XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

JOYCE+
MODELO, LENGUAJE Y METODOLOGIA DE PROGRAMACION DE SISTEMAS DISTRIBUIDOS SOBRE REDES DE COMUNICACION

MARIA CONSUELO FRANKY

Grupo de investigación SINBAD
Depto. Ingeniería de Sistemas y Computación
Universidad de los Andes
Apartado aéreo 4976
Bogotá - Colombia

RESUMEN:

Se propone un lenguaje JOYCE+ como una modificación del lenguaje JOYCE definido por Brinch Hansen *[Hansen 87]* . A diferencia de JOYCE que utiliza un esquema de comunicación entre procesos de tipo sincrónico "rendez vous", JOYCE+ propone un esquema de comunicación asincrónico para programar Sistemas Distribuidos que van operar sobre una red de comunicación de máquinas monoproceso y/o multiproceso. El lenguaje permite programar los procesos que componen un Sistema Distribuido de manera concisa, estructurada y con interfaces bien definidas. De hecho la comunicación entre procesos se hace de manera indirecta a través de objetos "puertos" y "canales", lo cual permite programar cada proceso sin tener que conocer nombres de otros procesos y adicionalemente, permite verificar desde compilación el uso correcto de las primitivas de comunicación. En este artículo, además del lenguaje JOYCE+ se propone una metodología para programar Sistemas Distribuidos en la cual se utiliza un esquema único para expresar cualquier clase de proceso en JOYCE+ .

1. INTRODUCCION

JOYCE es un lenguaje propuesto por Brinch Hansen *[Hansen 87]* para describir y programar Sistemas Distribuidos que operan en una sola máquina con capacidad real o simulada de multiproceso. El lenguaje presenta ventajas importantes en relación a otros lenguajes utilizados para el mismo fin, como son CSP *[Hoare 78]* , OCCAM *[Jones 86]* y ADA *[Alford 85]* . Entre las ventajas se pueden citar las siguientes:

- La interfaz de cada proceso está bien definida y el esquema de identificación entre procesos que se comunican es indirecto a través de objetos intermediarios llamados "canales". Esto permite independizar la programación de cada proceso de cualquier nombre utilizado por los demás procesos. Aún los procesos de una misma clase no necesitan distinguirse entre si con nombres o subíndices.

- Los mensajes que puede enviar o recibir un proceso están delimitados por su interfaz, lo cual permite verificar desde compilación el uso correcto de las primitivas de comunicación.

- La activación de procesos es dinámica y eventualmente recursiva, lo cual permite expresar problemas de procesamiento paralelo de manera concisa y sin necesidad de prever de manera estática el número de procesos que se van a activar.

En este artículo proponemos un lenguaje **JOYCE+** como una modificación del lenguaje JOYCE. En el nuevo lenguaje se conservan las ventajas mencionadas pero a diferencia de JOYCE, que utiliza un esquema de comunicación entre procesos de tipo sincrónico "rendez vous", JOYCE+ propone un esquema de comunicación asincrónico para programar Sistemas Distribuidos que van operar sobre una red de comunicación de máquinas monoproceso y/o multiproceso.

En este artículo presentaremos inicialmente (parte 2) el Modelo de Sistema Distribuido JOYCE+ en el cual se basa el lenguaje JOYCE+ . Es necesario tener en cuenta ese modelo cuando se quiere expresar un sistema distribuido en el lenguaje JOYCE+ .

En la parte 3 presentaremos los Elementos del lenguaje JOYCE+, definidos como una extensión de PASCAL con primitivas de comunicación inspiradas en CSP.

En la parte 4 propondremos una Metodología JOYCE+ para programar Sistemas Distribuidos la cual implica ciertas etapas y un esquema único para expresar cualquier clase de proceso componente de un Sistema Distribuido en el lenguaje JOYCE+ .

Finalmente en la parte 5 haremos un recuento de los Proyectos de Software que hemos emprendido con relación al lenguaje JOYCE+.

2. EL MODELO JOYCE+ DE UN SISTEMA DISTRIBUIDO

El lenguaje **JOYCE+** implica un Modelo específico de Sistema Distribuido. A continuación definiremos las clases de objetos que hacen parte del Modelo y sus interrelaciones. La definición se hará sin hacer referencia al lenguaje JOYCE *[Hansen 87]* que fue el que definió originalmente los componentes del Modelo : **JOYCE+** tomó los componentes de JOYCE y los modificó adaptándolos a un ambiente de sistemas distribuidos sobre red de comunicación.

2.1 ESTRUCTURA LOGICA DE UN S.D. EN JOYCE+

Un Sistema Distribuido en el Modelo **JOYCE+** es un conjunto de procesos paralelos llamados **agentes**, que trabajan con algún objetivo común, para lo cual cooperan entre si intercambiando mensajes. Cada agente pertenece a una clase, la cual especifica una secuencia fija de instrucciones que debe ejecutar. Cada agente se ejecuta en un sitio lógico específico de la red sobre la cual opera el Sistema Distribuido. Los agentes no comparten entre si ningún tipo de memoria, aún en el caso de agentes de un mismo sitio lógico. Cada **mensaje** enviado de un agente a otro pertenece a una clase y contiene un dato : para cada clase de mensajes existe un tipo de dato asociado.

La comunicación entre agentes se hace a través de **canales** que son máquinas virtuales que permiten la transmisión unidireccional de un mensaje a la vez. Cada canal es compartido por dos o más agentes pero solo uno de ellos puede asumir el rol de destinatario de los mensajes que se transmitan por el canal : los demás agentes asumen el rol de fuentes de mensajes. Cada canal es de cierto tipo que especifica un conjunto fijo de clases de mensajes que puede transmitir.

Para poder utilizar un canal, un agente debe poseer un **puerto** asociado al canal lo cual constituye una conexión virtual a dicho canal. Como los canales son unidireccionales, los puertos también lo son : un puerto de un agente le permite enviar o recibir mensajes por el canal asociado pero no las dos funciones. Cada puerto es de cierto tipo que especifica el tipo de canal al que se puede conectar. Cuando varios agentes comparten un mismo canal, todos sus puertos conectados al canal deben ser del mismo tipo, que corresponde al tipo del canal.

La estructura lógica de un sistema distribuido en el modelo **JOYCE+** está constituida por el conjunto de agentes, canales y puertos que componen el sistema y sus conexiones. Dicha estructura se puede expresar a través de un grafo en donde cada nodo es un agente junto con sus puertos, y los arcos son las conexiones entre agentes a través de canales. En la figura 1 se ilustra, a manera de ejemplo, el grafo de la estructura lógica de un sistema distribuido.

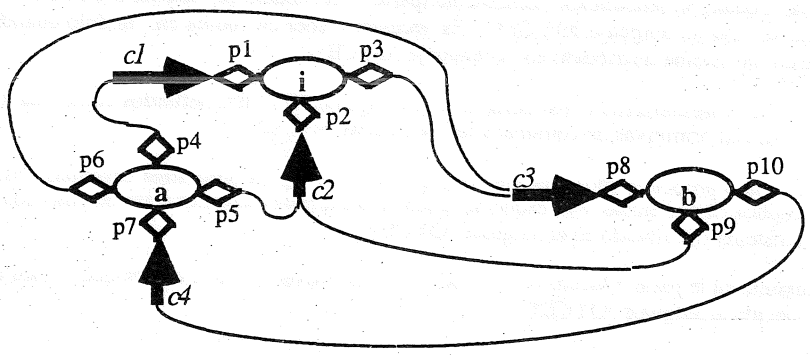

Figura 1 : Grafo de la estructura lógica de un sistema distribuido

Notación : los óvalos representan agentes (para el ejemplo se tienen los agentes **i** , **a** y **b**)
las flechas representan canales ; los rombos representan puertos
las lineas finas representan extensiones de los puertos para conectarse a los canales

En el modelo **JOYCE+** los agentes, los puertos y los canales son creados dinámicamente :

Para iniciar actividad en un sistema distribuido, solo uno de sus agentes (designado el agente inicial) es activado y asignado al sitio lógico 1 (suponiendo una numeración secuencial de los sitios lógicos a partir de 1). Cualquier agente activado puede crear un canal y conectarse a él a través de

uno de sus puertos. También puede activar otro agente asignándolo a cualquier sitio lógico e informándole (en el momento de activación) la referencia a un canal ya creado. Esta información permite al nuevo agente activado conectarse a su vez al canal referenciado a través de uno de sus puertos. Para cada canal compartido entre varios agentes, solo uno de esos agentes es responsable de su creación y los otros reciben en su activación la referencia al canal.

La activación de un agente da lugar entonces a un proceso que debe ejecutar una secuencia de instrucciones (de acuerdo a la clase del agente) en un sitio lógico específico de la red, y que podrá comunicarse con otros agentes ya existentes vía los canales cuya referencias reciba en el momento de activación.

Para el ejemplo del sistema distribuido mostrado en la figura 1, se puede asumir que el agente **i** es el agente inicial, el cual crea los canales *c1, c2* y *c3* y luego activa al agente **a** pasándole referencias a esos canales. El agente **a** a su vez crea el canal *c4* y luego activa al agente **b** pasándole referencias a los canales *c2* , *c3* y *c4* .

En general, la forma como se activan todos los agentes de un sistema distribuido corresponde a un árbol jerárquico. El grafo que representa la estructura lógica de un sistema distribuido es entonces dinámico pues, para cada momento de la vida del sistema, el grafo representa únicamente los agentes activos en ese momento.

La activación recursiva de agentes también es contemplada en el modelo JOYCE+ : un agente puede activar a otro de su misma clase creándose dos procesos paralelos que ejecutan las mismas instrucciones pero que pueden usar diferentes canales de comunicación .

2.2 TIPO DE COMUNICACION ENTRE AGENTES

Para que un agente **a** pueda enviar un mensaje de clase M a otro agente **b** se deben cumplir los siguientes requisitos :

- El agente **a** debe tener un puerto **pa** conectado a un canal **C** con destino otro puerto **pb** del agente **b** .
- Los puertos **pa** y **pb** deben ser del mismo tipo T correspondiente al tipo del canal.
- La clase M de mensajes debe estar considerada en el tipo T del canal.

Una vez cumplidos los anteriores requisitos, la transmisión de un mensaje del agente **a** hacia el agente **b** se realiza de manera asincrónica :

- El agente **a** ejecuta una primitiva enviar en la cual indica el puerto de salida **pa** a través del cual quiere transmitir y el mensaje que quiere enviar. Esta primitiva no implica el bloqueo del agente **a** hasta obtenerse una confirmación de recepción del mensaje por parte del agente **b** destinatario: únicamente hay bloqueo del agente **a** hasta lograr que el canal **C** esté libre para transmitir el mensaje. Sin embargo, el hecho de que el agente **a** obtenga servicio de transmisión por parte del canal **C** no le asegura que el mensaje llegará realmente a su destino (agente **b**) debido a la posibilidad de fallas de comunicación en la red.

- El canal **C** atiende en orden de llegada las solicitudes de envío de los diferentes agentes que tienen puertos conectados a él. Cuando atiende el envío del agente **a** se encarga de depositar el mensaje en una cola de recepción asociada al puerto de entrada **pb** de **b** . Se asume una red de comunicación en la que la única falla de comunicación visible para los canales es la caída o aislamiento de uno de los sitios lógicos de la red : en ese caso, el canal abandona la solicitud de envío que tiene por destino un sitio lógico aislado o caído (por ejemplo el sitio lógico donde se ejecuta el agente **b**).

- Para que el agente **b** reciba efectivamente el mensaje enviado por el agente **a** , debe ejecutar una primitiva recibir en la cual indica el puerto de entrada **pb** por el cual quiere recibir y la clase M de mensaje que quiere recibir. Esta primitiva bloquea al agente **b** hasta encontrar un mensaje de clase M en la cola de recepción asociada a **pb** .

- En el modelo JOYCE+ no se diferencia la comunicación local entre agentes de un mismo sitio lógico de la comunicación externa entre agentes de sitios lógicos distintos.

3. ELEMENTOS DEL LENGUAJE JOYCE+

El lenguaje **JOYCE+** se define como una extensión del lenguaje *PASCAL* para incluir conceptos de programación concurrente. Aquí presentaremos los elementos nuevos de JOYCE+ respecto a *PASCAL* sin hacer referencia al lenguaje JOYCE de Brinch Hansen *[Hansen 87]* que fue el que introdujo originalmente esos elementos nuevos : **JOYCE+** tomó los elementos de JOYCE y los modificó adaptándolos a un ambiente de sistemas distribuidos sobre red de comunicación.

Los elementos nuevos de JOYCE+ respecto a *PASCAL* son básicamente las declaraciones e instrucciones que permiten construir dinámicamente la estructura lógica de un sistema distribuido de acuerdo al modelo JOYCE+ : declaración de una clase de agentes, declaración de un tipo de puerto, declaración de una variable puerto, activación de un agente, creación de un canal, emisión de un mensaje, recepción de un mensaje.

3.1 ESTRUCTURA DE UN PROGRAMA EN JOYCE+

Un programa en **JOYCE+** tiene la siguiente estructura :

```
program  < nombre del programa >;
  < declaración de constantes globales >
  < declaración de tipos globales >
  < declaración del agente inicial >
end .
```

El programa declara solo un agente inicial que contiene a su vez las declaraciones de las clases de agentes que se quieren activar. Todos los agentes que hacen parte de un sistema distribuido deben quedar declarados en un solo programa. Cuando se inicia la ejecución del sistema distribuido, el agente inicial es activado en el sitio lógico 1.

Debe notarse que no hay declaración de variables globales, reflejando el hecho de que los agentes deben sincronizarse únicamente a través de mensajes y no a través de memoria compartida.

La < declaración de constantes globales > y < declaración de tipos globales > usa la misma sintaxis de *PASCAL* , extendiéndo esta última para considerar un tipo de datos adicional : el tipo puerto.

3.2 DECLARACION DE UN TIPO DE PUERTO

La declaración de tipos de puertos permite enumerar las clases de mensajes que pueden transmitirse a través de los diferentes canales que conectan los agentes de un sistema distribuido. El tipo de un

canal es el alfabeto de clases de mensajes que puede transmitir. Cada puerto de un agente debe pertenecer a un tipo de puerto, el cual indica el tipo de canal al cual puede conectarse.

Cada puerto de un agente corresponde en el programa a una variable de un tipo de puerto. El tipo enumera un alfabeto de clases de mensajes, donde cada clase tiene un nombre y un tipo de dato asociado. A continuación ilustramos la declaración de un tipo de puerto denominado "tpto" el cual contempla 3 clases de mensajes llamadas "número" , "código", y "ack":

```
TYPE   tpto = [número (integer), código (char), ack ( ) ]
```

Como se observa en el ejemplo, algunas clases de mensajes podrían no tener tipo de dato asociado (como "ack") : a estas clases se les denomina *señales* . El tipo de dato asociado a una clase de mensajes es cualquier tipo de *PASCAL*., simple o compuesto (en este último caso, debe declararse previamente).

3.3 DECLARACION DE UNA CLASE DE AGENTES

Cada agente del sistema distribuido descrito por un programa JOYCE+ pertenece a una clase, la cual está asociada a una secuencia específica de instrucciones. La < declaración del agente inicial > también declara una clase de agentes de la cual solo va a activarse una instancia : el agente inicial del sistema distribuido (esta activación se hará de forma automática cuando el sistema distribuido empiece su ejecución).

En general la declaración de una clase de agentes tiene la siguiente estructura :

```
agent <nombre de la clase> (<declaración de parámetros formales>);
  <declaración de constantes>
  <declaración de tipos>
  <declaración de clases de agentes>
  <declaración de variables locales>
  <declaración de procedimientos locales>
begin
  <cuerpo de instrucciones>
end
```

De forma semejante a un procedimiento de *PASCAL* , un agente en JOYCE+ puede declarar constantes, tipos (incluyendo tipos de puertos), variables locales (incluyendo variables de tipo puerto) y procedimientos locales.

Además una clase de agentes ***A*** puede declarar a otra clase de agentes ***B*** : ésto significa que una vez activado un agente de clase ***A*** , él puede activar (a través de instrucciones de su < cuerpo de instrucciones >) a uno o varios agentes de clase ***B***.

Cada agente puede recibir en el momento de su activación parámetros por valor (exceptuando al agente inicial): estos parámetros corresponderán generalmente a "apuntadores" a canales ya existentes pero también pueden incluir cualquier parámetro por valor válido en *PASCAL*. La ausencia de parámetros por referencia (en los cuales se pudieran devolver resultados) y de variables globales impide a los agentes compartir memoria y los obliga a sincronizarse únicamente por mensajes que se envían a través de los canales.

En el < cuerpo de instrucciones > asociado a una clase de agentes puede ir cualquier instrucción de *PASCAL* e instrucciones nuevas para crear canales, activar agentes, enviar mensajes y recibir mensajes. Estas instrucciones nuevas se explicarán en las secciones que siguen.

3.4 CREACION DE UN CANAL CON CONEXION A UN PUERTO

Para que un agente pueda crear un nuevo canal del sistema distribuido, debe poseer una variable del tipo puerto asociado al canal. En el momento de la creación del canal, dicha variable "puerto" representará el puerto del agente que va a estar conectado al canal. Teniendo en cuenta que los canales son unidireccionales en JOYCE+, cada puerto de un agente es de entrada (in) o de salida (out) de mensajes.

Una variable puerto de un agente es entonces una variable local que debe ser declarada dentro de su clase de agentes. A continuación se da, como ejemplo, la declaración de una variable puerto de salida llamada **pa** de tipo " tpto " (tipo definido en el ejemplo del numeral 3.2) :

```
VAR  pa : out tpto
```

Una variable puerto es en realidad un apuntador a ese elemento dinámico representado por el canal, el cual es proporcionado por el entorno del sistema distribuido.

Para crear un canal y conectarlo a una variable puerto, un agente debe utilizar la siguiente instrucción :

```
+ <nombre de variable puerto>
```

El efecto de esta instrucción es la creación de un canal que queda apuntado por la variable puerto indicada, y que es del tipo asociado a esta variable. Por ejemplo, **+pa** asigna a la variable puerto **pa** la "dirección" de un nuevo canal de tipo " tpto " .

Múltiples variables puertos de un agente se pueden agrupar en un arreglo si todas ellas son del mismo tipo : es decir, si van a conectarse a canales del mismo tipo en la misma dirección (i.e todos los canales son de entrada o de salida). Por ejemplo, la siguiente es la declaración de un arreglo **arrpa** de puertos de salida de tipo "tpto" :

```
VAR arrpa : ARRAY [1..max] OF out tpto
```

3.5 ACTIVACION DE UN AGENTE

Suponiendo que una clase ***A*** de agentes contiene la declaración de otra clase ***B*** de agentes, la siguiente instrucción (en el <cuerpo de instrucciones> de ***A***) permite a un agente de clase ***A*** activar a un nuevo agente de clase ***B*** para ser ejecutado en un sitio lógico **i** del sistema distribuido :

```
B ( <parámetros actuales> ) at i
```

Se asume aquí que los sitios lógicos del sistema distribuido se numeran con enteros diferentes (por ejemplo 1, 2, 3, ...) y que su asociación a sitios físicos de la red de comunicación se establece de forma dinámica a través de un catálogo externo al programa. Todos los sitios lógicos del sistema distribuido podrían asociarse a un mismo sitio físico permitiendo probar el sistema en una sola máquina antes de montarlo en una red de varias máquinas. En general, el sitio lógico donde se va a activar un agente se especifica dando una expresión de tipo *integer*.

Cualquier agente del sistema distribuido tiene un ciclo de vida consistente en 3 fases sucesivas:

- **fase de activación** : al ser activado el agente crea una instancia del conjunto de parámetros formales declarado en su clase y le asigna los valores de los parámetros actuales. También crea una instancia del conjunto de variables locales declarado en su clase. La unión de estas dos instancias constituye el conjunto de variables propias del nuevo agente (inaccesibles para cualquier otro agente, aún si es de su misma clase).

- **fase de ejecución** : el agente ejecuta el cuerpo de instrucciones asociado a su clase, trabajando sobre sus variables propias. A través de esas instrucciones puede, entre otras acciones, crear canales y activar a otros agentes (que consituyen sus subagentes).

- **fase de terminación** : al terminar sus instrucciones el agente debe esperar a que sus subagentes finalicen y desaparezcan completamente, y luego destruye sus variables propias y los canales que haya creado. Finalmente desaparece del sistema distribuido.

Un agente de clase *A* puede activar varios agentes de clase *B* : cada uno de los subagentes tendrá un conjunto distinto de variables propias y puede o no recibir diferentes valores en los parámetros actuales. Un agente de clase *A* puede activar a otro agente de su misma clase *A* , lo que constituye una activación recursiva (aspecto que da mucho poder al lenguaje).

3.6 CONEXION DE UN PUERTO A UN CANAL YA EXISTENTE

Una vez que un agente **a** ha creado un canal con conexión a un puerto local **pa** puede activar un subagente **b** pasándole como parámetro el valor de su puerto **pa**, es decir un apuntador al canal. Asumiendo que el subagente **b** es de clase *B* y que va a ejecutarse en el sitio i, su activación es de la forma : `B (pa) at i`

El subagente **b** debe recibir el apuntador al canal en un parámetro formal **pb** del mismo tipo de puerto de **pa**, quedando de esta forma conectado al canal a través del puerto local **pb**. Dependiendo de la dirección y agente destino del canal, los dos puertos **pa** y **pb** pueden ser de salida hacia el canal (en cuyo caso debe existir un tercer agente **c** destinatario del canal) , o el uno de entrada y el otro de salida. Asumiendo por ejemplo que el puerto **pb** es de entrada (i.e. el canal tiene como destino el subagente **b**) y que es de tipo "tpto", el encabezado de la clase *B* de agentes debe ser de la forma :

```
B ( pb : in tpto)
```

Con las suposiciones anteriores, la figura 2 ilustra la asociación de los agentes **a** y **b** a través de puertos conectados a un canal común :

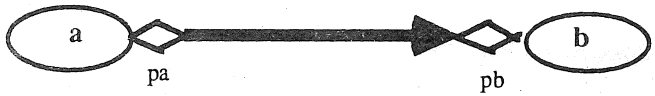

Figura 2 : Conexión del agente b a un canal creado por el agente a

La asociación de los agentes **a** y **b** permite que los dos intercambien mensajes a través del canal común usando las primitivas de comunicación que se presentarán en las siguientes secciones.

3.7 ENVIO DE UN MENSAJE

Asumiendo la asociación de los agentes **a** y **b** mostrada en la figura 2, el agente **a** puede enviar un mensaje *m* al agente **b**, siempre y cuando *m* pertenezca a una de las clases enumeradas en " tpto " que es el tipo de canal sobre el cual se va a transmitir.

Por ejemplo, si la declaración global de "tpto" es la siguiente :

```
TYPE tpto = [número(integer), código(char), ack( ) ]
```

entonces el agente **a** puede enviar al agente **b** el mensaje " número (2345 MOD 47) ", y para ello debe ejecutar la siguiente instrucción : **pa** ¡ número(2345 MOD 47) .

La forma general de la instrucción para enviar un mensaje es la siguiente :

```
<puerto de salida> ¡ <clase de mensaje> ( <expresión> )
```

En tiempo de compilación se puede verificar que la instrucción sea correcta chequeando los siguientes aspectos :

- El < puerto de salida > es una variable puerto o un parámetro formal de tipo puerto del agente que ejecuta la instrucción. En cualquier caso su declaración debe especificar que es un puerto de salida (out). Cuando el <puerto de salida> corresponde a un elemento de un arreglo de puertos, se debe especificar así : <nombre de arreglo de puertos> [<subíndice>] , donde el <subíndice> es una expresión de tipo *integer* .
- La <clase de mensaje> debe estar considerada en la declaración del tipo de puerto correspondiente al <puerto de salida>.
- La <expresión> debe ser del tipo de dato asociado a la <clase de mensaje> . Para aquéllos mensajes que son "señales " la <expresión> es nula.

En tiempo de ejecución de esta instrucción se realizan las siguientes acciones :

- Evaluar la <expresión> (si está presente) para calcular el dato que debe ir en el mensaje.
- Conformar el mensaje, el cual está compuesto de <clase de mensaje> y dato acompañante.
- Transmitir el mensaje a través del canal que está conectado al <puerto de salida> (hay una espera implícita mientras se obtiene turno para transmitir por el canal).

Después de estas acciones, el agente que ejecuta la instrucción de envío (agente "emisor") prosigue con la ejecución de la siguiente instrucción sin requerir antes la confirmación de recepción del mensaje por parte del agente destinatario del canal (i.e. comunicación asincrónica). Por su parte, el canal está encargado de depositar el mensaje en orden FIFO en la cola de recepción asociada a un puerto de entrada del agente destinatario (el puerto de entrada depende de la conexión establecida entre los agentes). El tipo de transmisión del mensaje será local si ambos agentes (emisor y destinatario) están asignados al mismo sitio lógico, o externa si están asignados a sitios lógicos distintos; sin embargo, en los dos casos se usan las mismas primitivas de comunicación.

3.8 DIFUSION DE UN MENSAJE

Cuando se tiene un arreglo de puertos de salida se puede difundir un mensaje a través de todos los canales conectados a esos puertos (que por definición son del mismo tipo). La forma general de la instrucción de difusión es la siguiente :

```
<nombre de arreglo de puertos> [<exp1>..<exp2>] ¡
                <clase de mensaje> ( <expresión> )
```

El rango de los puertos utilizados está delimitado por <exp1> y <exp2> que son 2 expresiones de tipo *integer*. La instrucción de difusión equivale a una instrucción FOR de *PASCAL* en la que se ejecuta una instrucción de envío del mensaje por cada uno de los puertos correspondientes al rango especificado.

3.9 RECEPCION SIMPLE DE UN MENSAJE

Asumiendo de nuevo la asociación de los agentes **a** y **b** mostrada en la figura 2 y siguiendo con el ejemplo de la sección 3.7, el agente **b** puede recibir el mensaje " número (2345 MOD 47) " enviado por el agente **a** ejecutando la siguiente instrucción :

```
pb ¿ número ( cont )
```

donde "cont" es una variable local de tipo *integer* , que es el tipo de dato asociado a la clase de mensaje "número". Esta instrucción implica un bloqueo del agente **b** hasta que el mensaje deseado llegue por su puerto de entrada **pb** (i.e. sea depositado en la cola de recepción asociada) y la posterior asignación del dato del mensaje a la variable "cont ".

La forma general de la instrucción para recibir un mensaje es la siguiente :

```
<puerto de entrada> ¿ <clase de mensaje> (<variable>)
```

En tiempo de compilación se puede verificar que la instrucción sea correcta, de manera análoga a la verificación de una instrucción de envío de un mensaje (explicada en la sección 3.7).

En tiempo de ejecución de esta instrucción se realizan las siguientes acciones :

- El agente que ejecuta la instrucción de recepción se bloquea hasta que algún mensaje correspondiente a la <clase de mensaje > llegue a la cola de recepción asociada al <puerto de entrada> . (El bloqueo en reallidad corresponde a un ciclo continuo en el cual el agente está examinando en orden FIFO el contenido de toda la cola de recepción).
- Cuando el mensaje llega y el agente lo detecta, el dato que lleva (si está presente) es asignado a la <variable> .
- El mensaje recibido es eliminado de la cola de recepción asociada al <puerto de entrada> .

3.10 RECEPCION DE MENSAJES DENTRO DE UNA SELECCION (POLLING)

La recepción simple de un mensaje implica el bloqueo del agente que ejecuta la instrucción esperando que le llegue un mensaje de una clase específica. En aquéllos casos en los cuales no se puede asumir el orden en que van a llegar los mensajes a un agente, es más flexible y eficiente que el agente se bloquee esperando no un mensaje específico sino uno entre varios mensajes posibles de diversas clases. La recepción efectiva de un mensaje específico entre varios presentes podría someterse además a la verificación de alguna condición.

Para lograr esta recepción flexible, muy adecuada para agentes que actúan como procesos "servidores", se debe utilizar la instrucción compuesta **POLL** cuya forma general es la siguiente :

```
Poll
  <recepción condicionada1>   -->  <tratamiento1>
/ <recepción condicionada2>   -->  <tratamiento2>
.................
/ <recepción condicionadan>   -->  <tratamienton>
end
```

Cada <recepción condicionada> es una instrucción de recepción simple acompañada de una condición opcional (donde la condición es cualquier expresión booleana válida en *PASCAL*). Cuando lleva condición, la <recepción condicionada> se especifica de la siguiente manera :

```
<puerto de entrada> ¿ <clase de mensaje> (<variable>) & <condición>
```

Con esta instrucción el agente intenta seleccionar y recibir un solo mensaje entre los varios posibles que aparecen en la lista de " recepciones condicionadas ". Cuando escoge y recibe efectivamente un mensaje, el agente debe ejecutar el tratamiento asociado. Cada < tratamiento > es una secuencia cualquiera de instrucciones JOYCE+ .

La ejecución de una instrucción **POLL** se realiza en 2 fases sucesivas :

- fase "polling": Se realiza un examen cíclico de cada <recepción condicionada> empezando por la primera, hasta lograr escoger una. Al examinar una <recepción condicionada> (en la que se especifica una posible condición y la recepción de un mensaje de cierta clase a través de un puerto determinado) ésta se escoge solo si su condición se cumple y si el mensaje asociado está presente en la cola de recepción del puerto. Si hay varios mensajes de la clase deseada presentes en la cola, se escoge el primero en orden FIFO. Cuando la condición no está presente equivale a una condición booleana siempre cierta (i.e. TRUE). El examen cíclico de las recepciones condicionada implica que si no se ha logrado escoger ninguna en el primer ciclo se vuelven a examinar todas empezando de nuevo por la primera, y así sucesivamente en los ciclos que sean necesarios.

- fase " completion " : Una vez escogida una <recepción condicionada>, en esta nueva fase se realizan las acciones asociadas a la recepción seleccionada (i.e. el dato del mensaje se asigna a una variable local y el mensaje se elimina de la cola de recepción del puerto). En seguida se ejecuta el <tratamiento> que figura en la instrucción POLL a la derecha de la <recepción condicionada> escogida.

Para evitar que un agente se quede indefinidamente en la fase "polling" de una instrcción **POLL** (en el caso que no se logre escoger una < recepción-condicionada > en ciclos sucesivos) se puede usar la claúsula opcional ELSE TIMEOUT dentro de la instrucción POLL, de la siguiente forma :

```
Poll
  <recepción condicionada1>  -->  <tratamiento1>
/ <recepción condicionada2>  -->  <tratamiento2>
.................
/ <recepción condicionadan>  -->  <tratamienton>
  else timeout( <máx> )        -->  <tratamienton+1>
end
```

El símbolo <máx> es una expresion de tipo *integer* que denota el máximo número de ciclos que se pueden realizar en la fase " polling ". Si al completar dicho número de ciclos no se ha logrado escoger una <recepción condicionada > , el agente ejecuta el tratamiento alterno <tratamiento$_{n+1}$> y da por terminada la instrucción POLL. La claúsula ELSE TIMEOUT de la instrucción POLL es muy útil en un ambiente de red de comunicación en el cual existe la posibilidad de sitios caídos o aislados : en este caso un agente puede llegar a encontrarse aislado de los demás agentes, y al no poder recibir ningún mensaje podría ejecutar alguna acción de recuperación en el tratamiento alterno. Otro ejemplo en el cual la claúsula ELSE TIMEOUT es útil es cuando un agente **a** quiere verificar que otro agente **b** recibió un mensaje que le envió previamente : al no recibir el mensaje de confirmación de recepción dentro de un período máximo puede asumir que el mensaje inicial no fue recibido (posiblemente por aislamiento del agente **b**).

3.11 ANALISIS DEL LENGUAJE JOYCE+

En comparación a otros lenguajes utilizados para programar sistemas distribuidos como son CSP *[Hoare 78]* , OCCAM *[Jones 86]* y ADA *[Maekawa 87, Alford 85]* , el lenguaje JOYCE+ presentan las siguientes ventajas (válidas en general también para el lenguaje original JOYCE *[Hansen 87]*) :

- Interfaces definidas entre procesos : Cada agente de JOYCE+ (que corresponde al concepto de proceso) tiene una interfaz definida respecto a los demás agentes. Esa interfaz la constituyen el conjunto de puertos de entrada y de salida utilizados por el agente, y que son declarados como parámetros formales o como variables locales. La interfaz de un agente junto con la declaración de los tipos de sus puertos especifica todas las clases de mensajes que puede recibir y enviar. En CSP y OCCAM la interfaz de cada proceso no está definida, y en ADA solo se define la interfaz respecto a los mensajes que pueden llegar al proceso.

- Activación dinamica y recursiva de procesos : En JOYCE+ los agentes se activan de forma dinámica y eventualmente recursiva, por lo cual no hay que prever desde compilación el número total de agentes que existirán en ejecución. La activación recursiva permite expresar de forma concisa y simple soluciones a problemas en los que intervienen múltiples agentes de la misma clase, sin necesidad de recurrir a subíndices para diferenciar los agentes como ocurre en CSP. Ninguno de los lenguajes CSP, OCCAM o ADA permite activación recursiva de procesos.

- Independencia entre procesos : Los agentes en JOYCE+ no se nombran explícitamente cuando intercambian mensajes; solo requieren nombrar sus puertos locales (que deben estar conectados a canales compartidos entre los agentes que quieren intercambiar mensajes). Este esquema de identificación indirecta permite independizar el diseño de cada agente de los nombres específicos de los demás agentes del sistema. Por otra parte, esta independencia entre agentes es fundamental cuando la activación de los agentes es dinámica y no se conoce en tiempo de compilación cúantos y cúales agentes se van a activar en ejecución. En los lenguajes CSP, OCCAM y ADA los procesos deben nombrarse explícitamente cuando ejecutan primitivas para enviar mensajes.

- Verificación de las conexiones entre procesos en compilación : En JOYCE+ hay declaración de clases de mensajes, de tipos de puertos y de puertos, que son los que permiten asociar agentes a través de canales. Estas declaraciones permiten verificar desde compilación que los mensajes intercambiados por agentes que van a estar conectados a un mismo canal pertenecen a un mismo alfabeto, es decir, a un mismo conjunto de clases de mensajes. También se puede chequear desde compilación que ningún agente en ejecución va a tratar de transmitir por un canal inexistente. Estas verificaciones en tiempo de compilación hacen de JOYCE+ un lenguaje seguro respecto a los lenguajes CSP, OCCAM y ADA.

- Sincronización de procesos únicamente por mensajes : Al igual que CSP, JOYCE+ considera la sincronización de procesos únicamente por mensajes y no por memoria compartida (ADA permite ambos tipos de sincronización). Este tipo de sincronización es el más adecuado para un ambiente en el cual los procesos pueden ejecutarse en sitios lógicos distintos y es impuesto en JOYCE+ por la ausencia de variables globales y de parámetros por referencia.

- Comunicación asincrónica, manejo de timeouts y especificación de sitios lógicos : Las ventajas expuestas hasta aquí son válidas tanto para JOYCE+ como para el lenguaje del cual se derivó : JOYCE. Lo que es propio de JOYCE+ es la comunicación asincrónica entre agentes a diferencia de la comunicación sincrónica "rendez-vous" considerada en JOYCE, y también en CSP, OCCAM y ADA. La comunicación asincrónica es más adecuada para sistemas distribuidos que operan sobre redes de comunicación no confiables, y con ella se busca bloquear a los agentes el menor tiempo posible cuando ejecutan las primitivas de comunicación *[Sloman 87]* . Además el manejo de timeout en una selección de primitivas de recepción (lo que corresponde a la claúsula ELSE TIMEOUT de la instrucción POLL) evita el bloqueo de un agente cuando se encuentra aislado de sus agentes interlocutores (también ADA tiene manejo de timeouts pero no lo tienen CSP, OCCAM ni JOYCE). Otro aspecto propio de JOYCE+ es la especificación del sitio lógico en la activación de un agente (aspecto no considerado en CSP, OCCAM, ADA ni JOYCE), lo cual es necesario en sistemas distribuidos de agentes que pueden ejecutarse en distintos sitios lógicos. Por último JOYCE+ incorpora otros aspectos nuevos respecto a JOYCE, como son la declaración de procedimientos locales a un agente (con lo cual se evita la proliferación de agentes en el sistema y se controla el "overhead" en comunicación) y la instrucción de difusión de un mensaje por puertos de salida pertenecientes a un mismo arreglo, la cual puede implementarse de forma eficiente en una red de comuncación local.

4. LA METODOLOGIA DE PROGRAMACION **JOYCE+**

En muchos sistemas distribuidos en donde la sincronización de los procesos (agentes) es únicamente por mensajes y no por memoria compartida, los procesos son diseñados como servidores de mensajes . Un proceso servidor de mensajes es aquél que después de su inicialización entra en un "ciclo de atención" casi permanente en el cual está capacitado para servir cualquier mensaje correcto que le llegue dándole el tratamiento adecuado, tratamiento que incluye el envío de mensajes de respuesta correspondientes al servicio. El proceso solo puede salir del ciclo de atención cuando se cumpla alguna condición que indique que debe terminar su ejecución.

En el modelo y lenguaje JOYCE+ un proceso servidor corresponde a un agente cuyo cuerpo de instrucciones consiste principalmente en un ciclo dentro del cual hay una sola instrucción POLL especificando recepciones para todas las clases de mensajes correspondientes a los puertos de entrada del agente. Las condiciones del POLL permiten diferenciar el tratamiento de mensajes de la misma clase según el orden relativo en que llegan.

Los Sistemas Transaccionales Distribuidos que operan sobre redes de comunicación utilizan generalmente el enfoque de procesos servidores de mensajes. Otras clases de sistemas distribuidos no lo utilizan pero podrían transformarse en sistemas destribuidos equivalentes de procesos servidores de mensajes.

La metodología JOYCE+ que expondremos en las siguientes secciones indica cómo programar en lenguaje JOYCE+ sistemas distribuidos de procesos servidores de mensajes. La metodología implica expresar primero la estructura lógica del sistema en el modelo JOYCE+, definir luego la interfaz de cada clase de agentes especificando los tipos de puertos involucrados, programar luego el cuerpo de cada clase de agentes especificando el ciclo de atención, y por último programar la conexión dinámica entre agentes a través de canales. Hemos restringido la metodología para el caso en el cual el agente inicial es el único responsable de la conexión dinámica entre agentes (i.e. el agente inicial es el único que crea canales y activa a los demás agentes). En un futuro pensamos ampliar la metodología para el caso general en el cual la conexión dinámica es responsabilidad de todos los agentes y se puede tener además activación recursiva de agentes.

4.1 ETAPA 1: ESTRUCTURA LOGICA DEL SISTEMA DISTRIBUIDO

La Etapa 1 de la medodología de programación JOYCE+ implica definir la estructura lógica del sistema distribuido que se quiere programar y expresarla en el modelo JOYCE+. Teniendo en cuenta que la estructura lógica es dinámica, nos interesa aquí la que correponde al momento en el cual el agente inicial ya ha activado todos los demás agentes del sistema distribuido. Como se explicó en la parte 2 de este artículo, dicha estructura lógica se puede expresar a través de un grafo en donde se muestran como nodos los agentes que componen el sistema distribuido junto con sus puertos locales, y en donde los arcos representan las conexiones de comunicación a través de canales.

Para nombrar los agentes de los diferentes sitios lógicos, en la estructura lógica se puede asumir como identificador único de un agente la concatenación de :

nombre de su clase, sitio lógico, número de instancia

El *número de instancia* es un orden secuencial relativo para los agentes de la misma clase asignados al mismo sitio lógico. Este esquema de identificación único de agentes solo es válido en esta Etapa 1 puesto que en el programa que describirá el sistema distribuido en lenguaje JOYCE+ solo se definen clases de agentes sin nombrar explícitamente instancias de los mismos.

Teniendo en cuenta que en esta medodología de programación JOYCE+ el agente inicial es el único responsable de la creación de canales, activación y conexión de todos los otros agentes, no es necesario representar en la estructura lógica del sistema distribuido al agente inicial : se asume que es un agente adicional a los representados en el grafo con un puerto de salida por cada canal del sistema

La figura 3 muestra a manera de ejemplo el grafo de la estructura lógica de un sistema distribuido :

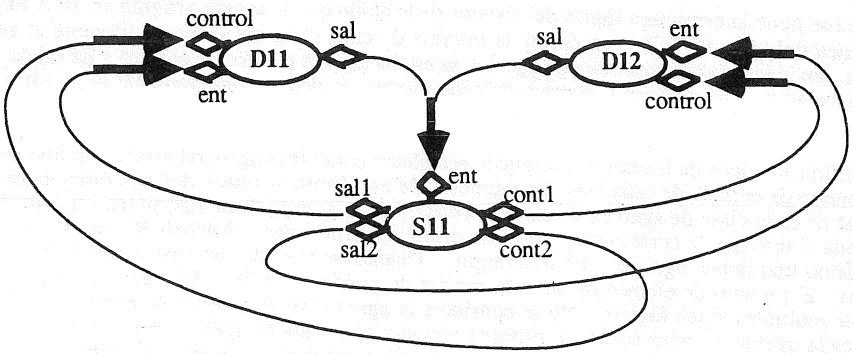

Figura 3. Etapa 1 de la metodología JOYCE+ : estructura lógica del sistema distribuido

El ejemplo de la figura 3 representa un sistema distribuido con dos agentes de clase **D** ("Despachador") en el sitio lógico 1 y un agente de clase **S** ("Servidor") en el sitio 2. Suponemos que los agentes **D** interactúan con los usuarios para ofrecer los servicios del sistema y generan mensajes de peticiones de servicios que envían al agente **S**. El agente **S** realiza los servicios accesando una Base de Datos y envía mensajes de respuestas de servicios hacia los agentes **D**. Para separar las respuestas normales a servicios de los mensajes de control (que indican, por ejemplo, que un servicio no se pudo realizar), cada agente **D** tiene 2 puertos de entrada. El agente inicial y sus conexiones no están representados en el grafo.

Para el caso de sistemas distribuidos " *homogéneos* " que tienen las mismas clases y número de instancias de agentes en todos los sitios lógicos, es suficiente expresar la estructura lógica para 2 o 3 sitios previendo su generalización a más sitios.

Para construir la estructura lógica de un sistema distribuido se pueden establecer las siguientes reglas generales :

- Cada agente debe tener por lo menos un puerto de entrada con conexión a un canal. Este puerto le servirá para recibir mensajes provenientes de otros agentes, mensajes que corresponden a peticiones de servicios que brinda el agente en cuestión o respuestas a servicios brindados por otros agentes. En el caso de que se quieran discriminar los mensajes de entrada a un agente por prioridad o por razones de claridad, se pueden considerar varios puertos de entrada para ese agente.

- Hay un canal conectado y destinado a cada puerto de entrada de un agente. Por lo tanto el número total de canales en el sistema distribuido es igual al número total de puertos de entrada de los diversos agentes.

- Un agente debe tener tantos puertos de salida como canales (de salida) necesite conectados para enviar mensajes a otros agentes. Estos mensajes corresponden a peticiones de servicios que brindan otros agentes o respuestas a servicios brindados por el agente en cuestión.

- Todos los agentes de la misma clase tienen los mismos puertos de entrada y de salida, y usan los mismos nombres (locales) para esos puertos. Como identificador único de cada agente se asume la concatenación de *nombre de su clase, sitio lógico, número de instancia* .

4.2 ETAPA 2: INTERFAZ DE CADA CLASE DE AGENTES

Una vez se tiene la estructura lógica del sistema distribuido que se quiere programar, en la Etapa 2 de la metodología JOYCE+ se define la interfaz de cada clase de agentes (diferente al agente inicial). En la interfaz de una clase de agentes se establecen los puertos de entrada y de salida y sus tipos asociados. El número y nombre de esos puertos se deduce directamente de la estructura lógica.

Para definir los tipos de los puertos se puede establecer como regla general asociar un tipo único a cada puerto de entrada de cada clase de agentes. De esta forma se puede definir completamente la interfaz de cada clase de agentes, asociando a cada puerto de entrada un tipo único y a cada puerto de salida el tipo que le corresponde al puerto destinatario del canal. Además los puertos de salida del mismo tipo deben agruparse en un arreglo. Cuando se trata de sistemas distribuidos "*homogéneos* " el número de elementos de cada arreglo de puertos de salida debe generalizarse para el caso de múltiples sitios lógicos. No se considera la agrupación de puertos de entrada en arreglos pues cada agente se mira como un proceso servidor que trata de igual forma las peticiones de servicio procedentes de diversos agentes: no requiere entonces un puerto de entrada por cada agente fuente sino uno solo común para todos. Por otra parte, para discriminar las clases de mensajes por prioridades sí se pueden considerar varios puertos de entrada, y en ese caso son de distinto tipo.

La figura 4 ilustra la interfaz de cada clase de agentes del sistema distribuido correspondiente a la estructura lógica que fue mostrada en la figura 3.

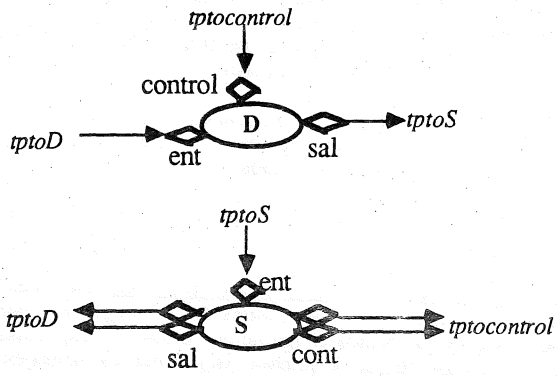

Figura 4 . Etapa 2 de la metodología JOYCE+ : interfaz de cada clase de agentes

Comparando las figuras 3 y 4 se observa que los puertos de salida de la clase de agentes S han sido agrupados en 2 arreglos (sal y cont) de acuerdo al tipo de los puertos.

Al asumir en la metodología de programación JOYCE+ que solo el agente inicial es responsable de la conexión dinámica entre agentes, todos los tipos de puertos deben ser declarados a nivel global del programa que describe un sistema distribuido particular. La declaración misma de cada tipo de

puerto, asociado a un puerto de entrada de una clase de agentes, debe enumerar todas las clases de mensajes que pueden llegar a ese puerto de entrada, las cuales corresponden a la unión de :
- peticiones de servicios que puede brindar un agente de la clase asociada
- respuestas a servicios que otros agentes brindan a un agente de la clase asociada

En la declaración de un tipo de puertos se asocia un tipo de dato a cada clase de mensaje, el cual puede ser cualquier tipo simple o compuesto válido en *PASCAL*. Ese tipo de dato debe resumir los campos (i.e. parámetros) que componen un mensaje de la clase en cuestión. Los tipos de datos se declaran también a nivel global precediendo la declaración de los tipos de puertos.

Cuando se tienen varios puertos de entrada para una misma clase de agentes, los conjuntos de clases de mensajes correspondientes a los tipos de esos puertos no requieren ser disyuntos : por ejemplo, para dar prioridad de tratamiento a un mensaje sobre otro podrían llegar por distintos puertos de entrada aún siendo los dos de la misma clase de mensajes.

Además de los tipos simples de puertos, es necesario declarar tipos arreglos de esos tipos para prever la declaración de los arreglos de puertos de salida que puedan tener las diversas clases de agentes.

Siguiendo con el ejemplo del sistema distribuido correspondiente a las interfaces de clases de agentes mostradas en la figura 4, se podrían tener las siguientes declaraciones globales de constantes, tipos de datos y tipos de puertos (que asocian esos tipos de datos a sus clases de mensajes) :

```
CONST   maxD          = 20;

TYPE tx               = record of
                            cuenta    :  integer;
                            cantidad  :  real;
                            sitio     :  integer
                        end;
     ty               = record of
                            cuenta    :  integer;
                            sitio     :  integer
                        end;

     tptoS            = [ consignar(tx), retirar(tx), saldo(ty) ];
     tptoD            = [ rta_operacion(boolean), rta_saldo(real) ];
     tptocontrol      = [ reactivacion( ) ];
     tarrptoD         = ARRAY [1..maxD] of tptoD;
     tarrptocontrol   = ARRAY [1..maxD] of tptocontrol;
```

Se muestran en estas declaraciones mensajes típicos de una aplicación bancaria : el agente **S** (Servidor) puede recibir peticiones de 3 clases de servicios (considerados en "tptoS") : consignar, retirar o solicitud de saldo. En los 3 casos el agente **S** consulta y/o modifica la Base de Datos de cuentas bancarias y genera un mensaje de respuesta de servicio : en el caso de una consignación o de un retiro la respuesta indica el éxito o fracaso de la operación (mensaje rta_operación) y en el caso de una solicitud de saldo el mensaje rta_saldo informa el saldo requerido. Estos mensaje de respuestas de servicios (considerados en "tptoD") son enviados al agente **D** (Despachador) que hizo la solicitud. El dato asociado a un mensaje que solicita el servicio de consignar o de retirar es un registro de tipo "tx" con 3 campos que indican : la cuenta, la cantidad que se quiere creditar o debitar, y el sitio lógico asignado al agente **D** que hace la solicitud. En el caso de una solicitud de saldo, el dato asociado es un registro de tipo "ty" con 2 campos : cuenta y sitio lógico del agente **D** solicitante.

El mensaje reactivación (considerado en "tptocontrol") es una señal de control enviada por el agente **S** a un agente **D** para indicar que el sitio lógico asignado a **S** está de nuevo en linea después de haber sufrido un aislamiento en términos de comunicación.

En total se declaran 3 tipos simples de puertos, correspondientes a los que aparecen en la interfaces de las clases de agentes (figura 4). Además se declaran 2 tipos arreglos de puertos en base a los anteriores, con el fin de poder declarar posteriormente los arreglos de puertos de salida que aparecen en la interfaz de la clase de agente **S**.

4.3 ETAPA 3 : ESPECIFICACION DE CADA CLASE DE AGENTES

Una vez establecida la interfaz de cada clase de agentes, en la Etapa 3 de la Metodología de programación JOYCE+, se procede a especificar completamente cada clase de agentes diferente al agente inicial.

Teniendo en cuenta que en esta metodología solo el agente inicial activa a los demás agentes, la especificación de una clase de agentes tiene el siguiente esquema general :

```
agent <nombre de la clase> (<declaración de parámetros formales>);
< declaración de constantes >
< declaración de tipos >
< declaración de variables locales >
< declaración de procedimientos locales >
begin
< cuerpo de instrucciones >
end
```

Las diferentes **clases de parámetros formales** (todos por valor) que se pueden declarar en el encabezado de una clase de agentes son los siguientes :

- Puertos de entrada considerados en la interfaz de esa clase de agentes.
- Puertos de salida considerados en la interfaz de esa clase de agentes. Se incluyen los arreglos de puertos de salida definidos en la interfaz.
- Enteros para definir en ejecución el número de elementos de los arreglos de puertos de salida que se van a utilizar (estos enteros pueden depender, por ejemplo, del número total de sitios lógicos).
- El sitio lógico en el que se va a ejecutar un agente específico de esa clase.

Las **declaraciones de constantes, tipos, variables y procedimientos locales** son aquéllas necesarias para poder definir el cuerpo de intrucciones de la clase de agentes. Sin embargo, hay que tener en cuenta las siguientes observaciones :

- La declaración de tipos no incluye nuevos tipos de puertos diferentes a los declarados a nivel global, puesto que se asume que los agentes diferentes al agente inicial no crean canales.

- La declaración de variables debe incluir variables de los diferentes tipos de datos asociados a los puertos de entrada o de salida de la clase de agentes (i.e. tipos de datos asociados a las clases de mensajes enumeradas en los tipos globales de puertos correspondientes).

- Los procedimientos locales pueden incluir instrucciones de envío o difusión de mensajes pero no de recepción : en efecto, al asumir que todo agente es un proceso servidor tiene un único ciclo de atención de recepción de mensajes en su cuerpo de instrucciones, y no en sus procedimientos. Tampoco hay instrucciones de creación de canales ni de activación de agentes, puesto que estas instrucciones son de responsabilidad única del agente inicial.

El **cuerpo de intrucciones** de una clase de agentes tiene el siguiente esquema general :

```
<tratamiento inicial>;
fin :=  false;
while ¬ fin  do
 <instrucción  POLL>
```

En el <tratamiento inicial> están las instrucciones que inicializan las variables locales, y en aquéllas clases de agentes que toman la iniciativa de pedir servicios a otros agentes, se encontrarán aquí instrucciones de envío o difusión de mensajes.

La instrucción WHILE constituye el ciclo de atención de cada agente de la clase especificada : en cada ciclo se ejecuta una instrucción POLL en la cual se describe el tratamiento para cada clase de mensaje recibido por cada puerto de entrada del agente. En la instrucción POLL se pueden enumerar primero las clases de mensajes correspondientes al puerto de entrada al cual se le quiere dar prioridad de atención sobre los demás. Las condiciones sobre las recepciones permiten establecer otros criterios de selección de un mensaje entre varios presentes en la cola asociada a un puerto de entrada.

Como regla general en la instrucción POLL del ciclo de atención de una clase de agentes debe figurar la recepción de todas las clases de mensajes asociadas a los tipos de puertos de entrada que figuran en la interfaz de esa clase de agentes.

En cada uno de los tratamientos de la instrucción POLL del ciclo de atención va una secuencia de instrucciones de los siguientes tipos :

- asignación a variables locales
- invocación de procedimientos locales
- instrucción de control de *PASCAL* como *if* , *while* , etc.
- instrucción para enviar o difundir un mensaje

Gracias a la posibilidad de expresar condiciones sobre las recepciones de la instrucción POLL, no es necesario considerar recepciones simples u otras instrucciones POLL dentro de los tratamientos del ciclo de atención. Tampoco deben figurar instrucciones de creación de canales ni de activación de agentes (que son responsabilidad del agente inicial).

La variable booleana " fin " permite eventualmente terminar la ejecución de un agente cuando asume el valor *false* por efecto de uno de los tratamientos de la instrucción POLL. La claúsula opcional ELSE TIMEOUT de la instrucción POLL permite especificar el tratamiento alterno que debe ejecutar un agente en caso de que no pueda recibir ningún mensaje dentro de cierto período máximo de tiempo (representado por cierto número máximo de ciclos asociados al POLL).

A continuación se muestra un **ejemplo de especificación de la clase de agentes D** (Despachador) correspondiente a las interfaz mostrada en la figura 4 :

```
AGENT D (misitio : integer;
         ent     : in tptoD;
         control : in tptocontrol;
         sal     : out tptoS);

CONST
    max          = 10000;
```

```
VAR
    regx   : tx;
    regy   : ty;
    cta    : integer;
    opcion : char;
    exito  : boolean;
    fin    : boolean;
    monto  : real;

< declaración del procedimiento menu >

PROCEDURE solicitar_servicio;
BEGIN
menu (opcion);
case opcion of
  'C': begin
          read(regx.cuenta, regx.cantidad);
          sal ¡ consignar(regx)
       end;
  'R': begin
          read(regx.cuenta, regx.cantidad);
          sal ¡ retirar(regx)
       end;
  'S': begin
          read(regy.cta);
          sal ¡ saldo(regy)
       end
end
END;

BEGIN (* cuerpo de D *)
fin := false;
regx.sitio := misitio;
regy.sitio := misitio;
solicitar_servicio;
while ¬ fin do
    poll
      control ¿ reactivacion      -->
          writeln('servicio del sistema nuevamente reestablecido');
          solicitar_servicio
    / ent ¿ rta_operacion (exito) -->
          if exito
              then writeln('operación efectuada')
              else writeln ('la operación no se puede realizar ')
          ;
          solicitar_servicio
    / ent ¿ rta_saldo (monto)     -->
          writeln ('el saldo es ', monto);
          solicitar_servicio
      else timeout (max)          -->
          writeln ('no hay servicio por fallas en comunicación')
    end
END;
```

De acuerdo al cuerpo de intrucciones dado en el ejemplo, un agente **D** supone el aislamiento del sitio lógico del agente **S** después de no recibir ningún mensaje en un periodo de 10000 ciclos de su

instrucción POLL. En ese momento informa el hecho a los usuarios y no vuelve a ofrecerles servicios hasta no recibir la señal de control "reactivacion" que indica la superación de la falla.

4.4 ETAPA 4 : CREACION DE CANALES, ACTIVACION Y CONEXION DE AGENTES

En la última etapa de la metodología de programación JOYCE+ se especifica completamente el agente inicial, el cual está encargado de crear todos los canales del sistema y activar todos los agentes conectándolos a los canales. Con esta especificación queda completo el programa en JOYCE+ que describe el sistema distribuido.

La estructura del programa, como se planteó en la parte 3 de este artículo, es la siguiente :

```
program < nombre del programa >;
 < declaración de constantes globales >
 < declaración de tipos globales >
 < declaración del agente inicial >
end .
```

Las declaraciones de constantes y tipos globales (de puertos y de datos asociados a las clases de mensajes) son aquéllas especificadas en la Etapa 2 de la metodología JOYCE+

La declaración de la clase del agente inicial (de la cual solo va a activarse una instancia) tiene el siguiente esquema general :

```
agent  < nombre de la clase inicial >;
  < declaración de clases de agentes >
  < declaración de variables locales >
  < declaración de procedimientos locales >
begin
  < cuerpo de instrucciones >
end
```

La **declaración de clases de agentes** corresponde a las especificaciones hechas en la Etapa 3 de la metodología JOYCE+ .

La **declaración de variables locales** corresponde básicamente a las variables puertos de salida que el agente inicial va a conectar a los canales que va a crear : hay que prever una por cada canal que aparece en la estructura lógica del sistema. También se pueden considerar otras variables locales necesarias para expresar el cuerpo de instrucciones del agente inicial y de sus procedimientos locales.

El **cuerpo de instrucciones** del agente inicial, combinado con las invocaciones a procedimientos locales, debe cumplir con los siguientes 2 objetivos principales :

- Crear todos los canales del sistema distribuido conectándolos a las variables puertos de salida del agente inicial.

- Activar todos los agentes del sistema distribuido conectándolos a los canales de acuerdo a la estructura lógica establecida en la Etapa 1 de la metodología JOYCE+ . Al activar un agente específico, el agente inicial le debe pasar como parámetros efectivos el valor de sus variables puertos conectadas a los canales que el nuevo agente, los enteros necesarios para dimensionar correctamente los arreglos de puertos de salida, y el sitio lógico donde se va a ejecutar el nuevo agente.

Para lograr estos objetivos se utilizan las instrucciones de JOYCE+ de creación de canales y de activación de agentes, las cuales deberán aparecer en el cuerpo de intrucciones y/o en los procedimientos locales del agente inicial.

La asociación de los sitios lógicos a los sitios físicos de la red de comunicación no se establece dentro del programa sino que constituye información del entorno del programa que debe conocerse en el momento de ejecución del sistema distribuido (por ejemplo, a través de un catálogo).

A continuación se presenta la especificación del agente inicial del ejemplo de sistema distribuido presentado en las secciones anteriores (su estructura lógica se presentó en la figura 3 y las interfaces de sus clases de agentes en la figura 4) :

```
AGENT INICIAL;
    AGENT D ( misitio : integer  ;
              ent     : in tptoD;
              control : in tptocontrol;
              sal     : out tptoS );
    ..............
    BEGIN
    ...........
    END;

    AGENT S ( misitio : integer;
              numD    : integer;
              ent     : in tptoS  ;
              sal     : out tarrptoD   ;
              cont    : out tarrptocontrol );
    ..............
    BEGIN
    ...........
    END;
VAR
    ptoentS      : tptoS;          (* variables de INICIAL *)
    ptoentD      : tarrptoD;
    ptocontrolD : tarrptocontrol;
    numD, i      : integer;
BEGIN (* de INICIAL *)
read (numD);
+ ptoentS;
for i := 1 to numD do begin
    + ptoentD[i];
    + ptocontrolD[i]
end;
S (2, numD, ptoentS, ptoentD, ptocontrolD ) at 2;
for i := 1 to numD do
    D (1, ptoentD[i], ptocontrolD[i], ptoentS ) at 1
;
END.
```

Teniendo en cuenta que el agente inicial tiene puertos de salida hacia todos los canales del sistema, puede utilizarlos para enviar a los demás agentes señales de control. Por ejemplo, para indicar la finalización del sistema distribuido, el agente inicial puede enviar una señal "fin" a cada agente con el objetivo de que finalice su ciclo de atención (lo cual logra cada agente asignando *true* a la variable fin dentro del tratamiento asociado a la recepción de la señal "fin"). De igual manera, el agente inicial puede enviar una señal de "reinicio" a cada agente cuando el sistema distribuido recomienza su actividad después de una interrupción : en este caso, cada agente debe tener previsto

un tratamiento de recuperación asociado a la recepción de esta señal (si es un solo sitio lógico el que recomienza su actividad, la señal "reinicio" debería enviarse solo a los agentes de ese sitio).

5. PROYECTOS RELATIVOS A **JOYCE+**

Actualmente hemos emprendido varios proyectos de software para construir un AMBIENTE DE PROGRAMACION BASADO EN EL MODELO Y LENGUAJE JOYCE+ :

- En el proyecto "**Subsistema de Comunicación para el Ambiente JOYCE+ en DECNET** " *[Torres 88]* se está desarrollando un susbsistema de comunicación basado en el modelo JOYCE+ sobre una red DECNET en la cual están conectados un equipo *Microwax - II* con sistema operacional *UNIX* y otro equipo *Microwax - II* con sistema *VMS* . El susbsistema trabaja sobre 3 catálogos que describen la estructura lógica de un sistema distribuido en el modelo JOYCE+ :

  a) Catálogo de agentes : Indica para cada instancia de agente el nombre de su clase, el sitio lógico de ejecución, el número de instancia (dentro de su clase en ese sitio), el nombre del programa que contiene su secuencia de intrucciones y el identificador del proceso asociado en la red DECNET.

  b) Catálogo de canales : Indica para cada canal del sistema distribuido el agente y el puerto destino.

  c) Catálogo de sitios : Indica para cada sitio lógico el sitio físico asociado y opcionalmente un nombre lógico para la aplicación.

  Con estos catálogos, que deben estar replicados en todos los sitios lógicos del sistema, el Subsistema de Comunicación asegura servicios de comunicación asincrónica entre agentes locales o remotos cuando dichos servicios son solicitados a través de las instrucciones de JOYCE+ para enviar o difundir un mensaje. Los mensajes son colocados en colas de recepción asociadas a los puertos de entrada de los agentes y se asume que la recepción misma es responsabilidad de los agentes.
  En el proyecto " **Traductor JOYCE+ para el Ambiente JOYCE+ en DECNET** " se desarrollará un pseudo-compilador capaz de traducir un programa fuente en lenguaje JOYCE+ en un conjunto de programas en lenguaje C que puedan ejecutarse sobre la red DECNET utilizando los servicios del Subsistema de Comunicación JOYCE+ descrito arriba. El conjunto de programas generados deben lograr los objetivos del sistema distribuido descrito por el programa fuente.

  A grandes rasgos el Traductor JOYCE+ generará un programa en lenguaje C por cada clase de agentes descrita en el programa fuente. Las instrucciones de creación de canales y de activación de agentes se deben traducir en generación de entradas de los catálogos de canales y de agentes. El conjunto de programas generados se deben compilar y estar disponibles en cada uno de los sitios físicos de la red de comunicación (DECNET). Para poner en actividad el sistema distribuido deberá activarse el Subsistema de Comunicación JOYCE+ y ponerse en ejecución el programa correspondiente al agente inicial en cada uno de los sitios lógicos.

  Inicialmente el Traductor JOYCE+ tendrá las mismas restriciones de la Metodología de programación JOYCE+ , es decir, asumirá que solo el agente inicial es responsable de la creación de canales, activación y conexión de los demás agentes. Posteriormente se ampliará en

un nuevo proyecto para permitir que estas funciones sean ejercidas por cualquier agente e incluirá el caso de la activación recursiva de agentes.

- En el proyecto " **Traductor JOYCE+ para el Ambiente JOYCE+ en PC-NET** " se replicarán los mismos objetivos de los 2 proyectos anteriores para una red PC-NET de microcomputadores PC compatibles. En este caso se involucrarán las funciones del Subsistema de Comunicación para Ambiente JOYCE+ (primer proyecto) dentro de los programas generados por el Traductor. De hecho, solo se debe generar un programa para replicarlo en cada microcomputador, el cual debe simular el procesamiento paralelo de los agentes de cada sitio.

- El proyecto " **TESEO : Ambiente de Programación de Sistemas Transaccionales Distribuidos** " *[Franky 87]* tiene como objetivo montar un Sistema Manejador general de Bases de Datos Distribuidas que permita automatizar la programación de sistemas transaccionales distribuidos, asegurando aspectos de atomicidad, recuperación y control de concurrencia de transacciones distribuidas. Nuestro objetivo es dejar una versión definitiva de TESEO sobre la red DECNET de equipos *Vax* (mencionada en el primer proyecto) y para ello planeamos programar TESEO como un sistema distribuido siguiendo el modelo, lenguaje y metodología JOYCE+ . Una vez expresado TESEO a través de un programa en lenguaje JOYCE+ utilizaremos el Traductor JOYCE+ (para Ambiente JOYCE+ en DECNET) para generar los programas definitivos en lenguaje C.

Los anteriores son los proyectos de software relativos a JOYCE+ con los cuales estamos comprometidos a corto, mediano y largo plazo (en nuestro grupo de investigación SINBAD). Todos pertenecen al *área de los Sistemas Distribuidos que operan sobre redes de comunicación.* Sin embargo, existen otras áreas en las cuales se podría explorar la aplicabilidad del modelo y lenguaje JOYCE+, como son las siguientes :

- La programación dirigida por eventos *[Green 86, Hill 86]* típica de las máquinas *Macintosh* y de aquéllas que presentan al usuario interfaces de íconos, ofrece muchas analogías con la programación de un sistema distribuido. En efecto, los dispositivos fuentes de eventos como son el teclado, el ratón, el disco, etc. se pueden considerar agentes que se ejecutan en paralelo y los eventos mismos se pueden considerar mensajes (pertenecientes a clases bien definidas) que son enviados de forma asincrónica al control del programa. Este último, a su vez, se puede ver como un agente "central" que recibe los eventos sacándolos de una cola de recepción, los trata y genera nuevos eventos hacia los otros agentes, quienes finalmente realizan las acciones (tratamientos) que el usuario ve reflejadas en la pantalla. Con todas estas analogías se podría pensar en expresar un programa para este tipo de máquinas en lenguaje JOYCE+ y luego usar un Traductor JOYCE+ específico que generara un programa ejecutable en la máquina. El objetivo de desarrollar tal Traductor sería el de reducir la complejidad de la programación dirigida por eventos ofreciéndole al programador la posibilidad de trabajar sobre el modelo abstracto JOYCE+ .

- En la programación orientada a objetos *[Cox 86]* los objetos que se programan se pueden ver como procesos servidores y podrían expresarse como agentes JOYCE+ . Cada clase de objetos se podría expresar como una clase de agentes con una interfaz bien definida en la cual se tendría un único puerto de entrada para recibir mensajes y varios puertos de salida para enviar mensajes a otros objetos. Los mensajes representarían peticiones de servicios y respuestas a los mismos. El cuerpo mismo de una clase de objetos contendría un ciclo único de atención. Un programa orientado a objetos se puede ver como un sistema distribuido de instancias de objetos que se comunican de forma asincrónica y, en términos generales, en su programación se podrían seguir las Etapas de la Metodología JOYCE+ extendiéndola para considerar la composición de objetos (lo cual corresponde en JOYCE+ a declarar y utilizar clases de objetos dentro de otras clases de objetos). También aquí podría ser útil un Traductor JOYCE+

específico que ofrezca al programador la posibilidad de trabajar sobre el modelo abstracto JOYCE+ esperando así reducir la complejidad de la programación.

- En la programación de sistemas distribuidos para control industrial se podrían utilizar el modelo y el lenguaje JOYCE+ extendiéndolos para manejar tiempo real, lo cual implicaría poder manejar "timeouts" reales, pausas y en general eventos asociados a los relojes de los agentes que componen un sistema distribuido. Se necesitarían entonces traductores JOYCE+ específicos para el tipo de máquinas y de redes de comunicación que se usan en esta área.

- En los problemas de procesamiento paralelo en una sola máquina el lenguaje JOYCE original *[Hansen 87]* presenta grandes ventajas respecto a otros lenguajes generalmente utilizados en esta área. Estas ventajas fueron presentadas en la sección 3.11 de este artículo. En esta área no se justifica utilizar la comunicación asincrónica de JOYCE+ pues en una sola máquina resulta más eficiente la comunicación sincrónica "rendez-vous" de JOYCE, la cual no requiere buffers intermedios. Para utilizar el lenguaje JOYCE en esta área es necesario disponer del traductor (o compilador) asociado : ya existe una versión de compilador JOYCE para un PC el cual mide únicamente 3000 lineas de código fuente y que fue desarrollado por el autor del lenguaje *[Hansen 87b]* .

6. CONCLUSIONES

En este artículo hemos presentado un modelo, un lenguaje y una metodología de programación de sistemas distribuidos que hemos denominado JOYCE+ . Con ellos esperamos facilitar la especificación y programación de sistemas distribuidos que se quieren montar sobre redes de comunicación. El modelo y el lenguaje presentan como principales virtudes la simplicidad, claridad y modularidad, las cuales permiten al programador trabajar a un nivel completamente transparente al ambiente específico de red en el cual va a montar su sistema distribuido. Además proponemos una metodología de programación que indica cómo describir y programar sistemas distribuidos que siguen el enfoque de "procesos servidores".

En primera instancia, el programador tendría que transformar su programa expresado en JOYCE+ a programas ejecutables sobre la red de comunicación específica sobre la cual quiere montar su sistema distribuido. Pero a largo plazo, podremos ofrecer traductores JOYCE+ que eviten al programador tener que conocer cómo se invocan los servicios de comunicación específicos que ofrece la red.

El trabajo que hemos desarrollado alrededor de JOYCE+ tuvo su origen en nuestro conocimiento del lenguaje JOYCE definido por Brinch Hansen *[Hansen 87]* , el cual presenta aspectos muy novedosos respectos a otros lenguajes que son utilizados para programar sistemas distribuidos en una sola máquina.

El desarrollo de ambientes generales basados en JOYCE+ será de gran utilidad para dominar la complejidad que representa la especificación y programación de sistemas distribuidos sobre redes de comunicación. Por otra parte, estos ambientes permitirán construir otros ambientes más sofisticados como son aquéllos relativos a proyectos de Sistemas Manejadores de Bases de Datos Distribuidas (por ejemplo, el proyecto TESEO *[Franky 87]* o el proyecto FCCC en sus etapas futuras *[Abásolo 87]*).

BIBLIOGRAFIA

[Abálsolo 87] : José Abásolo ,
"Protocolos de conversión para el intercambio de información entre Bases de Datos bibliográficas", XIV Conferencia Latinoamericana de Informática (CLEI), Bogotá - Colombia, Nov. 1987.

[Alford 85] : M. W. Alford, J.P. Ansart et al.,
"Distributed Systems", Lecture Notes in Computer Science Nº 190 (capítulo 6), Springer-Verlag, 1987.

[Cox 86] : Brad J. Cox,
"Object oriented programming : an evolutionary approach", Addison Wesley Publishing Co., 1987.

[Franky 87]: M. Consuelo Franky,
"TESEO : Ambiente de programación de sistemas transaccionales distribuidos", XIV Conferencia Latinoamericana de Informática (CLEI), Bogotá - Colombia, Nov. 1987.

[Green 86] : Mark Green,
"A Survey of three Dialogue Models", ACM Transactions on Graphics, Vol. 5 Nº 3, July 1986.

[Hansen 87] : Per Brinch Hansen,
"JOYCE : A programming Language for Distributed Systems", Software - Practice and experience, Vol. 17 Nº 1, Jan. 1987.

[Hansen 87b] : Per Brinch Hansen,
"A JOYCE Implementation", Software - Practice and experience, Vol. 17 Nº 4, April 1987.

[Hill 86] : R. D. Hill,
"Supporting concurrency, communication and synchronization in Human-Computer Interaction - The Sassafras UIMS", ACM Transactions on Graphics, Vol. 5 Nº 3, July 1986.

[Hoare 78] : C.A.R. Hoare,
"Communicating Sequential Processes", Comm. ACM, Vol. 21 Nº 8, Aug. 1978.

[Jones 86] : G. Jones,
"Programming in OCCAM", Prentice-Hall, 1986.

[Maekawa 87] : M. Maekawa, A. Oldehoeft, R. Oldehoeft,
"Operating Systems : advanced concepts" (capítulo 3), The Benjamin/Cummings Publishing Company, 1987.

[Sloman 87] : M. Sloman, J. Kramer,
"Distributed Systems and Computer Networks" (capítulos 3 y 4), Prentice-Hall, 1987.

[Torres 87] : Julio A. Torres,
"Subsistema de Comunicación para un ambiente de procesamiento distribuido basado en JOYCE+ ", tesis de pregrado en Ingeniería de Sistemas ISC-88-I-31, Universidad de los Andes, Sept. 1988.