por de los acoplados celulares, cabe notar aquí que no es necesario un procesador nuevo para las celdas atómicas ya que el mecanismo de simulación se mantiene inalterado.

Al igual que en DEVS cuando llegue un mensaje externo a la celda planificará su demora. Para ello se debe ejecutar la función de transición interna. Esto permite obtener un estado posible para la celda junto con el tiempo de demora indicado en la especificación del comportamiento de ella. Tomando el valor recién calculado la celda, se planficará para ejecutar su función de transición interna. Previo a la llamada de la función de transición interna se invocará a la función de salida, enviando un nuevo estado al coordinador. Estas acciones se repetirán para cada evento almacenado en la cola de espera. Cuando la cola de espera está vacía la cola se pasiva. El coordinador envía los cambios a cada celda en la vecindad a medida que éstas cambian.

Este es el comportamiento en general para cada celda, dentro de ese concepto debemos especializar dos tipos de comportamientos según la demora que tengan las celdas. Estos tipos son demora inercial o demora de transporte.

## *4.1 Celdas con demora de transporte*

Las celdas con demora de transporte poseen una cola en la cual van encolando los eventos a medida que se programan. Cuando llega un evento por medio de la función de transición externa, la función de transición local es invocada. Esta función analizará el valor de los vecinos de la celda para obtener como resultado del comportamiento de la celda, un estado al cual la celda debe cambiar y una demora. Si este estado es distinto al anterior debe encolarse en la cola de próximos eventos. Debido a que una celda es independiente de la otra, cada una guarda una copia de los estados de los vecinos, asi como el estado anterior.

Al arribar un evento interno se invocan primero la función de salida y luego la función de transición interna. La función de salida envía como resultado al primer valor encolado. La función de transición interna elimina el primer elemento de la cola y luego analiza si la cola está vacía. Si existe un elemento significa que el modelo debe reprogramarse en función de lo encolado, en caso contrario debe pasivarse y por lo tanto su próximo cambio de estado será a la hora infinito.

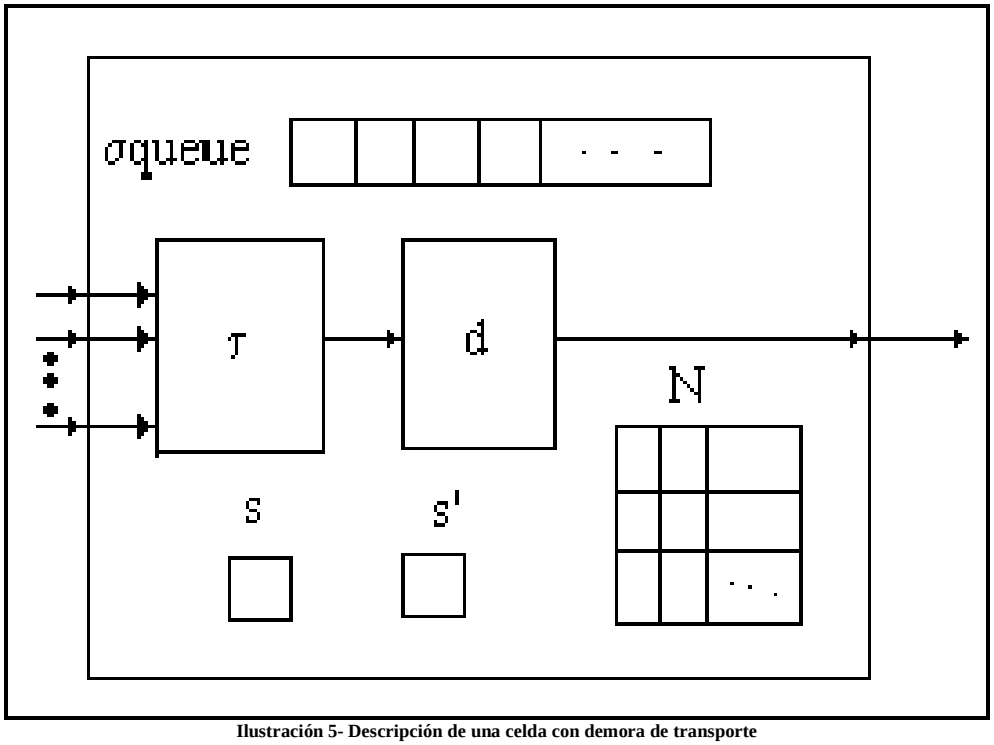



**Ilustración 5- Descripción de una celda con demora de transporte**

Formalmente la definición de una celda con demora de transporte (**T**ransport **D**elay **C**ell) es

$$\mathbf{TDC} = \langle I, X, Y, S, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D \rangle$$

donde:

**I** = $\langle \eta, P^x, P^y \rangle$ : Definición de la interfaz del modelo, donde
$\eta \in \boldsymbol{N}$ es el tamaño del vecindario
$P^x$ es un port de entrada que acepta solamente valores binarios.
$P^y$ es un port de salida que acepta solamente valores binarios.

***X*** ={0,1}: Conjunto de eventos externos de entrada.

***Y*** ={0,1}: Conjunto de eventos externos de salida.

**S** ={(s,phase,$\sigma_{queue}$, $\sigma$)}: Conjunto de estados donde
$s \in \{0,1\}$
phase $\in$ {active, passive}
$\sigma_{queue} = \{ (v_i, \sigma_i) / (\forall i: i \in [1,m] \wedge i \in \boldsymbol{N} : \sigma_i \in \boldsymbol{R_0}^+ \cup \infty \wedge v_i \in \{0, 1\} \}$
$\sigma \in \boldsymbol{R_0}^+ \cup \infty$

**N** ={ (i, j) / i $\in$ [-η, η] $\wedge$ j $\in$ [-η, η] }: Definición del vecindario como un desplazamiento.

**d** $\in \boldsymbol{R_0}^+$: Demora de transporte para cada celda.

$\boldsymbol{\delta_{int}}$: S$\rightarrow$S función de transición interna.

$\boldsymbol{\delta_{ext}}$: Q x X$\rightarrow$ S función de transición externa donde Q es el estado definido por Q={ (s, e) / s $\in$ S, and e $\in$ [0, D(s)]};

$\boldsymbol{\tau}$: N $\rightarrow$ N función de transición local

$\boldsymbol{\lambda}$: S$\rightarrow$*Y* función de salida y

**D**: S $\rightarrow \boldsymbol{R_0}^+ \cup \infty$, función de tiempo de vida del estado.

Esta definición permite a las celdas comportarse como modelos atómicos ya que dentro de su interfaz provee el manejo de puertos de entrada / salida, así como las funciones de transición interna, externa y de salida. La demora **d** es la demora que se aplicará cuando se ejecute la función local **τ.** Esta función se calcula sobre **N** que representa el conjunto de valores de los vecinos. Si el estado de la celda cambia todos los vecinos serán notificados.

La demora de transporte sólo permite analizar un evento cuando éste fue consumido ya que todo otro evento que arribe antes de ello será incorporado en la cola, si produce un cambio en el estado de la celda. La celda se mantiene activa mientras haya eventos en la cola, cuando ésta se vacía la celda se pasivará retornando como hora de próximo evento el valor infinito (∞).

## 4.2 Autómatas celulares con demora inercial

Otro tipo de demora es la demora inercial [GM76], ésta es muy útil cuando se desea representar una semántica con remoción para el comportamiento de una celda. La definición para las celdas que poseen demora inercial es idéntica a la dada para las celdas con demora de transporte en el punto anterior. La diferencia radica en el tratamiento de los eventos externos e internos y la estructura de la celda.

Una celda con demora inercial tiene por política el desalojo o descarte de los valores según corresponda, por lo tanto el procesamiento FIFO registrando todos los valores en caso de un cambio que realizaba la celda con demora de transporte es innecesario en este caso.

Frente a un arribo de un evento externo la celda ejecuta la función de cómputo local y obtiene como resultado un estado y una demora. Si el nuevo valor obtenido difiere del valor anterior la celda analiza tiempo restante para el próximo evento interno. Si no existe planificación alguna la celda se programa con la demora recién calculada. En caso de existir una programación se analiza el valor de σ (tiempo restante) para comprobar si es mayor que cero, de ser asi se descarta la programación anterior y se toma la nueva. De no ser así, contúa con lo planificado.

Frente a un evento interno la celda realiza la función de salida con el valor calculado, y en la función de transición interna cambia su estado a pasivo, ya que no tiene más eventos para programar.

## 4.3 Modelos celulares acoplados

Los modelos de celdas celulares pueden ser agrupados en otros modelos que los contienen llamados modelos celulares acoplados (*CellCoupled)* formando así un modelo multicomponente de celdas celulares y por ende todo el espacio de celdas.

Los modelos acoplados celulares generales (**G**eneral **C**oupled**)** se definen formalmente como :

$$\mathbf{GC} = \langle I, X, Y, X_{list}, Y_{list}, \eta, N, \{m, n\}, C, B, Z, select \rangle$$

Donde:

**{m, n}** / m, n ∈ ***Nat*** : Tamaño del espacio de celdas bidimensional.
**I**=<$P^x$,$P^y$>: Interfaz de comunicación con los modelos externos, donde
$P^x$ es un port de entrada que acepta solamente valores binarios.
$P^y$ es un port de salida que acepta solamente valores binarios.
***X***={0, 1}: Conjunto de eventos externos de entrada.
***Y***={0, 1}: Conjunto de eventos externos de entrada.
$\mathbf{Y_{list}}$={(k, j) / k ∈ [1,m] ∧ j ∈ [1,n]}: Lista de acoplamiento externo.
$\mathbf{X_{list}}$={(k, j) / k ∈ [1,m] ∧ j ∈ [1,n]}: Lista de acoplamiento interno.
**η** ∈ ***Nat***: Tamaño del vecindario
**N**={(i,j)/ i ∈ [-η, η] ∧ j ∈ [-η, η]}: Definición del vecindario como un desplazamiento.

**C**=$\{C_{ij} / i \in [1, m] \wedge j \in [1, n]\}$: Conjunto de celdas perteneciente al espacio celular.

**B**: Conjunto de celdas que representan el borde del modelo celular, este valor puede ser vacío indicando que el borde es circular (wrapped) o por extensión indicando las celdas que forman parte del conjunto borde, es decir

$\{C_{ij} \in C / (i=1 \vee i=m \vee j=1 \vee j=n)\}$

**Z** es la función de traducción definida por:

dado $q \in [0, \eta]$

$P_{ij}{}^{Y}q \rightarrow P_{kl}{}^{X}q$ donde $P_{ij}{}^{Y}q \in I_{ij} \wedge P_{ij}{}^{X}q \in I_{kl} \wedge (\forall f,g \in N: k=(i+f) \bmod m \wedge l=(j+g) \bmod n)$

$P_{kl}{}^{Y}q \rightarrow P_{ij}{}^{X}q$ donde $P_{kl}{}^{Y}q \in I_{kl} \wedge P_{ij}{}^{X}q \in I_{ij} \wedge (\forall f, g: f, g \in N: k = (i-f) \bmod m \wedge l = (j-g) \bmod n)$

**select** = { (k,l) / (k,l) ∈ **N** } **:** es la función que elije la celda inminente frente a una igualdad de posiblidades.

Esta especificación define a un modelo celular acoplado como un arreglo bidimensional de tamaño n x m. Dentro de sus características se encuentran el vecindario, que define a cuántas casillas de distancia se encuentra un vecino. Es por esto que para calcular el conjunto de celdas que son vecinas se toma las coordenadas de una celda y se les suma uno de los desplazamientos posibles que pertenecen a **N**.

La interfaz **I** del modelo indica cuáles son los puertos de entrada/salida que permiten interactuar al modelo con los modelos externos. Estos puertos deben aceptar solamente valores binarios ya que son los valores posibles para los estados de las celdas y por ende también son el resultado que producen. Esto es reflejado en los posibles conjuntos de valores de entrada (X) y de salida (Y).

Al igual que los modelos acoplados para indicar cómo deben relacionarse los modelos que lo componen existen dos listas, la $X_{list}$ y la $Y_{list}$. Estas representan la definición de las conexiones entre los puertos de las celdas. Cada celda tendrá una conexión de su puerto de salida al puerto de entrada de cada una de sus vecinas y viceversa. Si ella está incluída en su vecindario, entonces tendra una conexión a ella misma.

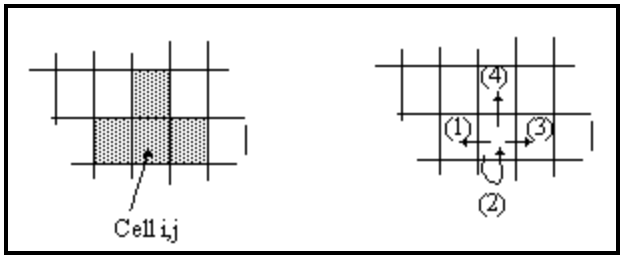



**Ilustración 6 - Conexión entre las celdas**

En el caso de los modelos acoplados celulares, cada port de entrada destino de una conexión de salida pertenecerá solamente a una celda del vecindario (dentro del modelo acoplado) o a un modelo externo. En los modelos acoplados comunes las relaciones entre los modelos hijos no tenían ningun tipo de restricción.

El conjunto de celdas **C** está dado por la combinación de todos los índices que se encuentren en rango de las filas y las columnas. El borde **B** del acoplado indica que puede tenerse un cuadrado de celdas que representen el borde o permitir que las referencias se hagan circulares, es decir que cuando una celda haga referencia a la posición de otra celda, esta posición debe calcularse tomando los valores congruentes a la fila y columna.

**Z** es la función que define el acoplamiento entre cada celda. Esta función traduce la salida de cada celda $C_{ij}$ en entrada de otra celda $C_{kl}$ usando el port m-esimo.

**select** es la función que permite aplicar un criterio frente a dos celdas que deben ejecutar a la misma hora. Al ejecutar cada celda independientemente de la otra ésta es una situación posible.

## 4.4 Modelos celulares de tres estados

Hasta el momento hemos analizado los modelos CELLS-DEVS que toman valores binarios. En muchos casos es útil contar con un tercer valor que nos permita identificar situaciones desconocidas. Para ello incorporaremos un tecer valor llamado *indefinido* y se nota con el símbolo **?** ó ⊥. Este permite identificar una celda cuyo valor binario no puede determinarse. Este valor amplía nuestra lógica binaria a lógica trivalente ampliando las operacines lógicas de la siguiente forma:

| AND | T | F | ? |
|---|---|---|---|
| **T** | T | F | ? |
| **F** | F | F | F |
| **?** | ? | F | ? |

| OR | T | F | ? |
|---|---|---|---|
| **T** | T | T | T |
| **F** | T | F | ? |
| **?** | T | ? | ? |

| NOT | |
|---|---|
| **T** | F |
| **F** | T |
| **?** | ? |

Los modelos de este tipo (**T**ree **S**tates **C**ell **D**evs) se definen formalmente como:

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

donde:

**I** = $<\eta, P^x, P^y>$ : Definición de la interfaz del modelo, donde
- $\eta \in \boldsymbol{N}$ es el tamaño del vecindario
- $P^x$ es un port de entrada que acepta solamente los tres valores definidos.
- $P^y$ es un port de salida que acepta solamente los tres valores definidos.

**X** ={0, 1, ⊥}: Conjunto de eventos externos de entrada.

**Y** ={0, 1, ⊥}: Conjunto de eventos externos de salida.

**delay** ={transport, inertial, none}: Demora asociada al modelo.

**S** = Conjunto de estados donde

- Si la demora es de tranporte

  **S** ={(s, phase, $\sigma_{queue}$, σ)}: donde

  $s \in \{0, 1, \perp\}$

  phase ∈ {active, passive}

  $\sigma_{queue} = \{ (v_i, \sigma_i) / (\forall i: i \in [1,m] \wedge i \in \boldsymbol{N} : \sigma_i \in \boldsymbol{R_0}^+ \cup \infty \wedge v_i \in \{0, 1\} \}$

  $\sigma \in \boldsymbol{R_0}^+ \cup \infty$

- Si la demora es de inercial

  **S** ={(s, phase, σ)}: donde

  $s \in \{0, 1, \perp\}$

  phase ∈ {active, passive}

  $\sigma \in \boldsymbol{R_0}^+ \cup \infty$

**N** ={ $\{0, 1, \perp\}^{\eta}$ }: Valores de los vecinos.

$\mathbf{d} \in \boldsymbol{R_0}^+$: Demora para cada celda.

$\boldsymbol{\delta_{int}}$: S→S: Función de transición interna.

$\boldsymbol{\delta_{ext}}$: QxX→S: Función de transición externa donde Q es el estado definido por Q={ (s, e) / $s \in S \wedge e \in [0, D(s)]$}

**τ**: N→N: Función de transición local.

**λ**: S→*Y:* Función de salida.

**D**: $S \rightarrow \boldsymbol{R_0}^+ \cup \infty$: Función de tiempo de vida del estado.

La definición de los modelos celulares con tres estados es similar a la hecha para los modelos celulares con dos estados posibles. El tipo de demora indica una de las demoras especificadas anteriormente, ésta está asociada a las funciones de transición interna, externa y la función de salida aplicando los algoritmos descritos en los puntos anteriores. Dependiendo de la demora se utilizarán unos u otros. El valor de los vecinos es guardado como una upla de dimension $\eta$ de los tres valores posibles para cada vecino. Estos valores son utlizados por la función de cómputo local para obtener el nuevo estado de la celda.

Los modelos acoplados celulares para este tipos de celdas son equivalentes a los anteriores ampliando los valores binarios en los puertos y conjuntos de entrada/salida para que incluyan el valor indefinido.

# 5 Implementación de modelos CELLS-DEVS

## *5.1 Introducción*

Debido a que la simulación se ha convertido en una metodología muy utilizada en estos tiempos, uno de nuestros objetivos será el desarrollo de herramientas que apliquen el formalismo DEVS y faciliten su empleo.

La construcción del modelo completo que se desea simular deberá poder especificarse describiendo uno a uno los componenetes en forma jerárquica y su interrelación. Generar los modelos en forma jerárquica es de mucha utilidad ya que esto permite la creación y prueba de los mismos en forma mucho más simple. A su vez esto permite definir responsabilidades claras favoreciendo así la reusabilidad de los modelos (que han sido verificados) en otras construcciones de mayor complejidad.

Una herramienta con estas características ampliará el horizonte de acción del campo de la simulación extendiendo su aplicación a usuarios que nunca hubieran podido tener acceso a estos temas debidoa su complejidad de implementación. Esto favorecerá el florecimiento de nuevas ideas y soluciones que permitirán encontrar nuevos caminos hacia horizontes inimaginados.

El lenguaje seleccionado fue C++, ya que además de cumplir con todos estos requisitos, nos permitirá realizar un diseño/programación orientado a objetos, con todos los beneficios que esto implica. La versión a utilizar es la ANSI/ISO C++ draft standard, publicado en Diciembre 96 (XJ316) por el ANSI/ISO comité para la estandarización de C++ (http://www.iso.org).

El compilador a utilizar será cualquiera que cumpla los requerimientos antes descriptos. Hemos seleccionado el compilador de C++ gcc de GNU, ya que su carácter de “distribución libre” (G.P.L. General Public License) y su adaptación al estándar lo hacen el ideal para este tipo de emprendimiento.

## *5.2 Diseño*

Para llevar la simulación a cabo existen dos grandes responsabilidades, administrar los datos necesarios por cada modelo y administrar el mecanismo de simulación. La clase *Model* es la clase base abstracta de todos los modelos y contiene todos los datos necesarios por un modelo. La clase *Procesador* es la clase base abstracta responsable de aplicar el mecanismo de simulación deseado. Estas dos clases deben conocerse ya que por cada modelo existirá un procesador que administrará los mensajes y aplicará las funciones que correspondan. Esto se ve reflejado en la asociación que existe entre *Processor* y *Model*.

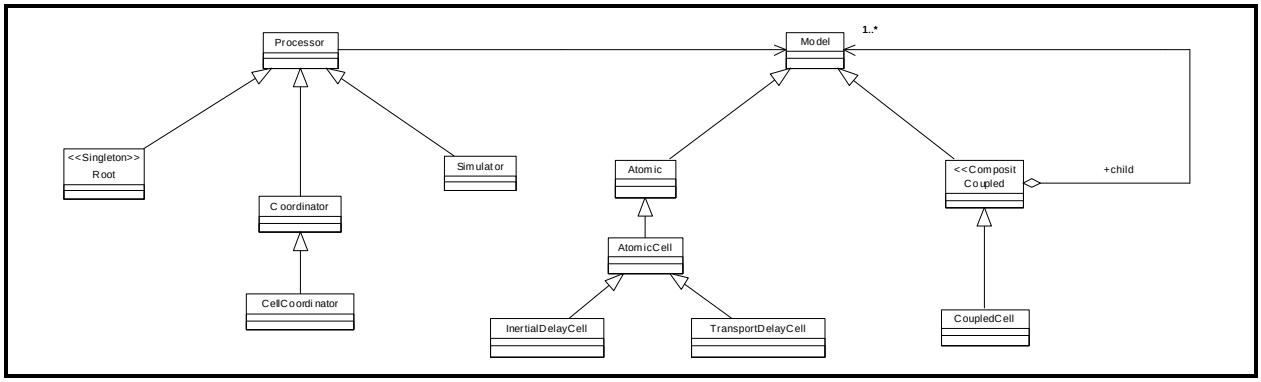



**Ilustración 7 – Jerarquía de Modelos y Procesadores**

### 5.2.1 Modelos

La jerarquía de modelos está compuesta por las siguientes clases *Atomic*, *Coupled* son las principales, respresentan a los modelos básicos y acoplados respectivamente. Para los modelos celulares se agregan las clases *AtomicCell* y sus derivadas *TransportDelayCell* e *InertialDelayCell* como extensiones de modelos atómicos y como extensiones de modelos acoplados se agrega *CellCoupled*.

Las principales responsabilidades de estas clases son:

Model

Es la clase base abstracta para todos los modelos, es responsable de:

- Administrar puertos de entrada.
- Administrar puertos de salida.
- Conocer la hora del próximo/anterior evento.
- Conocer el identificador del modelo.
- Conocer al padre del modelo.

Atomic

Es la clase base abstracta para todos los modelos atómicos, es responsable de:

- Permite cambiar la programación del modelo.
- Permitir cambiar el estado del modelo.
- Proveer interfaz para la Inicialización.
- Proveer interfaz para la función de Transición Interna.
- Proveer interfaz para la función Transición Externa.
- Proveer interfaz para la función de Salida.

Coupled

Representa a los modelos acoplados, utiliza el pattern Composite [Gamm95] para lograr una estructura recursiva de conjunto de modelos.

- Incorporación y manejo de componentes.
- Registración de dependencias entre los componentes hijos.

AtomicCell

Es la clase base abstracta para todos los modelos atómicos celulares (celdas), es responsable de:

- Conocer la función de transición local asociada.
- Conocer el vecindario y obtener el valor de sus vecinos.
- Conocer el valor de la celda.
- Conocer el puerto de entrada, el puerto de salido, y el puerto por el cual se comunican los vecinos.

- Responder a la función de inicialización recalculando el estado de la celda.
- Ejecutar la función de cálculo local.
- Responder al pedido de salida enviando el valor de la celda por el puerto de salida.

TransportDelayCell

Representa a las celdas que utilizan la demora de transporte, redefine el comportamiento de las funciones de transición externa, interna y de salida para aplicar este algoritmo. Es responsable de:

- Implementar los algoritmos de demora de transporte en las funciones de transición externa e interna.

InertialDelayCell

Representa a las celdas que utilizan la demora inercial, redefine el comportamiento de las funciones de transición externa, interna y de salida para aplicar este algoritmo. Es responsable de:

- Implementar la especificación de demora inercial.

CoupledCell

Representa a los modelos celulares acoplados, es responsable de:

- Conocer el tamaño de la grilla de celdas.
- Conocer el tipo de demora para las celdas.
- Conocer el tiempo de demora por defecto para las celdas.
- Conocer el tipo de borde.
- Conocer el valor inicial por defecto para cada celda.
- Conocer el función de transición local por defecto para las celdas.
- Conocer la función de transición local por defecto para las celdas y las regiones definidas con otro comportamiento.
- Crear el conjunto de celdas y los vínculos entre ellas.
- Cargar el valor inicial especificado para cada celda.

### 5.2.2 Procesadores

La jerarquía de procesadores está compuesta por las clases *Processor* y sus inmediatos sucesores *Root, Simulator* y *Coordinator*. La clase *Root* representa a la raíz del árbol de componentes en una simulación. *Simulator* y *Processor* son los procesadores responsables de simular a los modelos atómicos y acoplados respectivamente. Para los modelos celulares se agregan *CellCoordinator*, que está asociado a un modelo celular acoplado.

Processor

La clase *Processor* es responsable de:

- Recibir mensajes (Inicialización, Evento Externo, Evento Interno, Done).
- Conocer a su modelo asociado.
- Conocer a su procesador padre.
- Enviar mensajes de salida hacia su progenitor.

Root

La clase *Root* es el único procesador que no tiene modelo asociado. Esta clase aplica el pattern **Singleton** [Gamm95] ya que existe una sola instancia en toda la simulación y debe ser conocida públicamente. Es responsable de:

- Administrar los eventos externos.
- Avanzar la hora de simulación.
- Iniciar y detener la simulación.
- Generar la salida de la simulación en función de los mensajes de salida recibidos.

Simulator

La clase *Simulator* está asociada a los modelos atómicos. Es responsable de:

- Recibir mensajes y canalizarlos hacia el modelo atómico asociado.
- Enviar mensajes hacia su procesador padre indicando la hora del próximo evento de su modelo asociado.

Cuando arriba un mensaje externo invocará a la función de transición externa de su modelo asociado enviando luego un mensaje *done* en respuesta al padre con la nueva planificación del modelo asociado. Frente a un mensaje interno invocará primero a la función de salida del modelo y luego a la de transición interna. Nuevamente un mensaje de respuesta *done* es enviado al procesador padre indicando un posible cambio en la planificación.

Coordinator

La clase *Coordinator* está asociada a los modelos acoplados. Es responsable de:

- Redireccionar los mensajes de inicialización hacia todos sus hijos.
- Redireccionar los mensajes externos por medio de las influencias de los puertos. Para esto analiza su esquema de acoplamiento interno.
- Redireccionar los mensajes de salida por medio de las influencias de los puertos y el esquema de acoplamiento externos reenviando los mensajes hacia los hijos o hacia afuera a modelos externos.
- Redireccionar los mensajes internos hacia el Processor inminente.
- Enviar a su procesador padre la hora del próximo evento interno.

Cuando arriba un mensaje externo el coordinador utiliza la información que le provee el modelo acoplado asociado y determina los modelos influenciados por este mensaje para enviarles un mensaje externo a cada uno de ellos. Para enviar la respuesta al padre esperará el mensaje *done* de cada uno de sus hijos y luego tomará la hora de su hijo inminente para utilizarla en él. Frente a un mensaje interno enviará un mensaje interno al procesador asociado al modelo inminente. Una vez más se tomará la hora del hijo inminente para enviar una respuesta al padre.

Un mensaje de salida producirá un análisis de las influencias del puerto por el cual arribo el mensaje. Para ello utiliza al modelo acoplado asociado y determina si este mensaje debe ser reexpedido como mensajes externos a modelos que son parte del acoplado o hacia modelos externos como mensaje de salida.

CellCoordinator

La clase *CellCoordinator* está asociada a los modelos acoplados celulares, es responsable de:

- Administrar los hijos inminentes con una política específica para los modelos celulares.
- Administrar los mensajes de salida con una política específica para los modelos celulares.

El coordinador celular aprovecha los métodos de su clase base redefiniendo la política de selección de hijo inminente y el tratamiento de los mensajes de salida para evitar los mensajes repetidos a las celdas ya que estos solamente alertan una necesidad de recálculo y no un valor externo específico como en el resto de los modelos.

### 5.2.3 Mensajes

Todos los procesadores intercambian mensajes indicando el tipo de evento que deben aplicar en su modelo asociado, por ello existe una jerarquía de mensajes con *Message* como clase base y *InitMessage, ExternalMessage, InternalMessage, OutputMessage y DoneMessage* para representar a los eventos de inicialización, externos, internos de salida y done respectivamente. Los procesadores tienen como responsabilidad tomar decisiones en función del mensaje que arribe, sin embargo no son responsables

del mecanismo de pasaje de mensajes. Para esto existe un administrador de mensajes. Esta separación favorece la independencia de las clases y permite, por ejemplo, realizar una implementación distribuida sin impactar en el código.

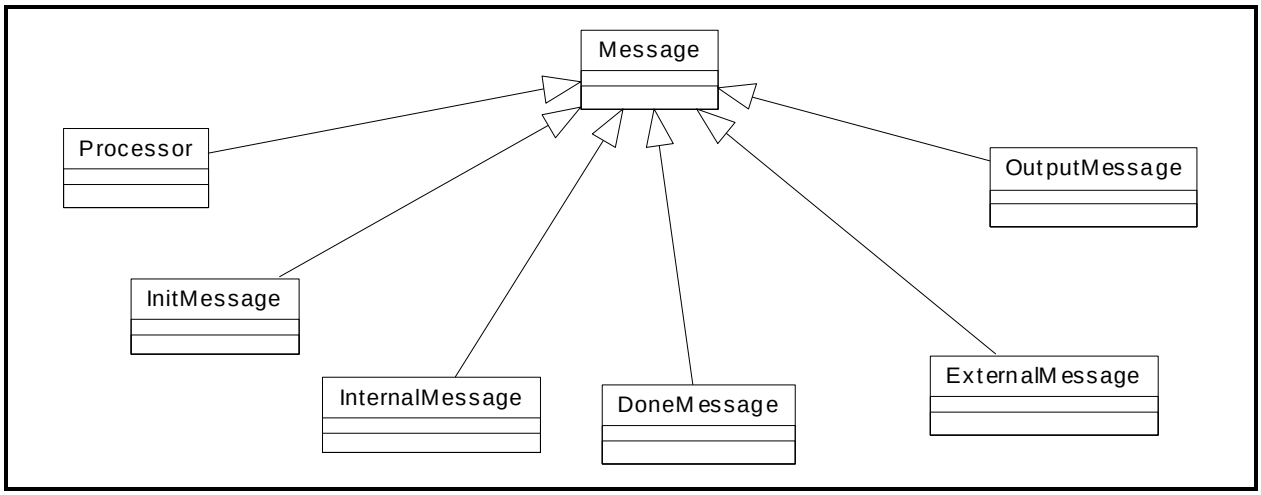



**Ilustración 8 - Jerarquía de mensajes**

Message

La clase *Message* representa la clase base abstracta para los mensajes, es responsable de:

- Conocer el procesador originador del mensaje.
- Conocer la hora en que es enviado el mensaje.

InitMessage

La clase *InitMessage* representa al mensaje de inicialización, no posee más responsabilidades que la clase base, solamente es utilizada para identificar el tipo de mensaje.

ExternalMessage

La clase *ExternalMessage* extiende a la clase *Message*. Representa el mensaje que indica un evento externo, se corresponde con el mensaje X entre procesadores.

- Conocer el puerto por el cual llega el mensaje.
- Conocer el valor del mensaje.

InternalMessage

La clase *InternalMessage* representa al mensaje interno (*), no posee más responsabilidades que la clase base, solamente es utilizada para identificar el tipo de mensaje.

DoneMessage

La clase *DoneMessage* representa el mensaje que reciben los procesadores de los hijos indicando la hora en la cuál tendrán su próximo cambio de estado, se corresponde con el mensaje D.

- Conocer la hora del próximo cambio de estado.

OutputMessage

La clase OutputMessage extiende a la clase Message. Representa a los mensajes de salida, se corresponde con los mensajes Y.

- Conocer el puerto por el cual sale el mensaje.
- Conocer el valor del mensaje.

### 5.2.4 Entorno de simulación

La clase *MainSimulator* es la responsable de crear el árbol de modelos/procesadores y los vínculos entre sus puertos a partir de una especificación.

La carga de modelos se realiza a partir de la definición de un modelo Root, es por esto que si no se encuentra ninguna etiqueta en la especificación de modelos que indique la definición de Root (top) el simulador aborta mostrando el error correspondiente.

Una vez armado el entorno de simulación la instancia de *Simulator* le informa a la instancia de *Root* los eventos externos especificados y la hora a la cual debe detener la simulación (por defecto es infinito).

En este momento se le solicita a *Root* que comience la simulación y ésta finaliza cuando llegue la hora de terminación o todos los modelos que componen la simulación se encuentre en estado pasivo y no se esperen nuevos eventos externos. La clase *MainSimulator* es responsable de:

- Creación del árbol de modelos/procesadores en base a la especificación obtenida a través de la clase Ini.
- Creación y vinculación de los puertos de todos los modelos que pertenezcan a la especificación.
- Creación de celdas para todos los modelos celulares.
- Registro de estados iniciales para todos los modelos celulares.
- Registro de funciones de transición para todas las celdas de todos los modelos celulares.

### 5.2.5 Lenguaje de Especificación GADCelLa

Para especificar el comportamiento de los modelos celulares atómicos se utilizan reglas que permiten definir un valor para la celda condicionado a una expresión lógica, utilizando como variables de entrada a los estados de los vecinos de la celda.

Si el acoplado celular posee un borde no circular, el estado de los vecinos de las celdas del borde es indefinido. Este es un caso donde se aplican las reglas de lógica de tres estados explicadas anteriormente. El lenguaje de especificación deberá ser capaz de evaluar expresiones lógicas con tres valores posibles, verdadero, falso e indefinido.

La definición de la especificación del comportamiento se realiza por separado, permitiendo la asociación de un lenguaje a más de un modelo celular. Esto permite la reusabilidad de un cierto comportamiento sin el tedio de la recodificación, ya que en un modelo celular es probable que existan rangos de celdas con distinto comportamiento.

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.

Estas reglas codificadas en GADCelLa se escriben de la siguiente forma:

```
rule : 1 1000 { (0,0) = 1 and (truecount = 3 or truecount = 4 ) }
rule : 1 1000 { (0,0) = 0 and truecount = 3 }
rule : 0 1000 { t }
```

Aquí puede verse que cada regla se especifica indicando el valor resultado deseado (1, 0 o ? para verdadero, falso e indefinido respectivamente), el tiempo de demora en milisegundos y por último la

expresión lógica que debe ser satisfecha para tomar esta regla como válida. Si ninguna regla puede ser satisfecha la herramienta marcará esta situación como un error indicando los detalles del vecindario que produce tal situación. Si existe más de una regla válida se toma la primera de ellas.

El orden de evaluación de las reglas respeta la secuencia indicada en la especificación. Es por esto que la última regla en el ejemplo, anterior especifica como resultado cero considerando la expresión verdadero como condicionante. Esta expresión satisface cualquier estado posible de las celdas vecinas y será evaluada solamente si las expresiones de las reglas anteriores no pueden ser satisfechas.

Es importante aclarar que las funciones *TrueCount*, *FalseCount* y*UndefCount* (cuentan la cantidad de celdas vecinas con su estado en verdadero, falso e indefinido respectivamente) se ajustan a la definición de vecindario especificada para el modelo donde se está aplicando la regla. Es por ello que para la codificación de una regla debe tenerse en cuenta si la propia celda pertenece o no al vecindario, ya que la cantidad de celdas con cierto estado, ahora la estarían incluyendo. Por lo general es frecuente que la celda pertenezca al vecindario ya que habitualmente utiliza su propio valor para el cálculo de su próximo estado, y por lo tanto se autoinfluencia.

La sintaxis del lenguaje puede definirse con la siguiente BNF:

```
Rule       := Bool Int '{' BoolExp '}'
BoolExp    := IntRelExp | NOT BoolExp | BoolExp AND BoolExp | BoolExp OR BoolExp
IntRelExp  := IdRef | IntExp OpRel IntExp
IntExp     := IdRef | IntExp Oper IntExp
IdRef      := CellRef | '(' BoolExp ')' | Constant | Function
Constant   := Int | Bool
Function   := TRUECOUNT | FALSECOUNT | UNDEFCOUNT
CellRef    :=  '(' IntExp ',' IntExp ')'
OpRel      := = | != | > | < | >= | <=
Oper       := + | - | * | /
Int        := [Sign] Digit {Digit}
Bool       := 0 | 1 | t | f | ?
Sign       := [+] | -
Digit      := 0 | 1 | ... | 9
```

# 6 Modelos CELLS-DEVS achatados

En los modelos acoplados, la interacción entre hijos y padres se logra vía intercambio de mensajes entre los procesadores correspondientes. Este circuito de comunicación incrementa la sobrecarga de la simulación. Siendo éste un factor fundamental que impacta directamente en el rendimiento del simulador, es importante hallar una alternativa que permita evadir estos mecanismos de comunicación para conseguir el mismo objetivo.

Uno de los puntos que pueden favorecer la aparición de una nueva técnica es el análisis de las características que poseen los modelos celulares. Para esto realizaremos un estudio de las características que distinguen a los modelos celulares del resto de los modelos acoplados. Las principales diferencias son:

- Todos los componentes poseen un estado interno del mismo tipo.
- Todos los componentes tienen el mismo comportamiento.
- Todos los componentes poseen los mismos puertos de entrada / salida.
- Definiendo los vecinos como una lista de desplazamientos, para todos los componentes puede calcularse sus vecinos absolutos.
- Todos los componentes tienen las mismas relaciones de influencia con sus vecinos.

Estudiando todos estos factores se puede hacer una distinción clara entre estructura y comportamiento. Esta separación nos permite replicar la estructura por cada instancia de celda reutilizando el comportamiento a medida que sea necesario.

Un modelo celular chato está compuesto por una matriz de estructuras que representan el estado de las celdas, la definición del vecindario y la función de cálculo local que define el comportamiento de la celda.

El modelo se incorpora al entorno de simulación respondiendo a los mismos mensajes a los que responde un coordinador.

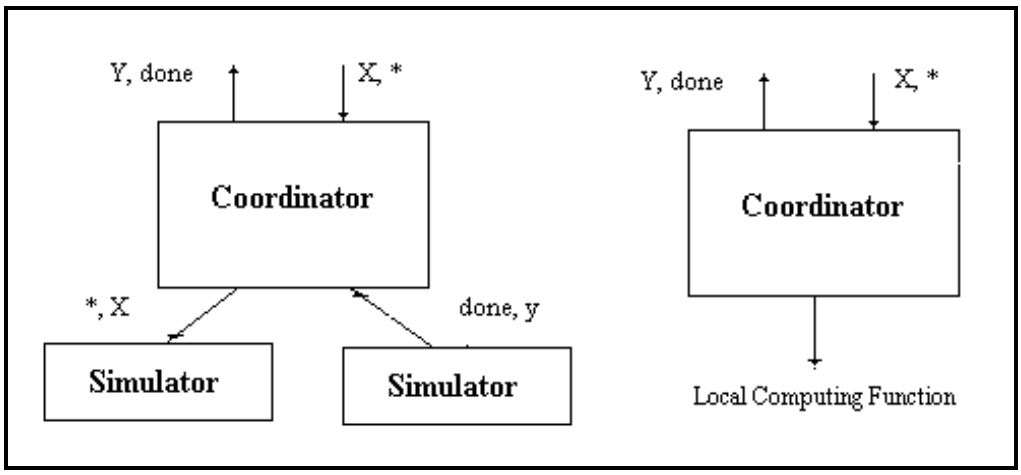



**Ilustración 9 – Modelo Acoplado vs Modelo plano**

Frente a un mensaje externo, resolverá cuáles son las celdas influenciadas por este mensaje, utilizando la estructura XList al igual que lo realiza un modelo acoplado celular convencional. Determinadas las celdas influenciadas en lugar de enviarle un mensaje externo a cada una de ellas, se invoca directamente la función de transición local invocándola con el estado y vecindario que le corresponde a cada una de ellas. Si el valor obtenido es distinto al que poseía originalmente, se aplicará la política de demora de acuerdo a la especificación del modelo y si corresponde se reprogramará a la celda virtual. Una vez finalizado esto para todas las celdas influenciadas, el coordinador achatado finaliza enviando un mensaje *done* al padre con la hora del próximo cambio de estado para la celda virtual inminente.

Un mensaje interno le indica al coordinador que al menos una de las celdas debe realizar su cambio de estado. Para todas las que corresponda se producirá la salida correspondiente, reiterando el análisis de influencias realizado para los eventos externos. Una vez finalizado el tratamiento de los eventos internos, para todas las celdas influenciadas por las salidas recientes se efectúa el mismo tratamiento que para los eventos externos, generando así nuevas reprogramaciones y en consecuencia una nueva hora de próximo cambio de estado para el modelo padre.

Este tratamiento de los eventos elimina por completo los mensajes entre el acoplado celular chato y sus hijos ya que sus hijos son virtuales y no existen como modelos. A su vez los mensajes de respuesta se reducen considerablemente debido a que, ante cada mensaje recibido, se efectúan todas las tareas posibles que correspondan al conjunto de celdas.

Estos factores reducen el tiempo total de ejecución, logrando obtener los mismos resultados en aproximadamente la mitad del tiempo y con menor requerimiento de recursos (principalmente memoria). A su vez también se obtienen grandes beneficios en el tiempo de carga y configuración del simulador, ya que este último no requiere la creación de grandes cantidades de modelos, tarea que consume un importante tiempo en modelos celulares de tamaño considerable.

# 7 Ejemplos de modelos celulares

## 7.1 Juego de la vida

El juego de la vida se desarrolla sobre una matriz de mxn valores binarios. Cuando una celda tiene el valor uno se dice que esta viva, en otro caso (valor cero) está muerta. El juego se desarrolla en base a una especificación inicial de celdas vivas y muertas. A partir de aquí comienzan las iteraciones aplicando las siguientes reglas:

- Una celda viva permanecerá viva si tiene cuatro vecinos vivos de otro modo fallece.
- Una celda muerta revivirá si tiene exactamente tres vecinos vivos.

Este juego se modela utilizando autómatas celulares especificando el estado inicial y luego las reglas común a todas las celdas y con la misma demora. La especificación es la siguiente:

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 1000
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 :  5  00000001110000000000
initialrowvalue :  7  00000100100100000000
initialrowvalue :  8  00000101110100000000
initialrowvalue :  9  00000100100100000000
initialrowvalue : 11  00000001110000000000
localtransition : conrad-rule

[conrad-rule]
rule : 1 1000 { (0,0) = 1 and (truecount = 3 or truecount = 4 ) }
rule : 1 1000 { (0,0) = 0 and truecount = 3 }
rule : 0 1000 { t }
```

## 7.2 Mall

El objetivo de este ejemplo es ilustrar el uso de modelos atómicos, acoplados y celulares relativamente simples que combinados correctamente pueden convertirse en un sistema complejo mucho mas difícil de especificar, diseñar y simular.

El ejemplo a estudiar centra su atención en la simulación del trafico en una zona urbana y su relación con la contaminación de humo en barrios residenciales. Para esto construiremos cinco modelos, cada uno con un comportamiento específico y de tipología diferente: *factory*, *Ferry Boat*, *Comercial Neighborhood* , *Highway* y *Residential Neighborhood*.

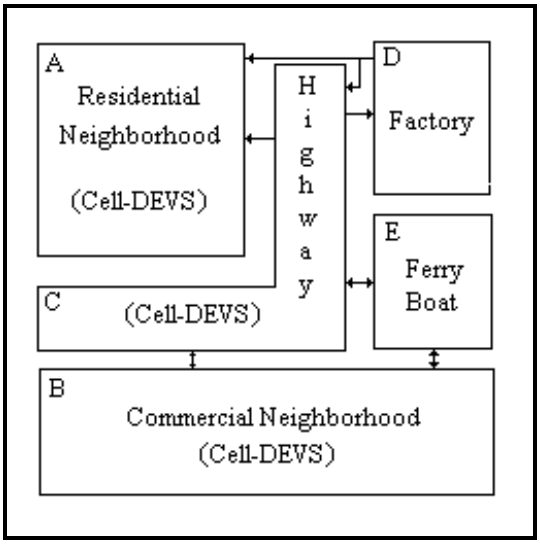



**Ilustración 10 - Acoplamiento de Modelos Celulares, Acoplados y Atómicos**

De la integración de estos cinco modelos se podrán obtener simulaciones que permitan el estudio de cada uno en particular pero bajo un marco general de integración.

El subsistema **A**, Residential Neighborhood, es un modelo celular que centra su objetivo en el estudio de la polución en un barrio residencial. Cada celda del autómata representará la existencia o ausencia de partículas de smog en el aire. El barrio modelado a estudiar tiene dos potenciales entradas de smog: una del trafico proveniente de la autopista y la otra del humo generado por una fábrica cercana.

El subsistema **B**, *Commercial Neighborhood*, también es un modelo celular que representa la circulación del trafico en una zona comercial cuya característica son las calles que poseen una sola dirección y no hay ningún semáforo instalado.

El subsistema **C**, *Highway*, representa una autopista de tránsito rápido de una sola dirección. Esta autopista atraviesa, a lo largo de su recorrido, por los vecindarios comercial y residencial modelados por los sistemas **A** y **B**.

El subsistema **D**, *Factory*, modela una fábrica a la cual llegan y salen camiones en forma continua. En este caso los resultados de la simulación pueden utilizarse para optimizar la programación de los camiones logrando mejoras en el flujo de los productos, minimizando la espera de cada uno de ellos, y de esta forma minimizando los efectos del smog en otras áreas ya que no se producirán (o al menos se minimizarán) los congestionamientos.

El subsistema que representa el comportamiento de la fábrica es un modelo acoplado que está compuesto por dos modelos atómicos: una cola y un procesador. La cola representa la playa de estacionamiento donde los camiones esperan para ser atendidos. El procesador es el operario que realiza la carga/descarga sobre el camión.

Por último, el subsistema **E**, *Ferry Boat*, modela el comportamiento de una compañía de barcos de traslado de automóviles que provee conexión con una isla vecina. En este caso, los resultados de la simulación podrán utilizarse para optimizar la frecuencia del servicio de pendiendo la hora del día, minimizando tanto los tiempos de espera de los clientes y maximizando la ocupación de los barcos en cada uno de sus viajes.

```
[top]
components : HighWay Residential Mall FerryBoat Factory
link : out1@HighWay in2@Residential
link : out2@HighWay in@Factory
link : out3@HighWay in1@Mall
```

```
link : out4@HighWay in@FerryBoat
link : out@Factory in1@Residential
link : out@Factory in2@HighWay
link : out1@FerryBoat in2@Mall
link : out2@FerryBoat in4@HighWay
link : out1@Mall in3@HighWay
link : out2@Mall in@FerryBoat


[Factory]
components : parking@queue working@CPU
in : in
out : out
Link : in in@parking
Link : out@parking in@working
Link : out@working out
Link : out@working done@parking

[parking]
preparation : 0:0:1:0

[working]
distribution : normal
mean : 80
deviation : 10


[FerryBoat]
components : fromManhatan@Generator fromNY@Generator
in : in
out : out1 out2
Link : out@fromManhatan out1
Link : out@fromNY out2
Link : in in@fromManhatan
Link : in in@fromNY

[fromManhatan]
distribution : chi
degreesfreedom : 4
initial : 1
increment : 0

[fromNY]
distribution : poisson
mean : 12
initial : 1
increment : 0


[Residential]
type : cell
width : 9
height : 10
delay : inertial
defaultDelayTime : 3000
border : nowrapped
neighbors :                CellCoupled(-1,0)
neighbors : CellCoupled(0,-1) CellCoupled(0,0) CellCoupled(0,1)
neighbors :                   CellCoupled(1,0)
initialValue : 0
in : in1 in2
link : in1 in@Residential(2,8)
link : in2 in@Residential(7,8)
localtransition : ResidentialSmog

[ResidentialSmog]
rule : 1 3000 { (0,0) = 1 and TrueCount > 1  }
rule : 1 3000 { (0,0) = 0 and ( (-1,0)=1 or (0,-1)=1 or (0,1)=1 or (1,0)=1 ) }
rule : 0 3000 { TrueCount = 1 and (0,0)=1 }
rule : 0 3000 {t}
```

```
[Mall]
type : cell
width : 50
height : 10
delay : transport
defaultDelayTime : 5000
border : nowrapped
initialValue : 0
neighbors :                CellCoupled(-1,0)
neighbors : CellCoupled(0,-1) CellCoupled(0,0) CellCoupled(0,1)
neighbors :                   CellCoupled(1,0)
in : in1 in2
out : out1 out2
link : in1 in@Mall(0,20)
link : in2 in@Mall(0,45)
link : out@Mall(0,20) out1
link : out@Mall(0,45) out2
localTransition : TransitMall


[TransitMall]
rule : 1 5000 { (0,0)=0 and (1,0)=1  }
rule : 1 5000 { (0,0)=0 and (1,0)=1  }
rule : 1 5000 { (0,0)=1 and (0,1)=1  }
rule : 1 5000 { (0,0)=1 and (1,0)=1  }
rule : 0 5000 { (0,0)=1 and (-1,0)=1  }
rule : 0 5000 { (0,0)=1 and (-1,0)=0  }
rule : 0 5000 { (0,0)=1 and (0,1)=0  }
rule : 0 5000 { (0,0)=0 and (1,0)=0 and (0,-1)=0  }
rule : 0 5000 { t }

[HighWay]
type : cell
width : 15
height : 4
delay : transport
defaultDelayTime : 2000
border : nowrapped
initialValue : 0
neighbors : CellCoupled(-1,-1) CellCoupled(-1,0) CellCoupled(-1,1)
neighbors : CellCoupled(0,-1)  CellCoupled(0,0)  CellCoupled(0,1)
neighbors : CellCoupled(1,-1)  CellCoupled(1,0)  CellCoupled(1,1)
in : in2 in3 in4
out : out1 out2 out3 out4
link : out@HighWay(2,0) out1
link : out@HighWay(3,2) out2
link : out@HighWay(3,13) out3
link : out@HighWay(3,10) out4
link : in2 in@HighWay(1,13)
link : in3 in@HighWay(3,13)
link : in4 in@HighWay(3,0)
localtransition : TransitHighWay

[TransitHighWay]
rule : 1 2000 { (0,0)=0 and (0,-1)=1 }
rule : 1 2000 { (0,0)=0 and (0,-1)=0 and (1,-1)=1 and (1,0)=1 }
rule : 1 2000 { (0,0)=0 and (0,-1)=0 and (-1,-1)=1 and (-1,0)=1 }
rule : 1 2000 { (0,0)=1 and (-1,1)=1 and (0,1)=1 and (1,1)=1 }
rule : 0 2000 { (0,0)=1 and (0,1)=0 }
rule : 0 2000 { (0,0)=1 and (0,1)=1 and (-1,1)=0 and (-1,0)=0 }
rule : 0 2000 { (0,0)=1 and (0,1)=1 and (1,1)=0 and (-1,1)=1 and (1,0)=0 }
rule : 0 2000 { (0,0)=0 }
rule : 0 2000 {t}
```

# 8 Resultados y comparaciones

Un objetivo primordial del nuevo formalismo, así como el de la herramienta desarrollada, reside en la reducción en los tiempos de desarrollo y mantenimiento de la simulaciones. Se registraron datos relacionados con los tiempos de desarrollo de las distintas soluciones, clasificándolas entre distintos tipos de usuarios. Los resultados obtenidos se resumen en las siguientes tablas:

Usuarios Novatos (Tiempo de aprendizaje de uso de la herramienta: 16 h/h)

| | Juego de la Vida | | Simulación de tráfico | |
|---|---|---|---|---|
| | Desarrollo | Testeo | Desarrollo | Testeo |
| Especificación de los primeros problemas | 0.75 h/h | 2.50 h/h | 2.50 h/h | 4.3 h/h |
| Especificación de los útlimos problemas | 0.25 h/h | 0.75 h/h | 1.00 h/h | 1.7 h/h |
| Mantenimiento | 0.25 h/h | 0.30 h/h | 0.25 h/h | 0.3 h/h |

Usuarios Expertos

| | Juego de la Vida | | Simulación de tráfico | |
|---|---|---|---|---|
| | Desarrollo | Testeo | Desarrollo | Testeo |
| Especificación de los primeros problemas | 0.50 h/h | 1.30 h/h | 0.50 h/h | 0.75 h/h |
| Especificación de los útlimos problemas | 0.10 h/h | 0.30 h/h | 0.30 h/h | 0.75 h/h |
| Mantenimiento | 0.25 h/h | 0.30 h/h | 0.25 h/h | 0.50 h/h |

Especificación, diseño y programación directa (autómatas celulares asincrónicos).

| | Juego de la Vida | | Simulación de tráfico | |
|---|---|---|---|---|
| | Desarrollo | Testeo | Desarrollo | Testeo |
| Especificación de los primeros problemas | 5 h/h | 9 h/h | 8 h/h | 11 h/h |
| Especificación de los útlimos problemas | 3 h/h | 5 h/h | 2 h/h | 3 h/h |
| Mantenimiento | 4 h/h | 12 h/h | 3 h/h | 7 h/h |

Los tiempos resultantes de las corridas comparando los modelos celulares chatos versus los no chatos puden verse en la siguiente tabla:

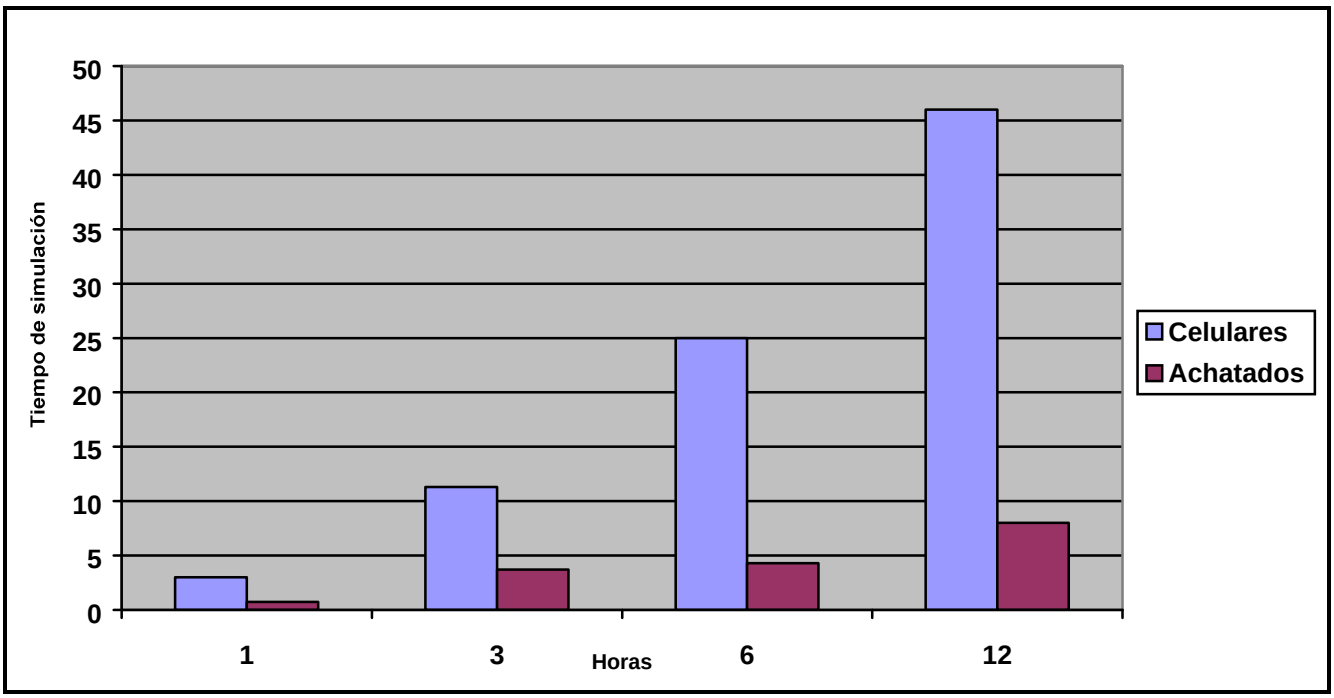



**Ilustración 11 - Comparación de achatados vs no achatados**

# 9 Conclusiones

La herramienta desarrollada provee un nuevo ambiente de desarrollo para la simulación de modelos utilizando DEVS.

Desde el punto de vista del usuario utilizar un lenguaje descriptivo para la especificación de los componentes de la simulación, las conexiones entre ellos y su configuración en general. Esta resultó ser una metodología de fácil acceso que exige una leve curva de aprendizaje como primer paso, pero luego, permite construir sistemas muy complejos con un costo ínfimo.

La utilización de estándares en el desarrollo del motor de simulación favoreció la portabilidad de la herramienta a distintas plataformas que conforman el estándar sin ningún costo. Este margen abarca desde una PC hogareña utilizando Windows95/NT pasando por Linux, AIX, HP-UX y Sun (plataformas testeadas con éxito hasta el momento) en sus distintas configuraciones de hardware. Esta característica permitió la disponibilidad de la herramienta en múltiples ámbitos de desarrollo llevando así la posibilidad de realizar simulaciones en grandes centros de cómputo hasta en una notebook. También incentivó a los usuarios a modelar y realizar simulaciones en su ámbito habitual para luego poder realizar estudios más profundos con máquinas de mayor poder de cómputo.

El motor de simulación provee una API (punto de acceso a los servicios de simulación) para la incorporación de nuevos modelos atómicos que ajustándose a los estándares bajos los cuales fue desarrollada la herramienta. Para lograr esto los desarrolladores debieron programar los nuevos modelos básicos utilizando C++ estándar. Para una minoría esto requirió un mínimo aprendizaje del lenguaje y programación orientada a objetos, sin embargo el resto se vió beneficiado por un lenguaje al cual ya estaba habituado, con muchísimas virtudes y de gran difusión en el ámbito científico desde hace al menos una década. Por otro lado no utilizar un lenguaje propietario de especificación permitió la disponibilidad de todas las características de un lenguaje orientado a objetos más la incorporación de rutinas y librerías con objetivos específicos potenciando así la diversidad y alcance de los nuevos modelos.

Se implementó una técnica de adaptación de los modelos celulares al formalismo DEVS. Esta fue incorporada en la herramienta de simulación permitiendo la incorporación de los modelos celulares como modelos acoplados con características especiales. Un lenguaje propietario de expresiones lógicas fue desarrollado para especificar el comportamiento de las celdas de los modelos celulares. Este lenguaje, llamado GADCelLa, permitió desarrollar simulaciones de modelos celulares con una gran facilidad y en muy poco tiempo debido a que la modificación de las reglas de comportamiento para las celdas no requiere tiempo de compilación ni de vinculación. De esta forma el desarrollo de nuevos modelos que incluyan componentes celulares se limita a la especificación del comportamiento de las celdas y su inmediata prueba minimizando así los tiempos de desarrollo y testeo.

Disponer de tres tipos de componentes DEVS, atómicos, acoplados y celulares cada uno con sus características, permite un crisol heteróclito de sistemas posibles para simular. Dentro de cada sistema encontraremos definiciones de modelos con responsabilidades claras. Esto permite extraer el componente deseado para reutilizarlo en otro sistema que necesite de sus servicios.

Implementamos los modelos acoplados celulares achatados que explotan una técnica para disminuir la sobrecarga que poseen los modelos acoplados celulares obteniendo los mismos resultados en una fracción de tiempo. Esta disminución fluctuó entre un tercio y un décimo del tiempo que consumió el acoplado celular estándar para realizar la misma tarea. Ya que la implementación esta relacionada con el pasaje de mensajes y esto es una característica interna, ni el desarrollador, ni el usuario se ven afectados por la utilización de este tipo de implementación particular ya que es completamente transparente a los modelos externos.

Habiendo desarrollado la herramienta con todas estas características y observando los beneficios que proveen el modelado jerárquico, la utilización del formalismo DEVS, los modelos celulares, el

rendimiento de los achatados y la facilidad de desarrollo se abre aquí la posibilidad para que en nuevos emprendimientos se implemente una versión distribuida que incorpore servidores de simulación que provean capacidades de base de datos de modelos utilizando todas las propiedades paralelizables que fueron descriptas a lo largo de este trabajo.

# Informe Técnico

## 10 Introducción

Para todos aquellos sistemas en los que no es posible o conveniente hallar soluciones analíticas se ha difundido el uso de metodologías y herramientas de simulación.

En la actualidad existe una gran variedad de aplicaciones complejas en las que se usan modelos y/o simulación, que van desde manufactura hasta diseño de circuitos para computadoras, pasando por aplicaciones bélicas y estudios de experimentos complejos. Las características comunes a estos sistemas son su complejidad y la falta de herramientas de evaluación de desempeño adecuadas.

Las ventajas de la simulación son múltiples: puede reducirse el tiempo de desarrollo del sistema, las decisiones pueden verificarse artificialmente, un mismo modelo puede usarse muchas veces, etc. La simulación es de empleo más simple que ciertas técnicas analíticas y precisa menos simplificaciones.

Debido a que la simulación se ha convertido en una metodología muy utilizada en estos tiempos, parte de nuestro objetivo fue el desarrollo de herramientas que faciliten su empleo, ampliando de esta manera sus potenciales aplicaciones tanto en el mundo comercial como en el ámbito científico. De esta forma, disponiendo de aplicaciones que permitan llevar a cabo la simulación de una gran variedad de modelos y sistemas, la aplicación de esta metodología será mucho más fácil de llevar a cabo y por ende su aplicación mucho más frecuente.

En concordancia con la creación de una herramienta genérica de simulación, otro de nuestros objetivos fue que la misma permita la construcción de modelos en forma jerárquica, es decir la posibilidad de combinar modelos simples para que el conjunto y la vinculación de los mismos conformen modelos mucho mas complejos, y que de otra manera seria mucho mas difícil de modelar y llevar a cabo su simulación.

La posibilidad de generar los modelos en forma jerárquica es de mucha utilidad ya que esto permite la creación y prueba de los mismos en forma mucho mas simple. Luego, al momento de crear sistemas complejos, la posibilidad de crearlos en combinación de otros modelos previamente verificados, asegura que si teóricamente la construcción jerárquica del modelo es correcta, el modelo en su conjunto será correcto, ya que cada uno de sus componentes también lo son.

De esta forma, disponiendo de una herramienta con estas características, podremos lograr que el campo de la simulación amplíe su horizonte de acción, extendiendo su aplicación a otros temas y favoreciendo el estudio de las metodologías y técnicas utilizadas hasta el momento en búsqueda de nuevas ideas y soluciones que permitan encontrar nuevos caminos hacia soluciones hoy desconocidas.

Con respecto a la implantación de esta herramienta, debemos considerar ciertos aspectos que favorecerán la aceptación de la misma, tales como el lenguaje y plataforma sobre la cual será realizada su programación.

Uno de los principales aspectos a considerar es que la herramienta pueda ejecutarse desde una simple PC con sistema operativo monousuario (por ejemplo windows) hasta en grandes equipos con múltiples procesadores y gran cantidad de memoria que den un marco adecuado a la simulación de sistemas extremadamente complejos. Teniendo en cuenta estas restricciones hemos preferido realizar el desarrollo sobre un lenguaje estándar que pueda ser compilado en múltiples plataformas sin ningún tipo de modificación.

El lenguaje seleccionado fue C++, ya que además de cumplir con todos estos requisitos, nos permitirá realizar un diseño/programación orientado a objetos, con todos los beneficios que esto implica. La versión a utilizar es la ANSI/ISO C++ draft standard, publicado en Diciembre 96 (XJ316) por el ANSI/ISO comité para la estandarización de C++ (http://www.iso.org).

El compilador a utilizar será cualquiera que cumpla los requerimientos antes descriptos. Hemos seleccionado el compilador de C++ gcc de GNU, ya que su carácter de "distribución libre" (G.P.L. General Public License) y su adaptación al estándar lo hacen el ideal para este tipo de emprendimiento. Otro gran beneficio de esta elección es que este compilador, distribuido bajo archivos fuentes, se encuentra actualmente disponible para muchos tipos de plataformas, desde simples PC con Windows95 o NT hasta prácticamente cualquier Plataforma RISC (Reduced Instruction Set Computer) con un sistema operativo Unix.

El simulador constara de dos partes: un motor de simulación y la programación especifica de los modelos que se quiera simular.

El software de simulación provee el motor de simulación junto con las estructuras que permiten configurar la especificación y asociación de los modelos que componen el sistema a simular. El mismo trabaja con una interfaz genérica de modelos atómicos, acoplados y celulares.

El comportamiento del los modelos celulares se especifica por medio de un archivo de configuración, mientras que para los modelos atómicos debe codificarse un módulo en lenguaje C++. Por esto es responsabilidad del programador generar los modelos atómicos correspondientes para llevar a cabo la simulación respetando los requerimientos del ambiente de simulación. Por ejemplo, si se desea simular una cola deberá programarse un modelo atómico con el comportamiento correspondiente y unirlo al motor de simulación (En el **Manual de desarrollador** se describe en detalle estos procedimientos).

El desarrollo del presente informe está dividido en dos áreas: la primera incluye la especificación de las clases participantes del desarrollo, explicando sus objetivos, métodos y colaboración entre ellas. La segunda parte abarcara varios ejemplos que ilustran el uso de la herramienta.

# 11 Jerarquía de Modelos

Dentro de la simulación los modelos son los encargados de describir el comportamiento deseado. Los modelos acoplados permiten aunar a otros modelos y conectarse entre ellos. Los modelos atómicos proveen la base para que los nuevos modelos puedan especificar nuevos criterios para la programación de los eventos internos y cambios de estado.

Todos ellos mantienen características en común y esto se ve plasmado en la clase *Model* que es la clase base abstracta y principal en la jerarquía de modelos. Esta clase es abstracta ya que provee solamente la interfaz para otras clases, nunca será instanciada.

Ahora veremos a todos los componentes de la jerarquía de modelos explicitando las responsabilidades y los métodos relevantes.

## *11.1 Model*

### 11.1.1 Responsabilidades:

- Administrar puertos de entrada.
- Administrar puertos de salida.
- Conocer la hora del próximo/anterior evento.
- Conocer el identificador del modelo.
- Conocer al padre del modelo.

### 11.1.2 Colaboradores:

- **SingleProcessorAdmin** : Referencia a los procesadores existentes.
- **Processor**: Salida del modelo.

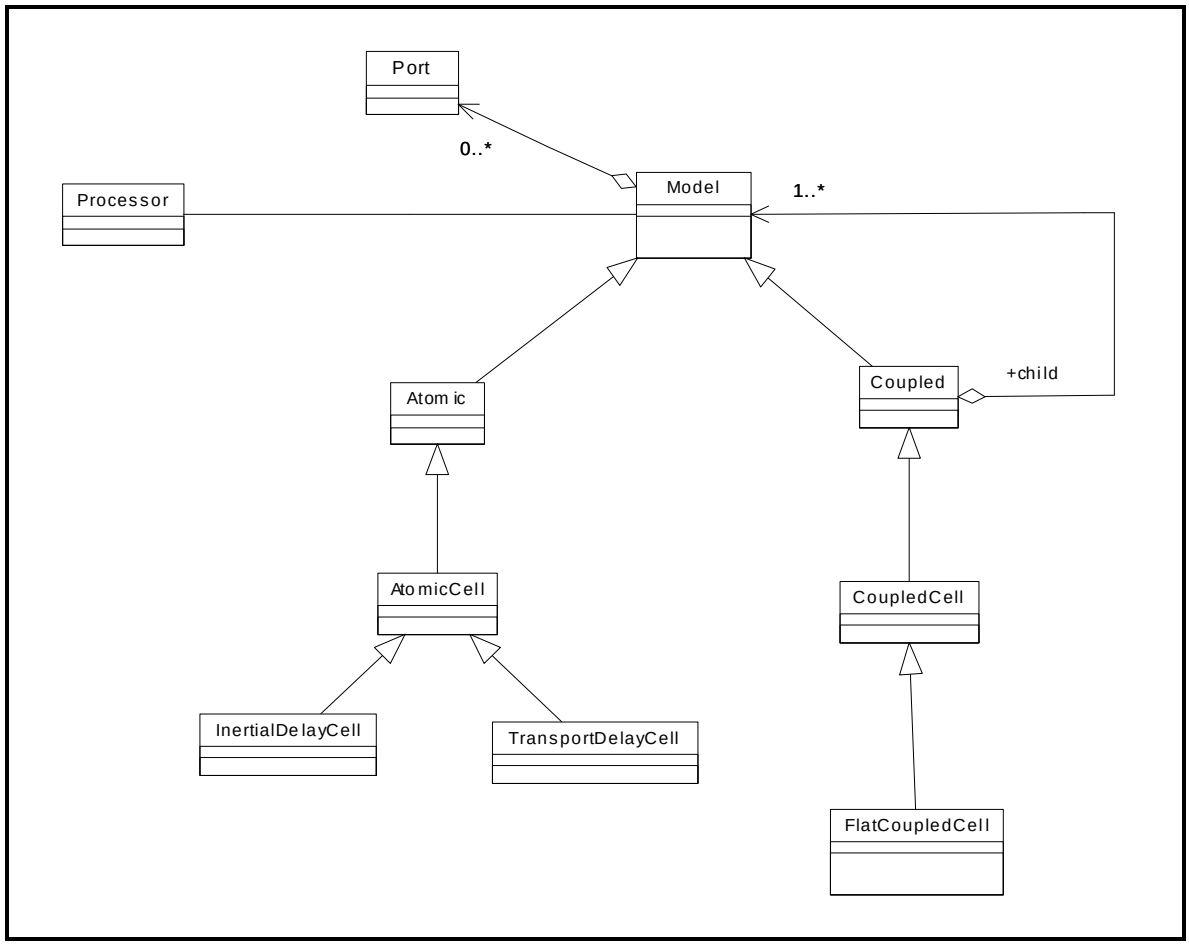



**Ilustración 12 - Jerarquía de modelos**

### 11.1.3 Métodos principales:

***addInputPort***

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

Descripción*:* Agrega un nuevo puerto de entrada a la lista de puertos de entrada del modelo. El parámetro representa el nombre del puerto que se desea crear. Esto permite identificar a este puerto por el nombre, por eso los nombres no pueden repetirse. De ser así al pedir la creación de

un nuevo puerto con un nombre existente se producirá un **InvalidPortException**. Si la operación se ha realizado con éxito retorna el nuevo puerto creado.

### ***addOutputPort***

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

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

### ***nextChange***

Signatura: `const Time &nextChange() const`

Descripción: Retorna el tiempo restante para el próximo cambio de estado (σ). Este método no modifica la instancia.

### ***lastChange***

Signatura: `const Time &lastChange()const`

Descripción: Retorna la hora del último cambio de estado. Este método no modifica la instancia.

### ***id***

Signatura: `const ModelId &id()const`

Descripción: Retorna el identificado unívoco del modelo. Este identificador es asignado cuando el modelo se crea.

### ***parent***

Signatura: `const Model &parent() const`

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

### ***nextChange***

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

Descripción: Modifica la hora de la próxima transición interna. El parámetro representa a qué ahora debe producirse. Este método solamente puede ser invocado por clases que extiendan a *Model* (protected).

### ***lastChange***

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

Descripción: Modifica la hora en la que se produjo la última transición interna. El parámetro representa la hora deseada. Este método solamente puede ser invocado por clases que extiendan a *Model* (protected).

### ***sendOutput***

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

Descripción: Envía la salida del modelo. Los parámetros especifican el puerto por el cual debe producirse la salida, el valor que debe llevar y la hora de esta. Este método solamente puede ser invocado por clases que extiendan a *Model* (protected).

### ***processor***

Signatura: `Processor &processor()`

Descripción: Retorna el procesador asociado al modelo. En caso de no tener procesador asociado se producirá un **InvalidProcessorIdException**.

## 11.2 Atomic

La clase *Atomic* es una especialización de la clase *Model* y representa la interfaz de un modelo básico. Como todo modelo hereda las facilidades para manejo de puertos, consulta de estado interno y planificación del próximo evento. Esta clase es *abstracta* ya que provee únicamente la interfaz para los modelos básicos.

### 11.2.1 Responsabilidades:

- Permite cambiar la programación del modelo.
- Permitir cambiar el estado del modelo.
- Proveer interfaz para la Inicialización.
- Proveer interfaz para la función de Transición Interna.
- Proveer interfaz para la función Transición Externa.
- Proveer interfaz para la función de Salida.

### 11.2.2 Métodos principales:

#### ***initFunction***

Signatura: `virtual Model &initFunction() = 0`

Descripción: Método abstracto que será invocado cuando se produzca la inicialización del modelo. Este método debe ser implementado obligatoriamente en las clases que extiendan a *Atomic*.

#### ***externalFunction***

Signatura*:* `virtual Model &externalFunction( const ExternalMessage & ) = 0`

Descripción: Método abstracto que será invocado cuando arribe un evento externo. El parámetro contiene el valor del evento (para más detalles ver Jerarquía de Mensajes). Dentro de esté método deberá indicarse por medio de un mensaje *Done* la hora del próximo evento interno.

Este método debe ser implementado obligatoriamente en las clases que extiendan a *Atomic*.

#### ***internalFunction***

Signatura: `virtual Model &internalFunction( const InternalMessage & ) = 0`

Descripción: Método abstracto que será invocado cuando arribe un evento interno. El parámetro contiene la hora y quién lo envía (para más detalles ver Jerarquía de Mensajes). Dentro de esta función deberá indicarse por medio de un mensaje *Done* la hora del próximo evento o pasar a estado pasivo.

Este método debe ser implementado obligatoriamente en las clases que extiendan a *Atomic*.

#### ***outputFunction***

Signatura: `virtual Model &outputFunction( const InternalMessage & ) = 0`

Descripción: Método abstracto que será invocado cuando deba producirse la salida del modelos. Esto debe hacerse por medio del método *sendOutput*.

Este método debe ser implementado obligatoriamente en las clases que extiendan a *Atomic*.

#### ***holdIn***

Signatura: `Model &holdIn( const State &, const Time & )`

Descripción: Cambia el estado del modelo al estado especificado en el parámetro y programa el próximo evento para la hora indicada en el segundo parámetro.

***passivate***

Signatura: `Model &passivate()`

Descripción: Cambia el estado del modelo a pasivo y la hora de próximo cambio en infinito.

***state***

Signatura: `Model &state( const State & )`

Descripción: Cambia el estado del modelo por el parámetro especificado.

***state***

Signatura: `const State &state() const`

Descripción: Retorna el estado actual del modelo. Este método no modifica el estado del objeto.

## 11.3 Coupled

La clase *Coupled* es una especialización de la clase *Model* y representa un modelo multi-componente, conocido como modelo acoplado. La clase *Coupled* aplica el pattern *Composite* [Gam95] ya que es un *Composite* de modelos. Esto permite la creación de grupos de modelos que a su vez contienen otros modelos.

### 11.3.1 Responsabilidades:

- Incorporación y manejo de componentes.
- Registración de dependencias entre los componentes hijos.

### 11.3.2 Colaboradores:

### 11.3.3 Métodos principales:

***addModel***

Signatura: `Model &addModel( Model & )`

Descripción: Agrega un nuevo modelo al acoplado. El parámetro indica el modelo que se debe agregar

***children***

Signatura: `const ModelList &children() const`

Descripción: Retorna la lista de modelos que conforman lo hijos del acoplado. Este método no modifica al objeto ni permite que se modifique la lista de hijos.

***addInfluence***

Signatura: `virtual Model &addInfluence( const string &sourceName, const string &sourcePort, const string &destName, const string &destPort)`

Descripción: Agrega una influencia entre dos de sus componentes hijos. En los parámetros se especifica el nombre del modelo y puerto origen y el nombre del modelo y puerto destino. Si uno de los modelos no pertenece a los hijos del acoplado esto producirá un **InvalidChildException**.

## 11.4 AtomicCell

La clase *AtomiCell* es una especialización de la clase *Atomic* y provee la interfaz para las celdas atómicas. Esta clase es asbtracta ya que no presenta ningún comportamiento.

### 11.4.1 Responsabilidades:

- Conocer la función de transición local asociada.
- Conocer el vecindario y obtener el valor de sus vecinos.
- Conocer el valor de la celda.
- Conocer el puerto de entrada, el puerto de salido, y el puerto por el cual se comunican los vecinos.
- Responder a la función de inicialización recalculando el estado de la celda.
- Ejecutar la función de cálculo local.
- Responder al pedido de salida enviando el valor de la celda por el puerto de salida.

### 11.4.2 Colaboradores:

- **SingleLocalTransAdmin**: Evalúa la función de cálculo local asociada a la celda.

### 11.4.3 Métodos principales:

#### ***value***

Signatura: `const TValBool &value() const`

Descripción: Retorna el valor que tiene la celda.

#### ***localFunction***

Signatura: `const LocalTransAdmin::Function &localFunction() const`

Descripción: Retorna el identificador de la función de cálculo local. Este valor es cargado cuando se construye la celda.

#### ***neighborhood***

Signatura: `NeighborhoodValue &neighborhood()`

Descripción: Retorna el vecindario asociado a la celda.

#### ***outputPort***

Signatura: `const Port &outputPort() const`

Descripción: Retorna el puerto de salida. Este puerto comunica a la celda con los modelos externos y las celdas vecinas.

#### ***inputPort***

Signatura: `const Port &inputPort() const`

Descripción: Retorna el puerto de entrada de la celda. Este puerto le permite tanto a los modelos externos comunicarse con la celda.

#### ***neighborPort***

Signatura: `const Port &neighborPort() const`

Descripción: Retorna el puerto de entrada por el cual la celda permite que los vecinos se comuniquen con ella.

## 11.5 TransportDelayCell

La clase *TransportDelayCell* es una especialización de la clase *AtomicCell*. Representa a las celdas que respondan a los eventos aplicando una demora de transporte. La demora de transporte aplica una política FIFO para los eventos. A medida que estos llegan si hay alguna programación pendiente el evento nuevo será encolado para ser ejecutado más tarde.

### 11.5.1 Responsabilidades:

- Implementar la los algoritmos de demora de transporte en las funciones de transición externa e interna.

### 11.5.2 Colaboradores:

- **CoupledCell**: Provee los valores por defecto para la celda.

### 11.5.3 Métodos principales:

#### *internalFunction*

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

Descripción: Responde a un evento interno. Al ser una celda con demora de transporte elimina el evento que paso de la cola y se reprograma en caso de ser necesario.

#### *externalFunction*

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

Descripción: Responde a un evento externo. Al ser una celda con demora de transporte si el cálculo de la función local provee un valor distinto este resultado es encolado junto con su demora y se reprograma la celda en caso de ser necesario.

#### *outputFunction*

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

Descripción: Responde al pedido de salida que se realiza antes de invocar a la función para los eventos internos. Por ser una celda con demora de transporte toma el valor de la cola y lo envía por el puerto de salida.

## 11.6 InertialDelayCell

La clase *InertialDelayCell* es una especialización de la clase *AtomicCell*. Representa a las celdas que aplican una política de demora inercial. Cuando un nuevo evento externo llega la celda evalúa la función de cálculo local obteniendo un valor y una demora. Luego analiza si la cantidad de tiempo faltante hasta el próximo evento. Si es mayor a cero hace remoción del valor y programa el próximo evento con la demora nueva. A diferencia de las celdas con demora de transporte el valor que seré enviado en la salida es el de la celda.

### 11.6.1 Responsabilidades:

- Implementar la especificación de demora inercial.

### 11.6.2 Colaboradores:

- **CoupledCell**: Provee los valores por defecto para la celda.

### 11.6.3 Métodos principales:

***internalFunction***

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

Descripción: Responde a un evento interno. Al llegar el evento la función de salida ya ha sido invocada y por lo tanto lo único que resta es pasivarse hasta la llegada de un nuevo evento externo.

***externalFunction***

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

Descripción: Responde a un evento externo. Cuando esto sucede se calcula la función de cómputo local obteniendo una demora nueva y un valor resultado. Si este último es distinto al valor anterior de la celda analiza el tiempo restante para el próximo evento si es mayor que cero hace remoción del último valor y se reprograma con el nuevo.

## *11.7 CoupledCell*

La clase *CoupledCell* es una especialización de la clase *Coupled*. Sus responsabilidades son:

### 11.7.1 Responsabilidades:

- Conocer el tamaño de la grilla de celdas.
- Conocer el tipo de demora para las celdas.
- Conocer el tiempo de demora por defecto para las celdas.
- Conocer el tipo de borde.
- Conocer el valor inicial por defecto para cada celda.
- Conocer el función de transición local por defecto para las celdas.
- Conocer la función de transición local por defecto para las celdas y las regiones definidas con otro comportamiento.
- Crear el conjunto de celdas y los vínculos entre ellas.
- Cargar el valor inicial especificado para cada celda.

### 11.7.2 Métodos principales:

***borderWrapped***

Signatura: `CoupledCell &borderWrapped( bool )`

Signatura: `bool borderWrapped() const`

Descripción: Registra/Proyecta si el borde de la grilla es circular o no.

***inertialDelay***

Signatura: `CoupledCell &inertialDelay( bool )`

Signatura: `bool inertialDelay() const`

Descripción: Registra/Proyecta si el acoplado celular trabaja con celdas de demora inercial o de demora de transporte.

***initialCellValue***

Signatura: `CoupledCell &initialCellValue( const TValBool & )`

Signatura: `const TvalBool& initialCellValue() const`

Descripción: Registra/Proyecta el valor inicial con el cual las celdas deben ser creadas. El parámetro indica el valor.

### ***defaultDelay***

Signatura: `CoupledCell &defaultDelay( const Time & )`

Signatura: `const Time &defaultDelay() const`

Descripción: Registra/Proyecta la demora por defecto que con la que las celdas deben ser creadas. El parámetro indica el valor.

### ***localTransition***

Signatura: `CoupledCell &localTransition(const LocalTransAdmin::Function &)`

Signatura: `const LocalTransAdmin::Function &localTransition() const`

Descripción: Registra/Proyecta la función de cálculo local que con la que las celdas deben ser creadas. El parámetro indica el valor.

### ***setLocalTransition***

Signatura: `virtual CoupledCell &setLocalTransition(const CellPosition &, const LocalTransAdmin::Function &)`

Descripción: Registra la función de cálculo local para una celda en particular. Los parámetros indica el valor y la celda referenciada.

### ***createCells***

Signatura: `virtual CoupledCell &createCells( const CellPositionList &neighbors )`

Descripción: Crea la grilla de celdas. Este método es utilizado en la creación del acoplado celular.

## *11.8 FlatCoupledCell*

La clase *FlatCoupledCell* es una especialización de la clase *CoupledCell*. Un acoplado celular achatado no debe crear la grilla de celdas ya guarda directamente el valor de la celda evitando al sobrecarga del pasaje de mensajes entre los modelos. Es por esto que debe proveer los métodos para aparente poseer un conjunto de celdas. Esta grilla de celdas virtuales permite que su uso sea completamente transparente.

### 11.8.1 Responsabilidades:

- Administración de celdas virtuales.
- Administrar las dependencias entre las celdas virtuales con respecto a sus vecinos.
- Inicializar las celdas.
- Ejecutar la función de transición externa para las celdas virtuales.
- Ejecutar la función de transición interna para las celdas virtuales.

### 11.8.2 Métodos principales:

#### ***createCells***

***Signatura:*** `CoupledCell &createCells( const CellPositionList &neighbors )`

Descripción: Crea la estructura para albergar los valores que representen a las celdas. El parámetro indica la lista de vecinos.

#### ***setLocalTransition***

Signatura: `CoupledCell &setLocalTransition( const CellPosition &, const LocalTransAdmin::Function & )`

Descripción: Registra la función de cálculo local para una celda en particular. Los parámetros indica el valor y la celda referenciada.

#### ***initFunction***

Signatura: `Model &initFunction()`

Descripción: Inicializa el conjunto de valores que representan a las celdas. Esta inicialización es equivalente a procesar un mensaje externo para cada una de las celdas.

#### ***externalFunction***

Signatura: `Model &externalFunction( const Time&, const CellPosition&, bool, TValBool eventValue)`

Descripción: Frente a un evento externo el acoplado celular chato debe analizar cual es la celda destinataria, el valor del evento, la hora y si es de un modelo externo. Estos son los valores pasados por parámetro. Una vez determinada la celda se le aplica la función de cálculo local y se aplica la política según sea demora inercial o de transporte. Como las celdas no existen es necesario registrar en una lista todos los eventos destinados a otras celdas.

#### ***internalFunction***

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

Descripción: Frente a un evento interno el acoplado celular chato debe producir las salidas correspondientes y actualizar la lista de celdas que deben ser activadas. Para realizar el mapeo de puertos de salida a puertos de entrada se utilizan otras dos listas que registran las conexiones entre las celdas.

#### ***addInfluence***

Signatura: `Model &addInfluence( const string &sourceName, const string &sourcePort, const string &destName, const string &destPort)`

Descripción: Registra una conexión entre dos celdas al igual que su clase base. Sin embargo las celdas son virtuales y por lo tanto son necesarias dos listas (Xlist, Ylist) para registrar las influencias.

# 12 Jerarquía de Procesadores

Esta jerarquía tiene como principal responsabilidad proveer el mecanismo de simulación necesario para que los modelos puedan llevar a cabo su comportamiento.

Las clase responsable de administrar la recepción de mensajes y tomar las acciones correspondientes es la clase processor. Esta es la clase base asbtracta y las clases Simulator, Coordinator, CellCoordinator, FlatCorrdinator y Root-Coordinator son las clases concretas. Los procesadores son los encargados de llevar a cabo la simulación distribuyendo el control entre los distintos modelos que conforman la simulación.

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

Ahora veremos a todos los componentes de la jerarquía de procesadores explicitando las responsabilidades y los métodos relevantes.

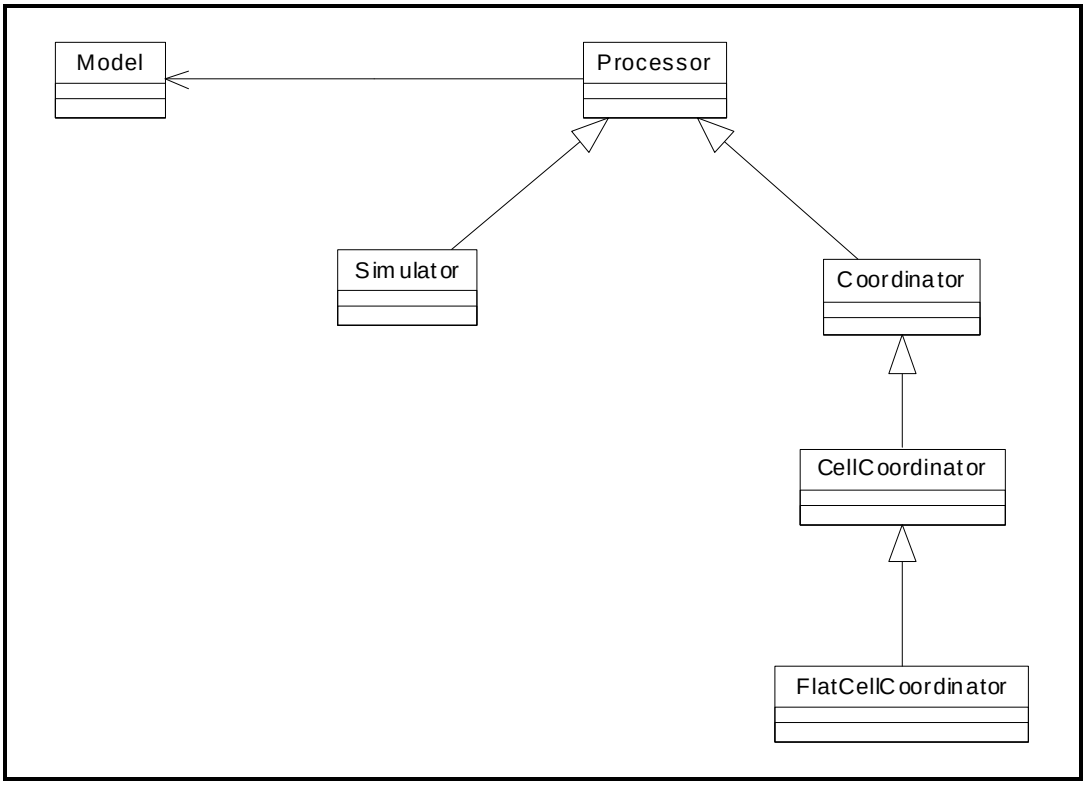



**Ilustración 13 - Jerarquía de procesadores**

## *12.1 Processor*

La clase *Processor* es la clase base abstracta para toda la jerarquía de procesadores. Representa el mecanismo de simulación utilizado. Sus responsabilidades son:

### 12.1.1 Responsabilidades:

Recibir mensajes (Inicialización, Evento Externo, Evento Interno, Done).

Conocer a su modelo asociado.

Conocer a su *Processor* progenitor.

Enviar mensajes de salida hacia su procesador padre.

### 12.1.2 Colaboradores:

- **SingleProcessorAdmin** : Referencia a los procesadores existentes.
- **Model**: modelo asociado.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.1.3 Métodos principales:

***receive***

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

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

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

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

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

Descripción**:** Produce un excepción **InvalidMessageException**, ya que es responsabilidad de las clases concretas redefinir este método en caso de que por definición puedan recibir un mensaje de tipo *Init, Internal, External, Output* y *Done.*

### ***id***

Signatura: `const ProcId &id()const`

Descripción: Retorna el identificado unívoco del procesador. Este identificador es asignado cuando el par modelo/procesador es creado, y es igual para ambas jerarquías.

### ***model***

Signatura: `Model &model()`

Descripción: Retorna el modelo asociado. El modelo debe ser un modelo correcto, ya que el par procesador/modelo fue creado en una única operación atómica. De haber un error se producirá un **InvalidModelIdException**.

### ***nextChange***

Signatura: `Processor &nextChange( const Time& )`

Descripción: Modifica la hora de la próxima transición interna. El parámetro representa a qué ahora debe producirse. Este método solamente puede ser invocado por clases que extiendan a *Processor* (protected).

### ***lastChange***

Signatura: `Processor &lastChange( const Time& )`

Descripción: Modifica la hora en la que se produjo la última transición interna. El parámetro representa la hora deseada. Este método solamente puede ser invocado por clases que extiendan a *Processor* (protected).

### ***sendOutput***

Signatura: `virtual Processor &`

`sendOutput(const Time &, const Port &, const Value & )`

Descripción: Envía un mensaje de salida (tipo Y) al padre con la hora, puerto y valor especificados. Este método solamente puede ser invocado por clases que extiendan a *Processor* (protected)

## *12.2 Simulator*

La clase *Simulator* es una especialización de la clase *Processor*. Los simuladores representan el mecanismo de simulación para los modelos atómicos. Sus responsabilidades son:

### 12.2.1 Responsabilidades:

Recibir mensajes y canalizarlos hacia el modelo atómico asociado.

Enviar mensajes hacia su procesador padre indicando la hora del próximo evento de su modelo asociado.

### 12.2.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.2.3 Métodos principales:

***receive***

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

Descripción**:** Actualiza la hora de ultimo y próximo evento en base al mensaje recibido e invoca la función de inicialización de su modelo atómico asociado y envía un mensaje *done* con la hora del próximo evento al procesador padre.

***receive***

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

Descripción**:** Actualiza la hora de ultimo y próximo evento en base al mensaje recibido e invoca la función de transición externa de su modelo atómico asociado y envía un mensaje *done* con la hora del próximo evento al padre.

***receive***

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

Descripción**:** Actualiza la hora de ultimo y próximo evento en base al mensaje recibido e invoca la función de salida del modelo atómico asociado y a la función de transición externa. Luego envía un mensaje *done* con la hora del próximo evento al padre.

## *12.3 Coordinator*

La clase *Coordinator* es una especialización de la clase *Processor*. Los coordinadores representan el mecanismo de simulación para los modelos acoplados.

### 12.3.1 Responsabilidades:

Redireccionar los mensajes de inicialización hacia todos sus hijos.

Redireccionar los mensajes externos por medio de las influencias de los puertos. Para esto analiza su esquema de acoplamiento interno.

Redireccionar los mensajes de salida por medio de las influencias de los puertos y el esquema de acoplamiento externos reenviando los mensajes hacia los hijos o hacia afuera a modelos externos.

Redireccionar los mensajes internos hacia el *Processor* inminente.

Enviar a su procesador padre la hora del próximo evento interno.

### 12.3.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.3.3 Métodos principales:

***receive***

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

Descripción: Envía un mensaje de inicialización (tipo I) a todos los procesadores que componen el modelo acoplado asociado. Registra en la variable de instancia *doneCount* la cantidad de mensajes enviados para poder actuar en consecuencia al recibir los mensajes *done* de respuesta de cada uno de los hijos.

***receive***

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

Descripción: Envía un mensaje de transición interna (tipo *) al procesador correspondiente del modelo inminente. Carga en 1 (uno) la variable de instancia *doneCount* para poder actuar en consecuencia al recibir el mensajes *done* en respuesta al mensaje enviado.

***receive***

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

Descripción: Envía un mensaje externo (tipo X) al todos los procesadores asociados a los modelos que son influenciados por el puerto desde donde ingreso el mensaje. Registra en la variable de instancia *doneCount* la cantidad de mensajes enviados para poder actuar en consecuencia al recibir los mensajes *done* de respuesta de cada uno de ellos.

***receive***

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

Descripción: Envía un mensaje externo (tipo X) al todos los procesadores asociados a los modelos que son influenciados por el puerto desde donde se recibió el mensaje de salida. Si alguno de los puertos influenciados es un puerto de salida del modelo acoplado asociado, se envía un mensaje de salida (tipo Y) al procesador padre. Por último se registra en la variable de instancia *doneCount* la cantidad de mensajes enviados para poder actuar en consecuencia al recibir los mensajes *done* en respuesta a cada uno de ellos.

***receive***

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

Descripción: Utiliza la variable de instancia *doneCount* para actuar recién después de haber recibido todos los mensajes *done* que son esperados. Luego de haber recibido todos los mensajes *done*, se recalcula la hora e identidad del hijo inminente, y envía un mensaje *done* al procesador padre indicándole la hora del próximo evento interno. En caso de ocurrir un error, o sea recibir un mensaje *done* no esperado se producirá un AssertException.

***receive***

Signatura: `Coordinator &Coordinator::recalcInminentChild()`

Descripción: Recalcula y registra el hijo inminente y la hora del próximo evento entre todos los modelos/procesadores que componen el modelo acoplado asociado. Aquí se encuentra implícita la función *select* ya que frente a dos modelos con la misma hora de futuro cambio de estado se toma para decidir el órden puesto en el archivo de especificación de modelos.

## *12.4 CellCoordinator*

La clase *CellCoordinator* es una especialización de la clase *Coordinator*. Representa el mecanismo de simulación para los modelos celulares.

### 12.4.1 Responsabilidades:

Administrar los hijos inminentes con una política especifica para los modelos celulares.

Administrar las los mensajes de salida con una política especifica para los modelos celulares.

### 12.4.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.4.3 Métodos principales:

***receive***

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

Descripción**:** Envía un mensaje de transición interna (tipo *) a todos los procesos asociados a los modelos inminentes (cuya hora de próximo cambio de estado es igual a la hora del mensaje * recibido), y registra en la variable de instancia *doneCount* para poder actuar en consecuencia al recibir el mensajes *done* en respuesta a los mensajes enviados. El orden en que se envían los mensajes (forzando así el orden de procesamiento) representa a la función *select* que por defecto, si no esta especificado en el archivo de configuración, aplica el órden de pares.

$$(x,y) < (z,w) \Leftrightarrow x < z \vee ( x = z \wedge y < w )$$

***receive***

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

Descripción**:** Para cada influencia hacia un puerto de salida, se envía un mensaje de salida (tipo Y) hacia el procesador padre. Para el resto de las influencias del puerto desde el que se recibió el mensaje, si no se el modelo dueño del puerto no se encuentran registrados en la lista *influenced* se envía un mensaje externo (tipo X) y se registra en la lista. Por último se registra en la variable de instancia *doneCount* la cantidad de mensajes externos enviados para poder actuar en consecuencia al recibir los mensajes *done* en respuesta a cada uno de ellos.

***receive***

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

Descripción**:** Utiliza la variable de instancia *doneCount* para actuar recién después de haber recibido todos los mensajes *done* que son esperados. Luego de haber recibido todos los mensajes *done*, se recalcula la hora e identidad del/los hijos inminentes y se registra en la estructura *inminentsChild*, y envía un mensaje *done* al procesador padre indicándole la hora del próximo evento interno. En caso de ocurrir un error, o sea recibir un mensaje *done* no esperado se producirá un AssertException. Por ultimo se borra la lista *influenced*.

## 12.5 FlatCoordinator

La clase *FlatCoordinator* es una especialización de la clase *Coordinator*. Representa el mecanismo de simulación para los modelos celulares achatados. Para este tipo de coordinador, que no posee hijos asociados, los métodos para recibir mensajes de *Output* y *Done* nunca serán recibidos.

### 12.5.1 Responsabilidades:

Recibir los mensajes y direccionarlos hacia el modelo acoplado chato asociado.

### 12.5.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.5.3 Métodos principales:

***receive***

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

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

***receive***

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

Descripción**:** Ejecuta el método de transición interna del modelo celular acoplado chato asociado, ajusta los valores de hora de último/próximo evento y luego envía un mensaje *done* de respuesta a su procesador padre indicando la hora de su próximo cambio de estado. El orden en que se envían los mensajes (forzando así el orden de procesamiento) representa a la función *select* que por defecto, si no esta especificado en el archivo de configuración, aplica el órden de pares.

$$(x,y) < (z,w) \Leftrightarrow x < z \vee ( x = z \wedge y < w )$$

***receive***

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

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

## 12.6 Root

La clase *Root* es una especialización de la clase *Processor*. Representa la raíz del árbol de procesadores a partir de la cual se comienza con el mecanismo de simulación, por eso existe una única instancia de esta clase (Pattern Singleton [Gam95]). Este procesador no tiene un modelo asociado.

### 12.6.1 Responsabilidades:

Administrar los eventos externos.

Avanzar la hora de simulación.

Iniciar y detener la simulación.

Generar la salida de la simulación en función de los mensajes de salida recibidos.

### 12.6.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 12.6.3 Métodos principales:

***initialize***

Signatura**:** `Root &initialize()`

Descripción**:** Carga el estado de próximo/anterior evento en cero, limpia la lista de eventos externos y registra la hora de finalización de la simulación en infinito.

***addExternalEvent***

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

Descripción**:** Agrega a la lista de eventos externos el evento con hora, puerto de entrada y valor indicado por los parámetros.

***stopTime***

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

Descripción**:** Registra la hora de finalización de la simulación. El valor deseado es indicado en el parámetro.

***simulate***

Signatura**:** `Root &simulate()`

Descripción**:** Envía un mensaje de inicialización al modelo acoplado de mas alto nivel en la jerarquía, y ejecuta el método *run* de la clase *MessageAdmin*, comenzando de esta forma el ciclo de envío y recepción de mensajes.

***receive***

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

Descripción**:** Envía por el dispositivo de salida una cadena de texto indicando la hora, el puerto y el valor del mensaje pasados por parámetro.

***receive***

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

Descripción**:** En caso de haber llegado a la hora de finalización especificada o haber consumido los eventos externos y todos los modelos que componen la jerarquía se encuentran pasivados (hora de próximo evento igual a infinito), la simulación se da por finalizada; de no ser así, el analiza el próximo evento a ser tratado y se envía un mensaje al coordinador de mayor nivel avanzando de esta forma la hora de la simulación.

# 13 Administración de modelos y procesadores

Tanto los procesadores como los modelos utilizados en la simulación son creados cuando se lee la especificación de modelos y eliminados al final de la ejecución. Durante el transcurso de la simulación se hace referencia constantemente a modelos padres e hijos por esto debe proveerse un mecanismo para conocer a estos modelos o procesadores.

Delegar esta responsabilidad en un administrador permite distinguir el uso de los modelos de su construcción, destrucción y ubicación. De esta forma fácilmente puede aplicarse una política distribuida sin impactar en la aplicación. Esta es la idea detrás del administrador de procesadores además el administrador de modelos registrará los nuevos modelos atómicos y proveerá la funcionalidad para crear nuevas instancias de ellos a medida que se necesite.

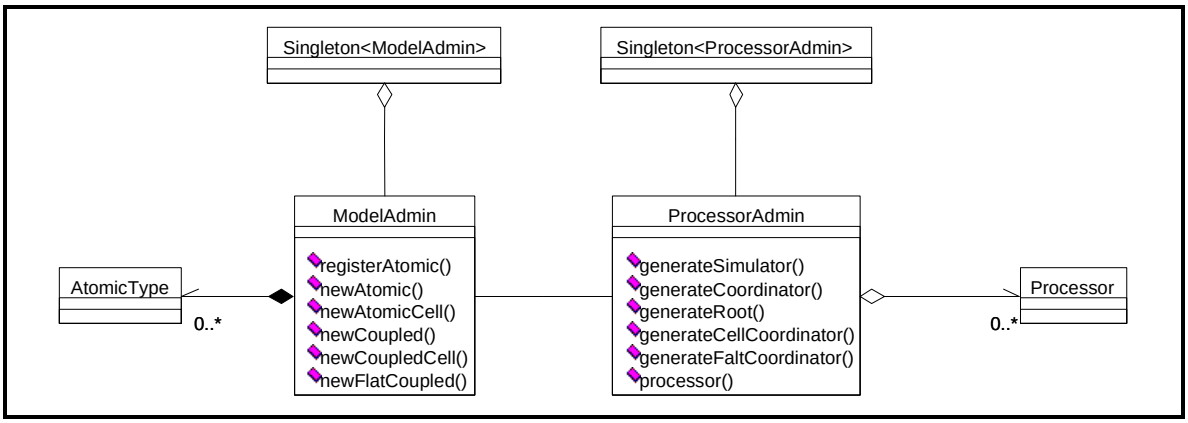



**Ilustración 14 - ModelAdmin y ProcessorAdmin**

## *13.1 ProcessorAdmin y SingleProcessorAdmin*

La clase *ProcessorAdmin* administra todos los procesadores de la simulación. Esta clase debe ser conocida por todos los componentes de la simulación existe una única instancia del administrador de procesadores (SingleProcessorAdmin, Pattern Singleton [Gam95]).

### 13.1.1 Responsabilidades:

Crear una nueva instancia de *Root*.

Crear una nueva instancia de *Simulator*.

Crear una nueva instancia de *Coordinator*.

Crear una nueva instancia de *CellCoordinator*.

Crear una nueva instancia de *FlatCoordinator*.

Desalocar los coordinadores existentes al finalizar la simulación.

Buscar un procesador a partir de su identificador.

### 13.1.2 Colaboradores:

- **SingleModelAdm**: Obtiene modelos a partir de su identificador.
- **SingleMsgAdm**: Envía mensajes entre procesadores.

### 13.1.3 Métodos principales:

***generateProcessor***

Signatura**:** `Processor &generateRoot()`

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

***generateProcessor***

Signatura**:** `Processor &generateProcessor( Atomic * )`

Descripción**:** Crea una instancia de la clase *Simulator* a partir del modelo atómico indicado como parámetro y lo agrega en la base de coordinadores. Retorna el procesador que acaba de crear.

***generateProcessor***

Signatura**:** `Processor &generateProcessor( Coupled * )`

Descripción**:** Crea una instancia de la clase *Coordinator* a partir del modelo acoplado indicado como parámetro y lo agrega en la base de coordinadores. Retorna el procesador que acaba de crear.

***generateProcessor***

Signatura**:** `Processor &generateProcessor( CoupledCell * )`

Descripción**:** Crea una instancia de la clase *CellCoordinator* a partir del modelo acoplado celular indicado como parámetro y lo agrega en la base de coordinadores. Retorna el procesador que acaba de crear.

***generateProcessor***

Signatura**:** `Processor &generateProcessor( FlatCoupledCell * )`

Descripción**:** Crea una instancia de la clase *FlatCoordinator* a partir del modelo acoplado celular achatada indicado como parámetro y lo agrega en la base de coordinadores. Retorna el procesador que acaba de crear.

***processor***

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

Descripción**:** Busca en la base de coordinadores el coordinador que coincida con el identificador especificado en el parámetro. Si existe lo retorna. En caso contrario se lanzará un **InvalidModelIdException**.

## *13.2 ModelAdmin y SingleModelAdm*

La clase *ModelAdmin* centraliza la creación de todos los modelos. Debido a que todos los modelos tienen un procesador asociado se hace la creación simultánea. Debido a que es la única clase capaz de esta tarea existe una única instancia (SingleModelAdm, Pattern Singleton [Gam95]).

### 13.2.1 Responsabilidades:

Crear una nueva instancia de *Atomic*.

Crear una nueva instancia de *AtomicCell*.

Crear una nueva instancia de *Coupled*.

Crear una nueva instancia de *CoupledCell*.

Crear una nueva instancia de *FlatCoupledCell*.

Desalocar los coordinadores existentes al finalizar la simulación.

Buscar un procesador a partir de su identificador.

### 13.2.2 Colaboradores:

- **SingleProcessorAdmin**: Crea los procesadores a partir del modelo.
- **NewFunction**: Provee la interfaz para la creación de nuevos modelos atómicos.
- **NewAtomicFunction**: Template que se instancia con el modelo atómico deseado.

### 13.2.3 Métodos principales:

#### ***registerAtomic***

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

Descripción**:** Registra un nuevo tipo de modelo atómico. Los parámetros indican la función de creación que debe utilizarse y la cadena indica el nombre del nuevo tipo de modelo atómico. Retorna un identificador indicando el tipo asociado al nombre.

#### ***newAtomic***

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

Descripción**:** Crea una nueva instancia de la clase pedida en el primer parámetro. El segundo parámetro especifica el nombre de la instancia. Retorna la instancia recién creada.

#### ***newAtomicCell***

Signatura**:** `AtomicCell &newAtomicCell(bool inertial, const string &)`

Descripción**:** Crea una nueva instancia de la clase *TransporDelayCell o InertialDelayCell* en función del primer parámetro especificado. El segundo indica el nombre de la nueva celda atómica. Retorna la instancia recién creada.

#### ***newCoupled***

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

Descripción**:** Crea una nueva instancia de la clase *Coupled*. El parámetro especifica el nombre. Retorna la instancia recién creada.

#### ***newCoupledCell***

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

Descripción**:** Crea una nueva instancia de la clase *CoupledCell*. El parámetro especifica el nombre. Retorna la instancia recién creada.

#### ***newFlatCoupledCell***

Signatura**:** `FlatCoupledCell &newFlatCoupledCell( const string &modelName )`

Descripción**:** Crea una nueva instancia de la clase *FlatCoupledCell*. El parámetro especifica el nombre. Retorna la instancia recién creada.

# 14 Pasaje de Mensajes

El pasaje de mensajes entre los coordinadores de la simulación representa el mecanismo del motor de simulación. Deben existir tantos mensajes distintos como distintos eventos posibles dentro del formalismo. A su vez cada mensaje podrá tener información particular asociada al tipo de evento que representa. Una vez armadas las estructuras para llevar los mensajes es necesaria una entidad que sea la responsable de enviar los mensajes entre los procesadores ya que de esta forma queda aislado el tratamiento de los mensajes del mecanismo de pasaje de mensajes permitiendo cambiar la implementación sin tener mayor impacto en la implementación.

En esta sección veremos los distintos tipos de mensajes y el responsable del pasaje de estos entre los procesadores.

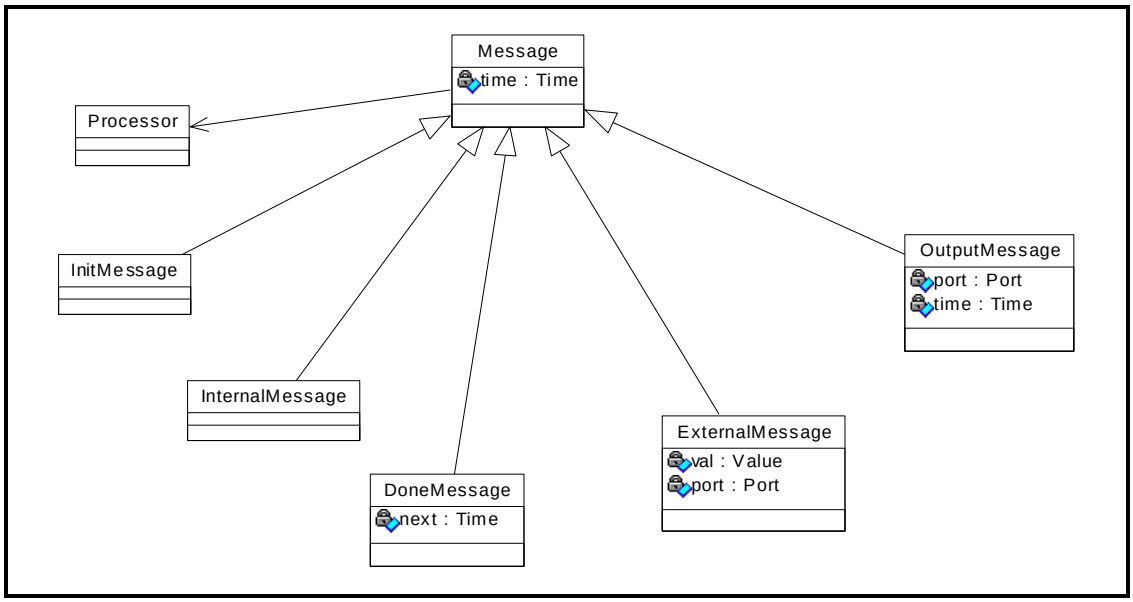



**Ilustración 15 - Jerarquía de Mensajes**

## *14.1 Message*

La clase *Message* es la clase base abstracta de los mensajes que intercambian los procesadores. Esta clase no puede ser instanciada ya que solamente provee la interfaz para el resto de los mensajes.

### 14.1.1 Responsabilidades:

Conocer el procesador originador del mensaje.

Conocer la hora en que es enviado el mensaje.

### 14.1.2 Colaboradores:

**Processor**: Recibe todos los tipos de mensaje.

### 14.1.3 Métodos principales:

***time***

Signatura: `const Time &time() const`

Signatura: `Message &time( const Time & )`

Descripción: Carga/Proyecta la hora en la cual el mensaje es enviado.

***procId***

Signatura**:** `const ProcId &procId() const`

Signatura: `Message &procId( const ProcId & )`

Descripción**:** Carga/proyecta el identificador del procesador que responsable de enviar el mensaje.

***sendTo***

Signatura**:** `virtual Message &sendTo( Processor & ) = 0`

Descripción**:** Permite aplicar el mecanismo polimórfico de C++ e invocar al método *receive* que se corresponda al mensaje en el procesador destino.

## ***14.2*** *InitMessage*

La clase *InitMessage* extiende a la clase *Message*. Representa el mensaje de inicialización que reciben los procesadores cuando la simulación comienza, se corresponde con el mensaje **I**. Las responsabilidades son equivalentes a la clase base, se utiliza como tipo de mensaje sin agregar información adicional.

## *14.3 InternalMessage*

La clase *InternalMessage* extiende a la clase *Message*. Representa la indicación de una función de transición interna que reciben los procesadores, se corresponde con el mensaje *****. Las responsabilidades son equivalentes a la clase base, se utiliza como tipo de mensaje sin agregar información adicional.

## *14.4 ExternalMessage*

La clase *ExternalMessage* extiende a la clase *Message*. Representa el mensaje que indica un evento externo, se corresponde con el mensaje X entre procesadores.

### 14.4.1 Responsabilidades:

Conocer el procesador el puerto por el cual llega el mensaje.

Conocer el valor del mensaje.

### 14.4.2 Métodos principales:

***port***

Signatura**:** `const Port &port() const`

Signatura**:** `Message &port( const Port & )`

Descripción**:** Carga/Proyecta el puerto por el cual el mensaje arriba.

***value***

Signatura**:** `const Value &value() const`

Signatura**:** `Message &value( const Value & )`

Descripción**:** Carga/Proyecta el valor que contiene el mensaje.

## *14.5 DoneMessage*

La clase *DoneMessage* extiende a la clase *Message*. Representa el mensaje que reciben los procesadores de los hijos indicando la hora en la cuál tendrán su próximo cambio de estado, se corresponde con el mensaje **D**.

### 14.5.1 Responsabilidades:

Conocer la hora del próximo cambio de estado.

### 14.5.2 Métodos principales:

***nextChange***

Signatura**:** `const Time &nextChange() const`

Signatura**:** `Message &nextChange( const Time & )`

Descripción**:** Carga/Proyecta la hora del próximo cambio de estado.

## *14.6 OutputMessage*

La clase *DoneMessage* extiende a la clase *Message*. Re

### 14.6.1 Responsabilidades:

Conocer el procesador originador del mensaje.

Conocer la hora en que es enviado el mensaje.

### 14.6.2 Colaboradores:

**Processor**: Recibe todos los tipos de mensaje.

### 14.6.3 Métodos principales:

***port***

Signatura**:** `const Port &port() const`

Signatura**:** `Message &port( const Port & )`

Descripción**:** Carga/Proyecta el puerto por el cual el mensaje debe salir.

***value***

Signatura**:** `const Value &value() const`

Signatura**:** `Message &value( const Value & )`

Descripción**:** Carga/Proyecta el valor de salida que contiene el mensaje.

## *14.7 MessageAdm y SingleMsgAdmin*

La clase *MessageAdm* administra los pedidos de envío de mensajes entre procesadores. Debido a que esta clase la deben conocer todos los procesadores, existe una sola instancia de esta clase pública (Pattern Singleton [Gam95]).

La centralización del pasaje de mensajes entre los modelos permite que la política de pasaje de mensajes quede encapsulada dentro del administrador. En esta implementación el criterio seleccionado es FIFO forzando así una simulación secuencial ya que el envío del próximo mensaje se produce cuando el modelo finaliza el procesamiento del mensaje anterior. Es por esto que la simulación comienza cuando *Root* invoca el método *run()* para que comience el ciclo de pasaje de mensajes.

Cambiar a otra política de pasaje de mensajes (por ejemplo una administración paralela) implicaría únicamente modificar el método de distribución de mensajes del administrador.

### 14.7.1 Responsabilidades:

Comenzar la administración de mensajes.

Finalizar la administración de mensajes.

Enviar un mensaje a un procesador.

### 14.7.2 Colaboradores:

**Processor**: Recibe todos los tipos de mensaje.

**MainSimulator:** Provee el log para registrar los mensajes enviados.

**Message:** Reenvía el tipo de mensaje que corresponde al procesador indicado.

### 14.7.3 Métodos principales:

***run***

Signatura**:** `MessageAdmin &run()`

Descripción**:** Comienza el ciclo de pasaje de mensajes.

***stop***

Signatura**:** `MessageAdmin &stop()`

Descripción**:** Finaliza el ciclo de pasaje de mensajes.

***send***

Signatura**:** `MessageAdmin &send( const Message &, const ProcId & )`

Descripción**:** Envía un mensaje a un procesador. Los parámetros indican el mensaje a enviar y el procesador destino deseado. Por ser una política FIFO el mensaje será encolado hasta que termine la iteración actual.

# 15 Jerarquía de loaders

La configuración del simulador permite especificar de dónde se desea cargar la especificación de modelos, los eventos externos a utilizar, hora de finalización de la simulación, el destino de el logging de mensajes de la simulación y de los valores de salida.

Existen dos formas de cargar estos valores, la primera es vía línea de comandos especificando los parámetros deseados. La segunda es ejecutando el simulador como servidor de simulación a la espera de clientes que especifiquen a través de la conexión la configuración deseada y obtengan los resultados por este medio.

La especificación de los modelos es cargada de un archivo escrito en un lenguaje propietario desarrollado para tal efecto. La descripción de cada modelo incluye sus puertos de entrada-salida y sus relaciones con el resto de los modelos.

La carga del sistema a simular incluye la lectura e interpretación de los archivos de especificación y la posterior generación del entorno que conformará el sistema a simular.

Una vez cargados todos los datos se procede a comenzar la simulación obteniendo los resultados de la misma, y direccionándolos hacia la salida correspondiente (archivos locales o canal de comunicación).

## *15.1 Simloader*

La clase *SimLoader* es la clase que provee la interfaz para la carga de la configuración del simulador. Esta clase es abstracta ya que sus derivadas son las que implementan el método concreto de obtención de parámetros.

La clase *StandAloneLoader*, deriva de *SimLoader* y es la responsable de la carga de parámetros del simulador por línea de comando.

La clase *NetworkLoader*, deriva de *SimLoader* y es la responsable de obtener los parámetros utilizando servicios de TCP/IP (sólo en ambientes Unix en esta implementación).

La instanciación del loader del simulador se realiza al comienzo del programa ya que sin el no es posible cargar los modelos y por lo tanto llevar a cabo la simulación. Una vez tomada la decisión sobre el loader a instanciar debe indicársele al simulador la instancia de loader elegida.

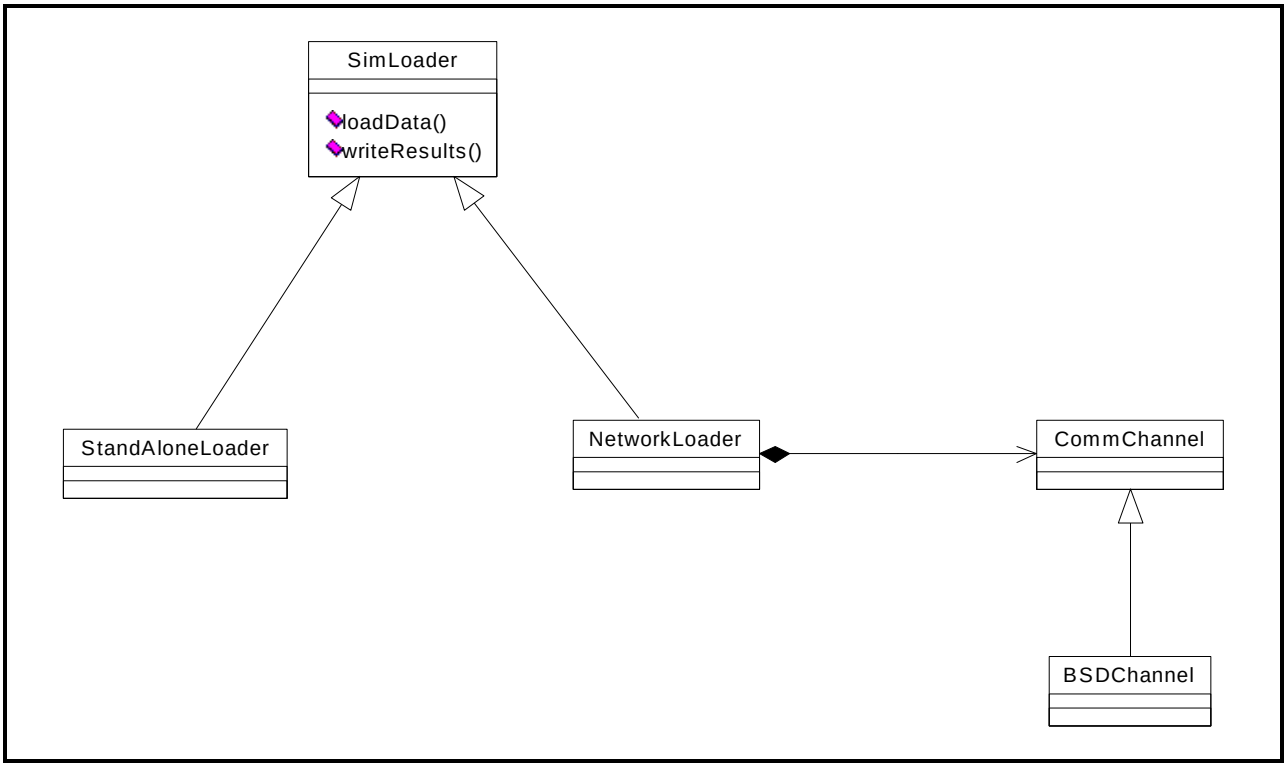



**Ilustración 16 - Jerarquía de Cargadores**

### 15.1.1 Responsabilidades:

Establecer la interface que deben redefinir las clases derivadas

Administrar los espacios de almacenamiento para la definición de los modelos, mensajes externos, mensajes internos y resultados de la simulación.

### 15.1.2 Métodos Principales:

#### ***modelStream***

Signatura: `virtual istream &modelsStream()`

Descripción: Retorna el espacio de almacenamiento reservado para la descripción de los modelos. Este método puede ser sobrecargado por una clase derivada que prefiera utilizar su propia estructura.

#### ***eventStream***

Signatura: `virtual istream &eventsStream()`

Descripción: Retorna el espacio de almacenamiento reservado para los eventos externos. Este método puede ser sobrecargado por una clase derivada que prefiera utilizar su propia estructura.

#### ***logStream***

Signatura: `virtual ostream &logStream()`

Descripción: Retorna el espacio de almacenamiento reservado para los mensajes internos producidos por el simulador. Este método puede ser sobrecargado por una clase derivada que prefiera utilizar su propia estructura.

#### ***outputStream***

Signatura: `virtual ostream &outputStream()`

Descripción: Retorna el espacio de almacenamiento reservado para los mensajes de salida generados por el simulador. Este método puede ser sobrecargado por una clase derivada que prefiera utilizar su propia estructura.

#### ***stopTime***

Signatura: `const Time &stopTime() const`

Descripción: Retorna la hora registrada como tope para el funcionamiento de la simulación. Esta hora indica cual es la hora máxima a la cual puede llegar el motor de simulación, para luego detenerse y terminar su funcionamiento.

#### ***loadData***

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

Descripción: Sirve solo como interface para ser redefinida por las clases derivadas, que implementarán a través de esta invocación la carga de los parámetros especificados por el usuario para llevar a cabo una simulación: definición de los modelos, eventos externos y hora de detención de la simulación.

#### ***writeResults***

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

Descripción**:** Sirve solo como interface para ser redefinida por las clases derivadas, que implementarán a través de esta invocación la carga devolución de los resultados de la simulación al usuario: mensajes de registro y de salida indicando los resultados obtenidos por el simulador.

## *15.2 StandAloneLoader*

La clase *AtomStandAloneLoader* es una especialización de la clase abstracta *SimLoader* y representa la interfaz de la clase encargada en obtener los parámetros estandar a través de simples archivos indicados por la línea de comando.

### 15.2.1 Responsabilidades:

Manejo de archivos que intervienen tanto en los parámetros de entrada como en los datos de salida.

Asumir valores por defecto para los parámetros opcionales

### 15.2.2 Métodos Principales:

#### ***StandAloneLoader***

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

Descripción**:** Es el constructor de la clase StandAloneLoader. Recibe los parámetros pasados por el usuario en la invocación de la aplicación. Su tarea es examinar los parámetros recibidos y obtener la hora de finalización de la simulación y los nombres de los archivos tanto de entrada (especificación de modelos y eventos externos) como de salida (registro y resultados de salida). También asumirá valores por defecto para los parámetros optativos no especificados, por ejemplo: si no se especifica la hora de terminación de la simulación se registrará la hora en infinito.

#### ***loadData***

Signatura**:** `SimLoader &loadData()`

Descripción**:** Se encarga de la apertura de los archivos a través de los cuales se obtendrá los parámetros para llevar a cabo la simulación (especificación de modelos y eventos externos), y en los que se depositarán los resultados una vez efectuada la misma (registro y resultados de salida). Para los datos de entrada, también leerá el contenido de los archivos y los almacenará en el espacio reservado en la clase base *SimLoader*.

#### ***writeResults***

Signatura**:** `SimLoader &writeResults()`

Descripción**:** Tiene como única función el cierre y desalocación de los recursos reservados para la devolución de los resultados de la simulación. En este caso en particular, las salidas se generan durante la simulación, y al final de la misma solo resta cerrar los archivos intervinientes.

## *15.3 NetworkLoader*

La clase *NetworkLoader* es una especialización de la clase abstracta *SimLoader* y representa la interfaz de un loader que obtiene y retorna información a través de un canal de comunicaciones, que le brinda las primitivas básicas para efectuar todas sus operaciones de lectura/escritura sobre la red.

### 15.3.1 Responsabilidades:

Obtención de los parámetros de configuración del simulador a través del canal de comunicaciones.

Envío de los resultados de la simulación a través del canal de comunicaciones.

### 15.3.2 Colaboradores:

CommChannel: Provee la interfaz a través de la cual se realizaran las comunicaciones a través de la red.

BSDChannel: Realiza las comunicaciones a través de la red utilizando una conexión TCP/IP.

### 15.3.3 Métodos Principales:

***loadData***

Signatura: `SimLoader &loadData()`

Descripción: Se encarga de la obtención de los parámetros (especificación de modelos y eventos externos) necesarios para llevar a cabo la simulación Para esto utiliza los servicios de la clase *BSDChannel* para efectuar las comunicaciones a través de la red.

***writeResults***

Signatura: `SimLoader &writeResults()`

Descripción: Retorna por la red los resultados obtenidos de la simulación y el registro de mensajes a través de los servicios de la clase *BSDChannel*.

## *15.4 CommChannel*

La clase *CommChannel* es la clase abstracta que especifica la interfaz que deberán redefinir las clases derivadas para implementar el comportamiento de un canal de comunicaciones concreto.

### 15.4.1 Responsabilidades:

Establecer la interface para la funcionalidad apertura del canal a través del constructor.

Establecer la interface para la funcionalidad lectura.

Establecer la interface para la funcionalidad escritura.

Establecer la interface para la funcionalidad cerrado del canal a través del destructor.

### 15.4.2 Métodos Principales:

***readLine***

Signatura: `virtual string readLine() = 0`

Descripción: Define la interface para que las clases derivadas redefinan y den comportamiento a la lectura de una línea de información a través del canal de comunicaciones concreto que ellas utilicen.

***writeLine***

Signatura: `virtual CommChannel &writeLine( const string & ) = 0`

Descripción: Define la interface para que las clases derivadas redefinan y den comportamiento a la escritura de una línea de información a través del canal de comunicaciones concreto que ellas utilicen.

## *15.5 BSDChannel*

La clase *BSDChannel* es una especialización de la clase abstracta *CommChannel* y representa la interfaz de la clase encargada de realizar la comunicación de datos a través de una conexión TCP/IP vía sockets.

### 15.5.1 Responsabilidades:

Apertura de la conexión TCP/IP a través del constructor.

Lectura de líneas a través de la conexión TCP/IP.

Escritura de líneas a través de la conexión TCP/IP.

Cierre de la conexión TCP/IP a través del destructor.

### 15.5.2 Métodos Principales:

#### ***BSDChannel***

Signatura**:** `BSDChannel( const string &serviceName )`

Descripción**:** Es el constructor de la clase *BSDChannel*. Recibe por parámetro el nombre del servicio bajo el cual figura en el archivo /etc/services (en plataformas Unix). Llamando a la función getservbyname obtiene el número de puerto correspondiente al servicio, o produce un **CommunicationError** de no estar configurado correctamente en el sistema. Una vez obtenido el número puerto invoca al método privado open para que inicie la comunicación.

#### ***BSDChannel***

Signatura**:** `BSDChannel( int portNumber )`

Descripción**:** Es un constructor alternativo de la clase *BSDChannel que r*ecibe por parámetro el número de puerto correspondiente al servicio y luego invoca al método privado open para que inicie la comunicación.

#### ***~BSDChannel***

Signatura**:** `~BSDChannel()`

Descripción**:** Es el destructor de la clase *BSDChannel. Su función es cerrar la conexión TCP/IP*.

#### ***readLine***

Signatura**:** `string readLine()`

Descripción**:** Lee a través de la conexión *TCP/IP una línea completa y la retorna como valor funcional.*

#### ***writeLine***

Signatura**:** `CommChannel &writeLine( const string & )`

Descripción**:** Escribe a través de la conexión *TCP/IP una cadena de caracteres recibida por parámetro..*

#### ***open***

Signatura**:** `BSDChannel &open( int portNumber )`

Descripción**:** Realiza la apertura del canal de comunicaciones cía TCP/IP utilizando el número de puerto recibido por parámetro. De encontrarse problemas al realizar la creación del vínculo se producirá una **CommunicationError**.

# 16 Lenguaje de especificación GADCeLa

El comportamiento de los autómatas celulares se describe utilizando el lenguaje propietario GADCeLa (GAD Cell Language) . Este puede ser descripto por la siguiente BNF:

```
Rule      := Bool Int '{' BoolExp '}'
BoolExp   := IntRelExp | NOT BoolExp | BoolExp AND BoolExp |
          BoolExp OR BoolExp
IntRelExp := IdRef | IntExp OpRel IntExp
IntExp    := IdRef | IntExp Oper IntExp
IdRef     := CellRef | '(' BoolExp ')' | Constant | Function
Constant  := Int | Bool
Function  := TRUECOUNT | FALSECOUNT | UNDEFCOUNT
CellRef   :=  '(' IntExp ',' IntExp ')'
OpRel     := = | != | > | < | >= | <=
Oper      := + | - | * | /
Int       := [Sign] Digit {Digit}
Bool      := 0 | 1 | t | f | ?
Sign      := [+] | -
Digit     := 0 | 1 | ... | 9
```

Este lenguaje permite escribir expresiones en función del valor del vecindario de la celda. La celda al ser de tres estados permite tres posibles: **TRUE (T o 1), FALSE (F o 0)** y **UNDEFINED (?)**. Por lo tanto la expresión lógica debe contemplar estos tres estados como resultado, por esto retorna un valor de la clase *TValBool* que representa el tipo lógico trivalente.

En esta sección analizaremos las clases que participan tanto del parseo del lenguaje como de la interpretación en tiempo de ejecución.

## *16.1 LocalTransAdmin*

Al crearse una celda se le indica cual es la función de cálculo local (especificación) que le corresponde. Es muy probable que varias celdas posean la misma función local el código es asociado a un identificador de función y cada celda registra el identificador que le corresponde.

La clase *LocalTransAdmin* es la encargada de registrar todos los lenguajes especificados. Cuando llegue un evento externo a la celda debe aplicarse la función local para obtener el valor y la demora que indique la regla, esto se hace por intermedio de *LocalTransAdmin*.

### 16.1.1 Responsabilidades:

Administrar reglas.

Evaluar una regla correspondiente a un identificador.

Realizar el parseo de la reglas especificadas.

### 16.1.2 Colaboradores:

**Parser**: Parsea las reglas y retorna una especificación para ejecutar en tiempo de ejecución.

### 16.1.3 Métodos Principales:

***registerTransition***

Signatura**:** `LocalTransAdmin &registerTransition(const string &, istream&)`

Descripción**:** Registra una nueva función local. El primer parámetro indica el nombre de la función y el segundo indica de dónde debe tomarse la especificación. Si el nombre ya existe se

lanzará un **InvalidTransId**. Si existe algún error en la sintaxis de la regla se lanzará un **ErrorParsingException**.

#### ***evaluate***

Signatura**:** `TValBool evaluate( const Function&, const NeighborhoodValue&, Time &delay )`

Descripción**:** Evalúa la función especificada en el parámetro y toma el vecindario especificado como parámetro como tabla de símbolos para resolver las referencias a los vecinos. Retorna el valor de lógica trivalente asociado a la regla y en el último parámetro la demora especificada. Si dos reglas se ven satisfechas se lanzará un **InvalidEvaluation**.

#### ***cellValue***

Signatura**:** `const TValBool &cellValue( const NeighborPosition & )`

Descripción**:** Permite evaluar el valor de una celda en el vecindario especificado.

## *16.2 Parser y SingleParser*

La clase *Parser* provee la funcionalidad necesaria para obtener código ejecutable en tiempo de ejecución a partir de una especificación en texto. Para realizar hemos utilizado al **yacc** para armar el árbol de evaluación.

La clase parser provee una única instancia a la cual se puede acceder públicamente (SingleParser, Pattern Singleton [Gamm 95]).

### 16.2.1 Responsabilidades:

Transformar la especificación texto en código interpretable en tiempo de ejecución.

Realizar el análisis léxico de la especificación.

### 16.2.2 Colaboradores:

**yyparse()**: Función que provee el yacc para ejecutar el parseo.

**ConstantNode, ANDNode, ORNode, NOTNode, EqualNode**...: Nodos del árbol que son retornados cuando se encuentra el valor léxico asociado.

### 16.2.3 Métodos Principales:

#### ***parse***

Signatura**:** `Parser &parse( istream &source )`

Descripción**:** Toma la cadena de entrada pasada por parámetro y llama a yyparse().

#### ***specification***

Signatura**:** `SpecNode *specification()`

Descripción**:** Retorna la el código interpretable de la última especificación parseada.

#### ***nextToken***

Signatura**:** `int nextToken()`

Descripción**:** Retorna el próximo valor léxico.

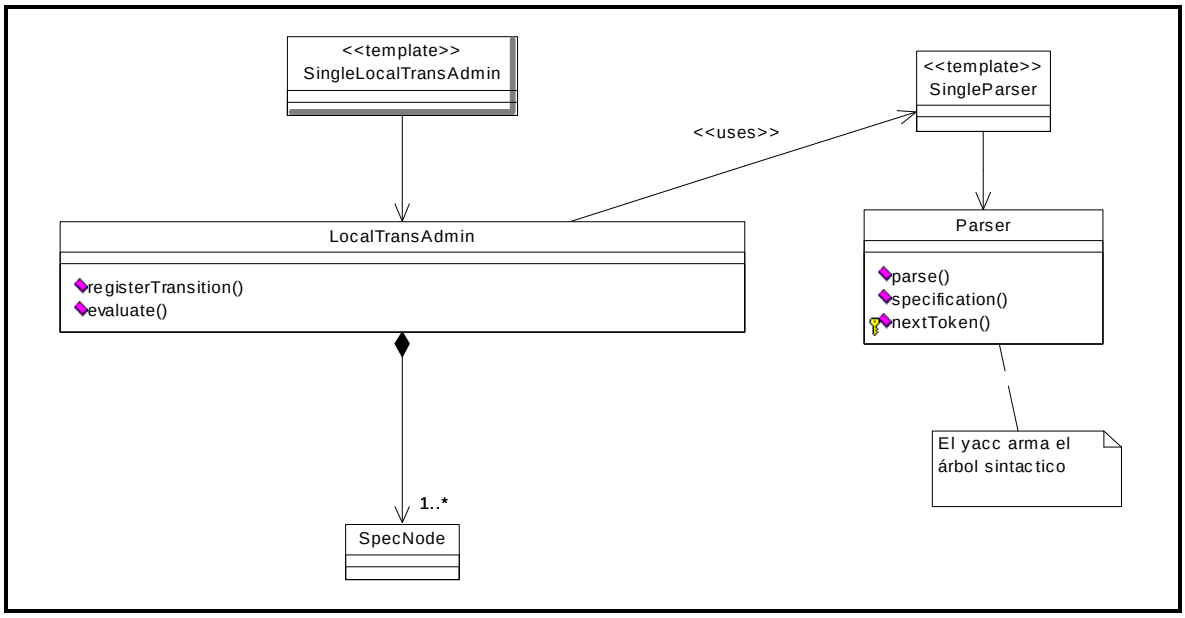



**Ilustración 17 - Parser y LocalTransAdmin**

## *16.3 SynNode*

La clase *SynNode* es la clase base abstracta para el árbol de evaluación. Por cada construcción posible en el lenguaje existe un nodo derivado de *SynNode*. El chequeo de tipos se realiza utilizando el mismo árbol ya que cada nodo conoce el tipo que deben tener sus hijos.

La construcción del árbol esta puesta dentro del código de la función yyparse a partir de la especificación de la gramática con la sintaxis del yacc.

### 16.3.1 Responsabilidades:

Proveer interfaz para evaluar el nodo y retorna un TValBool.

Proveer interfaz para conocer el tipo del nodo.

Proveer interfaz para realizar el chequeo de tipos.

### 16.3.2 Colaboradores:

**TypeValue**: Provee la jerarquía de tipos utilizados (Int y Bool)

### 16.3.3 Métodos Principales:

***evaluate***

Signatura**:** `virtual int evaluate() = 0`

Descripción**:** Evalúa el nodo y retorna el valor calculado, todo valor de lógica trivalente se puede ver como un entero.

***type***

Signatura**:** `virtual const TypeValue &type() const = 0`

Descripción**:** Retorno el tipo asociado al nodo.

***checkType***

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

Descripción**:** Realiza el chequeo de tipos del nodo.

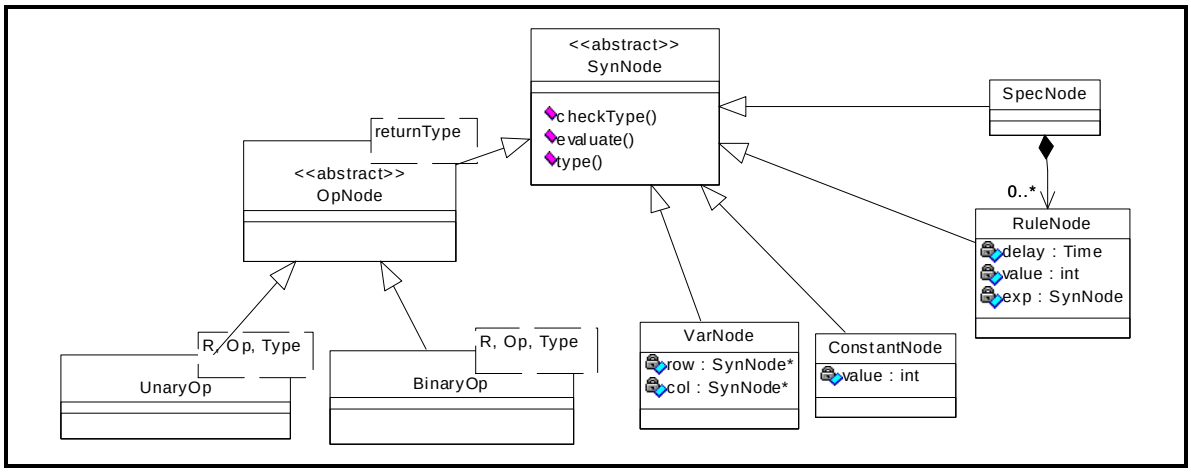



**Ilustración 7 - Jerarquía de nodos**

## *16.4 VarNode*

La clase *VarNode* extiende a la clase *SyntaxNode.* Representa la referencia a una celda. Las celdas son las únicas variables del lenguaje.

### 16.4.1 Responsabilidades:

Retornar el valor de la variable referenciada.

### 16.4.2 Colaboradores:

**SynNode**: El nodo posee dos nodos que representan la expresión aritmética para la fila y la columna.

### 16.4.3 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Retorna el valor asociado a la celda.

***type***

Signatura**:** `const TypeValue &type() const`

Descripción**:** Retorna BoolType::TheBool.

## *16.5 ConstantNode*

La clase *ConstantNode* extiende a la clase *SyntaxNode.* Representa un valor constante.

### 16.5.1 Responsabilidades:

Retornar el valor constante que representa.

### 16.5.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Retorna el valor que representa.

***type***

Signatura**:** `const TypeValue &type() const`

Descripción**:** Retorna el tipo asociado a la constante.

## *16.6 RuleNode*

La clase *RuleNode* extiende a la clase *SyntaxNode*. Representa una regla de la especificación. Dentro de sus componentes se encuentran la expresión que corresponde a la demora, el valor que debe retornar la regla, y la expresión de lógica trivalente asociada.

### 16.6.1 Responsabilidades:

Albergar la demora y el valor asociados a la expresión.

### 16.6.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Retorna la evaluación de la expresión asociada.

***type***

Signatura**:** `const TypeValue &type() const`

Descripción**:** Retorna BoolType::TheBool.

***checkType***

Signatura**:** `bool &checkType() const`

Descripción**:** Chequea que la expresión este correcta y luego verifica el tipo de sus componentes.

## *16.7 CountNode*

La clase *RuleNode* extiende a la clase *SyntaxNode*. Representa una regla de la especificación. Dentro de sus componentes se encuentran la expresión que corresponde a la demora, el valor que debe retornar la regla, y la expresión de lógica trivalente asociada.

### 16.7.1 Responsabilidades:

Albergar la demora y el valor asociados a la expresión.

### 16.7.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Retorna la evaluación de la expresión asociada.

***type***

Signatura**:** `const TypeValue &type() const`

Descripción**:** Retorna BoolType::TheBool.

***checkType***

Signatura**:** `bool &checkType() const`

Descripción**:** Chequea que la expresión este correcta y luego verifica el tipo de sus componentes.

## *16.8 SpecNode*

La clase *SpecNode* extiende a la clase *SyntaxNode.* Representa una lista de reglas. Esta clase aplica el pattern Composite [Gamm95].

### 16.8.1 Responsabilidades:

Agregar nuevas reglas.

Encontrar la regla que haya sido satisfecha.

Proyectar la demora y el valor de la regla satisfecha.

### 16.8.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Itera por todas las reglas buscando la que sea satisfecha, dependiendo del modo en que se este ejecutando el simulador buscar la primer regla que evalúa a verdadero o además valida que no haya otra que pueda ser satisfecha. Si se produce un error de este tipo se lanza un **InvalidEvaluation**.

***checkType***

Signatura**:** `bool &checkType() const`

Descripción**:** Itera por todas las reglas validando el chequeo de tipos.

***value***

Signatura**:** `int value() const`

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

***delay***

Signatura**:** `int delay() const`

Descripción**:** Retorna la demora asociada a la regla que fue satisfecha.

## *16.9 OpNode*

La clase *OpNode* extiende a *SynNode*. Esta clase representa la clase base paramétrica para los nodos que representan operaciones.

### 16.9.1 Responsabilidades:

Proveer la interfaz para especificar el tipo de la operación (pasado por parámetro).

### 16.9.2 Métodos Principales:

***type***

Signatura**:** `const TypeValue &type() const`

Descripción**:** Retorna el tipo especificado en el parámetro del template.

## *16.10 UnaryOpNode*

La clase *UnaryOpNode* extiende a *OpNode*. Esta clase representa la clase base paramétrica para los nodos que representan operaciones unarias. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>

### 16.10.1 Responsabilidades:

Evaluar la operación (pasado por parámetro) en el resultado de la evaluación de una expresión.

### 16.10.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Evalúa la expresión asociada y al valor obtenido le aplica la operación especificada en la creación. Retorna el resultado de la operación.

***checkType***

Signatura**:** `bool &checkType() const`

Descripción**:** Valida el tipo de la expresión contra el tipo especificado en los parámetros de creación de la clase.

## *16.11 BinaryOpNode*

La clase *BinaryOpNode* extiende a *OpNode*. Esta clase representa la clase base paramétrica para los nodos que representan operaciones binarias. Los parámetros de la clase son <Operación, Tipo de retorno, Tipo de los parámetros>

### 16.11.1 Responsabilidades:

Evaluar la operación (pasado por parámetro) en el resultado de la evaluación de dos expresiones.

### 16.11.2 Métodos Principales:

***evaluate***

Signatura**:** `int evaluate()`

Descripción**:** Evalúa las expresiones asociadas y a los valores obtenidos le aplica la operación especificada en la creación. Retorna el resultado de la operación.

***checkType***

Signatura**:** `bool &checkType() const`

Descripción**:** Valida el tipo de la expresión contra el tipo especificado en los parámetros de creación de la clase.

## *16.12 Nodos para operaciones relacionales*

Las operaciones relacionales aprovechan las clases unarias y binarias paramétricas definiéndose en función de ella a continuación mostramos la definición.

- `typedef BinaryOpNode<equal_to<int>,BoolType,IntType> EqualNode`
- `typedef BinaryOpNode<not_equal_to<int>,BoolType,IntType> NotEqualNode`
- `typedef BinaryOpNode<less< int >, BoolType, IntType > LessNode`
- `typedef BinaryOpNode<greater<int>,BoolType,IntType> GreaterNode`
- `typedef BinaryOpNode<less_equal<int>,BoolType,IntType> LessEqualNode`
- `typedef BinaryOpNode<greater_equal<int>,BoolType,IntType> GreaterEqualNode`

## *16.13 Nodos para operaciones aritméticas*

Las operaciones relacionales aprovechan las clases unarias y binarias paramétricas definiéndose en función de ella a continuación mostramos la definición.

- `typedef BinaryOpNode<plus<int>,IntType,IntType> PlusNode`
- `typedef BinaryOpNode<minus<int>,IntType,IntType> MinusNode`
- `typedef BinaryOpNode<divides<int>,IntType,IntType> DividesNode`
- `typedef BinaryOpNode<multiplies<int>,IntType,IntType> MultipliesNode`

## *16.14 Nodos de lógica trivalente*

Las operaciones relacionales aprovechan las clases unarias y binarias paramétricas definiéndose en función de ella a continuación mostramos la definición.

- `typedef UnaryOpNode<TVB_not,BoolType,BoolType> NOTNode`
- `typedef BinaryOpNode<TVB_and,BoolType,BoolType> ANDNode`
- `typedef BinaryOpNode<TVB_or,BoolType,BoolType> ORNode`

## *16.15 TValBool*

La clase *TValBool* representa el del tipo que soporta lógica trivalente. Así como el resto de los tipos básicos define las constantes del tipo (constantes de clase), ellas son:

- `TValBool::ttrue`
- `TValBool::tfalse`
- `TValBool::tundef`

### 16.15.1 Responsabilidades:

Extender las operaciones booleanos para lógica trivalente.

### 16.15.2 Métodos principales:

***operator &&***

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

Descripción**:** Aplica el operador lógico AND con la siguiente regla:

| AND | **T** | **F** | **?** |
|---|---|---|---|
| **T** | T | F | ? |
| **F** | F | F | F |
| **?** | ? | F | ? |

***operator ||***

Signatura**:** `TValBool operator || ( const TValBool & ) const`

Descripción**:** Aplica el operador lógico OR con la siguiente regla.

| OR | **T** | **F** | **?** |
|---|---|---|---|
| **T** | T | T | T |
| **F** | T | F | ? |
| **?** | T | ? | ? |

***operator !***

Signatura**:** `TValBool operator !() const`

Descripción**:** Aplica el operador lógico NOT con la siguiente regla.

| NOT | |
|---|---|
| **T** | F |
| **F** | T |
| **?** | ? |

# 17 Clases auxiliares

## *17.1 Time*

La clase *Time* representa al tipo hora. Esta clase provee las operaciones necesarias para operar con el tipo. Provee dos variables de clase que representan el valor de la hora zero (*Time::Zero*) y el valor infinito (*Time::Infinity).*

### 17.1.1 Responsabilidades:

Conocer la hora, minutos, segundos y milisegundos de un instante determinado.

Determinar si dos instancias representan el mismo valor.

Determinar si una instancias es menor que otra.

Aplicar la suma y resta de horas.

### 17.1.2 Métodos principales:

***send***

Signatura: `Time( Hours = 0, Minutes = 0, Seconds = 0, MSeconds = 0 )`

Signatura: `Time( const string & )`

Signatura: `Time( float mseconds )`

Descripción: Construyen una instancia a partir de los cuatro componentes explícitamente o a partir de una cadena de la forma "HH:MM:SS:MSEC" o de un total de milisegundos.

***hours***

Signatura: `Time &hours( const Hours & )`

Signatura: `const Hours &hours() const`

Descripción: Registran/proyectan el valor de la hora.

***minutes***

Signatura: `Time &minutes( const Minutes & )`

Signatura: `const Minutes  &minutes() const`

Descripción: Registran/proyectan el valor de los minutos.

***seconds***

Signatura: `Time &seconds( const Seconds & )`

Signatura: `const Seconds  &seconds() const`

Descripción: Registran/proyectan el valor de los segundos.

***mseconds***

Signatura: `Time &mseconds( const MSeconds & )`

Signatura: `const MSeconds &mseconds() const`

Descripción: Registran/proyectan el valor de los milisegundos.

***operator +***

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

Signatura: `const MSeconds &mseconds() const`

Descripción: Retorna una nueva instancia de *Time* con el valor de la suma.

***oeprator -***

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

Signatura: `const MSeconds &mseconds() const`

Descripción: Retorna una nueva instancia de *Time* con el valor de la resta.

***operator ==***

Signatura: `bool operator ==( const Time & ) const`

Signatura: `const MSeconds &mseconds() const`

Descripción: Retorna si al instancia tiene los mismos valores que el parámetro especificado.

***operator <***

Signatura: `bool operator <( const Time & ) const`

Signatura: `const MSeconds &mseconds() const`

Descripción: Retorna si la instancia es menor que el parámetro especificado.

## 17.2 Ini

La clase *Ini* representa un archivo de texto separado en secciones donde cada sección a su vez especifica definiciones de nombres de variables y sus respectivos valores. Esta clase es utilizada para leer la especificación del modelo a simular.

### 17.2.1 Responsabilidades:

Realizar el parseo del archivo formando los grupos y las definiciones.

Proyectar una lista de definiciones especificando el grupo.

Proyectar una lista de valores proveyendo la definición y el grupo.

### 17.2.2 Métodos principales:

***parse***

Signatura: `Ini &parse( istream & )`

Descripción: Realiza la interpretación de la especificación almacenada en la cadena de caracteres obtenida por parámetro, registrando en sus estructuras internas los grupos y secuencia de valores especificados.

***parse***

Signatura: `Ini &parse( const string & )`

Descripción: Abre el archivo con nombre especificado por parámetro, lee su contenido e invoca al método *parse* con los datos obtenidos.

***groupList***

Signatura: `const GroupList &groupList() const`

Descripción: Retorna la lista de grupos declarados en la especificación ya procesada (después de haber ejecutado el método parse sobre los datos de entrada).

***defList***

Signatura: `const DefList &group( const string & ) const`

Descripción: Retorna la lista de definiciones relacionadas con el grupo recibido por parámetro. De no existir tal grupo produce un IniException.

***definiton***

Signatura: `const IdList &`

```
definition( const string &groupName, const string &defName ) const
```

Descripción**:** Retorna la lista de valores relacionadas con el grupo/definición recibidos por parámetro. De no existir tal grupo o la definición dentro del grupo produce un IniException.

#### ***addGroup***

Signatura**:** `Ini &addGroup( const string &groupName )`

Descripción**:** Agrega el grupo a la lista de grupos existentes. Este método es usado principalmente por el método *parse* para introducir los datos después de haber procesado la especificación de entrada.

#### ***addDefinition***

Signatura**:** `Ini &addDefinition( const string &grp, const string &def )`

Descripción**:** Agrega el par grupo/definición a la lista de grupos y definiciones existentes. En caso que el grupo especificado no exista, simplemente se crea e instancia con el nuevo valor. Este método es usado principalmente por el método *parse* para introducir los datos después de haber procesado la especificación de entrada.

#### ***addId***

Signatura**:** `Ini &addId(const string &g,const string &def,const string &id)`

Descripción**:** Agrega un valor especificado en el parámetro al par grupo/definición también recibidos por parámetro y que ya fueron agregados anteriormente. En caso que el grupo no exista o que la lista de definiciones no esté incluidas en el grupo, simplemente se crean e instancian en el nuevo valor. Este método es usado principalmente por el método *parse* para introducir los datos después de haber procesado la especificación de entrada.

#### ***exists***

Signatura**:** `bool exists( const string &groupName ) const`

Descripción**:** Retorna verdadero en caso que el nombre de grupo pasado por parámetro exista como un grupo ya definido dentro de la estructura interna de la instancia. En caso contrario retorna el valor lógico: falso.

#### ***exists***

Signatura**:** `bool exists( const string &grp, const string &def ) const`

Descripción**:** Retorna verdadero en caso que el nombre de grupo / nombre de definición pasados por parámetro exista como un par grupo / definición ya definido dentro de la estructura interna de la instancia. En caso contrario retorna el valor lógico: falso.

## *17.3 CellPosition*

La clase *CellPosition* representa un simple par de fila/columna, e implementa todas las operaciones comunes para este nuevo tipo de dato (suma, resta, operaciones booleanas, etc.).

### 17.3.1 Responsabilidades:

Administrar sus dos componentes (fila y columna).

Realizar operaciones matemáticas entre objetos de este tipo.

Realizar operaciones matemáticas entre objetos de este tipo.

### 17.3.2 Métodos principales:

#### ***row***

Signatura: `int row() const`

Descripción: Retorna el valor correspondiente a la fila.

***col***

Signatura: `int col() const`

Descripción: Retorna el valor correspondiente a la columna.

***operator ==***

Signatura: `bool operator ==( const CellPosition & ) const`

Descripción: Retorna el valor de verdad de la igualdad entre el par fila/columna del parámetro y los valores de instancia.

***operator +***

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

Descripción: Retorna una nueva instancia de la clase CellPosition, tal que el valor de sus coordenadas (fila/columna) son la suma de los valores de instancia mas los especificados en el parámetro.

***operator -***

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

Descripción: Retorna una nueva instancia de la clase CellPosition, tal que el valor de sus coordenadas (fila/columna) son la resta de los valores de instancia menos los especificados en el parámetro.

***operator <***

Signatura: `bool operator <( const CellPosition & ) const`

Descripción: Retorna verdadero si los valores fila/columna de la instancia son menores a los valores fila/columna especificados en el parámetro.

***parseString***

Signatura: `CellPosition &parseString( const string & )`

Descripción: Obtiene y registra del parámetro los nuevos valores para fila/columna. El formato de la cadena de entrada deberá ser: “(y,x)”, y en caso de no poseerlo, se producirá un AssertException.

## 17.4 CellState

La clase *CelState* es la encargada de almacenar y administrar el estado de las celdas en un autómata celular.

### 17.4.1 Responsabilidades:

Administrar el estado de las celdas.

Configurar el espacio celular en forma circular o plana.

Dar comportamiento especial a las celdas fuera del espacio celular

Manejo de regiones.

### 17.4.2 Colaboradores:

CellState::Row: administra filas de estado de celdas.

### 17.4.3 Métodos principales:

#### ***width***

Signatura: `int width() const`

Descripción: Retorna la cantidad de columnas que posee el espacio celular.

#### ***height***

Signatura: `int height() const`

Descripción: Retorna la cantidad de filas que posee el espacio celular.

#### ***isWrapped***

Signatura: `bool isWrapped() const`

Descripción: Retorna el valor lógico: verdadero si el tipo del espacio celular es circular, de otra manera retorna falso.

#### ***includes***

Signatura: `bool includes( int row, int col ) const`

Signatura: `bool includes( const CellPosition & ) const`

Descripción: Retorna el valor lógico: verdadero si la posición indicada se encuentra dentro del espacio celular. Si el tipo es circular, cualquier posición pertenecerá al espacio celular, y en consiguiente siempre retornara verdadero.

#### ***calcRealRow***

Signatura: `void calcRealRow( int &row ) const`

Descripción: Si el tipo es circular, modifica el parámetro ajustándolo a valores de filas dentro del espacio celular.

#### ***calcRealCol***

Signatura: `void calcRealCol( int &col ) const`

Descripción: Si el tipo es circular, modifica el parámetro ajustándolo a valores de columnas dentro del espacio celular.

#### ***calcRealPos***

Signatura: `void calcRealPos( CellPosition &pos ) const`

Signatura: `void calcRealPos( int &row, int &col ) const`

Descripción: Invoca a los métodos *calcRealRow* y *calcRealCol* para ajustar los valores de fila/columna en caso que el espacio celular sea de tipo circular.

#### ***operator[]***

Signatura: `TValBool &operator[]( const CellPosition & )`

Descripción: Retorna el elemento indicado por la posición indicada en el parámetro. De no ser de tipo circular y estar fuera de las dimensiones del espacio, retorna un elemento *TValBool::tundef*.

## *17.5 NeighborhoodValue*

La clase *NeighborhoodValue* es una vista del vecindario de cada celda. Se crea y actúa directamente sobre encargada de almacenar y administrar el estado de las celdas en un autómata celular.

### 17.5.1 Responsabilidades:

Administrar el estado del vecindario de cada una de las celdas.

Actualizar el estado de la celda central del vecindario

### 17.5.2 Colaboradores:

CellState: administra el espacio celular.

### 17.5.3 Métodos principales:

***get***

Signatura: `const TValBool &get( const NeighborPosition &n ) const`

Signatura: `const TValBool &get() const`

Descripción: Retorna el estado del vecino indicado en el parámetro. De no recibirse el parámetro, se retorna el estado de la celda que hizo el requerimiento, o sea la celda dueña (central) del vecindario.

***set***

Signatura: `NeighborhoodValue &set( const TValBool & )`

Descripción: Registra en el estado de la celda central del vecindario el valor indicado en el parámetro.

# 18 Simulator (clase principal)

La clase *Simulator* es la responsable de crear el árbol de modelos/procesadores y los vínculos entre sus puertos a partir de una especificación.

Para realizar esta tarea toma los streams de entrada del loader y utiliza la clase *Ini* para realizar el parsing de la especificación de los modelos. La carga de modelos se realiza a partir de la definición de un modelo Root, es por esto que si no se encuentra ninguna etiqueta en la especificación de modelos que indique la definición de Root (top) el simulador aborta mostrando el error correspondiente.

El entorno de simulación se completa con los stream de salida para el logging de mensajes y el output del simulador. Una vez armado el entorno de simulación la instancia de *Simulator* le informa a la instancia de *Root* los eventos externos especificados y la hora a la cual debe detener la simulación (por defecto es infinito).

En este momento se invoca el método run() de *Root* para que la simulación de comienzo y termina recién cuando llegue la hora de terminación o todos los modelos que componen la simulación se encuentren en estado pasivo y no se esperen nuevos eventos externos.

### 18.1.1 Responsabilidades:

Creación del árbol de modelos/procesadores en base a la especificación obtenida a través de la clase Ini.

Creación y vinculación de los puertos de todos los modelos que pertenezcan a la especificación.

Creación de celdas para todos los modelos celulares.

Registro de estados iniciales para todos los modelos celulares.

Registro de funciones de transición para todas las celdas de todos los modelos celulares.

### 18.1.2 Colaboradores:

**Ini**: Realiza la interpretación de los archivos de configuración.

**SimLoader**: Es la base de la jerarquía de clases cuyo objetivo es la obtención de los datos de configuración de la simulación, y el posterior retorno de los resultados hacia el usuario.

**Root**: Procesador de mas alto nivel que controla toda la simulación.

**LocalTransAdmin**: Administrador de lenguajes de especificación utilizados para el comportamiento de las celdas.

### 18.1.3 Métodos Principales:

#### ***run***

Signatura**:** `MainSimulator &run()`

Descripción**:** Lleva a cabo todo el ciclo del simulador. Se encarga de la obtención de los parámetros (especificación de modelos, eventos externos y hora de finalización de la simulación), y da comienzo a la simulación invocando al método *simulate* de la clase *Root*. Por último, le solicita a la jerarquía de loaders que envíe los resultados a donde le haya sido indicado con anterioridad.

#### ***loadModels***

Signatura**:** `MainSimulator &loadModels( istream& )`

Descripción**:** Invoca al método *parse* de la clase *Ini* con los datos obtenidos por el loader, quien ya obtuvo los datos de entrada (especificación de modelos y eventos externos) de acuerdo al tipo de loader que haya sido generado. Una vez que la especificación se encuentra procesada y disponible por la clase *Ini*, se procede a invocar al método *loadModel* para que cree en forma jerárquica todos los modelos/procesadores que componen la simulación empezando por *Root*, que es el procesador de mayor nivel.

#### ***loadModel***

Signatura**:** `MainSimulator &loadModel( Coupled &, Ini & )`

Descripción**:** Se invoca al método *loadPorts* para que cree los puertos que posee el modelo que se está creando (obtenido por el parámetro). Luego, de acuerdo a si el modelo es un celular o un modelo básico, se invoca al método *loadCells* o *loadComponents*. Por último se procede a crear los links internos al modelo y de conexión con el exterior a través del método *loadLinks*.

#### ***loadPorts***

Signatura**:** `MainSimulator &loadPorts( Coupled &, Ini & )`

Descripción**:** Itera a través de las definiciones encontradas en la clase Ini y genera todos los vínculos relativos al modelo (tanto de entrada como de salida).

#### ***loadComponents***

Signatura**:** `MainSimulator &loadComponents( Coupled &, Ini & )`

Descripción**:** Crea todos los modelos atómicos que formen parte del acoplado y los vincula en la relación padre-hijo. Para cada hijo que sea de tipo acoplado hace una llamada recursiva al método *loadModel*, hasta llegar al final de la jerarquía de modelos especificadas en el archivo de configuración.

### ***loadCells***

Signatura: `MainSimulator &loadCells( CoupledCell &, Ini & )`

Descripción**:** Obtiene a través de llamadas a la clase Ini los datos de configuración del modelo celular (tipo: común o achatado, ancho, alto, es circular, etc.) y en base a esto, después de haber registrado todos estos valores en el modelo celular, invoca al método *createCell* de la clase *CupledCell* (o *FlatCupled*) para que cree todas las celdas y le asigne la configuración correcta. Luego invoca al método *loadInitialCellValues* para que actualice el valor inicial de las celdas que así lo tienen configurado en los datos de parametrización. Por último se invoca al método *loadLocalZones* para que le informe al modelo acoplado celular cuales son las celdas que tienen comportamiento particular, y cual es este (función de transición local).

### ***loadLinks***

Signatura: `MainSimulator &loadLinks( Coupled &, Ini & )`

Descripción**:**. Crea los vínculos entre dos puertos, en relación al modelo que se está configurando. Existen tres tipos de vínculos:

Vínculos internos: se trata de una conexión entre dos puertos que pertenecen a modelos hijos del modelo acoplado que se está configurando.

***Vínculos de Entrada: el primer puerto pertenece al acoplado y el segundo a un modelo*** hijo.

Vínculos de Salida: el primer puerto se encuentra en un hijo y el otro pertenece al ***acoplado.***

### ***loadInitialCellValues***

Signatura: `MainSimulator &loadInitialCellValues( CoupledCell &, Ini & )`

Descripción**:**. Para todas aquellas filas que sean configuradas en los datos de parametrización con valores diferentes al especificado por defecto, se modifica el estado inicial a través del método *setCellValue* de la clase *CoupledCell* (o *FlatCoupled*) de cada una de ellas. Si en el archivo de configuración fue especificado el valor para una fila que no pertenece al espacio celular, o la cantidad de columnas especificada no concuerda se produce un AssertException.

### ***loadLocalZone***

Signatura: `MainSimulator &loadLocalZones( CoupledCell &, Ini & )`

Descripción**:** Se registra para cada celda que pertenezca a alguna zona especificada en el archivo de configuración, la función de transición local que esté especificada para dicha zona.

### ***registerTransition***

Signatura: `MainSimulator &registerTransition(const Function &, Ini & )`

Descripción**:** Para cada especificación de función de transición local se genera una cadena con la especificación del lenguaje y se invoca al método *registerTransition* de la clase *SingleLocalTransAdmin*, quien se encargará de hacerlo interpretar, revisar su sintaxis, etc. y almacenar para ser evaluada cuando corresponda.

# 19 Simulación por dentro

En esta sección describiremos las distintas etapas por las que atraviesa el simulador para cumplir con los requerimientos del usuario.

## 19.1 Inicio y carga de modelos

La simulación comienza con la invocación del usuario especificando los argumentos por línea de comando. El método principal *main* toma los parámetros y analiza buscando si se desea invocar al simulador en modo servidor (ver manual de usuario). Una vez determinado el modo en que el usuario desea ejecutar la aplicación se instancia de acuerdo a ello el tipo de *SimLoader* que corresponda. Se le indica a la única instancia de *MainSimulator* cual es el cargador que debe utilizar y luego se le indica que comience el ambiente de simulación con el método *run.*

*MainSimulator* utiliza el cargador para obtener la configuración del simulador y con esto carga los modelos y los eventos externos dejando así el entorno preparado para que comience la simulación. A esta altura la jerarquía de *Processor-Model* está creada y lista para comenzar. Le indica a *Root* la hora de finalización especificada y da por comienzo la simulación invocando el método *simulate()*. Al finalizar se le indica al cargador que retorne los resultados obtenidos.

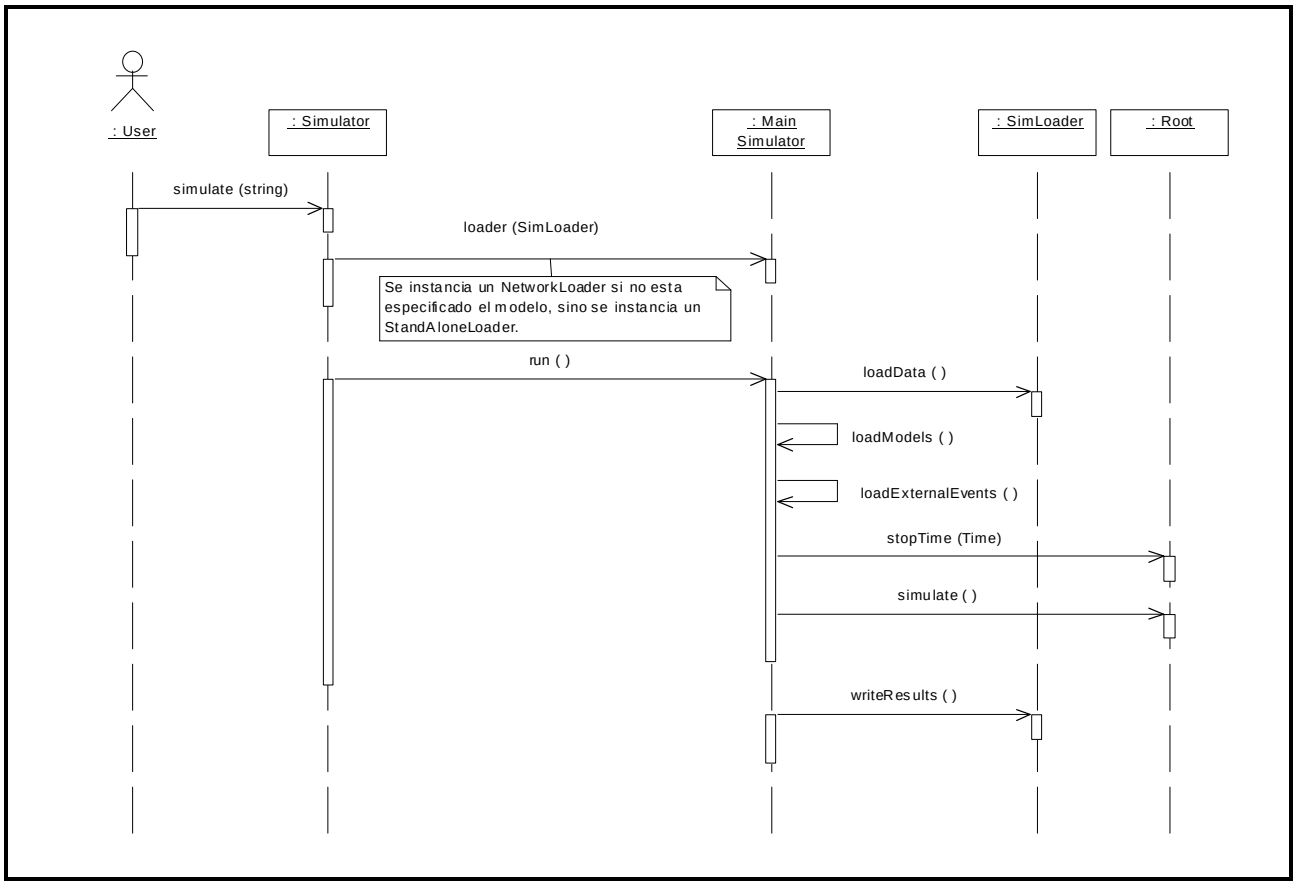



**Ilustración 8 – Vista global de la simulación**

## 19.2 Comienzo de la simulación

La simulación comienza con el envío de un mensaje de inicialización al acoplado de mayor nivel. El acoplado de mayor nivel reenvía el mensaje a todos sus hijos y espera un mensaje *done* de cada uno de ellos con la hora de su próximo evento. Una vez recibidos todos los mensajes en respuesta a los enviados, calcula cual de todos sus hijos es el inminente, es decir el que tenga la hora de próximo evento menor, y envía un mensaje *done* a su padre indicando su hora de próximo evento interno.

Cuando root recibe un mensaje *done* toma la hora del mensaje como la hora del próximo cambio de estado de su hijo inminente (top, otro no tiene), y la compara con el próximo evento externo que tiene programado. Se queda con el menor de los dos y envía en un mensaje en función de ello. Si es un evento

externo enviará un *ExternalMessage* y si es un interno enviará un *InternalMessage*, actualizando la hora global del simulador (ver ilustración 9).

Si la planificación del hijo inminente propone como hora de próximo cambio infinito, esto significa que todos los modelos que componen la simulación se han pasivado. En este punto la falta de eventos externos marca el fin de la simulación. Otra posibilidad de finalización puede darse si la hora del próximo evento elegido es mayor o igual que la hora de finalización especificada.

Este es el algoritmo general que aplica root para tratar a los mensajes done.

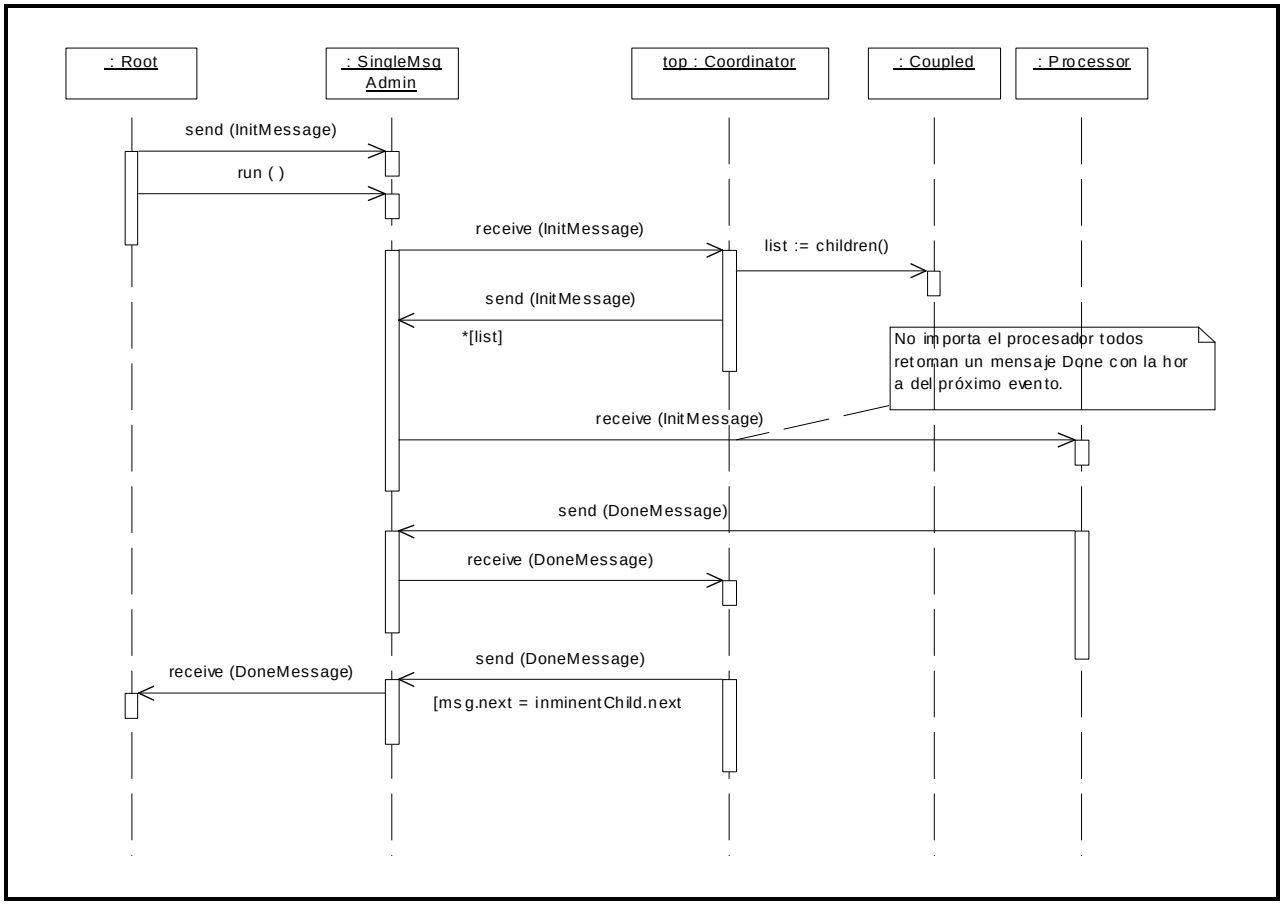



**Ilustración 9 – Ruteo de un mensaje de inicialización**

## *19.3 Eventos internos*

*Root* envía un *InternalMessage* al coordinador de mayor nivel indicando el arribo de un evento interno, es decir un cambio de estado programado para alguno de sus descendientes. Este mensaje llega al coordinador *top* y este lo redirecciona hacia el hijo inminente, este hijo es potencialmente cualquier especialización de *Processor*: *Simulator, Coordinator, CellCoordinator, FlatCoordinator*.

Una instancia de la clase *Simulator* al recibir un mensaje interno invoca el método *outputFunction()* y luego *internalFunction()* de su modelo atómico asociado. Al retornar de este último método arma un *DoneMessage* en respuesta indicando la hora de su próximo cambio de estado. Si el modelo atómico genera una salida esta será enviada por su simulador asociado a su coordinador padre vía el administrador de mensajes.

Una instancia de la clase *Coordinator* responde a un evento redireccionando el mensaje hacia el procesador hijo inminente. Cuando reciba el *DoneMessage* de su hijo toma la hora de su hijo inminente y la envía en un mensaje done hacia su procesador padre.

En la ilustración número diez (y de aquí en adelante) se obviará el administrador de mensajes y se mostrará la invocación y el resultado obtenido de su intervención ya que su funcionalidad esta descripta por completo en la ilustración del inicio de la simulación.

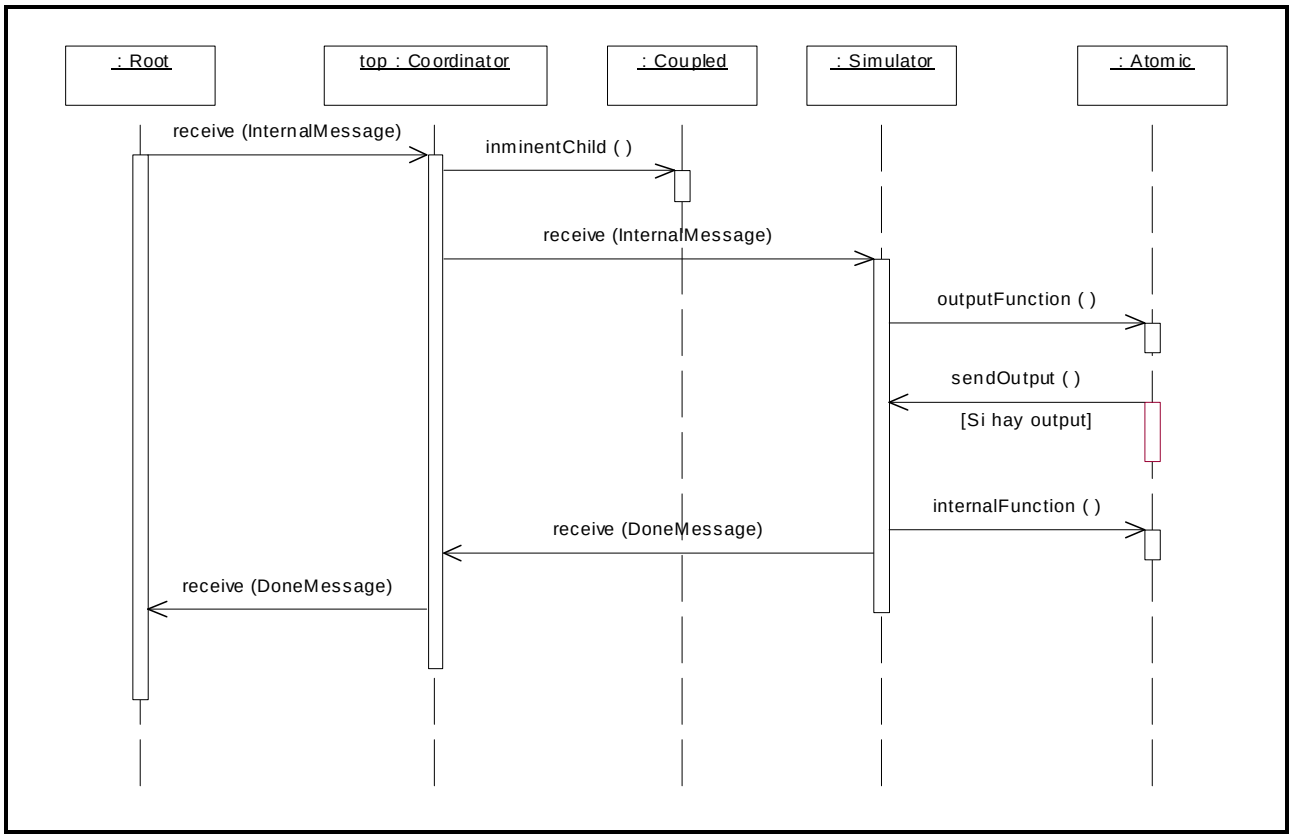



**Ilustración 10 – Mensaje interno a un Simulator**

## *19.4 Eventos externos*

*Root* envía un mensaje externo a su coordinador de mayor nivel indicando el puerto por el cual este arriba. Una instancia de *Coordinador* que recibe un mensaje externo debe analizar el mapeo de las influencias sobre el puerto destino del mensaje. Para esto utiliza la lista de influencia que posee el *Coupled* asociado. Por cada influencia se genera y envía un nuevo mensaje externo con el valor del mensaje externo original. Por último el coordinador registra la cantidad de mensajes done que debe esperar para luego enviar un mensaje done a su coordinador padre.

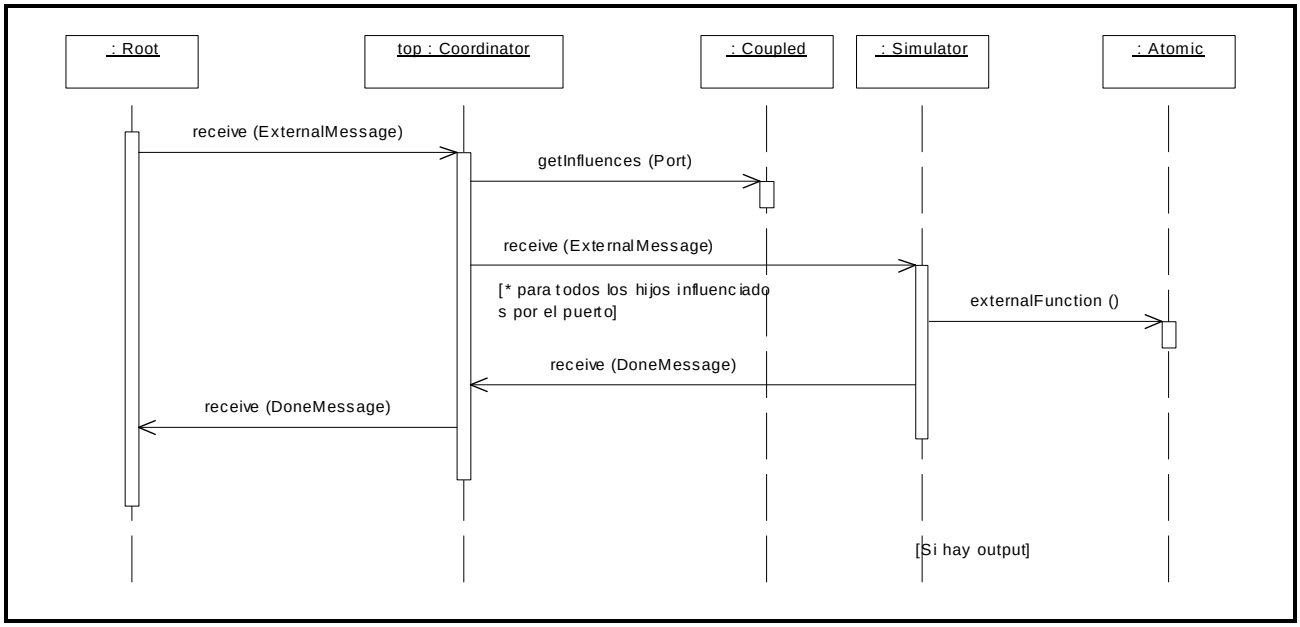



**Ilustración 11 - Mensaje externo**

## 19.5 Eventos de salida

Una instancia de *Coordinator* que recibe un mensaje de salida debe analizar las influencias del puerto por el cual se produce la salida. Si la influencia tiene destino en un modelo externo al coordinador debe reenviarse el mensaje de salida hacia el coordinador padre. El otro destino posible para la influencia es un modelo dentro del coordinador. En este caso el mensaje de salida es traducido a un *ExternalMessage* y enviado por el puerto que indique la dependencia esperando un *DoneMessage* de cada uno. Cuando arriben todos y cada uno de ellos retornará un mensaje done con la hora del hijo inminente al coordinador padre.

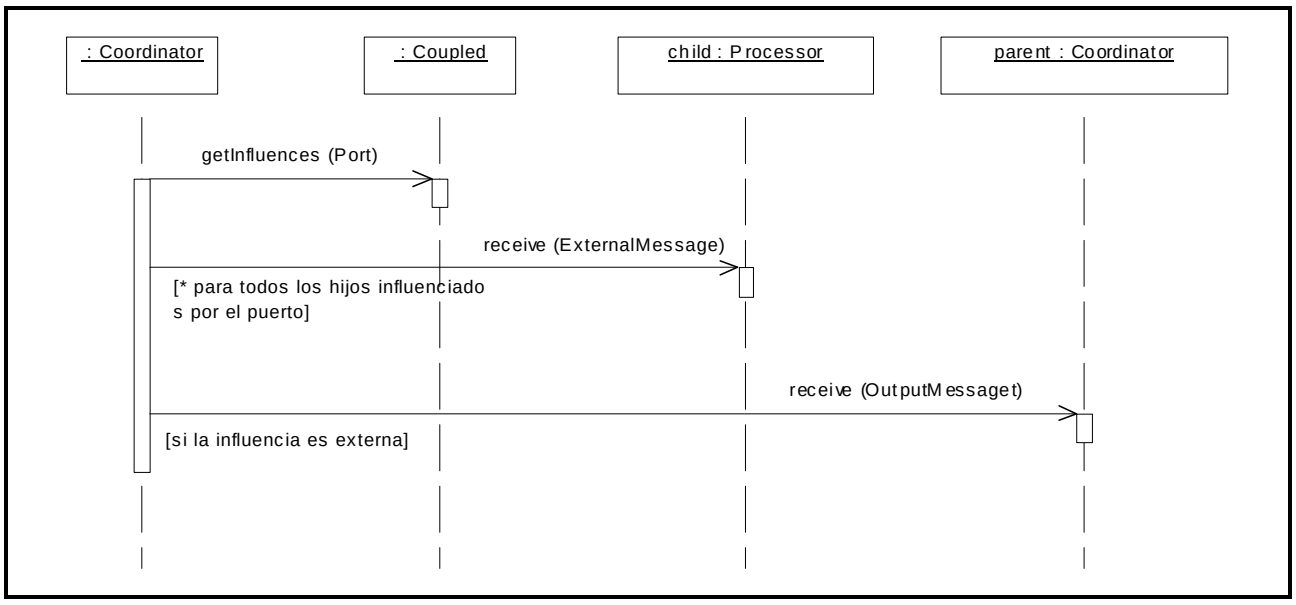



**Ilustración 12 - Mensaje de salida**

## 19.6 Modelos Celulares

La clase *CellCoupled* al igual que su clase base maneja a todos las celda como modelos hijos. A diferencia de los modelos acoplados comunes todos los hijos posibles tienen la misma estructura. En la

creación de las celdas se le asigna el comportamiento especificado a cada una y se la vincula con los modelos que especifique su vecindario. El comportamiento de las celdas está compuesto por dos factores, el tipo de demora y la función de transición local que a diferencia de los modelos atómicos tradicionales se especifica a través del lenguaje propietario GADCelLa (GAD Cell Language, se lee GODZILLA).

La clase *CellCoordinator* da un comportamiento especial a la recepción de los eventos internos y de salida. Frente a un mensaje interno se enviarán tantos *InternalMessage* asincrónicamente como hijos inminentes existan. Des esta forma se fuerza el orden en que los mensajes externos disparados por la salidas de las celdas serán tratados evitando así que una celda ejecute más de una vez en el mismo instante de tiempo. Para lograr esto el coordinador celular sobrecarga la recepción de los mensajes de salida guardando temporalmente las celdas ya afectadas para evitar los mensajes externos duplicados que en los modelos celulares solamente significan un recálculo del estado de la celda. Los menajes externos varían la semántica cuando provienen de un vecino ya que indican solamente un aviso de recálculo y no un valor externo.

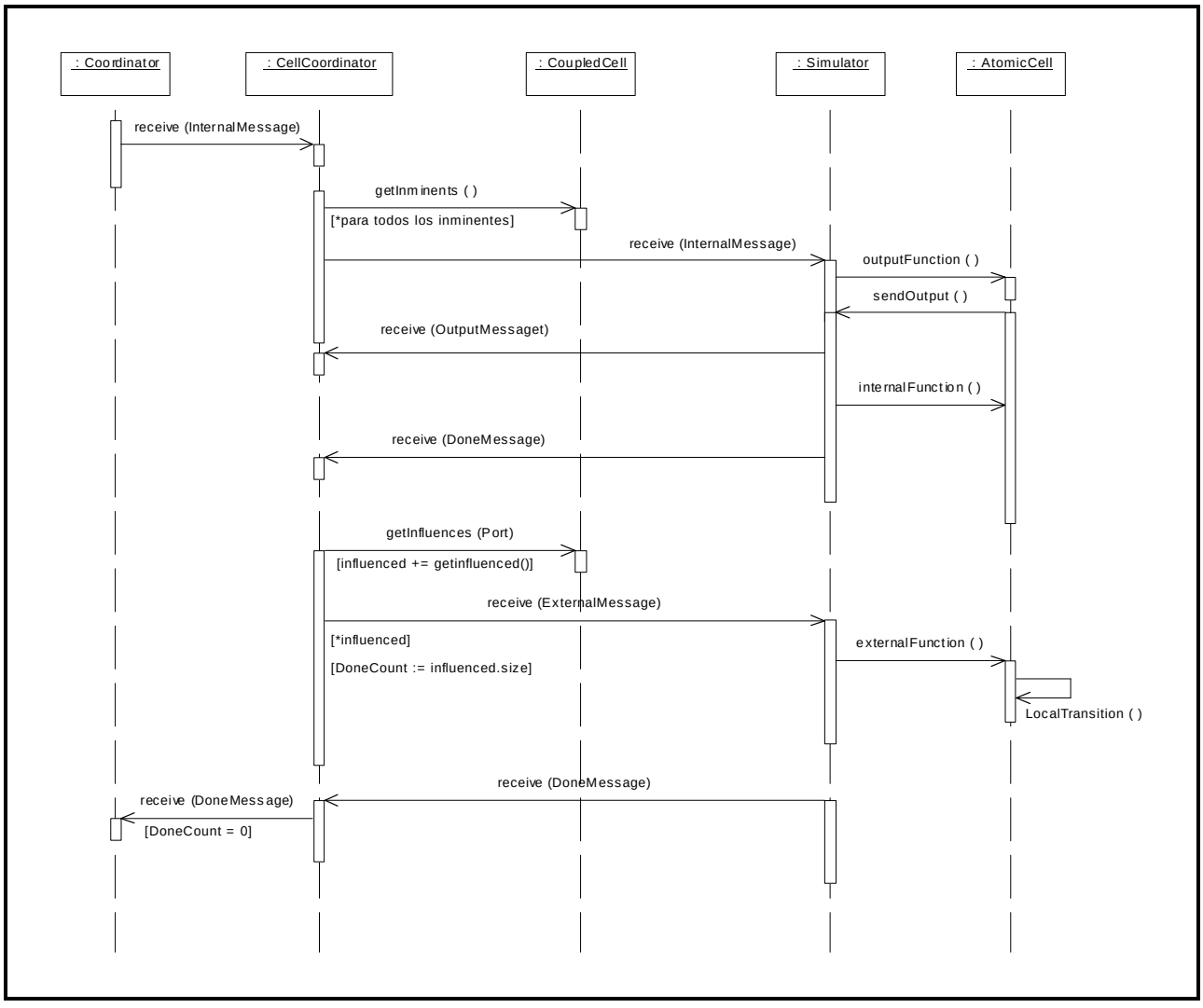



**Ilustración 13 - Evento para Modelos Celulares**

## *19.7 Modelos Celulares Achatados*

Los modelos celulares convencionales (implementados a través de DEVS) están compuestos de tantos modelos básicos como celdas posea el autómata. Esto produce una gran inversión de tiempo en la creación y una sobrecarga de mensajes por cada evento que arriba al modelo celular. Dado que todas las

celdas comparten la misma estructura, contienen los mismos puertos de entrada/salida, tienen el mismo comportamiento en función de su estado, y es posible obtener sus vecinos aplicando un desplazamiento a su ubicación (teniendo en cuenta el borde, que puede ser plano o circular), toda la estructura necesaria para llevar a cabo la simulación de este tipo de modelo puede simplificarse a una matriz de estados celulares junto a una especificación del lenguaje a utilizar. Esta simplificación permite disminuir drásticamente el tiempo de creación y elimina por completo el pasaje de mensajes dentro del modelo celular.

Cuando una instancia de *FlatCoordinator* recibe un mensaje externo invoca al método *externalFunction* del acoplado celular achatado asociado para todas las celdas virtuales (le indica la posición dentro del espacio de celdas) influencias por el puerto donde fue recibido el mensaje. El acoplado asociado calculará la función de transición local para la posición indicada y aplicará el algoritmo de demora correspondiente modificando la lista de próximos eventos en caso de ser necesario. Al finalizar se envía un *DoneMesage* al coordinador padre con la hora del hijo inminente (también virtual).

En el caso de recibir un *internalMessage* el *FlatCoordinator* invoca al método *internalFunction* del acoplado celular achatado asociado. Este último itera por todas las celdas con hora de próximo evento igual a la hora actual, y genera su output agregando todas las celdas influenciadas por la salida a la lista de próximos eventos. Por último se ejecuta el método *externalFunction* para todas las celdas virtuales que figuren en esta lista.

# 20 Ejemplos

## *20.1 Modelos atómicos y acoplados*

El objetivo de esta sección es ilustra el uso de los modelos atómicos y acoplados. Para ello construiremos un Transducer para realizar mediciones de throughput y uso de cpu en un esquema simple de un computador monoprocesador.

El transductor estará compuesto por un generador de trabajos que emula a un usuario produciendo procesos en la computadora con una distribución dterminada, una cola que representa al sistema operativo aplicando su política de no remoción de procesos y un procesador que consume el pedido.

Para ello describiremos primero las clases componentes y luego mostraremos el diagrama de acoplamiento final y los resultados obtenidos.

### 20.1.1 Queue

Una cola es un dispositivo de almacenamiento temporal con política FIFO (First In First Out). Para implementarlo usando GAD se debe crear una nueva clase *Queue* que extienda a *Atomic*.

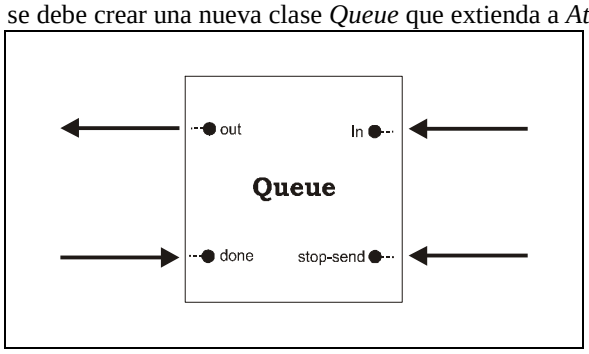



**Ilustración 14 - Diagrama de una cola**

Debe poseer un puerto por de entrada por el cual el resto de los modelos ingresen los elementos a ser almacendos 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 via el 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*.

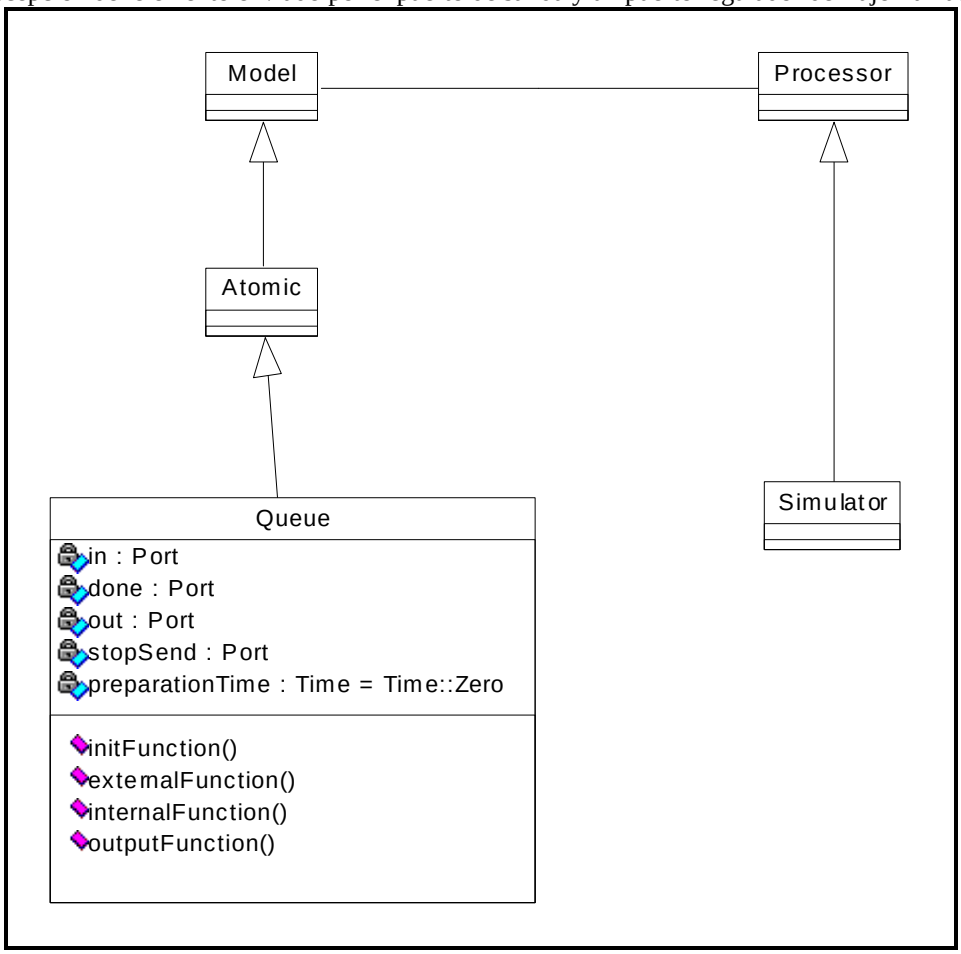



**Ilustración 15 - Jerarquia de Queue**

El modelo *Queue* sobrecarga los métodos de inicialización, función de transición interna, externa y salida. En la inicialización toman el valor incial sus variables, se eliminan todos los valores de la cola interna.

```
Model &Queue::initFunction()
{
   this->elements.erase( elements.begin(), elements.end() ) ;
   return *this ;
}
```

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 rgistra 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;
   }
```

La función de salida indica que ha finalizado el tiempo de preparación para el primer elemento de la cola y se envá este por el puerto *out*. Luego es ejecutada la función de transición interna indicando que se ha termindo 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 ;
   }
```

#### 1.1.1.1 Generator

Un *generator* imita la generación de trabajos que realiza un usuario en un computador. Por esto uno de sus parámetros es el tipo de distribución probabilistica con la que realiza esta tarea.

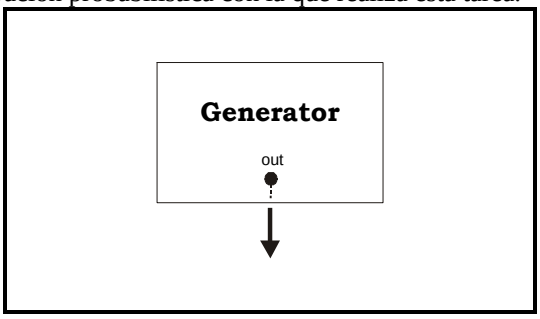



**Ilustración 15 - Generador de Trabajos**

EL modelo *Generator* extiende a la clase *Atomic* reprogramandose en función de los valores obtenidos de la distribución probabilistica indicada. Esta reprogramación se realiza por medio de la inicialización y la función de transición interna. Eventos externos no pueden ocurrir ya que el modelo no posee puertos de entrada.

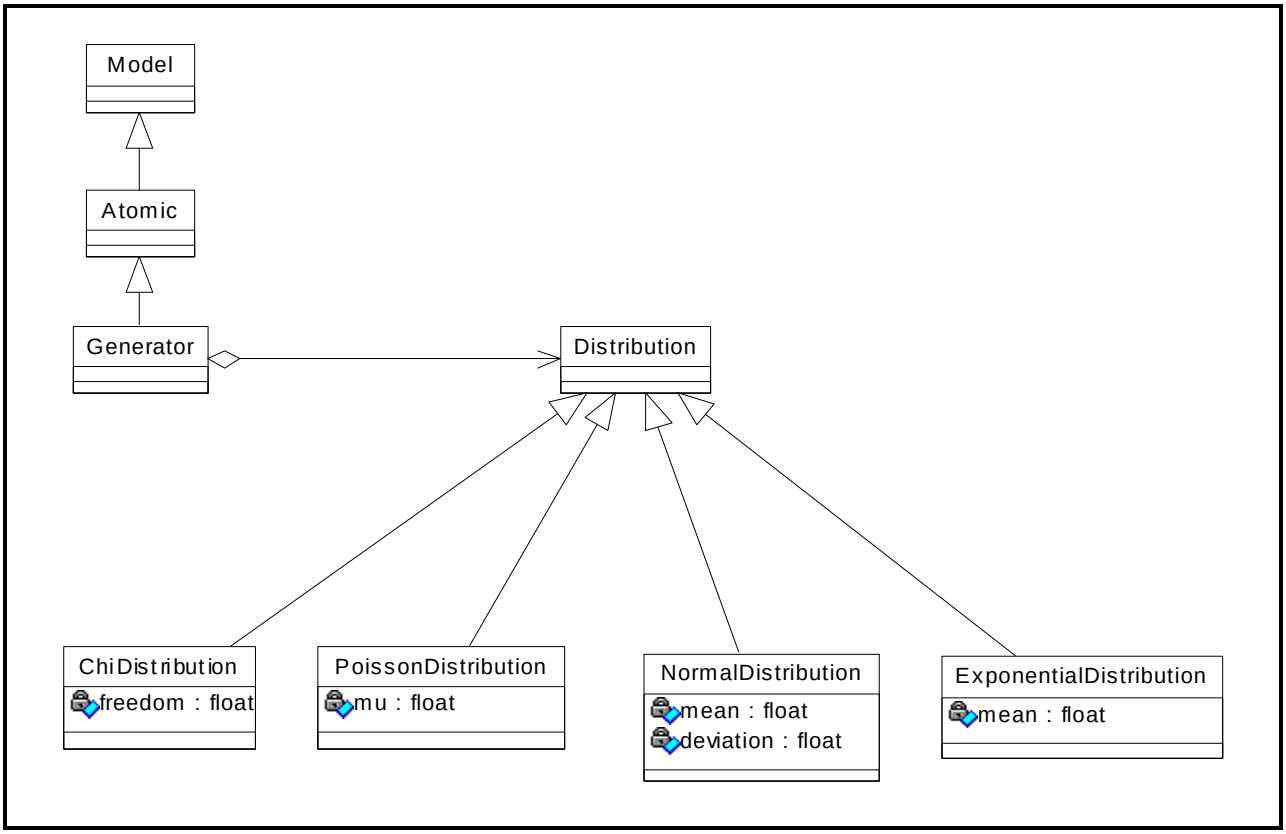



**Ilustración 16 – Jerarquía del Generador de Trabajos**

EL código de los métodos mas importantes es el siguiente:

```
Model &Generator::initFunction()
{
   this->holdIn( active, Time::Zero ) ;
   return *this ;
}

Model &Generator::internalFunction( const InternalMessage & )
{
   this->holdIn( active, this->distribution().get() );
   return *this ;
}

Model &Generator::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, this->newProcessId() ) ;
   return *this ;
}
```

#### 1.1.1.2 CPU

Un *CPU* imita el comportamiento de un procesador en un computador. Recibe procesos y demora en procesarlos un tiempo obtenido por una distribución probabilistica que es indicada incialmente.

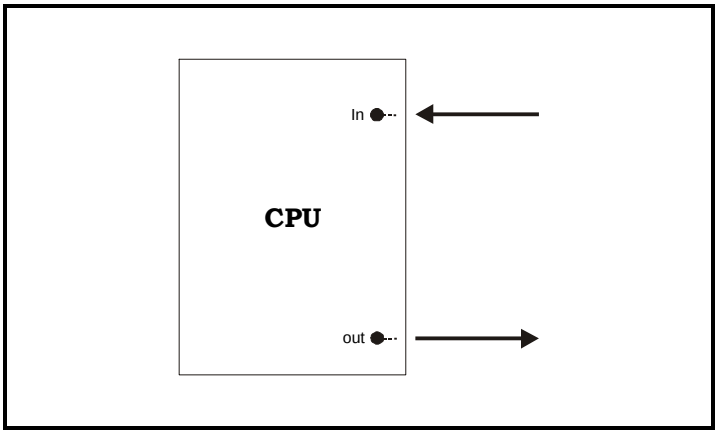



**Ilustración 17 – Consumista de Trabajos**

EL modelo *CPU* extiende a la clase *Atomic* recibiendo eventos externos por su puerto de entrada *in* que representan procesos que deben ser llevados a cabo por el procesador. El tiempo que le llevará realizar el trabajo recién arribad dependerá del tipo y parámetros de la distribución probabilistica con la que esté especificado el modelo. Al concluir el procesamiento del trabajo se informa su finalización por el puerto de salida *out*.

```
Model &CPU::externalFunction( const ExternalMessage &msg )
{
   pid = msg.value();
   this->holdIn( active, this->distribution().get() );
   return *this ;
}

Model &CPU::internalFunction( const InternalMessage & )
{
   this->passivate() ;
   return *this ;
}

Model &CPU::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, pid ) ;
   return *this ;
}
```

### 1.1.1.3 Transducer

Un *Transucer* analiza e informa las medidas observadas para un procesador en un tiempo determinado. El análisis del transductor permite obtener dos resultado al finalizar el período evaluativo. El primero es el throughput del sistema (cantidad de trabajos por unidad de tiempo) y el segundo el promedio de uso del procesador.

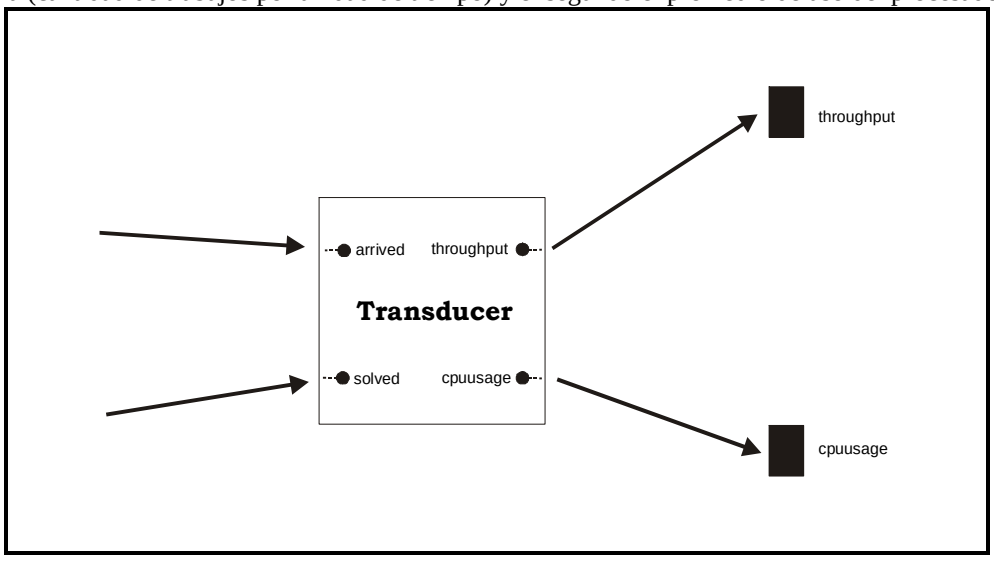



**Ilustración 18 –Transductor**

EL modelo *Transducer* extiende a la clase *Atomic* recibiendo eventos externos por su puertos *arrived* y

*solved*. En el primero se recibe la información sobre la creación de un nuevo trabajo (con su identificador de proceso dentro del mensaje), y por el otro se recibe el anuncio que un trabajo ha terminado su procesamiento. La registración de estos dos puertos le permite al transductor realizar los calculos para producir los dos resultados deseados, enviándolos por sendos puertos de salida a intervalos fijos.

```
Model &Transducer::initFunction()
{
   procCount = 0 ;
   cpuLoad = 0 ;
   unsolved.erase() ;
   this->holdIn( active, this->frecuence() ) ;
   return *this ;
}

Model &Transducer::externalFunction( const ExternalMessage &msg )
{
   cpuLoad += ( msg.time() - this->lastChange() ).asMsecs() * unsolved.size() ;
   if( msg.port() == arrived )
       unsolved[ msg.value() ] = msg.time() ;
   if( msg.port() == solved )
   {
      JobsList::iterator cursor( unsolved.find( msg.value() ) ) ;

       procCount ++ ;
       unsolved.erase( cursor ) ;
   }
    return *this ;
}

Model &Transducer::internalFunction( const InternalMessage &)
{
   this->holdIn( active, this->frecuence() ) ;
   return *this ;
}

Model &Transducer::outputFunction( const InternalMessage &msg )
{
    float time( msg.time().asMsecs() / this->timeUnit().asMsecs() ) ;
   this->sendOutput( msg.time(), throughput, procCount / time ) ;
   cpuLoad += ( msg.time() - this->lastChange() ).asMsecs() * unsolved.size() ;
   this->sendOutput( msg.time(), cpuUsage  , cpuLoad / msg.time().asMsecs() ) ;
   return *this ;
}
```

### 1.1.1.4 Transductor Completo

Una vez definidos todos los modelos básicos necesarios para la integración del modelo se procederá a incorporarlos en en un modelo acoplado realizando las tareas antes descriptas.

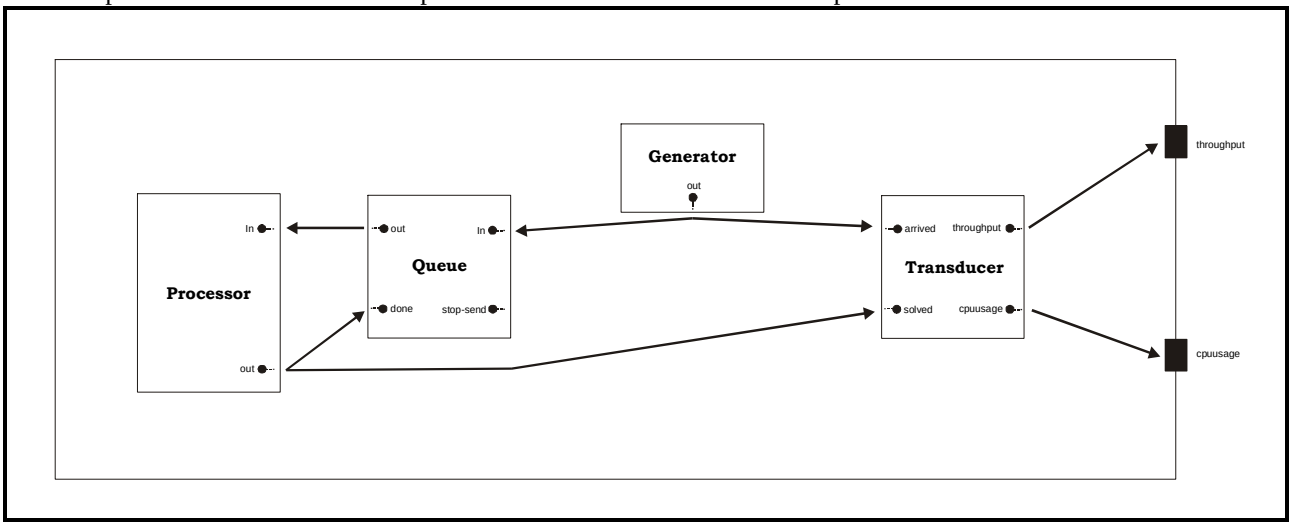



**Ilustración 19 – Modelo Acoplado Transuctor**

Para definir el acoplamiento dentro del modelo es necesario especificar en un archivo los componentes utilizados y la relación entre ellos. A continuación podrá observarse el archivo de configuración correspondiente. En él se puede observar un primer grupo con el rótulo *top* indicando que se especifica el acoplado de mayor nivel. Este grupo es de especificación obligatoria en todos los archivos de configuración de modelos. En las primeras líneas se encuentran la descripción de los componentes y sus relaciónes (links). El resto de los grupos representan configuraciones adicionales para los modelos.

```
[top]
components : Queue@queue Processor@CPU Transducer@transducer Generator@generator
Out : throughput
Out : cpuusage
Link : out@generator arrived@transducer
Link : out@generator in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor solved@transducer
Link : throughput@transducer throughput
Link : cpuusage@transducer cpuusage

[transducer]
frecuence : 0:0:2:0

[generator]
distribution : poisson
mean : 10

[CPU]
distribution : exponential
mean : 10

[queue]
preparation : 0:0:0:0
```

#### 1.1.1.5 Resultados

Se realizaron varias pruebas combinando la distintas distribuciones con distintos parámetros. En todos los casos se observó un mejor rendiemiento del procesador (velocidad de respuesta y trabajos por unidad de tiempo) cuando la media del tiempo consumido en procesar es menor o igual que la media del tiempo consumido en generación de trabajos.

En el gráfico correspondiente a la simulación con parámetros de generación Poisson 10 y proceso Chi$^2$ 30 la media del tiempo de proceso (30 seg.) es mayor al tiempo medio de generación (10 seg.). Esto produce que los trabajos se enconlen tendiendo a aumentar la cantidad de tareas esperando por el procesador. El valor de throughput tiende a dos procesos por unidad de tiempo ya que tiene una media de procesamiento de 30 seg. y el muestreo tiene base de tiempo un minuto.

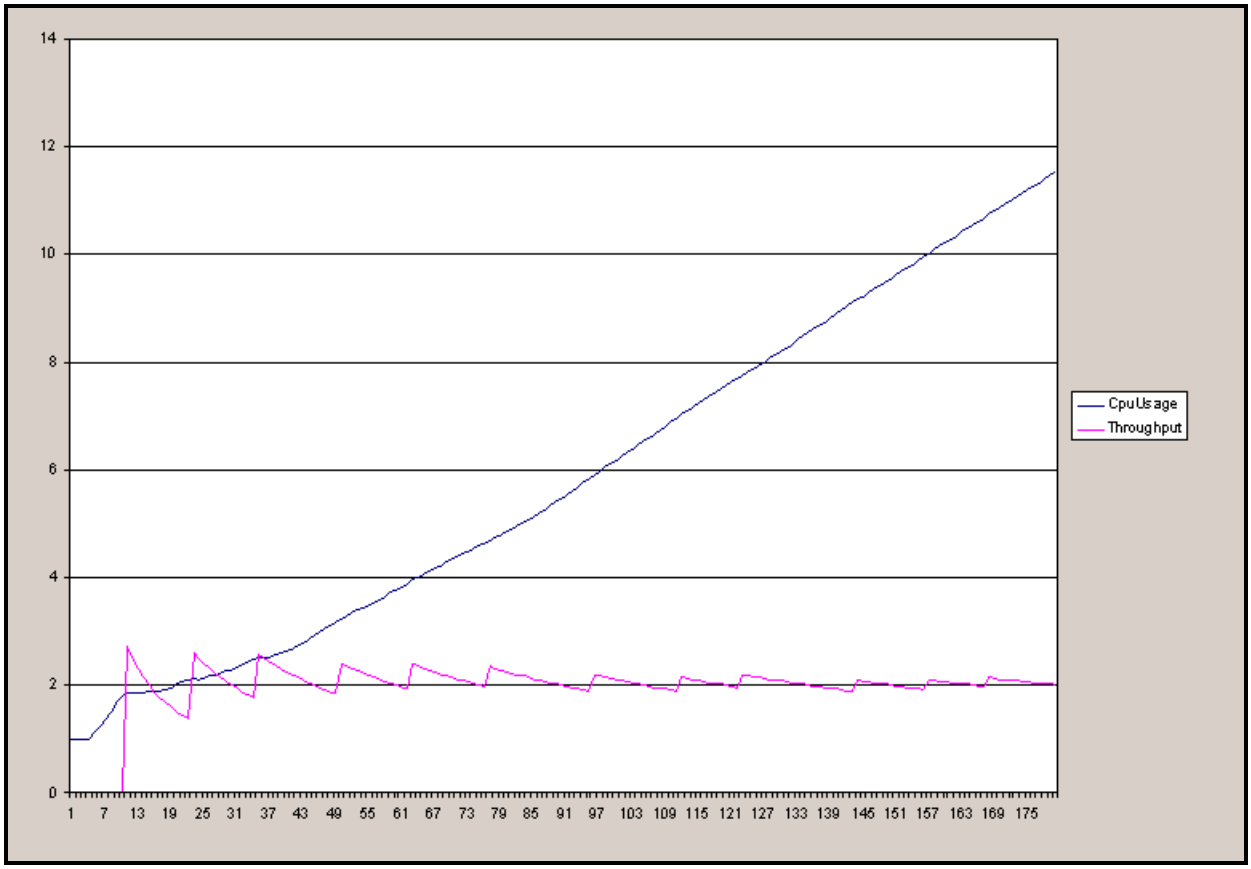



**Ilustración 20 – Generador con distribución Poisson 10 y Procesador con Chi$^2$ 30**

En los dos gráficos correspondientes al Generador con Poisson 10 y al Processador con distribuciones Exponencial 10 y Chi$^2$ 10 se observa un comportamiento similar ya que en ambos el throughput tiende a un valor de seis procesos por minutos pero afectan el encolamiento de nuevos trabajos.

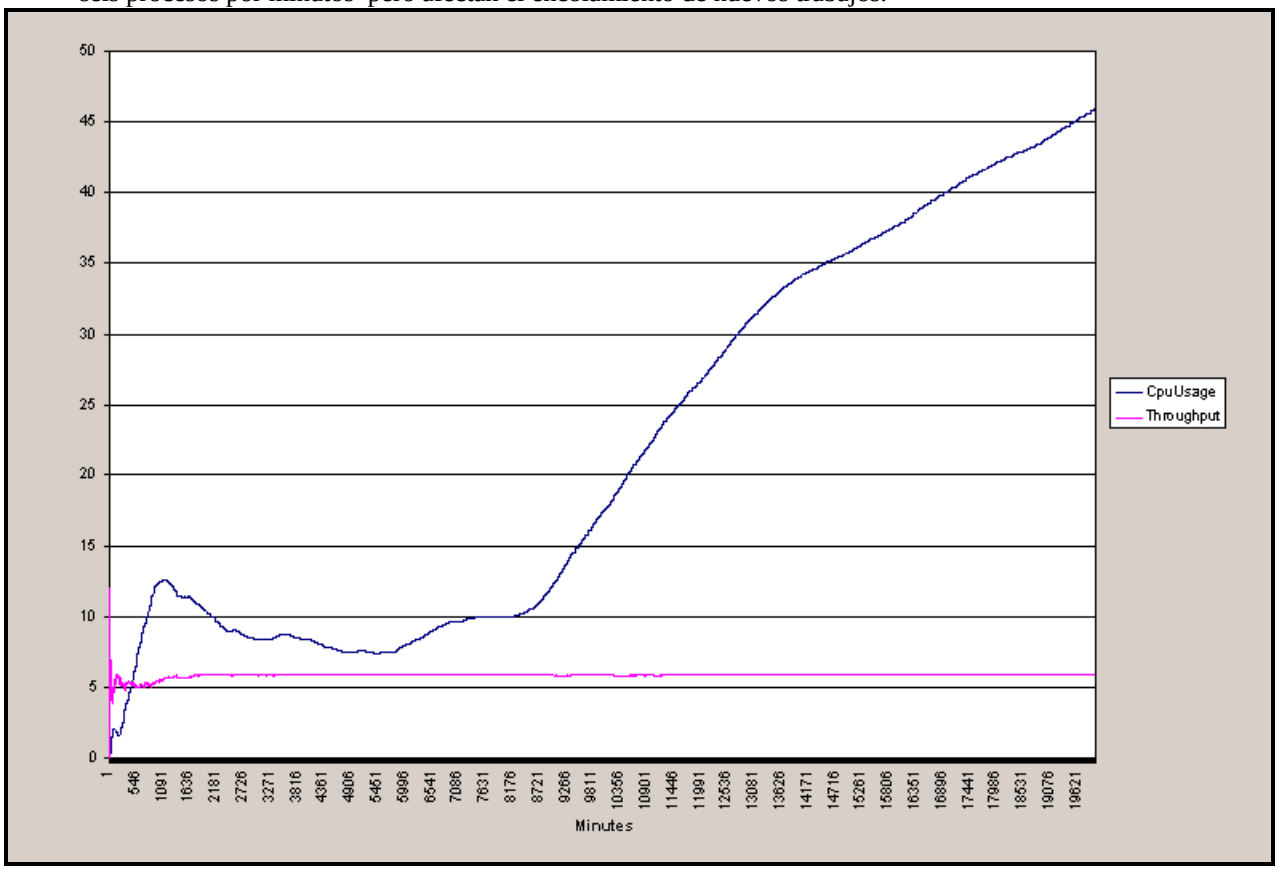



**Ilustración 21 - Poisson 10 y Exponencial 10**

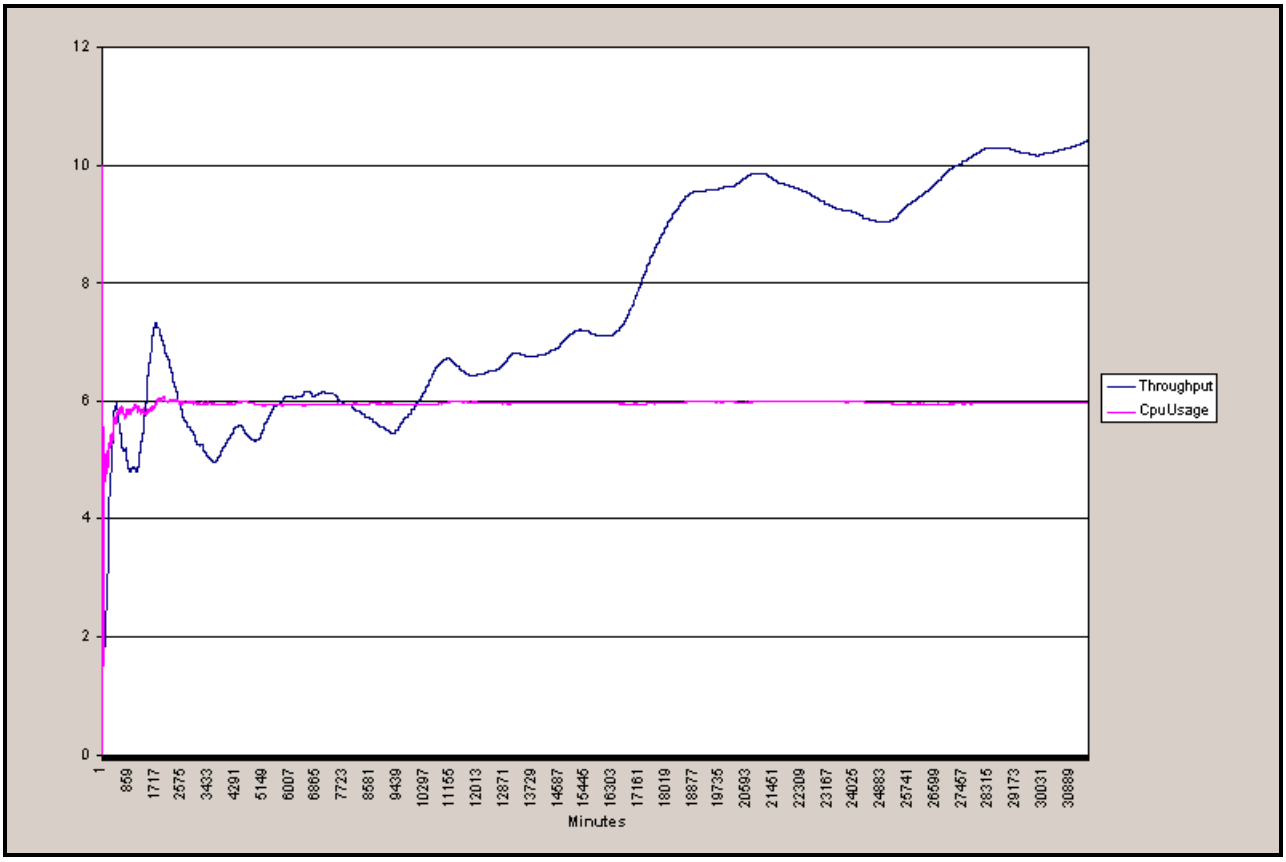



**Ilustración 22 - Poisson 10 y Chi$^2$ 10**

En el último gráfico tomando el generador con Poisson 30 y el procesador Exponencial 10 se observa que el throughput tiende a dos y el CPU esta libre la mayor parte del tiempo ya que consume los trabajos mucho más rápido que la frecuencia de aparición de nuevas tareas.

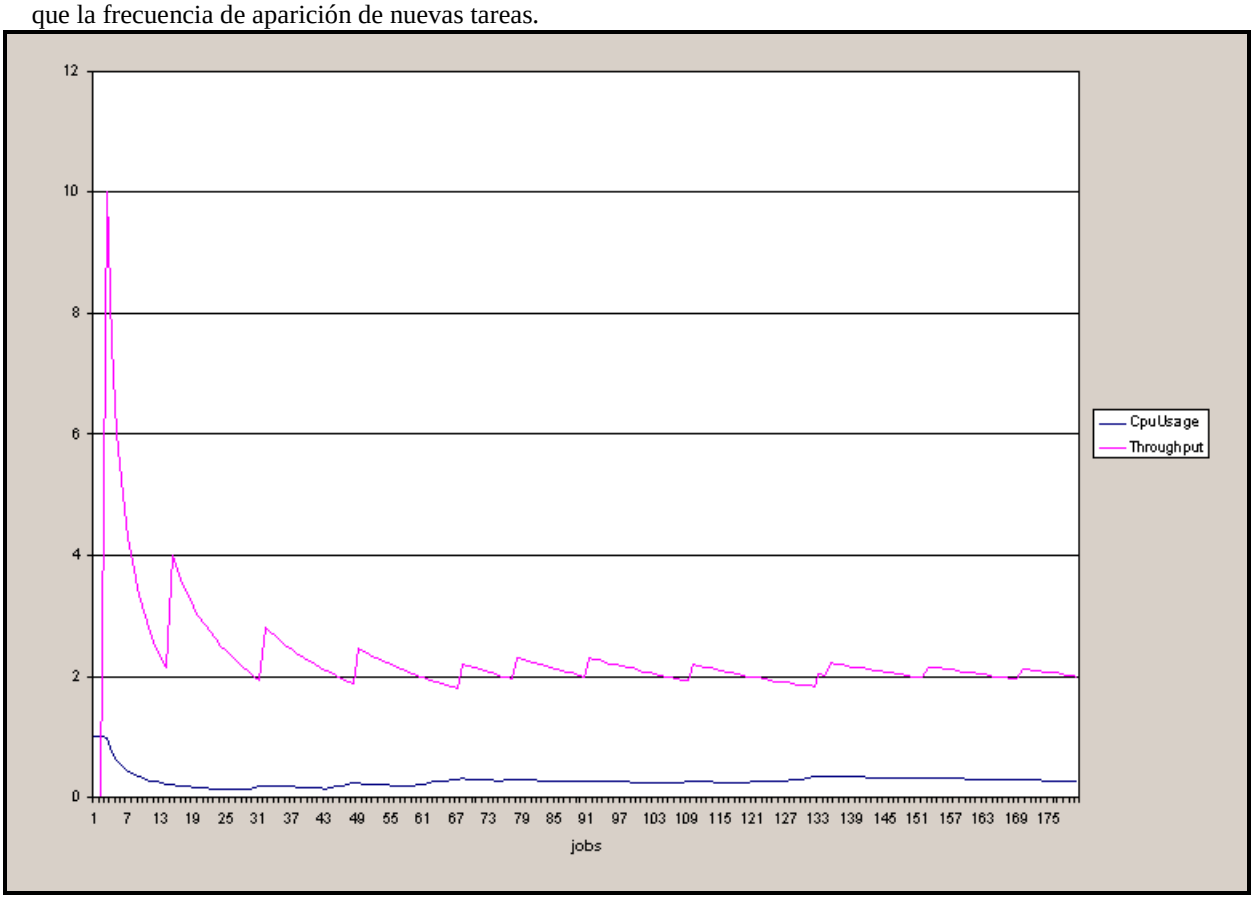



**Ilustración 23 - Poisson 30 y Exponencial 10**

## *20.2 Modelos Celulares*

### 20.2.1 Life game

El juego de la vida se desarrolla sobre una matriz de mxn valores binarios. Cuando una celda tiene el valor uno se dice que esta viva, en otro caso (valor cero) está muerta. El juego se desarrolla en base a una especificación inicial de celdas vivas y muertas. A partir de aquí comienzan las iteraciones aplicandos las siguientes reglas:

Una celda viva permanecerá viva si tiene cuatro vecinos vivos de otro modo fallece.

Una celda muerta revivirá si tiene exactamente tres vecinos vivos.

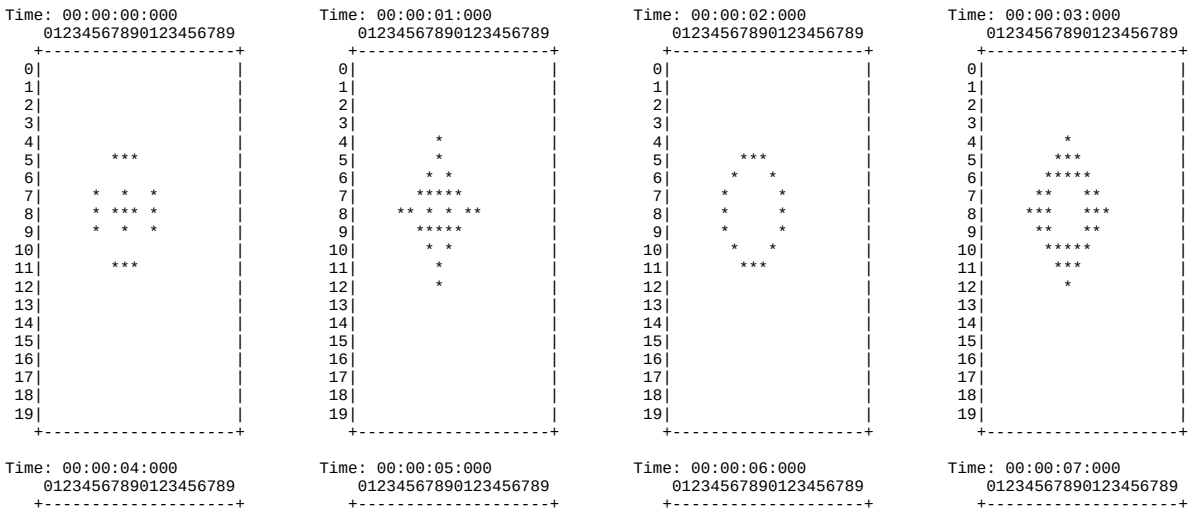
Este juego se modela utilizando autómatas celulares especificando el estado inicial y luego las reglas común a todas las celdas y con la misma demora. La especificación es la siguiente:

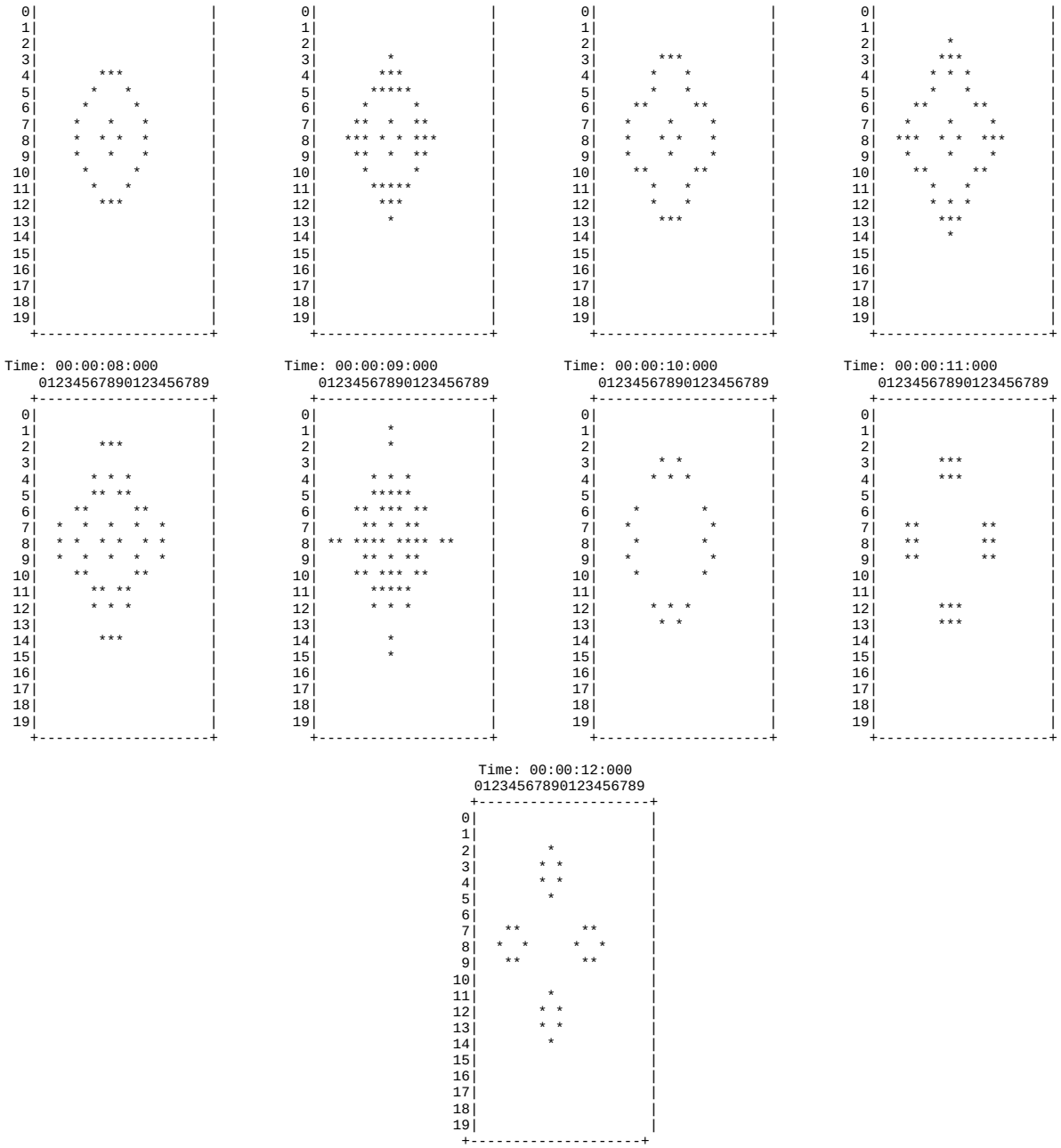
```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 1000
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 :  5  00000001110000000000
initialrowvalue :  7  00000100100100000000
initialrowvalue :  8  00000101110100000000
initialrowvalue :  9  00000100100100000000
initialrowvalue : 11  00000001110000000000
localtransition : conrad-rule

[conrad-rule]
rule : 1 1000 { (0,0) = 1 and (truecount = 3 or truecount = 4 ) }
rule : 1 1000 { (0,0) = 0 and truecount = 3 }
rule : 0 1000 { t }
```

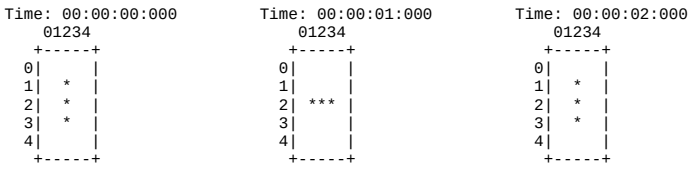
En la muestra inicial puede observarse la distribución inicial especificada en el archivo de configuración. Cada iteración se produce a intervalos de un segundo avanzando según las reglas hasta llegar al segundo doce donde se estabiliza. Los resultados fueron compardos con los del LifeGame V1.1 desarrollado sobre plataforma PalmPilot por Sean D. True, este puede ser contactado por email a mailto://seant@iname.com, obteniendo los mismos resultados.





Estas  prueba fueron realizadas utilizando modelos celulares comunes. Los mismos resultados pueden obtenerse utilizando modelos celulares achatados. Para ilustrar la diferencia de tiempo de ejecución que existe entre ambas implementaciónes detallaremos un nuevo ejemplo que no se estabiliza ya que cada solo posee dos estados, y poseen secuencia cíclica. Es decir, del primer estado se pasa al segundo a través de una interación, y luego por medio de una nueva iteración se vuelve al primer estado, y asi continua hasta llegar la hora de detención de la simulación.

Los estado son los siguientes:



Los tiempos de procesamiento del modelo celular achatado versus el modelo celular no achatado son:

- Modelo celular normal:48 segundos

- Modelo celular achatado: 15 segundos

Los siguientes tiempos fueron obtenidos mediante la ejecución del simulador en ambiente windows 95, con un cpu Intel mmx 233Mhz, 64Mb ram.

### 20.2.2 Tráfico en el Mall

# Manual del Desarrollador

## 21 Introducción

Este manual describe los procedimientos que deben realizarse para incorporar nuevos modelos atómicos a la entorno de simulación. Esto es necesario ya que para los modelos atómicos debe codificarse en C++ el comportamiento deseado. Para el resto de los modelos simulables (acoplados y celulares) por la herramienta es suficiente con la especificación a través de un archivo de inicialización; en este archivo también figura la definición de la vinculación de los modelos y de los componentes que lo forman. Esta información se encuentra disponible en el manual del usuario.

## 22 Ambiente de desarrollo

El lenguaje elegido para implementar el diseño del simulador fue C++.

Los fuentes del simulador se dividen en dos partes:

Librería de simulación.

Fuente principal y modelos atómicos.

Las rutinas de la librería llevan a cabo por completo todos los pasos de la simulación y se complementan con el fuente principal y los archivos de configuración para cargar el modelo a simular y configurar aspectos particulares de la plataforma en la cual el simulador correrá.

Todos los fuentes fueron compilados con el compilador gcc versión 2.8.x (http://www.sunsite.unc.edu/pub/gnu) del proyecto GNU Free Software Foundation Inc. (http://www.fsf.org) y testeados en las siguientes plataformas:

Linux Slackware v3

SunOs v5

Dos bajo Win95 (solo la versión stand-alone / sin soporte de red)

Los términos de distribución y copia son los mismos que aplica GNU para todas sus herramientas. Para mas información referirse a www.gnu.org.

El código usado respeta el standard ANSI C++ (http://www.ansi.org), e incluye las características recientemente incorporadas al lenguaje como **template function, C++ style cast y STL (Standard Template Library)**. La portabilidad del compilador permite realizar la compilación sin ningún cambio para la versión standalone en cualquier plataforma soportada. Los servicios de red fueron implementados vía una clase que abstrae el canal de comunicación entre el servidor de simulación y los clientes. Esta abstracción define la interfaz que utiliza el simulador para obtener los datos y luego retornar los resultados. La implementación realizada, hasta el momento en que se redactó este informe, fue hecha sobre BSD Sockets, standard para la mayoría de los ambientes UNIX de las universidades del mundo. Otras implementaciones son fácilmente realizables y transparentes por completo para el funcionamiento del simulador haciéndolo de esta forma independiente de la plataforma y el entorno en el que está corriendo.

### *22.1 Incorporación de nuevos modelos*

Los pasos para generar un nuevo modelo atómico son los siguientes:

Diseñar una clase que herede de la clase *Atomic* y **sobrecargarndo obligatoriamente** los métodos:

*initFunction*: antes de invocar al método sigma vale infinito y el estado es pasivo.

*externalFunction*: invocado cuando arriba un evento externo en alguno de los puertos del modelo.

*internalFunction*: antes de invocar al método sigma vale cero ya que se ha cumplido el intervalo para la transición interna.

*outputFunction*: antes de invocar al método sigma vale cero ya que se ha cumplido el intervalo para la transición interna.

*className*: nombre de la clase.

Incorporar en el fuente **register.cpp**, agregándole al método *Simulator::registerNewAtomics()* el nuevo tipo de modelo atómico, por ejemplo para la clase Queue:

```
SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>(), "Queue" ) ;
```

## 22.2 Interacción con el Simulador

En cada uno de los métodos que se nombró anteriormente se deben invocar ciertas primitivas para cumplir con las normas impuestas por el simulador. Estas primitivas tienen por función interactuar con el simulador para realizar tareas comunes a todos los modelos atómicos.

La primitivas disponibles son:

**holdIn(estado, tiempo):** indica al simulador que el modelo debe mantenerse en el estado por un tiempo fijo, y que después de transcurrido ese tiempo deberá producirse una cambio de estado. El estado podrá tener valor: active / passive.

**passivate():** indica al simulador que el modelo entra en modo pasivo y que únicamente deberá ser invocado por la llegada de un evento externo.

**sendOutput(hora, puerto, valor):** envía un mensaje de salida.

**nextChange():** método de consulta para obtener el tiempo restante para su próximo cambio de estado (sigma)

**lastChange():** método de consulta para obtener la hora de su último cambio de estado.

**state():** método de consulta del estado en que se encuentra el modelo.

**getParameter( nombreModelo, nombreParámetro ):** método de acceso a los parámetros de configuración de la clase. Este método pertenece a la clase Simulator.

Al ser *Model* una clase base abstracta define la interfaz para la recepción de mensajes, la clases *Atomic* y *Coupled* son la únicas encargada de recibir y enviar mensajes. Es **responsabilidad de las clases derivadas** de *Atomic* re-implementar las funciones de inicialización, transición interna, transición externa y salida.

Las clases derivadas de *Atomic* no deben enviar ningún tipo de mensaje, salvo los valores de salida que se informan mediante el método *sendOutput*. La clase *Atomic* es la responsable de

generar los mensajes Y y D al padre para cualquier instancia derivada, utilizando los valores de *sigma* y *state*.

### 22.2.1 Ejemplo: Implementación de una cola (class Queue)

Una cola es un dispositivo de almacenamiento con política FIFO. Posee tres puertos de entrada y un puerto de salida.

En el puerto **in** arrivan los datos de entradas. Por cada valor que se envía por el puerto **out** se esperará la confirmación de su recepción en el puerto **done.**

Si el cliente de la cola se encuentra saturado, podrá manejar el control de flujo a través del puerto **stop-send**.

El primer paso para implementar un nuevo modelo atómico es definir una clase derivada de la clase *Atomic* redefiniendo lost métodos para el manejo de transiciones. Estos métodos no son públicos ya que solamente la clase base *Atomic* los puede invocar.

En el ejemplo vemos la definición de variables para hacer referencia a los puertos de la cola y el tiempo que demora en preparar el dato antes de enviarlo por el puerto de salida. Además se encuentra la definición de una lista de valores para albergar los valores ingresados y el tiempo restante ante una interrupción por control de flujo.

```
class Queue : public Atomic {
public:
   Queue(); // Default constructor
   virtual string className() const ;

protected:
   Model &initFunction();
   Model &externalFunction( const ExternalMessage & );
   Model &internalFunction( const InternalMessage & );
   Model &outputFunction( const InternalMessage & );

private:
   const Port &in;
   const Port &stop;
   const Port &done;
   Port &out;
   Time preparationTime;

   typedef list<Value> ElementList ;
   ElementList elements ;

   Time timeLeft;

}; // class Queue
```

## *22.3 Constructor*

El constructor crea los tres puertos de entrada, el puerto de salida e inicializa la variable *preparationTime*. El parámetro preparation puede estar especificado en el archivo de configuración del modelo, de ser así se modifica el tiempo de preparación.

### 22.3.1 Código:

```
Queue::Queue()
   : preparationTime( 0, 0, 10, 0 )
   , in( this->addInputPort( "in" ) )
   , stop( this->addInputPort( "stop" ) )
   , done( this->addInputPort( "done" ) )
   , out( this->addOutputPort( "out" ) )
{
   this->description( "Queue" ) ;

   string time( Simulator::Instance().getParameter(
                          this->description(), "preparation" ) ) ;

   if( time != "" )
      preparationTime = time ;
}
```

## 22.4 initFunction

La función de inicialización debe borrar la lista que guarda los elementos encolados. El próximo cambio de estado se producirá ante la llegada de un evento externo por eso sigma permanece constante. Si se desea planificar el próximo cambio de estado debe modificarse usando el método *holdIn*.

### 22.4.1 Código:

```
Model &Queue::initFunction()
{
   elements.remove( elements.begin(), elements.end() ) ;
   return *this;
}
```

## 22.5 externalFunction

La clase Queue tiene tres puertos de entrada por los cuales puede recibir eventos externos.

Si el evento arriva en el puerto *in* significa que es un dato nuevo de entrada, por lo tanto debe encolarse. Si es el único en la cola se debe preparar el dato para ser enviado.

Si el evento arriva en el puerto *done* significa que el último elemento enviado fue recibido correctamente, por lo tanto debe ser elimindado de la cola. Si hay más elementos a enviar se debe preparar el próximo dato para ser enviado.

Por último el evento puede arrivar en el puerto *stop* indicando que se desea pausar o recomenzar el trabajo de la cola. Si la cola se encuetra en estado *active* y el valor del mensaje es distinto de cero esto indica que se desea hacer una pausa, por eso se calcula cuanto tiempo faltaba procesar para el próximo cambio de estado (fin de la preparación del dato a enviar) y luego la cola cambia su estado a *passive* llamando al método *passivate*. Si la cola se encuentra en estado *passive* y el valor del mensaje es cero la cola vuelve a estar activa planificando el próximo cambio de estado en base al tiempo que restaba processar cuando la cola fue pausada.

### 22.5.1 Código:

```
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;
}
```

## 22.6 outputFunction

La invocación de este método indica que debe enviarse el valor del tope de la cola por el puerto de salida ya que finalizó el intervalo de preparación del dato.

### 22.6.1 Código:

```
Model &Queue::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, elements.front() ) ;
   return *this ;
}
```

## 22.7 internalFunction

La invocación de este método indica que se cumplió el tiempo de preparación (la función de salida fue invocada previamente), por lo tanto debe esperar un evento en el puerto *done* indicando la correcta recepción del dato.

### 22.7.1 Código:

```
Model &Queue::internalFunction( const InternalMessage & )
{
   this->passivate();
   return *this ;
}
```

# Manual del Usuario

## 23 Invocación del Simulador

Existen dos formas de invocar al simulador:

Standalone

Servidor de simulación (vía conexión por red).

### *23.1 Standalone*

Para configurar la ejecución del simulador, son posibles los siguientes parámetros:

**-h:** muestra la siguiente ayuda

```
simu [-ehlmt]
e: events file (default none)
h: show this help
l: message log file (default: /dev/null)
m: model file (default: model.ma)
t: stop time (default: ∞)
o: output file (default /dev/null)
```

**-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.

**-e:** Nombre de archivo del cual se cargarán los eventos externos. Si se omite este parámetro el simulador no utilizará ningún evento externo.

**-l:** Nombre de archivo en el cual se graban los mensajes que reciben y emiten los modelos a lo largo de la simulación. Si se omite este parámetro el simulador no generará ningun log de actividad. Si se desea obtener el log por el standard output debe especificarse el parámetro "-" ( -l- ).

**-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 el standard output debe especificarse el parámetro "-" ( -o- ).

**-t:** Hora de finalización de la simulación. Si se omite este parámetro el simulador solo se detendrá ante la falta de eventos (externos o internos). 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)

## 23.2 Servidor de simulación

La invocación del simulador sin especificar parámetros indica al simulador que debe cargarse en modo servidor de simulación. La comunicación con el servidor se realizará utilizando los servicios TCP/IP (en esta implementación).

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 .

Hora de Finalización.

# 24 Archivos de configuración

## *24.1 Modelos*

El archivo esta compuesto por grupos de definiciones de modelos acoplados (obligatoria para *top* los demas acoplados que componen la simulación) y configuración de modelos atómicos (opcional). Cada definición indica el nombre del modelo (entre []) y sus atributos. **El grupo del modelo [top] es obligatorio**.

### 24.1.1 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 nombre de modelo es necesario ya que es posible construir un modelo acoplado con mas 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 básicos (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 output. Este atributo es opcional ya que un modelo puede no tener puertos de salida.

**In**: enumera los nombres de los puertos de input. Este atributo es opcional ya que un modelo puede no tener puertos de entrada.

**Link**: describe el esquema de acoplamiento interno, externo de entrada y externo de salida. El formato esta dado por el par:

**port_origen[@modelo] port_destino [@modelo]**

El 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@Consumidorr 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
```

### 24.1.2 Atómicos

En esta definición se configuran los modelos atómicos participantes de la simulación. En caso de no figurar ninguna referencia se tomarán valores por defecto que halla programado el desarrollador de dicha clase.

La configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
.
.
.
nombre_variablen : valor_varn
```

Los nombre de las variables dependen del desarrollador de la clase que se desea configurar y deben estar documentados junto con los fuentes de la clase.

Cada instancia de un tipo de modelos atómico podra 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
.
.
.

[processor2]
distribution : poisson
mean : 50

[processor]
distribution : exponential
mean : 10

[queue]
preparation : 0:0:0:0
```

### 24.1.3 Celulares

Los modelos celulares son una variante de los modelos acoplados y por esto se utiliza la misma especificación agregándole ciertos parámetros inherentes a las caracteristicas del modelo.

Parámetros

*type* : [CELL | FLAT] (si no se especifica es un acoplado común)

*width* : INTEGER

*height* : INTEGER

*in*: Igual a un acoplado común.

*out*: Igual a un acoplado común.

*link*: Igual a un acoplado común pero para hacer referencia a una celda se debe usar el nombre del acoplado mas (x,y) sin dejar espacios.

*border*: [ WRAPPED | NOWRAPPED ] (por default NOWRAPPED)

*delay*: [ TRASPORT | INERTIAL ] (por default TRANSPORT)

*defaultDelayTime:* Demora por defector para los eventos externos (en milisegundos).

*neighbors* : nombreCelular( $x_1$, $y_1$ )... nombreCelular( $x_n$, $y_n$ ) ( Donde $-1 \leq x_i, y_i \leq 1$, esto determina quiénes pertenecen al vecindario – 3x3 – de cada celda).

*initialvalue*: [ 0 | 1 | ? ] (Representa el valor inicial para toda la matriz de celdas, los valores posibles son: false, true o indefinido).

*initialRowValue*: $fila_i$ $valor_1$..$valor_{width}$ ( $0 \leq fila_i \leq$ height y los valores se indican uno al lado del otro sin ningún caracter separador respetando los mismos valores posibles usados en *initialValue*).

*localTransition*: transitionName (inidica el nombre del grupo que describe el lenguaje a utilizar para la función de transición local para todas las celdas).

*zone*: transitionName { $rango_1$..$rango_n$ } (El nombre del lenguajes es asociado a cada celda incluída dentro de la especificación de rango. Cada $rango_i$ es algo de la forma (x,y) describiendo una celda o (x,y)..(z,w) describiendo una zona rectangular de celdas).

Lenguaje de especificación GADCelLa

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 el mismo lenguaje como así también que varias zonas dentro de un celular lo utilicen sin necesidad de redefinirlo.

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 delay { condición }*

Donde:

*resultado:* [ 0 | 1 | ? ]

*delay:* INTEGER (cantidad de milisegundos)

*condición:* la condición es una expresión lógica que responde a la suiguiente gramática expresada en BNF.

```
BoolExp    := IntRelExp | NOT BoolExp | BoolExp AND BoolExp |
              BoolExp OR BoolExp
IntRelExp  := IdRef | IntExp OpRel IntExp
```

```
IntExp     := IdRef | IntExp Oper IntExp
IdRef      := CellRef | '(' BoolExp ')' | Constant | Function
Constant   := Int | Bool
Function   := TRUECOUNT | FALSECOUNT | UNDEFCOUNT
CellRef    :=  '(' IntExp ',' IntExp ')'
OpRel      := = | != | > | < | >= | <=
Oper       := + | - | * | /
Int        := [Sign] Digit {Digit}
Bool       := 0 | 1 | t | f | ?
Sign       := [+] | -
Digit      := 0 | 1 | ... | 9
```

Debugging

Si se desea que el simulador verifique en tiempo de ejecución que existe una única regla que puede ser satisfecha con los valores del vecindario debe setearse la variables de entorno **DEBUG_CELL_FUNCTION** con valor **1**. Esta posibilidad de chequeo de integridad y consistencia del lenguaje está disponible solamente en modo standalone.

Sintaxis según el shell (para activarlo):

- ***csh / tcsh***: setenv **DEBUG_CELL_FUNCTION 1**
- ***ksh*** / ***bash*****:** export **DEBUG_CELL_FUNCTION=1**

Sintaxis según el shell (para desactivarlo):

- ***csh / tcsh***: setenv **DEBUG_CELL_FUNCTION 0**
- ***ksh*** / ***bash*****:** export **DEBUG_CELL_FUNCTION=0**

Ejemplos

La siguiente es la especificación del juego de la vida:

```
[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 = 5 }
```

```
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { t }
```

La siguiente es la especificación del trafico en una zona comercial, donde hay 3 puntos de entrada (celdas (0,0) (7,1) y (6,3) ) y 2 de salida (celdas (9,0) y (9,1)):

```
[top]
components : CellCoupled
out : out
in : in
link : in in@CellCoupled
link : out@cellcoupled out

[CellCoupled]
type : cell
width : 50
height : 10
delay : transport
border : wrapped
neighbors : CellCoupled(-1,0)
neighbors : CellCoupled(0,-1) CellCoupled(0,1)
neighbors : CellCoupled(1,0)
in : in
out : out
link : in in@CellCoupled(0,0)
link : in in@CellCoupled(7,1)
link : in in@CellCoupled(6,3)
link : out@cellcoupled(9,0) out
link : out@cellcoupled(9,1) out
initialvalue : 1
initialrowvalue : 0  10101010101000101110010101010101?????101000000000
initialrowvalue : 1  01000101011101??0101??001101010101000101110010101010
initialrowvalue : 2  10101010101000101110010101010101?????101000000000
initialrowvalue : 3  00101010101010101?000000000101010101001?????10101
initialrowvalue : 4  00101010??10100000000001010001011100101010101001??
initialrowvalue : 5  10101010101010101?000000000101010101001?????10101
initialrowvalue : 6  00101010101010101?000000000101010101001?????10101
initialrowvalue : 7  10101010101000101110010101010101?????101000000000
initialrowvalue : 8  10101010101000101110010101010101?????101000000000
initialrowvalue : 9  00101010??10100000000001010001011100101010101001??
localtransition : transito
zone : bache { (8,8)..(9,9) (1,2) }

[transito]
rule : 1 5 { (0,0) = 0 and (1, 0) = 1  }
rule : 1 5 { ( 0, 0 ) = 0 and (0, -1) = 1  }
rule : 1 5 { (0,0) = 1 and (0,1) = 1  }
rule : 1 5 { ( 0,0) = 1 and (1,0) = 1  }
rule : 0 10 { (0,0) = 1 and (-1, 0) = 0 }
rule : 0 10 { ( 0, 0 ) = 0 and (0,0) = 1 }
rule : 0 10 { (0,1) = 0 and (0,0) = 0 }
rule : 0 10 { ( 0,0) = 1 and (1,0) = 1 }
rule : 1 5 { t }

[bache]
rule : 0 1 { t }
```

## 24.2 Eventos

El archivo es una simple 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 evento.

Valor: Valor numérico asociado al evento.

Ejemplo:

```
00:00:10:00 in 1
00:00:15:00 done 1
00:00:30:00 in 1
```

## 24.3 Archivo de Salida

Tiene el siguiente formato: Hora Puerto Valor.

Ejemplo:

```
00:00:01:00 out 0.000
00:00:02:00 out 1.000
```

## 24.4 Log de Mensajes

Este archivo muestra el flujo de mensajes entre los modelos que integran la simulación.

Cada línea del archivo muestra el tipo de mensaje, la hora a la que se produce, quien lo emite y el destinatario. Esta información es común a todos los mensajes. En caso de ser un mensaje **X** o **Y** aparecerá además el puerto y el valor. Para el mensaje **D** se agrega la hora del próximo evento, o "**…**" en caso de que la hora sea ∞.

Los números que figuran junto al nombre del simulador son a solo efecto de información para el desarrollador.

Ejemplo:

```
Mensaje I / 00:00:00:00 / Root(00) para Acoplado(06)
Mensaje I / 00:00:00:00 / Acoplado(06) para Queue(04)
Mensaje I / 00:00:00:00 / Acoplado(06) para Transducer(01)
Mensaje I / 00:00:00:00 / Acoplado(06) para Generator(02)
Mensaje I / 00:00:00:00 / Acoplado(06) para Processor(03)
Mensaje D / 00:00:00:00 / Queue(04) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Transducer(01) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje D / 00:00:00:00 / Processor(03) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Acoplado(06) / 00:00:10:00 para Root(00)
Mensaje * / 00:00:10:00 / Root(00) para Acoplado(06)
Mensaje * / 00:00:10:00 / Acoplado(06) para Generator(02)
Mensaje Y / 00:00:10:00 / Generator(02) / out / 3.000 para Acoplado(06)
Mensaje D / 00:00:10:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje X / 00:00:10:00 / Acoplado(06) / in / 3.000 para Queue(04)
Mensaje D / 00:00:10:00 / Transducer(01) / ... para Acoplado(06)
```

.

# 25 Simulation Tool-Kit

## *25.1 Drawlog*

Esta herramienta tiene por objeto mostrar en forma gráfica el analisis de la actividad del simulador en cada instante de tiempo para los modelos celulares. Toma como entrada los datos del log generado por el simulador y transforma los distintos tipos de mensajes para mostrar el estado al terminar cada ciclo.

Son posibles los siguientes parámetros:

**-h:** muestra la siguiente ayuda

```
drawlog [-hmtcl]
h: show this help
m: model file
t: start time (default: 0:0:0:0)
c: cell coupled name
l: message log file (default: stdandar input)
```

**-m:** Nombre de archivo del cual se cargará el modelo a analizar.

**-t:** Hora a partir de la cual se desean graicar los datos.

**-c:** Nombre del modelo celular que se desea graficar.

**-l:** Nombre de archivo del cual se obtien el log de actividad del simulador. Si se omite este parámetro se tomarán los datos del standard input.

Ejemplo:

```
drawlog –mlife.ma –clife –llife.log

o

simu –mlife.ma –l- | drawlog –mlife.ma –clife
```

Modelos Celulares Chatos (flat):

Los modelos celulares achatados no envían mensajes hacia las celdas y por lo tanto el drawlog no es de utilidad en este caso ya que el log no contendrá los mensajes para obtener los datos. Si se desea ver el estado de el autómata celular debes usarse la variable de entorno **FLATDEBUG** con valor 1. Esta posibilidad de visualización gráfica está disponible solamente en modo standalone.

Sintaxis según el shell (para activarlo):

- ***csh / tcsh***: setenv **FLATDEBUG 1**
- ***ksh*** / ***bash*****:** export **FLATDEBUG =1**

Sintaxis según el shell (para desactivarlo):

- ***csh / tcsh***: setenv **FLATDEBUG 0**
- ***ksh*** / ***bash*****:** export **FLATDEBUG =0**

# Manual del Desarrollador

## 26 Introducción

Este manual describe los procedimientos que deben realizarse para incorporar nuevos modelos atómicos a la entorno de simulación. Esto es necesario ya que para los modelos atómicos debe codificarse en C++ el comportamiento deseado. Para el resto de los modelos simulables (acoplados y celulares) por la herramienta es suficiente con la especificación a través de un archivo de inicialización; en este archivo tambien figura la definición de la vinculación de los modelos y de los componentes que lo forman. Esta información se encuentra disponible en el manual del usuario.

## 27 Ambiente de desarrollo

El lenguaje elegido para implementar el diseño del simulador fue C++.

Los fuentes del simulador se dividen en dos partes:

Librería de simulación.

Fuente principal y modelos atómicos.

Las rutinas de la librería llevan a cabo por completo todos los pasos de la simulación y se complementan con el fuente principal y los archivos de configuración para cargar el modelo a simular y configurar aspectos particulares de la plataforma en la cual el simulador correrá.

Todos los fuentes fueron compilados con el compilador gcc versión 2.8.x (http://www.sunsite.unc.edu/pub/gnu) del proyecto GNU Free Software Foundation Inc. (http://www.fsf.org) y testeados en las siguientes plataformas:

Linux Slackware v3

SunOs v5

Dos bajo Win95 (solo la versión stand-alone / sin soporte de red)

Los términos de distribución y copia son los mismos que aplica GNU para todas sus herramientas. Para mas información referirse a www.gnu.org.

El código usado respeta el standard ANSI C++ (http://www.ansi.org), e incluye las características recientemente incorporadas al lenguaje como **template function, C++ style cast y STL (Standard Template Library)**. La portabilidad del compilador permite realizar la compilación sin ningún cambio para la versión standalone en cualquier plataforma soportada. Los servicios de red fueron implementados vía una clase que abstrae el canal de comunicación entre el servidor de simulación y los clientes. Esta abstracción define la interfaz que utiliza el simulador para obtener los datos y luego retornar los resultados. La implementación realizada, hasta el momento en que se redactó este informe, fue hecha sobre BSD Sockets, standard para la mayoría de los ambientes UNIX de las universidades del mundo. Otras implementaciones son fácilmente realizables y transparentes por completo para el funcionamiento del simulador haciéndolo de esta forma independiente de la plataforma y el entorno en el que está corriendo.

## 27.1 Incorporación de nuevos modelos

Los pasos para generar un nuevo modelo atómico son los siguientes:

Diseñar una clase que herede de la clase *Atomic* y **sobrecargarndo obligatoriamente** los métodos:

*initFunction*: antes de invocar al método sigma vale infinito y el estado es pasivo.

*externalFunction*: invocado cuando arriba un evento externo en alguno de los ports del modelo.

*internalFunction*: antes de invocar al método sigma vale cero ya que se ha cumplido el intervalo para la transición interna.

*outputFunction*: antes de invocar al método sigma vale cero ya que se ha cumplido el intervalo para la transición interna.

*className*: nombre de la clase.

Incorporar en el fuente **register.cpp**, agregándole al método *Simulator::registerNewAtomics()* el nuevo tipo de modelo atómico, por ejemplo para la clase Queue:

```
SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>(), "Queue" ) ;
```

## 27.2 Interacción con el Simulador

En cada uno de los métodos que se nombró anteriormente se deben invocar ciertas primitivas para cumplir con las normas impuestas por el simulador. Estas primitivas tienen por función interactuar con el simulador para realizar tareas comunes a todos los modelos atómicos.

La primitivas disponibles son:

**holdIn(estado, tiempo):** indica al simulador que el modelo debe mantenerse en el estado por un tiempo fijo, y que después de transcurrido ese tiempo deberá producirse una cambio de estado. El estado podrá tener valor: active / passive.

**passivate():** indica al simulador que el modelo entra en modo pasivo y que únicamente deberá ser invocado por la llegada de un evento externo.

**sendOutput(hora, port, valor):** envía un mensaje de salida.

**nextChange():** método de consulta para obtener el tiempo restante para su próximo cambio de estado (sigma)

**lastChange():** método de consulta para obtener la hora de su último cambio de estado.

**state():** método de consulta del estado en que se encuentra el modelo.

**getParameter( nombreModelo, nombreParámetro ):** método de acceso a los parámetros de configuración de la clase. Este método pertenece a la clase Simulator.

Al ser *Model* una clase base abstracta define la interfaz para la recepción de mensajes, la clases *Atomic* y *Coupled* son la únicas encargada de recibir y enviar mensajes. Es **responsabilidad de las**

**clases derivadas** de *Atomic* re-implementar las funciones de inicializacion, transición interna, transición externa y salida.

Las clases derivadas de *Atomic* no deben enviar ningún tipo de mensaje, salvo los valores de salida que se informan mediante el método *sendOutput*. La clase *Atomic* es la responsable de generar los mensajes Y y D al padre para cualquier instancia derivada, utilizando los valores de *sigma* y *state*.

### 27.2.1 Ejemplo: Implementación de una cola (class Queue)

Una cola es un dispositivo de almacenamiento con política FIFO. Posee tres ports de entrada y un port de salida.

En el port **in** arrivan los datos de entradas. Por cada valor que se envía por el port **out** se esperará la confirmación de su recepción en el port **done.**

Si el cliente de la cola se encuentra saturado, podrá manejar el control de flujo a través del port **stop-send**.

El primer paso para implementar un nuevo modelo atómico es definir una clase derivada de la clase *Atomic* redefiniendo lost métodos para el manejo de transiciones. Estos métodos no son públicos ya que solamente la clase base *Atomic* los puede invocar.

En el ejemplo vemos la definición de variables para hacer referencia a los ports de la cola y el tiempo que demora en preparar el dato antes de enviarlo por el port de salida. Además se encuentra la definición de una lista de valores para albergar los valores ingresados y el tiempo restante ante una interrupción por control de flujo.

```
class Queue : public Atomic {
public:
   Queue(); // Default constructor
   virtual string className() const ;

protected:
   Model &initFunction();
   Model &externalFunction( const ExternalMessage & );
   Model &internalFunction( const InternalMessage & );
   Model &outputFunction( const InternalMessage & );

private:
   const Port &in;
   const Port &stop;
   const Port &done;
   Port &out;
   Time preparationTime;

   typedef list<Value> ElementList ;
   ElementList elements ;

   Time timeLeft;

}; // class Queue
```

## 27.3 Constructor

El constructor crea los tres ports de entrada, el port de salida e inicializa la variable *preparationTime*. El parámetro preparation puede estar especificado en el archivo de configuración del modelo, de ser así se modifica el tiempo de preparación.

### 27.3.1 Código:

```
Queue::Queue()
   : preparationTime( 0, 0, 10, 0 )
   , in( this->addInputPort( "in" ) )
   , stop( this->addInputPort( "stop" ) )
   , done( this->addInputPort( "done" ) )
   , out( this->addOutputPort( "out" ) )
{
   this->description( "Queue" ) ;

   string time( Simulator::Instance().getParameter(
                          this->description(), "preparation" ) ) ;

   if( time != "" )
      preparationTime = time ;
}
```

## 27.4 initFunction

La función de inicialización debe borrar la lista que guarda los elementos encolados. El próximo cambio de estado se producirá ante la llegada de un evento externo por eso sigma permanece constante. Si se desea planificar el proximo cambio de estado debe modificarse usando el método *holdIn*.

### 27.4.1 Código:

```
Model &Queue::initFunction()
{
   elements.remove( elements.begin(), elements.end() ) ;
   return *this;
}
```

## 27.5 externalFunction

La clase Queue tiene tres puertos de entrada por los cuales puede recibir eventos externos.

Si el evento arriva en el port *in* significa que es un dato nuevo de entrada, por lo tanto debe encolarse. Si es el único en la cola se debe preparar el dato para ser enviado.

Si el evento arriva en el port *done* significa que el último elemento enviado fue recibido correctamente, por lo tanto debe ser elimindado de la cola. Si hay más elementos a enviar se debe preparar el próximo dato para ser enviado.

Por último el evento puede arrivar en el port *stop* indicando que se desea pausar o recomenzar el trabajo de la cola. Si la cola se encuetra en estado *active* y el valor del mensaje es distinto de cero esto indica que se desea hacer una pausa, por eso se calcula cuanto tiempo faltaba procesar para el próximo cambio de estado (fin de la preparación del dato a enviar) y luego la cola cambia su estado a *passive* llamando al método *passivate*. Si la cola se encuentra en estado *passive* y el valor del mensaje es cero la cola vuelve a estar activa planificando el próximo cambio de estado en base al tiempo que restaba processar cuando la cola fue pausada.

### 27.5.1 Código:

```
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;
}
```

## 27.6 outputFunction

La invocación de este método indica que debe enviarse el valor del tope de la cola por el puerto de salida ya que finalizó el intervalo de preparación del dato.

### 27.6.1 Código:

```
Model &Queue::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, elements.front() ) ;
   return *this ;
}
```

## 27.7 internalFunction

La invocación de este método indica que se cumplió el tiempo de preparación (la función de salida fue invocada previamente), por lo tanto debe esperar un evento en el puerto *done* indicando la correcta recepción del dato.

### 27.7.1 Código:

```
Model &Queue::internalFunction( const InternalMessage & )
{
   this->passivate();
   return *this ;
}
```

# Manual del Usuario

## 28 Invocación del Simulador

Existen dos formas de invocar al simulador:

Standalone

Servidor de simulación (vía conexión por red).

### *28.1 Standalone*

Para configurar la ejecución del simulador, son posibles los siguientes parámetros:

**-h:** muestra la siguiente ayuda

```
simu [-ehlmt]
e: events file (default none)
h: show this help
l: message log file (default: /dev/null)
m: model file (default: model.ma)
t: stop time (default: ∞)
o: output file (default /dev/null)
```

**-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.

**-e:** Nombre de archivo del cual se cargarán los eventos externos. Si se omite este parámetro el simulador no utilizará ningún evento externo.

**-l:** Nombre de archivo en el cual se graban los mensajes que reciben y emiten los modelos a lo largo de la simulación. Si se omite este parámetro el simulador no generará ningun log de actividad. Si se desea obtener el log por el standard output debe especificarse el parámetro "-" ( -l- ).

**-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 el standard output debe especificarse el parámetro "-" ( -o- ).

**-t:** Hora de finalización de la simulación. Si se omite este parámetro el simulador solo se detendrá ante la falta de eventos (externos o internos). 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)

## 28.2 Servidor de simulación

La invocación del simulador sin especificar parámetros indica al simulador que debe cargarse en modo servidor de simulación. La comunicación con el servidor se realizará utilizando los servicios TCP/IP (en esta implementación).

El simulador esperará en un port 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 .

Hora de Finalización.

# 29 Archivos de configuración

## *29.1 Modelos*

El archivo esta compuesto por grupos de definiciones de modelos acoplados (obligatoria para *top* los demas acoplados que componen la simulación) y configuración de modelos atómicos (opcional). Cada definición indica el nombre del modelo (entre []) y sus atributos. **El grupo del modelo [top] es obligatorio**.

### 29.1.1 Acoplados

Existen cuatro posibles propiedades a configurar: componentes (**components**), ports de salida (**out**), ports 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 nombre de modelo es necesario ya que es posible construir un modelo acoplado con mas 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 básicos (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 ports de output. Este atributo es opcional ya que un modelo puede no tener ports de salida.

**In**: enumera los nombres de los ports de input. Este atributo es opcional ya que un modelo puede no tener ports de entrada.

**Link**: describe el esquema de acoplamiento interno, externo de entrada y externo de salida. El formato esta dado por el par:

**port_origen[@modelo] port_destino [@modelo]**

El modelo es opcional ya que si no se indica se toma como un port 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@Consumidorr 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
```

### 29.1.2 Atómicos

En esta definición se configuran los modelos atómicos participantes de la simulación. En caso de no figurar ninguna referencia se tomarán valores por defecto que halla programado el desarrollador de dicha clase.

La configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
.
.
.
nombre_variablen : valor_varn
```

Los nombre de las variables dependen del desarrollador de la clase que se desea configurar y deben estar documentados junto con los fuentes de la clase.

Cada instancia de un tipo de modelos atómico podra 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
.
.
.

[processor2]
distribution : poisson
mean : 50

[processor]
distribution : exponential
mean : 10

[queue]
preparation : 0:0:0:0
```

### 29.1.3 Celulares

Los modelos celulares son una variante de los modelos acoplados y por esto se utiliza la misma especificación agregándole ciertos parametros inherentes a las caracteristicas del modelo.

Parámetros

*type* : [CELL | FLAT] (si no se especifica es un acoplado común)

*width* : INTEGER

*height* : INTEGER

*in*: Igual a un acoplado común.

*out*: Igual a un acoplado común.

*link*: Igual a un acoplado común pero para hacer referencia a una celda se debe usar el nombre del acoplado mas (x,y) sin dejar espacios.

*border*: [ WRAPPED | NOWRAPPED ] (por default NOWRAPPED)

*delay*: [ TRASPORT | INERTIAL ] (por default TRANSPORT)

*defaultDelayTime:* Demora por defector para los eventos externos (en milisegundos).

*neighbors* : nombreCelular( $x_1$, $y_1$ )... nombreCelular( $x_n$, $y_n$ ) ( Donde $-1 \leq x_i, y_i \leq 1$, esto determina quiénes pertenecen al vecindario – 3x3 – de cada celda).

*initialvalue*: [ 0 | 1 | ? ] (Representa el valor inicial para toda la matriz de celdas, los valores posibles son: false, true o indefinido).

*initialRowValue*: $fila_i$ $valor_1$..$valor_{width}$ ( $0 \leq fila_i \leq$ height y los valores se indican uno al lado del otro sin ningún caracter separador respetando los mismos valores posibles usados en *initialValue*).

*localTransition*: transitionName (inidica el nombre del grupo que describe el lenguaje a utilizar para la función de transición local para todas las celdas).

*zone*: transitionName { $rango_1$..$rango_n$ } (El nombre del lenguajes es asociado a cada celda incluída dentro de la especificación de rango. Cada $rango_i$ es algo de la forma (x,y) describiendo una celda o (x,y)..(z,w) describiendo una zona rectangular de celdas).

Lenguaje de especificación

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 el mismo lenguaje como asi también que varias zonas dentro de un celular lo utilicen sin necesidad de redefinirlo.

El lenguajes de define como un nuevo grupo dentro de la especificación, donde cada componente del grupo es una regla con la siguiente sintaxis:

*rule : resultado delay { condición }*

Donde:

*resultado:* [ 0 | 1 | ? ]

*delay:* INTEGER (cantidad de milisegundos)

*condición:* la condición es una expresión lógica que responde a la suiguiente gramática expresada en BNF.

```
BoolExp :=  IntRelExp | NOT BoolExp | BoolExp AND BoolExp | BoolExp OR BoolExp
IntRelExp := IdRef | IntExp OpRel IntExp
IntExp :=  IdRef | IntExp Oper IntExp
```

IdRef := CellRef | '(' BoolExp ')' | Constant | Function
Constant := Int | Bool
Function := **TRUECOUNT** | **FALSECOUNT** | **UNDEFCOUNT**
CellRef := '(' IntExp ',' IntExp ')'
OpRel := **=** | **!=** | **>** | **<** | **>=** | **<=**
Oper := **+** | **-** | ***** | **/**
Int := [Sign] Digit {Digit}
Bool := **0** | **1** | **t** | **f** | **?**
Sign := **[+]** | **-**
Digit ::= **0** | **1** | **...** | **9**

Debugging

Si se desea que el simulador verifique en tiempo de ejecución que existe una única regla que puede ser satisfecha con los valores del vecindario debe setearse la variables de entorno **DEBUG_CELL_FUNCTION** con valor **1**. Esta posibilidad de chequeo de integridad y consistencia del lenguaje está disponible solamente en modo standalone.

Sintaxis segun el shell (para activarlo):

- ***csh / tcsh***: setenv **DEBUG_CELL_FUNCTION 1**
- ***ksh*** / ***bash*****:** export **DEBUG_CELL_FUNCTION=1**

Sintaxis segun el shell (para desactivarlo):

- ***csh / tcsh***: setenv **DEBUG_CELL_FUNCTION 0**
- ***ksh*** / ***bash*****:** export **DEBUG_CELL_FUNCTION=0**

Ejemplos

La siguiente es la especificación del juego de la vida:

```
[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 = 5 }
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { t }
```

La siguiente es la especificación del trafico en una zona comercial, donde hay 3 puntos de entrada (celdas (0,0) (7,1) y (6,3) ) y 2 de salida (celdas (9,0) y (9,1)):

```
[top]
components : CellCoupled
out : out
in : in
link : in in@CellCoupled
link : out@cellcoupled out

[CellCoupled]
type : cell
width : 50
height : 10
delay : transport
border : wrapped
neighbors : CellCoupled(-1,0)
neighbors : CellCoupled(0,-1) CellCoupled(0,1)
neighbors : CellCoupled(1,0)
in : in
out : out
link : in in@CellCoupled(0,0)
link : in in@CellCoupled(7,1)
link : in in@CellCoupled(6,3)
link : out@cellcoupled(9,0) out
link : out@cellcoupled(9,1) out
initialvalue : 1
initialrowvalue : 0  10101010101000101110010101010101?????101000000000
initialrowvalue : 1  01000101011101??0101??001101010101000101110010101010
initialrowvalue : 2  10101010101000101110010101010101?????101000000000
initialrowvalue : 3  00101010101010101?000000000101010101001?????10101
initialrowvalue : 4  00101010??10100000000001010001011100101010101001??
initialrowvalue : 5  10101010101010101?000000000101010101001?????10101
initialrowvalue : 6  00101010101010101?000000000101010101001?????10101
initialrowvalue : 7  10101010101000101110010101010101?????101000000000
initialrowvalue : 8  10101010101000101110010101010101?????101000000000
initialrowvalue : 9  00101010??10100000000001010001011100101010101001??
localtransition : transito
zone : bache { (8,8)..(9,9) (1,2) }

[transito]
rule : 1 5 { (0,0) = 0 and (1, 0) = 1  }
rule : 1 5 { ( 0, 0 ) = 0 and (0, -1) = 1  }
rule : 1 5 { (0,0) = 1 and (0,1) = 1  }
rule : 1 5 { ( 0,0) = 1 and (1,0) = 1  }
rule : 0 10 { (0,0) = 1 and (-1, 0) = 0 }
rule : 0 10 { ( 0, 0 ) = 0 and (0,0) = 1 }
rule : 0 10 { (0,1) = 0 and (0,0) = 0 }
rule : 0 10 { ( 0,0) = 1 and (1,0) = 1 }
rule : 1 5 { t }

[bache]
rule : 0 1 { t }
```

## *29.2 Eventos*

El archivo es una simple secuencia de líneas, donde cada línea describe un evento con el siguiente formato:

**HH:MM:SS:MS PORT VALOR**

Donde:

HH:MM:SS:MS indica la hora en que ocurrirá el evento.

Port: indica el nombre del port por el cual se lanzará evento.

Valor: Valor numérico asociado al evento.

Ejemplo:

```
00:00:10:00 in 1
00:00:15:00 done 1
00:00:30:00 in 1
```

## 29.3 Archivo de Salida

Tiene el siguiente formato: Hora Port Valor.

Ejemplo:

```
00:00:01:00 out 0.000
00:00:02:00 out 1.000
```

## 29.4 Log de Mensajes

Este archivo muestra el flujo de mensajes entre los modelos que integran la simulación.

Cada línea del archivo muestra el tipo de mensaje, la hora a la que se produce, quien lo emite y el destinatario. Esta información es común a todos los mensajes. En caso de ser un mensaje **X** o **Y** aparecerá además el port y el valor. Para el mensaje **D** se agrega la hora del próximo evento, o "**…**" en caso de que la hora sea ∞.

Los números que figuran junto al nombre del simulador son a solo efecto de información para el desarrollador.

Ejemplo:

```
Mensaje I / 00:00:00:00 / Root(00) para Acoplado(06)
Mensaje I / 00:00:00:00 / Acoplado(06) para Queue(04)
Mensaje I / 00:00:00:00 / Acoplado(06) para Transducer(01)
Mensaje I / 00:00:00:00 / Acoplado(06) para Generator(02)
Mensaje I / 00:00:00:00 / Acoplado(06) para Processor(03)
Mensaje D / 00:00:00:00 / Queue(04) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Transducer(01) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje D / 00:00:00:00 / Processor(03) / ... para Acoplado(06)
Mensaje D / 00:00:00:00 / Acoplado(06) / 00:00:10:00 para Root(00)
Mensaje * / 00:00:10:00 / Root(00) para Acoplado(06)
Mensaje * / 00:00:10:00 / Acoplado(06) para Generator(02)
Mensaje Y / 00:00:10:00 / Generator(02) / out / 3.000 para Acoplado(06)
Mensaje D / 00:00:10:00 / Generator(02) / 00:00:10:00 para Acoplado(06)
Mensaje X / 00:00:10:00 / Acoplado(06) / in / 3.000 para Queue(04)
Mensaje D / 00:00:10:00 / Transducer(01) / ... para Acoplado(06)
  .
```

# 30 Simulation Tool-Kit

## *30.1 Drawlog*

Esta herramienta tiene por objeto mostrar en forma gráfica el analisis de la actividad del simulador en cada instante de tiempo para los modelos celulares. Toma como entrada los datos del log generado por el simulador y transforma los distintos tipos de mensajes para mostrar el estado al terminar cada ciclo.

Son posibles los siguientes parámetros:

**-h:** muestra la siguiente ayuda

```
drawlog [-hmtcl]
h: show this help
m: model file
t: start time (default: 0:0:0:0)
c: cell coupled name
l: message log file (default: stdandar input)
```

**-m:** Nombre de archivo del cual se cargará el modelo a analizar.

**-t:** Hora a partir de la cual se desean graicar los datos.

**-c:** Nombre del modelo celular que se desea graficar.

**-l:** Nombre de archivo del cual se obtien el log de actividad del simulador. Si se omite este parámetro se tomarán los datos del standard input.

Ejemplo:

```
drawlog –mlife.ma –clife –llife.log

o

simu –mlife.ma –l- | drawlog –mlife.ma –clife
```

Modelos Celulares Chatos (flat):

Los modelos celulares achatados no envían mensajes hacia las celdas y por lo tanto el drawlog no es de utilidad en este caso ya que el log no contendrá los mensajes para obtener los datos. Si se desea ver el estado de el autómata celular debes usarse la variable de entorno **FLATDEBUG** con valor 1. Esta posibilidad de visualización gráfica está disponible solamente en modo standalone.

Sintaxis segun el shell (para activarlo):

- ***csh / tcsh***: setenv **FLATDEBUG 1**
- ***ksh*** / ***bash*****:** export **FLATDEBUG =1**

Sintaxis segun el shell (para desactivarlo):

- ***csh / tcsh***: setenv **FLATDEBUG 0**
- ***ksh*** / ***bash*****:** export **FLATDEBUG =0**