# ANEXO 1

En el siguiente documento se presenta el anexo 1 de la monografía “Estudio de la herramienta Ptolemy II para el Modelamiento y Diseño de Sistemas Telemáticos y Generación de una Metodología para la Construcción de Actores”, el cual describe los dominios que se implementan en la herramienta Ptolemy II versión 1.0.1.

## A.1.1. DOMINIO DE

### A.1.1.1. INTRODUCCION

El dominio de evento-discreto (DE) soporta modelos de tiempo orientados de sistemas tales como sistemas de hacer colas, redes de comunicación, y hardware digital. En este dominio, los actores se comunican por el envío de eventos, donde un evento es valor de dato (una señal) y un sello de tiempo. Un programa DE asegura que los eventos sean procesados cronológicamente de acuerdo a es te selle de tiempo por la activación de dichos actores cuyos eventos de entrada disponibles son los más viejos (teniendo temprano el sello de tiempo de todos los eventos pendientes).

Una fortaleza clave en nuestra implementación es que eventos simultáneos (aquellos con sellos de tiempo idénticos) son manejados sistemática y determinísticamente. Una segunda fortaleza clave es que la cola global de eventos usa una eficiente estructura que minimiza el overhead asociado con el mantener una lista ordenada con un gran número de eventos.

#### A.1.1.1.1. TIEMPO MODELO

En el modelo DE de computación, el tiempo es global, en el sentido que todos los actores comparten el mismo tiempo global. El tiempo actual del modelo es frecuentemente llamado el tiempo modelo o tiempo de simulación para evitar confusión con el tiempo real actual.

Como en la mayoría de dominios de Ptolemy II, los actores son comunicados por el envío de señales a través de puertos. Los puertos pueden ser de entrada, de salida o ambos. Las señales son enviadas por un puerto de salida y recibida por todos los puertos de entrada conectados al puerto de salida mediante relaciones. Cuando una señal es enviada desde un puerto de salida, es empaquetada como un evento y acumulada en la cola global de eventos. Por defecto, el sello de tiempo de una salida es el tiempo modelo, aunque los actores DE especializados pueden producir eventos con sellos de tiempo futuros.

Los actores pueden requerir ser activados en el mismo tiempo en futuro llamando el método fireAt() del director. Esto ubica un evento puro (uno con sello de tiempo pero sin datos) sobre la cola de eventos. Un evento puro puede ser sin embargo de ambiente reloj de alarma para ser despertado en el futuro. Fuentes (actores sin entradas) son más capaces de ser activados a pesar de no tener entradas iniciar una activación. Además, los actores que introducen delay (las salidas tienen sello de tiempo más grande que las entradas) pueden usar este mecanismo para programar una activación en el futuro para producir una salida.

En la cola global de eventos, los eventos son ordenados con base en sus sellos de tiempo y sus puertos de destinación (o actores, en el caso de eventos puros). Un evento es removido de la cola global de eventos cuando el tiempo modelo alcanza su sello de tiempo, y si tiene señal de datos, entonces la señal es puesta dentro del puerto de entrada destino.

En cualquier punto de la ejecución de un modelo, los eventos ordenados en la cola global de eventos tienen sellos de tiempo mayores o iguales que el tiempo modelo. El director DE es responsable para avanzar (es decir, incrementando) el tiempo modelo cuando todos los eventos con sellos de tiempo igual al tiempo modelo actual han sido procesados (es decir, la cola global de eventos solamente contiene eventos con sellos de tiempo estrictamente mayores que el tiempo actual). El tiempo actual es procesado para el más pequeño sello de tiempo de

todos los eventos en la cola global. Este avance marca el comienzo de una iteración.

### A.1.1.1.2. EVENTOS SIMULTANEOS

Un importante aspecto de un dominio DE es la priorización de los eventos simultáneos. Esto da al dominio un comportamiento semejante del flujo de datos para eventos con sellos de tiempo idénticos. Esto es completado por la asignación de rangos para los actores. Los rangos son dibujados desde el conjunto de integradores no negativos. Ellos son asignados únicamente; es decir, dos actores distintos no son asignados en el mismo rango. Eventos simultáneos con la más alta prioridad son aquellos destinados para actores con los más bajos rangos. Los rangos son determinados por una clasificación topológica de una gráfica acíclica dirigida (DAG) de los actores.

La DAG de los actores sigue la topología de la gráfica, excepto cuando son declarados delays. Considerar la topología simple mostrada en la figura 1.1. Se asume que el actor Y es un actor cero-delay, representando que sus eventos de salida tienen el mismo sello de tiempo como los eventos de entrada (esto es sugerido por la flecha punteada). Se supone que el actor X produce un evento con sello de tiempo $\tau$ . El evento está disponible en los puertos B y D, así el programa podría escoger para activar los actores Y o Z. Lo cual debería activarlos? La intuición nos dice que debería activarse upstream del primero, Y, porque esta activación puede producir otro evento con el sello de tiempo $\tau$ en el puerto D (lo cual es presumiblemente un multipuerto). Parece lógico que si un actor Z va a recibir un evento en cada canal de entrada con el mismo sello de tiempo, entonces debería observar dichos eventos en la misma activación. Si hay varios eventos simultáneos en B y D, entonces uno en B debería tener más alta prioridad.

Una vez la DAG es construida, es ordenada topológicamente. Esto significa simplemente que un ordenamiento de los actores es asignado tal que un actor upstream en la DAG es ordenado más temprano que un actor downstream. Este ordenamiento no es único, significa que las prioridades asignadas para los actores osn algo arbitrarias. Con tal de que los actores estén comunicados solamente por vía eventos, sin embargo, luego estas elecciones no tendrán impacto sobre el resultado final de la ejecución del modelo. Se dice que la ejecución es determinística.

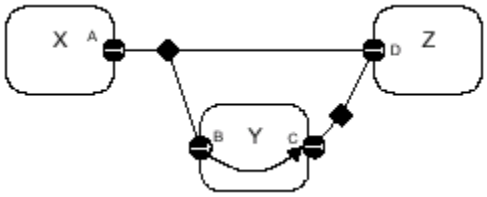


**Figura 1.1.**

Hay situaciones donde construir una DAG siguiendo la topología no es posible. Se considera la topología mostrada en la figura 1.2. Es evidente en la figura que la topología no es acíclica. De hecho, la figura 1.2 describe una curva dirigida cero-delay donde el orden topológico no puede ser realizado. El director rechazará correr el modelo, y terminará con un mensaje de error. Esto es llamado una curva cero-delay.

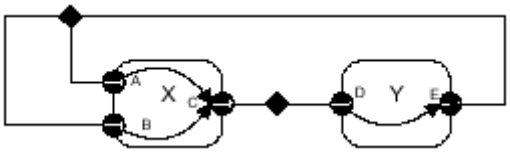


**Figura 1.2.**

El actor delay en DE es un actor de dominio específico que defiende una relación entre su entrada y su salida. Además, si se inserta una actor en la curva, como muestra la figura 1.3, entonces la construcción de la DAG es posible una vez más. El actor delay rompe las precedencias.

Note en particular que el actor delay rompe las presencias igual si su parámetro delay se pone en ceros. Además, el dominio DE es perfectamente capaz de modelar curvas cero-delay, pero el constructor del modelo tiene que especificar el orden en el cual los eventos deberían ser procesados colocando el actor delay con un valor de cero para este parámetro.

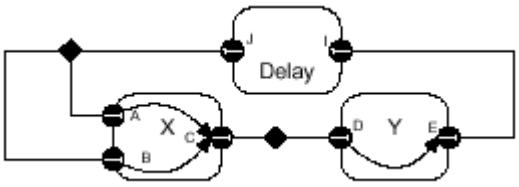


**Figura 1.3.**

### A.1.1.1.3. ITERACION

En cada iteración, después de avanzar el tiempo actual, el director elige algunos eventos en la cola global de eventos basado en sus sellos de tiempo y actores destinos. El director DE selecciona eventos de acuerdo con las siguientes reglas:

- Encontrar un conjunto no vacío E de eventos asociado con el sello de tiempo más pequeño.
- Si el conjunto de eventos contiene más de un evento, encontrar uno con la más alta prioridad, $e_{max}$. Eventos simultáneos son priorizados cuidadosamente por promedios de orden topológico de los actores (para asegurar un comportamiento determinístico).
- Escoger todos los eventos en E que tienen el mismo actor destino como $e_{max}$. Notar que el actor destino de un evento es el actor que contiene el puerto de salida destino de ese evento, o si es un evento puro, el actor a ser activado.

Los eventos escogidos son entonces removidos de la cola global de eventos y sus señales datos son insertadas dentro de los puertos de entrada apropiados del actor destino. Entonces, el director itera el actor destino, es decir, invoca prefire(), fire(), y postfire().

Una activación produce eventos adicionales en el tiempo modelo actual ( el actor reacciona instantáneamente, o tiene delay cero). También pueden ser otros eventos con sello de tiempo igual al tiempo modelo actual todavía pendientes en la cola de eventos. El director DE repite el procedimiento anterior hasta que no hayan más eventos con sello de tiempo igual al tiempo actual. Esto concluye una iteración del modelo.

#### A.1.1.1.4. HACIENDO UN MODELO INICIADO

Antes una de las iteraciones descritas anteriormente puede ser corrida, así tiene para ser eventos iniciales en la cola global de eventos. Los actores pueden producir eventos puros iniciales en su método initialize() porque la resolución tipo no ha sido completada, así los tipos de los puertos no son conjuntos. Además, para recibir un modelo iniciado, por lo menos un actor debería ser usado y producir dichos eventos puros. Todas las fuentes cronometradas de dominio polimórfico descritas en el capítulo Librería de actores producen tales eventos. Se puede definir el tiempo de iniciación para ser el sello de tiempo más pequeño de los eventos iniciales.

#### A.1.1.1.5. DETENER UNA EJECUCION

La ejecución se detiene cuando una de las siguientes condiciones llega a ser verdadera:

- El tiempo actual alcanza el tiempo de parada, el conjunto llama el método setStopTime() del director DE.
- La cola global de eventos llega a estar vacía.

Los eventos en el tiempo de parada son procesados antes de parar la ejecución del modelo. La ejecución termina llamando el método wrapup() de todos los actores.

### A.1.1.2. ARQUITECTURA DEL SOFTWARE DEL DOMINIO DE

El diagrama de la estructura estática UML para el paquete kernel es mostrado en la figura 1.4. Para los constructores de modelos, las clases importantes son DEDirector, DEActor y DEIOPort. El corazón de DEDirector es la cola global de eventos que organiza eventos de acuerdo con su sello de tiempo y prioridades.

El DEDirector usa una eficiente implementación de la cola global de eventos, un calendario de la estructura de datos en cola [11]. La complejidad del tiempo para esta implementación particular es O(1) en las operaciones de enqueue y dequeue. Esto representa que la complejidad del tiempo para operaciones de enqueue y dequeue es independiente del número de eventos pendientes en la cola global de eventos. Por extensibilidad, implementaciones diferentes de la cola global de eventos pueden ser realizadas por la implementación de la interfase DEEventQueue y especificando la cola de eventos usando el constructor apropiado para DEDirector.

La clase DEActor provee métodos convenientes para acceder al tiempo, el tiempo es una parte esencial de un dominio cronometrado semejante a DE. No obstante, los actores en un modelo DE no son requeridos para ser derivados desde la clase DEActor. Simplemente derivando del TypedAtomicActor da la misma capacidad,

pero sin la conveniencia. En el último caso, el tiempo es accesible a través del director.

La clase DEIOPort es usada por actores que son especializados en el dominio DE. Soportan anotaciones que informan el programa sobre el despliegue a través del actor. También provee dos métodos adicionales, versiones sobrecargadas de broadcast() y send(). Las versiones sobrecargadas tienen un segundo argumento para el despliegue del tiempo, permitiendo a los actores enviar datos de salida con un despliegue de tiempo (relativo al tiempo actual).

Los actores de dominio polimórfico, tales como los descritos en el capítulo Librería de actores, tienen como puertos instancias de TypedIOPort, no DEIOPort, y por lo tanto no puede producir eventos en el futuro directamente por envíos a través de los puertos de salida. Notar que las señales enviadas a través de TypedIOPort son tratadas como si ellas fueran enviadas a través de DEIOPort con el argumento del despliegue de tiempo igual a cero. Los actores de dominio polimórfico pueden producir eventos en el futuro indirectamente usando el método fireAt() del director. Llamando fireAt(), el actor requiere reactivarse en el futuro. El actor puede entonces producir un evento desplegado durante al reactivación.

### A.1.1.3. LIBRERÍA DE ACTORES DEL DOMINIO DE

El dominio DE tiene una pequeña librería de actores en ptolemy.domains.de.lib.package. Estos actores son particularmente caracterizados por la implementación de las interfases TimedActor y SequenceActor. Estos actores usan el tiempo modelo actual, y además, asume que ellos están desplegando con secuencias de eventos discretos. Algunos de ellos usan la infraestructura dominio-específico, tal como la conveniencia de la clase DEActor y la clase base DETransformer. La clase DETransformer provista en puertos de entrada y de salida que son instancias de DEIOPort. Los actores de

despliegue y servidores usan facilidades de dichos puertos para influenciar la activación de prioridades.

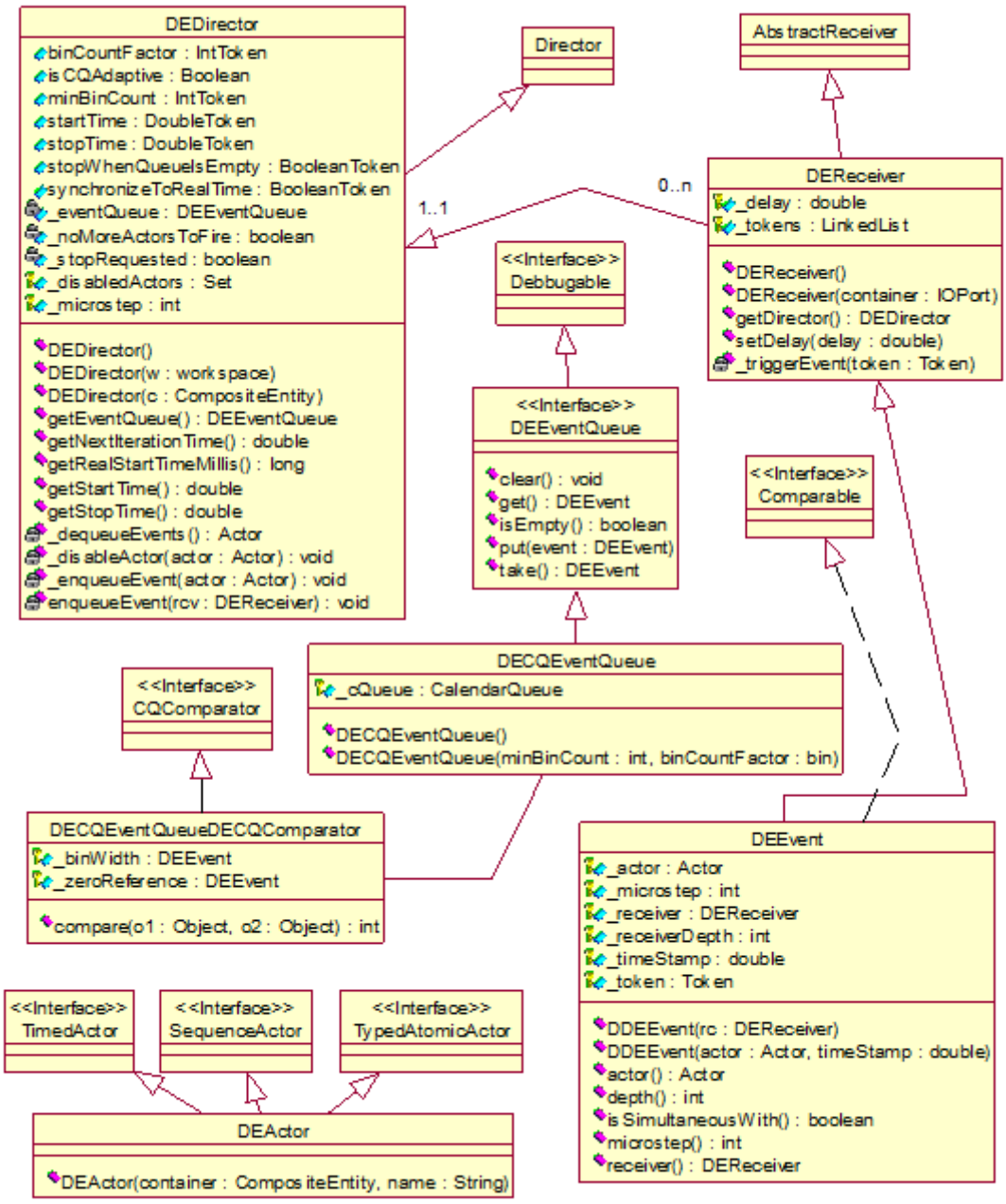


**Figura 1.4.**

### A.1.1.4. MUTACIONES

El director DE tolera cambios en el modelo durante la ejecución. El cambio debería hacer cola con el director o manejador usando requestChange(). Mientras se invocan dichos cambios, el método invalidateSchedule() es esperado para ser anunciado, notificando al director la topología usada para calcular las prioridades de los actores no es más válida. Esto resultará en que las prioridades siendo recalculadas en el próximo tiempo prefire() sean invocadas.

Sin embargo, hay una astucia. Si un actor produce eventos en el futuro vía DEIOPort, entonces el actor destino será activado igual si ha sido removido de la topología porque el tiempo de ejecución alcanza el tiempo futuro. Este puede no ser siempre el comportamiento esperado. El actor Delay en la librería DE se comporta de esta forma.

### A.1.1.5. ESCRIBIENDO ACTORES DE

Es muy común en la modelación DE incluir actores constructores de comportamientos. La librería actor predefinida parece no proveer suficiente para todas las aplicaciones. Para la mayor parte, los actores que escriben para el dominio DE no son diferentes a los actores que escriben para cualquier otro dominio. Algunos actores, sin embargo, necesitan ejercitar un control particular sobre los sellos de tiempo y prioridades actor. Dichos actores usan instancias de DEIOPort más que TypedIOPort. La primera sección abajo da unas directrices generales actores DE escritores y actores polimórficos de dominio que trabajan en DE. La segunda sección explica en detalle las prioridades, en preparación para la siguiente sección, la cual da un ejemplo. La sección final discute actores que operan como un “hilo” de Java.

### A.1.1.5.1. DIRECTRICES GENERALES

Los puntos para mantener en la mente son:

- Cuando un actor se activa, no todos los puertos tienen señales, y algunos puertos pueden tener más de una señal. Los sellos de tiempo de los eventos que contienen dichas señales no están más explícitamente disponibles. El tiempo modelo actual es asumido para ser el sello de tiempo de los eventos.

- Si un actor deja señales sin consumir en sus puertos de entrada, entonces será iterado de nuevo antes de que el tiempo modelo sea avanzado. Esto asegura que el tiempo modelo actual es en efecto el sello de tiempo de los eventos de entrada. Sin embargo, ocasionalmente, un actor querrá dejar señales sin consumir sobre sus puertos de entrada, y no será activado de nuevo hasta que haya algún evento nuevo para ser procesado. Para obtener este comportamiento, se debería colocar en falso prefire(). Esto indica al director DE que no desee ser iterado.

- Si el actor postfire() está en falso, entonces el director no activa el actor de nuevo. Eventos que son destinados para que el actor sea descartado.

- Cuando un actor produce una señal de salida, el sello de tiempo para el evento de salida es tomado para ser el tiempo modelo actual. Si el actor desea producir un evento en un tiempo modelo futuro, una manera para lograrlo es llamar al director método fireAt() para programar una activación futura, y entonces producir la señal en ese tiempo. Una segunda manera para lograrlo es usar instancias de DEIOPort y utilizar los métodos broadcast() o send() sobrecargado que toman un argumento de despliegue del tiempo.

- La clase DEIOPort puede producir eventos en el futuro, pero hay una astucia importante usando dichos métodos. Cuando un evento ha sido producido, no puede ser retractado. En particular, incluso si el actor es borrado antes de que el tiempo modelo alcance el del evento futuro, el evento será entregado para su destino. Si en cambio se usa fireAt() para generar eventos desplegados, entonces el actor es borrado ( o aplica falso en postfire()) antes del evento futuro, entonces el evento futuro no será producido.

- Por convención en Ptolemy II, los actores actualizan su estado únicamente en el método postfire(). En DE, el método fire() es solamente es invocado una vez por iteración, así no hay una razón particular para adherir a esta convención. No obstante, se recomienda que en nuestro caso el actor sea muy útil en otros dominios. La manera más simple para asegurar es continuar con el modelo siguiente. Para cada variable de estado, tal como una variable privada llamada _count,

```
private int count;
```

Crear una imagen variable

```
private int _countShadow;
```

Entonces escribir los métodos como sigue:

```
public void fire ()  {
        _countShadow = _count;
        . . .perform some computation that may modify _countShadow
. . .
        }
```

```
public Boolean postfire () {
        _count = _shadowCount;
        return super.postfire();
}
```

Esto asegura que el estado es actualizado únicamente en postfire().

En una forma similar, las salidas desplegadas (producidas por cualquier mecanismo) deberían ser producidas solamente en el método postfire(), ya que unas salidas desplegadas son estados persistentes. Además, fireAt() debería ser llamado solamente en postfire(), como deberían broadcast() y send() sobrecargado de DEIOPort.

### A.1.1.5.2. EJEMPLOS

Actor delay simplificado. Este actor despliega eventos de entrada por alguna cantidad especificada por un parámetro. Las características del dominio específico del actor son mostradas en negrilla. Ellas son:

- Usa DEIOPort más que TypedIOPort.
- Tiene la declaración:

```
Input.delayTo (output) ;
```

Esta afirmación declara al director que este actor implementa un delay desde la entrada hacia la salida. El actor usa esto para romper las precedencias cuando se construye DAG para encontrar prioridades.

- Usa el método send() sobrecargado, lo cual toma un argumento delay, para producir la salida. Note que la salida es producida en el método postfire(), por la convención en Ptolemy II, el estado persistente no es actualizado en el método fire(), pero es más actualizado en el método postfire().

Actor server. El actor server en la librería DE usa un abundante conjunto de propiedades del comportamiento del dominio DE. Un server es un proceso que toma alguna cantidad de tiempo para servir “clientes.” Mientras se está sirviendo un cliente, otros clientes que van llegando tienen que esperar. Este actor puede tener un tiempo de servicio fijo (vía parámetro serviceTime, o un tiempo de servicio variable, provisto vía puerto de entrada newServiceTime). Un uso típico habría de ser suministrar números aleatorios para el puerto newServiceTime para generar tiempos de servicio aleatorios. Dichos tiempos pueden ser provistos en el mismo tiempo en que los clientes van llegando para recibir un efecto donde cada cliente experimenta un diferente tiempo de servicio seleccionado aleatoriamente.

Este actor extiende DETransformer, lo cual tiene dos miembros públicos, entrada y salida, ambas instancias de DEIOPort. El constructor hace uso del método delayTo() de aquellos puertos para indicar que el actor introduce delay entre sus entradas y salidas.

El actor conserva registro del tiempo en el que estará libre luego el _nextTimeFree variable privado. Este es inicializado con menos infinidad para indicar que siempre que el modelo esté siendo ejecutado, es servidor está libre. El método prefire() determina si el servidor está libre comparando la variable privada contra el tiempo modelo actual. Si está libre, entonces el método indica verdadero, mostrando el programa que pudo ser procesado con la activación del actor. Si el servidor no está libre, entonces el método prefire() chequea para observar si hay una entrada pendiente, y si la hay, requiere una activación cuando el actor llegue a estar libre. Si indica falso, mostrando al programa que no desea ser activado en este tiempo. Note que el método prefire() esa los métodos getCurrentTime() y fireAt() de DEActor, los cuales son simplemente interfases convenientes para métodos del mismo nombre en el director.

El método fire() es invocado solamente si el servidor está libre. Si el primer chequeo para mirar si el puerto newserviceTime es conectado par cualquier cosa, y si esto es, si tiene una señal. Si esto es hecho, la señal es leída y usada para actualizar el parámetro serviceTime. Nomás de una señal es leída, igual si hay más en el puerto de entrada, en este caso una señal está siendo provista por un cliente pendiente.

El método fire() entonces continua leyendo una señal de entrada, si hay una, y actualizando _nextTimeFree. La señal de entrada que es leída es acumulada temporalmente en la variable privada currentInput. El método postfire() entonces produce esta señal sobre el puerto de salida, con un delay apropiado. Esto es completado en el método postfire() más que en el método fire() manteniendo la política en Ptolemy II el estado persistente no es actualizado en el método fire(). La salida es producida con un sello de tiempo futuro, entonces es un estado persistente.

Note que cuando un actor no recibirá señales que están disponibles en el método fire(), es esencial que prefire() quede en falso. En otro caso, el programa DE mantiene activado el actor hasta que todas las entradas sen consumidas, lo cual nuca sucederá si el actor no está consumiendo las entradas!.

El actor SimpleDelay, produce salidas con sellos de tiempo futuro, usando el método send() sobrecargado de DEIOPort que toma un argumento delay. Hay astucia asociada con este diseño. Si el modelo cambia durante la ejecución, y el actor server es borrado, no puede retractar eventos si ya tiene envíos a la salida. Dichos eventos serán observados por el actor destino, igual si por ningún tiempo el servidor ni el destino están en la topología! Esto podría llevar a algunos resultados inesperados, pero esperanzadoramente, si el actor destino no es conectado a cualquier cosa, entonces no hará mucho con la señal.

### A.1.1.5.3. ACTORES THREAD

En algunos casos, es útil describir un actor como un thread que espera señales de entrada sobre sus puertos de entrada. El thread suspende mientras espera señales de entrada y es resumido cuando alguno o todos de sus puertos de entrada tiene señales de entrada. Mientras esta descripción es funcionalmente equivalente a la descripción estándar explicada abajo, esto influye sobre la infraestructura multifibra Java para guardar el estado de información.

Se considera el código para el actor ABRecognizer mostrado en la figura 1.5. Las dos listas de códigos implementan dos actores con un comportamiento equivalente. La izquierda implementa un actor thread, mientras la derecha implementa un actor estándar. Desde ahora nos referiremos a la izquierda como la descripción threaded y a la derecha como la descripción estándar. En ambas, el actor tiene dos puertos de entrada, inportA y inportB, y un puerto de salida, outport. El comportamiento es como sigue.

> Produce un evento de salida en un puerto de salida en cuanto eventos en importa y inportB ocurren en el orden particular, y repite este comportamiento.

Note que la descripción estándar necesita una variable de estado state, al contrario del caso de la descripción threaded. En general la descripción threaded codifica la información del estado en la posición del código, mientras la descripción estándar codifica la información explícitamente usando variables de estado. Mientras es verdadero que el contexto de cambiar sobre la cabeza asociado con aplicación mutifibra reduce el desarrollo, nosotros argumentamos que la simplicidad y claridad de los actores escritores en la forma threaded es bien valioso el costo en algunas aplicaciones.

```
public class ABRecognizer extends DEThreadActor {
   StringToken msg = new StringToken("Seen AB");

   // the run method is invoked when the thread
   // is started.
   public void run() {
      while (true) {
         waitForNewInputs();
         if (inportA.hasToken(0)) {
            IOPort[] nextinport = {inportB};
            waitForNewInputs(nextinport);
            outport.broadcast(msg);
         }
      }
   }
```

```
public class ABRecognizer extends DEActor {
   StringToken msg = new StringToken("Seen AB");

   // We need an explicit state variable in
   // this case.
   int state = 0;

   public void fire() {
      switch (state) {
         case 0:
            if (inportA.hasToken(0)) {
               state = 1;
               break;
            }
         case 1:
            if (inportB.hasToken(0)) {
               state = 0;
               outport.broadcast(msg);
            }
      }
   }
}
```

**Figura 1.5.**

La infraestructura para esta característica es mostrada en la figura 1.4. Para escribir un actor en la forma threaded, es simplemente derivar de la clase DEThreadActor e implementar el método run(). En muchos casos, el contenido del método run() es encerrado en la curva infinita ' while (true) ' de muchos actores thread útiles no terminados.

El método waitForNewInputs() es sobrecargado y tiene dos aspectos, uno que no toma argumentos y otro que toma una serie de IOPort como argumento. El primero suspende el thread hasta que hay por lo menos una señal de entrada en por lo menos en uno de los puertos de entrada. , mientras que el segundo suspende hasta que hay por lo menos una señal de entrada en cualquiera de los puertos de entrada especificados, ignorando todas las otras señales.

En la implementación actual, ambas versiones de waitForNewInputs() limpia todos los puertos de entrada antes de suspender thread. Esto garantiza que el thread resuma, todas las señales disponibles son nuevas, en el sentido que ellas no fueron disponibles antes de llamar el método waitForNewInput().

La implementación también garantiza que entre llamadas al método waitForNewInputs(), el resto del modelo DE es suspendido. Esto es equivalente a decir que la sección de códigos que llama al método waitForNewInputs() es una sección crítica. Una inmediata implicación es que el resultado del método llama un chequeo en la configuración del modelo (por ejemplo, hasToken() para chequear el receptor) no será invalidado durante la ejecución en una sección crítica. Esto significa también que esto no debería ser visto como una forma para recibir una ejecución paralela en DE. Para eso, se considera el dominio DDE.

Es importante notar que la implementación serializa la ejecución de threads, representando que en cualquier tiempo dado hay solamente una thread corriéndose. Cuando un actor threaded está corriéndose (es decir, dentro de una ejecución su método run()), todos los otros actores threaded y el director son suspendidos. Esto mantendrá corriendo hasta que sea alcanzado un estado de waitForNewInputs(), dondeel flujo de la ejecución será transferido al director anterior. Note que el director thread ejecuta todos los actores no threaded. Esta serialización es necesitada porque el dominio DE tiene una noción del tiempo global, lo cual hace al paralelismo mucho más difícil de lograr.

La serialización es ejecutada por el uso del monitor en la clase DEThreadActor. Básicamente, el método fire() de la clase DEThreadActor suspende el llamado de thread (es decir, el director thread) hasta que el actor threaded se suspenda a sí mismo (llamando waitForNewInputs()). Un punto clave de esta implementación es que los actores threaded aparecen justo como un actor DE ordinario al director DE. La clase base DEThreadActor encapsula la ejecución threaded y provee las interfases regulares al director DE. Por lo tanto la descripción threaded puede ser usada siempre que un actor ordinario pueda, lo cual es en todas partes.

Un trabajo futuro sobre esta área puede incluir extender la infraestructura para soportar la concurrencia de varios constructores, tales como prioridad, ejecución

paralela, etc. Sería muy interesante este poderío para explora nuevas semánticas de concurrencia similares al threaded DE, pero sin la serialización “forzada”.

### A.1.1.6. COMPOSICIÓN DE CON OTROS DOMINIOS

Uno de los principales conceptos en Ptolemy II es modelar sistemas heterogéneos a través del uso de heterogeneidad jerárquica. Los actores en el mismo nivel de jerarquía obedecen al mismo conjunto de reglas semánticas. Dentro de algunos de dichos actores pueden haber otros dominios con un diferente modelo de computación. Este mecanismo es soportado a través del uso de actores compuestos opacos. Un ejemplo es mostrado en la figura 1.6. El dominio más exterior es DE y contiene siete actores, dos de ellos son opacos y compuestos. Los actores compuestos opacos contienen subsistemas, los cuales en este caso son los dominios DE y CT.

#### A.1.1.6.1. DE DENTRO DE OTRO DOMINIO

El subsistema DE completa una iteración cuando el actor compuesto opaco es activado por el dominio externo. Una de las complicaciones en la mezcla de dominios es la sincronización del tiempo. Se denota el tiempo actual del subsistema DE por $t_{inner}$ y el tiempo actual del dominio externo por $t_{outer}$. Una iteración del subsistema DE es similar a una iteración de un modelo DE de nivel superior, excepto que prioriza la iteración de señales que son transferidas desde los puertos de los actores compuestos opacos dentro de los puertos del subsistema DE contenido, y después de finalizar la iteración, el director pide una reactivación del sello de tiempo más pequeño en la cola de eventos del subsistema DE.

Lo primero es completar en el método transferInputs() del director. Este método es extendido de su implementación por defecto en el clase directora. La implementación en la clase DEDirector avanza el tiempo actual del subsistema DE al tiempo actual del dominio externo, entonces llama super.transferInputs(). Esto es completado con el fin de asociar correctamente las señales vistas en los puertos de entrada del actor compuesto opaco, si cualquier, con eventos en el tiempo actual del dominio externo, $t_{outer}$, y pone dichos eventos dentro de la cola global de eventos. Este mecanismo es , en efecto, cómo el subsistema DE sincroniza su tiempo actual, $t_{inner,}$ con el tiempo actual del dominio externo, $t_{outer}$ (llama nuevamente al director DE avanza el tiempo mirando el sello de tiempo más pequeño en la cola de eventos del subsistema DE). Específicamente, antes el avance del tiempo actual del subsistema $t_{inner}$ es menor o igual que $t_{outer}$, y después el avanza $t_{inner}$ es igual a $t_{outer.}$

Requiriendo una reactivación es completada en el método postfire() del director llamando el método fireAt() del director ejecutivo. Su propósito es asegurar que los eventos en el subsistema DE sean procesados sobre el tiempo con respecto al tiempo actual del dominio externo, $t_{outer.}$

Notar que si el subsistema DE es activado debido al dominio externo procesando una reactivación requerida, entonces pueden no ser cualquier señales en el puerto de entrada del actor compuesto opaco en el comienzo de la iteración del subsistema DE. En este caso, no hay eventos nuevos con sellos de tiempo igual a $t_{outer}$ los cuales serán puestos dentro de la cola global de eventos. Interesantemente, en este caso, el tiempo de sincronización aquietará el trabajo porque $t_{inner}$ será avanzado al más pequeño sello de tiempo en la cola global de eventos lo cual, a su vez, tiene que ser igual a $t_{outer}$ porque siempre requiere una reactivación de acuerdo con el sello de tiempo.

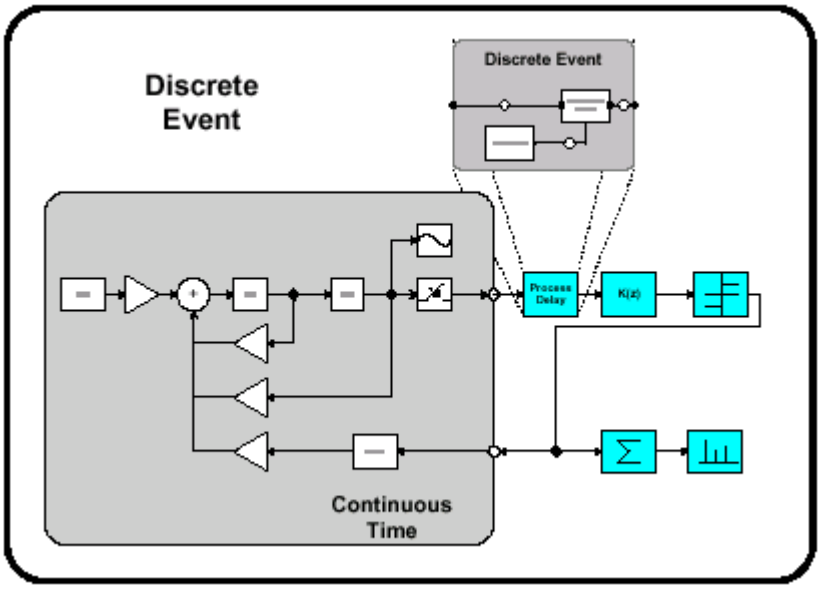


**Figura 1.6.**

### A.1.1.6.2. OTRO DOMINIO DENTRO DEL DOMINIO DE

Debido a su naturaleza, el actor compuesto opaco es opaco y por lo tanto, hasta donde el director DE está involucrado, se comporta exactamente como un actor de dominio polimórfico. Se recuerda que los actores de dominio polimórfico son tratados como funciones con delay cero en el tiempo de cálculo. Para producir eventos en el futuro, los actores de dominio polimórfico requieren una reactivación del director DE y entonces producen los eventos cuando es reactivado.

## A.1.2. DOMINIO DDE

### A.1.2.1. INTRODUCCION

El modelo de computación de eventos discretos distribuidos (DDE) incorpora una noción distribuida de tiempo dentro de un estilo de comunicación de flujo de datos. El tiempo progresa en un modelo DDE cuando los actores en el modelo se ejecutan y comunican. Los actores en un modelo DDE se comunican enviando mensajes a través de limitados canales FIFO. El tiempo en un modelo DDE es distribuido y localizado, y los actores de un modelo DDE cada uno mantiene su propia noción local de tiempo concurrente. La información de tiempo local es compartida entre dos actores conectados siempre que una comunicación entre dichos actores ocurra. Recíprocamente, la comunicación entre dos actores conectados puede ocurrir únicamente cuando las restricciones en la información de tiempo local relativo de los actores se adhieren.

El dominio DDE esta basado en el procesamiento de eventos discretos distribuidos e influencia una rica investigación consagrada a este tema. El dominio DDE implementa una variación especifica de los sistemas de eventos discretos distribuidos (DDES) como es expuesto por Chandy y Misra [18]. Aún cuando el dominio DDE tiene similitudes con DDES, el dominio de eventos discretos distribuidos sirve como una estructura para el estudio de DDES con dos énfasis especiales. Primero se va a considerar a DDES desde una perspectiva de flujo de datos; se va a ver a DDE como una implementación del modelo de flujo de datos de Kahn [41] con tiempo distribuido adicionado en la parte superior. Segundo se estudia DDES no con el objetivo de mejorar la velocidad de ejecución (como ha sido tradicionalmente el caso). En lugar de eso se estudia DDES para aprender su

utilidad en el modelamiento y diseño de sistemas que son temporizados y distribuidos.

### A.1.2.2. USANDO EL DOMINIO DDE

El dominio DDE es tipeado (typed) así que los actores usados en un modelo deben ser derivados de ptolemy/actor/TypedAtomicActor. El dominio DDE es diseñado para usar actores específicos DDE así como actores polimorficos. Los actores específicos DDE pueden tomar ventaja de DDEActor y DDEIOPort los cuales son diseñados para proporcionar soporte conveniente para la especificación de tiempo en la producción y el consumo de señales.

#### A.1.2.2.1. DDEActor

El modelo de computación DDE hace una fortísima suposición acerca de la ejecución de un actor: todos los puertos de entrada de un actor operando en un modelo DDE deben ser censadas regularmente para determinar cual canal de entrada tiene el evento pendiente más viejo. Cualquier actor que adhiere a esta suposición puede operar en un modelo DDE. Así, muchos actores polimorficos encontrados en Ptolemy/actor/[lib, gui] son adecuados para la operación en los modelos DDE. Por conveniencia, el DDEActor fue desarrollado para simplificar la construcción de actores que tienen semántica DDE. El DDEActor tiene tres métodos claves como los que sigue:

getCurrentTime(). Este método retorna la noción de tiempo local del hilo del actor. Este método depende de Thread.currentThread() de Java retornando el hilo que es asignado al actor. Si el actor A llama a getCurrentTime() del actor B, entonces el resultado será el tiempo actual de A (no de B como uno podría esperar).

getNextToken(). Este método censa cada puerto de entrada de un actor y retorna la señal (no nula) que representa el evento más viejo.

getLastPort(). Este método retorna la entrada IOPort de la cual la última señal (no nula) fue consumida. Este método supone que getNextToken() esta siendo usado para el consumo de la señal.

#### A.1.2.2.2. DDEIOPort

DDEIOPort extiende TypedIOPort con parámetros para especificación de valores de huellas de tiempo de señales que están siendo enviadas a actores vecinos. Ya que DDEIOPort extiende TypedIOPort, el uso de DDEIOPort no violará el proceso de resolución de tipo. DDEIOPort no es necesario para facilitar la comunicación entre actores que se ejecutan en un modelo DDE; los TypedIOPorts estándar son suficientes en muchas comunicaciones. DDEIOPorts se hace útil cuando la huella de tiempo que esta asociada con una señal saliente es mayor que el tiempo actual del actor de envío. Por lo tanto, los DDEIOPorts son útiles únicamente en conjunción con actores retardos. La mayoría de los actores polimorficos disponibles para Ptolemy II no son actores retardo.

#### A.1.2.2.3. TOPOLOGÍAS REALIMENTADAS

Con el fin de ejecutar topologías realimentadas que no se deadlock, los actores FBDelay deben ser usados. El FBDelay es encontrado en el paquete del kernel DDE. Los actores FBDelay no ejecutan la computación, pero en lugar de eso incrementan las huellas de tiempo de las señales que fluyen a través de ellas por un retardo especifico. El valor del retardo de un actor FBDelay debe ser escogido para ser menor que el delta de tiempo del ciclo realimentado en el cual el actor FBDelay es contenido. La elaboración de los valores de retardo puede ser especificada anulando el método getDelay() en subclases de FBDelay. Un ejemplo

de esto puede ser encontrado en ptolemy/domains/dde/demo/LocalZeno/ ZenoDelay.java.

Una dificultad encontrada en los ciclos realimentados ocurre en la inicialización de la ejecución de un modelo. En la figura 2.1 se ve que aún si el actor B es un actor FBDelay, el sistema se deadlock si el primer evento es creado por A ya que C se bloqueará en un evento de B. Para aliviar este problema un valor especial de huella de tiempo ha sido reservado: TimeQueuedReceiver.IGNORE. Cuando un actor encuentra un evento con una huella de tiempo de IGNORE (un evento ignorar), el actor ignorará el evento y el canal de entrada con el que esta asociado. El actor entonces considera los otros canales de entrada determinando el próximo evento disponible. Después de que un evento no ignorado es encontrado y consumido por el actor, todos los eventos ignorar serán aclarados. Si todos los canales de entrada de un actor contienen eventos ignorar, entonces el actor aclarará todos los eventos ignorar y luego procede con la operación normal.

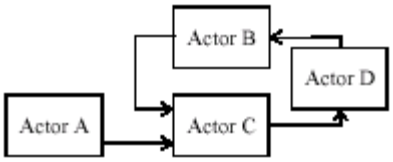


**Figura 2.1.**

El método initialize de FBDelay produce un evento ignorar. Así, en la figura 2.1, si B es un actor FBDelay, el evento ignorar que produce será enviado al canal de entrada superior de C permitiendo a C consumir el primer evento de A. La producción de señales nulas y retardos realimentados serán entonces suficientes para continuar la ejecución en ese punto. Nótese que la producción de un evento ignorar por un actor FBDelay sirve como una mayor distinción entre él y todos los otros actores. Si un retardo es deseado simplemente para representar el retardo computacional de un modelo dado, un actor FBDelay no debe ser usado.

La operación intrincada de ignorar eventos requiere una consideración especial cuando se determina el posicionamiento de un actor FBDelay en una topología realimentada. Un actor FBDelay debe ser colocado a fin de que el evento ignorar que produce sea ignorado con respecto a el primer evento real que entra en un ciclo realimentado. Así, la escogencia de un actor D como un actor FBDelay en la figura 2.1 no sería útil dado que el primer evento real entrando en el ciclo es creado por A.

### A.1.2.3. PROPIEDADES DEL DOMINIO DDE

Operacionalmente, la semántica del dominio DDE puede ser separada en dos funcionalidades. La primera funcionalidad se relaciona con como el tiempo avanza durante la comunicación de datos y como la comunicación procede vía bloqueos de lectura y escritura. La segunda dependencia considera como un modelo DDE previene un deadlock debido a las dependencias de tiempo local. La técnica para prevenir un deadlock involucra la comunicación de mensajes nulos que consisten solamente de información de tiempo local.

#### A.1.2.3.1. HABILITANDO COMUNICACIONES

**Comunicación de señales**. Un modelo DDE consiste de una red de actores que están conectados vía una cola FIFO limitada unidireccional. Las señales son enviadas desde un actor trasmisor a un actor receptor colocando una señal en la cola apropiada donde la señal es almacenada hasta que el actor receptor la consume. Si un proceso intenta leer una señal de una cola que esta vacía, entonces el proceso se bloqueará hasta que una señal este disponible en el canal. Si un proceso intenta escribir una señal en una cola que esta llena, entonces el proceso se bloqueará hasta que un espacio este disponible para más señales en esa cola. Nótese que este paradigma del bloqueo de lectura/escritura es

equivalente a la semántica operacional encontrada en las redes de proceso sin tiempo (PN) como es implementada en Ptolemy II (ver el capítulo del dominio PN).

Si todos los procesos en un modelo DDE simultáneamente se bloquean, entonces el modelo tiene un deadlock. El deadlock es debido a procesos que están esperando a leer de una cola vacía, bloqueo de lectura, o esperando a escribir en una cola llena, bloqueo de escritura, entonces se dice que el modelo ha experimentado un deadlock no temporizado. Un deadlock no temporizado es equivalente a la noción de deadlock encontrada en los problemas de planeación de redes de procesos limitadas como lo bosqueja Parks [68]. Si un deadlock no temporizado es debido a un modelo que consiste exclusivamente de procesos que están bloqueados en lectura, entonces se dice que un deadlock real ha ocurrido y el modelo es terminado. Si un deadlock no temporizado es debido a un modelo que consiste de al menos un proceso que esta bloqueado en escritura, entonces la capacidad de las colas llenas es incrementada hasta que no exista un deadlock. Tales deadlocks son llamados deadlocks artificiales, y la política de incrementar la capacidad de las colas llenas fue mostrada por Parks para garantizar la ejecución de un modelo en memoria limitada siempre que sea posible.

**Comunicación de Tiempo**. Cada actor en un modelo DDE mantiene una noción de tiempo local. Como las señales son comunicadas entre actores, las huellas de tiempo están asociadas con cada señal. Cuando quiera que un actor consume una señal, el tiempo actual del actor es puesto para ser igual a la huella de tiempo de la señal consumida. El valor de la huella de tiempo aplicado a las señales salientes de un actor es equivalente al tiempo de salida de ese actor. Para actores que modelan un proceso en el cual hay retardo entre las huellas de tiempo de llegada y las correspondientes huellas de tiempo de salida, el tiempo de salida es siempre más grande que el tiempo actual; de otra forma, el tiempo de salida es igual al tiempo actual. Se refiere a los actores del caso anterior como actores retardos (delay actors).

Para una cola dada que contiene señales con huella de tiempo, la huella de tiempo de la primera señal generalmente contenida por la cola es referida como el tiempo de recepción de la cola. Si una cola esta vacía, su tiempo de recepción es el valor de la huella de tiempo asociada con la última señal para correr a través de la cola, o 0.0. si ninguna señal ha viajado a través de la cola. Un actor podría consumir una señal de una cola de entrada dado que la cola tiene una señal disponible y el tiempo de recepción es menos que los tiempos de recepción de todas las otras colas de entrada contenidas por el actor. Si la cola con el tiempo de recepción más pequeño esta vacía, entonces el actor bloquea hasta que esta cola reciba una señal, en cuyo tiempo el actor considera el tiempo de recepción actualizado en selección una cola para leer.

La figura 2.2 muestra tres actores, cada uno con tres colas de entrada. El actor A tiene dos señales disponibles en la cola superior, ninguna señal disponible en la cola del medio y una señal disponible en la cola del fondo. Los tiempos de recepción de las colas superior, del medio y del fondo son 17.0, 12.0 y 15.0 respectivamente. Ya que la cola con el mínimo tiempo de recepción (la cola del medio) esta vacía, A debe bloquear en esta cola antes de proceder. En el caso del actor B, el tiempo de recepción mínimo pertenece a la cola del fondo. Así, B procedería consumiendo la señal encontrada en la cola del fondo. Después de consumir esta señal, B compararía luego todos sus tiempos de recepción para determinar cual señal podría consumir a continuación. El actor C es un ejemplo de un actor que contiene múltiples colas de entrada con idénticos tiempos de recepción. Para ajustar esta situación, cada actor asigna una única prioridad a cada cola de entrada. Un actor puede consumir una señal de una cola si ninguna otra cola tiene un tiempo de recepción más bajo y si todas las colas que tienen un tiempo de recepción idéntico también tienen una prioridad más baja.

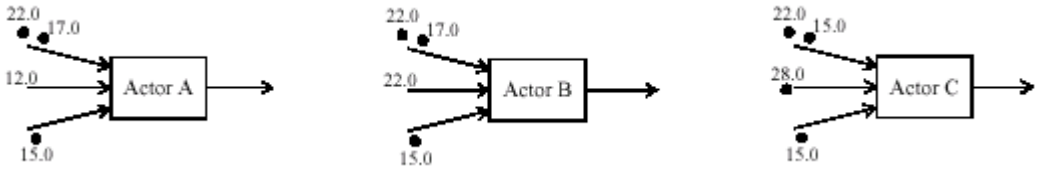


**Figura 2.2.**

Cada receptor tiene un tiempo de terminación (completion time) que es puesto durante la inicialización de un modelo. El tiempo de terminación del receptor especifica el tiempo después del cual el receptor ya no operará. Si la huella de tiempo de la señal más vieja en un receptor excede el tiempo de terminación, entonces ese receptor se convertirá en inactivo.

### A.1.2.3.2. MANTENIENDO LA COMUNICACION

Los puntos muertos (deadlocks) pueden ocurrir en un modelo DDE en una forma que difiere de los puntos muertos descritos en las secciones anteriores. Esta forma alternativa de deadlock ocurre cuando un actor bloquea en lectura en un puerto de entrada aún cuando contiene otros puertos con señales. La topología de un modelo DDE puede llevar a un deadlock como los actores bloqueados en lectura esperando uno al otro por señales con huellas de tiempo que nunca aparecerán. La figura 2.3 ilustra este problema. En esta topología, considere una situación en la cual un actor A crea señales en su cola de salida más baja. Esto llevará a las señales siendo creadas en la cola de salida del actor C pero ninguna señal será creada en la cola de salida de B (ya que B no tiene señales para consumir). Esta situación resulta en un bloque de lectura indefinido en D en su cola de entrada superior aún cuando sea claro que ninguna señal correrá a través de esta cola. El resultado: un deadlock temporizado! La situación mostrada en la figura 2.3 es únicamente un ejemplo de un deadlock temporizado. De hecho hay dos tipos de deadlock temporizado: feedforward y feedback (retroalimentación).

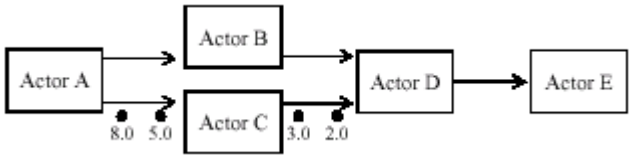


**Figura 2.3.**

La figura 2.3 es un ejemplo de un deadlock feedforward. Un deadlock feedforward ocurre cuando un grupo de actores conectados están en deadlock tal que todos los actores en el grupo están bloqueados en lectura y al menos uno de los actores en el grupo esta bloqueado en lectura en una cola de entrada que tiene un tiempo de recepción que es menor que el reloj local del actor fuente de la cola de entrada. En el ejemplo mostrado anteriormente, la cola de entrada superior de B tiene un tiempo de recepción de 0.0 aún cuando el reloj local de A ha avanzado a 8.0.

Un deadlock feedback ocurre cuando un grupo de actores conectados cíclicamente están en un deadlock tal que todos los actores en el grupo están bloqueados en lectura y al menos un actor en el grupo, por ejemplo el actor X, esta bloqueado en lectura en una cola de entrada que puede leer señales que son directa o indirectamente un resultado de salida de ese mismo actor (el actor X). La figura 2.4 es un ejemplo de un deadlock temporizado feedback. Nótese que B no puede producir una salida basada en el consumo de la señal con una huella de tiempo en 5.0 puesto que debe esperar por una señal en la entrada superior que depende de la salida de B!

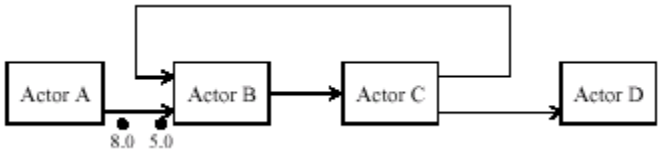


**Figura 2.4.**

**Prevención de un punto muerto de tiempo feedforward**. Para manejar un punto muerto de tiempo feedforward, las señales nulas son empleadas. Una señal

empleada proporciona un actor con unos medio de comunicación de avance de tiempo aún cuando los datos (señales reales) no están siendo transmitidos. Siempre que un actor consume una señal, coloca una señal nula en cada una de sus colas de salida tal que la huella de tiempo de la señal nula es igual al tiempo actual del actor. Así, si el actor A de la figura 2.3, produjo una señal en su cola de salida inferior en el tiempo 5.0, también produciría una señal nula en su cola de salida superior en el tiempo 5.0.

Si un actor encuentra una señal nula en una de sus colas de entrada, entonces el actor hace lo siguiente. Primero consume las señales de todas las otras colas de entrada que contiene dado que las otras colas de entrada tienen tiempos de recepción que son menores o iguales a la huella de tiempo de la señal nula. A continuación el actor remueve la señal nula de la cola de entrada, coloca su tiempo actual igual a la huella de tiempo de la señal nula y produce una señal nula en todas las colas de salida tal que las señales nulas producidas están con la huella de tiempo del tiempo actual. Como un ejemplo, si B en la figura 2.3 consume una señal nula en su entrada con una huella de tiempo de 5.0 entonces también produciría una señal nula en su salida con una huella de tiempo de 5.0.

El resultado de usar señales nulas es que la información del tiempo es uniformemente propagada a través de una topología del modelo. La belleza de las señales nulas es que ellas informa a los actores de la inactividad en otros componentes de un modelo sin requerir la difusión centralizada de esta información. Dado el uso de las señales nulas, un deadlock temporizado feedforward es evitado en la ejecución de los modelos DDE. Es importante que las señales nulas son usadas solamente para el propósito de evitar deadlocks. Las señales nulas no representan ningún componente existente del sistema físico que esta siendo modelado. Por lo tanto, una señal nula no puede ser vista como una señal real. Además, la producción de una señal nula que es el directo resultado del consumo de una señal nula no es considerada computación del punto de vista

del sistema que esta siendo modelado. La idea de las señales nulas fue adoptada primero por Chandy and Misra [18].

**Prevención de un punto muerto de tiempo realimentado**. Se maneja un punto muerto de tiempo realimentado como sigue. Todos los ciclos realimentados son necesarios para tener un incremento en la huella de tiempo acumulativo que es mayor que cero. En otras palabras, los ciclos realimentados son necesarios para contener actores de retardo.

La arquitectura del software de DDE proporciona un actor retardo para usarlo en la prevención de deadlock temporizado feedback: el FBDelay. Se ha omitido un detalle operacional de FBDelay que se relaciona con el proceso de inicialización de la ejecución en un modelo con un retardo realimentado.

### A.1.2.3.3. METODOS ALTERNATIVOS DE EVENTOS DISCRETOS DISTRIBUIDOS

El campo de la simulación de los eventos discretos distribuidos, también se refiere a la simulación de eventos discretos paralelos, (PDES), ha sido un área activa de investigación desde finales de los 70’s. Recientemente ha habido un resurgimiento de la actividad. Esto es debido en parte a la gran disponibilidad de infraestructuras distribuidas para simulaciones de alojamiento en servidores (hosting) y la aplicación de técnicas de simulación paralela para dominios orientados a la no-investigación. Por ejemplo, varios motores de búsqueda WWW están basados en la tecnología de estaciones de trabajo de red.

### A.1.2.4. ARQUITECTURA DEL SOFTWARE DEL DOMINIO DDE

Para un modelo tener semántica DDE, debe tener un DDEDirector controlándolo. Esto asegura que los receptores en los puertos son DDEReceivers. Como con todos los dominios de procesos, cada actor en un modelo DDE está bajo el control de un ProcessThread, o en el caso de DDE, un DDEThread. Un DDEThread contiene un timeKeeper que maneja la noción de tiempo local que es asociado con el actor de DDEThread.

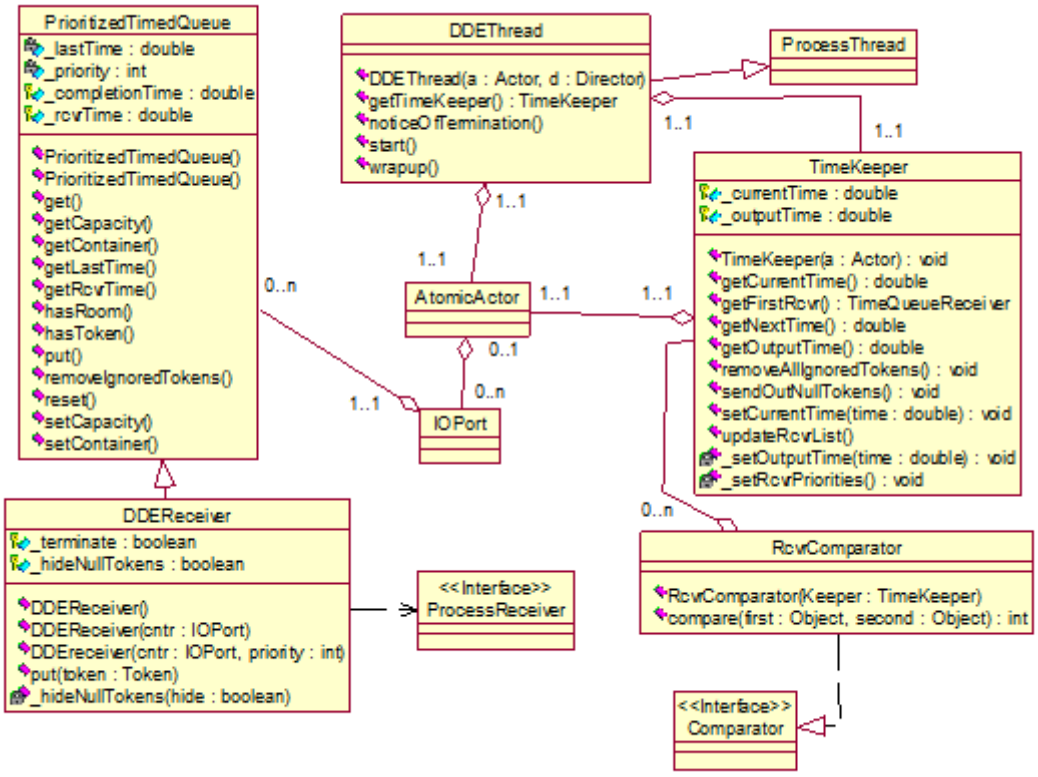


**Figura 2.5.**

### A.1.2.4.1. MANEJO DEL TIEMPO LOCAL

El sistema de manejo del tiempo local del dominio DDE consiste de las clases TimedQueueReceiver, DDEReceiver, DDEThread y TimeKeeper. Ya que el tiempo

es localizado, el DDEDirector no tiene un rol directo en este proceso. Nótese que DDEReceiver es derivado de TimedQueueReceiver. El propósito primario de TimedQueueReceiver es seguir la información de tiempo local de un receptor. El DDEReceiver adiciona la funcionalidad de bloqueo de lectura/escritura a TimedQueueReceiver.

Cuando un DDEDirector es inicializado, el instancia un DDEThread para cada actor que el director maneja. Los DDEThreads son divididos de ProcessThreads. ProcessThreads proporcionan funcionalidad que es común para todos los dominios de procesos (e.g., CSP, DDE y PN). Los directores de todos los dominios de procesos (incluyendo DDE) asignan un simple actor a cada ProcessThread. ProcessThreads toman la responsabilidad de la ejecución de sus actores asignados invocando los métodos de iteración del actor. Los métodos de iteración son prefire(), fire() y postfire(); ProcessThreads también invocan a wrapup() en los actores que ellos controlan.

El DDEThread extiende la funcionalidad de ProcessThread. Encima de la instanciación, un DDEThread crea un objeto TimeKeeper y asigna este objeto al actor que controla. El TimeKeeper consigue acceso a cada uno de los DDEReceivers que el actor contiene. Cada uno de los receptores puede accesar el TimeKeeper y a través del TimeKeeper los receptores pueden entonces determinar sus tiempos de recepción relativos. Con esta información, los receptores están completamente equipados para aplicar las reglas de bloqueo apropiadas así como conseguir y poner señales con huellas de tiempo.

### A.1.2.4.2. DETECTANDO PUNTOS MUERTOS

El propósito del DDEDirector es detectar y (si es posible) resolver deadlocks temporizados o/y no temporizados del modelo que controla. Siempre que un receptor se bloquea, informa al director. El director sigue el número de procesos

activos, y el número de procesos que están bloqueados ya sea en lectura o escritura. Los deadlocks artificiales son resueltos incrementando la capacidad de la cola de los receptores bloqueados en escritura.

### A.1.2.4.3. FINALIZANDO LA EJECUCION

La ejecución de un modelo termina si un deadlock que no tiene solución ocurre, el tiempo de terminación del director es excedido por todos los actores que maneja, o una terminación temprana es requerida (e.g., por un botón de interfaz de usuario). El tiempo de terminación del director es puesto vía el parámetro público stopTime de DDEDirector. El tiempo de terminación es pasado a cada DDEReceiver. Si el tiempo de recepción de un receptor excede el tiempo de terminación, entonces el receptor se vuelve inactivo. Si todos los receptores de un actor se vuelven inactivos y el actor no es un actor fuente, entonces el actor terminará la ejecución y su método wrapup() será llamado. En tal escenario, se dice que el actor ha terminado normalmente.

Las terminaciones tempranas y los deadlocks que no tienen solución comparten un mecanismo común para terminar la ejecución. Cada DDEReceiver tiene una bandera _terminate booleana. Si la bandera es puesta en verdadero, entonces el receptor lanzará un TerminateProcessException la próxima vez que cualquiera de sus métodos sea invocado. Los TerminateProcessExceptions son parte del paquete ptolemy/actor/process y ProcessThreads sabe que para terminar la ejecución de un actor si esta excepción es atrapada. En el caso de un deadlock que no tiene solución, la bandera _terminate de todos los receptores bloqueados es puesta en verdadero. Los receptores son entonces despertados del bloqueo y cada uno de ellos lanza la excepción.

### A.1.2.5. EJEMPLOS DE APLICACIONES DDE

Para ilustrar la ejecución de los eventos discretos distribuidos, se ha desarrollado un applet que presenta una topología realimentada e incorpora polimorfismo así como actores específicos DDE. El modelo, mostrado en la figura 2.6, consiste de un actor fuente sencillo (ptolemy/actor/lib/Clock) y una rama superior e inferior de cuatro actores cada una. Las ramas superior e inferior tienen idénticas topologías y son alimentadas un flujo idéntico de señales de la fuente Clock con la excepción que en la rama inferior ZenoDelay reemplaza a FBDelay.

Como con todas las topologías realimentadas en los modelos DDE (y DE), un retardo de tiempo positivo es necesario en ciclos realimentados para prevenir deadlocks. Si el retardo de tiempo de un ciclo dado es definido inferior por cero pero no puede ser garantizado para ser mayor que el valor positivo fijo, entonces una condición Zeno ocurre en cuyo tiempo no avanzará más allá de un cierto punto aún cuando los actores del ciclo realimentado continúen la ejecución sin deadlocking. El ZenoDelay extiende FBDelay y es diseñado para que una condición Zeno sea encontrada. Cuando la ejecución de un modelo empieza, FBDelay y ZenoDelay son usados para realimentar señales nulas en Wire para que el modelo no deadlock. Después de que el tiempo excede un valor prefijado, ZenoDelay reduce su retardo para que la rama inferior aproxime una condición Zeno.

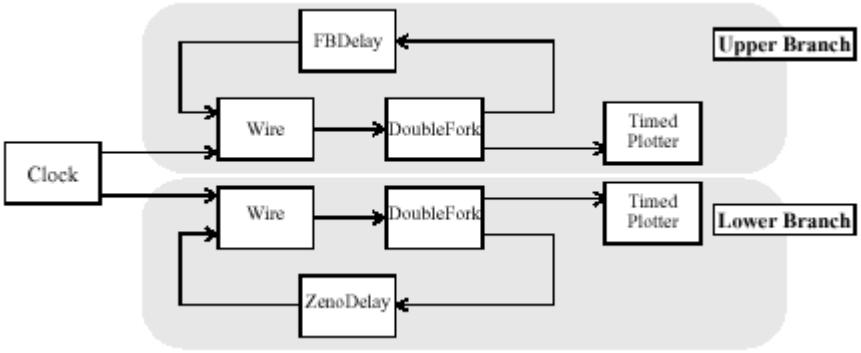


**Figura 2.6.**

En sistemas de eventos discretos centralizados, las condiciones Zeno previenen progresos en el modelo completo. Esto es cierto porque el ciclo realimentado que experimenta la condición Zeno previene el tiempo avance en el modelo completo. En contraste, los sistemas de eventos discretos distribuidos localizan las condiciones Zeno tanto como sea posible basada en la topología del sistema. Así, una condición Zeno puede existir en la rama inferior y la rama superior continuará su ejecución sin impedimentos. La localización de las condiciones Zeno puede ser útil en el modelamiento de gran escala en los cuales una condición Zeno podría no ser descubierta hasta que una gran cantidad de tiempo ha sido invertida en la ejecución del modelo. En tales situaciones, la colección de datos parciales podría proceder previamente a la corrección del error del retardo que resulta en la condición Zeno.

## A.1.3. DOMINIO SDF

### A.1.3.1. INTRODUCCION

El dominio de flujo de datos sincrónico (SDF) es utilizado para el modelamiento de sistemas de flujo de datos simples sin un complicado flujo de control, tales como sistemas de procesamiento de señales. Bajo el dominio SDF, el orden de la ejecución de los actores es determinado estáticamente antes de la ejecución. Esto produce una ejecución con un mínimo de encabezado, así como un uso limitado de memoria y la garantía de que un deadlock nunca ocurrirá. Las aplicaciones que requieran una planeación dinámica podrían usar el dominio de redes de procesos en lugar de este.

### A.1.3.2. USANDO EL DOMINIO SDF

Hay tres aspectos importantes que se deben tener en cuenta cuando se trabaja con el dominio SDF.

- Puntos muertos (deadlocks).
- Consistencia de tasas de datos.
- El valor del parámetro iteraciones.

### A.1.3.2.1. PUNTOS MUERTOS (Deadlocks)

Considere el modelo SDF mostrado en la figura 3.1. Este actor tiene un ciclo realimentado de la salida del actor AddSubtract a su propia entrada. Al intentar ejecutar el modelo se mostrara la excepción que aparece a la derecha en la figura. El director es incapaz de planear el modelo porque la entrada del actor AddSubtract depende de los datos de su propia salida. En general, los ciclos realimentados pueden resultar en tales condiciones.

El arreglo para esta condición de deadlock es utilizar un actor SampleDelay, el cual se muestra en la figura 3.2. Este actor inyecta dentro del ciclo realimentado una señal inicial, el valor de esta es dado por el parámetro initialOutputs del actor. En la figura, este parámetro tiene valor {0}. Este es un arreglo con una sola señal, un entero con valor 0. Un retardo doble con valores iniciales 0 y 1 puede ser especificado usando un arreglo de dos elementos, como {0, 1}.

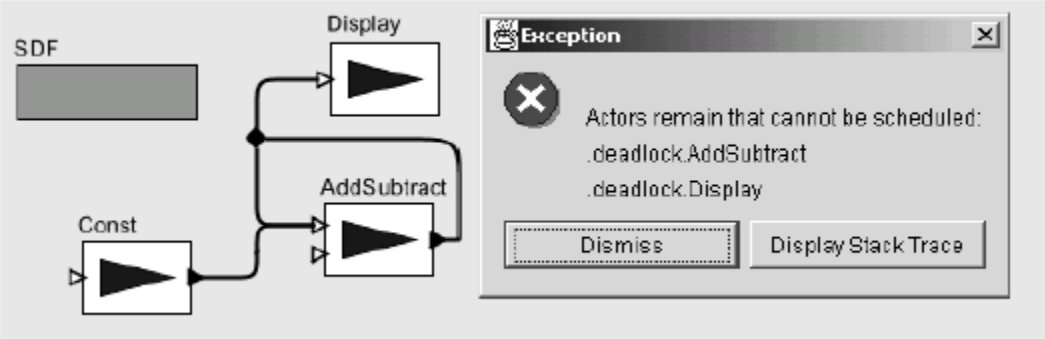


**Figura 3.1.**

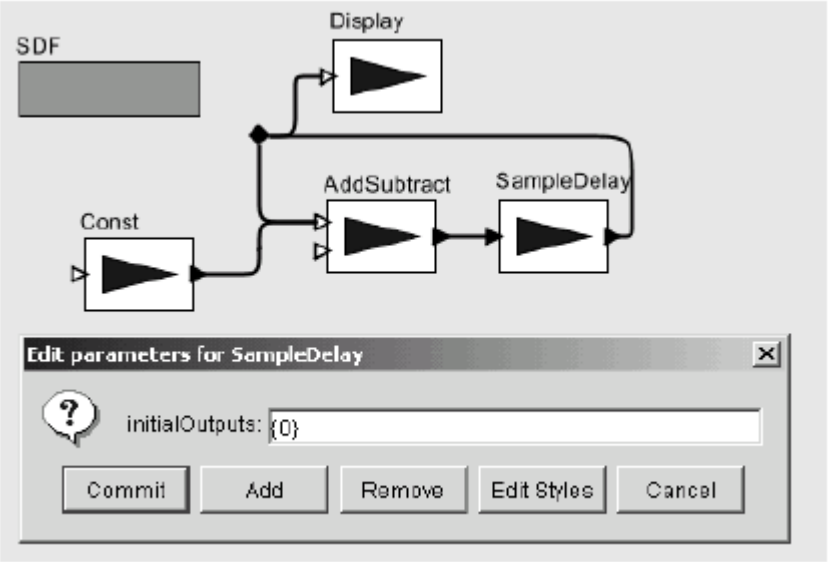


**Figura 3.2.**

### A.1.3.2.2. CONSISTENCIA DE TASA DE DATOS

Considere EL modelo mostrado en la figura 3.3. El modelo esta intentando dibujar una onda seno y su muestra especificada por el actor DownSample. Sin embargo, hay un error porque el número de señales en cada canal del puerto de entrada del diagramador no puede ser igual. El actor DownSample declara que consume 2 señales usando el parámetro tokenConsuptionRate de su puerto de entrada. Su puerto de salida similarmente declara que produce únicamente una señal, así que habrán únicamente la mitad de señales siendo dibujadas del actor DownSample como de la onda seno.

El modelo arreglado es mostrado en la figura 3.4, el cual usa dos graficadores separados. Cuando el modelo es ejecutado, el graficador de abajo se activara cada vez que lo haga el de arriba. Nótese que el problema aparece porque uno de los actores (en este caso el actor DownSample) produce o consume más de una señal en uno de sus puertos. una forma fácil de asegurar la consistencia de tasa es utilizar actores que únicamente producen y consumen una señal a la vez. Este

caso especial es conocido como SDF homogéneo. Nótese que actores como SequencePlotter el cual no especifica parámetros de tasa se asumen como homogéneos.

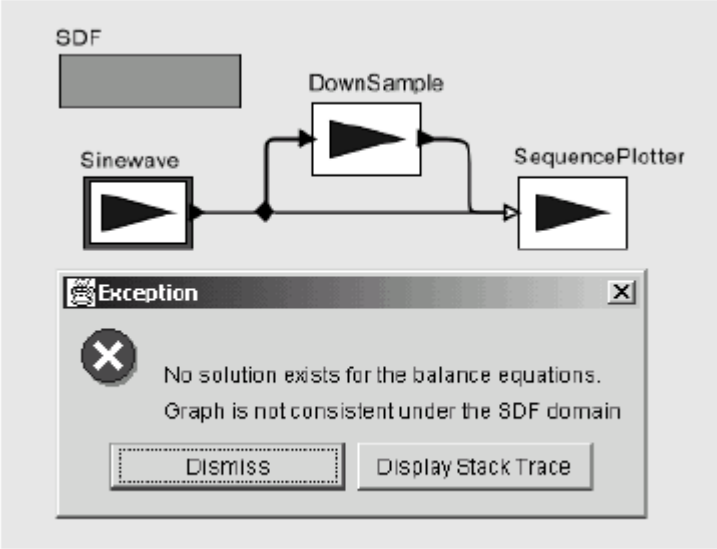


**Figura 3.3.**

### A.1.3.2.3. ¿CUÁNTAS ITERACIONES?

Un aspecto final cuando se esta usando el dominio SDF tiene que ver con el valor del parámetro iteraciones del director SDF. En modelos homogéneos una señal es usualmente producida por cada iteración. Sin embargo, cuando las tasas de señales con excepción de una son usadas, más de un valor de salida interesante puede ser creado por cada iteración. Por ejemplo, considere la figura 3.5 la cual contiene un modelo que grafica la transformada de Fourier de la señal de entrada. Lo importante de este modelo es que el actor FFT declara que consume 256 señales de su puerto de entrada y produce 256 señales en su puerto de salida, correspondiente a una transformada de orden 8. Esto quiere decir que únicamente una iteración es necesaria para producir todos los 256 valores de la transformada.

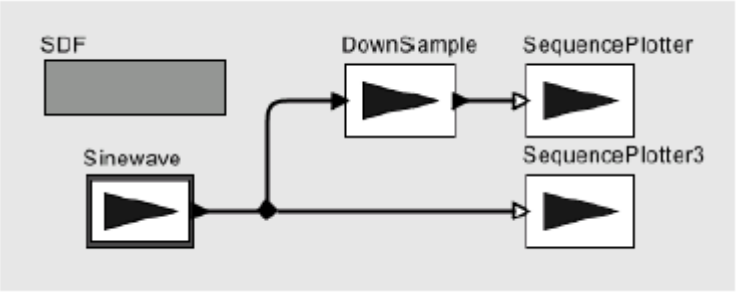


**Figura 3.4.**

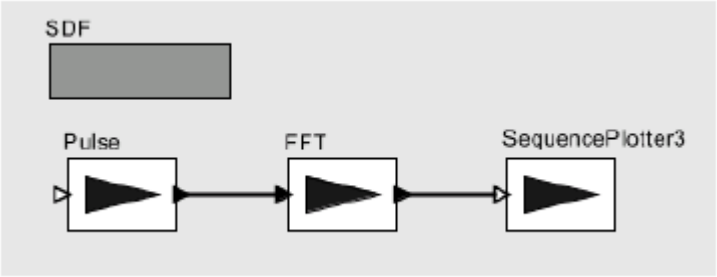


**Figura 3.5.**

Compare este con el modelo de la figura 3.6. Este modelo grafica valores individuales de la señal. Aquí 256 iteraciones son necesarias para ver toda la señal de entrada, ya que únicamente un valor de salida es graficado en cada iteración.

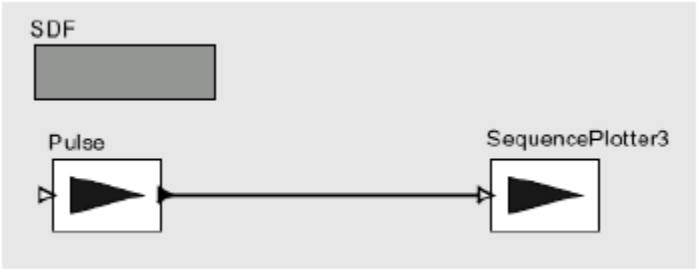


**Figura 3.6.**

### A.1.3.3. PROPIEDADES DEL DOMINIO SDF

SDF es un modelo de computación sin noción de tiempo. Todos los actores bajo el dominio SDF consumen señales de entrada, realizan su computación y producen salidas en una operación atómica. Si un modelo SDF es empotrado entre un modelo de tiempo, entonces el modelo SDF se comportara como un actor de retraso cero.

Además, SDF es un dominio planeado estáticamente. La activación de un actor compuesto corresponde a una simple iteración del modelo contenido. Una iteración SDF consiste de una ejecución de la planeación SDF precalculada. La planeación es calculada de manera que el número de señales en cada relación sea el mismo en el final de una iteración así como en el principio. Así, un número infinito de iteraciones pueden ser ejecutadas, sin un deadlock o una acumulación infinita de señales en cada relación.

La ejecución en SDF es extremadamente eficiente debido a la ejecución planeada. Sin embargo, para ejecutar tan eficientemente, alguna información extra debe ser dada al planeador. Es de importancia que la velocidad de los datos en cada puerto debe ser declarada antes de la ejecución. La velocidad de los datos representa el número de señales producidas o consumidas en un puerto durante cada activación. La velocidad de los datos debe ser determinada antes de la ejecución y debe ser constante a lo largo de la ejecución. Además, los retardos explícitos deben ser adicionados para realimentar los ciclos para prevenir deadlocks.

#### A.1.3.3.1. PLANEACION

El primer paso en la construcción de la planeación es resolver las ecuaciones de balance [47]. Estas ecuaciones determinan el número de tiempos que cada actor se activará durante una ejecución. Por ejemplo, considere el sistema de la figura

3.7. El planeador creara el siguiente sistema de ecuaciones, donde ProductionRate y ConsumptionRate están declarando propiedades de cada puerto, y Firings es una propiedad de cada actor que será resuelto por:

Firings(A) X ProductionRate(A1) = Firings(B) X ConsumptionRate(B1) (39)
Firings(A) X □ProductionRate(A2) = Firings(C) X ConsumptionRate(C1) (40)
Firings(C) X □ProductionRate(C2) = Firings(B) X ConsumptionRate(B2) (41)

Estas ecuaciones expresan restricciones en que el número de señales creadas en una relación durante una iteración es igual al número de señales consumidas. Estas ecuaciones usualmente tienen un número infinito de soluciones dependientes linealmente, y la solución entera menos positiva para Firings es escogida como el vector firing.

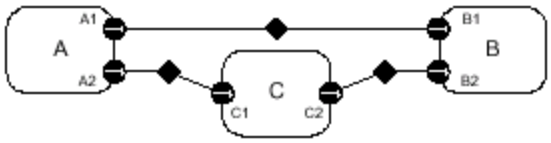


**Figura 3.7.**

En algunos casos, una solución no-cero en las ecuaciones de balance no existe. Tales modelos se dice que son inconsistentes, y su ejecución es prohibida bajo SDF. Los gráficos inconsistentes inevitablemente resultan en un deadlock o en uso de memoria ilimitado para cualquier planeación. Como tal, los gráficos inconsistentes son usualmente bugs en el diseño de un modelo. Sin embargo, los gráficos inconsistentes pueden seguir siendo ejecutados usando el dominio PN, si el comportamiento es verdaderamente necesario. Ejemplos de gráficos consistentes e inconsistentes son mostrados en la figura 3.8.

El segundo paso en la construcción de una planeación SDF es el análisis del flujo de datos. El análisis del flujo de datos ordena la activación de los actores, basado en las relaciones entre ellos. Ya que cada relación representa el flujo de datos, el

actor produciendo datos debe activar antes el actor consumidor. Convirtiendo estas dependencias de datos a una lista secuencial de actores planeados adecuadamente es equivalente a clasificar topologicamente un gráfico SDF, si el gráfico es aciclico. Los gráficos de flujos de datos con ciclos causan un poco de problemas, ya que tales gráficos no pueden ser clasificados topologicamente. A fin de determinar cual actor del ciclo se activa primero, un retardo debe ser explícitamente insertado en alguna parte del ciclo. Este retardo es representado por una señal inicial en alguna relación en el ciclo. La presencia del retardo permite al planeador romper el ciclo de la dependencia y determinar cual actor en el ciclo se activa primero. Los gráficos cíclicos no son anotados correctamente con retardos que no pueden ser ejecutados bajo SDF. Un ejemplo de un grafico cíclico propiamente anotado con un retardo es mostrado en la figura 3.9.

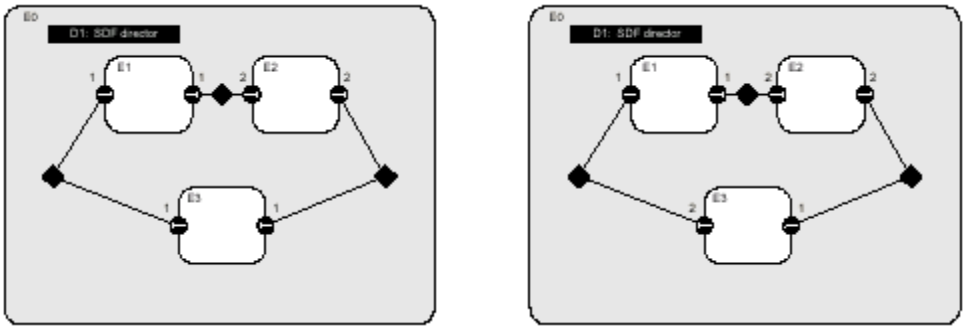


**Figura 3.8.**

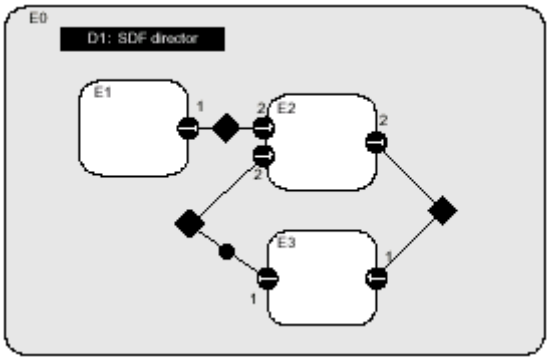


**Figura 3.9.**

### A.1.3.4. ARQUITECTURA DEL SOFTWARE DEL DOMINIO SDF

El paquete del kernel SDF implementa el modelo de computación SDF. La estructura de las clases en este paquete es mostrada en la figura 3.10.

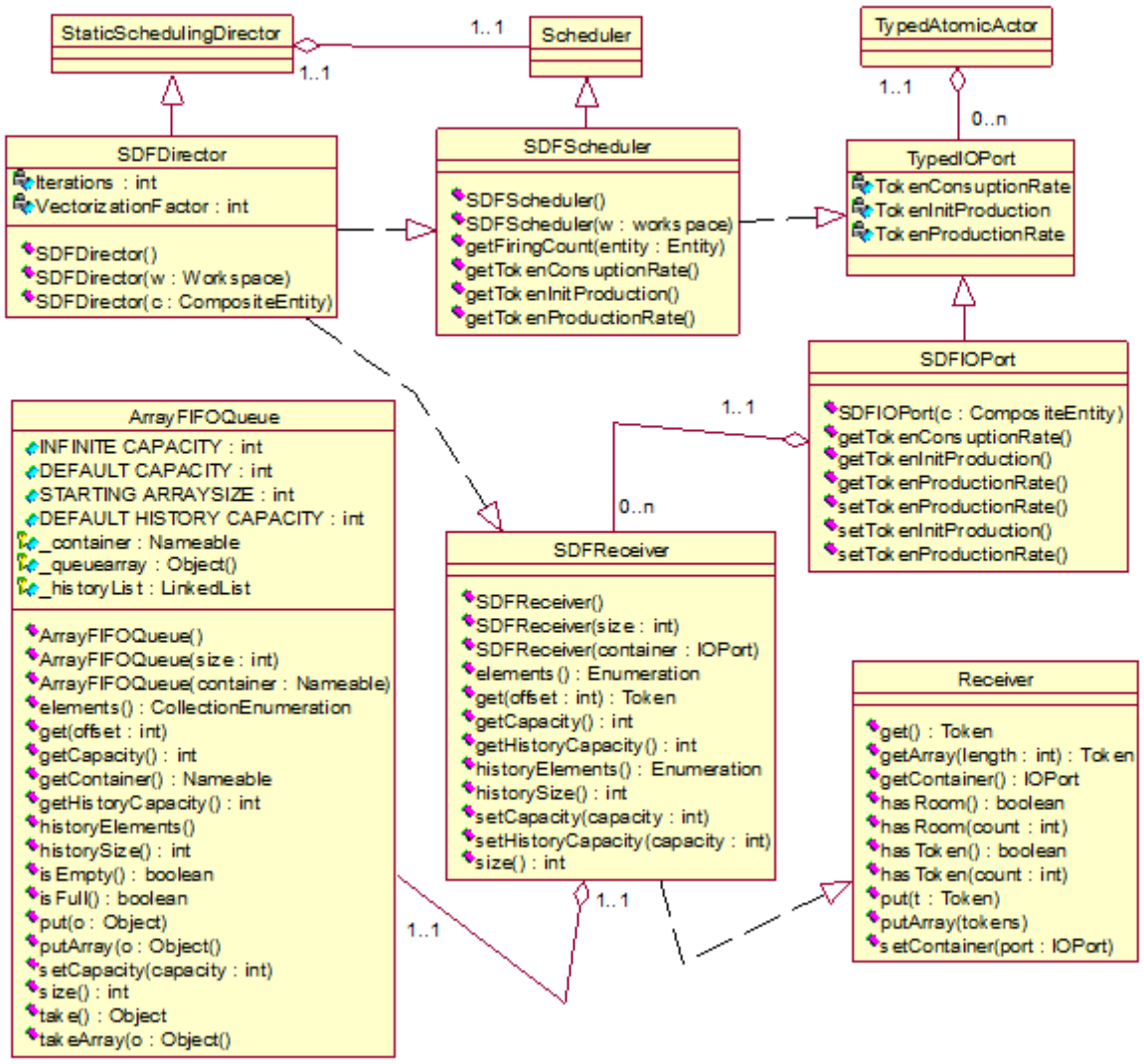


**Figura 3.10.**

### A.1.3.4.1. DIRECTOR SDF

La clase SDFDirector extiende la clase StaticSchedulingDirector. Cuando un director SDF es creado, es automáticamente asociado con una instancia de la clase scheduler por defecto, SDFScheduler. Este planeador (scheduler) es pensado para ser relativamente rápido y valido, pero no optimo en todas las situaciones. Como tal, futuros desarrollos probablemente resultarán en un amplio rango de planeadores con diferentes objetivos de desempeño y trade-off. El planeador SDF no restringe los planeadores que actualmente podrían ser usados con él.

El director tiene un solo parámetro, iterations, el cual determina un límite en el número de veces que el director desea ser activado. Después que el actor ha sido quemado el número de veces dado, siempre retornará falso en su método postfire(), indicando que no desea ser quemado de nuevo. Este parámetro debe contener un valor entero no negativo. El valor por defecto es un IntToken con valor 0, indicando que no hay un límite prefijado para el número de veces que el director se activará. Los usuarios probablemente especificarán un valor no cero para el número de iteraciones para el actor compuesto de nivel superior. Cuando es usado de esta forma, este parámetro actúa similarmente al parámetro Time-to-Stop en Ptolemy clásico.

El método newReceiver() en los directores SDF es sobrecargado para retornar instancias de la clase SDFReceiver. Este receptor contiene un método optimizado para leer y escribir bloques de señales. Para más información acerca de los receptores de SDF y los métodos extras que ellos soportan.

### A.1.3.4.2. PLANEADOR SDF

El básico SDFScheduler se deriva directamente de la clase Scheduler. Este planeador no proporciona ciclos, planeaciones secuenciales apropiadas para el uso en un solo procesador. No hace ningún esfuerzo por optimizar la planeación minimizando el tamaño del buffer de datos, minimizando el tamaño de la planeación, o detectando el paralelismo para permitir la ejecución en múltiples procesadores.

El algoritmo de planeación esta basado en los simples algoritmos multitasa en [47]. Actualmente, únicamente la planeación en un solo procesador es soportada. El algoritmo de planeación multitasa confía en los actores en el sistema declarando las tasas de datos en cada puerto. Si las tasas no están declaradas, entonces el planeador asume que el actor es homogéneo, significando que consume exactamente una señal de cada puerto de entrada y produce exactamente una señal en cada puerto de salida.

Las tasas de datos en los puertos son especificadas usando tres parámetros: tokenConsumptionRate, tokenProductionRate y tokenInitProduction. Los parámetros de producción son validos únicamente para puertos de salida, mientras que el parámetro de consumo es valido únicamente para puertos de entrada. Si un parámetro existe y no es valido para un puerto dado, entonces el valor del parámetro debe ser cero, o el planeador lanzará una excepción. Si un parámetro valido no es especificado cuando un planeador corre, entonces los valores apropiados de los parámetros serán asumidos 1, sin embargo los parámetros no son creados.

En Ptolemy Clásico, los modelos SDF jerárquicos estaban generalmente aplastados antes de la planeación. Esta técnica permitía la más eficiente planeación para ser construida por un modelo, y evitaba ciertos problemas de composability. En Ptolemy II, este algoritmo puede ser duplicado usando actores

compuestos transparentes para definir la jerarquía. Sin embargo, Ptolemy II también soporta una versión más fuerte de jerarquía, en la forma de actores compuestos opacos. En este caso, el planeador necesita hacer un poco más de trabajo. Antes de la planeación de un gráfico que contiene actores compuestos opacos, el planeador pregunta a cada actor compuesto opaco contenido que podría contener otro planeador y llamar a schedule() en ese planeador. El planeador SDF también crea un grupo de parámetros de tasa apropiados en cualquiera de los puertos que encuentra que están contenidos en el contenedor de su director.

Gráficos Desconectados (Disconnected graphs). Los gráficos SDF generalmente deben ser conectados fuertemente. Si un gráfico SDF no esta conectado fuertemente, entonces hay alguna concurrencia entre las partes desconectadas que no esta capturada por los parámetros de tasa SDF. En tales casos, otro modelo de computación (como redes de procesos) debe ser usado para especificar explícitamente la concurrencia. Como tal, el actual planeador SDF desaprueba los gráficos desconectados, y lanzará una excepción si usted intenta planear tal gráfico. Sin embargo, algunas veces es útil evitar introducir otro modelo de computación, ya que es probable que un futuro planeador permitirá gráficos desconectados con una noción de concurrencia por defecto.

Multipuertos (Multiports). Note que es imposible colocar un parámetro de tasa en canales individuales de un puerto. Esto es intencional, y todos los canales de un actor son asumidos para tener la misma tasa. Por ejemplo, cuando el actor AddSubtract activa bajo SDF, consumirá exactamente una señal de cada canal de su puerto plus de entrada, consume una señal de cada canal de su puerto minus, y produce una señal de cada canal de su puerto de salida. Note que aunque el sumador polimorfico en dominio es escrito para ser más general que este (consumirá hasta una señal en cada canal del puerto de entrada), el planeador SDF asegurará que siempre hay al menos una señal en cada puerto de entrada antes de las activaciones del actor.

Puertos Colgantes (Dangling ports). Los puertos deben, en general, ser conectados bajo el dominio SDF. Un puerto regular que no esta conectado no puede ser cumplido por el planeador SDF y el planeador lanzará siempre una excepción. El planeador SDF también detecta multipuertos que no están conectados a nada (y así tienen ancho cero). Tales puertos son interpretados para no existir actualmente, y son legales bajo SDF. El planeador ignorará la presencia de un multipuerto desconectado.

### A.1.3.4.3. PUERTOS Y RECEPTORES SDF

A diferencia de la mayoría de los dominios, los sistemas SDF multitasa tienden a producir y consumir grandes bloques de señales durante cada activación. Ya que puede haber un encabezado significante en el transporte de datos para estos grandes bloques, los puertos y receptores SDF tienen métodos optimizados para el envío y recepción de un bloque de señales en masa.

La clase SDFReceiver implementa la interfaz Receiver. En lugar de utilizar la clase FIFOQueue para almacenar datos, la cual es basada en una estructura de lista enlazada, los receptores SDF utilizan la clase ArrayFIFOQueue, la cual esta basada en un buffer circular. Esta elección es mucho más apropiada para SDF, ya que el tamaño del buffer es limitado, y puede ser determinado estáticamente. Los buffers circulares también tienen menos memoria y encabezado de asignación de objetos para una cola de un tamaño dado.

Aparte de los métodos normales de los receptores, la SDFReceiver proporciona los métodos sendArray() y getArray(). Estos dos métodos operan en arreglos de señales idénticamente a la forma en que los métodos send y get operan en señales individuales. Llamar al método sendArray() en un arreglo de señales es

equivalente a llamar al método send() en cada elemento del arreglo, únicamente es más rápido.

La clase SDFIOPort extiende la clase TypedIOPort. Adiciona dos métodos, sendArray() y getArray(). Si el puerto remoto contiene un receptor SDF, entonces el método sendArray() del receptor será usado en lugar del método send(). El método getArray() opera similarmente. Actualmente, los puertos SDF no soportan operaciones de bloque en las señales de la historia de los puertos.

#### A.1.3.4.4. ArrayFIFOQueue

La clase ArrayFIFOQueue implementa una cola FIFO (primeros que entran, primeros que salen) por medio de un buffer de arreglo circular. Funcionalmente es muy similar a la clase FIFOQueue. Proporciona una historia de la señal y un ajustable, posiblemente infinito, límite en el número de señales que contiene.

Si el límite en el tamaño es finito, entonces el arreglo es exactamente del tamaño del límite. En otras palabras, la cola esta llena cuando el arreglo se llena. Sin embargo, si el límite es infinito, entonces semejante arreglo no puede ser creado! En este caso, al buffer circular le es dado un tamaño de empezar pequeño, pero que se permite crecer. Siempre que el buffer circular se llene, es copiado en nuevo buffer que es dos veces del tamaño original.

#### A.1.3.4.5. SDFAtomicActor

La clase SDFAtomicActor extiende la clase TypedAtomicActor. Esta existe principalmente por conveniencia cuando se crean actores en el dominio SDF. Anula el método newPort() para crear puertos SDF. También proporciona métodos para colocar y acceder los parámetros de tasa en los puertos de los actores.

## A.1.4. DOMINIO CSP

### A.1.4.1. INTRODUCCION

Las aplicaciones para el dominio de CSP incluyen manejo del recurso y modelamiento primitivo del sistema de alto nivel en el ciclo de diseño. La gestión de recurso se requiere a menudo al modelar sistemas empotrados, y para apoyar mas esto, una noción de tiempo se ha agregado al modelo de cómputo usado en el dominio. Esto diferencia nuestro modelo CSP de otros más comúnmente encontrados, qué no tiene cualquier noción de tiempo típicamente, aunque se han propuesto varias versiones de CSP de tiempo. Podría ser así más exacto referirse al dominio que usa a nuestro modelo de cómputo como el dominio “Cronometró CSP”, pero ya que puede usarse con y sin tiempo, está simplemente llamado dominio CSP.

### A.1.4.2. USANDO EL DOMINIO CSP

Hay dos aspectos importantes que deben tenerse en cuenta al usar el dominio CSP:

- Incondicionales vs. rendezvous condicional
- Tiempo

### A.1.4.2.1. INCONDICIONAL VS. RENDEZVOUS CONDICIONAL

Las declaraciones de comunicación básicas send() y get() corresponda a rendezvous de comunicación en el dominio de CSP. Debido a la estructura del dominio, el hecho que un rendezvous está ocurriendo en cada comunicación es transparente al código del actor. Sin embargo, este rendezvous es incondicional; un actor sólo puede intentar la comunicación en un puerto en un momento. Para comprender el todas las potencialidades del dominio CSP que permite un rendezvous no determinística, es necesario escribir actores habituales que usan constructores de comunicación condicional en la clase básica CSPActor. Hay tres pasos involucrados:

1. Crear una sección ConditionalReceiver o ConditionalSend para cada declaración de comunicación protegida dependiendo de la comunicación. Convertir cada sección en un identificador entero único, iniciando de cero cuando se crea.

2. Convertir las secciones al método chooseBranch() en el CSPActor. Este método evalúa las protecciones, y decida cual sección obtiene un rendezvous, desarrolla el rendezvous y retorna el numero de identificación de la sección que tuvo éxito. Si todas las protecciones son falsas, retorna un –1.

3. Ejecute las declaraciones para la comunicación protegida que tuvo éxito.

Una muestra de la plantilla para ejecutar una comunicación condicional se muestra en figura 4.1. Creando las secciones ConditionalSend y ConditionalReceive, el primer argumento representa al guardia. El segundo y tercer argumentos representan el puerto y el canal para enviar o recibir el mensaje. El cuarto argumento es el identificador asignado a la sección. La opción de poner al guardia en el constructor fue hecha manteniendo la sintaxis de usar declaraciones de comunicación defendidas al mínimo, y al tener las clases de la sección parecidas a

las declaraciones de comunicación protegidas que ellos representan tan estrechamente como posible. Esto puede dar lugar al caso donde el Token específico en una sección de ConditionalSend no puede existir todavía, pero esto no tiene efecto porque una vez el guardia es falso, la señal en un ConditionalSend nunca es referenciada.

El código para usar un CIF es similar al de la figura 4.1 excepto que el entorno mientras que el lazo es omitido y el caso cuando el identificador retornado es –1 no hace nada. En el mismo escenario los pasos involucrados usando un CIF o un CDO pueden ser automatizados utilizando un preanalizador, peor por ahora el usuario debe seguir la aproximación descrita abajo.

```
boolean continueCDO = true;
while (continueCDO) {
   // step 1:
   ConditionalBranch[] branches = new ConditionalBranch[#branchesRequired];
   // Create a ConditionalReceive or a ConditionalSend for each branch
   // e.g. branches[0] = new ConditionalReceive((guard), input, 0, 0);

   // step 2:
   int result = chooseBranch(branches);

   // step 3:
   if (result == 0) {
      // execute statements associated with first branch
   } else if (result == 1) {
      // execute statements associated with second branch.
   } else if ... // continue for each branch ID

   } else if (result == -1) {
      // all guards were false so exit CDO.
      continueCDO = false;
   } else {
      // error
   }
}
```

**Figura 4.1.**

La figura 4.2 muestra algún código real basado en la plantilla sobre eso lleva a cabo un proceso almacenado. Este proceso repetidamente citado en sus puertos de entrada y salida, almacenando los datos si el proceso de lectura no está todavía listo para el proceso de escritura. Merece la pena que si la mayoría de los canales en un modelo son almacenados de esta manera, puede ser más

razonable crear al modelo en el dominio PN que implícitamente tiene un buffer ilimitado en cada canal.

### A.1.4.2.2. TIEMPO

El dominio CSP no usa actualmente el mecanismo fireAt() para modelar tiempo. Si un actor desea ser retardado una cierta cantidad de tiempo durante la ejecución del modelo, debe derivar de CSPActor. Cada proceso en el dominio CSP puede retardarse, o para algún periodo de tiempo del modelo actual o hasta el próximo bloqueo de tiempo se alcanza en el tiempo de modelo actual. Los dos métodos para llamar son delay() y waitForDeadlock (). Si un proceso se tarda por cero tiempo del tiempo actual, el proceso continuará inmediatamente. Así delay(0.0) no es equivalente a waitForDeadlock ().

```
boolean guard = false;
boolean continueCDO = true;
ConditionalBranch[] branches = new ConditionalBranch[2];
while (continueCDO) {
   // step 1
   guard = (_size < depth);
   branches[0] = new ConditionalReceive(guard, input, 0, 0);
   guard = (_size > 0);
   branches[1] = new ConditionalSend(guard, output, 0, 1, _buffer[_readFrom]);

   // step 2
   int successfulBranch = chooseBranch(branches);

   // step 3
   if (successfulBranch == 0) {
      _size++;
      _buffer[_writeTo] = branches[0].getToken();
      _writeTo = ++_writeTo % depth;
   } else if (successfulBranch == 1) {
      _size--;
      _readFrom = ++_readFrom % depth;
   } else if (successfulBranch == -1) {
      // all guards false so exit CDO
      // Note this cannot happen in this case
      continueCDO = false;
   } else {
      throw new TerminateProcessException(getName() + ": " +
            "branch id returned during execution of CDO.");
   }
}
```

**Figura 4.2.**

En lo que respecta a cada proceso, el tiempo sólo puede aumentar mientras que se bloquea esperando el rendezvous o cuando se retarda. Un proceso puede ser consciente del tiempo del modelo actual, pero sólo debe afectar el tiempo del modelo retardando su ejecución, así forzando al tiempo a avanzar. El método setCurrentTime() nunca debe llamarse desde un proceso. Sin embargo, si ningún proceso se retarda, es posible poner el tiempo del modelo llamando al método setCurrentTime () del director. Sin embargo, este método sólo está presente para componer CSP con otros dominios.

Por defecto, cada modelo en el dominio CSP esta cronometrado. Para usar CSP sin una noción de tiempo, simplemente, no use el método delay(). La infraestructura que soporta tiempo no afecta la ejecución del modelo si este método no se usa.

### A.1.4.3. PROPIEDADES DEL DOMINIO CSP

En el núcleo de la semántica de comunicación CSP están dos ideas fundamentales. Primero es la noción de comunicación atómica y segundo es la noción de opción no determinística. Merece la pena un modelo relacionado de cómputo conocido como el cálculo de comunicación de sistemas (C.C.P.) que fue desarrollado independientemente por Robin Milner en 1980 [65]. La semántica de comunicación de CSP es idéntica a aquélla de C.C.P.

#### A.1.4.3.1. COMUNICACION ATOMICA: RENDEZVOUS

La comunicación atómica se lleva a cabo a través de rendezvous e implica que el envío y recepción de un mensaje ocurre simultáneamente. Durante el rendezvous tanto el bloque del proceso de envío como el de recepción hasta el otro lado está listo para comunicar; el acto de enviar y recibir son actividades indistinguibles ya

que el uno no puede pasar sin el otro. Una analogía del mundo real para reunirse puede encontrarse en comunicaciones telefónicas. Tanto el llamante como el llamado deben estar simultáneamente presentes para que ocurra una conversación telefónica. La figura 4.3 muestra el caso donde un proceso está listo a enviar antes de que el otro proceso esté listo para recibir. La comunicación de información de esta manera puede verse como una declaración de asignación distribuida.

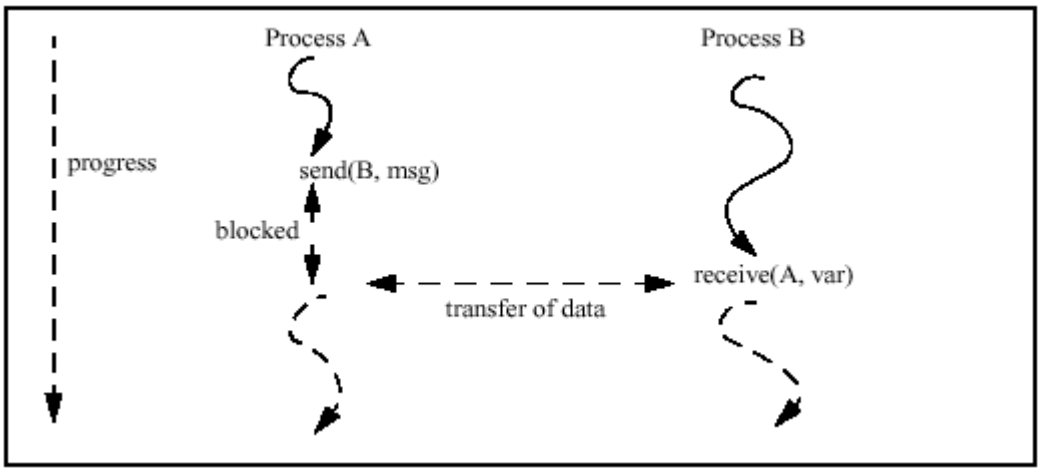


**Figura 4.3.**

El proceso de envio pone algunos datos en el mensaje que quiere enviar. El proceso receptor asigna los datos en el mensaje a una variable local. Por supuesto, el proceso receptor puede decidir ignorar los contenidos del mensaje y sólo interesarse en el hecho que un mensaje llegó.

### A.1.4.3.2. ELECCIÓN: RENDEZVOUS NO DETERMINISTA

La opción no deterministica proporciona la habilidad de seleccionar al azar entre un conjunto de posibles comunicaciones atómicas. Esto se refiere a la habilidad como rendezvous no determinística. El dominio CSP lleva a cabo rendezvous no

determinísticas a través de declaraciones de comunicación protegidas. Una declaración de comunicación protegida tiene la forma

guard; communication => statements;

El guardia sólo se permite al referenciar variables locales, y su evaluación no puede cambiar el estado del proceso. Por ejemplo no se permite asignar a las variables, sólo referenciarlas. La comunicación debe ser un simple envío o recepción, es decir otra declaración de comunicación condicional no puede ponerse aquí. Las declaraciones pueden contener cualquier sucesión arbitraria de declaraciones, incluyendo comunicaciones más condicionales.

Si el guardia es falso, entonces la comunicación no se intenta y las declaraciones no se ejecutan. Si el guardia es verdadero, entonces la comunicación se intenta, y si tiene éxito, las declaraciones siguientes se ejecutan. El guardia puede omitirse en que el caso se asuma que es verdadero.

Hay dos estructuras de comunicación condicionales construidas en las declaraciones de comunicación defendida: CIF y CDO. Éstos son análogos a las if y while en la mayoría que lenguajes de programación. Ellos deben leerse como “el condicional if” y “el condicional do”. Note que cada declaración de comunicación defendida representa una sección rama del CIF o CDO. La declaración de comunicación en cada sección puede ser un send o un receive, y pueden mezclarse libremente.

CIF: La forma de un CIF es

```
CIF    {
          G1; C1 => S1;
[ ]
          G2; C2 => S2;
```

```
[ ]
        ...
}
```

Para cada sección en el CIF, el guardia (G1, G2,...) se evalúa. Si es verdadero (o ausente, qué implica verdadero), entonces la declaración de comunicación asociada se habilita. Si una o más secciones son habilitadas, entonces la estructura entera bloquea hasta que una de las comunicaciones tiene éxito. Si más de uno sección se habilita, la opción desde la cual se habilitó la sección que tiene éxito con su comunicación se hace no determinística. Una vez la comunicación exitosa se lleva a cabo, las declaraciones asociadas se ejecutan y el proceso continúa. Si todos los guardias son falsos, entonces el proceso continúa ejecutando declaraciones después de la finalización del CIF.

Es importante notar que, aunque esta estructura es análoga a la estructura de programación común if, su conducta es muy diferente. En particular, se evalúan todos los guardias de las secciones concurrentemente, y la opción de la cuál tiene éxito no depende de su posición en la estructura. La notación “[ ]” se usa para indicar al paralelismo en la evaluación de los guardias. En un if común, las secciones se evalúan secuencialmente y la primera sección que se evalúa como verdadera se ejecuta. La estructura CIF también depende de la semántica de la comunicación entre procesos, y puede así parar el progreso del hilo si ninguna de las secciones habilitadas es capaz de reunirse.

CDO: La forma del CDO es

```
CDO  {
        G1; C1 => S1;
[ ]
        G2; C2 => S2;
[ ]
```

...

}

El comportamiento del CDO es similar al CIF en que para cada sección el 000rdia se evalúa y la opción de la cual permite la comunicación a hacer se toma no-determinísticamente. Sin embargo, el CDO repite el proceso de evaluar y ejecutar las secciones hasta que todos los guardias retornan falso. Cuando esto pasa el proceso continúa ejecutando declaraciones después de la estructura de CDO.

Un ejemplo del uso de un CDO está en un proceso buffer en que los dos pueden aceptar y pueden enviar mensajes, pero tiene que estar listo a hacer los dos en cualquier fase. El código para esto parecería similar a la figura 4.4. Note que en este caso ambos guardias nunca pueden ser simultáneamente falsos de manera que este proceso ejecutará el CDO por siempre.

```
CDO {
    (room in buffer?); receive(imput, beginningOfBuffer) => update pointer to beginning of buffer;
[]
    (messages in buffer?); send(output, endOfBuffer) => update pointer to end of buffer;
}
```

**Figura 4.4.**

### A.1.4.3.3. PUNTO MUERTO (Deadlock)

Una situación de bloqueo es aquella en la cual ninguno de los procesos pueden hacer progreso: ellos son todos bloqueados intentando juntarse o se retardan (vea la próxima sección). Así, pueden distinguirse dos tipos de bloqueo:

real deadlock – todos los procesos activos son bloqueados intentando comunicarse.

time deadlock – todos los procesos activos o son bloqueados intentando comunicarse o son retardados, y por lo menos un proceso es retardado.

### A.1.4.3.3. TIEMPO

En el dominio CSP, el tiempo esta centralizado. Esto es, todos los procesos en un modelo comparten el mismo tiempo, llamado el tiempo de modelo actual. Cada proceso sólo puede escoger retardarse por algún periodo relativo al tiempo de modelo actual, o un proceso puede esperar por el bloqueo de tiempo a ocurrir en el tiempo de modelo actual. Se dice que un proceso es retardado en ambos casos.

Cuando un proceso se retarda por algún espacio de tiempo desde el tiempo de modelo actual, este es suspendido mientras que él tiene un avance suficientemente. A que la fase se despierta y continua. Si los procesos se retrasan durante un tiempo cero, esto no tendrá efecto y el proceso continuará ejecutándose.

Un proceso también puede escoger tardar su ejecución hasta que alcance el próximo bloqueo de tiempo. El proceso reasume en el mismo tiempo de modelo al cual tardó, y esto es útil puesto que un modelo puede tener varias sucesiones de acciones en el mismo tiempo de modelo. La siguiente ocasión el tiempo de bloqueo es alcanzado, cualquier proceso retardado de esta manera continuará, y tiempo no se adelantará.

El tiempo puede adelantarse cuando todos los procesos se retardan o se bloquean intentando juntarse, y por lo menos un proceso es retardado. Si uno o más procesos están tardando hasta que ocurra un bloqueo de tiempo, estos procesos son despertados y el tiempo no está avanzado. Por otra parte, el tiempo de

modelo actual está avanzado sólo lo suficiente para despertar por lo menos un proceso. Note que hay una diferencia semántica entre un proceso retardado durante un tiempo cero, que no tendrá efecto y un proceso retardado hasta que se alcanza el próximo bloqueo de tiempo.

También note que el tiempo, percibido por un solo proceso, no puede cambiar durante su ejecución normal; sólo a los puntos del rendezvous o cuando los retrasos del proceso pueden cronometrar cambio. Un proceso puede ser consciente del tiempo centralizado, pero no puede influir en el tiempo de modelo actual excepto retardándose.

### A.1.4.4. ARQUITECTURA DEL SOFTWARE DEL DOMINIO CSP

#### A.1.4.4.1. ESTRUCTURA DE LAS CLASES

En un modelo CSP, el director es una instancia de CSPDirector. Ya que el modelo se controla por un CSPDirector, todos los receptores en los puertos son CSPReceivers. La combinación del CSPDirector y CSPReceivers en los puertos da una semántica de CSP ejemplar. El dominio CSP asocia cada canal con exactamente un receptor, localizado al extremo receptor del canal. Así cualquier proceso que envía o recibe a cualquier canal se juntará a un CSPReceiver.

**CSPDirector**: Este da una semántica de CSP ejemplar. Cuida del arranque de todos los procesos y control / respuesta tanto real como bloqueos de tiempo. También mantiene y adelanta el tiempo de modelo cuando necesario.

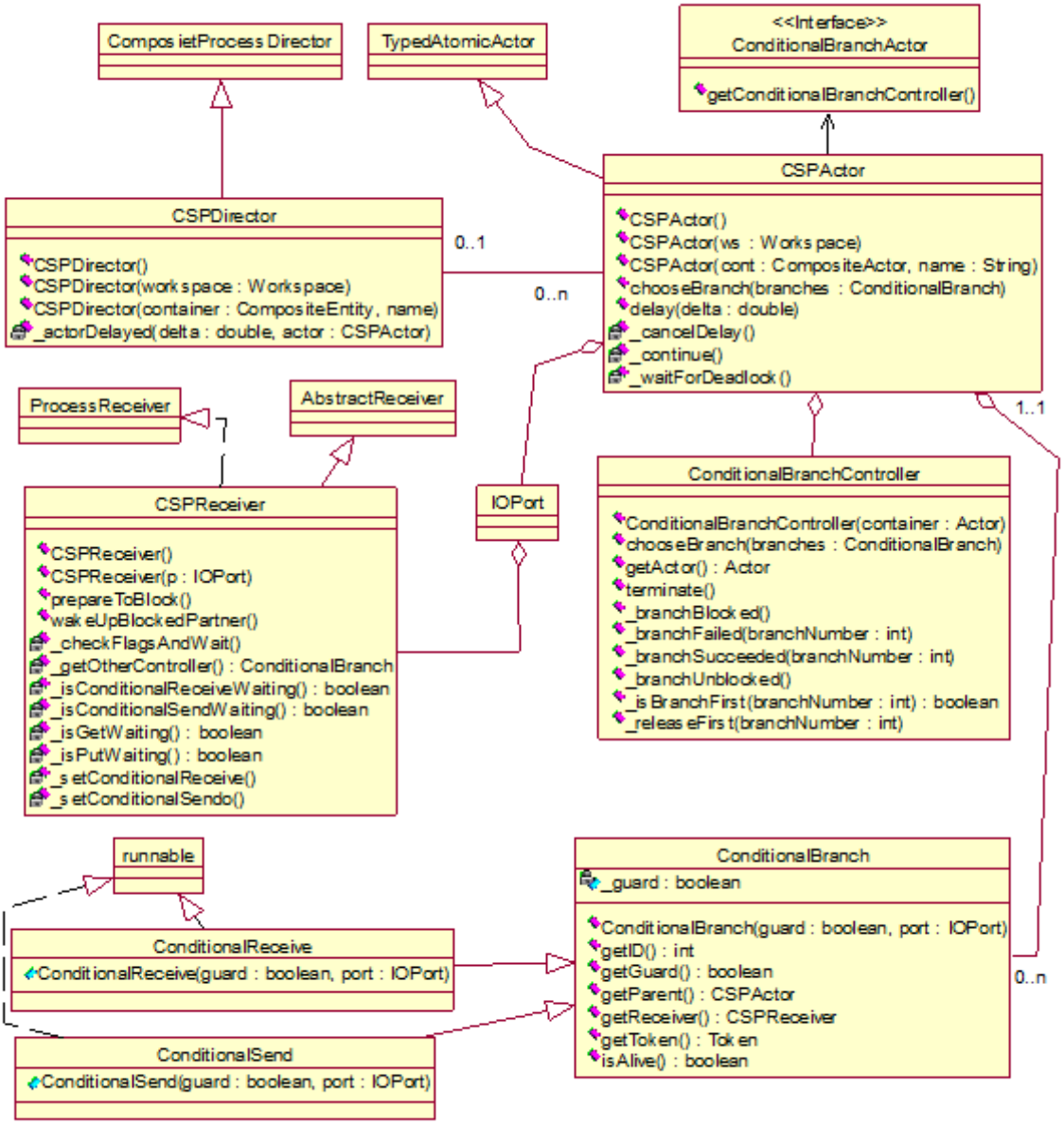


**Figura 4.5.**

**CSPReceiver**: Este asegura que la comunicación de mensajes entre los procesos sea por medio de rendezvous.

**CSPActor**: Este agrega la noción de tiempo y la habilidad para realizar comunicación condicional.

**ConditionalReceive, ConditionalSend**: Este se usa para construir la comunicación las declaraciones de comunicación protegida necesaria para las estructuras de comunicación condicionales.

### A.1.4.4.2. INICIANDO EL MODELO

El director crea un hilo para cada actor bajo su mando en su método inicialice(). Él también invoca el método inicialice() en cada actor en este momento. El director inicializa los hilos en su método prefire(), y detecta y responde a los bloqueos en su método fire(). El hilo para cada actor es una instancia de ProcessThread que invoca los métodos prefire(), fire() y postfire() para el actor hasta que termine o sea terminado. Invoca entonces el método wrapup() y los hilos terminan.

### A.1.4.4.3. DETECTANDO PUNTOS MUERTOS

Para descubrir puntos muertos, el director mantiene tres cuentas:

- El número de procesos activos que son hilos que han empezado pero no han terminado todavía.

- El número de procesos bloqueados que son el número de procesos que se bloquean esperando juntarse, y

- El número de procesos retardados que son el número de procesos que esperan por tiempo para adelantar más el número de procesos que esperan por el bloqueo de tiempo para ocurrir en el tiempo de modelo actual.

Cuando el número de procesos bloqueados iguala el número de procesos activos, el bloqueo real ha ocurrido y el método fire del director retorna. Cuando el número de bloqueados más el número de procesos retardados iguala el número de procesos activos, y por lo menos un proceso se retarda, entonces el bloqueo de tiempo ha ocurrido. Si por lo menos un proceso es retardado esperando por el bloqueo de tiempo para ocurrir en el tiempo de modelo actual, entonces el director despierta todos esos procesos y no adelanta tiempo. Por otra parte el director mira su lista de procesos que esperan por tiempo para adelantar, escoge el más temprano y adelanta suficientemente el tiempo para despertarlos. El director verifica para ver si hay un punto muerto en cada ocasión en que un proceso se bloquee, se retarde o termine.

Para que el director trabaje correctamente, estas tres cuentas necesitan estar en todas las fases del modelo de ejecución, para que cuando ellos se ponen al día se pone importante. Guardando la cuenta activa exacta es relativamente simple; el director la incrementa cuando empieza el hilo, y la disminuye cuando los termina el hilo. Igualmente la cuenta de procesos retardados es directa; cuando un proceso retarda, aumenta la cuenta de procesos retardados, y el director guarda registro de cuándo despertarlo. La cuenta se disminuye cuando se reanudan los procesos retardados.

Sin embargo, debido a la estructura de comunicación condicional, guardar la cuenta bloqueada requiere un poco más esfuerzo. Para un básico envío o recibo, un proceso es registrado como bloqueado cuando llega al punto de rendezvous antes de la comunicación se empareje. La cuenta bloqueada es entonces disminuida por uno cuando la comunicación correspondiente llega. ¿Sin embargo

qué pasa cuándo un actor está llevando a cabo una estructura de comunicación condicional? En este caso el proceso guarda registro de todas las secciones para las que los guardias eran verdaderos, y cuando todos ellos que intentan juntarse se bloquean, registra el proceso como bloqueado. Cuando uno de las secciones tiene éxito con un rendezvous, el proceso es registrado como desbloqueado.

### A.1.4.4.4. TERMINANDO UN MODELO

Un proceso puede terminar de dos maneras: volviendo falsos sus métodos prefire() o postfire() en este caso se dice que ha terminado normalmente, o terminado con anticipación por un TerminateProcessException. Por ejemplo, si un proceso fuente esta intentando enviar diez señales y terminar, terminaría su método fire() después de enviar la décima ficha, y retorna falso en su método postfire(). Esto causa el ProcessThread; vea la figura 4.6, representando el proceso, al terminar el bucle while y ejecuta la cláusula final. La cláusula final llama wrapup () en el actor que representa, disminuye la cuenta de procesos activos en el director, y el hilo que representa las terminaciones del proceso.

```
public void run() {
    try {
        boolean iterate = true;
        while (iterate) {
            // container is checked for null to detect the termination
            // of the actor.
            iterate = false;
            if ({Entity}_actor}.getContainer() != null && _actor.prefire()) {
                _actor.fire();
                iterate = _actor.postfire();
            }
        }
    } catch (TerminateProcessException t) {
        // Process was terminated early
    } catch (IllegalActionException e) {
        _manager.fireExecutionError(e);
    } finally {
        try {
            _actor.wrapup();
        } catch (IllegalActionException e) {
            _manager.fireExecutionError(e);
        }
        _director.decreaseActiveCount();
    }
}
```

**Figura 4.6.**

Una excepción TerminateProcessException se arroja siempre que un proceso intente comunicar a través de un canal cuyo receptor tiene su bandera finished fijada en verdadero. Cuando un TerminateProcessException es capturado en ProcessThread, la cláusula final también se ejecuta y el hilo que representa el fin del proceso.

Al terminar al modelo, el director fija la bandera terminada en cada receptor. La siguiente ocasión que un proceso intente enviar a o recibir del canal asociado con ese receptor, un TerminateProcessException se arroja. Este mecanismo también puede usarse en un modo selectivo para terminar con anticipación cualquier proceso que se comunica por medio de un canal particular. Cuando el director que controla la ejecución del modelo descubre un bloqueo real, retorna de su método fire(). En la ausencia de jerarquía, esto causa que el método wrapup() del director sea invocado. Este es el método wrapup() del director que fija la bandera terminada en cada receptor. Note que el TerminateProcessException es una excepción del tiempo de ejecución entonces no necesita ser declarada como arrojada.

Hay también la opción de terminar todos los procesos abruptamente en el modelo llamando terminate() en el director. Este método difiere del acercamiento descrito en el párrafo anterior en que detiene todos los hilos inmediatamente y no les da la oportunidad para poner al día el estado del modelo. Después de llamar este método el estado del modelo es desconocido y así el modelo debe recrearse después de llamar este método. Este método sólo se piensa para las situaciones cuando la ejecución del modelo ha salido obviamente mal; para finalizar normalmente tomarían seria mas largo o no podría pasar. Raramente debe llamarse.

### A.1.4.4.5. PAUSANDO / REINICIANDO EL MODELO

Pausar y reiniciar un modelo no afecta el resultado final de una ejecución particular del modelo, únicamente la tasa de progreso. La ejecución de un modelo puede ser pausada en cualquier etapa llamando el método pause() en el director. Este método está bloqueado, y únicamente se restituirá cuando la ejecución del modelo sea exitosamente pausada. Al pausar la ejecución de un modelo, el director pone una bandera paused en cada receptor, y la próxima ocasión un proceso intenta enviar o recibir desde el canal asociado con ese receptor, ha hecho una pausa. El modelo entero es pausado cuando todos los procesos activos son retardados, pausados o bloqueados. Para reasumir el modelo, el método resume() puede llamarse similarmente en el director. Este método restablece la bandera pause en cada receptor y despierta a cada proceso que espera una cerradura del receptor. Si un proceso fuera pausado, mira que ya no esta pausado y continúa. La habilidad de hacer una pausa y reasumir la ejecución de un modelo se piensa principalmente para el control de interface de usuario.

### A.1.4.5. EJEMPLOS DE APLICACIONES CSP

Se han desarrollado varias aplicaciones ejemplo qué sirven para ilustrar las capacidades de modelamiento del modelo de computación CSP Ptolemy II. Cada demostración incorpora varios rasgos de CSP y la estructura general de Ptolemy II. Las aplicaciones se describen aquí, pero no el código. Vea el directorio $PTII/ptolemy/domains/csp/demo ver para el código.

La primera demostración, dining philosophers, sirve como un ejemplo natural del núcleo de la semántica de comunicación CSP. Esta demostración modela contención de recursos no determinísticos, ejemplo, cinco filósofos accediendo aleatoriamente a los recursos del palillo. Los rendezvous nodeterminísticas sirven

como una herramienta de modelamiento natural para este ejemplo. El segundo ejemplo, hardware bus contention, modela la contención de recursos determinístico en el contexto de tiempo. Como se mostrará, el determinacy de esta demostración reprime el nondeterministico natural de la semántica de CSP y resulta en dificultades. Afortunadamente estas dificultades pueden ser suavizadas fácilmente por el modelo cronometrando que se ha integrado en el dominio CSP.

### A.1.4.5.1. DINING PHILODOPHERS ( Filósofos Cenando)

Contención de recursos no deterministica. Esta aplicación del problema de los filósofos cenando ilustra tanto el tiempo y la comunicación condicional en el dominio CSP. Cinco filósofos se sientan a un mesa con un cuenco grande de comida en el medio. Entre cada par de filósofos está un palillo, y a comer, un filósofo necesita los dos palillos al lado de él. Cada filósofo gasta su vida en el ciclo siguiente: piensa durante algún tiempo, se pone hambriento, escoge uno de los palillos al lado de él, entonces el otro, come durante algún tiempo y suelta el palillo de nuevo en la mesa. Si un filósofo intenta agarrar un palillo pero ya es esta siendo usando por otro filósofo, entonces el filósofo espera hasta ese palillo se pone disponible. Esto implica que ningún filósofo vecino puede comer al mismo tiempo y a lo sumo que dos filósofos pueden comer en un tiempo.

El problema de los Filósofos Cenando fue propuesto primero por Edsger W. Dijkstra en 1965. Es un clásico problema de la programación concurrente que ilustra las dos propiedades básicas de programación concurrente:

Liveness. Cómo se puede diseñar el programa para evitar bloqueo, donde ninguno de los filósofos puede progresar porque cada uno está esperando que alguien más haga algo?

Fairness. Cómo se puede diseñar el programa para evitar la inanición, donde uno de los filósofos pueda hacer progresos pero no los hace porque otros siempre van primero?

Esta aplicación usa un algoritmo que permite aleatoriamente a cada filósofo escoger qué palillo recoger primero (por medio de un CDO), y todos los filósofos comen y piensan en las mismas proporciones. Cada filósofo y cada palillo son representado por un proceso separado. Cada palillo tiene que estar listo para ser usado por cualquier filósofo al lado de él en cualquier momento, por lo tanto el uso de un CDO. Después de que se agarra, se bloquea esperando un mensaje del filósofo que está usándolo. Después un filósofo agarra ambos palillos al lado de él, y come durante un tiempo aleatorio. Esto es representado llamando delay() con el intervalo aleatorio para comer. El mismo acercamiento se usa cuando un filósofo está pensando.

Este algoritmo es aceptable, como en cualquier tiempo un palillo no está usándose, y ambos filósofos intentan usarlo, los dos tienen una oportunidad igual de tener éxito. Sin embargo este algoritmo no garantiza la ausencia de un punto muerto, y si se permite que se ejecute bastante tiempo este eventualmente ocurrirá. La probabilidad que el bloqueo ocurre pronto como los tiempos de pensar se disminuye relativo a los tiempos de comer.

### A.1.4.5.2. HARDWARE BUS CONNECTION

Contención de recursos Determinístico. Esta demostración consiste en un director, N procesadores y un bloque de memoria, como se muestra en la figura 4.7. A los puntos seleccionados aleatoriamente en tiempo, cada procesador pide permiso del director para acceder al bloque de memoria. Cada uno de los procesadores tiene prioridades asociadas con ellos y en casos donde hay una

demanda de acceso a memoria simultánea, el director concede permiso al procesador con la prioridad más alta. Debido a la naturaleza atómica de el rendezvous, es imposible para el controlador verificar prioridades de requerimientos entrantes al mismo tiempo que están ocurriendo demandas. Para superar esta dificultad, se emplea una alarma. La alarma es iniciada por el director inmediatamente después que el primer requerimiento de acceso a memoria en un momento dado. Se despierta cuando un bloque de retraso ocurre para indicar al director que no hay más demanda de memoria que ocurra a un punto dado en el tiempo. De esta forma, la alarma usa la noción de CSP de bloqueo de retraso para hacer deterministica una actividad inherentemente no-deterministica.

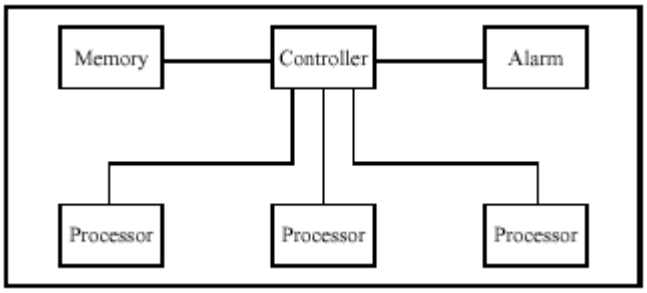


**Figura 4.7.**

## A.1.5. DOMINIO PN

### A.1.5.1. INTRODUCCION

El dominio redes de procesos (PN) en Ptolemy II modela un sistema como una red de procesos secuenciales, implementados como hilos de Java [65], que se comunican mediante el paso de mensajes a través de canales FIFO (los primeros en entrar son los primeros en salir) unidireccionales. Un proceso se boquea cuando trata de leer de un canal que esta vacío hasta que un mensaje se encuentre disponible en el canal. Este modelo de computación es determinístico en el sentido de que si los procesos son deterministicos y comunicados únicamente por vía de canales, entonces la secuencia de valores comunicados por los canales es completamente determinada por el modelo.

El dominio PN es un modelo natural para la descripción de sistemas procesadores de señales donde los flujos infinitos de muestras de datos son transformados incrementalmente por una colección de procesos ejecutándose en paralelo. Los sistemas empotrados de procesamiento de señales están típicamente diseñados para operar indefinidamente con recursos limitados. Así, se quiere ejecutar programas de redes de procesos siempre con almacenamiento definido en los canales de comunicación siempre que sea posible.

El dominio PN también puede ser usado para modelar la concurrencia en los diferentes componentes hardware de un sistema empotrado. El modelo de computación original de redes de procesos puede modelar el comportamiento funcional de estos sistemas y probarlos para su correcto funcionamiento, pero no

puede modelar directamente su comportamiento en tiempo real. Para ver esto, el dominio PN extiende el modelo introduciendo una noción de tiempo.

Además, algunos sistemas podrían presentar comportamientos adaptables como códigos de migración, agentes, y llegadas y salidas de procesos. Para esto, se provee un mecanismo de mutación que soporta adición, supresión, y cambio de procesos y canales. Con PN sin tiempo, esto podría desplegarse no deterministico, mientras que con PN con tiempo, este se convierte en deterministico.

El modelo de computación PN es un super grupo del modelo de computación flujo de datos sincrónicos (ver el dominio SDF). Así cualquier actor del dominio SDF puede ser usado en el dominio PN. Similarmente cualquier actor polimorfico en dominio puede ser usado en el dominio PN. Un proceso separado es creado para cada uno de estos actores. Estos procesos son implementados como hilos de Java.

### A.1.5.2. USANDO EL DOMINIO PN

Hay dos aspectos importantes ha ser tenidos en cuenta con el dominio PN:

- Puntos muertos en ciclos realimentados.
- Diseño de actores.

#### A.1.5.2.1. PUNTOS MUERTOS EN CICLOS REALIMENTADOS

Los ciclos realimentados deben ser manejados en gran medida de la misma forma como en los actores SDF. Uno de los actores en el ciclo realimentado debe crear un número de señales en su ciclo realimentado para quebrar la dependencia de

datos. Así como en el dominio SDF, el actor SampleDelay puede ser usado para este propósito. Recuerde, sin embargo que el dominio PN no (y no puede) analiza estáticamente el modelo para determinar el tamaño del retardo necesario en el ciclo realimentado. Es responsabilidad del diseñador del modelo especificar la cantidad correcta del retardo.

#### A.1.5.2.2. DISEÑO DE ACTORES

Debido a la manera en que el dominio PN es implementado, no es posible para un actor comprobar si hay un dato presente en un puerto de entrada. El método hasToken() siempre retorna verdadero indicando que una señal esta presente, y si una señal no esta presente, entonces el método get() se bloqueará hasta que haya alguna disponible. Esto permite a los modelos ejecutarse deterministicamente. Sin embargo, los actores que toman entradas de más de una entrada generalmente son difíciles de escribir. La forma común de crear tal actor es similar a la forma en que trabaja el actor Select. Otra entrada es leída primero, y los datos de ese puerto determinan de cual puerto de entrada hay que leer.

### A.1.5.3. PROPIEDADES DEL DOMINIO PN

Dos propiedades importantes del dominio PN implementadas en Ptolemy II son que los procesos se comunican de manera asincrónica (por colas ordenadas) y que la memoria usada en la comunicación es limitada. El dominio PN en Ptolemy II puede ser usado con o sin una noción de tiempo.

### A.1.5.3.1. COMUNICACION ASINCRONICA

Kahn y MacQueen [40][41] describen un modelo de computación donde los procesos son conectados por los canales de comunicación que forman una red. Los procesos producen elementos de datos o señales (tokens) y los envían a lo largo de un canal de comunicación unidireccional donde son almacenados en una cola FIFO hasta que el proceso de destino los consuma. Esta es una forma de comunicación asincrónica entre procesos. Los canales de comunicación son el único método en que los procesos pueden hacer uso del intercambio de información. Un grupo de procesos que se comunican a través de una red de colas FIFO define un programa.

Kahn y MacQueen requieren que la ejecución de un proceso este suspendida cuando este intente extraer de un canal de entrada vacío (bloqueo de lectura). Un proceso podría no censar el canal para la presencia o ausencia de datos. En cualquier punto dado, un proceso o hace alguna computación (habilitado) o esta bloqueado esperando datos (bloqueo de lectura) en exactamente uno de sus canales de entrada; no puede esperar datos de más de un canal simultáneamente. Los sistemas que obedecen este modelo están determinados; la historia de las señales producidas por los canales de comunicación no dependen de la orden de ejecución. Por consiguiente, los resultados producidos por la ejecución de un programa no están afectados por la planeación de los diferentes procesos.

En el caso de que todos los procesos en un modelo estén bloqueados mientras intentan leer de algún canal, se tiene un real deadlock. En tal caso, ninguno de los procesos puede hacer nada. La determinancia del programa también garantiza que un real deadlock es un estado del programa y no depende de la planeación de los diferentes procesos en el modelo.

### A.1.5.3.2. EJECUCION LIMITADA DE MEMORIA

El alto nivel de concurrencia que presentan las redes de procesos las hace ideales para el software de sistemas empotrados y para el modelado de implementaciones hardware. Pero la semántica de Kahn-MacQueen no garantiza la ejecución definida de memoria de las redes de procesos aún si es posible para la aplicación ejecutar en memoria definida. La mayoría de las aplicaciones empotradas y procesos de hardware son hechos para ejecutarse indefinidamente con una cantidad limitada de memoria.

Parks [68] se ocupa de este aspecto de las redes de procesos y proporciona un algoritmo para hacer una aplicación de redes de procesos ejecutable en memoria definida siempre que sea posible. El proporciona una implementación de la semántica de Kahn-MacQueen usando bloqueos de escritura (blocking writes) que asignan una capacidad fija para cada canal FIFO y obliga a los procesos a bloquearse temporalmente si un canal esta lleno. Para evitar introducir deadlocks (puntos muertos), estas capacidades son aumentadas si es absolutamente necesario. Un proceso puede no censar el canal para alojarse. Así un proceso tiene tres estados: ejecutándose (running), bloqueado para lectura (read blocked), o bloqueado para escritura (write blocked).

Los deadlock pueden ocurrir cuando todos los procesos están bloqueados ya sea escribiendo o leyendo a/de un canal. En el caso de que todos los procesos en un modelo estén bloqueados con al menos un proceso bloqueado en escritura, entonces se tiene un deadlock artificial. En la detección de un deadlock artificial, Parks escogió el canal con la menor capacidad entre los canales cuyos procesos estaban bloqueados en escritura e incremento su capacidad para romper el deadlock.

### A.1.5.3.3. TIEMPO

En algunos sistemas de tiempo real y aplicaciones empotradas, el comportamiento de tiempo real de un sistema es tan importante como la correcta funcionalidad. Los desarrolladores pueden usar redes de procesos en aplicaciones para probar la correcta funcionalidad y usar algún otro modelo de computación de tiempo, como el DE, para probar su comportamiento en el tiempo. Introducir una noción de tiempo al modelo de computación redes de procesos es por consiguiente una extensión natural de PN. Esto esta hecho en el dominio PN en Ptolemy II.

En el dominio PN el tiempo es global. Esto es, todos los procesos en un modelo comparten el mismo tiempo, conocido como tiempo actual o tiempo modelo. Un proceso puede esperar explícitamente la hora de seguir. Este puede elegir retardarse a sí mismo por algún periodo del tiempo actual. Cuando un proceso se retarda a sí mismo por alguna longitud de tiempo del tiempo actual, este es suspendido hasta que haya avanzado suficientemente el tiempo, en el cual la etapa se despierta y continua. Si el proceso se retarda a si mismo al tiempo cero, esto no tendrá efecto y el proceso continuara ejecutándose.

Para un proceso, el tiempo no puede cambiar durante su normal ejecución (i.e. cuando el proceso esta habilitado). El tiempo para un proceso podría avanzar en únicamente uno de los dos estados:

1. El proceso es retardado y esta explícitamente esperando el tiempo de seguir (bloqueo de retardo).

2. El proceso esta esperando datos de llegada en uno de sus canales de entrada (bloqueo de lectura).

Cuando todos los procesos en un programa están en uno de estos dos estados, entonces el programa esta en un estado de punto muerto de tiempo. El hecho de

que al menos un proceso este retardado, diferencia un punto muerto de tiempo de otro punto muerto. Cuando un punto muerto de tiempo es detectado, el tiempo actual es adelantado hasta que al menos un proceso pueda despertarse de un bloqueo de retardo y el modelo continua ejecutándose.

#### A.1.5.3.4. MUTACIONES

El dominio PN tolera las mutaciones, las cuales son cambios en tiempo de ejecución en la estructura del modelo. Normalmente, las mutaciones son realizadas como requerimientos de cambio en una cola con el director o administrador. En el caso de PN con tiempo, los requerimientos de mutación no son procesados hasta que haya un timed deadlock. Como la ocurrencia de un timed deadlock es determinado, las mutaciones en el PN con tiempo son determinadas.

En el caso del PN sin tiempo, no hay un punto determinado donde las mutaciones puedan ocurrir. El único punto determinado en el caso de un PN sin tiempo es en un deadlock de lectura. Realizar mutaciones en este punto es improbable porque un deadlock real podría nunca ocurrir. Un modelo con hasta una fuente sin terminación nunca experimenta un deadlock real. Así en caso de un PN sin tiempo, las mutaciones son realizadas tan pronto como son requeridas (si están en la cola con el director) y cuando un deadlock real ocurre (si ellas están en la cola del administrador). Ya que diferentes planeaciones resultarán en diferentes estados del modelo cuando las mutaciones son realizadas, el anterior introduce no-determinismo en el dominio PN sin tiempo.

### A.1.5.4. ARQUITECTURA DEL SOFTWARE DEL DOMINIO PN

El kernel del dominio PN esta contenido en el paquete ptolemy.domains.pn.kernel. el diagrama de estructura de paquetes es mostrado en la figura 5.1.

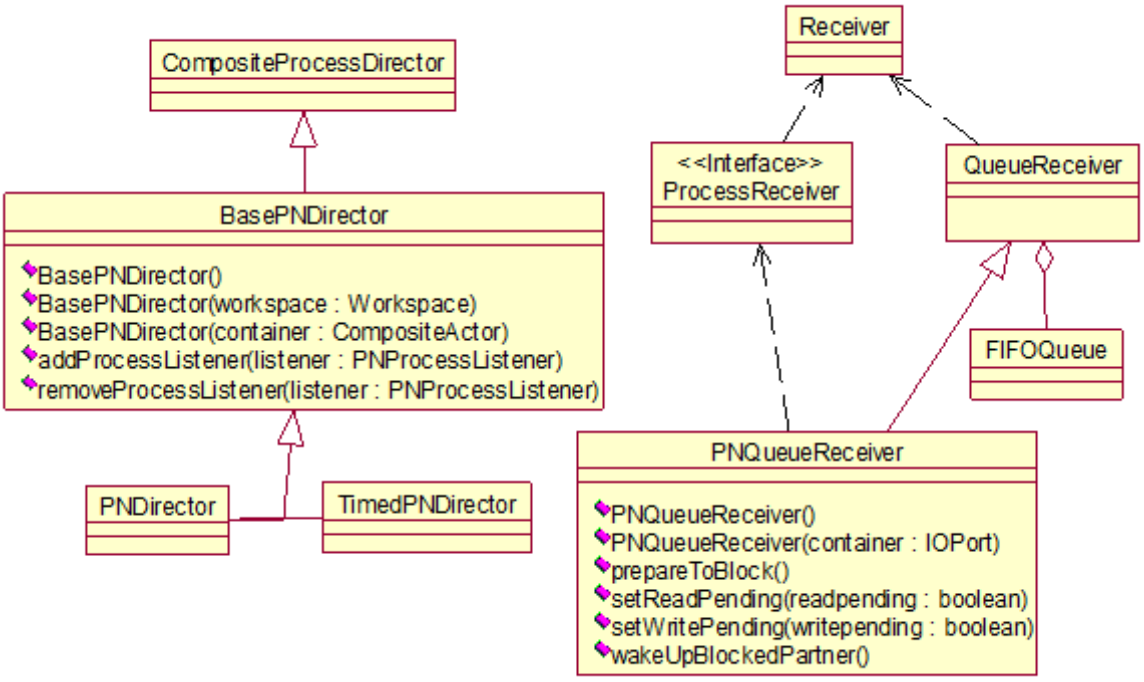


**Figura 5.1.**

#### A.1.5.4.1. BasePNDirector

Esta es la clase básica para los directores que gobiernan la ejecución de un CompositeActor con la semántica de redes de procesos (PN) de Kahn. Esta clase básica añade la semántica de redes de procesos de Kahn-MacQueen a un actor compuesto. Este director no soporta mutaciones o una noción de modelo de tiempo. Este proporciona un mecanismo para realizar bloqueos de lectura y escritura definida de memoria (usando bloqueos de escritura) siempre que sea posible. Así es capaz de manejar deadlocks reales y artificiales.

Este director es capaz de manejar puntos muertos reales y artificiales. Un punto muerto artificial es resuelto tan pronto como se origina usando el algoritmo de Park que fue explicado anteriormente. Un punto muerto real, sin embargo, no puede ser manejado localmente y debe depender del ambiente externo para proporcionar más datos para que la ejecución continúe.

#### A.1.5.4.2. PNDirector

El PNDirector extiende a BasePNDirector para manejar mutaciones localmente. Esto es únicamente una optimización, ya que permite a una mutación ejecutarse más rápido de lo que lo haría en otro caso, y no adiciona ninguna capacidad interesante al modelo. Lo más importante es que la mutación no es deterministica y puede pasar en cualquier punto durante la ejecución del modelo.

#### A.1.5.4.3. TimedPNDirector

El TimedPNDirector extiende a BasePNDirector y tiene dos funcionalidades que lo diferencian de este. Introduce una noción de tiempo global al modelo y permite mutaciones deterministicas. Las mutaciones son realizadas lo más pronto posible en que ocurre un timed deadlock después de que están en cola. Ya que la ocurrencia de un timed-deadlock es completamente deterministica, realizar mutaciones en este punto en hace que estas sean deterministicas.

#### A.1.5.4.4. PNQueueReceiver

La PNQueueReceiver implementa la interfaz ProcessReceiver y contiene una cola FIFO que representa un canal de comunicaciones de una red de procesos. Estos

receptores también son responsables de implementar los bloqueos de lectura y escritura a través de los métodos get() y put().

Cuando el método get() es llamado, el receptor verifica primero si una cola FIFO tiene alguna señal. Si no es así, entonces reporta al director esta bloqueado esperando datos. También pone una bandera interna para indicar que un hilo esta bloqueado en lectura. Entonces el hilo de lectura es suspendido hasta que algún otro hilo ponga una señal dentro de la cola FIFO. En este punto, la bandera del receptor es puesta en falso, el director es notificado que el proceso ha sido desbloqueado, el proceso de lectura retoma la primera señal de la cola FIFO y la ejecución continua.

El método put() de los receptores trabaja de una forma similar primero verificando si la cola FIFO esta llena. Si es así, reporta al director que un hilo de escritura esta bloqueado esperando un espacio en la cola. También pone una bandera interna para indicar que un hilo esta bloqueado en escritura. El hilo de escritura se bloquea hasta que algún otro hilo tome una señal de la cola FIFO, o que el tamaño de la cola sea incrementado por el director porque la red ha alcanzado un punto muerto artificial. En todo caso, el director es notificado que un proceso de escritura se desbloqueo y la bandera interna es puesta en falso. El hilo de escritura es despertado y su señal es colocada dentro del receptor.

### A.1.5.4.5. MANEJO DE PUNTOS MUERTOS

Cada vez que un actor en PN se bloquea, la cuenta de actores bloqueados se incrementa. Si el número total de actores bloqueados o pausados es igual al número total de actores activos en la simulación, un punto muerto es detectado. En la detección de un punto muerto, si uno o más actores están bloqueados en escritura, entonces este es un punto muerto artificial. El canal con la capacidad más pequeña entre todos los canales con actores bloqueados en escritura es

escogido y su capacidad es incrementada en 1. esto implementa la ejecución limitada de memoria. Si un punto muerto real es detectado, entonces el método FIRE del director retorna, permitiendo al modelo presentar más datos en las entradas de la red de procesos.

### A.1.5.4.6. ITERACIONES FINITAS

Un aspecto importante de Ptolemy II es que la activación (firing) de un actor, o de un modelo completo es garantizado para ser completo. En los dominios de procesos el final de una activación ocurre cuando se alcanza un punto muerto. El punto muerto puede ser real o de tiempo. Sin embargo, en una red de procesos un punto muerto real podría nunca ocurrir. En este caso, para para una ejecución manualmente o ejecutar mutaciones debe haber una forma de parar todos los hilos ejecutándose en la red. Esto es manejado por el método stopFire() de la interfaz ejecutable. El director de proceso implementa este método para poner una bandera en cada proceso los cuales causan que el proceso se detenga. Nótese que como con la mayoría de los dominios, no es posible simplemente llamar el método wrapup() del director de proceso, ya que el método FIRE no ha retornado aún.

## A.1.6. DOMINIO CT

### A.1.6.1. INTRODUCCION

El Dominio de tiempo continuo (Dominio CT) tiene como meta ayudar al diseño y simulación de sistemas que pueden ser modelados usando ecuaciones diferenciales ordinarias (ODE's). Las ODE's son usadas generalmente para modelar circuitos analógicos, plantas dinámicas en sistemas de control, conjuntos de parámetros en sistemas de carga y muchos otros sistemas físicos.

El uso de computadores digitales para simular sistemas de tiempo continuo ha sido estudiado por más de dos décadas. Una de las herramientas más conocidas es Spice. El dominio CT difiere de Spice en los simuladores de tiempo continuo en dos formas, la especificación del sistema es algo diferente, y esta diseñado para interactuar con otros modelos de computación.

### A.1.6.2. PROPIEDADES DEL DOMINIO CT

Las propiedades del dominio CT se describen a continuación.

### 6.2.1. TERMINOLOGIA BASICA

Como se dijo el dominio CT es utilizado para simular sistemas que han sido modelados usando ecuaciones diferenciales ordinarias (ODE's).

En general, una ODE basada en un sistema de tiempo continuo tiene la siguiente forma:

$$x' = f(x, u, t) \quad (1)$$
$$y = g(x, u, t) \quad (2)$$
$$x(t_0) = x_o \quad (3)$$

donde t pertenece a los números reales, $t \geq t_o$, y es el tiempo continuo. Y x' es la derivada de x con respecto al tiempo. Las ecuaciones 1, 2 y 3 se llaman dinámica del sistema, plano de salida y condición inicial del sistema.

El dominio CT es útil para modelar sistemas físicos que se pueden describir con ecuaciones algebraicas o diferenciales, tales como circuitos analógicos, circuitos de microondas y sistemas o componentes mecánicos, los cuales se encuentran generalmente en sistemas empotrados.

**A.1.6.2.2. TIEMPO**

La característica más importante del dominio CT es la continuidad del tiempo. Esto implica que un sistema continuo en el tiempo tiene un comportamiento en cualquier instancia de tiempo. La maquina de simulación del dominio CT es capaz de computar el comportamiento del sistema en cualquier punto del tiempo.

El tiempo en este modelo también es global, lo cual significa que todos los componentes en el sistema comparten la misma noción de tiempo.

### A.1.6.2.3. DISCONTINUIDAD

La existencia y unicidad de la solución de una ODE permite que la parte derecha de la ecuación 1 sea discontinua en un número contable de puntos discretos D, los cuales son llamados puntos de quiebre (breakpoints). Estos puntos de quiebre pueden ser causados por la discontinuidad de una señal u. En teoría, las soluciones en estos puntos no están bien definidas. Por lo tanto, para encontrar las soluciones en estos puntos, se debe intentar encontrar los límites por la derecha y la izquierda en donde estas ecuaciones están bien definidas. Así que en lugar de resolver las ODE en estos puntos, se esta intentado encontrar los límites izquierdo y derecho.

Un punto de quiebre podría ser conocido de antemano, en cuyo caso es llamado un punto de quiebre predecible (PBP). Por ejemplo, un actor fuente de onda cuadrada puede pedir su próximo tiempo de cruce. Esta información puede ser usada para controlar la discretización de tiempo. Un punto de quiebre también puede ser impredecible (UBP), lo cual quiere decir que no se conoce hasta que el tiempo ocurre. Por ejemplo, un actor que varia su funcionalidad cuando la señal de entrada cruza un umbral puede únicamente reportar un punto de quiebre “perdido” después de que un paso de integración es finalizado.

### A.1.6.2.4. ESPECIFICACION DEL SISTEMA

Hay usualmente dos maneras para especificar un sistema de tiempo continuo, el modelo de ley conservativa y el modelo de flujo de señal. El modelo de ley conservativa, también llamado análisis nodal en simulación de circuitos, define un sistema por sus componentes físicos, los cuales especifican relaciones de cruce y través de variables, y las leyes conservativas son usadas para compilar las relaciones componentes dentro de las ecuaciones de un sistema global. Por ejemplo, en simulaciones de circuitos, las variables de cruce son voltajes, las

variables directas son corrientes, y las leyes conservativas son las leyes de Kirchhoff. Este modelo refleja directamente los componentes físicos de un sistema, de esta manera es fácil construirlo de una implementación potencial. La forma matemática real del sistema es escondida. En el modelo de flujo de señales, las entidades en el sistema son mapas que definen la relación matemática entre las señales de entrada y salida. Las entidades se comunican pasando señales. Este modelo refleja directamente las relaciones matemáticas entre las señales, y es más conveniente para la especificación de sistemas que todavía no tienen una implementación explicita.

En el dominio CT de Ptolemy II, es escogido el modelo de flujo de señales como la semántica de iteración. La semántica de ley conservativa puede ser usada dentro de una entidad para definir su relación I/O. Hay cuatro para esta decisión:

1. El modelo de flujo de señales es más abstracto. Ptolemy esta enfocado en el diseño del sistema y en la simulación de su comportamiento. Este es usualmente el caso que en esta etapa de diseño, los usuarios trabajan con modelos matemáticos simplificados de un sistema, y los detalles de implementación son desconocidos o no son tenidos en cuenta.

2. El modelo de flujo de señales es más flexible y extensible, en el sentido que es fácil hacer cambios en la topología en el nivel del problema. Para modelos como sistemas híbridos, es más conveniente manipular el sistema en este nivel.

3. El modelo de flujo de señales es consistente con otros modelos de computación en Ptolemy II. La mayoría de los dominios en Ptolemy utilizan el paso de mensajes como semántica de iteración. La escogencia del modelo de flujo de señales para el dominio CT hace que este sea

consistente con otros dominios, así que la interacción de sistemas heterogéneos es fácil de estudiar e implementar.

4. El modelo de flujo de señales es compatible con el modelo de ley conservativa. Para sistemas físicos que están basados en leyes conservativas, usualmente es posible encapsularlos dentro de un entidad en el modelo de flujo de señales. Las entradas de la entidad son las excitaciones, como los voltajes en fuentes de voltaje, y las salidas son las variables en las que el resto del sistema podría estar interesado.

### A.1.6.3. ARQUITECTURA DEL SOFTWARE DEL DOMINIO CT

El dominio CT consiste de los siguientes paquetes, ct.kernel, ct.kernel.util, ct.kernel.solver y ct.lib; estos paquetes implementan la semántica del dominio y los actores y directores encargados de trabajar en este dominio.

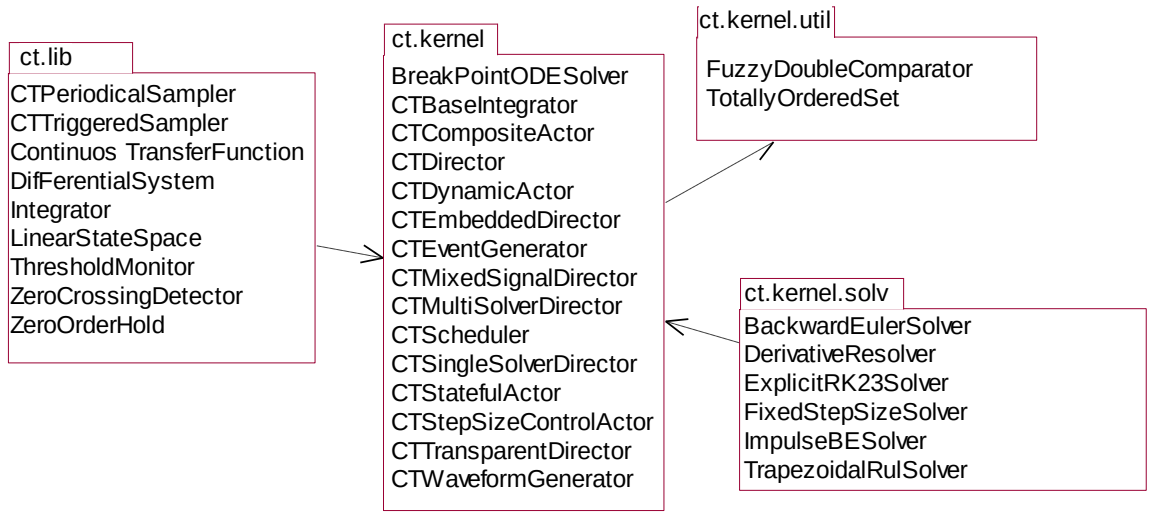


**Figura 6.1.**

#### A.1.6.3.1. PAQUETE ct.kernel

Este paquete es el paquete más importante del dominio CT ya que proporciona las interfaces para clasificar los actores, planeadores, directores, y la clase básica para resolver las ODE's.

#### A.1.6.3.2. PAQUETE ct.kernel.util

Este paquete proporciona una estructura básica de datos, la cual es usada para almacenar los puntos de quiebre y luego procesarlos en el orden en que aparecieron.

#### A.1.6.3.3. DIRECTORES CT

En el dominio CT hay tres directores encargados de la ejecución de los actores de este dominio. Estos directores son: CTMultiSolverDirector, CTMixedSignalDirector y CTEmbeddedDirector. El primero es usado para dirigir la ejecución global de un actor, el segundo es usado para dirigir la ejecución de un actor compuesto (un grupo de varios actores) y el tercero es usado para dirigir actores CT que se encuentran trabajando en otros dominios o modelos de computación.