050653

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Unidad Académica de los Ciclos Profesional y de Posgrado del Colegio de Ciencias y Humanidades

Desarrollo e Implantación de Nuevos Mecanismos para la Comunicación y Sincronización en OCCAM y TRANSPUTERS

T E S I S

que para obtener el grado de

MAESTRO EN CIENCIAS DE LA COMPUTACION

p r e s e n t a

MANUEL ROMERO SALCEDO

Director: Dr. Víctor Germán Sánchez Arias

México, D.F. 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Contenido

Lista de Figuras

Lista de Tablas

Introducción

"Entities should not be multiplied beyond necessity"
William of Occam

Las primeras generaciones de computadoras estuvieron basadas en el modelo arquitectónico de *von Neumann*, es decir, consistían de dispositivos de entrada y salida, una memoria que almacenaba datos e instrucciones, una unidad de control que interpretaba las instrucciones y una unidad aritmético-lógica para procesar los datos. El enfoque de procesamiento de esta configuración era estrictamente *secuencial*, debido a que siempre una instrucción tenía que finalizar antes de que la próxima pudiera iniciar su ejecución.

La computación ha tenido, a lo largo de su historia, un desarrollo vertiginoso que puede ser caracterizado a través de dos vertientes:

- *Hardware* (arquitectura de computadoras).
- *Software* (lenguajes de programación).

Este desarrollo jamás se ha quedado estancado; por el contrario, la necesidad de un mayor poder y velocidad en el proceso de cómputo; el costo cada vez mayor para la creación de procesadores secuenciales más veloces; y el requerimiento que exigía poder dar solución a problemas que involucraban la ejecución de un conjunto de actividades de manera simultánea, son sólo algunas de las razones que dieron auge al surgimiento de las *arquitecturas paralelas*, las cuales involucran una forma diferente de procesamiento, y por tanto, una nueva forma para su programación.

Para programar un sistema con arquitectura paralela se puede pensar básicamente en dos alternativas:

- A partir del texto de un programa secuencial, el compilador genera un programa paralelo.
- A través de un lenguaje que incluya mecanismos explícitos para la expresión abstracta del paralelismo.

Este trabajo centra su interés en esta segunda alternativa. Se utiliza una arquitectura paralela de memoria distribuida basada en un procesador del tipo VLSI[1] conocido como *Transputer* [24]. Éste se caracteriza por su alto desempeño y gran velocidad, permitiendo el procesamiento paralelo a través de su propio hardware. Además puede ser utilizado como unidad de proceso para crear diversas topologías, obteniendo de esta manera sistemas de procesamiento paralelo y distribuido cada vez más complejos.

Con la finalidad de explotar al máximo el poder del paralelismo, el Transputer se programa a través de un lenguaje simple, elegante y poderoso, el cual representa su nivel *ensamblador*, su nombre, *OCCAM* [22]. Se trata de un lenguaje de programación basado en el concepto de ejecución paralela, además de la secuencial, el cual permite la expresividad del paralelismo de forma inherente en la solución de un problema. La sencillez de su semántica para la expresión del paralelismo hace que la cooperación y coordinación de los diferentes componentes (procesos) de una aplicación sea muy simple.

Tanto la arquitectura Transputer como el lenguaje OCCAM están basados en un lenguaje de especificación formal del paralelismo llamado *Procesos Secuenciales Comunicantes*[2] o CSP[3][17][18], el cual provee una metodología formal orientada a la modelación, especificación, análisis y verificación de algoritmos paralelos y distribuidos.

Las aplicaciones paralelas y distribuidas comprenden múltiples procesos ejecutándose paralelamente sobre diferentes procesadores. Los procesos cooperan, se coordinan e interactúan entre sí con la finalidad de conseguir un objetivo. *En OCCAM-Transputer* se ofrece un mecanismo básico para el control de la simultaneidad, cooperación y coordinación *(comunicación y sincronización)* de un conjunto de procesos en ejecución paralela, a saber: el *canal*. Se trata de un mecanismo que provee una vía de comunicación entre dos procesos, cada uno de los cuales accesa al canal a través de alguna de las dos operaciones primitivas incorporadas en el lenguaje, el *envío* y la *recepción*, estableciendo así la comunicación síncrona de mensajes.

[1] *"Very Large Scale Integration"* o Integración a Muy Grande Escala.

[2] Por esta razón, en lo subsecuente se relacionará a *OCCAM-Transputer* como la pareja lenguaje-arquitectura.

[3] Acrónimo de *"Communicating Sequential Processes"* del término en Inglés.

Ahora bien, los modelos de solución de la mayoría de las aplicaciones paralelas y distribuidas complejas involucran el uso de mecanismos más elaborados, los cuales deben ofrecer patrones, estructuras o esquemas de comunicación y sincronización de más alto nivel que permitan precisamente resolver los problemas complejos de interacción entre los procesos. Por lo tanto, el programador de este tipo de aplicaciones, usando OCCAM y Transputers, se enfrenta generalmente al problema de tener que invertir tiempo y esfuerzo en la programación de los diferentes mecanismos, que le permitirán obtener una comunicación y sincronización adecuada a los requerimientos de sus modelos de solución. Lo ideal sería entonces, invertir mejor esos recursos en la búsqueda de una solución óptima del problema, a partir de mecanismos de comunicación y sincronización más completos. De otra forma, existirá siempre la necesidad de un esfuerzo mayor, por parte del programador, para conseguir este objetivo con éxito.

Si bien el *canal* constituye, por lo tanto, un mecanismo de comunicación de bajo nivel, es también un mecanismo general a través del cual es posible la construcción de mecanismos más elaborados, es decir, de mayor nivel de abstracción. En este trabajo se presenta el diseño y la implantación de nuevos mecanismos (de más alto nivel) para la comunicación y sincronización de procesos, los cuales proporcionarán al programador de aplicaciones una herramienta de apoyo en el momento de la programación, brindando de esta manera mayor versatilidad, flexibilidad y eficiencia en el desarrollo de sistemas paralelos y distribuidos complejos sobre una red de procesadores Transputer.

Algunas ideas entorno a este problema se pueden observar en las propuestas de [40] [19] [46] [52]. La propuesta presentada en este trabajo utiliza todas las características y elementos incorporados en el lenguaje OCCAM para la construcción de los nuevos mecanismos, los cuales se basan en otros mecanismos implantados en diferentes lenguajes de programación paralela y distribuida [50] [45] [1]. Además, se ha considerado importante la construcción de una *biblioteca de software*, la cual permita reunir a los nuevos mecanismos, enriqueciendo de esta manera al propio lenguaje. El trabajo ha sido organizado de la siguiente forma:

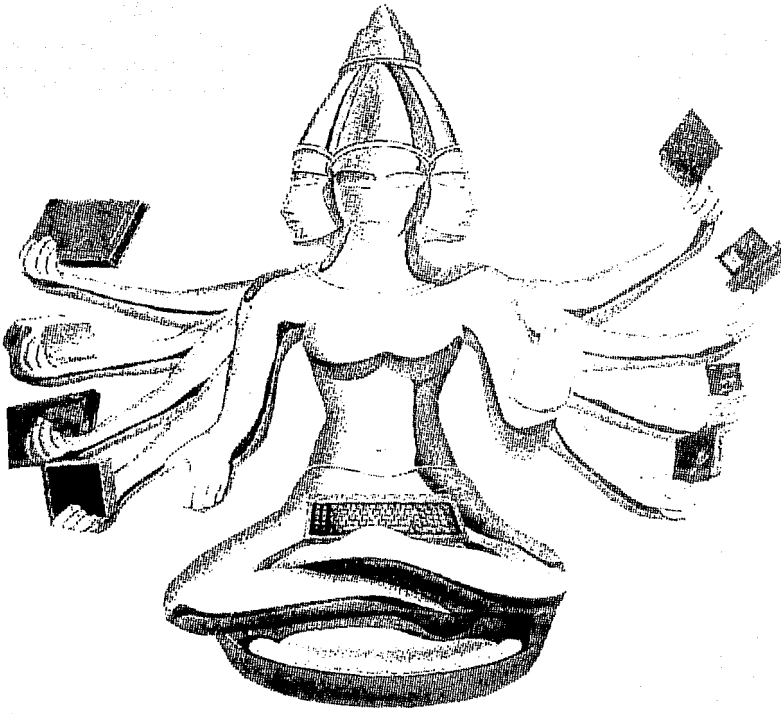
En el primer capítulo se introducen los conceptos básicos necesarios para el estudio de la programación y el procesamiento paralelo y distribuido, lo cual conforma el contexto en el que se desarrolla este trabajo. Asimismo, se presenta una clasificación tanto de arquitecturas paralelas como de lenguajes de programación paralelos en la cual se sitúa al lenguaje OCCAM y a la arquitectura Transputer.

En el segundo capítulo se describe, de manera general, el lenguaje de especificación formal de algoritmos paralelos CSP, el cual conforma la base de donde nace y fundamenta tanto el lenguaje OCCAM como la arquitectura Transputer. Se da un interés especial al estudio de la comunicación y la sincronización entre procesos.

En el tercer capítulo se especifican tanto el lenguaje OCCAM como la arquitectura Transputer. Se realiza una descripción de este lenguaje, dando mayor énfasis al estudio y análisis del mecanismo básico que se integra para la comunicación y sincronización entre procesos. Asimismo, en este capítulo se examina al procesador Transputer, el cual constituye el equipo de cómputo en paralelo. Se analiza también con detalle la manera de llevar a cabo la comunicación y sincronización entre procesadores Transputer.

En el cuarto y último capítulo se consideran las propiedades analizadas y examinadas de OCCAM-Transputer para hacer una crítica a esta pareja lenguaje-arquitectura, lo cual permitirá justificar el objetivo del trabajo. Enseguida se realiza la construcción de los nuevos mecanismos para la comunicación y sincronización, describiendo el proceso de análisis y especificación, diseño conceptual, codificación e implantación de cada uno de ellos. Asimismo, se ilustran las ventajas y propiedades de los mecanismos creados, resolviendo algunos de los problemas clásicos encontrados en programación paralela y distribuida, los cuales son ejecutados en un Transputer y en una red de ellos. Finalmente, se describe la construcción de una biblioteca de software, la cual reune todos estos mecanismos.

A manera de conclusión se dan a conocer los resultados obtenidos en la implantación, los alcances logrados con la utilización de los nuevos mecanismos de comunicación y sincronización y las perspectivas de desarrollo.



Capítulo 1

Conceptos Básicos

En este capítulo se introducen los conceptos básicos necesarios para el estudio de la programación y el procesamiento paralelo y distribuido, lo cual conforma el contexto en el que se desarrolla este trabajo. Asimismo, se presenta una clasificación tanto de arquitecturas paralelas como de lenguajes de programación paralelos en la cual se sitúa al lenguaje OCCAM y a la arquitectura Transputer.

1.1 Programación y Procesamiento Paralelo y Distribuido

La historia de la *programación paralela o concurrente*[1] se debe principalmente a los grandes desarrollos en hardware y a la respuesta de distintos cambios tecnológicos. Quizá los primeros ejemplos más significativos de programas concurrentes sean los *sistemas operativos* [8], debido a la solución que ofrecían, en razón a la gran diferencia de velocidad que ha existido entre el procesador y sus periféricos. Estos sistemas poseían en su organización interna procesos que controlaban los recursos de la computadora, así como procesos que ejecutaban las tareas del usuario.

A esta forma de implantar el procesamiento en la cual un sistema de un sólo procesador permite la ejecución de un conjunto de procesos, uno a la vez, dividiendo su tiempo entre cada uno de ellos, se le conoce como *multiprogramación*.

Los grandes avances tecnológicos y la notable reducción del costo y tamaño de los procesadores dieron origen a los sistemas multiprocesadores y multicomputadoras[2].

De la gran variedad de estos sistemas que existen se encuentran básicamente aquellos en donde los procesadores:

[1]Se le llamará *programación concurrente* de acuerdo a la mayoría de los autores, cuando se desee hacer referencia a la ejecución sobre un sólo procesador.

[2]En la mayor parte de la literatura, a estos sistemas se les conoce también como *arquitecturas de memoria distribuida* o *sistemas de cómputo distribuido*.

- *Comparten una memoria común* (multiprocesadores). En este tipo de sistema, la ejecución de los procesos se realiza sobre procesadores separados, los cuales se comunican accesando en forma coherente una memoria única[3]. A esto se le conoce como *multiprocesamiento.*

- *Poseen su propia memoria local* (multicomputadoras). En este tipo de sistema, la ejecución de los procesos se realiza sobre procesadores separados, llamados comúnmente *nodos*, los cuales poseen su propia memoria local y se comunican a través de mecanismos de paso de mensajes sobre una red de comunicación. A esto se le conoce como *procesamiento distribuido.*

Cualquiera que sea el tipo de sistema ya sea multiprocesador o multicomputadora siempre se deberá contar con alguna forma de poder programarlo. De esto se encarga precisamente la *programación paralela.* Muchos autores la definen de distintas maneras; sin embargo, todos ellos coinciden con la siguiente definición:

> La *programación paralela* es una técnica que permite expresar tanto el comportamiento simultáneo e interdependiente como la cooperación y coordinación de un conjunto de acciones, actividades, tareas o procesos[4], los cuales tienen como finalidad la realización de un objetivo común.

De esta manera, un programa paralelo se puede definir a través de:

- Un conjunto de dos o más procesos autónomos que se ejecutan en forma simultánea.

- La cooperación, coordinación e interacción entre los procesos por medio de la comunicación y la sincronización.

Los objetivos fundamentales de la programación paralela son:

- *Aumentar la eficiencia del procesamiento.* El hecho de mantener ocupado al procesador mucho más tiempo consigue hacer más eficiente la ejecución, logrando incluso, el máximo aprovechamiento de su poder de rendimiento.

- *Incrementar la velocidad de respuesta.* Con el uso de varias unidades de procesamiento, cada una de ellas encargada de realizar una parte del trabajo, es posible conseguir un incremento notable en la velocidad de respuesta.

[3]Este acceso implica la modificación del estado de las variables comunes utilizando mecanismos apropiados.

[4]Se utilizará preferentemente el término *proceso* a lo largo del trabajo.

- *Expresar la solución a problemas de naturaleza paralela.* La solución de gran parte de los problemas que existen en el mundo real se traduce en una descomposición natural de actividades interdependientes, las cuales cooperan entre sí para lograr un fin común.

Por tanto, la programación paralela se utiliza principalmente para el desarrollo de sistemas:

- *operativos.*
- *de procesamiento de transacciones.*
- *de control de procesos industriales.*
- *de cómputo científico paralelo a gran escala.*
- *empotrados*[5].
- *de comunicaciones.*
- *en tiempo real.*

La programación paralela se convierte en *programación distribuida* cuando la comunicación entre los procesos se lleva a cabo a través del paso de mensajes. Este nombre surge por el hecho de que los programas son ejecutados comúnmente sobre arquitecturas distribuidas que no comparten memoria común.

En general, es posible apreciar cómo la noción de *proceso* constituye un concepto fundamental en la programación paralela y distribuida. En la sección siguiente se especifica este concepto.

1.2 Procesos

En los lenguajes de programación paralelos y distribuidos, la unidad fundamental del paralelismo es el *proceso.* De esta manera, en general, se puede decir que:

> Un *proceso* es una unidad de código autónoma, la cual se ejecuta concurrente o paralelamente junto con otros procesos en un medio ambiente. Es en sí, la abstracción de un programa en ejecución [41].

Cada lenguaje posee mecanismos especiales que permiten especificar para un conjunto de procesos, por ejemplo: su ejecución simultánea, su creación, las situaciones de cooperación y coordinación entre ellos, su terminación, etc. En las secciones siguientes se especifican, de manera general, algunos de estos mecanismos; asimismo, se describen las propiedades de seguridad de los procesos. En la sección 1.3 se analiza con detalle el tema de la cooperación entre procesos.

[5]Del término *"embedded systems"* en Inglés.

1.2.1 Especificación de Ejecución Simultánea

Para especificar sintácticamente la ejecución simultánea de procesos se han creado y propuesto diversas notaciones. Entre las más importantes están las:

- *Corrutinas* [6].
- Instrucción *"Fork"/"Join"* [53].
- Constructor *"Cobegin/"Coend"*[6].

En este trabajo se estudia un lenguaje que hace referencia a una variante [17] de esta última notación estructurada. Se tiene sintácticamente:

$$\textbf{Cobegin } P_1 \parallel P_2 \parallel \ldots \parallel P_n \textbf{ Coend}$$

Este constructor denota explícitamente la activación simultánea de los procesos $P_1, P_2, \ldots, P_n$. La terminación correcta de todos los procesos involucrados hará que el constructor *Cobegin/Coend* termine también con éxito.

1.2.2 Creación

Los procesos que existen en un programa paralelo en el momento de su ejecución, pudieron haber sido creados por cualquiera de las siguientes formas:

- *Dinámica.*[7] Se refiere a la posibilidad de crear en tiempo de ejecución aquellos procesos necesarios para el funcionamiento del programa o sistema. La creación dinámica es importante para aquellos sistemas en los cuales los requerimientos de cómputo cambian durante el tiempo. Esto trae algunas ventajas como podrían ser: mayor flexibilidad, uso de recursos de manera dinámica, manejo del balanceo de cargas, etc.
- *Estática.*[8] Se refiere al hecho de especificar y definir en el texto del programa o sistema, antes de realizarse la compilación, todos aquellos procesos que serán ejecutados. La creación estática se utiliza para aquellos sistemas en los cuales los requerimientos de cómputo implican la ejecución sobre una configuración de procesadores fija, por lo que la habilidad de predecir el desempeño se vuelve más importante que la flexibilidad. Esto trae como ventaja principal, la posibilidad de obtener mayor eficiencia en la implantación del sistema.

[6] Basada en el *"Parbegin" y "Parend"* de Dijkstra [7].
[7] Llamada también creación *explícita.*
[8] Llamada también creación *implícita.*

1.2.3 Propiedades de Seguridad

En la ejecución de un programa paralelo y distribuido, donde intervienen un conjunto de procesos, pueden originarse problemas importantes que podrían alterar el funcionamiento del sistema. De esta manera, se han establecido propiedades de seguridad para los procesos con el fin de evitar estos problemas. Estas propiedades son básicamente:

- Ausencia de *bloqueo mutuo* o *abrazo mortal*[9]. Se refiere al hecho de evitar un estado que puede suceder en el cual dos o más procesos esperan infinitamente por eventos que nunca ocurrirán.
- Proporcionar *justicia*. Se refiere al hecho de proporcionar la oportunidad de que todos los procesos ejecuten sus actividades y ninguno de ellos quede suspendido o bloqueado permanentemente.
- Asegurar *terminación*[10]. Se refiere al hecho de asegurar que todos los procesos involucrados en un programa paralelo finalicen correctamente, ya que si al menos uno de ellos queda bloqueado por alguna causa, el programa simplemente no terminará con éxito.

El desarrollo de metodologías para la especificación y verificación de programas paralelos [48] [32] [18] ha contribuido al apoyo de estas propiedades de seguridad, brindando la posibilidad de modelar y analizar formalmente los programas. Sin embargo, estos métodos se encuentran en etapa de madurez y están siendo continuamente refinados y mejorados, ya que aún no proporcionan todos los elementos necesarios para asegurar la correctitud[11] de programas paralelos y distribuidos.

1.3 Cooperación entre Procesos

Los procesos involucrados en un programa paralelo interactúan unos con otros con el fin de cooperar, coordinarse y alcanzar un objetivo. Esta *cooperación* se realiza a través de la *comunicación* y la *sincronización* entre procesos ejecutándose paralelamente.

1.3.1 Comunicación y Sincronización

A través de la *comunicación* se logra el conocimiento del estado de los procesos y permite además que la ejecución de un proceso influya en la ejecución de otro.

[9]Del término *"deadlock"* en Inglés.

[10]Sólo para aquellos problemas, cuya especificación de su solución defina la terminación en un tiempo finito.

[11]Del término *"correctness"* en Inglés.

Se establece que la comunicación entre procesos puede estar basada en:

- *Memoria compartida.*
- *Paso de mensajes.*

De esta manera, en la comunicación un dato es transferido de un proceso a otro, ya sea mediante una área de memoria compartida, la cual puede ser accesada por más de un proceso; o mediante el envío de un mensaje por un proceso y recepción por otro a través de enlaces de comunicación.

En la comunicación entre procesos se requiere frecuentemente que los procesos se sincronicen, ya que estos se ejecutan a velocidades arbitrarias e impredecibles; de esta manera, la *sincronización* se puede ver como un conjunto de restricciones sobre el orden de ejecución de los eventos. Es decir, para lograr que los procesos sigan un orden correcto de ejecución, es indispensable que cumplan con ciertas restricciones. Se trata de restricciones implantadas a través de mecanismos de sincronización apropiados, los cuales permiten bloquear la ejecución por un tiempo determinado. Algunos intentos se han realizado para formalizar estos mecanismos de sincronización de procesos, este es el caso de [11].

Enseguida se especifican los principales mecanismos de sincronización basados en *memoria compartida* y en *paso de mensajes*; sin embargo, se dará mayor interés al estudio de estos últimos, ya que este trabajo se basa en ellos.

1.3.1.1 Principales Mecanismos de Sincronización Basados en Memoria Compartida

Cuando se utiliza memoria compartida para la comunicación entre procesos se utilizan básicamente dos tipos de sincronización: *exclusión mutua* y *sincronización condicional*, los cuales han sido implantados a través de los siguientes mecanismos:

- *Espera activa*[12] [7] [31].
- *Semáforos* [8] [7].
- *Regiones críticas* [15] [12].
- *Monitores* [16] [13].

Todos estos mecanismos proveen diferentes formas estructuradas para controlar el acceso a recursos compartidos.

[12]Del término *"busy waiting"* en Inglés.

1.3.1.2 Principales Mecanismos de Sincronización Basados en Paso de Mensajes

Cuando se utiliza paso de mensajes para la comunicación entre procesos, en cierta forma la sincronización va implícita, ya que un mensaje puede ser únicamente recibido después de haber sido enviado, lo cual restringe el orden de ocurrencia de los eventos. La transmisión de mensajes se realiza comúnmente a través de las operaciones básicas de *envío* y *recepción*[13], las cuales tienen la notación general siguiente:

send *lista_de_expresiones* **to** *destino*

Esta operación especifica el envío de los valores calculados por cada una de las expresiones en *lista_de_expresiones* para ser recibidos por el proceso *destino.*

receive *lista_de_variables* **from** *fuente*

Esta operación especifica la recepción de valores enviados por el proceso *fuente*, los cuales serán asignados a cada una de las variables en *lista_de_variables*

1.3.1.3 Especificación de Mecanismos para la Comunicación

Tanto el proceso emisor como el proceso receptor definen implícitamente un canal de comunicación. Se han propuesto varios mecanismos que permiten especificar diferentes tipos de enlaces de comunicación entre procesos emisores y receptores, entre ellos se tienen principalmente los mecanismos:

- *Punto a punto.* Especifica un enlace para una comunicación de un proceso hacia otro *(uno a uno).*
- *Puertos.* Especifica un enlace para una comunicación de varios procesos hacia otro *(muchos a uno).*
- *Buzón.* Especifica un enlace para la comunicación de varios procesos hacia otros *(muchos a muchos).*

Asimismo, existen otros mecanismos más complejos para la interacción de procesos que proporcionan un mayor nivel de abstracción. Algunos ejemplos de ellos son los mecanismos:

- *"Pipeline".* Especifica una colección de procesos ejecutándose paralelamente en los cuales la salida de cada proceso está conectada a la entrada de otro.

[13]De los términos *"send"* y *"receive"* en Inglés.

- *Cliente-Servidor.* Especifica procesos *servidores* que ofrecen servicios a procesos *clientes.* Cada uno de los clientes hace la petición de un servicio, enviando un mensaje a uno de los servidores. Cada servidor recibe constantemente las requisiciones de servicios por parte de los clientes, enseguida se ejecuta el servicio, y si existe necesidad, se regresa un mensaje de terminación o un resultado al cliente.

1.3.1.4 Comunicación Síncrona y Asíncrona

La sincronización constituye una propiedad importante en los mecanismos de paso de mensajes, ya que permite establecer el orden de las secuencias de ejecución de los procesos. De esta manera, la comunicación a través de paso de mensajes puede ser:

- *Síncrona*[14]. En este tipo de comunicación, el intercambio de información se realiza como una operación atómica que requiere la participación de un proceso emisor y uno receptor al mismo tiempo. Éstos consiguen establecer la comunicación únicamente cuando ambos se encuentran listos para realizarla. En el caso de que alguno de ellos no esté listo, el proceso complementario se quedará bloqueado esperando entablar la comunicación. El acto de este tipo de comunicación sincroniza las secuencias de ejecución de los dos procesos. Por tanto, el mensaje recibido por el receptor siempre corresponde al estado actual del emisor.

- *Asíncrona.* En este tipo de comunicación, el intercambio de información se realiza como una operación atómica que requiere la participación de los procesos emisor y receptor, pero no necesariamente al mismo tiempo, ya que, por ejemplo, un proceso emisor podría hacer el envío de su mensaje sin tener después que permanecer bloqueado hasta que el proceso receptor se encuentre listo para recibirlo. Para tal fin, se posee generalmente una estructura con capacidad de almacenamiento temporal, llamada *"buffer"*[15], la cual permite guardar los mensajes en tránsito que son enviados del proceso emisor al receptor. Cuando el "buffer" posee capacidad no acotada, el proceso emisor nunca es bloqueado. En caso contrario, el proceso emisor podría quedar eventualmente bloqueado sincronizándose con el "buffer". El proceso receptor es comúnmente bloqueado, ya que no se realiza, en la mayoría de las ocasiones, ninguna actividad mientras se espera la recepción del mensaje; sin embargo, puede también existir la versión que provee la recepción no bloqueante. Debido al tipo de conexión entre los procesos no se puede asegurar que el mensaje recibido por el receptor corresponda al estado actual del emisor.

[14]Tipo *"rendezvous"* o *cita.*

[15]Este es el término en Inglés que se maneja para este concepto. En lo subsecuente se usará este término.

1.3.1.5 Comunicaciones Selectivas

Una operación de recepción bloqueante puede lograr los mismos efectos semánticos que una no bloqueante, utilizando *comunicaciones selectivas*, basadas en las *instrucciones custodiadas*[16] de Dijstra [9].

La forma general del mecanismo de comunicación selectiva es la siguiente:

$$\begin{array}{l} \textbf{if } C_1 \rightarrow I_1 \\ \Box\ C_2 \rightarrow I_2 \\ \quad \ldots \\ \Box\ C_n \rightarrow I_n \\ \textbf{fi} \end{array}$$

Un mecanismo de *comunicación selectiva* se compone de un conjunto de *custodias* C_i, cada una de ellas asociadas a un conjunto de una o más *instrucciones* I_i a ser ejecutadas, estableciendo así una estructura de selección alternativa.

Este tipo de mecanismo es muy útil cuando un conjunto de procesos desean interactuar de manera no determinística. El control al no determinismo se logra debido a que se posee la característica de la evaluación de todas las custodias de manera simultánea y la selección de aquella que tenga éxito. En el caso de que varias custodias tengan éxito al mismo tiempo, sólo una de ellas es seleccionada arbitrariamente. Una vez seleccionada la custodia se ejecuta el conjunto de instrucciones que tenga asociada ésta.

Cada custodia se forma de una o más expresiones booleanas y/o una operación de recepción de mensaje. Así pués, una operación de recepción no bloqueante se construye fácilmente, utilizando comunicaciones selectivas junto con el constructor paralelo *Cobegin/Coend* de la siguiente manera:

```
Cobegin
  if Operación_de_recepción → Lista_de_Instrucciones1 fi
  ||
  Lista_de_Instrucciones2
Coend
```

En este ejemplo se muestra como una operación de recepción bloqueante puede tener los mismos efectos semánticos que una no bloqueante, permitiendo la ejecución de la *Lista_de_Instrucciones*$_2$. Nótese que este conjunto de instrucciones no depende del mensaje que se reciba; en cambio, la *Lista_de_Instrucciones*$_1$ sólo podrá ser ejecutada hasta que la *Operación_de_recepción* tenga éxito.

[16]Del término *"guarded commands"* en Inglés.

Es importante mencionar que gran parte de los lenguajes de programación paralelos y distribuidos que han sido desarrollados hasta nuestros días poseen alguna versión implantada de este mecanismo de *comunicación selectiva*, ya que los patrones de interacción entre los procesos no siempre son determinísticos y algunas veces dependen de las condiciones en tiempo de ejecución. De esta manera, fue indispensable introducir en los lenguajes de programación un mecanismo para la expressión y el control del *no determinismo*.

En el capítulo 2, sección 2.1.3 se explica con mayor detalle la semántica del mecanismo de comunicación selectiva antes presentado.

1.4 Arquitecturas y Lenguajes Paralelos

1.4.1 Clasificación General de las Arquitecturas Paralelas

Una *arquitectura paralela* provee una plataforma para el desarrollo de soluciones en programación paralela y distribuida utilizando múltiples procesadores, los cuales cooperan para resolver algún problema a través de la ejecución simultánea [10].

Las arquitecturas paralelas se han intentado clasificar tomando en cuenta básicamente tres aspectos:

- *La relación de la memoria con los procesadores.*[17] Dentro de esta clasificación se encuentran las arquitecturas de:
  1. *Memoria Compartida.* En la figura 1.1 se muestra un esquema de una arquitectura de memoria compartida.
  2. *Memoria Local.* En la figura 1.2 se muestra un esquema de una arquitectura de memoria local.
- *El procesamiento del flujo de instrucciones y datos.* Dentro de esta clasificación se encuentran principalmente las categorías siguientes:
  1. *SIMD*[18]. Implica tener diversos procesadores autónomos ejecutando la misma instrucción sobre datos diferentes.
  2. *MIMD*[19]. Implica tener diversos procesadores autónomos ejecutando instrucciones diferentes sobre datos diferentes.

[17] Las características de las arquitecturas que pertenecen a esta categoría han sido ya explicadas en la sección 1.1.

[18] *Acrónimo de "Single Instruction, Multiple Data"* o Única Instrucción, Múltiples Datos.

[19] *Acrónimo de "Multiple Instruction, Multiple Data"* o Múltiples Instrucciones, Múltiples Datos.

- *La forma en cómo coordinan la ejecución simultánea.* Dentro de esta clasificación se encuentran principalmente las siguientes categorías:

  1. *Síncronas.* Son las que coordinan la ejecución simultánea a través de relojes globales, unidades de control central o controladores de unidad vectorial.
  2. *Asíncronas.* Son las que coordinan la ejecución simultánea a través del paso de mensajes sobre una red de interconexión o por el acceso a los datos con el uso de unidades de memoria compartida.

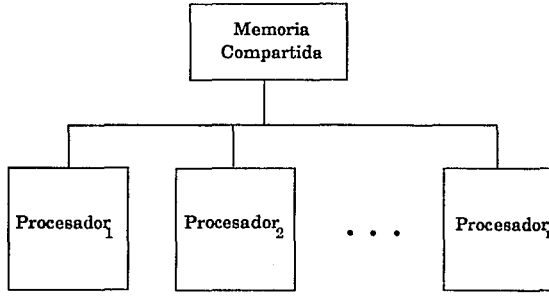


Figura 1.1: Esquema de una Arquitectura de Memoria Compartida

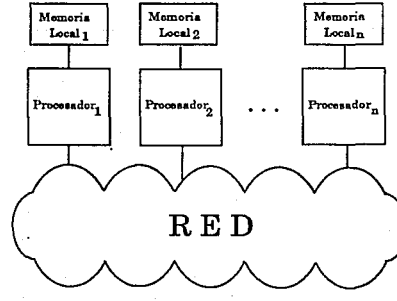


Figura 1.2: Esquema de una Arquitectura de Memoria Local

Tomando en cuenta todos estos aspectos y categorías de clasificación, la propuesta de [10], mostrada en la figura 1.3, presenta una taxonomía de arquitecturas paralelas[20] que establece una clasificación de alto nivel, ya que reune la gran diversidad de arquitecturas que han surgido hasta nuestros días, las cuales permiten el desarrollo de soluciones paralelas de alto nivel a través de la expresión abstracta del paralelismo.

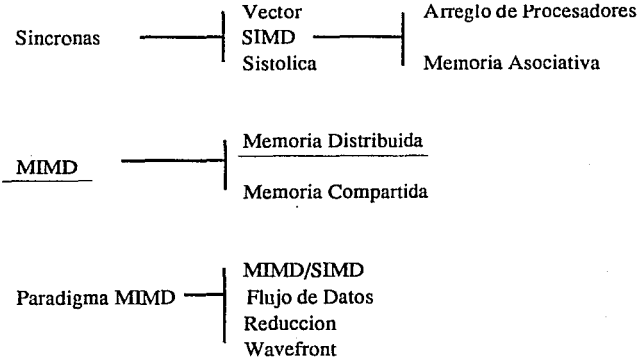


Figura 1.3: Taxonomía de Alto Nivel de Arquitecturas Paralelas

El enfoque de este trabajo se hará sobre una arquitectura paralela asíncrona de memoria distribuida basada en la categoría MIMD, ésta es la *arquitectura Transputer*. Se trata de una arquitectura, cuya filosofía establece reunir un conjunto de procesadores operando en forma autónoma y sincronizándose a través del paso de mensajes explícito. Cada nodo de procesamiento posee su propia memoria local y se conecta con otros nodos utilizando enlaces físicos[21] de comunicación punto a punto. Por todas las características anteriores se dice que este tipo de sistema distribuido es *fuertemente acoplado*[22].

En la figura 1.3 se ha señalado en donde se encuentra ubicada la arquitectura Transputer.

[20]Se sugiere al lector consultar [10] para analizar con detalle esta taxonomía.
[21]De gran velocidad y confiables.
[22]Del término *"closely coupled"* en Inglés.

1.4.2 Clasificación de los Lenguajes de Programación Paralelos

Los lenguajes de programación paralelos se clasifican por la forma en cómo habilitan la expresión paralela y permiten la construcción de programas [30].

De esta manera, se dice que los lenguajes de programación permiten la programación paralela:

- *Síncrona.* En este paradigma, a todos aquellos conjuntos de datos que no tengan dependencias entre sí, se permite que les sean aplicadas las mismas secuencias de operaciones en paralelo, restringiendo; de esta manera, que todos los procesos actúen en unísono. Los lenguajes de esta clasificación pertenecen a cualquiera de los siguientes rubros:

  1. *Detección del paralelismo por el compilador.* Se utiliza algún lenguaje de programación secuencial, siendo responsabilidad del compilador determinar cuáles partes del programa van a ser ejecutadas en paralelo.
  2. *Expresión del paralelismo usando el hardware.* Se utiliza ya sea un lenguaje, cuya sintáxis refleje directamente la arquitectura de la máquina, o uno que permita tanto la construcción de rutinas —las cuales incluyen instrucciones del hardware— como el llamado a éstas. A cualquier lenguaje en este rubro se le conoce como *lenguaje ensamblador de alto nivel.*

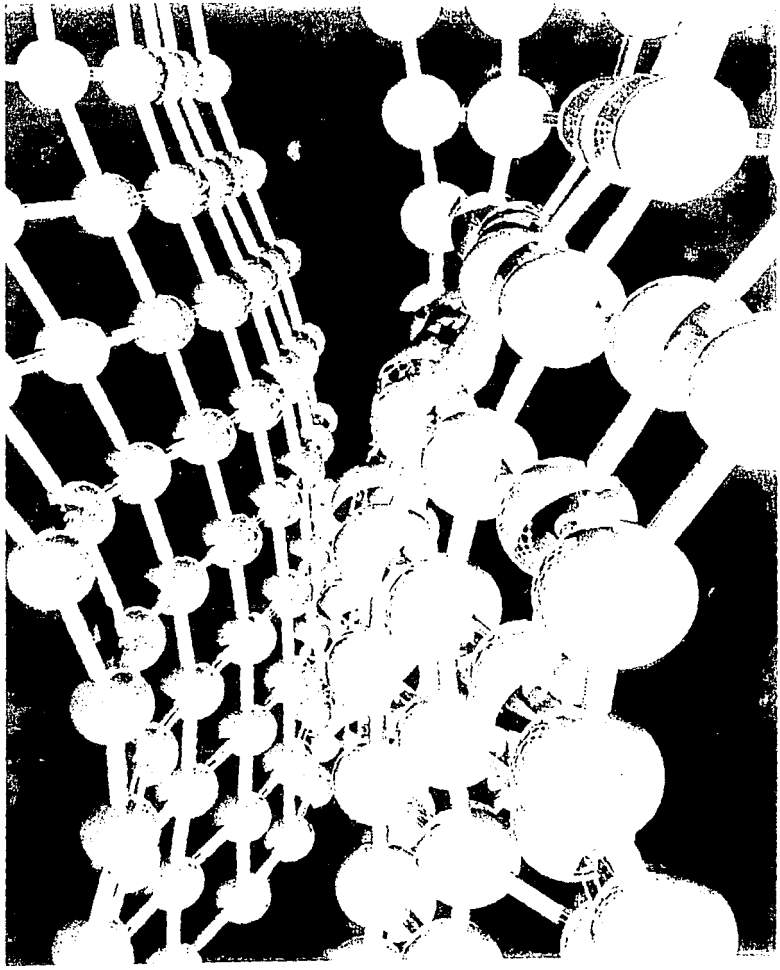
- *Asíncrona.* En este paradigma, las partes independientes de un programa (procesos) compiten y cooperan unas con otras con la finalidad de cumplir un objetivo. Los lenguajes que pertenecen a esta clasificación proporcionan mecanismos para especificar procesos y para regular las situaciones de cooperación e interacción entre ellos. Estas situaciones de cooperación entre procesos marcan la división entre aquellos lenguajes de programación paralelos que poseen mecanismos explícitos de sincronización basados en:

  1. *memoria compartida y*
  2. *paso de mensajes*

  para la comunicación entre procesos.

- *A través de flujo de datos.* En este paradigma, si hay disponibilidad de datos para varias instrucciones y no existen dependencias entre estos datos, entonces las instrucciones pueden ser ejecutadas en paralelo. Los lenguajes que pertenecen a esta clasificación ofrecen un método para determinar cuáles partes del programa serán ejecutadas en paralelo a través de la construcción de una gráfica de dependencia de datos.

El enfoque de este trabajo se hará sobre un lenguaje de programación paralelo asíncrono basado en mecanismos de paso de mensajes, éste es el *lenguaje OCCAM.* Se trata de un lenguaje, cuya filosofía establece describir un programa o sistema como una colección de procesos ejecutándose simultáneamente, los cuales se comunican y sincronizan con otros procesos a través del envío y recepción de mensajes, utilizando canales de comunicación.



Capítulo 2

El Lenguaje de Especificación Formal CSP

En este capítulo se describe, de manera general, el lenguaje de especificación formal de algoritmos paralelos CSP, el cual conforma la base de donde nace y fundamenta tanto el lenguaje OCCAM como la arquitectura Transputer. Se da un interés especial al estudio de la comunicación y la sincronización entre procesos.

2.1 Descripción de CSP

En la propuesta de *Procesos Secuenciales Comunicantes* o CSP[1] [17] [18], Hoare establece un método formal, descrito por una colección sistemática de leyes algebraicas, el cual fue creado con propósitos teóricos para la modelación, especificación, análisis y verificación de algoritmos paralelos y distribuidos, pensando ser implantados en una red de procesadores fija y de memoria distribuida.

El concepto fundamental en CSP es el proceso. Un proceso consiste de un nombre que lo identifica (P), variables locales y una lista de instrucciones (LI). En general, una instrucción[2] especifica el comportamiento de un dispositivo ejecutando esa instrucción. Por definición, una instrucción inicia su ejecución, realiza una o más acciones, y después termina con éxito o falla. La lista de instrucciones especifica la ejecución de una o más instrucciones que la constituyen, conformando así el cuerpo del proceso. De esta manera, un proceso se expresa sintácticamente de la forma siguiente:

$$P_i::LI_i$$

Se especifica el proceso llamado P_i, el cual está definido *(::)* por su lista de instrucciones correspondiente LI_i, la cual constituye el cuerpo del proceso.

[1] Acrónimo de *"Communicating Sequential Processes"* del término en Inglés.
[2] El término *instrucción* se traduce de *"command"* en Inglés.

Un programa en CSP se compone de un conjunto finito de procesos cooperantes que se comunican y sincronizan explícitamente a través del paso de mensajes síncrono, realizado por instrucciones de envío y recepción[3]. De esta manera, los procesos interactúan, cooperan y se coordinan unos con otros con la finalidad de poder resolver un problema.

El lenguaje de especificación formal CSP puede ser descrito a través de los elementos siguientes:

- *Paralelismo*
- *Comunicación y sincronización*
- *No determinismo*
- *Secuencialidad*

En las secciones siguientes se describen cada uno de estos elementos.

2.1.1 Paralelismo

2.1.1.1 Instrucción Paralela

La ejecución paralela se especifica a través de una *instrucción paralela*[4], la cual establece la activación simultánea de un conjunto de dos o más listas de instrucciones que la constituyen, es decir, de un número fijo de procesos disjuntos[5] $P_1, P_2, \ldots, P_n$, cuya velocidad relativa de ejecución es arbitraria. La instrucción paralela se expresa sintácticamente de la forma siguiente:

$$\mathbf{P_1}::\mathbf{LI_1} \parallel \mathbf{P_2}::\mathbf{LI_2} \parallel \ldots \parallel \mathbf{P_n}::\mathbf{LI_n}$$

Esta instrucción termina exitosamente sólo cuando todos los procesos involucrados hayan terminado con éxito; en caso contrario, la instrucción paralela falla. Si se hace la suposición de que ninguno de los procesos falla, entonces el tiempo que le tomará a la instrucción paralela en terminar será igual al tiempo que se tarde en terminar el proceso más lento.

CSP permite crear arreglos de procesos en donde cada proceso posee un comportamiento similar, es decir, que cada uno posee la misma lista de instrucciones. Estos procesos pueden ejecutarse en paralelo; sin embargo, para diferenciar, uno de otro, se utiliza un valor en forma de subíndice, el cual es asignado a cada proceso antes de la ejecución del programa y puede además ser utilizado dentro del proceso. De esta manera, el número de procesos siempre deberá ser establecido a priori.

[3]Hoare en su artículo [17], les llama instrucciones de *entrada* y *salida*, respectivamente.
[4]Basada en el constructor *"Parbegin" / "Parend"* de Dijkstra [7].
[5]Procesos ninguno de los cuales poseen variables comunes entre sí.

Enseguida un ejemplo para ilustrar la instrucción paralela:

[Cuarto::CUARTO || Tenedor(i:0..4)::TENEDOR || Filósofo(i:0..4)::FILOSOFO]

Se ilustra el comportamiento de solución al problema clásico de los *"Cinco Filósofos"* en el cual se especifica la ejecución simultánea de once procesos *(Cuarto, Tenedor*$_0$, ..., *Tenedor*$_4$, *Filósofo*$_0$, ..., *Filósofo*$_4$*)*, cuyos comportamiendos están descritos por sus listas de instrucciones correspondientes.

2.1.2 Comunicación y Sincronización

La comunicación y sincronización entre procesos se realiza a través de *instrucciones de envío y recepción*, las cuales establecen una comunicación síncrona. Asimismo, especifican una comunicación unidireccional entre dos procesos secuenciales *(P_1 y P_2)* ejecutándose simultáneamente. De esta manera, la comunicación ocurre siempre que:

- La instrucción de recepción en el proceso P_1 especifique como su fuente el nombre del proceso P_2.
- La instrucción de envío en el proceso P_2 especifique como su destino el nombre del proceso P_1.
- La variable de la instrucción de recepción concuerde (en el tipo de dato) con el valor calculado de la expresión en la instrucción de envío.

En otras palabras, la comunicación ocurre cuando un proceso nombra al otro como destino de un envío y el segundo proceso nombra al primero como fuente de una recepción. En este caso, el valor del envío es copiado del primer proceso al segundo sin existir la capacidad de poder almacenarlo temporalmente.

Cualquiera de las instrucciones, ya sea la de envío o la de recepción, implantada en un proceso, permanecerá bloqueada hasta que su instrucción complementaria, en el otro proceso, sea ejecutada con éxito; de esta manera, se establece una comunicación en la cual existe una simetría en la relación de sincronización, esto constituye precisamente el protocolo síncrono de comunicación entre los procesos. Por tanto, se dice que las instrucciones de envío y recepción *corresponden*. Así pues, dos procesos deben primero indicar que desean comunicarse entre sí, para después sincronizar sus actividades antes de transmitirse el mensaje. A esta forma de sincronización entre dos procesos se le conoce como *"rendezvous"*[6]. Una vez que ambos procesos han establecido la comunicación prosiguen sus ejecuciones en paralelo.

Hoare [18] expone cuatro razones principales que justifican la conveniencia de la comunicación síncrona:

[6]Del término *encuentro* o *cita* en Inglés.

- Se ajusta más al modelo físico de los enlaces que conectan a los agentes de procesamiento, los cuales no pueden almacenar mensajes.
- Se ajusta al efecto que tienen las llamadas a subrutinas y el regreso de los resultados dentro de un sólo procesador, copiando los valores de los parámetros y los resultados.
- Cuando se requiere de un "buffer", éste se puede implantar simplemente como un proceso extra, cuya capacidad de almacenamiento puede ser controlado con precisión por el programador.
- Existen ciertas aplicaciones que pueden verse perjudicadas en su desempeño de ejecución al contar con una comunicación de tipo *bufferizada*. Son principalmente aquellas aplicaciones que requieren de interacciones y respuestas rápidas en vez de una utilización más intensa del procesador y de la memoria.

Se ha podido apreciar, que las relaciones de comunicación y sincronización entre procesos son expresadas en forma explícita por el intercambio de mensajes a través de las instrucciones de envío y recepción, las cuales proveen la comunicación síncrona entre procesos.

2.1.2.1 Instrucción de Envío

Esta instrucción especifica el nombre del proceso receptor (NPR) y proporciona el valor a ser enviado, el cual puede ser una constante o un valor calculado de una expresión. La instrucción de envío se expresa sintácticamente de la forma siguiente:

NPR ! expresión

Enseguida un ejemplo para ilustrar la instrucción de envío:

Pantalla ! (dato * 7)

Se especifica el envío al proceso *Pantalla* del valor calculado por la expresión *(dato * 7)*.

La instrucción de envío falla cuando:

- El proceso receptor ha fallado.
- El proceso receptor ha terminado su ejecución sin invocar la instrucción de recepción.
- El valor de *expresión* está indefinido.

2.1.2.2 Instrucción de Recepción

Esta instrucción especifica el nombre del proceso emisor (NPE) y proporciona una variable en la cual se asigna el valor recibido. La instrucción de recepción se expresa sintácticamente de la siguiente forma:

NPE ? variable

Enseguida un ejemplo para ilustrar la instrucción de recepción:

Teclado ? dato

Se especifica la recepción de un valor enviado por el proceso *Teclado*, el cual es asignado en la variable *dato*.

La instrucción de recepción falla cuando:

- El proceso emisor ha fallado.
- El proceso emisor ha terminado su ejecución sin invocar la instrucción de envío.
- La variable recibe un valor de diferente tipo de dato del esperado.

Enseguida otro ejemplo que involucra a las instrucciones de envío y recepción:

[Formula:: Triangulo ! (base * altura) / 2 || Triangulo:: Formula ? area]

Se especifica la ejecución simultánea de dos procesos *Formula* y *Triangulo*. El proceso *Formula* contiene una instrucción de envío que invoca como destino al proceso *Triangulo*; de igual manera, este último proceso contiene una instrucción de recepción que invoca como fuente al primer proceso. Se ilustra, con este ejemplo, la manera de cómo hacer una asignación distribuida. Nótese que el resultado de este programa sería el mismo que si se ejecutara:

area := (base * altura) / 2

En la figura 2.1 se aprecia gráficamente el comportamiento de las instrucciones de envío y recepción que comunican a dos procesos. Las flechas verticales indican el flujo de ejecución con respecto al tiempo. En (a) se aprecia como el proceso P_1 es el primero en estar listo para el envío del mensaje. En (b) se aprecia como ambos procesos P_1 y P_2 llegan a estar listos al mismo tiempo para establecer la comunicación. En (c) se aprecia como el proceso P_2 es el primero en estar listo para la recepción del mensaje. Es importante mencionar que siempre se realiza primero la sincronización, y después la comunicación.

$[P_1 :: P_2 \,!\, \text{exp} \;||\; P_2 :: P_1 \,?\, \text{var}]$

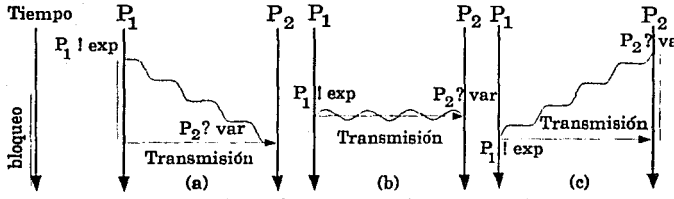


Figura 2.1: **Comportamiento de las Instrucciones de Envío y Recepción**

Un proceso ejecutando ya sea una instrucción de envío o una instrucción de recepción es bloqueado hasta que su proceso *"pareja"* haya ejecutado la instrucción complementaria. Esto involucra implícitamente la sincronización entre los procesos; sin embargo, se puede dar el caso de que alguno de los procesos termine sin haber realizado la instrucción complementaria o que el proceso falle inesperadamente, esto provocará que el bloqueo jamás se libere. De manera general, cuando un grupo de procesos esperan comunicarse, pero ninguna de sus instrucciones de envío y recepción corresponden una con otra, entonces los procesos permanecerán en *bloqueo mutuo*.

En la figura 2.2 se aprecia gráficamente el comportamiento de un conjunto de procesos en *bloqueo mutuo*. Se puede observar como cada uno de los procesos $P_1, \ldots, P_4$ esperará bloqueado por un evento que jamás ocurrirá.

$[P_1 :: P_2 \,!\, \text{exp} \;||\; P_2 :: P_3 \,?\, \text{var} \;||\; P_3 :: P_2 \,?\, \text{var} \;||\; P_4 :: P_3 \,!\, \text{exp}]$

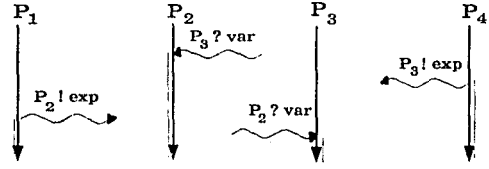


Figura 2.2: **Comportamiento de un Conjunto de Procesos en Bloqueo Mutuo**

Como se ha podido apreciar, CSP constituye un lenguaje *orientado a mensajes*, ya que provee las instrucciones de envío y recepción como únicos medios para la interacción y cooperación entre los procesos.

2.1.3 No Determinismo

CSP integra una versión de comunicaciones selectivas como un mecanismo para el control al *no determinismo*. Se trata de la *instrucción alternativa*, la cual especifica una colección de listas de instrucciones custodiadas.

2.1.3.1 Instrucción Alternativa

Esta instrucción se expresa sintácticamente de la forma siguiente:

$$
\begin{array}{l}
[\ \ \mathbf{C_1} \rightarrow \mathbf{LI_1} \\
\square\ \ \mathbf{C_2} \rightarrow \mathbf{LI_2} \\
\qquad \cdots \\
\square\ \ \mathbf{C_n} \rightarrow \mathbf{LI_n} \\
]
\end{array}
$$

Se especifica un conjunto de custodias $C_1,\ldots,C_n$ asociadas a un conjunto de listas de instrucciones $LI_1,\ldots,LI_n$, respectivamente. Una custodia consiste en una lista (que puede ser vacía) de expresiones booleanas y/o una instrucción de recepción[7].

Todas las custodias se evalúan al mismo tiempo y sólo una de las listas de instrucciones se ejecuta siempre que su custodia asociada tenga éxito. Una custodia tiene éxito cuando todas sus expresiones booleanas son verdaderas y su instrucción de recepción tiene éxito[8]. Si más de una de las custodias tienen éxito, sólo una de ellas es seleccionada arbitrariamente[9], permitiendo así la ejecución de su lista de instrucciones correspondiente. Una vez ejecutada esta lista, si ninguna de las instrucciones involucradas en ella falló, entonces la instrucción alternativa termina con éxito.

En el caso de que todas las instrucciones de recepción se encuentren bloqueadas esperando la recepción de un mensaje, la instrucción alternativa permanecerá también bloqueada hasta que alguna instrucción de envío correspondiente a una de recepción en una de las custodias llegue a estar lista. Una instrucción alternativa especifica la ejecución de exactamente una de sus listas de instrucciones custodiadas, por lo tanto, si todas las custodias fallan, la instrucción alternativa también falla.

Enseguida un ejemplo para ilustrar la instrucción alternativa:

$$
\begin{array}{l}
[\ \ \mathsf{x} \geq \mathsf{y} \rightarrow \mathsf{mayor} := \mathsf{x} \\
\square\ \ \mathsf{y} \geq \mathsf{x} \rightarrow \mathsf{mayor} := \mathsf{y} \\
]
\end{array}
$$

[7]No se permite que aparezcan instrucciones de envío en las custodias para evitar problemas de simetría durante el *"rendevous"* (una discusión sobre este tópico puede ser encontrada en [4]).

[8]La instrucción de envío correspondiente debe también ser ejecutada con éxito.

[9]Hoare establece que esta selección debe ser justa, dejando este problema a la implantación.

Se especifican dos custodias, cada una de ellas compuesta por una expresión booleana. Se ilustra una forma de obtener el valor mayor entre dos variables. Se puede observar que si $x \geq y$, entonces x es asignado a *mayor*; o si $y \geq x$, entonces y es asignado a *mayor*; o si ambas custodias tienen éxito al mismo tiempo *($x = y = mayor$)*, entonces cualquiera de las dos asignaciones puede ser ejecutada, pero sólo una de ellas.

Enseguida otro ejemplo que contempla custodias compuestas por expresiones booleanas e instrucciones de recepción:

```
[ volumen < máximo ; Q ? señal → volumen := volumen + 1
□ volumen > mínimo ; P ? señal → volumen := volumen − 1
]
```

Se ilustra un fragmento de programa necesario para la simulación del control de un dispositivo electrónico en el cual se especifican dos custodias, cada una de ellas compuestas por una expresión booleana y una instrucción de recepción. Se pretende aumentar o disminuir el volumen del dispositivo, bajo las condiciones de que éste no haya llegado al máximo o al mínimo volumen, respectivamente; y que se haya recibido además la señal para realizar dicha orden.

2.1.3.2 Instrucción Repetitiva

Por lo general se requiere que una instrucción alternativa sea evaluada un número de veces, para este fin, CSP incorpora una *instrucción repetitiva*. la cual especifica tantas iteraciones como sea posible de la instrucción alternativa; de esta manera, la instrucción repetitiva terminará con éxito cuando todas las custodias fallen; en caso contrario, la instrucción repetitiva se ejecutará otra vez, haciendo que la instrucción alternativa evalúe todas sus custodias nuevamente.

La instrucción repetitiva se expresa sintácticamente colocando como prefijo el símbolo * a la instrucción alternativa, quedando de la forma siguiente:

*[*instrucción alternativa*]

Enseguida un ejemplo para ilustrar la instrucción repetitiva:

```
*[ x > y → x := x − y
□ y > x → y := y − x
 ]
```

Se especifica el cálculo del *mayor común divisor* (mcd) de dos variables. En cada repetición se evalúan ambas custodias y aquella que tenga éxito, permitirá que sea ejecutada la instrucción correspondiente. Esta tarea es repetida hasta que ambas custodias fallen *(cuando $x = y = mcd$)* en cuyo caso el *mcd* de los dos números se habrá encontrado.

2.1.4 Secuencialidad

2.1.4.1 Instrucción Secuencial

La ejecución secuencial se especifica a través de una *instrucción secuencial*, la cual establece la activación en secuencia de un conjunto de dos o más listas de instrucciones que la constituyen, es decir, la ejecución de un número fijo de procesos disjuntos $P_1, P_2, \ldots, P_n$ en el orden en que aparecen escritos. La *instrucción secuencial* se expresa sintácticamente de la forma siguiente:

$$\mathbf{P_1::LI_1\ ;\ P_2::LI_2\ ;\ \ldots\ ;\ P_n::LI_n}$$

Esta instrucción termina exitosamente sólo cuando todos los procesos involucrados hayan terminado con éxito; en caso contrario, la instrucción secuencial falla. Si se hace la suposición de que ninguno de los procesos falla, el tiempo que tardará en terminar la instrucción secuencial es igual a la suma de los tiempos que tardan en terminar cada uno de los procesos.

Enseguida un ejemplo para ilustrar la instrucción secuencial:

Leer::LECTURA ; Procesar::PROCESAMIENTO ; Imprimir::IMPRESION

Se especifica la ejecución secuencial de tres procesos, *(Leer, Procesar e Imprimir)*, cuyos comportamiendos están descritos por sus listas de instrucciones correspondientes.

2.1.4.2 Instrucción de Asignación

El estado local de un proceso puede ser modificado a través de la *instrucción de asignación*, la cual se expresa sintácticamente en su forma más general como sigue:

$$\mathbf{LV := LE}$$

Esta instrucción especifica una lista de expresiones (LE) que denotan valores calculados por cada expresión, los cuales serán asignados a una lista de variables (LV). El efecto de esta instrucción consiste en asignar simultáneamente a cada una de las variables, el valor correspondiente evaluado por cada expresión, según el orden en que fueron especificadas.

Enseguida un ejemplo para ilustrar la instrucción de asignación:

(x, y, z) := (2*x+1, z, y)

Se especifican las asignaciones: $x := 2*x+1$, $y := z$ y $z := y$, las cuales se ejecutan simultáneamente, primero evaluando cada una de las expresiones, y después asignando cada valor a la variable correspondiente.

Una instrucción de asignación falla cuando algún valor calculado está indefinido o cuando el tipo de dato del valor no concuerda con el que espera la variable. Fallará también, si el número de variables es diferente al número de expresiones correspondientes.

2.2 Construcción de un Programa en CSP

Cualquier programa en CSP se construye sólo de instrucciones:

- *Primitivas.*
  En este rubro pertenecen las instrucciones:
  1. *Envío.* (NPR ! expresión)
  2. *Recepción.* (NPE ? variable)
  3. *Asignación.* (LV := LE)
  4. *Nula.* (SKIP)[10]
  5. *Alto.* (STOP)[11]
- Estructuradas.
  En este rubro pertenecen las instrucciones:
  1. *Paralela.* (P_1::LI_1 || P_2::LI_2 || ... || P_n::LI_n)
  2. *Alternativa.* ([$C_1 \rightarrow LI_1 \square C_2 \rightarrow LI_2 \ldots \square C_n \rightarrow LI_n$])
  3. *Repetitiva.* (*[$C_1 \rightarrow LI_1 \square C_2 \rightarrow LI_2 \ldots \square C_n \rightarrow LI_n$])
  4. *Secuencial.* (P_1::LI_1 ; P_2::LI_2 ; ... ; P_n::LI_n)

Enseguida se muestra el fragmento de un programa que simula el *control de un dispositivo digital.* El objetivo es ilustrar la notación CSP en un programa paralelo completo.

[10] La *instrucción Nula* no tiene efecto y siempre termina con éxito.
[11] La *instrucción Alto* no tiene efecto y siempre falla.

```
[ Subir:: *[ Usuario ? señalsubir → Controlador ! señalsubir
           ]
||
  Bajar:: *[ Usuario ? señalbajar → Controlador ! señalbajar
           ]
||
  Apagar:: *[ Usuario ? señalapagar → Controlador ! señalapagar
            ]
||
  Controlador:: *[ volumen < máximo ; Subir ? señalsubir → volumen := volumen + 1
                     Amplificador ! volumen
                   volumen > mínimo ; Bajar ? señalbajar → volumen := volumen − 1
                     Amplificador ! volumen
                   Apagar ? señalapagar → STOP
                 ]
||
  Amplificador:: *[ Controlador ? volumen → SistemaBocinas ! volumen
                  ]
]
```

Se especifica un programa que controla el funcionamiento básico de un dispositivo digital. Se tiene el control de aumentar o disminuir el volumen, así como el de apagar el dispositivo. Cuando un usuario oprime cualquiera de los botones (subir, bajar o apagar), los procesos correspondientes reciben una señal, la cual a su vez la envían al proceso controlador. Este proceso realiza la operación indicada y envía posteriormente un nuevo estado al proceso que actúa sobre el sistema de bocinas.

Es interesante notar que este mismo problema puede ser resuelto a través de un programa secuencial; sin embargo, el hecho de poderlo resolver a través de un programa paralelo permite obtener un mejor desempeño en la ejecución, ya que se proporciona una solución mucho mas cercana a la realidad.

A diferencia de las estructuras que se utilizan para la programación secuencial, las únicas instrucciones que se añaden y que son necesarias para la programación paralela en CSP son: las instrucciones de *envío* y *recepción*, la instrucción *paralela* y la instrucción *alternativa*. Hoare indica que es posible obtener mecanismos más elaborados para la interacción entre procesos a partir de las instrucciones que son incluidas en este lenguaje de especificación.

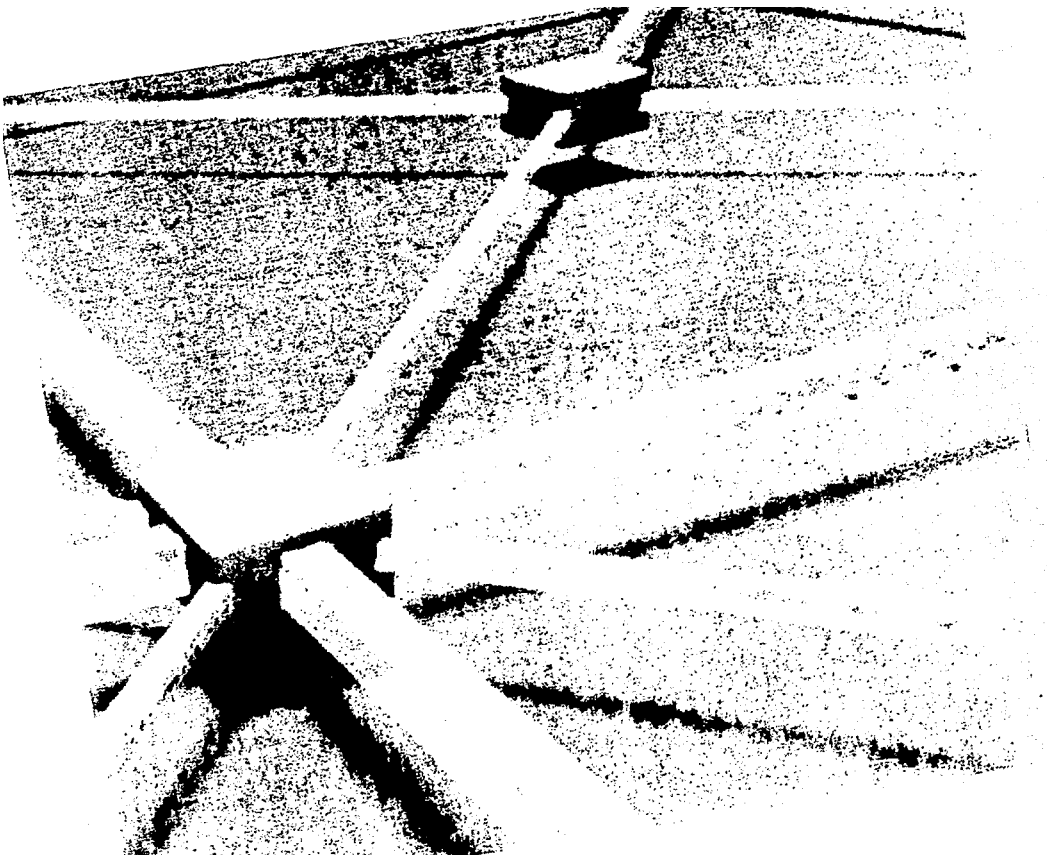
CSP establece un método formal basado en los conceptos de paralelismo y comunicación; asimismo, constituye un lenguaje de especificación ideal y apropiado para resolver aplicaciones del tipo distribuido, las cuales se construyen a partir de un número de procesos, creados en forma estática, que cooperan e interactúan entre sí. Gracias a su poder de expresividad, CSP permite la especificación de sistemas paralelos, los cuales pueden ser mapeados en arquitecturas de procesadores comunicantes. Estas arquitecturas están formadas por múltiples procesadores autónomos y distribuidos, cada uno de los cuales posee su propia memoria local. El objetivo fundamental de CSP fue dar solución a problemas originados en: control de procesos, sistemas de tiempo real, métodos y análisis numéricos, simulación de eventos discretos y diseño de sistemas operativos.

2.3 Especificación y Verificación en CSP

El formalismo CSP posee mecanismos que permiten la expresión de condiciones que deben satisfacerse en un sistema paralelo —esto a través de la especificación algebraica parcial o total del sistema—. CSP se constituye esencialmente como un formalismo matemático bien definido, ya que obedece a un número de leyes y reglas de inferencia básicas, las cuales están basadas en conceptos elementales de la lógica matemática y de la teoría de conjuntos. Estas leyes se han formalizado en una colección de reglas de inferencia, las cuales se pueden utilizar como bloques de construcción de demostraciones básicas. Por tanto, se provee verdadera asistencia al programador en la tarea de especificación, diseño, implantación, verificación y validación de sistemas de cómputo paralelo.

No es intensión en este trabajo describir ninguno de los detalles necesarios para escribir especificaciones y demostraciones formales con CSP[12]. Debido a su poder de expresividad y a su fundamento teórico, el formalismo CSP ha influido en el diseño de varios lenguajes paralelos y distribuidos, así como de sistemas operativos [51], y uno de estos lenguajes, el más notable e importante de ellos es el lenguaje *OCCAM*.

[12]El lector interesado puede consultar [18] para obtener una descripción completa del formalismo CSP y sus aplicaciones para modelar sistemas paralelos y distribuidos.



Capítulo 3

El Lenguaje OCCAM y el Procesador Transputer

En este capítulo se especifican tanto el lenguaje OCCAM como la arquitectura Transputer. Se realiza una descripción de este lenguaje, dando mayor énfasis al estudio y análisis del mecanismo básico que se integra para la comunicación y sincronización entre procesos. Asimismo, en este capítulo se examina al procesador Transputer, el cual constituye el equipo de cómputo en paralelo. Se analiza, también con detalle, la manera de llevar a cabo la comunicación y sincronización entre procesadores Transputer.

Las ideas presentadas en CSP fueron utilizadas como la base para el desarrollo del lenguaje OCCAM. Cuando Hoare propuso y definió CSP, no le preocupaba la forma en cómo lo implantaría, a él le interesaba contar primordialmente con una herramienta que permitiera la expresividad del paralelismo a través de un concepto bien fundamentado y también la creación de una notación de lenguaje susceptible a la especificación, análisis y verificación formal de algoritmos paralelos. Poco tiempo después, se retoma CSP para el planteamiento y definición del lenguaje OCCAM.

En la sección siguiente se ponen en relieve las diferencias esenciales entre el lenguaje de especificación formal CSP y el lenguaje OCCAM.

3.1 CSP–OCCAM

El lenguaje OCCAM hereda todos los conceptos fundamentales encontrados en CSP, es decir, cuenta también con elementos tales como: *paralelismo, comunicación y sincronización, no determinismo y secuencialidad.* En la tabla 3.1 se presenta una comparación que muestra las instrucciones encontradas en CSP y su equivalente en OCCAM.

CSP	OCCAM
P ! exp	canal ! exp
Q ? var	canal ? var
var := exp	var := exp
SKIP	SKIP
STOP	STOP
P ‖ Q ‖ R	PAR P Q R
[P ? dato → R Q ? dato → S]	ALT $canal_p$? dato R $canal_q$? dato S
*[x > y → x := x − y y > x → y := y − x]	WHILE x <> y x > y x := x − y y > x y := y − x
P ; Q ; R	SEQ P Q R

Tabla 3.1: Tabla Comparativa: CSP y OCCAM

A partir de la tabla 3.1 se pueden establecer las diferencias siguientes:

- Al igual que en CSP, el concepto fundamental en OCCAM es el proceso. Un proceso OCCAM puede ser una acción simple *(proceso primitivo*[1]*)* o un conjunto de acciones (procesos) combinadas o agrupadas en un constructor *(proceso constructor*[2]*)*. De esta manera, lo que CSP llama instrucciones en OCCAM son procesos.
- En CSP, las instrucciones de *envío* y *recepción* deben especificar los nombres de los procesos *destino* y *fuente*, respectivamente. En OCCAM este tipo de conexión punto a punto entre dos procesos se realiza a través de un canal común de comunicación.
- En OCCAM, para identificar sintácticamente que un conjunto de procesos pertenece a algún proceso constructor, los procesos deben ser identados exactamente dos espacios a partir de la primera letra del nombre del proceso constructor. En CSP esto es irrelevante.
- Aunque entre CSP y OCCAM se establecen sólo algunas diferencias sintácticas (de notación), OCCAM es un lenguaje que posee otras características y facilidades[3] que permiten una programación con mayor nivel de abstracción.

El lenguaje OCCAM representa la culminación de varios años de esfuerzo de investigación sobre el formalismo CSP. El resultado es un lenguaje simple, elegante y a la vez poderoso, ya que está basado en el concepto de ejecución paralela, permitiendo la expresividad del paralelismo de forma inherente en la solución de un problema.

3.2 Descripción del lenguaje OCCAM

El lenguage de programación OCCAM provee una estructura general para la especificación, diseño, desarrollo, implantación y verificación de sistemas paralelos y distribuidos. Constituye tanto un formalismo de diseño y descripción de algoritmos paralelos[4] como un lenguaje de desarrollo e implantación sobre redes de procesadores distribuidos. Los objetivos principales de diseño de OCCAM fueron proveer un lenguaje que:

- Pudiera ser directamente implantado en una red de elementos de procesamiento.
- Permitiera la expresión directa de algoritmos paralelos.
- Brindara las mismas técnicas de programación paralela para un sólo procesador y para una red de ellos.

[1] En CSP equivale a la instrucción primitiva.
[2] En CSP equivale a la instrucción estructurada.
[3] No incluidas en la tabla 3.1.
[4] En [39] [47] se describe el conjunto de leyes algebraicas de programación en OCCAM.

Procesos OCCAM. El proceso es la unidad fundamental que constituye todo programa paralelo en OCCAM. Un programa paralelo, en este lenguaje, se compone de un conjunto de procesos que se ejecutan simultáneamente, los cuales interactúan, cooperan y se coordinan entre sí a través de la comunicación y sincronización basada en el paso de mensajes, utilizando enlaces de comunicación llamados *canales*. Cada proceso describe el comportamiento de un componente de la implantación y cada canal describe la conexión entre los componentes. Un proceso, por definición, inicia su ejecución, realiza un número de acciones, y después termina[5].

Los procesos más simples en OCCAM son los procesos primitivos. Estos ejecutan alguna de las cinco *acciones* primitivas siguientes: *asignación*, envío, recepción, nula y alto. El nivel siguiente está constituido por procesos constructores, los cuales permiten la creación de procesos más complejos (combinando procesos más simples) en cualquiera de las construcciones siguientes: *paralela*, *alternativa*, *repetitiva*, *secuencial*, *condicional* y *selectiva*.

El lenguaje OCCAM es ideal para resolver problemas, cuyas soluciones se traducen en la descomposición jerárquica de actividades, las cuales son implantadas a través de una red de procesos comunicantes. De esta manera, un proceso puede estar compuesto internamente de otros procesos, por lo que un programa o sistema completo puede ser considerado desde el nivel más alto como un sólo proceso.

En la figura 3.1 se aprecia gráficamente un programa paralelo que ilustra la jerarquía de procesos. Los círculos representan los procesos y las líneas representan los canales de comunicación entre los procesos.

Enseguida se especifica el lenguaje OCCAM, haciendo una descripción de los procesos primitivos y constructores. En la sección 3.3.2 se profundiza el tema de la comunicación y sincronización entre procesos OCCAM.

3.3 Procesos Primitivos

Los procesos más simples en OCCAM son los procesos primitivos. Todos ellos se muestran enseguida:

var := exp	Asignación
SKIP	Nulo
STOP	Alto
canal ! exp	Envío
canal ? var	Recepción

[5]Recuerdese, al igual que en CSP, un proceso OCCAM puede terminar con éxito o fallar.

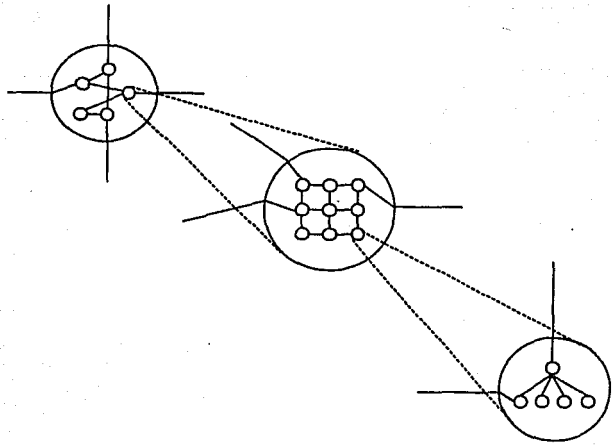


Figura 3.1: **Jerarquía de Procesos en un Programa Paralelo**

3.3.1 Proceso de Asignación

Este proceso modifica el valor de una variable cuando a ésta se le asigna el valor calculado de una expresión. El *proceso de asignación* se expresa sintácticamente de la forma siguiente:

$$\mathbf{LV := LE}$$

Se especifica una lista de expesiones (LE), las cuales son evaluadas en paralelo. Después, cada valor es asignado (también en paralelo) a la variable correspondiente de la lista del lado izquierdo (LV).

Enseguida un ejemplo que ilustra el proceso de asignación:

$$\mathsf{perimetro,\ area := pi * radio,\ pi * radio^2}$$

Es indispensable que se cumplan las condiciones siguientes para que el proceso de asignación no falle:

- El valor calculado de la expresión debe ser del mismo tipo de dato que la variable a la cual será asignado.
- No debe aparecer una variable repetida más de una vez del lado izquierdo.
- El número de variables en LV debe ser igual al número de expresiones en LE.

3.3.2 Comunicación y Sincronización entre Procesos

Los procesos en un programa paralelo requieren cooperar y coordinarse entre sí para alcanzar un objetivo. Como se ha mencionado, esta cooperación y coordinación entre procesos se lleva a cabo a través de la comunicación y sincronización. En OCCAM, los procesos primitivos de *envío* y *recepción* permiten establecer la comunicación síncrona entre pares de procesos[6] enlazados por un *canal* común de comunicación.

3.3.2.1 Canales OCCAM

La comunicación y sincronización entre procesos se basa en el paso de mensajes. Dos procesos que se ejecutan en paralelo se transmiten información el uno al otro a través de un canal. Un canal de comunicación OCCAM, llamado también *canal lógico*[7], posee las características siguientes:

- *Establece un tipo de conexión punto a punto.* Un canal permite enlazar sólo a un par de procesos, cada uno de los cuales se encuentra en los extremos del canal.
- *Permite una comunicación unidireccional.* Un canal puede transmitir información únicamente en una sóla dirección, esto es, de un proceso a otro. Si se requiere comunicar en ambas direcciones, se deben usar dos canales.
- *Se le pueden aplicar dos operaciones elementales: el envío y la recepción.* Para que sobre un canal se puedan transmitir mensajes, los procesos emisor y receptor deben invocar a los procesos primitivos respectivos. Siempre uno enviando y el otro recibiendo.
- *No posee capacidad de almacenamiento temporal o bufferización.* Un canal no puede almacenar mensajes, por consiguiente las operaciones de envío y recepción implican la sincronización entre los procesos. Se puede apreciar entonces, que un canal de este tipo unifica tanto la comunicación como la sincronización en un mecanismo básico.

[6] Uno de ellos es llamado *proceso emisor* y el otro *proceso receptor.*
[7] Se usará indistintamente canal OCCAM o canal lógico.

- *Tiene un protocolo síncrono tipo "rendezvous"*. Un canal establece una comunicación síncrona entre dos procesos ejecutándose simultáneamente, ya que estos antes de poderse comunicar, primero sincronizan sus actividades, es decir, llegan al punto en el cual los procesos primitivos de envío y recepción *corresponden*.
- *Es un objeto con un tipo de dato asociado para el intercambio de información.* Un canal se declara definiéndole un tipo de dato[8] igual al del mensaje que se desea transmitir.
- *Constituye un objeto de comunicación confiable.* Se asegura que la comunicación sobre un canal lógico será totalmente confiable en el sentido de que no existe la posibilidad de que algún mensaje pueda perderse o ser alterada su información.

3.3.2.2 Proceso de Envío

Este proceso transmite un valor a través de un canal, el valor puede ser una constante o ser calculado de una expresión. El *proceso de envío* se expresa sintácticamente de la forma siguiente:

NC ! expresión

Se especifica el nombre del canal (NC), y después se proporciona el valor a ser enviado.

Enseguida un ejemplo que ilustra el proceso de envío:

canal.datos ! (b + sqrt($b^2 - 4 * a * c$)) / (2 * a)

Es indispensable que se cumplan las condiciones siguientes para que el proceso de envío no falle:

- El proceso receptor no debe fallar.
- El proceso receptor no debe terminar su ejecución sin invocar antes al proceso de recepción, utilizando el mismo canal.
- El valor de la expresión no debe estar indefinido.

3.3.2.3 Proceso de Recepción

Este proceso recibe un valor a través de un canal y lo asigna a una variable. El *proceso de recepción* se expresa sintácticamente de la forma siguiente:

NC ? variable

Se especifica el nombre del canal (NC), y después se proporciona una variable en la cual se asigna el valor recibido.

[8]En la sección 3.3.2.4 se especifican los tipos de datos disponibles en OCCAM.

Enseguida un ejemplo que ilustra el proceso de recepción:

```
canal.datos ? solución.raíz
```

Es indispensable que se cumplan las condiciones siguientes para que el proceso de recepción no falle:

- El proceso emisor no debe fallar.
- El proceso emisor no debe terminar su ejecución sin antes invocar al proceso de envío, utilizando el mismo canal.
- La variable no debe recibir un valor de diferente tipo de dato del esperado.

En la figura 3.2 se aprecia gráficamente el concepto de un canal lógico.

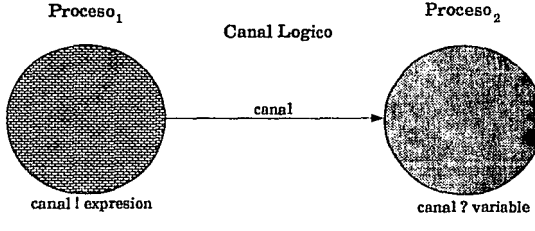


Figura 3.2: **Un Canal OCCAM Entre Dos Procesos**

Si se usa un canal para hacer la conexión entre un proceso emisor y un proceso receptor, entonces la comunicación ocurrirá únicamente cuando ambos procesos se encuentren listos para realizarla, es decir, cuando los procesos hayan llegado al punto, llamado *punto de sincronización*, en el que ejecutan cada uno de ellos, ya sea su proceso de envío o su proceso de recepción, respectivamente. En este caso, el valor se transmite del proceso emisor al proceso receptor.

Cualquiera de los procesos, ya sea el emisor o el receptor, permanecerá bloqueado hasta que su proceso correspondiente se encuentre listo para establecer la comunicación. Si alguno de los procesos falla sin haber ejecutado antes el envío o la recepción,

entonces se dará lugar a un estado de *bloqueo mutuo*. Los procesos emisor y receptor continuarán su ejecución únicamente cuando el valor enviado por el emisor haya llegado al receptor con éxito.

El estado interno de un proceso no es visible al mundo exterior, ya que todas las interacciones con el proceso ocurren vía los canales de comunicación. Esta filosofía elimina los problemas asociados con variables compartidas [23].

Como en la mayoría de los lenguajes, los objetos que se manejan en OCCAM (variables, constantes, literales, canales, etc.) deben poseer un tipo de dato asociado. En la sección siguiente se describen los tipos de datos disponibles en el lenguaje OCCAM.

3.3.2.4 Tipos de Datos

Un tipo de dato determina el conjunto de valores que pueden ser tomados por un objeto. Enseguida se especifican los diferentes objetos que pueden ser manejados en el lenguaje OCCAM:

OBJETOS	DESCRIPCION
Constantes	Nombres simbólicos que poseen un valor constante.
Variables	Nombres simbólicos que poseen un valor, el cual puede ser cambiado por una asignación o una recepción.
Literales	Representación textual de valores conocidos.
Indices	Nombres simbólicos que poseen un valor, el cual varía progresivamente. Son utilizados por los replicadores.
Canales	Nombres simbólicos necesarios para la transmisión de valores entre procesos emisores y procesos receptores.
Marcadores de tiempo	Nombres simbólicos que proveen un valor, el cual se lee de un reloj.

El lenguaje OCCAM incorpora solamente dos clases de tipos de datos: los *primitivos* y los *estructurados*. Para indicar el tipo de dato de un objeto, éste debe ser previamente declarado[9].

[9]Al final de cualquier tipo de declaración siempre se colocan ";".

Tipos de datos primitivos. Enseguida se muestran los tipos de datos primitivos[10] disponibles:

TIPO	VALORES
BOOL	Booleanos (TRUE y FALSE)
BYTE	Enteros de 0 a 255
INT	Enteros con signo
INT16	Enteros con signo de -32768 a 32767
INT32	Enteros con signo de -2^{31} a $(2^{31}-1)$
INT64	Enteros con signo de -2^{63} a $(2^{63}-1)$
REAL32	Punto flotante de 32 bits
REAL64	Punto flotante de 64 bits

Constante. Una *constante* se especifica a través de una *abreviación*. Una abreviación se utiliza para dar un nombre a un valor constante o a uno calculado por una expresión.

Enseguida un ejemplo de declaración de abreviaciones[11]:

```
VAL pi IS 3.1416 :
VAL exp IS ( x + ( y * z ) / 2 ) - ( 1 / 250 ) :
VAL esc IS 27 :
```

Una expresión en una abreviación puede contener variables; sin embargo, estas variables deben permanecer constantes, es decir, que no se debe modificar su contenido, ya sea por una asignación o por una recepción, a lo largo del alcance[12] que tenga la declaración de la abreviación. Por tanto, cualquier abreviación siempre se comportará como una constante.

Variable. Una *variable* posee un valor de un tipo de dato específico. Este valor puede ser modificado a través de una asignación, la cual se realiza utilizando ya sea el proceso de asignación o el de recepción. Cuando se hace la declaración de una variable se debe especificar su tipo de dato y su nombre.

Enseguida un ejemplo de declaración de variables:

```
BOOL bandera :
INT32 m, n, p, q :
REAL64 raiz :
```

[10] Se sugiere consultar [22] para mayor información al respecto.

[11] Se sugiere consultar [22] para mayor información acerca de las abreviaciones.

[12] Región de programa en la cual es válida una declaración. Se sugiere consultar [22] para profundizar en el tema de *alcance*.

Literal. Una *literal* es un valor conocido, el cual es representado textualmente y posee además un tipo de dato. Enseguida algunos ejemplos de literales válidas:

EJEMPLO	LITERAL
719	Entera decimal
#A7	Entera hexadecimal
'M'	Byte
"Hola mundo"	Cadena
TRUE	Booleana
77(BYTE)	Valor byte
'G'(INT)	Valor entero
97.7(REAL32)	Valor punto flotante de 32 bits

Índice. Un *índice* es un tipo de variable que no requiere ser previamente declarada, y que toma valores en un rango especificado por un *replicador*. Un replicador[13] permite que un proceso constructor pueda ser replicado con la finalidad de producir un número similar de procesos. Su comportamiento es equivalente al de un mecanismo repetitivo convencional. Un índice puede además ser utilizado como un operando en expresiones. Es importante mencionar que su valor no debe cambiar, ya sea por una asignación o por una recepción.

Canal. Un *canal*[14] es un objeto tipificado para el intercambio de información entre dos procesos, uno emisor y otro receptor, ejecutándose simultáneamente.

Marcador de tiempo. Un *marcador de tiempo*[15] es un objeto de tipo TIMER que provee el valor de un reloj, el cual puede ser accesado por un conjunto de procesos ejecutándose simultáneamente.

Tipos de datos estructurados. El único tipo de dato estructurado disponible es el *tipo arreglo*. Un arreglo se compone de un conjunto de localidades del mismo tipo de dato primitivo.

Enseguida un ejemplo de declaración de variables tipo arreglo:

```
[50]BYTE caracter :
[1000]INT16 vector :
[75]REAL32 complejo :
```

[13]Los replicadores se discutirán más adelante cuando se describan cada uno de los procesos constructores (sección 3.4).

[14]Los protocolos de un *canal* se consideran en la sección 3.3.2.5.

[15]Los marcadores de tiempo o *"Timers"*, en inglés, se describen en la sección 3.3.2.6.

Los elementos de un arreglo se numeran iniciando desde *0* hasta $N - 1$, donde N es el número de localidades declaradas. Cualquier arreglo tiene por lo menos una localidad.

Los arreglos multidimensionales se construyen como arreglos de arreglos. En teoría no existe restricción en las dimensiones de un arreglo; sin embargo, los arreglos requieren de memoria, la cual tiene un límite.

Enseguida un ejemplo de declaración de arreglos multidimensionales.

```
[20][20]INT64 matriz :
[35][35]BOOL criba :
[10][5][20]INT cubo :
```

Segmentos de arreglo. El lenguaje OCCAM permite manipular segmentos de arreglo, los cuales son también en sí mismos arreglos. La expresión sintáctica es la siguiente:

nombre del arreglo **FROM** *índice* **FOR** *contador*

Un segmento de arreglo inicia con el nombre del arreglo, enseguida la palabra reservada FROM, después un valor de tipo *INT* (índice), el cual indica el primer elemento del segmento, se continúa con la palabra reservada *FOR* y por último otro valor de tipo *INT* (contador)[16], el cual especifica el número de elementos en el segmento. Un segmento de arreglo tiene *0* o más elementos y conserva además la misma dimensión del arreglo.

Enseguida un ejemplo de utilización de los segmentos de arreglo:

```
[100]INT vector.uno :
[50]INT vector.dos :
SEQ
  [vector.uno FROM 0 FOR 50] := vector.dos
  [vector.uno FROM 20 FOR 30] := [vector.dos FROM 10 FOR 30]
```

Se especifica la declaración de dos arreglos: *vector.uno* y *vector.dos*. La primera asignación indica que los primeros 50 elementos del arreglo *vector.dos* son asignados a las primeras 50 localidades del arreglo *vector.uno*, es decir:

```
vector.uno[0],...,vector.uno[49] := vector.dos[0],..., vector.dos[49]
```

[16]El valor de *contador* no debe ser negativo, ni debe violar el rango declarado en el arreglo.

La segunda asignación sería equivalente a las asignaciones siguientes:

```
vector.uno[20],...,vector.uno[49] := vector.dos[10],..., vector.dos[39]
```

Se permite el uso de abreviaciones para nombrar arreglos o segmentos de arreglo. Enseguida un ejemplo:

```
VAL seg.uno IS [vector.uno FROM 20 FOR 30] :
VAL seg.dos IS [vector.dos FROM 10 FOR 30] :
SEQ
  seg.uno := seg.dos
```

3.3.2.5 Protocolos de un Canal

Un canal provee una comunicación punto a punto, unidireccional y no bufferizada entre dos procesos en ejecución simultánea. Para declarar un canal se debe especificar su nombre y su tipo de protocolo. El *protocolo del canal* se utiliza para indicar el formato y tipo de valores que van a ser transmitidos.

Existen cuatro tipos de protocolos: *simple*, *secuencial*, *variante* y *anárquico*.

Protocolo simple. Este protocolo consiste de tipos de datos primitivos o estructurados.

Enseguida un ejemplo de declaración de canales con protocolo simple y de procesos de envío o recepción, utilizando estos canales:

```
CHAN OF BOOL canal.terminación :
CHAN OF BYTE canal.pantalla :
CHAN OF [15]BYTE canal.mensaje :
PAR
  canal.terminación ! TRUE
  canal.pantalla ! dato
  -- Un canal que transmite un arreglo de
  -- 15 elementos tipo BYTE
  canal.mensaje ! "Lasquinceletras"
```

Para mayor flexibilidad, cuando se desea recibir un arreglo de BYTE's y el número de elementos no se conoce, se utiliza un protocolo especial llamado protocolo de *arreglo contado*.

Enseguida un ejemplo de declaración y uso de un canal con protocolo de arreglo contado:

```
CHAN OF INT::[]BYTE canal.frase :
[9]BYTE frase :
INT longitud :
PAR
  canal.frase ! 9::"Je t'aime"
  canal.frase ? longitud::frase
```

Se especifica la declaración tanto de un canal con protocolo de arreglo contado como la de un arreglo de BYTE's. En el envío, primero se transmite el valor entero (longitud del texto), y después se envía la cadena de bytes. En la recepción, primero se recibe el valor entero, el cual se asigna a la variable *longitud*, y después se reciben los bytes, los cuales se asignan a las primeras localidades del arreglo de BYTE's.

Protocolo secuencial. Este protocolo especifica un protocolo de comunicación que consiste de una secuencia de protocolos simples. De esta manera, un canal es capaz de transmitir una secuencia o combinación de tipos de datos, ya sea primitivos o estructurados.

Enseguida un ejemplo de declaración de canales con protocolo secuencial y de procesos de envío o recepción, utilizando estos canales:

```
PROTOCOL COMPLEJO IS REAL64; REAL64 :
PROTOCOL REGISTRO IS [20]BYTE; [30]BYTE; [10]BYTE; INT; BYTE; BOOL :
CHAN OF COMPLEJO canal.complejo :
CHAN OF REGISTRO canal.registro :
REAL64 parte.real, parte.imaginaria :
[20]BYTE nombre :
[30]BYTE dirección :
[10]BYTE teléfono :
...
PAR
  canal.complejo ? parte.real; parte.imaginaria
  canal.registro ? nombre; dirección; teléfono; edad; sexo; edo.civil
```

Un envío o recepción, a través de un canal con protocolo secuencial, es una secuencia de envíos o recepciones con valores diferentes. En el caso del proceso de envío, cada valor se envía en secuencia y en el caso del proceso de recepción, cada valor se recibe también en secuencia y se asigna a cada variable correspondiente.

Se permite asignar un nombre a un protocolo a través de un *nombramiento de protocolo*. Este nombramiento se puede efectuar solamente en los protocolos simple y secuencial. Enseguida un ejemplo:

```
PROTOCOL CARACTER IS BYTE :
PROTOCOL PIXEL IS REAL16; REAL16 :
CHAN OF CARACTER canal.teclado :
CHAN OF PIXEL canal.punto :
PAR
  canal.teclado ? dato
  canal.punto ! 101.7, 99.3
```

Protocolo variante. Este protocolo especifica un número de formatos posibles para comunicación sobre un mismo canal. Un formato puede ser únicamente de protocolo simple o secuencial. De esta manera, una canal puede ser capaz de enviar o recibir diferentes tipos de datos, distinguiendo cada formato con el uso de etiquetas.

Enseguida un ejemplo de declaración de un canal con protocolo variante y de procesos de envío o recepción, utilizando este canal:

```
PROTOCOL REGISTRO
  CASE
    nom; [15]BYTE
    dir; [30]BYTE
    tel; [10]BYTE
    fin -- Es posible transmitir etiquetas
:
CHAN OF REGISTRO canal.datos :
PAR
  canal.datos ! nom; "M. Mitterrand"
  canal.datos ! dir; "57, rue Champs Elysées"
  canal.datos ! tel; "2-95-23-85"
  canal.datos ? CASE
    nom; nombre
      canal.imprime ! nombre
    dir; dirección
      canal.imprime ! dirección
    tel; teléfono
      canal.imprime ! teléfono
    fin
      canal.salida ! TRUE
```

Una comunicación de envío sobre un canal con protocolo variante, primero transmite la etiqueta, informando al proceso de recepción el formato que tiene el resto de la comunicación. El proceso de recepción incorpora un mecanismo de selección (? CASE) que permite agrupar a un conjunto de variantes de recepción, identificadas cada una de ellas por diferentes etiquetas.

De esta manera, primero se recibe la etiqueta, y después se compara ésta con cada una de las etiquetas que tienen la variantes de recepción. Cuando se encuentra el empatamiento de etiquetas, entonces se procede a recibir el resto del mensaje. Un vez concluida esta recepción existe la posibilidad de ejecutar un proceso asociado, el cual deberá especificarse en la línea de código siguiente.

Protocolo anárquico. Este protocolo es una clase de protocolo en el que no se define un formato. Se utiliza generalmente cuando se requiere de la comunicación con dispositivos externos, tal como puede ser el caso de una impresora, unidades de disco, interfaces, etc.

Enseguida un ejemplo de declaración de canales con protocolo anárquico y de procesos de envío o recepción, utilizando estos canales:

```
CHAN OF ANY canal.impresora :
CHAN OF ANY canal.disco.duro :
CHAN OF ANY canal.RS232 :
...
PAR
  canal.impresora ! resultado
  canal.disco.duro ? registro
  canal.RS232 ! #FA02
```

El efecto de un envío sobre un canal con protocolo anárquico es mapear el valor a sus bytes respectivos para después transmitirlo como un arreglo de bytes. En la recepción, primero se recibe el arreglo de bytes, después este arreglo se convierte al tipo de dato que tiene la variable en el proceso de recepción y por último se realiza la asignación.

3.3.2.6 Marcadores de Tiempo

Un *marcador de tiempo* provee un reloj que puede ser accesado por un conjunto de procesos ejecutándose simultáneamente. Su declaración se realiza de manera similar a la de las variables y los canales. Su comportamiento es análogo al de un canal, pero sólo se permiten recepciones.

Enseguida un ejemplo de declaración y uso de los marcadores de tiempo:

```
[2]TIMER relojA :
TIMER relojB :
INT tiempoA, tiempoB, tiempoC, retardo :
BOOL bandera :
...
PAR
  relojA[0] ? tiempoA
  relojA[1] ? tiempoB
  SEQ
    tiempoA := tiempoA PLUS tiempoB
    tiempoB := tiempoB MINUS tiempoA
    bandera := tiempoA AFTER tiempoB
    relojB ? tiempoC
    relojB ? AFTER tiempoC PLUS retardo
```

Un marcador de tiempo provee un valor de tipo entero (`INT`) que representa el tiempo tomado de un reloj físico, es decir, de un reloj que se encuentra físicamente integrado en el procesador. El tiempo va cambiando por un incremento en intervalos regulares; además, este valor del reloj es cíclico, es decir, que cuando alcanza el mayor valor entero positivo se vuelve negativo y empieza a contar en retroceso hacia cero.

Los operadores `PLUS` y `MINUS` permiten hacer la adición y la sustracción, respectivamente, con valores de tipo `TIMER`. El operador `AFTER` se utiliza para comparar tiempos, por lo que *x AFTER y* regresa un valor verdadero si *(x MINUS y) > 0*. Este operador se utiliza comúnmente para hacer retardar la ejecución de un proceso hasta que sea alcanzado un valor establecido.

En las últimas dos líneas del ejemplo anterior, primero se toma el tiempo actual del reloj y se asigna en *tiempoC*, después este valor se utiliza como operando en el proceso siguiente que retrasará la ejecución hasta que *(relojB MINUS (tiempoC PLUS retardo)) > 0*, es decir, hasta que pase el tiempo estipulado en *retardo*.

Es importante mencionar que los marcadores de tiempo se utilizan principalmente en la programación y procesamiento en tiempo real.

3.3.3 Proceso Nulo

Este proceso inicia su ejecución, no realiza ninguna acción y termina inmediatamente con éxito. El *proceso nulo* se expresa sintácticamente de la forma siguiente:

SKIP

SKIP se puede ver como un proceso que simplemente no hace nada. Puede utilizarse en un programa que aún no ha sido terminado, colocándolo en lugar de un proceso que será posteriormente escrito. Se asume que se trata de un proceso, sin el cual, el programa podría continuar su ejecución. La sintáxis de OCCAM muchas veces requiere que el proceso nulo esté presente con la finalidad de evitar que un proceso constructor falle.

Enseguida un ejemplo que ilustra el proceso nulo:

```
ALT
  canal.botón ? señal
    SKIP -- Proceso que prueba un motor eléctrico
```

3.3.4 Proceso Alto

Este proceso inicia su ejecución, no realiza ninguna acción y nunca termina. El *proceso alto* se expresa sintácticamente de la forma siguiente:

STOP

STOP se puede ver como un proceso que inhibe la ejecución sin que sea posible terminar con éxito. Al igual que SKIP, el proceso alto puede utilizarse en lugar de un proceso que aún no ha sido elaborado y que será posteriormente escrito. Se asume que se trata de un proceso, sin el cual, el programa no podría continuar su ejecución. Otro de los usos del proceso alto es en la depuración de programas.

Enseguida un ejemplo que ilustra el proceso alto:

```
ALT
  canal.botón1 ? señal
    Prueba.Motor.Eléctrico()
  canal.botón2 ? señal
    STOP -- Proceso que pone en marcha el dispositivo
```

Cuando un proceso OCCAM falla, se dice que éste se comporta como el proceso alto. No se debe tomar a este proceso como una manera de forzar a que termine un programa, ni para provocar en un programa una clase de condición de error.

3.4 Procesos Constructores

A través de los *procesos constructores* se pueden crear procesos[17] cada vez más complejos, combinando procesos más simples[18], esto se debe a que un constructor es en sí mismo un proceso y puede además ser usado como un componente de otro constructor; se pueden también obtener redes o estructuras jerárquicas de procesos. Un constructor permite crear procesos en cualquiera de las clases mostradas enseguida:

PAR	Paralelo
ALT	Alternativo
WHILE	Repetitivo
SEQ	Secuencial
IF	Condicional
CASE	Selección

3.4.1 Proceso Paralelo

El *proceso constructor paralelo* PAR es uno de los constructores más útiles e importantes del lenguaje OCCAM. Permite combinar un conjunto de procesos más simples, los cuales son ejecutados simultáneamente. Su expresión sintáctica es la siguiente:

PAR
  $\mathbf{P_1}$
  ...
  $\mathbf{P_n}$

Se especifica la ejecución simultánea de los procesos $P_1,\ldots,P_n$, los cuales componen el proceso constructor paralelo. La comunicación de mensajes y la sincronización entre los procesos involucrados se realiza a través de los canales que enlazan a parejas de procesos.

Enseguida un ejemplo que ilustra el proceso constructor paralelo:

```
PAR
  Entrada(canal.entrada.datos)
  Procesamiento(canal.entrada.datos, canal.salida.datos)
  Salida(canal.salida.datos)
```

En la figura 3.3 se aprecia gráficamente la interconexión de los procesos del ejemplo anterior a través de sus canales lógicos.

[17] Recuérdese que la creación de procesos es estática.

[18] Los procesos involucrados en cualquiera de los constructores se codifican sintácticamente con una identación de dos espacios.

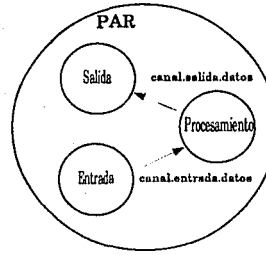


Figura 3.3: **Proceso Constructor PAR: Procesos en Ejecución Simultánea**

Para que el proceso constructor paralelo no falle se debe cumplir que todos y cada uno de los procesos involucrados finalicen su ejecución con éxito.

3.4.1.1 Replicador Paralelo

El *replicador paralelo* permite construir un conjunto de procesos similares, los cuales serán simultáneamente ejecutados. Su expresión sintáctica es la siguiente:

PAR *índice* = *valor de inicio* **FOR** *número de repeticiones*
  P

Este replicador especifica un *índice*, cuyo primer valor para la primera réplica es *valor de inicio*. El número de veces que el proceso P es replicado está determinado por *número de repeticiones*.

Enseguida un ejemplo de un replicador paralelo:

```
PAR i = 30 FOR 80
  canal.usuario[i] ! mensaje

Lo cual es equivalente a:

PAR
  canal.usuario[30] ! mensaje
  canal.usuario[31] ! mensaje
  ...
  canal.usuario[109] ! mensaje
```

3.4.2 Proceso Alternativo

El *proceso constructor alternativo* ALT es otro de los constructores más útiles e importantes del lenguaje OCCAM. Provee el tratamiento al no determinismo a través de la combinación de un conjunto de procesos custodiados, cada uno de los cuales está compuesto por un proceso de recepción y/o una expresión booleana. Su expresión sintáctica es la siguiente:

ALT
  custodia$_1$
    proceso$_1$
  ...
  custodia$_n$
    proceso$_n$

Se especifica la ejecución del proceso *proceso*$_i$, si la custodia asociada *custodia*$_i$ tiene éxito. Una custodia está constituida por un proceso de recepción y puede opcionalmente iniciar con una expresión booleana, la cual permite habilitar o deshabilitar la verificación de la operación de recepción, en el caso de que esta expresión evalúe verdadera o falsa, respectivamente. La expresión sintáctica de una custodia es la siguiente:

expresión booleana & *proceso de recepción*

Una custodia compuesta sólo por una *expresión booleana* debe completarse con & SKIP para ser válida sintácticamente. Si se desea que una custodia siempre tenga éxito, entonces ésta debe tener la forma siguiente:

TRUE & SKIP

Cuando son evaluadas todas las custodias y ninguna de ellas tiene éxito, entonces el proceso alternativo espera hasta que alguna tenga éxito. Si dos o más custodias tienen éxito, sólo una de ellas es seleccionada no determinísticamente y el proceso asociado es ejecutado.

Enseguida un ejemplo para ilustrar el proceso constructor alternativo:

```
ALT
  canal.izquierdo ? paquete.datos
    canal.flujo ! paquete.datos
  canal.centro ? paquete.datos
    canal.flujo ! paquete.datos
  canal.derecho ? paquete.datos
    canal.flujo ! paquete.datos
```

En la figura 3.4 se aprecia gráficamente el comportamiento de comunicación del ejemplo anterior. Nótese en esta figura la línea ondulada, la cual indica el flujo de comunicación, ilustrando el éxito de una de las custodias.

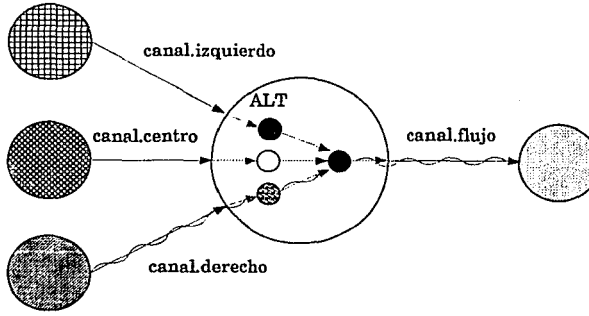


Figura 3.4: **Proceso Constructor ALT: Unión de un Flujo de Datos**

Para que el proceso constructor alternativo no falle se debe cumplir lo siguiente:

- Debe existir por lo menos una custodia.
- Ninguna de las custodias debe fallar.
- El proceso que sea ejecutado, correspondiente a la custodia que tuvo éxito, no debe fallar.

3.4.2.1 Replicador Alternativo

El *replicador alternativo* permite construir un conjunto de custodias similares en donde para cada una de ellas existe un proceso asociado. Su expresión sintáctica es la siguiente:

ALT *índice* = *valor de inicio* **FOR** *número de repeticiones*
  custodia
    P

Este replicador especifica un *índice*, cuyo primer valor para la primera réplica es *valor de inicio*. El número de veces que la *custodia*, junto con el proceso asociado P son replicados está determinado por *número de repeticiones*.

Enseguida un ejemplo de un replicador alternativo:

```
ALT i = 1 FOR 50
  canal.llega[i] ? mensaje
    canal.sale ! mensaje
```

Lo cual es equivalente a:

```
ALT
  canal.llega[1] ? mensaje
    canal.sale ! mensaje
  canal.llega[2] ? mensaje
    canal.sale ! mensaje
  ...
  canal.llega[50] ? mensaje
    canal.sale ! mensaje
```

3.4.3 Prioridad en los Procesos Paralelo y Alternativo

Un aspecto importante en la programación y procesamiento en tiempo real es la *prioridad* con la cual se ejecutan los procesos. OCCAM maneja el concepto de prioridad en el lenguaje. En teoría se pueden manejar varios niveles de prioridades a nivel lenguaje; sin embargo, en la implantación se permiten sólo dos niveles. Por tanto, se recomienda la construcción de programas tomando en cuenta solamente los niveles de prioridad: *alta y baja*.

La prioridad se asigna de acuerdo al orden textual en el que aparecen escritos los procesos involucrados en el proceso constructor paralelo o alternativo. De esta

manera, el primer proceso involucrado en la lista tiene la prioridad alta y el siguiente tiene la prioridad baja[19].

Prioridad en el Proceso Constructor Paralelo. Para indicar sintácticamente la prioridad se debe anteponer al proceso constructor PAR la palabra reservada PRI. Enseguida un ejemplo:

```
PRI PAR
  Manejador.Interrupciones() -- Proceso con prioridad alta
  Procesamiento.Datos() -- Proceso con prioridad baja
```

El primer proceso (prioridad alta) siempre inicia la ejecución. El segundo (prioridad baja) puede ser eventualmente ejecutado, cuando el primero quede bloqueado esperando completar un envío o una recepción; o cuando haya terminado su ejecución.

Prioridad en el Proceso Constructor Aternativo. Para indicar sintácticamente la prioridad se debe anteponer al proceso constructor ALT la palabra reservada PRI. Enseguida un ejemplo:

```
PRI ALT
  canal.termina ? señal -- Custodia con prioridad alta
    bandera := FALSE
  TRUE & SKIP -- Custodia con prioridad baja
    proceso.principal()
```

Cuando las dos custodias tienen éxito al mismo tiempo, el proceso correspondiente a la custodia con prioridad alta es el que se ejecuta. Aunque en el ejemplo anterior la segunda custodia siempre tiene éxito, con el uso de PRI ALT se obliga la inspección de la primera custodia, ya que tiene la prioridad más alta, y de esta manera se podrían evitar resultados inesperados.

3.4.4 Proceso Repetitivo

El *proceso constructor repetitivo* WHILE permite la repetición de un proceso mientras su expresión booleana asociada es verdadera.

[19]Recuerdese que un proceso A puede estar compuesto de un conjunto de procesos, sean estos: $A_1, A_2, \ldots, A_n$, en este caso, el conjunto de procesos siempre conservará la misma prioridad que el proceso A.

Su expresión sintáctica es la siguiente:

WHILE *expresión booleana*
P

Se especifica la ejecución del proceso P mientras que ***expresión booleana*** no evalúe falso.

Enseguida un ejemplo que ilustra el proceso constructor repetitivo:

```
WHILE NOT termina
  ALT
    canal.entrada ? dato
      canal.salida ! dato
    canal.termina ? señal
      termina := TRUE
```

Para que el proceso constructor repetitivo no falle se debe cumplir que su proceso asociado termine con éxito.

3.4.5 Proceso Secuencial

El *proceso constructor secuencial* SEQ combina un conjunto de procesos, cuya ejecución se realiza secuencialmente. Su expresión sintáctica es la siguiente:

SEQ
$\mathbf{P}_1$
...
$\mathbf{P}_n$

Se especifica la ejecución secuencial de los procesos $P_1,\ldots,P_n$, siguiendo ese orden, los cuales componen el proceso constructor secuencial.

Enseguida un ejemplo que ilustra el proceso constructor secuencial:

```
SEQ
  canal.teclado ? caracter
  canal.pantalla ! caracter
  canal.pantalla ! CR
  canal.pantalla ! LF
```

Para que el proceso constructor secuencial no falle se debe cumplir que todos y cada uno de los procesos involucrados finalicen su ejecución con éxito.

3.4.5.1 Replicador Secuencial

El replicador secuencial permite construir un conjunto de procesos similares, los cuales se ejecutan en secuencia. Su expresión sintáctica es la siguiente:

SEQ *índice* = *valor de inicio* **FOR** *número de repeticiones*
  P

Este replicador especifica un *índice*, cuyo primer valor para la primera réplica es *valor de inicio*. El número de veces que el proceso P es replicado está determinado por *número de repeticiones*.

Enseguida un ejemplo de un replicador secuencial:

```
SEQ i = 7 FOR 44
  canal.resultado ! ( a + i ) / 365
```

Lo cual es equivalente a:

```
SEQ
  canal.resultado ! ( a + 7 ) / 365
  canal.resultado ! ( a + 8 ) / 365
  ...
  canal.resultado ! ( a + 50 ) / 365
```

3.4.6 Proceso Condicional

El *proceso constructor condicional* IF permite combinar un número de procesos, cada uno de los cuales es custodiado por una expresión booleana. Su expresión sintáctica es la siguiente:

IF
  expresión booleana$_1$
    P_1
  ...
  expresión booleana$_i$
    P_i
  ...
  TRUE
    P_n

Se especifica la evaluación secuencial de cada una de las expresiones booleanas. Si se encuentra que alguna de ellas es verdadera, entonces se ejecuta el proceso asociado y el proceso condicional termina. Cuando se tiene la posibilidad de que ninguna de las *expresiones booleanas* previas evalúen verdadero, se puede utilizar la constante booleana TRUE, la cual garantiza que al menos un proceso será ejecutado; y de esta manera se termine con éxito.

Enseguida un ejemplo que ilustra el proceso constructor condicional:

```
IF
  potencia < 1
    potencia := potencia + k
  potencia > 1000
    potencia := potencia − m
  TRUE
    SKIP
```

Para que el proceso constructor condicional no falle se debe cumplir lo siguiente:

- Al menos una de las expresiones booleanas debe ser verdadera.
- El proceso ejecutado correspondiente a la expresión booleana verdadera debe finalizar su ejecución con éxito.

3.4.6.1 Replicador Condicional

El replicador condicional permite construir un conjunto de custodias compuestas por expresiones booleanas en donde, para cada una de ellas, existe un proceso asociado. Su expresión sintáctica es la siguiente:

IF *índice* = *valor de inicio* **FOR** *número de repeticiones*
expresión booleana
P
TRUE
Q

Este replicador especifica un *índice*, cuyo primer valor para la primera réplica es *valor de inicio*. El número de veces que la *expresión booleana* y el proceso asociado P son replicados está determinado por *número de repeticiones*.

Enseguida un ejemplo de un replicador condicional:

```
IF i = 1 FOR 5000
  cadena[i] = objeto
    encontrado := TRUE
  TRUE
    encontrado := FALSE
```

Lo cual es equivalente a:

```
IF
  cadena[1] = objeto
    encontrado := TRUE
  cadena[2] = objeto
    encontrado := TRUE
  ...
  cadena[5000] = objeto
    encontrado := TRUE
  TRUE
    encontrado := FALSE
```

3.4.7 Proceso Selectivo

El *proceso constructor selectivo* CASE permite combinar un número de opciones, una de las cuales es seleccionada si el valor de un selector es igual al valor de una constante asociado con la opción. Una opción se compone de un valor constante o varios de ellos separados por comas. Todos estos valores usados en el proceso selectivo deben ser distintos entre sí. Además, tanto el valor del selector como el constante deben ser del mismo tipo de dato (INT o BYTE). Su expresión sintáctica es la siguiente:

CASE *selector*
  opción$_1$
    $\mathbf{P}_1$
  ...
  opción$_i$
    $\mathbf{P}_i$
  ...
  ELSE
    $\mathbf{P}_n$

Se especifica la comparación del selector con cada una de las opciones. Si se encuentra un empatamiento, entonces se ejecuta el proceso asociado y el proceso selectivo termina. Cuando exista la posibilidad de que ninguna de las opciones previas empaten con el selector, se debe utilizar una forma de opción llamada ELSE, para garantizar que al menos un proceso será ejecutado, y de esta manera terminar con éxito.

Enseguida un ejemplo que ilustra el proceso constructor selectivo:

```
CASE letra
  a,e,i,o,u
    vocal := TRUE
  á,é,í,ó,ú
    acentuada := TRUE
  ELSE
    SEQ
      consonante := TRUE
      vocal := FALSE
      acentuada := FALSE
```

Para que el proceso constructor selectivo no falle se debe cumplir lo siguiente:

- Al menos debe existir un empatamiento.
- El proceso ejecutado, correspondiente a la opción en la cual se encontró empatamiento, debe finalizar su ejecución con éxito.

3.5 Procedimientos y Funciones

Procedimientos. La definición de un procedimiento en OCCAM permite la especificación de un nombre para un proceso o un conjunto de ellos. Su expresión sintáctica es la siguiente:

```
PROC nombre del proceso( parámetros formales )
  P
:
```

Se especifica el *nombre del proceso*, enseguida una lista de *parámetros formales*, la cual puede ser vacía, y después el cuerpo del procedimiento P, proceso al cual se le da un nombre. Los parámetros formales se pueden especificar por valor (en este caso se precede al tipo de dato la palabra reservada VAL) o por referencia.

En la declaración de un procedimiento, cada uno de los parámetros formales debe estar precedido por su tipo de dato; sin embargo, cuando varios parámetros tengan el mismo tipo de dato se deben agrupar y se debe especificar sólo una vez el tipo de dato que corresponda a cada grupo. Un parámetro formal es una especificación del parámetro actual, utilizada en una instanciación o llamada del procedimiento. Al momento de la instanciación, el cuerpo del proceso es ejecutado. El efecto de la instanciación es el de substituir el código del procedimiento a partir del punto en el que se le llama. La instanciación incluye la lista de parámetros actuales, cuyo número debe ser el mismo que el de parámetros formales, y corresponde además directamente con la lista de parámetros formales especificados.

Los procedimientos permiten también la construcción de módulos, cada uno compuesto de uno o más procesos, los cuales puede ser reutilizados en otros programas; de esta manera, se brinda la posibilidad de diseñar programas estructurados. Es importante tomar en cuenta que el lenguaje OCCAM no permite que los procedimientos sean recursivos.

Enseguida un ejemplo de declaración e instanciación de un procedimiento:

```
PROC Retardo( VAL INT tiempo.de.retardo ) -- Declaración
  TIMER reloj :
  INT tiempo :
  SEQ
    reloj ? tiempo
    reloj ? AFTER tiempo PLUS tiempo.de.retardo
:
SEQ
  canal.entrada ? dato
  Retardo(5000) -- Instanciación
  canal.salida ! dato
```

Funciones. Una función especifica un nombre para una clase especial de proceso, conocido como *proceso valor*. Su expresión sintáctica es la siguiente[20]:

```
Tipo de dato FUNCTION nombre( parámetros formales )
  VALOF
    P
    RESULT expresión
:
```

[20]Existen otras formas de declarar e instanciar una función, se sugiere consultar [22] para mayor información al respecto.

Se especifica el *tipo de dato*[21] del valor que será regresado por la función. Las funciones pueden regresar varios valores; de esta manera, sus tipos de datos tiene que ser especificados en la declaración y ser asignados a diferentes variables simultáneamente en la instanciación. Enseguida se tiene la palabra reservada `FUNCTION`, se continúa con el nombre de la función y posteriormente con una lista de *parámetros formales*, la cual puede ser vacía y cuyos parámetros se especifican únicamente por valor. Por último se encuentra el cuerpo de la función P, proceso valor al cual se le da un nombre.

Las funciones regresan un valor consecuentemente su instanciación se realiza en la forma de un operando en una expresión; además, se siguen las mismas reglas de instanciación como en los procedimientos. Las funciones se utilizan principalmente en algoritmos matemáticos que regresan un valor, comparten muchas de las bondades que ofrecen los procedimientos y facilitan la factorización de programas. Es importante señalar que no se permite incluir en una función procesos de envío o recepción, ni hacer asignaciones a variables que estén fuera de su contexto declarativo, ni utilizar tampoco los procesos constructores paralelo y alternativo.

Enseguida un ejemplo de declaración e instanciación de una función:

```
INT, INT FUNCTION Min.Max( VAL INT a, b ) -- Declaración
  INT mínimo, máximo :
  VALOF
    IF
      a ≥ b
        SEQ
          mínimo := b
          máximo := a
      b > a
        SEQ
          mínimo := a
          máximo := b
    RESULT mínimo, máximo
:
...
SEQ
  Lee( a, b )
  min, max := Min.Max( a, b ) -- Instanciación
```

[21]Una función puede solamente producir valores de tipos de datos primitivos.

3.6 OCCAM–Transputer

La filosofía de diseño de la arquitectura *Transputer* [24] se fundamenta tanto en el formalismo CSP como en el lenguaje OCCAM. Es decir, es una arquitectura que incorpora los elementos básicos de: *paralelismo, comunicación y sincronización, no determinismo y secuencialidad*, los cuales están implantados en el propio hardware.

Como se ha mencionado, un programa paralelo en OCCAM consiste en un conjunto de procesos ejecutándose simultáneamente, los cuales se comunican y sincronizan a través del paso de mensajes, utilizando canales lógicos de comunicación. De manera análoga, una red de procesadores consiste en un conjunto de Transputers, operando en forma autónoma y paralela, los cuales cooperan y se coordinan a través del paso de mensajes, utilizando enlaces físicos de comunicación. Por tanto, en el lenguaje OCCAM los conceptos abstractos de *proceso y canal* son equivalentes a los elementos físicos de *procesador y liga*, respectivamente, encontrados en el procesador Transputer.

Procesos y procesadores. El comportamiento externo de un procesador Transputer se puede comparar con el modelo formal de un proceso OCCAM. Un proceso es una unidad autónoma que puede ser ejecutado junto con otros procesos simultáneamente. Del mismo modo, cada procesador es una unidad física autónoma, cuya unión conforma un sistema de procesamiento paralelo, en donde cada Transputer ejecuta un proceso OCCAM.

Canales lógicos y ligas físicas. El concepto de un canal OCCAM se puede comparar con el de una liga física en un Transputer. Un canal lógico asocia a un par de procesos en una comunicación síncrona, unidireccional y sin capacidad de almacenamiento temporal. Del mismo modo, dos Transputers establecen comunicación por medio de una liga física que posee exactamente las mismas características antes mencionadas.

El Transputer constituye una implantación eficiente de OCCAM, ya que provee internamente (en el microcódigo del hardware) este lenguaje de programación y posee además las propiedades fundamentales para la ejecución paralela. Se utiliza además como la base para la construcción de sistemas de procesamiento paralelo y distribuido, mientras que el lenguaje OCCAM se utiliza como un formalismo asociado para el diseño de programas. Por tanto, cuando la arquitectura Transputer es programada en OCCAM, cada procesador implanta un proceso OCCAM y cada liga física implanta un canal lógico. En la figura 3.5 se aprecia gráficamente la relación de comportamiento *uno a uno* entre proceso $\equiv$ procesador y canal lógico $\equiv$ liga física.

Enseguida se describe y examina el procesador Transputer. En la sección 3.7.3 se profundiza el tema de la comunicación y sincronización entre procesadores Transputer.

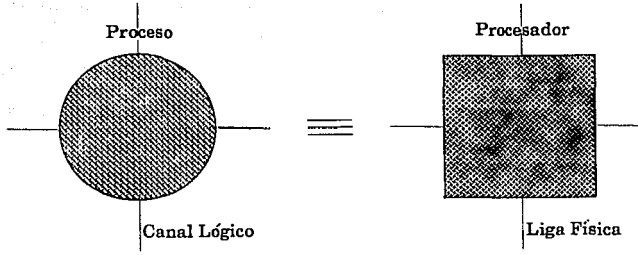


Figura 3.5: Equivalencia Proceso ≡ Procesador / Canal Lógico ≡ Liga Física

3.7 Descripción de la Arquitectura Transputer

Un Transputer es un dispositivo de *integración a gran escala* o VLSI[22] que incorpora básicamente en su arquitectura: una memoria local de muy rápido acceso, un procesador de alto rendimiento, un conjunto de ligas de comunicación[23] de gran velocidad, una inteface de aplicación específica para agregar más memoria externa y un módulo de servicios del sistema. En la figura 3.6 se muestra la arquitectura general de un Transputer.

La arquitectura Transputer es una tecnología que se caracteriza por su gran desempeño y velocidad, permitiendo el procesamiento paralelo a través de su propio hardware, es decir, que a este nivel, un Transputer posee la capacidad de procesamiento paralelo, utilizando la técnica de compartimiento de tiempo entre los procesos ejecutándose paralelamente. Además, gracias a su procesador de comunicaciones dedicado, el Transputer puede simultáneamente, por un lado, comunicar a través de sus ligas físicas, y por otro, ejecutar un proceso internamente.

Por su facilidad de interconexión con otros procesadores del mismo tipo, el Transputer puede ser utilizado como unidad de proceso para crear diversas topologías, obteniendo de esta manera sistemas de procesamiento paralelo y distribuido cada vez más complejos. Por tanto, un sistema de este tipo se construye a partir de una colección de Transputers, los cuales operan en forma paralela y asíncrona, comunicándose y sincronizándose por medio de sus ligas físicas de comunicación.

[22]Acrónimo de *"Very Large Scale Integration"* del término en Inglés.

[23]Son generalmente cuatro pares, en cada uno de los cuales, las ligas transmiten en direcciones opuestas).

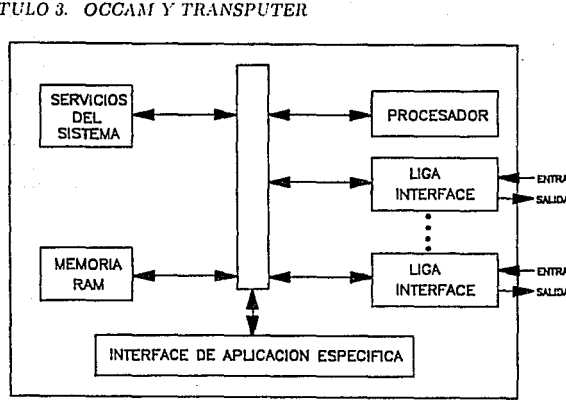


Figura 3.6: **Arquitectura General de un Transputer**

El término Transputer proviene de *"TRANSistor comPUTER"*, ya que este dispositivo se puede ver igualmente como un componente de silicón *(ej. transistor)*, así como una computadora completa. Constituye también un término genérico que describe a toda una familia de dispositivos programables del tipo VLSI que poseen una arquitectura común que integra: procesador, memoria y un conjunto de ligas de comunicación para la conexión con otros Transputers. Algunos de los dispositivos que incluye la familia Transputer son: controladores de disco, procesadores de punto flotante, de gráficas y de señales, interfaces, así como procesadores de propósito general de 32 y 64 bits, etc.

Una de las características más atractivas de la arquitectura Transputer es su bajo costo vs. su alto rendimiento; de esta manera, se convierte en una plataforma ideal para cualquier institución educativa que desee incorporar el estudio de la programación paralela sin tener que realizar altas inversiones, las cuales se podrían presentar al adquirir una mini o macrocomputadora o un conjunto de arquitecturas heterogéneas. Asimismo, se vuelve una opción ideal para la industria en donde se tienen problemas complejos que resolver, los cuales involucran principalmente cálculos intensivos de información con restricciones importantes en el tiempo de respuesta *(tiempo real)*; o problemas que requieren para su solución de una coordinación especial y compleja entre los procesos.

3.7.1 Manejo para el Procesamiento Secuencial.

El Transputer utiliza seis registros y un conjunto de instrucciones elementales para la ejecución de un proceso secuencial. Esto permite al procesador tener una lógica de control y un flujo de datos relativamente simple, pero veloz. Los seis registros son los siguientes:

- *Apuntador al espacio de trabajo.* Señala el área donde se almacenan las variables locales.
- *Apuntador de instrucción.* Señala la próxima instrucción a ser ejecutada.
- *Registro operando.* Usado en la formación de operandos de instrucciones de bajo nivel.
- *Registros A, B, C.* Forman una pila de evaluación para la mayoría de las operaciones aritmético-lógicas.

3.7.2 Manejo para el Procesamiento Paralelo.

El Transputer posee un despachador[24], implantado en el microcódigo del hardware, el cual habilita la ejecución simultánea de un número de procesos, compartiendo el tiempo del procesador. Esto elimina la necesidad de un núcleo de sistema operativo en software.

En cualquier momento un proceso ejecutándose simultáneamente puede estar:

- *Activo*
  1. siendo ejecutado.
  2. en una cola esperando ser ejecutado.
- *Inactivo*
  1. listo para una recepción.
  2. listo para un envío.
  3. esperando un tiempo específico.

Los procesos *activos* que serán ejecutados esperan en una cola. Se trata de una estructura de lista ligada, cuyos elementos son los espacios de trabajo de los procesos y que se implanta utilizando dos registos, estos son: *primero* y *último*, los cuales señalan al primer proceso y al último en la cola, respectivamente. El despachador no permite que los procesos *inactivos* consuman tiempo de procesador.

[24]Llamado también programador. Del término *"scheduler"* en Inglés.

Un proceso es ejecutado hasta el punto en que no pueda continuar debido a que tenga que esperar por una recepción, un envío o en un marcador de tiempo. Cuando un proceso llega a este punto, su apuntador de instrucción se almacena en su espacio de trabajo y se toma el próximo proceso en la cola para ser ejecutado.

En la figura 3.7 se aprecia gráficamente la ejecución simultánea de cinco procesos. El proceso T está inactivo esperando un tiempo específico, R está activo siendo ejecutado y P, Q y S están activos en cola esperando ser ejecutados.

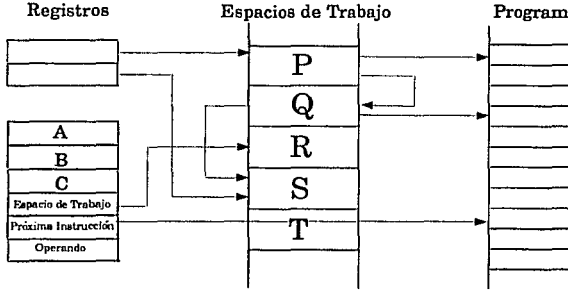


Figura 3.7: **Ejecución Simultánea de Procesos**

3.7.3 Comunicación y Sincronización entre Transputers

En una red de Transputers, los procesadores operan de manera asíncrona, comunicándose y sincronizándose entre sí a través de operaciones de envío y recepción, utilizando enlaces físicos de comunicación. El sistema de interface de liga del Transputer habilita el paso de mensajes para establecer una comunicación síncrona entre procesos implantados en distintos Transputers.

3.7.3.1 Canales Transputer

Dos procesadores en ejecución paralela se transmiten información el uno al otro a través de una liga física de comunicación, la cual posee las características siguientes:

- *Establece una conexión punto a punto.* Una liga permite enlazar a un par de procesadores.

- *Permite una comunicación unidireccional.* Una liga transmite la información de manera serial y únicamente en una sóla dirección, optimizando de esta manera la velocidad de transmisión. Las ligas comúnmente se agrupan en pares, esto es, dos ligas unidireccionales en sentidos opuestos.

- *Se le pueden aplicar dos operaciones: el envío y la recepción.* Para que sobre una liga se puedan transmitir mensajes, los procesos emisor y receptor ejecutándose uno en cada procesador, deben invocar a los procesos primitivos respectivos.

- *No posee bufferización.* Una liga no puede almacenar temporalmente ningún mensaje.

- *Tiene un protocolo síncrono tipo "rendezvous"* . Una liga establece una comunicación síncrona entre dos procesadores en ejecución paralela.

- *Es un objeto tipificado para el intercambio de información.* A una liga física se le asigna un canal lógico, el cual define un tipo de dato igual al del mensaje que se desea transmitir.

- *Constituye un objeto de comunicación confiable.* Se asegura que la comunicación sobre un canal físico será totalmente confiable en el sentido de que no existe la posibilidad de que algún mensaje pueda perderse o ser alterada su información.

En la figura 3.8 se aprecia gráficamente el concepto de una liga física.

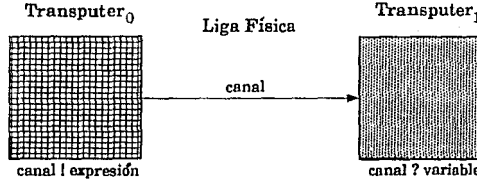


Figura 3.8: **Una Liga Física Entre Dos Transputers**

Un canal entre dos procesos, que se ejecutan en el mismo Transputer, es implantado como una palabra de memoria, mientras que un canal entre dos procesos ejecutándose en diferentes Transputers es implantado por una liga física. El Transputer utiliza dos operaciones de bajo nivel para llevar a cabo el paso de mensajes, estas son: *input message* y *output message.*

Un proceso ejecuta un envío o una recepción de mensaje, cargando en la pila de evaluación: el apuntador al mensaje, la dirección del canal y el contador del número de bytes a ser transmitidos, después ejecuta cualquiera de las operaciones de bajo nivel, según sea el caso.

Estas operaciones utilizan la dirección del canal para determinar si el canal es interno o externo, es decir, para saber si se trata de un canal lógico o una liga física, respectivamente. Esto significa que la misma secuencia de operaciones funciona para cualquier tipo de canal, permitiendo que un proceso sea escrito y compilado sin el conocimiento de como serán mapeados sus canales.

3.7.3.2 Comunicación por Canales Internos

Un canal interno puede tomar alguno de los dos valores siguientes: el *identificador* de un proceso o el valor *vacío*. Cuando se transmite un mensaje utilizando un canal interno: el identificador del primer proceso que llega a estar listo es almacenado en el canal, después este proceso pasa a un estado inactivo y por último el procesador inicia la ejecución del proceso siguiente en la cola. Cuando el segundo proceso llega a estar listo: el mensaje es copiado, enseguida, el primer proceso es agregado a la cola y el segundo continúa su ejecución, por último, el canal vuelve a su estado inicial. Este procedimiento es el mismo sin importar cual de los procesos, el emisor o el receptor, llega a estar listo primero.

En la figura 3.9 se muestra el paso de un mensaje utilizando un canal de comunicación interno.

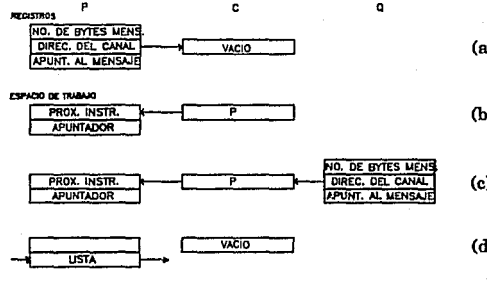


Figura 3.9: **Transmisión Sobre un Canal Interno**

Se especifica la comunicación entre dos procesos P y Q a través del canal interno C. El canal está inicialmente vacio (a). Supóngase que el proceso P es el primero en estar listo para la comunicación, entonces la pila de evaluación carga el apuntador al mensaje, la dirección del canal y el contador con el número de bytes del mensaje. Después, el canal toma el valor del identificador del proceso P, lo cual hará que el canal apunte al espacio de trabajo de P (b) y se almacena entonces el valor del registro *apuntador de instrucción* en su espacio de trabajo. En este momento, el proceso siguiente en la cola puede ser ejecutado. Cuando el proceso Q llega al punto en el cual está listo para comunicar, el mensaje es copiado, es decir, la comunicación se establece (c). Finalmente, el proceso P es agregado a la cola, Q continúa su ejecución y el canal C vuelve a su estado inicial (d).

3.7.3.3 Comunicación por Canales Externos

Si la dirección del canal, en una operación de envío o recepción, indica que se trata de un canal externo, entonces, el procesador delega a una interface de liga autónoma el trabajo de transferencia del mensaje. Mientras se realiza esta tarea, el proceso queda inactivo y el procesador puede continuar la ejecución de otros procesos en cola. Cuando el mensaje ha sido transferido, la interface de liga permite que el proceso quede activo en cola esperando ser ejecutado.

Cuando dos procesos enlazados para comunicación son ejecutados en Transputers adyacentes, ambos procesos son suspendidos y las dos interfaces de liga intercambian el mensaje en la dirección establecida. El protocolo utilizado para la transferencia está basado en la transmisión serial de bytes.

Cada interface de liga utiliza tres registros especiales, estos son: el apuntador al espacio de trabajo de los procesos, el apuntador al mensaje y el contador del número de bytes del mensaje. Esta característica permite que las acciones de la interface de liga no tengan interferencia directa con el procesador, como resultado de esta autonomía, el Transputer puede efectivamente procesar y comunicar al mismo tiempo.

En la figura 3.10 se muestra el paso de un mensaje utilizando un canal de comunicación externo.

Se especifica la comunicación entre dos procesos P y Q, ejecutándose en procesadores adyacentes, a través del canal externo C (a). Supóngase que P envía y Q recibe. Cuando P ejecuta su operación de envío, los registros en la interface de liga del procesador ejecutando P son inicializados y P queda inactivo. Lo mismo ocurre para Q cuando ejecuta su operación de recepción (b). Enseguida el mensaje es copiado a través de la liga física, una vez terminada la tarea de transferencia, los espacios de trabajo de los procesos P y Q regresan a la cola correspondiente (c).

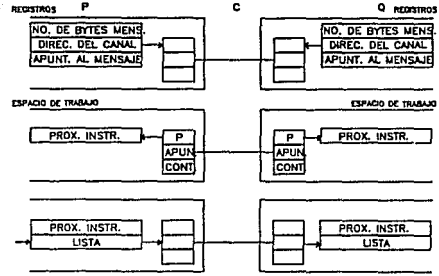


Figura 3.10: Transmisión Sobre un Canal Externo

3.7.3.4 Prioridad

El Transputer posee dos niveles de prioridad, por tanto, el proceso constructor PRI PAR sólo debe tener dos componentes. Enseguida un ejemplo:

```
PRI PAR
  PAR
    - -conjunto de procesos de prioridad alta
  PAR
    - -conjunto de procesos de prioridad baja
```

Se tienen implantadas en hardware dos colas de prioridades, cada una de las cuales tiene sus respectivos registros apuntadores al primer y último elemento. Asimismo, existe un algoritmo despachador, implantado para cada nivel de prioridad. Es evidente que un proceso con prioridad alta tiene mayor preferencia de ejecución que uno de baja; sin embargo un proceso con prioridad baja no podrá ser ejecutado, a menos que uno de alta no pueda proceder, ya sea porque haya terminado su ejecución, o porque haya quedado inactivo.

3.7.3.5 Ligas de Comunicación Físicas

Un mensaje es transferido byte por byte, cada uno de los cuales es confirmado mediante un acuse de recibo[25]; de esta manera, se requiere la presencia de un "buffer" de un byte en el Transputer receptor con la finalidad de asegurar que no habrá pérdida de información alguna. Nótese que cada liga se utiliza simultáneamente para transferir

[25]Del término *"acknowledge"* en Inglés.

datos en una dirección y confirmar mensajes en la otra, por tanto, cada liga lleva información de control y datos. El protocolo de la liga provee la comunicación síncrona en OCCAM.

Para transmitir un byte se conforma un paquete llamado *byte de datos* de la forma siguiente: primero un bit de inicio, seguido por un bit 1, seguido por los ocho bits de datos (byte) y por último un bit de paro. En la figura 3.11 se muestra un byte de datos y una señal de acuse de recibo.

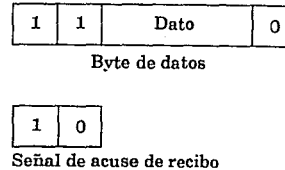


Figura 3.11: **Byte de Datos y Señal de Acuse de Recibo**

Después de haber transmitido el byte de datos, el emisor espera el arribo de un acuse de recibo. Este acuse consiste de un bit de inicio seguido por un bit 0. El acuse significa, por un lado, que el proceso recibió el byte sin problema, y por otro, que la liga conectada al receptor está lista para recibir un nuevo byte. Los mensajes pueden ser transmitidos a 10 Mbytes/seg. aprox. a través de un canal interno y a 2 Mbytes/seg. aprox. a través de un canal externo.

3.7.4 Topologías Creadas con Transputers

El patrón de conexión de los procesadores es conocido como topología, red o configuración. La arquitectura Transputer permite la construcción de una variedad muy amplia de configuraciones diferentes, conectando entre sí a los procesadores a través de sus ligas físicas. Algunos ejemplos podrían ser: hipercubo, árbol, anillo, "bus", "pipeline", malla, etc. En sí, no hay nada que restringa la construcción de cualquier tipo de topología, excepto el número de ligas que existen en cada Transputer. En la figura 3.12 se aprecian algunos ejemplos de topologías que pueden ser creadas con Transputers.

Es importante mencionar que el diseño de una topología puede llegar a ser, en algunas ocasiones, un trabajo difícil o complejo, ya que lo ideal es lograr que el desempeño de ejecución sea lo más eficiente posible, lo cual podría lograrse, por un lado, analizando la estructura de comunicación de los procesos para después reflejar esta estructura en una configuración de Transputers, y por otro lado, realizando un mapeo adecuado sobre esta configuración del modelo de solución; sin embargo, esto no es un

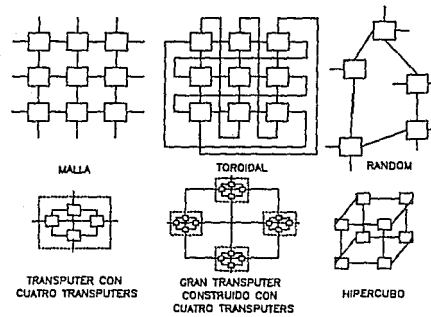


Figura 3.12: **Topologías Creadas con Transputers**

trabajo fácil, por lo que se debe recurrir generalmente al auxilio de algunas técnicas tales como: teoría del paralelismo, teoría de gráficas, teoría de colas, etc.

El número de ligas en un Transputer son suficientes para permitir un rango muy amplio de configuraciones útiles; sin embargo, existen problemas cuyos modelos de solución demandan un número mayor de conexiones, éste puede ser el caso de problemas en espacios n-dimensionales, el modelo Cliente-Servidor, etc. Para resolver este problema existen, hoy en día, sistemas comerciales y prototipos en desarrollo que permiten la reconfiguración dinámica de una arquitectura, ya sea vía software y hardware o automáticamente.

3.7.5 Generaciones del Transputer

En la tabla 3.2 se muestra la evolución de la arquitectura Transputer a través del desarrollo de tres generaciones, obsérvese el avance en cuanto al desempeño del procesador, manejo y capacidad de la memoria y el sistema de comunicaciones.

3.7.5.1 Generación Transputer T9000

La familia Transputer T9000 [26] ofrece un mayor avance en el cómputo paralelo, en el multiprocesamiento y en las comunicaciones de alta velocidad. Presenta una arquitectura en la cual incorpora un procesador de canales virtuales que ofrece la posibilidad de tener múltiples canales compartiendo una sóla liga física de comunicación, rom-

Primera Generación Transputer T414	Segunda Generación Transputer T800	Tercera Generación Transputer T9000
Nace en 1985	Nace en 1987	Nace en 1991
Procesador de 32 bits con 20 MIPS	Procesador de 32 bits con 30 MIPS	Procesador de 32 bits con 200 MIPS
2 Kb RAM en el Chip	4 Kb RAM en el Chip	16 Kb Memoria Cache
4 Gb espacio de direccionamiento de memoria	4 Gb espacio de direccionamiento de memoria	4 Gb espacio de direccionamiento de memoria
4 Ligas seriales de 20 Mbits / seg.	4 Ligas seriales de 20 Mbits / seg.	4 Ligas seriales de 100 Mbits / seg.
Reloj interno 15 Mhz.	Reloj interno 30 Mhz.	Reloj interno 50 Mhz.
	Unidad de punto flotante de 64 bits con 2.25 Mflops pico	Unidad de punto flotante de 64 bits con 25 Mflops pico
	Soporte de gráficas	Soporte de gráficas
		Procesadores en arreglo "pipeline"
		Procesador de canales virtuales

Tabla 3.2: **Generaciones de la Arquitectura Transputer**

piendo con el esquema lógico de conexión punto a punto. En la figura 3.13 se aprecia gráficamente una liga física compartiendo un conjunto de canales lógicos. Incorpora también una memoria *"cache"* y un procesador de tipo *"pipeline"* logrando aumentar en forma significativa la velocidad de procesamiento. Esta nueva generación mantiene la compatibilidad con los productos Transputer ya existentes. En la figura 3.14 se muestra la arquitectura del Transputer T9000.

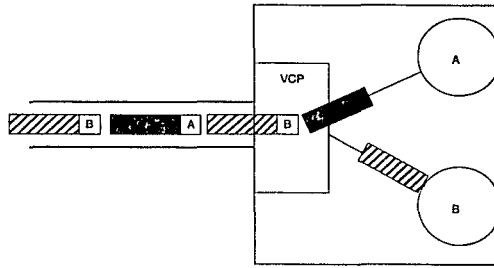


Figura 3.13: **Liga Compartiendo Múltiples Canales**

3.7.6 Programación en la Arquitectura Transputer

3.7.6.1 Sistema de Desarrollo Integral TDS

El Sistema de Desarrollo Transputer o TDS[26] es un ambiente de programación que está compuesto por un conjunto de herramientas de desarrollo que permiten: la verificación, la compilación, el ligado y la ejecución de programas OCCAM en un Transputer o en una red de ellos.

Una red de Transputers puede ser instalada sobre varios tipos de plataformas, que van desde una IBM PC/AT hasta una DEC VAX, a las cuales se les conoce como *computadoras anfitrión*, ya que permiten únicamente alojar los procesadores, así como utilizar sus recursos periféricos tales como: unidades de almacenamiento secundario (discos, cintas, etc.), el teclado y la pantalla. En la figura 3.15 se aprecia gráficamente la interacción del Transputer con la computadora anfitrión.

El TDS interactúa con la computadora anfitrión para utilizar sus recursos. De hecho, la computadora anfitrión al ser ligada con el Transputer tiene un comportamiento

[26]Acrónimo de *"Transputer Development System"* [25] del término en Inglés.

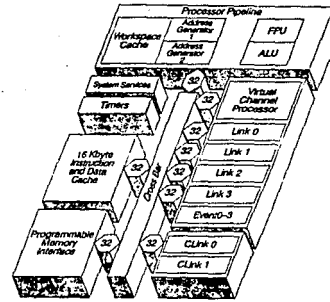


Figura 3.14: **Arquitectura del Transputer T9000**

análogo al de un proceso y su interface opera como un canal de datos serial al Transputer. Es importante mencionar que únicamente un Transputer, llamado *raíz*, es el que puede accesar directamente los recursos de la computadora anfitrión. Si otros Transputers conectados en la red desean utilizar estos recursos, entonces deberán hacerlo a través del Transputer raíz.

3.7.6.2 Programación en un Transputer

Enseguida se describen las etapas del proceso de desarrollo de programas OCCAM, utilizando las herramientas del TDS.

Verificación. Una vez escrito el código de un programa OCCAM, éste debe pasar por un proceso de verificación sintáctica y semántica antes de ser compilado. En la verificación se pueden detectar y reportar los errores cometidos en la codificación del programa; de esta manera, se obtiene una explicación más detallada de la que ofrece el compilador. Para este fin se utiliza la herramienta ICHEK. Enseguida un ejemplo:

ICHECK <nombre del programa>.OCC

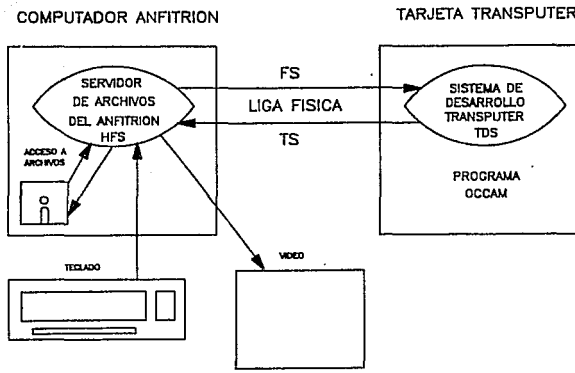


Figura 3.15: **Interacción del Transputer con el Anfitrión**

Compilación. Una vez verificado el programa, éste se encuentra listo para ser compilado. Para este fin se utiliza la herramienta `OCCAM`. El compilador puede producir código según el modelo[27] de Transputer sobre el cual será ejecutado el programa, esto se indica a través de un parámetro. Enseguida un ejemplo:

OCCAM /T8 <nombre del programa>.OCC

El compilador genera un código que será almacenado en un archivo llamado *<nombre del programa>.T8I*[28]. Existe además otra herramienta llamada `ILIBR`, la cual permite reunir código para ser utilizado como biblioteca. De esta manera, se pueden construir bibliotecas de módulos de procesos que pueden ser posteriormente reutilizados. Enseguida un ejemplo:

ILIBR <nombre de la biblioteca>.OCC

Esta herramienta genera un código que será almacenado en un archivo llamado *<nombre de la biblioteca>.LIB*.

[27] El *modelo* se refiere al tipo de procesador disponible (ej.: T200, T400, T800, etc.). Se sugiere consultar [24] para encontrar mayor información sobre los diferentes modelos de procesadores Transputer existentes.

[28] En este ejemplo se asume que el programa será ejecutado sobre un Transputer del tipo T800.

Ligado. Una vez compilado el programa, éste se encuentra listo para ser ligado con todas aquellas bibliotecas que requiera o con otras unidades de programa (procesos) ya compiladas previamente. Para este fin se utiliza la herramienta `ILINK`. Enseguida un ejemplo:

ILINK /F <nombre de archivo>.LNK

El parámetro `/F` indica la utilización de un archivo, el cual contiene el nombre del programa a ser ligado, junto con todos los nombres de las bibliotecas y/o procesos necesarios. El contenido de este archivo podría ser:

<nombre del programa>.T8H
<nombre de biblioteca$_1$>.LIB
...
<nombre de biblioteca$_n$>.LIB

El ligador genera un código que será almacenado en un archivo llamado *<nombre del programa>.C8H*.

Ejecución. Una vez ligado el programa, éste se encuentra listo para que se le agrege código de inicialización; y de esta manera ser ejecutado en un Transputer. Para este fin se utiliza la herramienta `IBOOT`. Enseguida un ejemplo:

IBOOT <nombre del programa>.C8H

Esta herramienta genera un código binario que será almacenado en un archivo llamado *<nombre del programa>.B8H*, el cual se encuentra listo para ser cargado en el Transputer y habilitar su ejecución. Por último, para este fin se utiliza la herramienta `ISERVER`[29]. Enseguida un ejemplo:

ISERVER <nombre del programa>.B8H

3.7.6.3 Programación en una Red de Transputers

Dada la relación *uno a uno* entre proceso ≡ procesador y canal lógico ≡ liga física, se tiene la posibilidad de que un programa pueda ser ejecutado en un Transputer o en una red de ellos, especificando únicamente el mapeo de los procesos sobre los procesadores y el de los canales lógicos sobre las ligas físicas a través de un archivo de configuración.

[29]Esta herramienta utiliza generalmente algunos parámetros para la inicialización del Transputer. Se sugiere consultar [24] para mayor información al respecto.

Una de las principales características de OCCAM-Transputer es que al construir un programa paralelo no es indispensable conocer cual será la configuración final de procesadores sobre la cual será ejecutado éste, ya que es posible ejecutarlo en un procesador, aunque su diseño involucre la utilización de muchos de ellos, siendo la única limitante la cantidad de memoria disponible.

En la figura 3.16 se aprecia gráficamente el mapeo de un conjunto de procesos para su ejecución en un Transputer (a) y en una red de ellos (b).

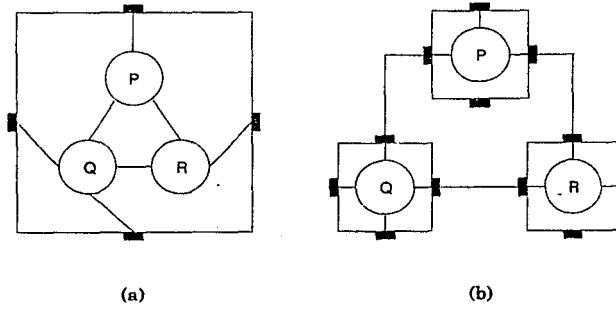


Figura 3.16: **Mapeo de Procesos en un Transputer y en una Red de Ellos**

Planeación de la descripción de la configuración. La configuración asocia los componentes de un programa en OCCAM con un conjunto de recursos físicos. Durante la configuración, los procesos de un programa son mapeados sobre un conjunto de dispositivos de procesamiento interconectados, los cuales están disponibles en el ambiente en el cual se ejecuta el programa. Asimismo, los canales lógicos que interconectan los procesos son mapeados a las ligas físicas que comunican a los dispositivos de procesamiento.

Esta configuración debe ser codificada en un archivo separado, al igual como se codifica un programa OCCAM. Sin embargo, para realizar esta tarea es indispensable la utilización de un proceso especial llamado *colocador paralelo.*

Ejecución sobre varios procesadores. Tanto los procesos componentes de un proceso constructor paralelo como los canales lógicos que interconectan estos procesos pueden ser mapeados en el hardware de los procesadores, utilizando el *colocador paralelo* PLACED PAR. Su expresión sintáctica es la siguiente:

PLACED PAR
  PROCESSOR *n modelo*
    PLACE *canal lógico*$_1$ **AT** *liga física*$_1$:
    ...
    PLACE *canal lógico*$_n$ **AT** *liga física*$_n$:
    nombre proceso(*canal lógico*$_1$,...,*canal lógico*$_n$)

El mapeo de los procesos sobre los procesadores restringe a que haya únicamente un proceso por procesador. Sin embargo, es claro que cada proceso puede estar compuesto de otros procesos internamente.

Enseguida un ejemplo que ilustra el colocador paralelo:

```
PLACED PAR
  PROCESSOR 0 T800
    PLACE pipe AT Link1.out :
    Envía( pipe )
  PROCESSOR 1 T800
    PLACE pipe AT Link3.in :
    Recibe( pipe )
```

Se especifica el mapeo de los procesos *Envía* y *Recibe* sobre los Transputers 0 y 1 del tipo T800, respectivamente. Ambos procesos se comunican a través del canal *pipe*, el cual es mapeado a una liga física que conecta a los procesadores.

En la figura 3.17 se aprecia gráficamente la configuración de Transputers necesaria para la ejecución del programa anterior.

El colocador paralelo puede también ser replicado con la finalidad de producir un número similar de procesos, los cuales serán mapeados al mismo número de procesadores. Enseguida un ejemplo:

```
PLACED PAR i = 0 FOR 10
  PROCESSOR i T800
    PLACE pipe[i] AT Link1.out :
    PLACE pipe[i+1] AT Link3.in :
    Transmite( pipe[i], pipe[i+1] )
```

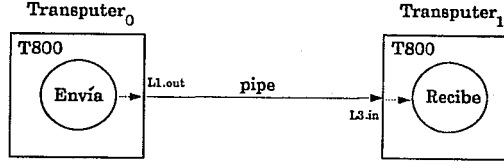


Figura 3.17: **Ejecución Sobre Dos Procesadores**

Se especifica el mapeo de los procesos *Transmite(pipe[0], pipe[1]),...*, *Transmite(pipe[9], pipe[10])* sobre los Transputers 0,...,9 del tipo T800, respectivamente. El procesador *i* recibe su mensaje por *pipe[i]* y lo envía por *pipe[i+1]*. Estos canales son mapeados a dos ligas físicas del procesador, formando una estructura de tipo *"pipeline"* en la cual la salida de cada procesador se conecta a la entrada del siguiente.

En la figura 3.18 se aprecia gráficamente la configuración de Transputers necesaria para la ejecución del programa anterior.

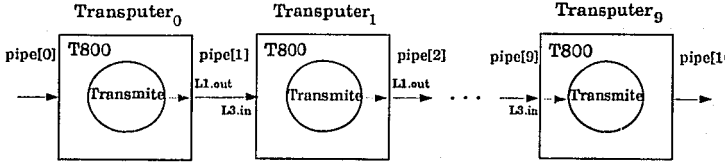
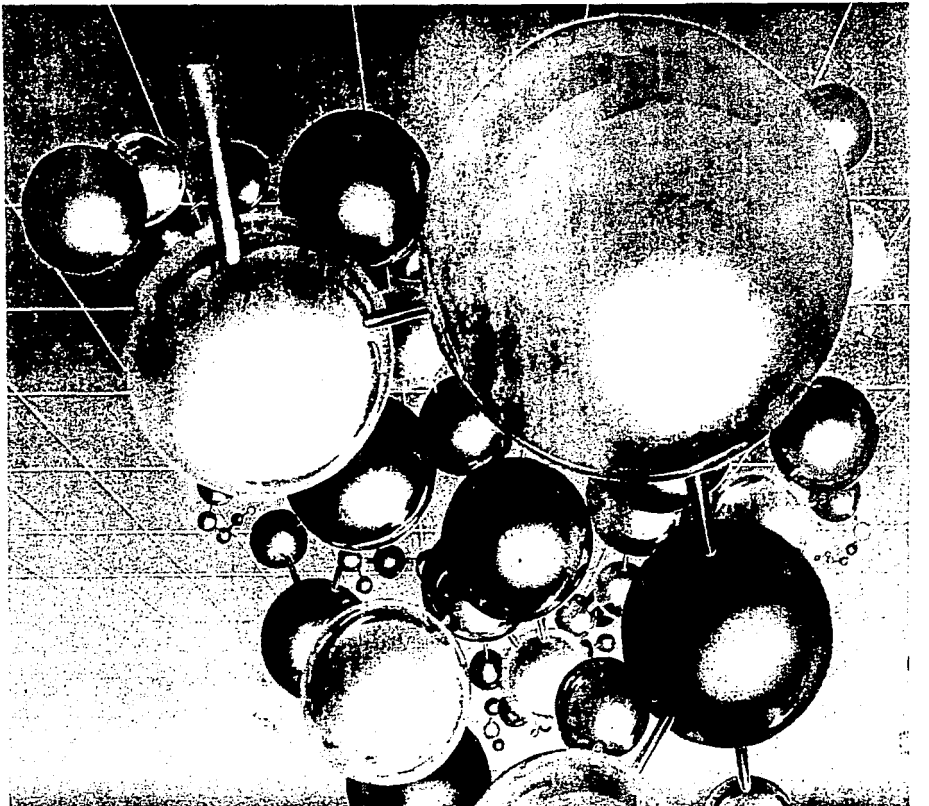


Figura 3.18: **Ejecución Sobre Varios Procesadores**

Una vez construido el archivo de configuración, éste debe ser compilado, obteniendo de esta manera código ejecutable para redes de Transputers. Para este fin se utiliza la herramienta `ICONF`. Enseguida un ejemplo:

ICONF <nombre del archivo de configuración>.PGM



Esta herramienta genera un código ejecutable que será almacenado en un archivo llamado *<nombre de archivo configuración>.BTL*, el cual se encuentra listo para ser cargado en una red de Transputers. Finalmente, la herramienta que permite cargar el código en la red y habilitar su ejecución es:

ISERVER <nombre del archivo de configuración>.BTL

3.7.7 Áreas de Aplicación

Desde el surgimiento de la primera generación de la arquitectura Transputer se han desarrollado numerosas y diversas aplicaciones en áreas tales como: análisis numérico, simulación, control industrial, inteligencia artificial, reconocimiento de patrones, voz y señales, procesamiento en tiempo real, procesamiento de imágenes, robótica, neurociencia, graficación, física, comunicaciones, sistemas operativos, bases de datos, supercómputo, etc.

Capítulo 4

Nuevos Mecanismos para la Comunicación y Sincronización

En este capítulo se consideran las propiedades analizadas y examinadas de OCCAM-Transputer para hacer una crítica a esta pareja lenguaje-arquitectura, lo cual permitirá justificar el objetivo del trabajo. Enseguida se realiza la construcción de los nuevos mecanismos para la comunicación y sincronización, describiendo el proceso de análisis y especificación, diseño conceptual, codificación e implantación de cada uno de ellos. Asimismo, se ilustran las ventajas y propiedades de los mecanismos creados, resolviendo algunos de los problemas clásicos encontrados en programación paralela y distribuida, los cuales son ejecutados en un Transputer y en una red de ellos. Finalmente, se describe la construcción de una *biblioteca de software*, la cual reune todos estos mecanismos.

4.1 Crítica a OCCAM-Transputer

Para facilitar la implantación de aplicaciones paralelas y distribuidas, se requiere contar con arquitecturas paralelas y lenguajes de programación que permitan expresar de manera clara y flexible la simultaneidad y la coordinación de los procesos involucrados en una aplicación. Algunas alternativas en arquitecturas y lenguajes, que existen hoy en día, ofrecen mecanismos para el control de la simultaneidad y la coordinación de procesos a través de mecanismos muy elementales, tal es el caso de *OCCAM-Transputer*.

OCCAM es un lenguaje de programación basado en el concepto de ejecución paralela, el cual permite la expresividad del paralelismo de forma inherente en la solución de un problema. Transputer es un procesador de alto desempeño y gran velocidad, el cual permite el procesamiento paralelo a través de su propio hardware. Además, como unidad de proceso es ideal para construir redes o arreglos de procesadores, obteniendo sistemas de procesamiento paralelo y distribuido cada vez más complejos.

Gracias a la sencillez de la semántica del lenguaje OCCAM, la cooperación, coordinación e interacción de los diferentes componentes de un sistema paralelo y distribuido se pueden expresar de una forma muy simple. Sin embargo, aunque se trata de un lenguaje elegante y de gran poder de expresividad, su filosofía de diseño no deja de ser elemental y de bajo nivel, ya que posee principalmente las limitaciones siguientes:

- Contiene un conjunto muy elemental y reducido de procesos (primitivos y constructores), lo cual puede conducir a un estilo programación restringido en la obtención de grandes sistemas paralelos y distribuidos.
- Carece de mecanismos para estructurar los datos, esto es, para la creación de estructuras tales como: registros, apuntadores, listas, arboles, etc.
- No permite la recursividad en los procesos.
- No maneja el concepto de abstracción de datos usando módulos (Tipos de Datos Abstractos)[1].
- La creación de procesos es estática.
- No permite el manejo de excepciones para el control de errores en tiempo de ejecución.
- El balanceo de carga o mapeo de los procesos sobre los procesadores es un trabajo que debe ser realizado por el programador[2].
- El concepto de comunicación y sincronización entre procesos está basado en un sólo mecanismo: el *canal*, el cual se establece como un mecanismo básico y de bajo nivel.

En este trabajo se presenta una alternativa de solución a este último punto, creando y construyendo nuevos mecanismos para la comunicación y sincronización, los cuales brindarán al programador mayor versatilidad, flexibilidad y eficiencia en el desarrollo de aplicaciones paralelas y distribuidas complejas sobre una red de procesadores Transputer.

[1]Las únicas formas de encapsulamiento son: *procedimientos* y *funciones*.

[2]Aunque en la actualidad existen herramientas que realizan la tarea del mapeo en forma automática, éstas no garantizan que se haga de forma óptima.

4.1.1 Patrones de Comunicación para la Interacción entre Procesos

Dada la especificación de un problema, es indispensable tomar decisiones de cuántos procesos serán utilizados y cómo van a interactuar entre ellos. Estas decisiones dependen de la aplicación misma y del hardware en donde vaya a ejecutarse. Uno de los problemas más críticos es el de asegurar que el comportamiento de ejecución de los diferentes procesos en una aplicación será sincronizado de forma apropiada.

Existen diversos tipos de patrones de cooperación, coordinación e interacción entre procesos basados en paso de mensajes, los cuales se encuentran incorporados en diferentes lenguajes de programación paralela y distribuida [50] [45] [1] y que se implantan a través de mecanismos. De esta manera, cada mecanismo corresponde a un patrón de comunicación y sincronización entre procesos, el cual se utiliza para resolver una variedad de problemas.

Los lenguajes basados en paso de mensajes permiten la construcción de programas (cuyos procesos comparten canales), habilitando así un camino de comunicación entre los procesos. Cada lenguaje incorpora uno o más de estos mecanismos (junto con sus primitivas de envío y recepción correspondientes), los cuales constituyen diferentes patrones de comunicación y sincronización. De alguna forma, estos mecanismos varían entre sí por la forma en como se declaran, se utilizan y permiten la comunicación y la sincronización entre procesos.

Con base en las experiencias [36] [55] [56] [37] [38] obtenidas al trabajar con OCCAM-Transputer, en donde el paralelismo está basado en un concepto formal y bien fundamentado, se ha comprobado que esta pareja lenguaje-arquitectura presenta características interesantes que permiten la especificación, diseño, desarrollo, implantación y verificación de sistemas paralelos y distribuidos. La estructuración jerárquica de procesos, junto con los mecanismos para la ejecución simultánea, el paso de mensajes y el control al no determinismo que posee OCCAM-Transputer, pueden llegar a ser elementos suficientes para la construcción de muchas aplicaciones.

Ahora bien, OCCAM-Transputer incorpora el canal como el único mecanismo para el control de la simultaneidad, la cooperación y la coordinación de un conjunto de procesos. Dadas las características generales de un canal de comunicación lógico y físico, especificadas en el capítulo 3, sección 3.3.2 y 3.7.3, respectivamente, se puede apreciar que el canal se establece como un mecanismo básico, el cual constituye un objeto de comunicación y sincronización de bajo nivel, desde el momento en que se requiere de una interacción de procesos mucho más compleja y elaborada. Consecuentemente, el programador de este tipo de aplicaciones siempre necesita de un esfuerzo mayor para lograr que su programación sea versátil, flexible y eficiente.

Por ejemplo, si se desean resolver problemas que requieran de patrones de comunicación y sincronización entre procesos mucho más complejos y elaborados, como podrían ser los casos de: *conexiones multipunto*; esquemas tipo *cliente-servidor* y

"pipeline"; *multiplexación* y *"bufferización"* de mensajes; comunicación *bidireccional* y en *difusión*; etc., esto se convierte prácticamente en una tarea extra y, en la mayoría de las ocasiones, difícil para el programador, ya que la única forma de poder conseguirlo es programando todos estos patrones, utilizando y basándose en el único mecanismo incorporado en OCCAM-Transputer para la comunicación y sincronización: el *canal*.

4.2 Objetivos de Diseño y Desarrollo

Con todo lo expuesto anteriormente se justifica la importancia del poder contar con mecanismos mucho más elaborados y estructurados, es decir, de más alto nivel, que permitan al programador de aplicaciones una mayor versatilidad, flexibilidad y eficiencia en la programación de sistemas paralelos y distribuidos complejos a ser implantados en una red de procesadores Transputer. Por tanto, los objetivos de diseño y desarrollo que se han planteado en este trabajo son los siguientes:

- Partir del único mecanismo incorporado en OCCAM-Transputer para la coordinación e interacción de los procesos: el *canal*.
- Basar el diseño de los nuevos mecanismos en ideas de otros mecanismos de comunicación y sincronización incorporados en diferentes lenguajes de programación de sistemas paralelos y distribuidos.
- Realizar el proceso de análisis y especificación, diseño conceptual y codificación para cada uno de los nuevos mecanismos.
- Resolver un problema en programación paralela y distribuida, para cada mecanismo creado, haciendo la implantación tanto en un procesador Transputer como en una red de ellos.
- Hacer la implantación creando una biblioteca de software con los nuevos mecanismos.
- Describir los resultados obtenidos en la implantación, los alcances logrados con la utilización de los nuevos mecanismos y las perspectivas de desarrollo.

Limitaciones en el Mapeo. En cuanto a la implantación, en una red de Transputers, de un sistema paralelo y distribuido escrito en OCCAM, existe el problema de cómo realizar el mapeo de los procesos en los procesadores de la forma más óptima, tomando en cuenta factores tales como: la minimización del tiempo de ejecución, la conservación del paralelismo intrínseco en la aplicación y la reducción del tiempo y costo de la comunicación.

En ciertas ocasiones, un conjunto de procesos puede llegar a ser mapeados en forma eficiente sobre la red de procesadores, dando al programador el control total y adecuado de los recursos. Sin embargo, en la mayoría de las ocasiones, el programador no es tan afortunado y difícilmente consigue un mapeo óptimo.

Si bien, OCCAM es un lenguaje apropiado para expresar sistemas paralelos, se vuelve rudimentario en el momento de expresar el mapeo de sistemas: que involucren una jerarquización compleja (de múltiples niveles) de procesos, o una comunicación entre procesos utilizando múltiples canales. Todo esto, tomando en consideración el límite físico que impone el hardware por el número de ligas de comunicación existentes. De este modo, el programador se enfrenta al problema de definir explícitamente la distribución tanto de los procesos en cada uno de los procesadores disponibles en la red, como de los canales lógicos en las ligas físicas de cada uno de los Transputers.

Una solución a este problema se obtiene utilizando una arquitectura reconfigurable dinámicamente, la cual incorpora un conjunto de dispositivos electrónicos programables, llamados *conmutadores de liga*[3] que permiten la interconexión automática de los procesadores, construyendo configuraciones particulares, de acuerdo a los modelos de solución de las distintas aplicaciones; no obstante, este tipo de arquitecturas no garantizan que el mapeo se realice en forma óptima.

No es intensión en este trabajo dar una solución al problema del mapeo; sin embargo, algunos de los mecanismos creados tratan con esta problemática. Asimismo, es importante mencionar que tanto el mapeo de todos los programas de ejemplo presentados a lo largo del trabajo como la configuración de la red de procesadores en los cuales se ejecutan estos, fue determinado por el autor.

4.3 Nuevos Mecanismos de Comunicación y Sincronización

En el capítulo 1 de las secciones 1.3.1.2 a la 1.3.1.4 se han especificado algunos de los principales mecanismos de comunicación y sincronización encontrados en programación paralela y distribuida. Asimismo, se han establecido los tipos esenciales de comunicación, esto es, síncrona y asíncrona. En el capítulo 3 se ha descrito y examinado el *canal* como el único mecanismo incorporado en el lenguaje OCCAM, y cuyas características enseguida se destacan:

- Comunica de forma directa a dos procesos, es decir, punto a punto.
- Provee comunicación en un sólo sentido.
- No posee capacidad de bufferizar mensajes.

[3]Del término *"link switches"* en Inglés.

- Establece una comunicación síncrona.
- Constituye un mecanismo de comunicación confiable.
- Es un objeto con un tipo de dato asociado.
- Tanto la comunicación como la sincronización se encuentran unificadas en el mismo mecanismo.
- El acceso se realiza utilizando las primitivas de envío y recepción.

Con base en el canal, los nuevos mecanismos desarrollados fueron los siguientes:

- Filtro
- Puente
- "Buffer"
- Multiplexor "Muchos a Uno"
- Multiplexor "Uno a Muchos"
- Difusor

Enseguida se describe el proceso de análisis y especificación, diseño conceptual, codificación[4] e implantación de cada uno de los nuevos mecanismos desarrollados[5]. Asimismo, se presenta, para cada uno de ellos, un ejemplo que ilustra sus características de utilización[6]. Estas propuestas han sido fundamentadas en otros mecanismos implantados en diferentes lenguajes de programación paralela y distribuida [50] [45] [1].

[4]En el apéndice A se muestra la codificación, en lenguaje OCCAM, de todos los mecanismos creados, implantados en una *biblioteca de software*.

[5]La versión final de cada uno de estos mecanismos incorpora el control para la terminación con éxito.

[6]Se anexa a este trabajo un diskette que incluye la codificación, en lenguaje OCCAM, de todos los ejemplos presentados en el capítulo. Las versiones de estas aplicaciones incorporan procesos que permiten la interface con el exterior.

4.3.1 Mecanismo Filtro

4.3.1.1 Análisis y Especificaciones

Los procesos componentes en un sistema paralelo y distribuido cooperan, se coordinan e interactúan entre sí. Cada proceso posee un número determinado de canales de entrada y de salida, los cuales permiten la comunicación con otros procesos distintos. Sin embargo, un caso particular de proceso es aquel que posee sólo un canal de entrada y uno de salida. Su propósito es obtener un mensaje de salida como una función del mensaje de entrada (y = F(x)). En la figura 4.1 se aprecia gráficamente este caso:

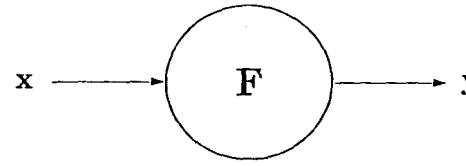


Figura 4.1: **Proceso y = F(x)**

Nótese que este proceso especifica un mecanismo que recibe mensajes por un canal de entrada, los cuales modifica, uno a la vez, para después enviarlos por un canal de salida. Bajo esta idea se propone:

> Construir un mecanismo que permita establecer una alternativa de comunicación punto a punto, unidireccional, asíncrona y cuyo valor de salida sea función del valor de entrada, esto es, un *filtro*.

4.3.1.2 Diseño Conceptual

El cuerpo del proceso que implanta el mecanismo *Filtro* está constituido simplemente por una invocación a otro proceso llamado *"función"* (definido por el programador), el cual deberá incorporar en su encabezado los mismos parámetros que el *Filtro* y en su cuerpo de proceso: una operación de recepción que permita ingresar cada vez un mensaje, enseguida el código necesario para la transformación de éste y por último, una operación de envío que permita mandarlo de salida. Para que el *Filtro* pueda efectuar la invocación al proceso *"función"*, se incorpora la inclusión externa del código compilado de este proceso.

El *Filtro* es accesado por dos canales: uno de entrada y otro de salida, implantados a través de *canal.entrada* y *canal.salida*, respectivamente. Los mensajes son recibidos por *canal.entrada*, estos pasan por el proceso *"función"* que los transforma y finalmente son enviados por *canal.salida*.

Se debe considerar la condición de sincronización siguiente: el proceso conectado al canal de salida del *Filtro* no podrá recibir ningún mensaje si el proceso conectado al canal de entrada aún no lo ha enviado.

En la figura 4.2 se aprecia gráficamente el comportamiento del mecanismo *Filtro*.

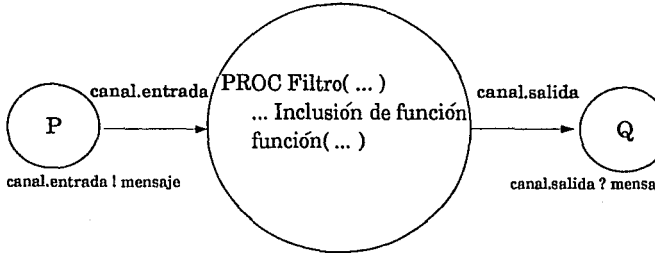


Figura 4.2: **Comportamiento del Mecanismo Filtro**

4.3.1.3 Codificación e Implantación

El encabezado del proceso que implanta el mecanismo *Filtro* incluye los parámetros correspondientes a los canales *canal.entrada* y *canal.salida*, cuyos protocolos son *anárquicos*[7]. Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo, el programador deberá construir y compilar previamente el proceso "función".

Enseguida se muestra la especificación del proceso *Filtro*:

```
PROC Filtro( CHAN OF ANY canal.entrada, canal.salida )
  ... Inclusión externa del código compilado del
      proceso "función" (definido por el programador)
  función( canal.entrada, canal.salida )
```

[7]Se refiere a dejar indefinido el formato del protocolo (`CHAN OF ANY`). Ver Capítulo 3, sección 3.3.2.5.

Enseguida se muestra la especificación del caso general de un proceso *"función"*:

```
PROC función( CHAN OF ANY canal.entrada, canal.salida )
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    -- Se recibe un mensaje por canal.entrada
    canal.entrada ? mensaje
    ... Se transforma este mensaje
    -- Se envía el mensaje modificado por canal.salida
    canal.salida ! mensaje
:
```

4.3.1.4 El Problema de los Números Primos

Especificación. Se desea generar una secuencia de números primos usando una criba de Eratóstenes. La criba está formada por un conjunto de *Filtros* conectados en un arreglo tipo *"pipeline"*. Los *Filtros* están originalmente vacíos, es decir, sin ningún valor almacenado. Cada *Filtro* recibe de su predecesor un número. Si en el momento de la recepción, un *Filtro* está vacío, el número es almacenado y considerado como *primo*. En caso contrario, quiere decir que ya existe un número primo almacenado y por lo tanto es indispensable determinar si ese nuevo número recibido es múltiplo del primo existente. Si es múltiplo, entonces se descarta, sino, entonces se envía al *Filtro* sucesor.

La criba obtiene los números de un *generador de números naturales*, el cual se conecta al primer *Filtro*. Este generador alimenta de valores consecutivos a la criba. Al final de la generación, los valores que permanecen almacenados en los *Filtros* conformarán la secuencia de números primos. En la figura 4.3 se aprecia gráficamente la sólución de este problema.

Solución en un Transputer. Se especifican los procesos que son simultáneamente ejecutados en un sólo procesador.

Enseguida se muestra la especificación del proceso *Generador.números.naturales*:

```
PROC Generador.números.naturales( CHAN OF INT canal.num.generado )
  ... Inclusión del archivo con la declaración externa del número
      máximo de primos (num.max.primos) a obtener (definido por el programador)
  ... Declaración de la variable local i
  SEQ
    i := 2
```

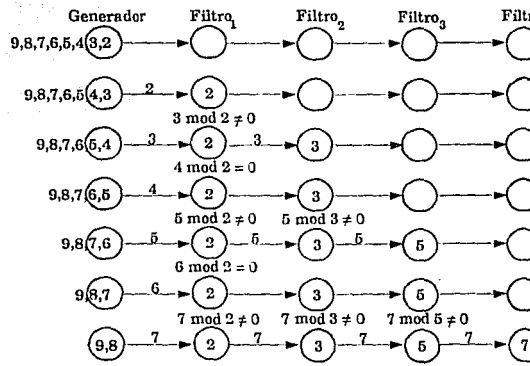


Figura 4.3: Representación del Problema de los Números Primos

```
WHILE TRUE
  IF
    i ≤ num.max.primos
      SEQ
        -- Se envía i por canal.num.generado
        canal.num.generado ! i
        i := i + 1
    TRUE
      SKIP
```

Este proceso genera números naturales desde 2 hasta *num.max.primos*, los cuales son enviados, uno por uno, a través de *canal.num.generado*.

Enseguida se muestra la especificación del proceso *"función"*:

```
PROC función( CHAN OF ANY canal.entrada, canal.salida )
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    -- Se recibe por canal.entrada un número
    canal.entrada ? es.primo
```

```
WHILE TRUE
  SEQ
    -- Se recibe por canal.entrada el siguiente número
    canal.entrada ? supuesto.primo
    -- Se verifica si supuesto.primo es en realidad
       un número primo
    IF
      ... supuesto.primo no es múltiplo de es.primo
        -- Se envía supuesto.primo por canal.salida
        canal.salida ! supuesto.primo
      TRUE
        SKIP
```

Este proceso recibe una secuencia de números, uno por uno, por *canal.entrada*. La primera vez que ingresa un número, éste es considerado como un número primo. Para cada número subsecuente se verifica la posibilidad de que sea un número primo, si existe la posibilidad, se envía este número por *canal.salida*, sino, se descarta.

Enseguida se muestra la especificación del proceso principal:

```
PROC Generador.de.numeros.primos()
  ... Inclusión del proceso Generador.números.naturales
  ... Inclusión del Mecanismo Filtro
  ... Declaración de canales
  --Ejecuta en paralelo:
  PAR
    Generador.números.naturales( canal.filtro[0] )
    -- Se especifica la ejecución simultánea de los Filtros
    PAR i = 0 FOR num.max.primos
      Filtro( canal.filtro[i], canal.filtro[i+1] )
```

Este proceso especifica la ejecución simultánea de: *Generador.números.naturales*, junto con un número de *Filtros* determinado por *num.max.primos*.

Solución en una red de Transputers. Se cuenta con una red de dos procesadores conectados en línea. Se realiza el mapeo sobre esta red, colocando el proceso *Generador.números.naturales* en un procesador y todos los *Filtros* en el otro. Enseguida se asocian los canales lógicos con las ligas físicas. Por último, se codifica un archivo de descripción con esta configuración.

Enseguida se muestra la especificación del archivo de la descripción de la configuración:

```
... Inclusión del proceso Generador.números.naturales
... Inclusión del Mecanismo Filtro
PLACED PAR
 - PROCESSOR 0 T800
     PLACE canal.filtro[0] AT link1.out :
     Generador.números.naturales( canal.filtro[0] )
   PROCESSOR 1 T800
     PLACE canal.filtro[0] AT link3.in :
     PAR i = 0 FOR num.max.primos
       Filtro( canal.filtro[i], canal.filtro[i+1] )
```

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Generador.números.naturales	0
$Filtro_1, \ldots, Filtro_{num.max.primos}$	1

	Fuente		Destino	
Canal	Transputer	Liga	Transputer	Liga
canal.filtro[0]	0	1	1	3

En la figura 4.4 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

4.3.2 Mecanismo Puente

4.3.2.1 Análisis y Especificaciones

Dos procesos, que se encuentran mapeados en procesadores distintos, desean comunicarse. Sin embargo, estos procesadores no están conectados directamente. Se requiere entonces de un mecanismo que permita la retransmisión de mensajes entre procesadores intermedios.

En la figura 4.5 se aprecian gráficamente estos requerimientos:

El proceso P en el $procesador_1$ envía un mensaje al proceso Q en el $procesador_3$; sin embargo, estos procesadores no están conectados directamente por una liga física,

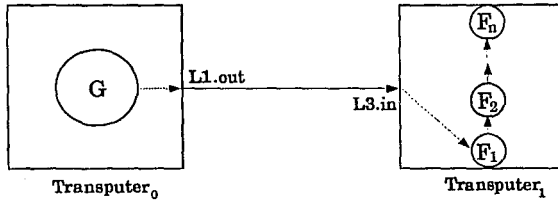


Figura 4.4: Mapeo Problema Generador de Números Primos

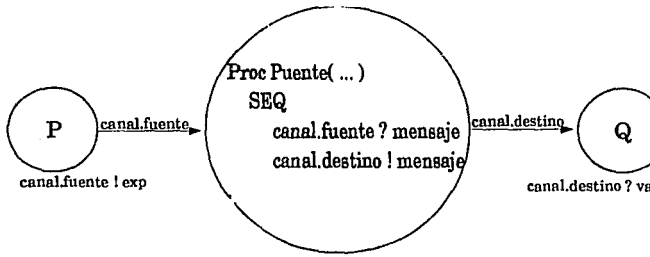


Figura 4.5: Requerimientos del Mecanismo Puente

sino que se interconectan a través del *procesador*$_2$. En este último debe existir un mecanismo que reciba el mensaje enviado por el proceso *P* para después retransmitirlo al proceso *Q*.

Bajo esta idea se propone:

> Construir un mecanismo que permita establecer una alternativa de comunicación entre dos procesos, mapeados en procesadores distintos, los cuales no poseen una conexión directa. Este mecanismo constituye un canal unidireccional y asíncrono, esto es, un *puente*.

En realidad, el mecanismo *Puente* constituye un caso particular del *Filtro* con x = F(x), en donde el cuerpo del proceso *"función"* se compone solamente de una operación de recepción, la cual se encarga de recibir un mensaje; y de una operación

de envío, la cual se encarga de reenviar el mensaje sin ninguna modificación. Se ha deseado ilustrar la creación de un nuevo mecanismo a partir de otro más general con el fin de mostrar las posibilidades de desarrollo y aplicación que se pueden lograr a partir de los mecanismos ya creados. De esta manera, el *Puente* se tratará como un nuevo mecanismo.

4.3.2.2 Diseño Conceptual

El *Puente* es accesado por dos canales: uno de entrada y otro de salida, implantados a través de *canal.fuente y canal.destino*, respectivamente. Los mensajes son recibidos por *canal.fuente* y reenviados inmediatamente por *canal.destino*.

El tipo de dato de la *variable local* que alojará el mensaje que se reciba deberá ser previamente declarado por el programador.

Se debe considerar la condición de sincronización siguiente: tanto el *Puente* como el proceso Q no podrán recibir ningún mensaje si el proceso P aún no lo ha enviado.

En la figura 4.6 se aprecia gráficamente el comportamiento del mecanismo *Puente*.

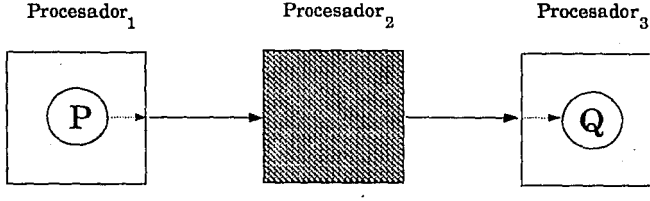


Figura 4.6: **Comportamiento del Mecanismo Puente**

4.3.2.3 Codificación e Implantación

El encabezado del proceso que implanta el mecanismo *Puente* incluye los parámetros correspondientes a los canales *canal.fuente* y *canal.destino*, cuyos protocolos son *anárquicos*. Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo, el programador podrá accesar el archivo correspondiente a la declaración del tipo de dato de la *variable local*.

Enseguida se muestra la especificación del proceso *Puente*:

```
PROC Puente( CHAN OF ANY canal.fuente, canal.destino )
  ... Inclusión de la declaración externa del tipo de dato
      de la variable local (definida por el programador)
  SEQ
    -- Se recibe el mensaje por canal.fuente
    canal.fuente ? mensaje
    -- Se reenvía el mensaje por canal.destino
    canal.destino ! mensaje
:
```

4.3.2.4 El Problema de la Comunicación Remota

Especificación. Un proceso *Emisor* requiere transferir sus mensajes a otro proceso *Receptor*. Ambos procesos se encuentran mapeados en procesadores distintos. El camino más corto entre los procesos involucra la transmisión de mensajes a través de otros tres procesadores. En la figura 4.7 se aprecia gráficamente la especificación de este problema.

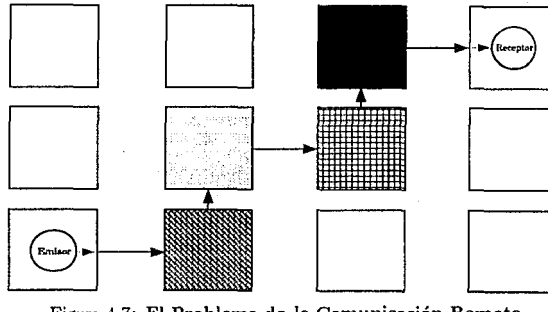


Figura 4.7: **El Problema de la Comunicación Remota**

El objetivo es encontrar un mecanismo satisfactorio para la transferencia de los mensajes entre el *Emisor* y el *Receptor*. En la figura 4.8 se aprecia gráficamente la solución del problema de la *Comunicación Remota*, incorporando el nuevo mecanismo *Puente*. El caso de solución en un Transputer no es incluido, ya que la aplicación más

importante del *Puente*, se establece cuando existe la necesidad de comunicar procesos mapeados en procesadores distintos, los cuales no poseen una conexión directa.

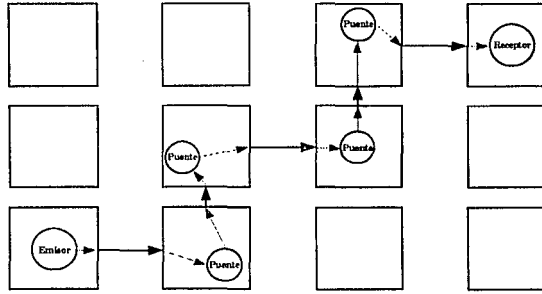


Figura 4.8: **Representación del Problema de Comunicación Remota**

Solución en una red de Transputers. Se cuenta con una red de cinco procesadores conectados en línea. Se realiza el mapeo sobre esta red, colocando el proceso *Emisor* en el primer procesador y el proceso *Receptor* en el último, así como los tres mecanismos *Puente* en los procesadores intermedios. Enseguida se asocian los canales lógicos con las ligas físicas. Por último, se escribe un archivo de descripción con esta configuración.

Enseguida se muestra un ejemplo de especificación del proceso *Emisor*:

```
PROC Emisor( CHAN OF ANY canal.salida )
  ... Declaración de variables locales y canales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      -- Se envía un mensaje por canal1
      canal.salida ! mensaje
```

Este proceso envía mensajes, uno por uno, a través de *canal.salida*.

Enseguida se muestra un ejemplo de especificación del proceso *Receptor*:

```
PROC Receptor( CHAN OF ANY canal.entrada )
  ... Declaración de variables locales y canales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      -- Se recibe un mensaje por canal4
      canal.entrada ? mensaje
```

Este proceso recibe mensajes, uno por uno, a través de *canal.entrada*.

Enseguida se muestra la especificación del archivo de la descripción de la configuración:

```
... Inclusión de los procesos: Emisor y Receptor
... Inclusión del Mecanismo Puente
... Declaración de canales
PLACED PAR
  PROCESSOR 2 T800
    canal[1] AT link1.out :
    Emisor( canal[1] )
  PROCESSOR 5 T800
    PLACE canal[1] AT link3.in :
    PLACE canal[2] AT link2.out :
    Puente( canal[1], canal[2] )
  PROCESSOR 4 T800
    PLACE canal[2] AT link4.in :
    PLACE canal[3] AT link1.out :
    Puente( canal[2], canal[3] )
  PROCESSOR 7 T800
    PLACE canal[3] AT link3.in :
    PLACE canal[4] AT link2.out :
    Puente( canal[3], canal[4] )
  PROCESSOR 6 T800
    PLACE canal[4] AT link4.in :
    PLACE canal[5] AT link1.out :
    Puente( canal[4], canal[5] )
  PROCESSOR 9 T800
    PLACE canal[5] AT link3.in :
    Receptor( canal[5] )
```

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Emisor	2
$Puente_1$	5
$Puente_2$	4
$Puente_3$	7
$Puente_4$	6
Receptor	9

Canal	Fuente		Destino	
	Transputer	Liga	Transputer	Liga
canal[1]	0	1	5	3
canal[2]	5	2	4	4
canal[3]	4	1	7	3
canal[4]	7	2	6	4
canal[5]	6	1	9	3

En la figura 4.9 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

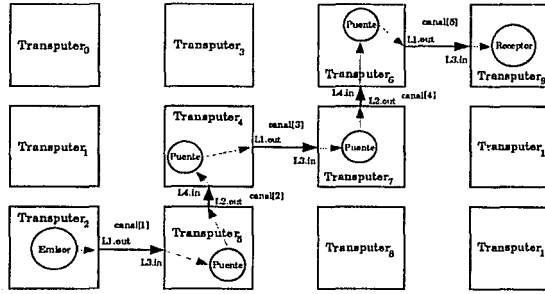


Figura 4.9: Mapeo Problema Comunicación Remota

4.3.3 Mecanismo "Buffer"

4.3.3.1 Análisis y Especificaciones

Existen mecanismos que poseen la capacidad de almacenar temporalmente los mensajes, así como de controlar el orden de su aceptación, estos son aceptados comúnmente en orden *FIFO*[8].

Tomese el ejemplo de una aplicación en donde existe un proceso comportándose como un *productor*, el cual requiere comunicar sus elementos a otro proceso *consumidor*. En este caso, el productor deposita sus elementos en una cola y el consumidor los retira de ella. De manera general, el par de llamadas que realizarían el *productor* y el *consumidor* podrían ser, respectivamente:

Depositar(cola, expresión)

Se especifica *depositar* en la *cola* un valor o elemento calculado por *expresión*.

Retirar(cola, variable)

Se especifica *retirar* de la *cola* un elemento y asignarlo en *variable*.

En la figura 4.10 se aprecia gráficamente el comportamiento del mecanismo para bufferizar mensajes antes descrito.

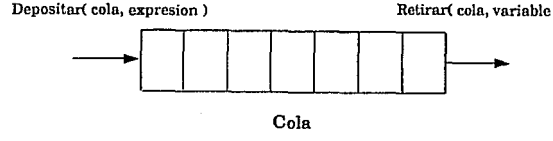


Figura 4.10: **Mecanismo para Bufferizar Mensajes**

Nótese que la estructura *cola* en realidad especifica un *canal asíncrono* que permite conectar al *productor* y al *consumidor*, estableciendo así un canal de comunicación con capacidad de almacenamiento temporal. Bajo esta idea se propone:

> Construir un mecanismo que permita establecer una alternativa de comunicación punto a punto, unidireccional y con capacidad de almacenamiento temporal, esto es, *un canal con protocolo asíncrono* o *"buffer"*.

[8]Acrónimo de *"Fist In First Out"* o Primeras Entradas Primeras Salidas

4.3.3.2 Diseño Conceptual

El cuerpo del proceso que implanta el mecanismo *"Buffer"* está constituido por una estructura de datos *cola circular acotada*, la cual está representa como un vector (con elementos del tipo de dato apropiado) con dos apuntadores. Uno de ellos, indicando la próxima localidad vacía en el vector; y el otro, apuntando al próximo objeto que será tomado del vector. Ver figura 4.11. En cada ocasión que son aplicadas las operaciones de *depósito* y *retiro* de elementos en el vector, los apuntadores son actualizados a través de operaciones aritméticas de módulo. Un *contador* lleva el control del número de elementos contenidos en el vector.

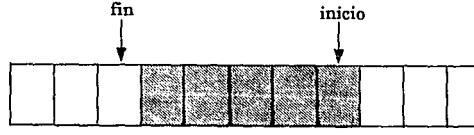


Figura 4.11: **Estructura de Datos Cola Circular Acotada**

El *"Buffer"* es accesado por dos canales: uno de entrada y otro de salida, conectando en línea a los procesos involucrados en la comunicación. Estos canales son implantados a través de *canal.izquierdo* y *canal.derecho*, respectivamente. Los elementos son recibidos por *canal.izquierdo*, enseguida son almacenados en la cola y cuando estos sean requeridos son enviados por *canal.derecho*.

La cola tiene una capacidad finita de almacenamiento que es controlada por una constante, la cual debe ser definida a priori por el programador. Asimismo, tanto el tipo de dato de la estructura *cola circular* como el de la *variable local* que alojará al elemento que se recibe deberán ser previamente declarados por el programador.

Se deben considerar dos condiciones de sincronización: no debe ser posible retirar elementos cuando la cola esté vacía (contador = 0) o insertarlos cuando esté llena (contador = num.max.elem). Asimismo, se considera que tanto las peticiones de inserción como las de retiro se dan simultánea y no determinísticamente. Esta situación de sincronización es implantada por medio del *proceso constructor alternativo* ALT que posee OCCAM.

En la figura 4.12 se aprecia gráficamente el comportamiento del mecanismo *"Buffer"*.

4.3.3.3 Codificación e Implantación

El encabezado del proceso que implanta el mecanismo *"Buffer"* incluye los parámetros correspondientes a los canales *canal.izquierdo* y *canal.derecho*, cuyos protocolos son

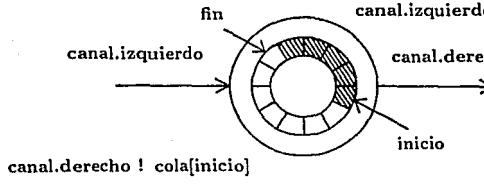


Figura 4.12: Comportamiento del Mecanismo "Buffer"

anárquicos. Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo se podrán accesar los archivos correspondientes: a la declaración del *número máximo de elementos* de la cola circular; y al de la declaración del tipo de dato tanto de la estructura de *cola circular* como de la *variable local.*

Enseguida se muestra la especificación del proceso "Buffer":

```
PROC Buffer( CHAN OF ANY canal.izquierdo, canal.derecho )
  ... Inclusión de la declaración externa del número
      máximo de elementos (num.max.elem) de la cola circular,
      así como de la declaración externa del tipo de
      dato tanto de la estructura cola circular como de la
      variable local (definidos por el programador)
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      ALT
        -- Si hay lugar en la cola y además se recibe
           un elemento por canal.izquierdo
        (contador < num.max.elem) & canal.izquierdo ? elemento
          SEQ
            -- Se inserta el elemento en la cola
            cola.circular[fin] := elemento
            ... Se actualizan apuntadores
        -- Si la cola no está vacía
        (contador > 0)
          SEQ
            -- Se envía el siguiente elemento por canal.derecho
            canal.derecho ! cola.circular[inicio]
            ... Se actualizan apuntadores
:
```

4.3.3.4 El Problema del Productor-Consumidor

Especificación. Existen dos procesos, el primero llamado *Productor*, el cual genera elementos y los transfiere al segundo llamado *Consumidor*, el cual recibe los elementos y los consume. En la figura 4.13 se aprecia gráficamente la especificación de este problema.

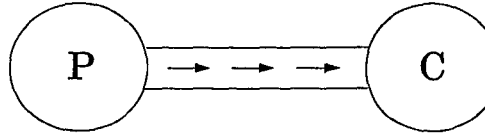


Figura 4.13: **El Problema del Productor-Consumidor**

El objetivo es encontrar un mecanismo satisfactorio para la transferencia de los elementos entre el *Productor* y el *Consumidor*. En la figura 4.14 se aprecia gráficamente la solución del problema *Productor-Consumidor*, incorporando el nuevo mecanismo *"Buffer"*.

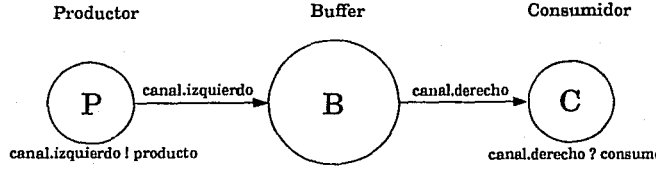


Figura 4.14: **Representación del Problema Productor-Consumidor**

Solución en un Transputer. Se especifican los procesos que son simultáneamente ejecutados en un sólo procesador.

Enseguida se muestra la especificación del proceso *Productor*:

```
PROC Productor( CHAN OF ANY canal.izquierdo )
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      SEQ
        ... Se produce elemento (producción)
        -- Se envía elemento por canal.izquierdo
        canal.izquierdo ! producto
:
```

Este proceso produce elementos, los cuales son enviados, uno por uno, a través de *canal.izquierdo*

Enseguida se muestra la especificación del proceso *Consumidor*:

```
PROC Consumidor( CHAN OF ANY canal.derecho )
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      SEQ
        -- Se recibe elemento por canal.derecho (consumo)
        canal.derecho ? consumo
        ... Se utiliza consumo
:
```

Este proceso consume elementos, los cuales son recibidos, uno por uno, a través de *canal.derecho*

Enseguida se muestra la especificación del proceso principal:

```
PROC Productor.Consumidor()
  ... Inclusión del Mecanismo Buffer
  ... Declaración de canales
  -- Ejecuta en paralelo:
  PAR
    Productor( canal.izquierdo )
    Buffer( canal.izquierdo, canal.derecho )
    Consumidor( canal.derecho )
:
```

Este proceso especifica la ejecución simultánea de los procesos: *Productor*, *"Buffer"* y *Consumidor*.

Solución en una red de Transputers. Se cuenta con una red de tres procesadores conectados en línea. Se realiza el mapeo sobre esta red, colocando cada uno de los procesos en un procesador y asociando los canales lógicos con las ligas físicas. Por último, se codifica un archivo de descripción con esta configuración.

Enseguida se muestra la especificación del archivo de la descripción de la configuración:

```
... Inclusión de los procesos: Productor y Consumidor
... Inclusión del Mecanismo Buffer
... Declaración de canales
PLACED PAR
  PROCESSOR 0 T800
    PLACE canal.izquierdo AT link1.out :
    Productor( canal.izquierdo )
  PROCESSOR 1 T800
    PLACE canal.izquierdo AT link3.in :
    PLACE canal.derecho AT link1.out :
    Buffer( canal.izquierdo, canal.derecho )
  PROCESSOR 2 T800
    PLACE canal.derecho AT link3.in :
    Consumidor( canal.derecho )
```

Nótese que los procesos: *Productor*, *Consumidor* y *"Buffer"* no sufrieron modificación alguna.

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Productor	0
Buffer	1
Consumidor	2

Canal	Fuente		Destino	
	Transputer	Liga	Transputer	Liga
canal.izquierdo	0	1	1	3
canal.derecho	1	1	2	3

En la figura 4.15 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

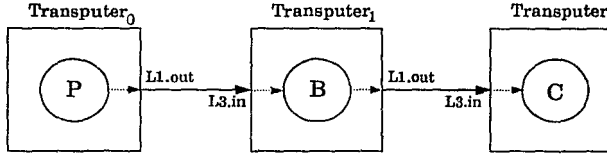


Figura 4.15: **Mapeo Problema Productor-Consumidor**

4.3.4 Mecanismo Multiplexor

4.3.4.1 Análisis y Especificaciones

Un canal lógico conecta punto a punto a dos procesos; asimismo, una liga física conecta punto a punto a dos procesadores. Sin embargo, considerando que dos Transputers están enlazados en forma directa, cuando existe la necesidad de comunicar un proceso, mapeado en uno de los procesadores, con un conjunto de procesos, mapeados en otro de los procesadores, o viceversa, se requieren entonces mecanismos que permitan multiplexar la liga física, es decir, que múltiples canales lógicos tengan la capacidad de compartir una misma liga.

En la figura 4.16 se aprecian gráficamente estos requerimientos:

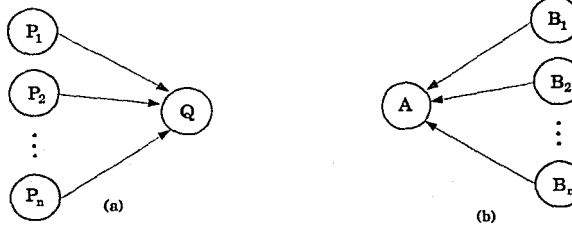


Figura 4.16: **Requerimientos de los Mecanismos Multiplexores**

En (a) los procesos $P_1, P_2, \ldots, P_n$ envían mensajes al proceso Q. Este envío de mensajes se realiza en forma arbitraria. La recepción de mensajes en el proceso Q se realiza en forma no determinística. No existe reordenamiento de mensajes, ni estos se pierden.

En (b) los procesos $A_1, A_2, \ldots, A_n$ reciben mensajes del proceso B. Cuando alguno de los procesos A_i se encuentre listo para recibir un mensaje, realizará la petición de éste, enviando una señal de petición. El número de procesos A_i listos para recibir un mensaje es arbitrario. Cuando el proceso B recibe una señal de petición, éste envía el mensaje al proceso P_i correspondiente. La recepción de señales de petición en el proceso B se realiza en forma no determinística. No existe reordenamiento de mensajes, ni estos se pierden.

Bajo esta idea se propone:

> Construir un mecanismo que permita establecer una alternativa de comunicación de varios procesos hacia uno sólo y otro mecanismo que permita la comunicación de un proceso hacia varios de ellos. Cada mecanismo constituye un canal unidireccional, asíncrono y con características de comunicación *"muchos a uno"* y *"uno a muchos"*, esto es, un *multiplexor*.

4.3.4.2 Diseño Conceptual

Multiplexor para comunicación "muchos a uno". El cuerpo del proceso que implanta este mecanismo está constituido por un proceso constructor alternativo, el cual se encarga de controlar el envío y recepción de mensajes, entre los procesos emisores y el proceso receptor, respectivamente.

El *Multiplexor* es accesado por $n + 1$ canales: n de entrada y uno de salida, implantados a través del arreglo de n canales *canal.izquierdo[1],...,canal.izquierdo[n]*; y *canal.derecho*, respectivamente. Los mensajes de los procesos $P_1, P_2, \ldots, P_n$ son recibidos, uno a la vez, por *canal.izquierdo[1],...,canal.izquierdo[n]*, respectivamente. El mensaje recibido (correspondiente al proceso P_i) es enviado por *canal.derecho* hacia el proceso B.

El tipo de dato de la *variable local*, para almacenar temporalmente el mensaje que recibe el *Multiplexor*, deberá ser previamente declarado por el programador.

Se deben considerar las condiciones de sincronización siguientes: no es posible enviar un mensaje al proceso B, si ninguno de los procesos $P_1, P_2, \ldots, P_n$ ha enviado por lo menos un mensaje al *Multiplexor*. Asimismo, se considera que cualquiera de los procesos P_i puede enviar su mensaje arbitrariamente.

En la figura 4.17 se aprecia gráficamente el comportamiento del mecanismo *Multiplexor "muchos a uno"*.

Multiplexor para comunicación "uno a muchos". El cuerpo del proceso que implanta este mecanismo esta constituido por un proceso constructor alternativo, el

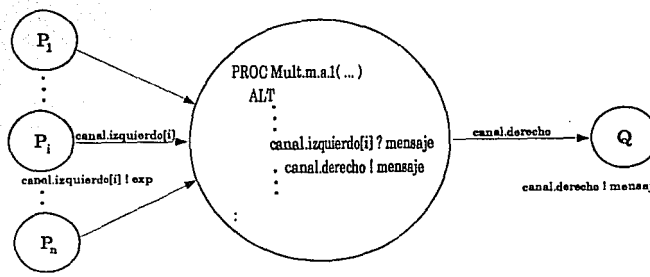


Figura 4.17: Comportamiento del Mecanismo Multiplexor "Muchos a Uno"

cual se encarga de controlar tanto el envío de mensajes como la petición y recepción de estos, entre el proceso emisor y los procesos receptores, respectivamente.

El *Multiplexor* es accesado por $n * 2 + 1$ canales: uno de entrada, n de salida y n de entrada de señales de petición hacia el *Multiplexor*, implantados a través de *canal.izquierdo*; el arreglo de n canales *canal.derecho[1],...,canal.derecho[n]*; y el arreglo de n canales *canal.petición[1],...,canal.petición[n]*, respectivamente.

Los mensajes son recibidos del proceso B, uno cada vez, por *canal.izquierdo*. Los procesos $A_1, A_2, \ldots, A_n$ envían una señal de petición al *multiplexor* por *canal.petición[1], ...,canal.petición[n]*, respectivamente, indicandole que están listos para recibir, cualquiera de ellos, el mensaje. Los procesos $A_1, A_2, \ldots, A_n$ reciben el mensaje, uno a la vez, por *canal.derecho[1],...,canal.derecho[n]*, respectivamente.

El tipo de dato de la *variable local* para almacenar temporalmente el mensaje que recibe el *multiplexor* deberá ser previamente declarado por el programador.

Se deben considerar las condiciones de sincronización siguientes: no es posible que los procesos $A_1, A_2, \ldots, A_n$ reciban algún mensaje, si estos no envían al proceso *multiplexor* la petición del mismo, ni tampoco es posible que reciban algún mensaje, si el proceso B no lo ha enviado al *multiplexor*. Asimismo, se considera que cualquiera de los procesos A_i puede enviar la señal de petición de mensaje arbitrariamente.

En la figura 4.18 se aprecia gráficamente el comportamiento del mecanismo *Multiplexor "uno a muchos"*.

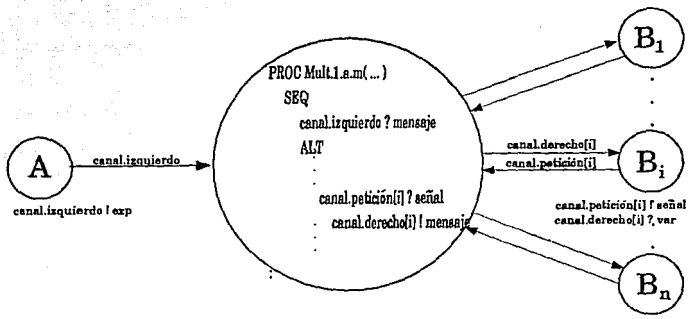


Figura 4.18: **Comportamiento del Mecanismo Multiplexor "Uno a Muchos"**

4.3.4.3 Codificación e Implantación

Multiplexor para comunicación "muchos a uno". El encabezado del proceso que implanta el mecanismo *Multiplexor "muchos a uno"* incluye los parámetros correspondientes al arreglo de canales *canal.izquierdo* y al *canal.derecho*, cuyos protocolos son *anárquicos*. Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo, es posible accesar el archivo correspondiente a la declaración del tipo de dato de la *variable local.*

Enseguida se muestra la especificación del proceso *Multiplexor "muchos a uno"*:

```
PROC Muxltiplexor.muchos.a.uno( []CHAN OF ANY canal.izquierdo,
                                CHAN OF ANY canal.derecho )
  ... Inclusión de la declaración externa del tipo de dato
      de la variable local (definida por el programador)
  WHILE TRUE
    -- Si por canal.izquierdo[i] (donde i = 1,..,número máximo
       de canales izquierdos) se recibe un mensaje, entonces
         Se envía el mensaje por canal.derecho
    ALT i = 1 FOR num.max.can.izq
      canal.izquierdo[i] ? mensaje
        canal.derecho ! mensaje
:
```

Multiplexor para comunicación "uno a muchos". El encabezado del proceso que implanta el mecanismo *Multiplexor "uno a muchos"* incluye los parámetros correspondientes al *canal.izquierdo*; al arreglo de canales *canal.derecho* (con protocolo *anárquico*); y al arreglo de canales *canal.petición* (con protocolo *variante*). Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo, es posible accesar los archivos correspondientes: a la declaración del protocolo del arreglo de canales *canal.petición*; y al de la declaración del tipo de dato de la *variable local*.

Enseguida se muestra la especificación del proceso *Multiplexor "uno a muchos"*:

```
... Inclusión de la declaración externa del protocolo del canal.petición
PROC Multiplexor.uno.a.muchos( CHAN OF ANY canal.izquierdo,
                               []CHAN OF ANY canal.derecho,
                               []CHAN OF PETICION canal.petición )
  ... Inclusión de la declaración externa del tipo de dato de la variable
      local (definida por el programador)
  WHILE TRUE
    -- Si por canal.izquierdo se recibe un mensaje, entonces
       Si por canal.petición[i] (donde i = 1,..,número máximo de
       canales derechos) se recibe señal de petición, entonces
         Se envía el mensaje por canal.derecho[i]
    ALT
      canal.izquierdo ? mensaje
        ALT i = 1 FOR num.max.can.der
          canal.petición[i] ? CASE señal
            canal.derecho ! mensaje[i]
:
```

4.3.4.4 El Problema de los Productores-Consumidores

Especificación. Existen procesos *Productores* que generan elementos y los transfieren a procesos *Consumidores*, los cuales reciben los elementos y los consumen. En la figura 4.19 se aprecia gráficamente la especificación de este problema.

El objetivo es encontrar un mecanismo satisfactorio para la transferencia de los elementos entre los *Productores* y los *Consumidores*. En la figura 4.20 se aprecia gráficamente la solución del problema *Productores-Consumidores*, incorporando los nuevos mecanismos *Multiplexor "muchos a uno"* y *Multiplexor "uno a muchos"*, así como el mecanismo *"Buffer"* ya creado.

Solución en un Transputer. Se especifican los procesos que son simultáneamente ejecutados en un sólo procesador. Los procesos *Productor*, *"Buffer"* y *Consumidor* son los mismos que se especificaron en la sección anterior.

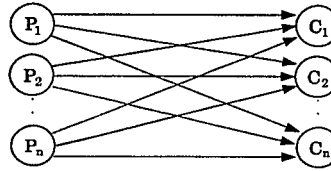


Figura 4.19: **El Problema de los Productores-Consumidores**

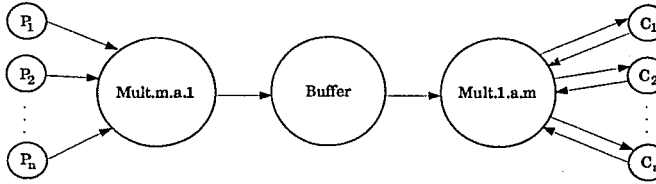


Figura 4.20: **Representación del Problema Productores-Consumidores**

Enseguida se muestra la especificación del proceso principal:

```
PROC Productores.Consumidores()
   ... Inclusión del archivo con la declaración externa del número
       máximo de productores (num.max.prod) y el número máximo de
       consumidores (num.max.cons) (definido por el programador)
   ... Inclusión de los Mecanismos Multiplexor Muchos a Uno
       y Multiplexor Uno a Muchos
   ... Declaración de canales
   - - Ejecuta en paralelo:
   PAR
      PAR i = 1 FOR num.max.prod
         Productor( canal.productor[i] )
      Multiplexor.muchos.a.uno( canal.productor, canal.izquierdo )
      Buffer( canal.izquierdo, canal.derecho )
      Multiplexor.uno.a.muchos( canal.derecho, canal.consumidor, canal.petición )
      PAR i = 1 FOR num.max.cons
         Consumidor( canal.consumidor[i], canal.petición[i] )
:
```

Este proceso especifica la ejecución simultánea de los procesos: *Productor*, *Multiplexor.muchos.a.uno*, *"Buffer"*, *Multiplexor.uno.a.muchos* y *Consumidor*.

Solución en una red de Transputers. Se cuenta con una red de tres procesadores conectados en línea. Se realiza el mapeo sobre esta red, colocando grupos de procesos en cada uno de los procesadores. En el primer procesador se mapean los procesos *Productores* junto con el *Multiplexor.muchos.a.uno*. En el segundo se mapea el proceso *"Buffer"* y en el tercero se mapea el proceso *Multiplexor.uno.a.muchos* junto con los *Consumidores*. Enseguida se asocian los canales lógicos con las ligas físicas. Por último, se codifica el archivo de descripción con esta configuración.

Para que un conjunto de procesos puedan ser ejecutados en un Transputer, estos deben agruparse en un sólo proceso, estableciendo adecuadamente los canales que serán utilizados para la comunicación hacia el exterior, es decir, con otros Transputers.

De esta manera, se debe realizar la codificación que agrupe a los procesos de la manera anteriormente especificada.

Enseguida se muestra la especificación del proceso *Grupo.Transputer0*:

```
PROC Grupo.Transputer0( CHAN OF ANY canal.izquierdo )
  -- Ejecuta en paralelo:
  PAR
    PAR i = 0 FOR num.max.prod
      Productor( canal.productor[i] )
      Multiplexor.muchos.a.uno( canal.productor, canal.izquierdo )
:
```

Enseguida se muestra la especificación del proceso *Grupo.Transputer2*:

```
PROC Grupo.Transputer2( CHAN OF ANY canal.derecho )
  -- Ejecuta en paralelo:
  PAR
    Multiplexor.uno.a.muchos( canal.derecho, canal.consumidor, canal.petición )
    PAR i = 0 FOR num.max.cons
      Consumidor( canal.consumidor[i], canal.petición[i] )
:
```

Enseguida se muestra la especificación del archivo de la descripción de la configuración:

```
... Inclusión de los procesos: Grupo.Transputer0 y Grupo.Transputer2
... Inclusión del Mecanismo Buffer
... Inclusión de los Mecanismos: Multiplexor Muchos a Uno
    y Multiplexor Uno a Muchos
... Declaración de canales
PLACED PAR
  PROCESSOR 0 T800
    canal.izquierdo AT link1.out :
    Grupo.Transputer0( canal.izquierdo )
  PROCESSOR 1 T800
    PLACE canal.izquierdo AT link3.in :
    PLACE canal.derecho AT link1.out :
    Buffer( canal.izquierdo, canal.derecho )
  PROCESSOR 2 T800
    PLACE canal.derecho AT link3.in :
    Grupo.Transputer2( canal.derecho )
```

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Grupo.Transputer0	0
Buffer	1
Grupo.Transputer0	2

	Fuente		Destino	
Canal	Transputer	Liga	Transputer	Liga
canal.izquierdo	0	1	1	3
canal.derecho	1	1	2	3

En la figura 4.21 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

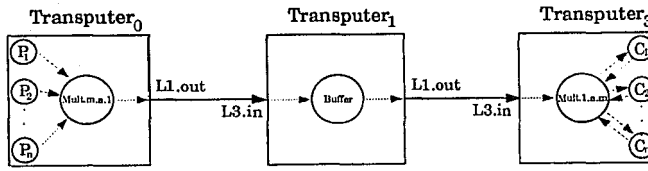


Figura 4.21: **Mapeo Problema Productores-Consumidores**

4.3.5 Mecanismo Difusor

4.3.5.1 Análisis y Especificaciones

Existen mecanismos que permiten la difusión de mensajes en forma simultánea de un proceso emisor a un conjunto de procesos conectados a él. A través de un canal lógico se pueden enviar mensajes, pero sólo hacia un proceso. De esta manera, se requiere entonces de un mecanismo que permita realizar la difusión de mensajes en forma simultánea a un conjunto de procesos. Se debe tener la posibilidad también de que todos los procesos receptores puedan estar mapeados en otros procesadores conectados directamente al procesador en el que se encuentra el proceso emisor.

En la figura 4.22 se aprecian gráficamente estos requerimientos:

El proceso P envía simultáneamente un mensaje a los procesos $Q_1, Q_2, \ldots, Q_n$.

Bajo esta idea se propone:

> Construir un mecanismo que permita establecer una alternativa de comunicación de un proceso (P) hacia varios de ellos (Q_i con i = 1,...,n), en donde se realice la difusión de un mensaje en forma simultánea de P a $Q_1, \ldots, Q_n$. Este mecanismo constituye un canal unidireccional, asíncrono y con características de comunicación *"uno a muchos"*, esto es, un *difusor*.

La diferencia entre el mecanismo *Difusor* y el *Multiplexor "uno a muchos"* (especificado en la sección 4.4.4.1), radica en el hecho de que en este segundo, cualquier mensaje que es enviado hacia el *Multiplexor*, sólo es recibido por uno de los procesos conectados a él, esto es, por aquel que haya realizado la petición del mensaje. En el primer mecanismo en cambio, cualquier mensaje que es enviado hacia el *Difusor*, es inmediatamente reenviado en forma simultánea al conjunto de procesos conectados a él.

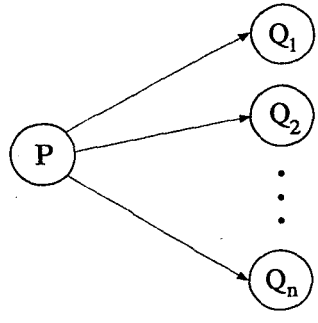


Figura 4.22: **Requerimientos del Mecanismo Difusor**

4.3.5.2 Diseño Conceptual

El cuerpo del proceso que implanta el mecanismo *Difusor* está constituido por una operación de recepción y la ejecución simultánea de un conjunto de operaciones de envío, las cuales se encargan de controlar la recepción del mensaje, así como de la difusión de éste a todos los procesos conectados, respectivamente.

El *Difusor* es accesado por $n + 1$ canales: uno de entrada y n de salida, implantados a través de *canal.izquierdo*; y un arreglo de n canales *canal.derecho[1],...*, *canal.derecho[n]*, respectivamente. Los mensajes son recibidos del proceso P, uno la vez, por *canal.izquierdo*. Inmediatamente después, cada uno de ellos es reenviado simultáneamente por *canal.derecho[1],...,canal.derecho[n]* hacia los procesos $Q_1, Q_2, \ldots, Q_n$.

El tipo de dato de la *variable local* para almacenar temporalmente el mensaje que recibe el *Difusor* deberá ser previamente declarado por el programador.

Se debe considerar la condición de sincronización siguiente: Tanto el *Difusor* como los procesos $Q_1, Q_2, \ldots, Q_n$ no podrán recibir ningún mensaje, si el proceso P aún no lo ha enviado.

En la figura 4.23 se aprecia gráficamente el comportamiento del mecanismo *Difusor*.

4.3.5.3 Codificación e Implantación

El encabezado del proceso que implanta el mecanismo *Difusor* incluye los parámetros correspondientes al *canal.izquierdo* y al arreglo de canales *canal.derecho*, cuyos proto-

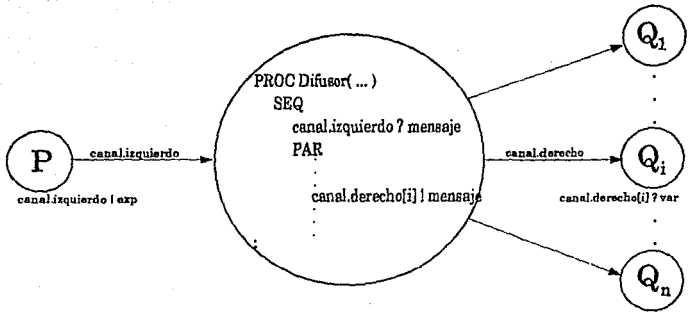


Figura 4.23: **Comportamiento del Mecanismo Difusor**

colos son *anárquicos*. Con el fin de realizar los ajustes necesarios para la utilización de este mecanismo se podrá accesar el archivo correspondiente a la declaración del tipo de dato de la *variable local*.

Enseguida se muestra la especificación del proceso *Difusor*:

```
PROC Difusor( CHAN OF ANY canal.izquierdo,
              []CHAN OF ANY canal.derecho,
  ... Inclusión de la declaración externa del tipo de dato
      de la variable local (definida por el programador)
  WHILE TRUE
    SEQ
      -- Se recibe un mensaje por canal.izquierdo
      canal.izquierdo ? mensaje
      -- Se envía simultáneamente el mensaje a los
         procesos Qi por canal.derecho[i]
         (donde i = 1,..,número de canales derechos)
      PAR i = 1 FOR num.max.can.der
        canal.derecho[i] ! mensaje
:
```

4.3.5.4 El Problema de la Difusión en Intervalos Regulares

Especificación. Un proceso *emisor* realiza la difusión de mensajes a un conjunto de procesos *receptores* conectados a él, durante intervalos regulares de tiempo. En la figura 4.24 se aprecia gráficamente la solución del problema de la *Difusión en Intervalos Regulares*, incorporando el nuevo mecanismo *Difusor*.

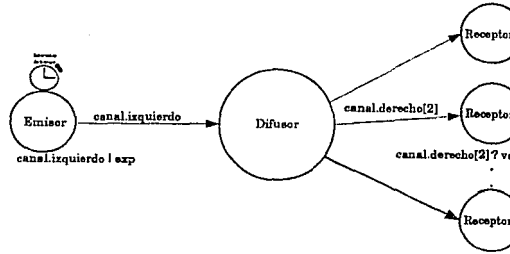


Figura 4.24: **Repres. del Problema de Difusión en Intervalos Regulares**

El objetivo es encontrar un mecanismo satisfactorio para la transferencia del mensaje por parte del proceso *emisor* a todos los procesos *receptores*.

Solución en un Transputer. Se especifican los procesos que son simultáneamente ejecutados en un sólo procesador.

Enseguida se muestra la especificación del proceso *Emisor*:

```
PROC Emisor( CHAN OF ANY canal.izquierdo )
  ... Declaración de variables locales y canales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      SEQ
        -- Se envía un mensaje por canal.izquierdo
        canal.izquierdo ! mensaje
        ... Realiza un retardo de tiempo t
:
```

Este proceso especifica el envío de mensajes, uno a la vez, por *canal.izquierdo* en intervalos regulares de tiempo definidos previamente.

Enseguida se muestra la especificación del proceso *Receptor*:

```
PROC Receptor( CHAN OF ANY canal.derecho )
  ... Declaración de variables locales
  SEQ
    ... Inicialización de variables locales
    WHILE TRUE
      SEQ
        -- Se recibe mensaje por canal.derecho
        canal.derecho ? mensaje
        ... Se utiliza mensaje
:
```

Este proceso especifica la recepción de mensajes, uno a la vez, por *canal.derecho* en intervalos regulares de tiempo definidos previamente.

Enseguida se muestra la especificación del proceso principal:

```
PROC Difusión.en.Intervalos.Regulares()
  ... Inclusión del Mecanismo Difusor
  ... Declaración de canales
  - Ejecuta en paralelo:
  PAR
    Emisor( canal.izquierdo )
    Difusor( canal.izquierdo, canal.derecho )
    -- Se recibe simultáneamente un mensaje
       por canal.derecho[i] (donde
       i = 1,...,número de canales derechos)
    PAR i = 1 FOR num.max.can.der
      Receptor( canal.derecho[i] )
:
```

Este proceso especifica la ejecución simultánea de los procesos: *Emisor*, *Difusor* y $Receptor_1, \ldots, Receptor_{num.max.can.der}$

Solución en una red de Transputers. Se cuenta con una red de procesadores en configuración tipo estrella. Se realiza el mapeo sobre esta red, colocando los procesos *Emisor* y *Difusor* en el procesador central de la estrella y cada uno de los procesos *Receptor* en los procesadores contiguos. Enseguida, se asocian los canales lógicos con las ligas físicas. Por último, se escribe un archivo de descripción con esta configuración.

Enseguida se muestra la especificación del archivo de la descripción de la configuración:

```
... Inclusión de los procesos: Emisor y Receptor
... Inclusión del Mecanismo Difusor
... Declaración de canales
PLACED PAR
  PROCESSOR 0 T800
    PLACE canal.derecho[1] AT link2.out :
    PLACE canal.derecho[2] AT link1.out :
    PLACE canal.derecho[3] AT link4.out :
    PAR
      Emisor( canal.izquierdo )
      Difusor( canal.izquierdo, canal.derecho )
  PROCESSOR 1 T800
    PLACE canal.derecho[1] AT link4.in :
    Receptor( canal.derecho[1] )
  PROCESSOR 2 T800
    PLACE canal.derecho[2] AT link3.in :
    Receptor( canal.derecho[2] )
  PROCESSOR 3 T800
    PLACE canal.derecho[3] AT link2.in :
    Receptor( canal.derecho[3] )
```

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Emisor	0
Difusor	0
$Receptor_1$	1
$Receptor_2$	2
$Receptor_3$	3

Canal	Fuente		Destino	
	Transputer	Liga	Transputer	Liga
canal.derecho[1]	0	2	1	4
canal.derecho[2]	0	1	2	3
canal.derecho[3]	0	4	3	2

En la figura 4.25 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

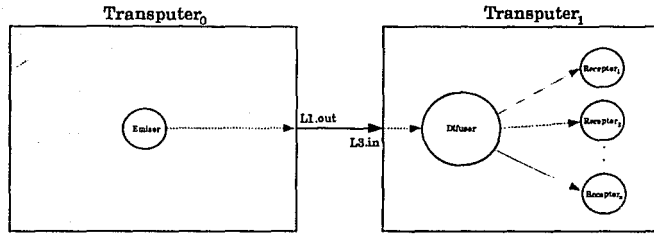


Figura 4.25: **Mapeo Problema Difusión en Intervalos Regulares**

4.4 Descripción de la Biblioteca de Nuevos Mecanismos

Los nuevos mecanismos de comunicación y sincronización desarrollados se reunieron en un archivo llamado `COMUSINC.OCC`, el cual constituye una biblioteca de software que puede ser utilizada como cualquier otra biblioteca de procesos incorporada en el compilador OCCAM. El programador tan sólo debe verificar la sintáxis y las reglas que se deben seguir para la utilización de los nuevos mecanismos (implantados como procesos), y de existir necesidad, recompilar la biblioteca.

Para recompilar la biblioteca debe ser invocada la siguiente herramienta del TDS:

ILIBR COMUSINC.OCC

Si se han seguido adecuadamente los pasos de utilización, se generará entonces un código que será almacenado en un archivo llamado `COMUSINC.LIB`, el cual constituye la Biblioteca de Nuevos Mecanismos. La forma general de un proceso utilizando esta biblioteca sería:

PROC *nombre.proceso(lista de parámetros)*
#USE "COMUSINC.LIB"
... Declaración de variables locales y canales
... Cuerpo del proceso que incluye la invocación a uno o más de los mecanismos creados
:

Es importante mencionar que los parámetros básicos que pueden ser modificados se encuentran en archivos separados, los cuales se refieren principalmente a:

- Valores de constantes.
- Protocolos de los canales.
- Tipos de datos de variables locales.
- Número de procesos

Los nombres de estos archivos se especifican en cada uno de los nuevos mecanismos incorporados en la biblioteca de software. Para realizar las modificaciones necesarias deben editarse estos archivos y se deben seguir las reglas de utilización descritas en cada uno de los mecanismos incorporados en la bilblioteca.

Para ejemplificar el uso de la *Biblioteca de Nuevos Mecanismos*, en el apéndice B se muestra la implantación, en lenguaje OCCAM, del problema de los *Productores-Consumidores*, la cual se realizó en una tarjeta MicroWay^{TM} *Quadputer* con cuatro procesadores Transputer del tipo TS00.

Conclusión

Se ha descrito y examinado una alternativa de lenguaje-arquitectura ideal para la programación y el procesamiento paralelo y distribuido. El lenguaje OCCAM, fundamentado en un formalismo de diseño bien definido, constituye tanto un modelo de especificación de sistemas como un lenguaje de programación paralelo. La arquitectura paralela Transputer constituye una plataforma de desarrollo idónea para la implantación de sistemas paralelos y distribuidos escritos en OCCAM.

Aunque OCCAM se consolida como un lenguaje sencillo, elegante y poderoso, brindando: la expresión clara y flexible del paralelismo, la comunicación y sincronización entre procesos y el control al no determinismo (elementos claves e indispensables para el desarrollo y la construcción de sistemas paralelos y distribuidos), su filosofía de diseño no deja de ser elemental y de bajo nivel, ya que posee algunas limitaciones y restricciones, las cuales no son arbitrarias, sino que se deben al hecho de que se trata de un lenguaje orientado exclusivamente al manejo de una clase muy particular de hardware: el *Transputer*.

Uno de los principales problemas que se identifica en el desarrollo de sistemas paralelos y distribuidos complejos, utilizando OCCAM y Transputers, radica en el hecho de que el concepto de comunicación y sincronización entre procesos en esta pareja lenguaje-arquitectura está basado en un sólo mecanismo básico: el *canal*, el cual si bien permite el control de la simultaneidad, así como la cooperación y la coordinación de un conjunto de procesos en ejecución paralela, constituye también un mecanismo de comunicación de bajo nivel, ya que cuando se requieren resolver problemas complejos de interacción entre procesos en una aplicación, el programador debe invertir tiempo y esfuerzo programando estos patrones de interacción a través de mecanismos más elaborados, permitiendo así dar respuesta a los requerimientos del modelo de solución de su aplicación.

En este trabajo se ha presentado una solución a esta problemática, creando y construyendo a partir del canal (como único mecanismo básico y de bajo nivel que se integra en OCCAM-Transputer) nuevos mecanismos (de más alto nivel) para la comunicación y sincronización de procesos. Cabe señalar que todos los mecanismos fueron

construidos a través de procesos OCCAM, junto con sus canales de comunicación respectivos. Para cada uno de ellos fue descrito el proceso de análisis y especificación, diseño conceptual, codificación e implantación; asimismo, se han ilustrado sus ventajas y propiedades, resolviendo algunos problemas en programación paralela y distribuida, los cuales fueron ejecutados en un Transputer y en una red de ellos.

Los mecanismos creados fueron agrupados en una *biblioteca de software*, de fácil utilización y mantenimiento, la cual representa una herramienta de apoyo para el programador de aplicaciones, puesto que no tendrá que invertir tiempo y gran esfuerzo, deteniéndose a pensar y meditar la forma en cómo programar los patrones complejos y elaborados de comunicación y sincronización entre procesos, los cuales se encuentran inherentes en muchos de los problemas en programación paralela y distribuida. Ahora tan sólo prestará su atención en especificar adecuadamente el modelo de solución del problema planteado, y tomará de la biblioteca aquellos mecanismos que requiera.

Enseguida se muestra el nivel jerárquico en donde se encuentran localizados los nuevos mecanismos construidos:

Programas de Aplicación
Biblioteca de Nuevos Mecanismos
Filtro
Puente
"Buffer"
Multiplexor "muchos a uno"
Multiplexor "uno a muchos"
Difusor
OCCAM
canal
Hardware del Transputer

Se puede apreciar que para la construcción de todos los mecanismos se tomó como base únicamente al lenguaje OCCAM; de esta manera, se han ampliado las opciones de comunicación y sincronización entre procesos a través de nuevos mecanismos puestos a disposición como una biblioteca más del lenguaje. Los resultados que se obtienen al utilizar la Biblioteca de Nuevos Mecanismos son los siguientes:

- *Versatilidad.* Cualquiera de los mecanismos puede ser fácilmente implantado en distintas aplicaciones. Simplemente es requerido el ajuste de los parámetros para su utilización, los cuales en general corresponden a: tipos de datos de variables, protocolos de canales, valores constantes y número de procesos.

- *Flexibilidad.* Cualquiera de los mecanismos puede ser modificado en su cuerpo de proceso, con el objeto de: incrementar su desempeño, acoplarse a las necesidades particulares de alguna aplicación especial, así como para la creación de un nuevo mecanismo.

- *Eficiencia.* Se considera que los nuevos mecanismos son eficientes en el sentido que logran hacerle la vida más fácil al programador de aplicaciones, permitiendo que la labor de desarrollo e implantación de mecanismos para la solución a patrones complejos de interacción entre procesos sea transparente.

- *Productividad.* El programador puede llegar a ser más productivo al crear una aplicación, ya que el tiempo que antes empleaba para el desarrollo e implantación de todos los mecanismos necesarios, ahora puede ser mejor utilizado, por ejemplo, para optimizar el desempeño de ejecución de la aplicación.

- *Reducción de costos.* Con todo lo anterior, de alguna forma, se estima que el tiempo total necesario para la creación y puesta en marcha de una aplicación será menor, obteniendo consecuentemente una reducción en los costos de desarrollo e implantación.

Sin embargo, es importante mencionar que los nuevos mecanismos poseen también algunas limitaciones, las cuales se citan enseguida:

- Cualquiera de los mecanismos sólo puede comunicar valores de acuerdo al formato que establece para ello el *protocolo simple* implantado; de esta manera, un mecanismo sólo permite transmitir un valor de cada vez. Hubiera existido la posibilidad de construir todos los mecanismos, implantando algún otro tipo de protocolo, por ejemplo, el secuencial o el variante, permitiendo así aumentar la capacidad de transmisión de mensajes sobre un sólo canal; sin embargo, no fue tomada esta decisión, ya que traería como consecuencias principalmente: un aumento considerable en el número de líneas de código de cada mecanismo y menor versatilidad para ser utilizados por distintas aplicaciones.

- Cada vez que sea actualizado alguno de los parámetros de un mecanismo, se debe recompilar toda la biblioteca. Una solución a este problema sería el tratar por separado cada mecanismo; de esta manera, cada uno de ellos tendría que ser compilado individualmente, de tal forma que las actualizaciones de los parámetros impliquen la recompilación sólo de aquellos mecanismos involucrados.

- Todos los mecanismos deben "conocer" a los procesos que estarán involucrados en la comunicación. Esta situación es consecuencia del carácter estático del lenguaje para la creación de procesos.

Enseguida se mencionan los elementos fundamentales del lenguaje que hicieron posible la creación de los nuevos mecanismos:

- Dadas las características del canal junto con sus primitivas de comunicación y sincronización, fue factible la construcción de mecanismos más elaborados.

- El proceso constructor paralelo junto con el replicador paralelo permitieron la expresión clara y consistente de la ejecución simultánea de procesos.

- El proceso constructor alternativo fue crucial en la obtención tanto de la comunicación asíncrona como de la multipunto.

- La posibilidad de inclusión de archivos, utilizados para la parametrización de los mecanismos, permitió dar completa versatilidad en la utilización de estos por distintas aplicaciones.

- El protocolo anárquico, establecido para los canales que comunican hacia al exterior a cada uno de los mecanismos, evitó la utilización de archivos extras como parámetros para definir el tipo de protocolo necesario para la transmisión.

- La posibilidad de crear arreglos de canales, así como de manipular sus elementos simultáneamente, brindaron gran versatilidad en la programación, ofreciendo consistencia y facilidad en el uso de múltiples canales.

Una solución parcial al problema de la comunicación y sincronización de conjuntos de procesos, mapeados en procesadores distintos, ha sido resuelta vía hardware en el Transputer T9000. En esta arquitectura se ha incorporado un procesador de canales virtuales, el cual permite a una liga física multiplexarse, habilitando que un número de procesos compartan la liga física de manera transparente a través de canales lógicos virtuales. Es evidente que esta solución representa gran ventaja en cuanto a la velocidad de transferencia debido a su implantación en hardware; sin embargo, no constituye una solución general que permita resolver todos los problemas complejos de cooperación y coordinación entre procesos, ya que se trata de un mecanismo cuyo único objetivo es permitir a una liga física compartir múltiples canales lógicos.

Como perspectiva de trabajo futuro se tiene planeada la creación de diversos mecanismos cada vez más complejos a partir de los ya existentes, así como a partir de nuevas especificaciones siguiendo la metodología de desarrollo propuesta en este trabajo, y de esta manera, enriquecer la biblioteca. Asimismo, se planea la implantación de los nuevos mecanismos en la familia de procesadores *Transputer T9000*, ya que se considera que esta combinación de mecanismos, junto con el soporte de un sistema de comunicación de canales virtuales para la conectividad entre procesos ofrecerá mayores ventajas en el desarrollo de aplicaciones paralelas y distribuidas complejas.

En todo el mundo día a día aumenta cada vez más el uso de los Transputers, por tal motivo, en México no será sorprendente observar en un futuro muy próximo que un número mayor de personas, instituciones, centros de investigación e industrias harán suya esta tecnología con la finalidad de poder resolver distintos problemas para cualquiera de sus áreas de aplicación.

Apéndice A

Biblioteca de Nuevos Mecanismos

Los archivos correspondientes a los parámetros de los mecanismos creados son los siguientes:

```
>>>>>>>>>>>>>>>>>>> Archivo: FUNCION.OCC

--
-- Modelo de proceso "función" para utilizarse en el mecanismo Filtro
--

---((( Proceso función
PROC funcion( CHAN OF ANY canal.entrada, canal.salida,
              CHAN OF BOOL canal.alto )
  ---((( Declaraciones
  BOOL termina :
  ... Declarar el tipo de dato de la variable "mensaje"
  ---)
  ---((( Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    termina := TRUE
    WHILE termina
      ALT
        -- Si aún no se ha recibido la señal de terminación
        TRUE & SKIP
          SEQ
            -- Se recibe el mensaje por el canal.entrada
            canal.entrada ? mensaje

            ... Insertar aquí el código necesario para la transformación
                del mensaje

            -- Se envía el mensaje modificado por el canal.salida
            canal.salida ! mensaje
        -- Si por el canal.alto se recibe señal de terminación
        canal.alto ? termina
          -- Se cambia el estado de la variable
          termina := FALSE
  ---)))
:
---)))
```

```
>>>>>>>>>>>>>>>>>>> Archivo: PUENTE.INC

--
-- Parámetro del mecanismo Puente
--

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: BUFFER.INC

--
-- Parámetros del mecanismo Buffer
--

-- Especificar el número máximo de elementos "num.max.elem" en la cola
-- circular. Por omisión se han especificado 5000 elementos.
VAL num.max.elem IS 5000 :

-- Especificar el tipo de dato de la estructura cola circular "cola.circ"
[num.max.elem]INT cola.circ :

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: MUXMAU.INC

--
-- Parámetro del mecanismo Multiplexor.muchos.a.uno
--

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: PMUXUAM.INC

--
-- Parámetro del mecanismo Multiplexor.uno.a.muchos
--

-- Especificar el protocolo para el "canal.peticion"
-- Por omisión se ha especificado el siguiente
PROTOCOL PETICION
  CASE
    senal
:

-- Se sugiere consultar Cap. 3 sec. 3.3.2.5, pag. 45
```

```
>>>>>>>>>>>>>>>>>>> Archivo: MUXUAM.INC

--
-- Parámetro del mecanismo Multiplexor.uno.a.muchos
--

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: DIFUSOR.INC

--
-- Parámetro del mecanismo Difusor
--

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT
```

El archivo que conforma la Biblioteca de Nuevos Mecanismos para la Comunicación y Sincronización es el siguiente:

```
>>>>>>>>>>>>>>>>>>> Archivo: COMUSINC.OCC

#COMMENT   "Biblioteca de Nuevos Mecanismos para la Comunicación y "
#COMMENT   "Sincronización en OCCAM y TRANSPUTERS, V3.00, 07/08/93"
#COMMENT   "(c) Derechos Reservados Manuel Romero Salcedo, 1993"
#COMMENT   "Maestría en Ciencias de la Computación, UACPyP del CCH,"
#COMMENT   "Universidad Nacional Autónoma de México"


---{{{ Mecanismo Filtro
---{{{ Reglas de utilización
#COMMENT "Uso: Filtro( canal.entrada, canal.salida, canal.alto )"
#COMMENT "Parámetros: Se debe construir el proceso función, respetando"
#COMMENT "los parámetros establecidos. Ver el modelo de este proceso función"
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto."
---}}}
PROC Filtro( CHAN OF ANY canal.entrada, canal.salida,
             CHAN OF BOOL canal.alto )
  ---{{{ Referencias
  -- Inclusión de la declaración externa del código compilado
  -- del proceso función (definido por el programador)
  #USE "funcion"
  ---}}}
  ---{{{ Cuerpo del proceso
  funcion( canal.entrada, canal.salida, canal.alto )
  ---}}}
:
---}}}
```

```
---((( Mecanismo Puente
---((( Reglas de utilización
#COMMENT "Uso: Puente( canal.fuente, canal.destino, canal.alto )"
#COMMENT "Parámetros: Se debe declarar el tipo de dato de la variable"
#COMMENT "local (mensaje)."
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto."
---)))
PROC Puente( CHAN OF ANY canal.fuente, canal.destino,
             CHAN OF BOOL canal.alto )
  ---((( Referencias
  -- Inclusión de la declaración externa del tipo de dato
  -- de la variable local "mensaje" (definida por el programador)
  #INCLUDE "puente.inc"
  ---)))
  --((( Declaraciones
  BOOL termina :
  --)))
  --((( Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    termina := TRUE
    -- Mientras no se reciba señal de terminación, haz
    WHILE termina
      ALT
        -- Si aún no se ha recibido la señal de terminación
        TRUE & SKIP
          SEQ
            -- Se recibe el mensaje por el canal.fuente
            canal.fuente ? mensaje
            -- Se reenvía el mensaje por el canal.destino
            canal.destino ! mensaje
        -- Si por canal.alto se recibe la señal de terminación
        canal.alto ? termina
          -- Se cambia el estado de la variable
          termina := FALSE
  ---)))
:
---)))
```

```
--{{{ Mecanismo Buffer
--{{{ Reglas de utilización
#COMMENT "Uso: Puente( canal.fuente, canal.destino, canal.alto )"
#COMMENT "Parámetros: Se debe declarar el número máximo de elementos"
#COMMENT "en la cola (num.max.elem), el tipo de dato de la variable"
#COMMENT "local (mensaje) y el tipo de dato de la estructura cola"
#COMMENT "circular (cola.circ)."
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto,"
#COMMENT "además se debe cumplir que no existan mensajes almacenados en la c
--}}}
PROC buffer( CHAN OF ANY canal.izquierdo, canal.derecho,
             CHAN OF BOOL canal.alto )
  --{{{ Referencias
  -- Inclusión de la declaración externa del número máximo de
  -- elementos de la cola circular "num.max.elem", del tipo de
  -- dato tanto de la estructura cola circular "cola.circ" como
  -- de la variable local "mensaje" (definidos por el programdor)
  #INCLUDE "buffer.inc"
  --}}}
  --{{{ Declaraciones
  INT contenido, inicio, fin :
  BOOL termina :
  --}}}
  --{{{ Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    contenido, inicio, fin, termina := 0, 0, 0, TRUE
    -- Mientras no se reciba señal de terminación, haz
    WHILE termina
      ALT
        -- Si hay lugar en la cola circular y además se recibe un
        -- mensaje por el canal.izquierdo
        ( contenido < num.max.elem ) & canal.izquierdo ? mensaje
          SEQ
            -- Se inserta el mensaje en la cola circular
            cola.circ[fin] := mensaje
            -- Se actualizan apuntadores
            contenido, fin := contenido + 1, ( fin + 1 ) REM num.max.elem
        -- Si la cola no está vacía
        ( contenido > 0 ) & SKIP
          SEQ
            -- Se envía el siguiente mensaje por el canal.derecho
            canal.der ! cola.circ[inicio]
            -- Se actualizan apuntadores
            contenido, inicio := contenido - 1, ( inicio + 1 ) REM num.max.e
        -- Si ya no hay lugar en la cola circular y además se recibe
        -- la señal de terminación
        ( contenido = 0 ) & canal.alto ? termina
          -- Se cambia el estado de la variable
          termina := FALSE
  --}}}
:
--}}}
```

```
--{{{ Mecanismo Multiplexor "Muchos a Uno"
--{{{ Reglas de utilización
#COMMENT "Uso: Multiplexor.muchos.a.uno( canal.izquierdo, canal.derecho,"
#COMMENT "                              canal.alto )"
#COMMENT "Parámetros: Se debe declarar el tipo de dato de la variable"
#COMMENT "local (mensaje)."
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto."
--}}}
PROC Multiplexor.muchos.a.uno([]CHAN OF ANY canal.izquierdo,
                              CHAN OF ANY canal.derecho,
                              CHAN OF BOOL canal.alto )
  --{{{ Referencias
  -- Declaración externa del tipo de dato de la variable local
  -- "mensaje" (definidos por el programador)
  #INCLUDE "muxmau.inc"
  --}}}
  --{{{ Declaraciones
  BOOL termina :
  --}}}
  --{{{ Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    termina := TRUE
    -- Mientras no se reciba señal de terminación, haz
    WHILE termina
      ALT
        ALT i = 0 FOR SIZE canal.izquierdo
          -- Si por el canal.izquierdo[i] (donde i = 1,...,número de canales
          -- izquierdos) se recibe un mensaje
          canal.izquierdo[i] ? mensaje
            -- Se envía el mensaje por el canal.derecho
            canal.derecho ! mensaje
        -- Si por canal.alto se recibe la señal de terminación
        canal.alto ? termina
          -- Se cambia el estado de la variable
          termina := FALSE
  --}}}
:
--}}}
```

```
--{{{ Mecanismo Multiplexor "Uno a Muchos"
--{{{ Reglas de utilización
#COMMENT "Uso: Mulitplexor.uno.a.muchos( canal.izquierdo, canal.derecho,
                                         canal.peticion, canal.alto )"
#COMMENT "Parámetros: Se debe declarar el tipo de dato de la variable"
#COMMENT "local (mensaje). No es indispensable modificar el protocolo"
#COMMENT "del canal.petición (PETICION), a menos que sea realmente necesario"
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto."
--}}}
--{{{ Referencias
-- Declaración externa del protocolo del canal.petición (definido por
-- el programador)
#INCLUDE "pmuxuam.inc"
--}}}
PROC Multiplexor.uno.a.muchos(  CHAN OF ANY canal.izquierdo,
                              []CHAN OF ANY canal.derecho,
                              []CHAN OF PETICION canal.peticion,
                                CHAN OF BOOL canal.alto )
  --{{{ Referencias
  -- Inclusión de la declaración externa del tipo de dato de la
  -- variable local "mensaje" (definida por el programador)
  #INCLUDE "muxuam.inc"
  --}}}
  --{{{ Declaraciones
  BOOL termina :
  --}}}
  --{{{ Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    termina := TRUE
    -- Mientras no se reciba señal de terminación, haz
    WHILE termina
      ALT
        -- Si por el canal.izquierdo se recibe un mensaje
        canal.izquierdo ? mensaje
          ALT i = 0 FOR SIZE canal.derecho
            -- Si por el canal.petición[i] (donde i = 1,...,número de
            -- canales derechos) se recibe señal de petición
            canal.peticion[i] ? CASE senal
              -- se envía el mensaje por canal derecho[i]
              canal.derecho[i] ! mensaje
          -- Si por canal.alto se recibe la señal de terminación
          canal.alto ? termina
            -- Se cambia el estado de la variable
            termina := FALSE
  --}}}
:
--}}}
```

```
---{{{ Mecanismo Difusor
---{{{ Reglas de utilización
#COMMENT "Uso: Difusor( canal.izquierdo, canal.derecho, canal.alto )"
#COMMENT "Parámetros: Se debe declarar el tipo de dato de la variable"
#COMMENT "local (mensaje)."
#COMMENT "Terminación: Cuando se desee terminar con éxito se debe enviar"
#COMMENT "algún valor booleano hacia el mecanismo, utilizando canal.alto."
---}}}
PROC Difusor( CHAN OF ANY canal.izquierdo, []CHAN OF ANY canal.derecho,
              CHAN OF BOOL canal.alto )
  ---{{{ Referencias
  -- Inclusión de la declaración externa del tipo de dato
  -- de la variable local "mensaje" (definida por el programador)
  #INCLUDE "difusor.inc"
  ---}}}
  ---{{{ Declaraciones
  BOOL termina :
  --}}}
  ---{{{ Cuerpo del proceso
  SEQ
    -- Inicialización de variables
    termina := TRUE
    -- Mientras no se reciba señal de terminación, haz
    WHILE termina
      SEQ
        -- Se recibe un mensaje por el canal.izquierdo
        canal.izquierdo ? mensaje
        ALT
          -- Si aún no se ha recibido la señal de terminación
          TRUE & SKIP
            PAR i = 0 FOR SIZE canal.derecho
              -- Se envía simultáneamente el mensaje a cada uno de los
              -- procesos conectados a canal.derecho[i] (donde
              -- i = 1,...,número de canales derechos)
              canal.derecho[i] ! mensaje
          -- Si por canal.alto se recibe la señal de terminación
          canal.alto ? termina
            -- Se cambia el estado de la variable
            termina := FALSE
  ---}}}
:
---}}}
```

Apéndice B

Implantación del Problema de los Productores-Consumidores

Con el fin de ejemplificar el uso de la Biblioteca de Nuevos Mecanismos a continuación se muestra la implantación del problema Productores-Consumidores, la cual se realizó en una tarjeta *MicroWay*^{TM} *Quadputer* con cuatro procesadores Transputer del tipo T800. Esta tarjeta se muestra en la figura B.1.

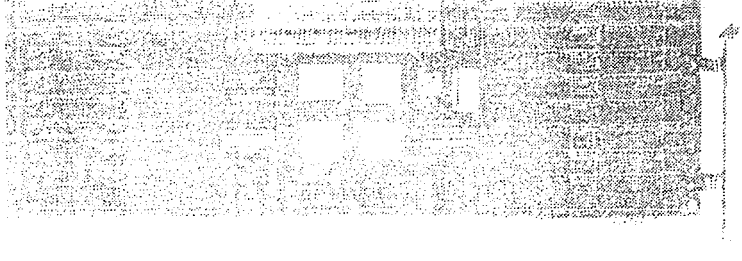


Figura B.1: Tarjeta Quadputer con Cuatro Procesadores T800

Esta tarjeta Quadputer posee una *tarjeta hija de conexión completa*, la cual permite enlazar a los cuatro procesadores. En la figura B.2 se aprecia este enlace de ligas físicas entre los Transputers.

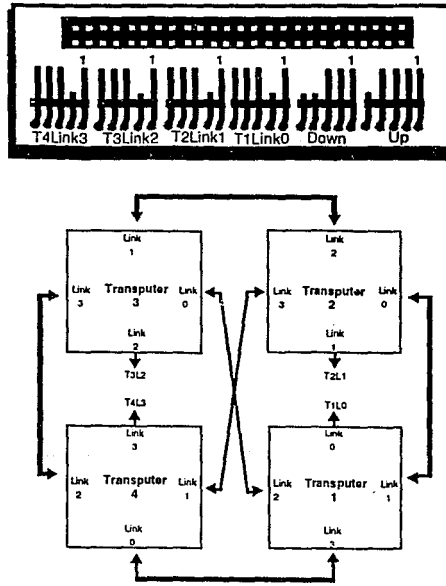


Figura B.2: **Tarjeta Hija e Interconexión de Transputers**

Los mecanismos utilizados para implantar el problema del los Productores-Consumidores fueron:

* Buffer
* Multiplexor.muchos.a.uno
* Multiplexor.uno.a.muchos
* Puente

Se actualizaron los archivos correspondientes a los parámetros de estos mecanismos. Estos archivos son los siguientes:

```
>>>>>>>>>>>>>>>>>>> Archivo: BUFFER.INC

-- Especificar el número máximo de elementos "num.max.elem" en la cola
-- circular. Por omisión se han especificado 5000 elementos.
VAL num.max.elem IS 5000 :

-- Especificar el tipo de dato de la estructura cola circular "cola.circ"
[num.max.elem]INT cola.circ :

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: MUXMAU.INC

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: PMUXUAM.INC

-- Especificar el protocolo para el "canal.peticion"
-- Por omisión se ha especificado el siguiente
PROTOCOL PETICION
  CASE
    senal
:

-- Se sugiere consultar Cap. 3 sec. 3.3.2.5, pag. 45


>>>>>>>>>>>>>>>>>>> Archivo: MUXUAM.INC

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT


>>>>>>>>>>>>>>>>>>> Archivo: PUENTE.INC

-- Especificar el tipo de dato de la variable "mensaje"
INT mensaje :

-- Se sugiere consultar Cap. 3 sec. 3.3.2.4, pag. 40
-- Por omisión se ha especificado el tipo INT
```

Los archivos que componen la aplicación son los siguientes:

```
>>>>>>>>>>>>>>>>>>> Archivo: NUMALEAT.OCC

--{{{ Función Random
-- Esta función obtiene un número aleatorio en el rango de los enteros
-- a partir de una semilla
INT FUNCTION Random( VAL INT semilla )
  --{{{ Declaraciones
  INT resultado :
  --}}}
  --{{{ Cuerpo de la función
  VALOF
    IF
      semilla = 0
        resultado := #1357
      TRUE
        SEQ i = 0 FOR 16
          resultado := (((((semilla/#1)<<15)><((semilla/#10)<<11))><
                        ((semilla/#2000)<<2))><(semilla/#8000))/ (semilla>>1)
    RESULT resultado
  --}}}
:
--}}}

>>>>>>>>>>>>>>>>>>> Archivo: RETARDO.OCC

--{{{ Proceso Retardo
-- Este proceso cuando es invocado ejecuta un retardo de tiempo aleatorio
PROC Retardo( VAL BOOL inicializa )
  --{{{ Referencias
  #INCLUDE "numaleat.occ"
  --}}}
  --{{{ Declaraciones
  TIMER reloj :
  INT ahora, retardo :
  --}}}
  --{{{ Cuerpo del proceso
  SEQ
    IF
      inicializa
        retardo := MOSTPOS INT
      TRUE
        retardo := MOSTNEG INT
    retardo := Random( retardo )
    reloj ? ahora
    reloj ? AFTER ahora PLUS retardo
    retardo := ahora
  --}}}
:
--}}}
```

```
>>>>>>>>>>>>>>>>>>> Archivo: PRO-CON.INC

--((( Declaraciones
-- Definición de valores constantes del problema Productores-Consumidores
VAL num.productores  IS 20 :
VAL num.consumidores IS 40 :
VAL num.producciones IS 400 :
VAL num.consumos IS ( num.producciones * num.productores ) / num.consumidore
--)))



>>>>>>>>>>>>>>>>>>> Archivo: PRODUCTO.OCC

--((( Proceso Productor
-- Este proceso genera productos y los envía por el canal.izquierdo;
-- asimismo los envía por el canal.sal.prod para ser desplegados
PROC Productor( CHAN OF INT canal.izquierdo, CHAN OF INT canal.sal.prod )
  --((( Referencias
  #USE "retardo"
  #INCLUDE "pro-con.inc"
  --)))
  --((( Declaraciones
  INT produccion :
  --)))
  --((( Cuerpo del proceso
  SEQ
    produccion := 0
    WHILE produccion < num.producciones
      SEQ
        produccion := produccion + 1
        PAR
          canal.izquierdo ! produccion
          canal.sal.prod ! produccion
        Retardo( TRUE )
  --)))
:
--)))



>>>>>>>>>>>>>>>>>>> Archivo: CONSUMID.OCC

--((( Proceso Consumidor
-- Este proceso consume productos. Para lo cual primero realiza la petición
-- del mismo y después lo recibe por el canal.derecho; asimismo los envía po
-- el canal.sal.cons para ser desplegados
--((( Referencias
#INCLUDE "pmuxuam.inc"
--)))
PROC Consumidor( CHAN OF INT canal.derecho, CHAN OF INT canal.sal.cons,
                 CHAN OF PETICION canal.peticion )
  --((( Referencias
  #USE "retardo"
  #INCLUDE "pro-con.inc"
  --)))
  --((( Declaraciones
  INT consumo, contador :
  --)))
```

```
    --((( Cuerpo del proceso
    SEQ
      contador := 0
      WHILE contador < num.consumos
        SEQ
          canal.peticion ! senal
          canal.derecho ? consumo
          canal.sal.cons ! consumo
          contador := contador + 1
          Retardo( FALSE )
    --)))
:
--)))


>>>>>>>>>>>>>>>>>> Archivo: SUPERVIS.OCC

    --((( Proceso Supervisor
    --Este proceso despliega todas las producciones y los consumos en pantalla
    --((( Referencias
    #INCLUDE "hostio.inc"
    --)))
    PROC supervisor( CHAN OF SP fs, ts, CHAN OF INT canal.sal.prod, canal.sal.co
                     CHAN OF BOOL alto )
      --((( Referencias
      #USE "hostio.lib"
      --)))
      --((( Declaraciones
      INT elemento :
      BOOL termina :
      --)))
      --((( Cuerpo del proceso
      SEQ
        termina := TRUE
        WHILE termina
          SEQ
            so.write.nl( fs, ts )
            ALT
              canal.sal.prod ? elemento
                SEQ
                  so.write.string( fs, ts, "Producción" )
              canal.sal.cons ? elemento
                SEQ
                  so.write.string( fs, ts, "Consumo" )
              alto ? termina
                termina := FALSE
            IF
              termina
                SEQ
                  so.write.string( fs, ts, "  del elemento:" )
                  so.write.int( fs, ts, elemento, 4 )
                  so.write.nl( fs, ts )
              TRUE
                SKIP
        so.exit( fs,ts, sps.success )
      --)))
:
--)))
```

```
>>>>>>>>>>>>>>>>>>> Archivo: GRUTRA1.OCC

--((( Proceso Grupo.Transputer1
-- Este proceso agrupa a los procesos Multiplexor.uno.a.muchos, Consumidores
-- y Multiplexor.muchos.a.uno para poder ser mapeados en el Transputer 1
--((( Referencias
#INCLUDE "pro-con.inc"
--)))
PROC Grupo.Transputer1(CHAN OF ANY canal.derecho, CHAN OF BOOL canal.alto4,
                       canal.alto2, CHAN OF ANY canal.consumo )
  --((( Referencias
  -- ******* Inclusión de la Biblioteca de Nuevos Mecanismos *******
  #USE "comusinc.lib"
  #USE "consumid"
  --)))
  --((( Declaraciones
  [num.consumidores]CHAN OF ANY canal.izquierdo, canal.sal.cons :
  [num.consumidores]CHAN OF PETICION canal.peticion :
  CHAN OF BOOL canal.termina :
  --)))
  --((( Cuerpo del proceso
  PAR
    Multiplexor.uno.a.muchos( canal.derecho, canal.izquierdo,
                              canal.peticion, canal.alto4 )
    SEQ
      PAR i = 0 FOR num.consumidores
        Consumidor( canal.izquierdo[i], canal.sal.cons[i], canal.peticion[i]
      PAR
        canal.alto2  ! FALSE
        canal.termina ! FALSE
    Multiplexor.muchos.a.uno( canal.sal.cons, canal.consumo, canal.termina )
  --)))
:
--)))



>>>>>>>>>>>>>>>>>>> Archivo: GRUTRA2.OCC

--((( Proceso Grupo.Transputer2
-- Este proceso agrupa a los procesos Buffer y Puente para poder ser
-- mapeados en el Transputer 2
--((( Referencias
#INCLUDE "buffhead.inc"
--)))
PROC Grupo.Transputer2( CHAN OF ANY canal.izquierdo, canal.derecho,
                        CHAN OF BOOL canal.alto0, canal.alto1,
                        CHAN OF ANY canal.2, canal.3 )
  --((( Referencias
  -- ******* Inclusión de la Biblioteca de Nuevos Mecanismos *******
  #USE "comusinc.lib"
  --)))
  --((( Cuerpo del proceso
  PAR
    SEQ
      Buffer( canal.izquierdo, canal.derecho, canal.alto0 )
      canal.alto1 ! FALSE
    Puente( canal.alto2, canal.alto3 )
  --)))
:
--)))
```

```
>>>>>>>>>>>>>>>>>>> Archivo: GRUTRA3.OCC

--{{{ Proceso Grupo.Transputer3
-- Este proceso agrupa a los procesos Productores, Multiplexor.muchos.a.uno
-- y Puente para poder ser mapeados en el Transputer 3
PROC Grupo.Transputer3( CHAN OF BOOL canal.alto0, CHAN OF ANY canal.alto1,
                        canal.alto4, canal.izquierdo, canal.produccion )
  --{{{ Referencias
  -- ******* Inclusión de la Biblioteca de Nuevos Mecanismos *******
  #USE "comusinc.lib"
  #USE "producto"
  #INCLUDE "pro-con.inc"
  --}}}
  --{{{ Declaraciones
  [num.productores]CHAN OF ANY canal.derecho, canal.sal.prod :
  CHAN OF BOOL canal.alto, canal.termina :
  --}}}
  --{{{ Cuerpo del proceso
  PAR
    SEQ
      PAR i = 0 FOR num.productores
        Productor( canal.derecho[i], canal.sal.prod[i] )
      PAR
        canal.alto ! FALSE
        canal.termina ! FALSE
    SEQ
      Multiplexor.muchos.a.uno( canal.derecho, canal.izquierdo, canal.alto )
      canal.alto0 ! FALSE
    Multiplexor.muchos.a.uno( canal.sal.prod, canal.produccion, canal.termin
    Puente( canal.alto1, canal.alto4 )
  --}}}
:
--}}}



>>>>>>>>>>>>>>>>>>> Archivo: PRO-CON.PGM

--{{{ Archivo de configuración
-- Este archivo de configuración especifica el mapeo de los procesos en los
-- procesadores y el de los canales lógicos en las ligas físicas
--{{{ Referencias
#INCLUDE "hostio.inc"
#INCLUDE "linkaddr.inc"
#USE "supervis.c8h"
#USE "grutra1.c8h"
#USE "grutra2.c8h"
#USE "grutra3.c8h"
--}}}
--{{{ Declaraciones
CHAN OF ANY canal.izquierdo, canal.derecho :
CHAN OF ANY canal.produccion, canal.consumo :
[5]CHAN OF ANY canal.alto :
CHAN OF BOOL canal.termina :
CHAN OF SP fs, ts :
--}}}
```

```
--{{{ Cuerpo del mapeo
PLACED PAR
  PROCESSOR 0 T800
    PLACE ts AT link0.out:
    PLACE fs AT link0.in:
    PLACE canal.produccion AT link3.in :
    PLACE canal.consumo    AT link1.in :
    PLACE canal.alto[3]    AT link2.in :
    Supervisor( fs, ts, canal.produccion, canal.consumo, canal.alto[3] )
  PLACED PAR
    PROCESSOR 1 T800
      PLACE canal.derecho AT link2.in  :
      PLACE canal.alto[4] AT link3.in  :
      PLACE canal.consumo AT link0.out :
      PLACE canal.alto[2] AT link2.out :
      Grupo.Transputer1( canal.derecho, canal.alto[4], canal.alto[2],
                         canal.consumo )
  PLACED PAR
    PROCESSOR 2 T800
      PLACE canal.izquierdo AT link3.in  :
      PLACE canal.derecho   AT link1.out :
      PLACE canal.alto[0]   AT link2.in  :
      PLACE canal.alto[1]   AT link2.out :
      PLACE canal.alto[2]   AT link1.in  :
      PLACE canal.alto[3]   AT link0.out :
      Grupo.Transputer2( canal.izquierdo, canal.derecho, canal.alto[0],
                         canal.alto[1], canal.alto[2], canal.alto[3] )
  PLACED PAR
    PROCESSOR 3 T800
      PLACE canal.alto[0]    AT link3.out :
      PLACE canal.alto[1]    AT link3.in  :
      PLACE canal.alto[4]    AT link1.out :
      PLACE canal.izquierdo  AT link2.out :
      PLACE canal.produccion AT link0.out :
      Grupo.Transputer3( canal.alto[0], canal.alto[1], canal.alto[4],
                         canal.izquierdo, canal.produccion )
--}}}
```

La especificación de esta descripción se resume en las tablas siguientes que corresponden al mapeo de procesos en los procesadores y de canales lógicos en ligas físicas, respectivamente.

Procesos	Transputer
Supervisor	0
Grupo.Transputer1	1
Grupo.Transputer2	2
Grupo.Transputer3	3

	Fuente		Destino	
Canal	Transputer	Liga	Transputer	Liga
ts	0	0	HOST	–
fs	HOST	–	0	0
canal.produccion	3	0	0	3
canal.consumo	1	0	0	1
canal.alto[3]	2	0	0	2
canal.derecho	2	1	1	2
canal.alto[4]	3	1	1	3
canal.alto[2]	1	2	2	1
canal.izquierdo	3	2	2	3
canal.alto[0]	3	3	2	2
canal.alto[1]	2	2	3	3

En la figura B.3 se aprecia gráficamente el mapeo de los procesos y de los canales lógicos en el hardware.

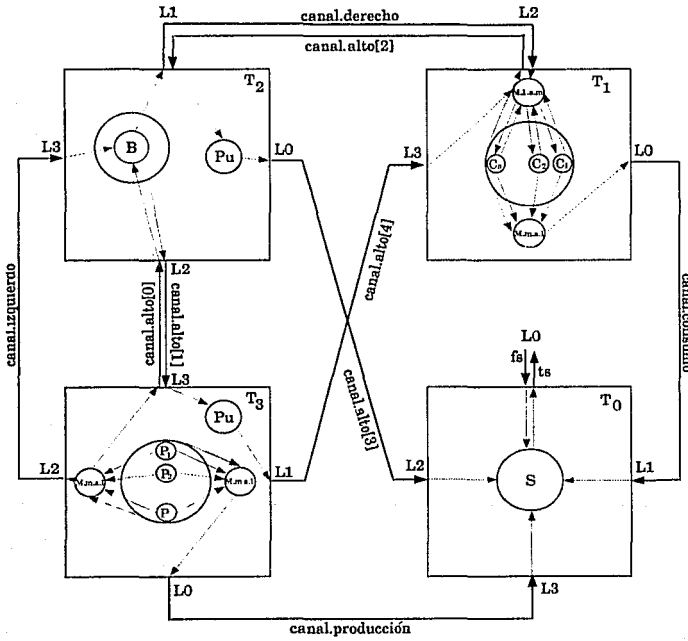


Figura B.3: Productores-Consumidores Implantado en un Quadputer

Referencias y Bibliografía

[1] G. Andrews. "Paradigms for Process Interaction in Distributed Programs". *Computing Surveys*, 23(5):49–90, 1991.

[2] T. Axford. *"Concurrent Programming"*. John Wiley & Sons, 1989.

[3] M. Ben-Ari. *"Principles of Concurrent and Distributed Programming"*. Prentice-Hall, 1990.

[4] A. Bernstein. "Output Guards and Nondeterminism in Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems*, 2(3):234–238, 1980.

[5] A. Burns. *"Programming in OCCAM 2"*. Addison-Wesley, 1988.

[6] M. Conway. "Design of a Separable Transition Diagram Compiler". *Communications of the ACM*, 6(7):396–408, 1963.

[7] E. Dijkstra. *"Cooperating Sequential Processes"*. Academic Press, 1968.

[8] E. Dijsktra. "The Structure of the THE Multiprogramming System". *Communications of the ACM*, 11(5):341–346, 1968.

[9] E. Dijsktra. "Guarded Commands, Nondeterminancy and Formal Derivation of Programs". *Communications of the ACM*, 18(8):453–457, 1975.

[10] R. Duncan. "A Survey of Parallel Computer Architectures". *Computer*, 23(2):5–16, 1990.

[11] A. Habermann. "Sinchronization of Communicating Processes". *Communications of the ACM*, 15(3):171–176, 1972.

[12] P. Brinch Hansen. "Structured Multiprogramming". *Communications of the ACM*, 15(7):574–578, 1972.

[13] P. Brinch Hansen. "The Programming Language Concurrente Pascal". *IEEE Transactions on Software Engineering*, SE-1(2):199–206, 1975.

[14] P. Brinch Hansen. "Distributed Processes: A Concurrent Programming Concept". *Communications of the ACM*, 21(11):934-941, 1978.

[15] C.A.R. Hoare. "Towards a Theory of Parallel Programming". En "Operating Systems Techniques", Academic Press, N.Y., 1972.

[16] C.A.R. Hoare. "Monitors: An Operating System Structuring Concept". *Communications of the ACM*, 17(10):549-557, 1974.

[17] C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, 21(8):666-677, 1978.

[18] C.A.R. Hoare. *"Comunicating Sequential Processes"*. Prentice-Hall, 1985.

[19] G. Jones. "On Guards". Proc. 7th. OCCAM User Group Int., Workshop on Parallel Programming of Transputer Based Machines, editor, Grenoble, Francia, 1987.

[20] G. Jones. *"Programming in OCCAM"*. Prentice-Hall, 1987.

[21] J. Kerridge. *"OCCAM Programming: a Practical Approach"*. Blackwell Scientific Publications, 1987.

[22] INMOS Limited, editor. *"OCCAM 2 Reference Manual"*. Prentice-Hall, 1988.

[23] INMOS Limited, editor. *"Transputer Instruction Set"*. Prentice-Hall, 1988.

[24] INMOS Limited, editor. *"Transputer Reference Manual"*. Prentice-Hall, 1988.

[25] INMOS Limited, editor. *"OCCAM 2 User Manual"*. INMOS Lim., 1989.

[26] INMOS Limited, editor. *"The T9000 Transputer"*. INMOS Lim., 1991.

[27] S. Mullender. *"Distributed Systems"*. Addison-Wesley, 1989.

[28] H. Oktaba. "Programación Concurrente". Serie Azul, Primera Parte, 86, Comunicaciones Técnicas del IIMAS, 1985.

[29] H. Oktaba. "Programación Concurrente". Serie Azul, Segunda Parte, 100, Comunicaciones Técnicas del IIMAS, 1987.

[30] R.H. Perrott. *"Parallel Programming"*. Addison-Wesley, 1987.

[31] G. Peterson. "Myths about the the Mutal Exclusion Problem". *Information Processing Letters*, 12(3):115-116, 1981.

[32] J. Peterson. *"Petry Net Theory and the Modeling of Systems"*. Prentice-Hall, 1981.

[33] D. Pountain. "OCCAM II". *BYTE*, 14(10):279-284, 1989.

[34] D. Pountain. "Virtual Channels: The Next Generation of Transputers". *BYTE*, 15(4):E&W 3-E&W 12, 1990.

[35] D. Pountain. "The Transputer Strikes Back". *BYTE*, 16(8):265-275, 1991.

[36] M. Romero. "Programación Paralela en OCCAM". Reportes de la Maestría en Ciencias de la Computación, 4, UACPyP del CCH / IIMAS, UNAM, 1991.

[37] M. Romero. "Programación Paralela en OCCAM". *Revista ContactoS, UAM-Iztapalapa*, Parte I, Nueva Época(6):47-57, 1992.

[38] M. Romero. "Programación Paralela en OCCAM". *Revista ContactoS, UAM-Iztapalapa*, Parte II, Nueva Época(7):49-56, 1992.

[39] A. Roscoe. *"Denotational Semantics for OCCAM"*. Presentado en el seminario sobre concurrencia NSF/SERC, Carnegie-Mellon University, 1984.

[40] A.W. Roscoe. "Routing Messages Through Networks: An Exercice in Deadlock Avoidance". Proc. 7th. OCCAM User Group Int., Workshop on Parallel Programming of Transputer Based Machines, editor, Grenoble, Francia, 1987.

[41] A. Tanenbaum. *"Sistemas Operativos Diseño e Implementación"*. Prentice-Hall, 1988.

[42] N. Tucker. "Commercial Issues: Parallel Processing and the Transputer". *Microprocessors and Microsystems*, 13(2):139-141, 1989.

[43] P. Walker. "The Transputer". *BYTE*, 10(3):219-235, 1985.

[44] J. Wexler. *"Concurrent Programming in OCCAM 2"*. John Wiley & Sons, 1989.

[45] H. Bal y A. Tanenbaum et al. "Programming Languages for Distributed Computing Systems". *Communications of the ACM*, 21(3):261-322, 1989.

[46] P.C.Capon y A.J. West. "Monitoring OCCAM Channels by Program Transformation". Proc. 7th. OCCAM User Group Int., Workshop on Parallel Programming of Transputer Based Machines, editor, Grenoble, Francia, 1987.

[47] A. Rascoe y C. Hoare. "The Laws of OCCAM Programming". Monografía técnica No. PRG-53, Oxford University Computing Laboratory, Programming Research Group, 1986.

[48] S. Owicki y D. Gries. "Verifying Properties of Paralell Programs: An Axiomatic Aproach". *Communications of the ACM*, 19(5):279–285, 1976.

[49] D. May y D. Pountain. *"A Tutorial Introduction to OCCAM Programming"*. INMOS Lim., 1988.

[50] G. Andrews y F. Schneider. "Concepts and Notations for Concurrent Programming". *Computing Surveys*, 15(1):4–43, 1983.

[51] C. Hull y M. McKeag et al. "Communicating Sequential Processes for Centralized and Distributed Operating System Design". *ACM Transactions on Programming Languages and Systems*, 6(9):175–191, 1984.

[52] A. Knowles y Todor Kantchev. "Messages Passing in a Transputer System". *Microprocessors and Microsystems*, 13(2):111–123, 1989.

[53] J. Dennis y V. Horn. "Programming Semantics for Multiprogrammed Computations". *Communications of the ACM*, 9(3):143–155, 1966.

[54] M. Romero y V.G. Sánchez. "Desarrollo e Implantación de Nuevos Mecanismos para la Comunicación y Sincronización en OCCAM y Transputer". En "Memorias del Simposium Nacional de Computación Avances y Perspectivas de la Computación", Centro Nacional de Cálculo-IPN, editor, México, D.F., (noviembre 6,7, y 8), 1991.

[55] M. Romero y V.G. Sánchez. "Laboratorio de Sistemas Distribuidos Basado en Transputers". En "Memorias de la Séptima Conferencia Internacional Las Computadoras en las Instituciones de Educación y de Investigación", DGSCA-UNAM, editor, México, D.F., 1991.

[56] M. Romero y V.G. Sánchez. "Un Paradigma y una Arquitectura Paralela: OCCAM y Transputer, una Tendencia Tecnológica de los 90's". En "Memorias del Simposium Nacional de Computación Tendencias de la Computación en la Década de los 90", Centro Nacional de Cálculo-IPN, editor, México, D.F., (octubre 23-25), 1991.