INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
CAMPUS ESTADO DE MÉXICO - TOLUCA

PROTOCOLO DE COMUNICACIÓN EN UNA ARQUITECTURA DE TRANSPUTERS PARA CONTROLAR UNA CELDA FLEXIBLE DE MANUFACTURA.

TESIS QUE PARA OPTAR EL GRADO DE
MAESTRO EN CIENCIAS COMPUTACIONALES
PRESENTA

SARA DEL SOCORRO MOTA GONZÁLEZ

Asesor: Dr. JESÚS SÁNCHEZ VELÁZQUEZ

té de tesis: Dr. LUIS TREJO RODRÍGUEZ
Dr. CARLOS RODRÍGUEZ LUCATERO
Dr. ISAAC RUDOMIN GOLDBERG

Jurado:

Dr. LUIS TREJO RODRÍGUEZ,	Presidente
Dr. ISAAC RUDOMIN GOLDBERG,	Secretario
Dr. CARLOS RODRÍGUEZ LUCATERO,	Vocal
Dr. JESÚS SÁNCHEZ VELÁZQUEZ,	Vocal

Atizapán de Zaragoza, Edo. Méx., Mayo de 1997.

A mi esposo Oscar, mi hija Jared, mi bebe y mis Padres.

Gracias Oscar y Jary por su solidaridad,
Jesus por trabajar a la par conmigo,
Lupita, por tu apoyo y
a todas las personas que me apoyaron.

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

1. INTRODUCCIÓN

Hoy en día el área de manufactura es un factor importante para el desarrollo económico de un país. La competencia internacional aumenta y puede enfrentarse sólo si los sistemas de manufactura pueden adaptarse rápidamente a los cambios que el mercado demanda. Por esta razón los sistemas de manufactura deben ser flexibles, reconfigurables y fáciles de programar.

En el proyecto "Tecnologías avanzadas de comunicación en Robótica y Celdas Flexibles de Manufactura" [1] estructurado bajo el programa Conacyt-NSF, se propone contribuir a mejorar la flexibilidad de sistemas de manufactura usando un controlador paralelo distribuido "inteligente" [2]. Con este trabajo de tesis damos los primeros pasos para la consecución de este objetivo, proponiendo algoritmos de comunicación entre el controlador y la Celda Flexible de Manufactura (CFM).

CIM (Celda Integrada de Manufactura) es un término usado para expresar la presencia de computadoras dentro del diseño, planeación, despacho y control de las operaciones de manufactura.

Una red de computadoras industriales no es un CIM, pues este término implica la integración lo más completa posible de todas las funciones de manufactura, buscando consecuentemente la creación de un sistema de procesamiento distribuido de datos.

Por otro lado, las Celdas Flexibles de Manufactura son sistemas de producción computarizados que toleran cambios en sus rutinas de trabajo ya sea por fallas o por modificaciones en los diferentes tipos de partes o piezas que se desean producir [3]. Por ejemplo, si una celda es usada para fabricar clavos, y se desean fabricar los mismos clavos pero sin cabeza, hay que cambiar la configuración de la CFM dado que las actividades necesarias para producir la nueva pieza serán diferentes.

Para lograr una ejecución coordinada de todas las partes de la CFM, es necesaria la creación de un sistema de control distribuido y unidades de manufactura que realicen su trabajo concurrentemente en tiempo real. El Sistema de Control tiene dos funciones principales:

1. Control de los componentes de la celda
2. Permitir la comunicación entre sus componentes

Y debe tener los siguientes requerimientos fundamentales:

- Confiabilidad
- Corrección
- Seguridad
- Operación en Tiempo Real
- Flexibilidad
- Comunicación eficiente
- Mecanismos de Sincronización

Existen muchas áreas de investigación que se derivan de ésto. Dedicamos este trabajo al desarrollo de los protocolos de comunicación entre los elementos del controlador propuesto por el Ing. Alarcón [4] en una tesis que se realizó bajo el mismo proyecto. El Centro Integrado de Manufactura básico sobre el cual se trabajó tiene la distribución que se muestra en la Fig. 1.1.

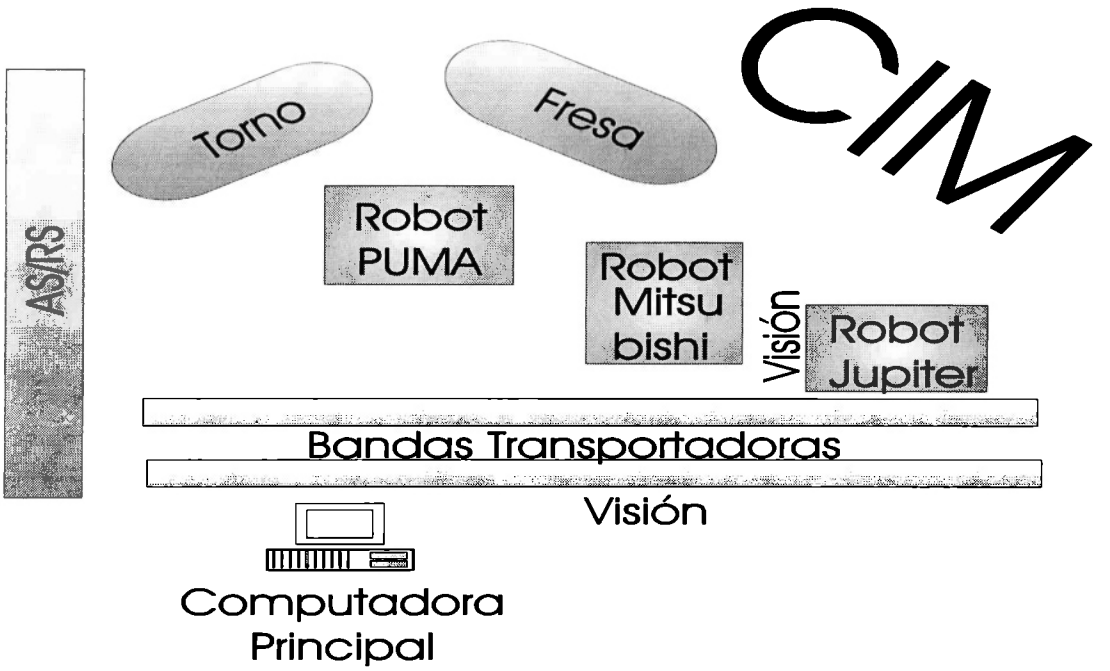


Fig. 1.1: Centro Integrado de Manufactura del ITESM Campus Toluca

1.1 Definición del Problema

Desde hace tiempo se ha desarrollado mucho trabajo de investigación en el área de manufactura para lograr optimizar el tiempo, costos y calidad de producción.

Se han realizado avances, sin embargo la tecnología y elementos usados no han logrado llenar los requerimientos para el óptimo funcionamiento de las áreas de producción en tiempo real.

Con el fin de automatizar la manufactura de productos, se han creado máquinas de control numérico (con distintos programas para su funcionamiento ante diferentes etapas de producción), robots programables y más aún, se han puesto en marcha celdas de manufactura que involucran los elementos anteriores con la ayuda de algunos elementos adicionales, como bandas para transportar material y/o piezas.

Sin embargo no se han logrado todavía las condiciones básicas que garantizan un buen desempeño en las Celdas Flexibles de Manufactura [1]:

1. Flexibilidad en planeación: habilidad de entender y procesar diferentes planes y de adaptarse a cambios en el ambiente y/o en los objetivos.
2. Reconfigurabilidad: habilidad para usar diferentes configuraciones de máquinas y robots.
3. Detección y recuperación a fallas: habilidad para detectar, identificar y recuperarse ante fallas.

El ITESM tiene años de experiencia desarrollando software y hardware para flexibilizar Celdas de Manufactura, participando principalmente en el diseño de la estructura del hardware para el control, del software de operación y de la comunicación entre los elementos de la celda.

En una Celda Flexible de Manufactura cada máquina cuenta con un sistema de control computarizado, el cual permite programar todas las operaciones que realizará. Es necesario, sin embargo contar con un controlador que permita coordinar la operación conjunta de todas las máquinas de la celda. En un proyecto anterior [5] se diseñó y construyó en el ITESM-CEM un controlador secuencial para celdas de manufactura (MIP) el cual utiliza una computadora central con la capacidad de comunicarse con todas las máquinas. El Módulo de interfaz Programable (MIP) toma las señales de varias máquinas a nivel de enlace y las multiplexa y demultiplexa para procesarlas. Esta solución presenta dos problemas: al ser centralizado el control, la respuesta no siempre es en tiempo real y además no es un sistema tolerante a fallas, pues al caer el procesador de control caería todo el sistema.

En esta tesis trabajaremos con una arquitectura que permita la tolerancia a fallas y el funcionamiento en tiempo real de una CFM [2]. Describiremos el uso de un controlador distribuido con una arquitectura reconfigurable basada en transputers T805 [11], así como la forma de utilizar la capacidad de paralelismo de este procesador en la creación de un protocolo con procesos paralelos y con velocidades de procesamiento que permitan respuestas en tiempo real.

1.2 Aplicación del Paralelismo en una CFM

El rápido desarrollo de la tecnología VLSI (Very Large Scale Integration) en la última década ha hecho posible realizar sistemas computacionales rápidos, confiables y poderosos interconectando varios procesadores entre sí. Hablamos entonces de máquinas paralelas, las cuales resuelven un problema usando el poder combinado de varios procesadores. En estas máquinas, uno de los temas de mayor preocupación para lograr que no se degrade la capacidad del sistema, es el esquema de comunicación entre procesadores.

Una arquitectura paralela puede ayudarnos a resolver el problema que estudiamos en esta tesis, pues hay elementos distribuidos físicamente que requieren ser controlados simultáneamente, y existe un requerimiento de redundancia para tolerar fallas. Diversos autores [2,3,4] han propuesto el uso de arquitecturas paralelas, compuestas de procesadores especializados como los *transputers*, para controlar celdas flexibles de manufactura.

Un Transputer [11] es un microprocesador de propósito general que reúne en un circuito integrado las facilidades de procesamiento paralelo, comunicación entre procesadores y memoria. Entre sus ventajas se encuentran la incorporación en hardware de un protocolo de comunicación que facilita su conexión con otros transputers [6] y el contar con un ordenador de tareas microprogramado que facilita el paralelismo. Fue desarrollado en 1980 como un módulo programable que puede ser utilizado en sistemas paralelos y tiene asociado un lenguaje de programación de alto nivel llamado OCCAM, el cual está basado en el lenguaje de especificación formal CSP [6].

Los transputers logran un excelente desempeño en máquinas paralelas ya que su estructura interna ayuda a la comunicación rápida [8] así como a la realización de operaciones en forma concurrente. Los elementos que componen al Transputer se muestran en la Fig. 1.2 y serán descritos a detalle en el capítulo 3.

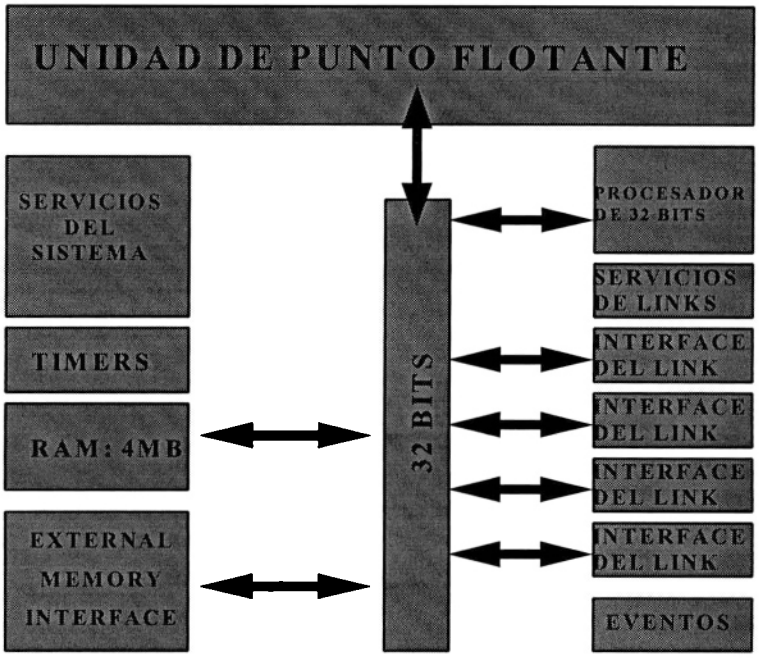


Fig. 1.2: Arquitectura general del transputer T805

1.3 Objetivos y Justificación del trabajo de tesis

Un protocolo según Holzmann [12] es un conjunto de normas para el intercambio de información en un sistema distribuido. Sin embargo, si quisiéramos dar una definición completa, diríamos que un protocolo es muy parecido a un lenguaje, pues involucra las tres partes siguientes:

- Define el formato preciso para los mensajes válidos (sintaxis).
- Define las reglas de procedimiento para el intercambio de datos (gramática).
- Y define un vocabulario de mensajes válidos que pueden ser intercambiados así como su significado (semántica).

En esta tesis se desarrollan los protocolos para la arquitectura del controlador paralelo de CFM propuesta por Alarcón [4], haciendo uso del lenguaje nativo del Transputer: (OCCAM).

Para el desarrollo del controlador de la CFM son de gran importancia los protocolos de comunicación, mismos que deberán permitir obtener respuestas en tiempo real. En este trabajo desarrollamos protocolos desde la capa de aplicación (el usuario) que indica los comandos a realizar, hasta la capa física (CFM, Robots). En la figura Fig. 1.3 se muestra el camino que debe seguir la información entre la celda de manufactura y el controlador.

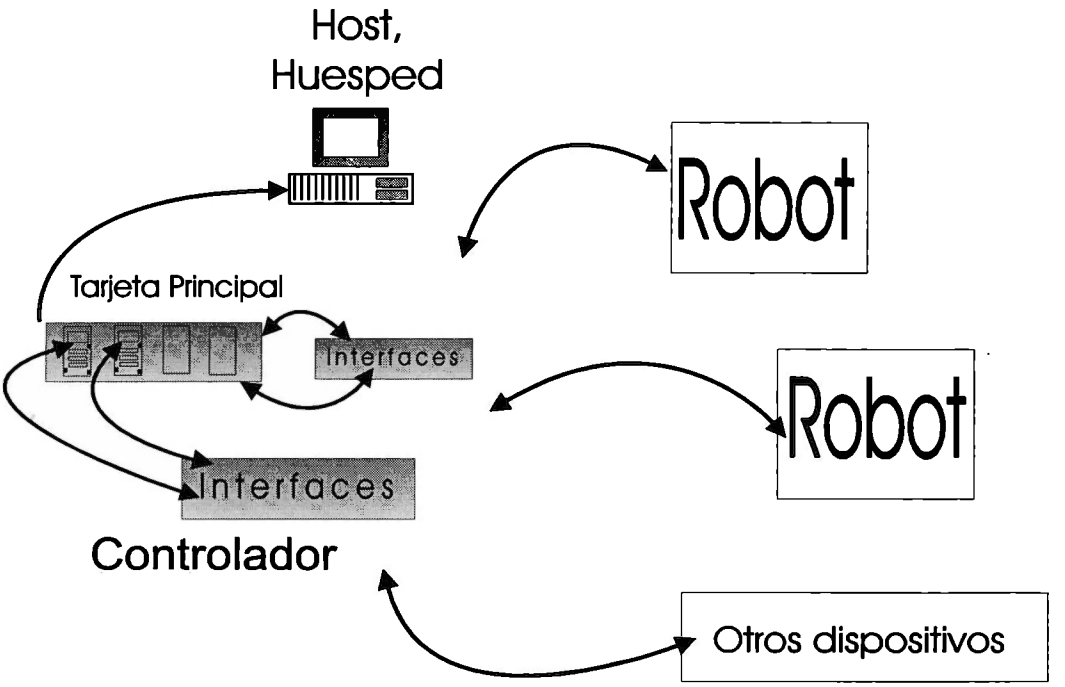


Fig. 1.3: Flujo de información entre la CFM y el controlador

Los protocolos que desarrollamos se ejecutan concurrentemente en los transputers y permiten que se comuniquen entre sí. También mostramos la forma de realizar la comunicación entre Transputers y elementos de la CFM.

En la Fig. 1.4 mostramos algunos elementos fundamentales de la arquitectura del controlador diseñada por Alarcón [4]. Se muestra esta arquitectura para ilustrar por donde deberá pasar la información y por ende dar una idea de los requerimientos de diseño de nuestros protocolos.

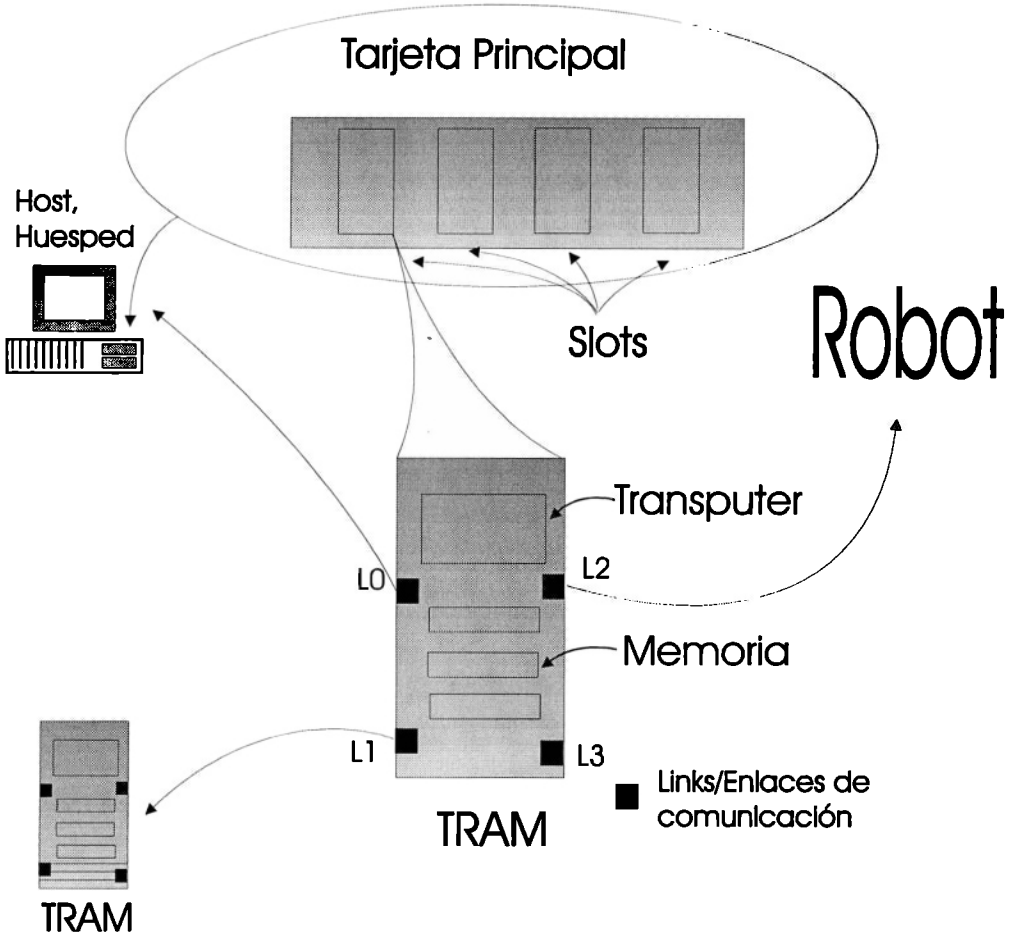


Fig. 1.4: Algunos elementos básicos de un controlador con Transputers

La tarjeta principal del controlador se inserta en una computadora huésped PC. En los slots (ranuras) de la tarjeta principal se insertan tarjetas que contienen Transputers y memoria. Estas tarjetas se llaman TRAM (Transputer RAM, Fig. 1.4), el cual es un estándar entre fabricantes de transputers.

Un solo transputer puede trabajar concurrentemente, por lo que se le podrían asignar varias tareas al mismo tiempo y así controlar todos los elementos involucrados en la CFM. Sin embargo, debido a los requerimientos de la celda, debemos prever la falla de algún transputer o enlace, por lo que se propone una arquitectura paralela redundante incluyendo más de un transputer para que compartan sus tareas. Ésto es posible gracias a los enlaces con que cuenta el transputer representados en la Fig. 1.4 por L0, L1, L2 y L3.

El diseño de los protocolos paralelos que presentamos en esta tesis se basa en la capacidad de paralelismo del transputer, ya que sus procesos se ejecutan concurrentemente[1], lo cual permite estar atendiendo diversas operaciones al mismo tiempo y dar una respuesta en tiempo real.

La disponibilidad de enlaces de comunicación de alta velocidad en el transputer (4 X 20 Mbps) es una de las principales razones por las cuales se consideró a este procesador como elemento base del controlador en [4]. Esta elección trajo desafortunadamente algunas desventajas.

Podemos mencionar, por ejemplo, que el costo del transputer es relativamente elevado (alrededor de 1000 USD) tomando como punto de comparación el costo de otros procesadores con producción en masa (por ejemplo la familia X86). Sin embargo, utilizar otra tecnología no nos permitiría alcanzar los requerimientos para el controlador anteriormente mencionados.

El contenido de esta tesis se divide en 6 capítulos. En el capítulo siguiente se describen los recursos, tanto en software como en hardware, con que se contó para la investigación. En el capítulo 3 se detalla la forma en que se usa la arquitectura del transputer a través de su lenguaje OCCAM. En el capítulo 4 se describen las técnicas usadas para diseñar y validar los protocolos de comunicación propuestos, así como los requerimientos de funcionamiento. En el capítulo 5 se describen las características de comunicación inherentes al transputer y a los elementos del controlador. Finalmente en el capítulo 6 se muestra el diseño de nuestro protocolo de comunicación para la CFM en todas sus capas.

[1] Cabe aclarar que el diseño fue paralelo pero la ejecución en el transputer es concurrente, donde los procesos compiten por los recursos de una manera muy eficiente

2. CARACTERÍSTICAS PARTICULARES DE LA CELDA, EQUIPO Y SOFTWARE CON QUE SE CUENTA

Para el desarrollo de esta tesis utilizamos las siguientes herramientas de hardware y software:

1. Una PC 80486 con 8 Mbytes en RAM, que de aquí en adelante llamaremos huésped.
2. Sistema Operativo MS-DOS ver. 6.22.
3. Un sistema de desarrollo para transputers en lenguaje OCCAM.
4. Una tarjeta principal modelo SMT004A para la comunicación entre Transputers y el huésped, así como para la administración de la comunicación entre los transputers usados.
5. Uno o más TRAMs, es decir, transputers con memoria de 4 Mbytes y los elementos necesarios de comunicación interna a RAM y hacia la tarjeta principal.
6. Los siguientes elementos de Hardware que forman parte de la arquitectura del controlador [4]:
   + Un Adaptador de enlace IMS C011 + Un microcontrolador 8051
   + Un registro 74LS373 +Una interfaz RS-232 (MAX 232)
7. El acceso a los elementos que componen al CIM del Campus Toluca, principalmente a los robots Puma y Mitsubishi.

A continuación describiremos más detalladamente los componentes del CIM.

2.1 CIM

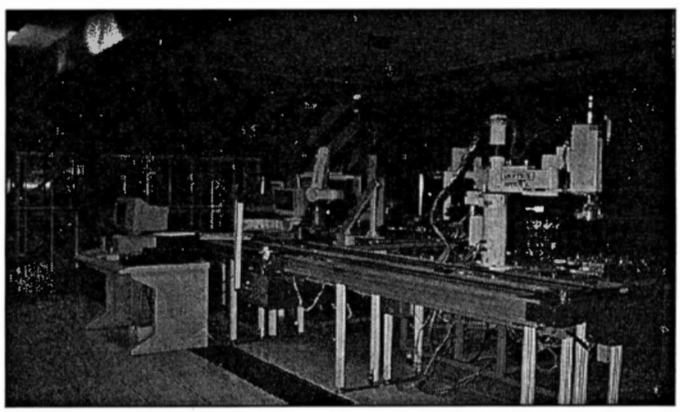


Fig. 2.1 Fotografía del CIM

El CIM del campus Toluca [9] cuenta con tres robots que se desempeñan de acuerdo a diferentes criterios de programación y de áreas de trabajo; con una fresadora y un torno que tienen integrada una máquina de control numérico; también tiene un mecanismo AS/RS y un conveyor (ver Fig. 2.1). La filosofía del CIM radica en mantener una base de datos centralizada en la cual todas las actividades, partes y subprocesos son actualizados durante el tiempo del proceso.

2.1.1 Robot Puma

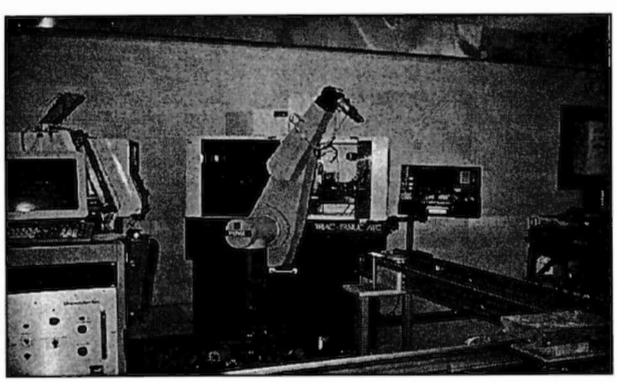


Fig. 2.2 Fotografía del Robot Puma

Es un robot de fabricación sueca (Fig. 2.2) de seis grados de libertad controlados por servomotores. Es clasificado como un robot de tipo esférico dado que su área de trabajo tiene esta característica. Actualmente sólo cuenta con 1 actuador neumático fijo en forma de garra.

Sus actividades están enfocadas hacia la alimentación al centro de maquinado y al torno con piezas para su maquinado.

2.1.2 Robot Mitsubishi

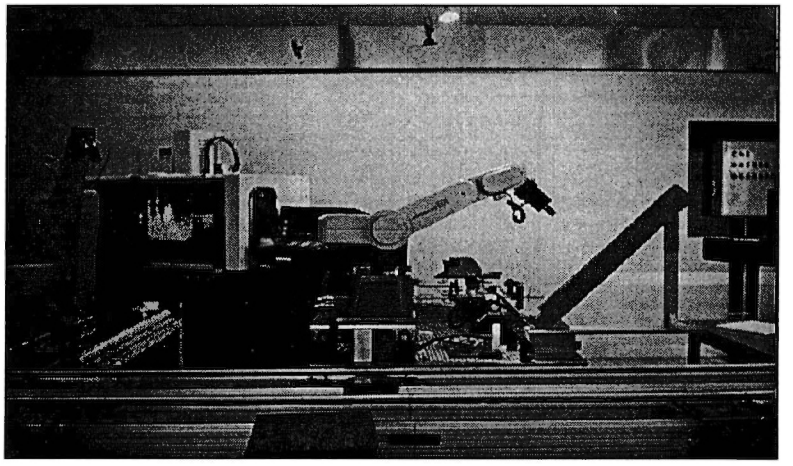


Fig. 2.3: Fotografía del Robot Mitsubishi

Este robot es de fabricación japonesa (Fig. 2.3) y tiene cinco grados de libertad servocontrolados con un actuador eléctrico al que puede regularse la presión de trabajo. El dispositivo de la base es no-servocontrolado y traslada al robot a una mesa de control de calidad. Al igual que el robot anterior, éste tiene un área de trabajo esférica.

2.1.3 Robot Amatrol Júpiter

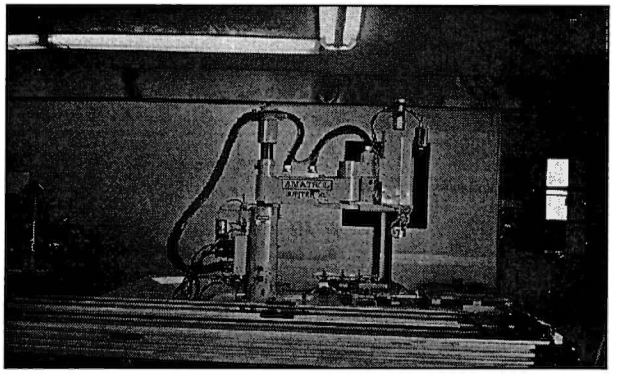


Fig. 2.4: Fotografía del Robot Júpiter

Este robot es de fabricación norteamericana (Fig. 2.4) y tiene cuatro grados de libertad también servocontrolados.

A diferencia de los anteriores, este robot tiene 3 actuadores neumáticos finales intercambiables (dos tenazas de diferente tamaño y un destornillador) y su área de trabajo tiene la forma de una rebanada de pastel, tipo "Scara". Gracias a su forma, este robot nunca pierde la verticalidad de sus piezas, por lo que muchas de sus aplicaciones están en la inserción de componentes.

2.1.4 Mecanismo AS/RS

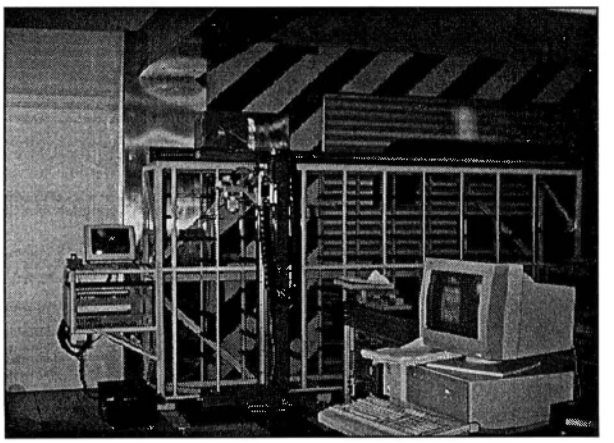


Fig. 2.5: Fotografía del mecanismo AS/RS

La marca de este mecanismo es Amatrol (Fig. 2.5), por lo cual su interfaz es igual a la del robot Júpiter y cuenta con sólo dos grados de libertad servocontrolados.

Sus movimientos (girar, entrar/salir de la bahía y del actuador) son controlados en forma neumática. Su función principal es proporcionar materia prima a la celda y almacenar el producto terminado. Su área de trabajo es un prisma rectangular.

2.1.5 Torno y Fresadora

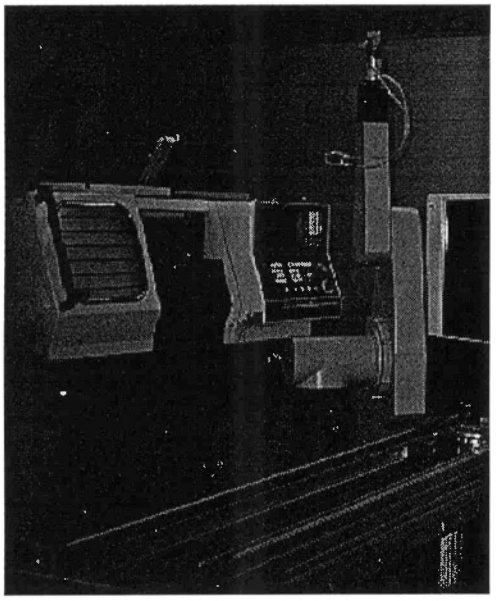


Fig. 2.6: Fotografía del Torno

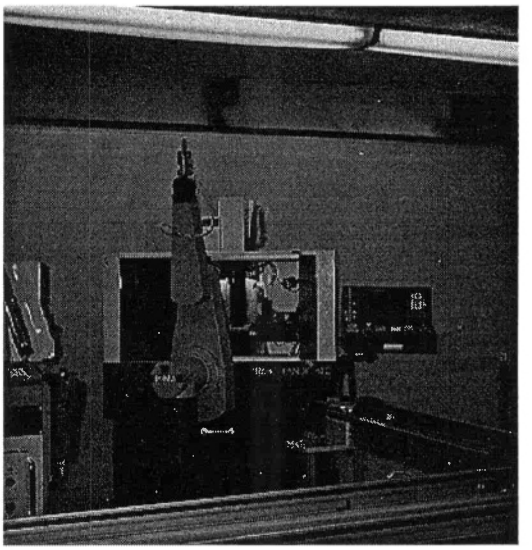


Fig. 2.7: Fotografía de la Fresadora

Son las máquinas herramientas que cuentan con computadoras para control numérico (CNC) para poder responder al control automatizado de la celda (Fig. 2.6, Fig. 2.7). Estas máquinas son las que realizan el trabajo final sobre las piezas que les provean los robots.

2.1.6 Control e interfaz

La máquina central que controla a la celda tiene integrada una tarjeta Artic [9] por medio de la cual se tiene la interfaz hacia todos los elementos de la celda, mismos que cuentan con una computadora individual para su funcionamiento. Cada uno de éstos soporta la misma interfaz RS 232 para su comunicación.
Las características de comunicación con las que trabaja cada elemento son:

Elemento	Bauds	Paridad	Datos
AS/RS y Júpiter	2400	I	8
Puma y Mitsubishi	9600	P	7

El CIM, en cada uno de sus componentes cuenta con dispositivos de control de calidad como:

- Un sistema de visión integrado por dos cámaras de vídeo y un sistema analizador de imágenes.
- Sistemas de seguridad para su programación y manejo.

La tarjeta Artic maneja la interfaz con todos los elementos de la celda de manera centralizada, lo cual no es recomendable pues no hay tolerancia a fallas. Además, al tener conexiones predeterminadas, un cambio en la configuración resulta difícil, restándole así flexibilidad al sistema. Es una de las razones por la cual en [4] se propuso la creación de interfaces independientes y reconfigurables para el controlador de una celda.

2.2 Computadora Personal (PC)

El transputer necesita de una PC huésped para su funcionamiento. Ésta no requiere de gran capacidad para dar un servicio básico pues todo el procesamiento

es realizado por los transputers. La PC sobre la cual trabajamos tiene las siguientes características:Procesador Intel 80486 a 60 Mhz. Dx

- Un monitor superVGA
- Un Disco Duro con capacidad 500 Mbytes
- 8 Mbytes en RAM

2.3 Tarjeta principal y Transputer

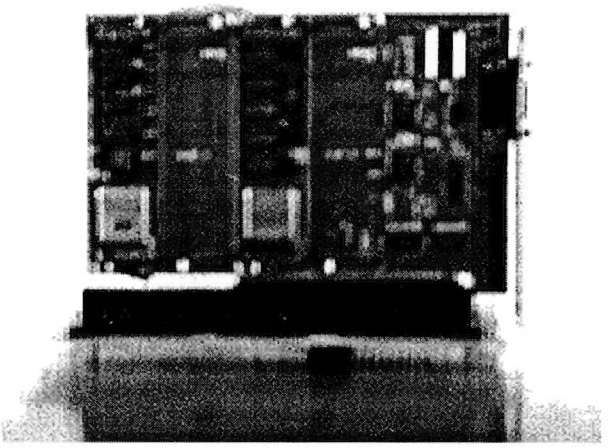


Fig. 2.8: Fotografía de la tarjeta Principal SMT004A y el TRAM

La tarjeta principal (Fig. 2.8) donde se colocan los transputers es un modelo SMT004A [10] fabricado por Semiconductors Technology. Se trata de una tarjeta de bajo costo diseñada para insertar hasta 4 módulos TRAM dentro de una computadora IBM PC/AT, PC/XT o compatible, ocupando un slot de expansión de ésta. La interfaz de la tarjeta permite que la PC se comunique con la red de transputers.

La tarjeta que corresponde al TRAM usado es la IMS B426, la cual incorpora un IMS T805 y 4 Mbytes de RAM dinámica ligada al transputer. Un TRAM integra un procesador, memoria y funciones periféricas, lo que da flexibilidad a los sistemas basados en transputers.

Los TRAM´s pueden conectarse entre sí de diferentes maneras: anillo, jerárquica, pipeline, etc. (Fig. 2.9). La tarjeta principal SMT004A tiene por default dos enlaces de

cada transputer dedicados a una conexión tipo pipeline. La topología de la tarjeta incorpora conmutadores (Fig. 2.10), los cuales pueden concentrar los enlaces de varios transputers para comunicarlos entre sí. En el controlador con el que trabajaremos (Fig. 2.10 c) tenemos conmutadores 2 x 2 que permiten una configuración dinámica entre los transputers y los elementos de la celda (robots, máquinas). Podemos observar que, por características de la misma tarjeta con la que se cuenta, los transputers siguen conectados en forma pipeline, por lo que hay redundancia en hardware y caminos alternos entre procesadores que dan facilidades para poder implementar tolerancia a fallas.

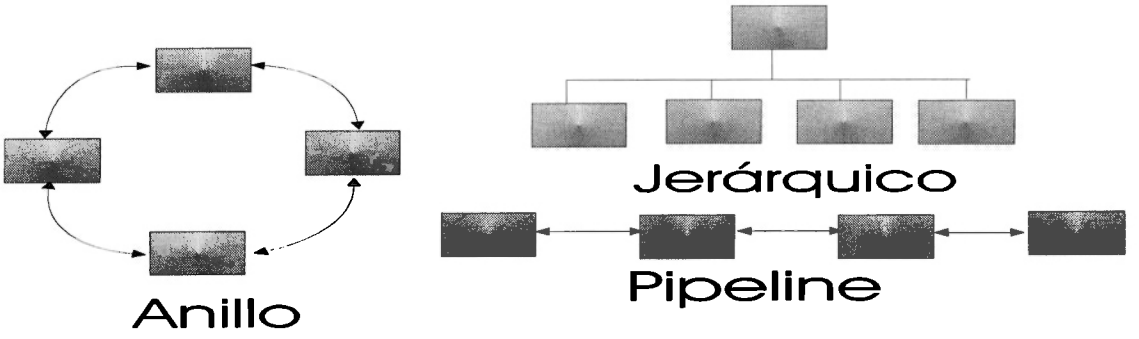


Fig. 2.9: Topologías de conexión

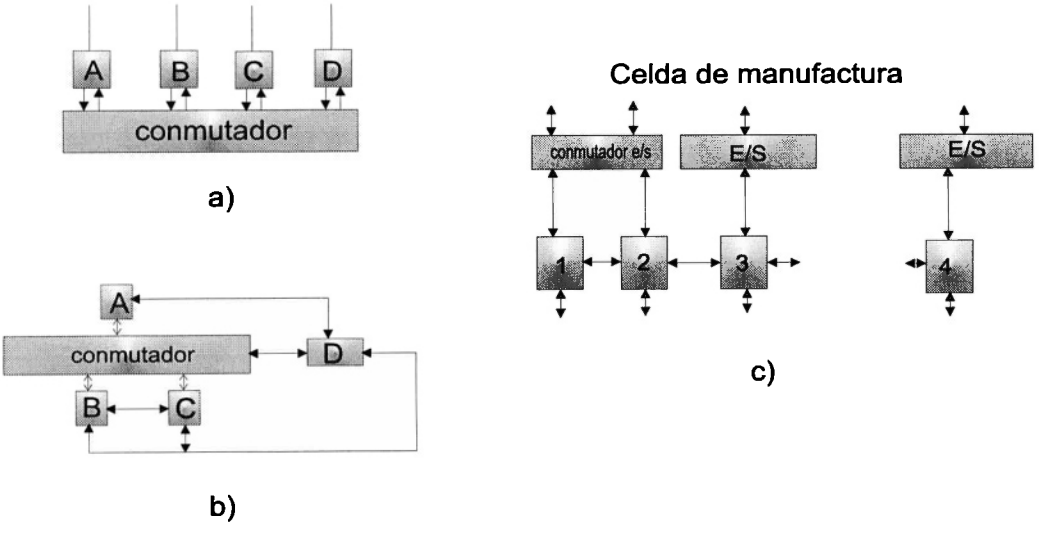


Fig. 2.10: Arquitecturas Dinámicas usando conmutadores

2.4 Sistema de desarrollo para Transputers en base al lenguaje OCCAM

El software incluido en la adquisición del transputer cuenta con un conjunto de herramientas para el desarrollo de programas en OCCAM sobre transputers [6]. Estos programas corren bajo el sistema operativo de la PC y los archivos que usan son de formato ASCII estándar. Todas estas herramientas pueden usarse en conjunto con las existentes en el mercado para la edición de archivos.

La implementación de componentes del programa paralelo es realizada en módulos separados de OCCAM. Estos se compilan en forma independiente y después se ligan para producir un código ejecutable.

Para programas que se van a ejecutar sobre una red de transputers, los procesos son colocados en cada transputer y los canales externos son mapeados [7] a los enlaces de comunicación por medio de un **archivo de configuración** como se muestra a continuación (ejemplo para dos procesadores).

```
#INCLUDE "hostio.inc"
#INCLUDE "protoc1.inc"
#INCLUDE "linkaddr.inc"
#USE "aplicacion.c8h"
#USE "enlace.c8h"
CHAN OF SP from.host, to.host :
CHAN OF ENTERO enlace.a.aplic, al.robot :
PLACED PAR
        PROCESSOR 0 T800
                PLACE from.host AT link0.in:
                PLACE to.host AT link0.out :
                PLACE aplic.a.enlace AT link1.out :
                aplicacion (from.host, to.host, aplic.a.enlace)
        PROCESSOR 1 T800
                PLACE aplic.a.enlace AT link0.in :
                PLACE al.robot AT link1.out :
                enlace (aplic.a.enlace, al.robot)
```

Tabla 2-1: Tabla de las herramientas de OCCAM

Programa	Descripción
iboot	Bootstrap tool. Programa que carga el código de la PC al transputer
icheck	Occam 2 syntax checker. Verifica la sintaxis de los programas en Occam en una forma más amigable que el mismo compilador. Da un poco más de indicaciones en los errores encontrados.
Iconf	Configurer. Crea código ejecutable para múltiples transputers
idebug	Toolset debugger. Permite depurar a nivel de símbolos y de ensamble
idump	Memory dumper. Para almacenar el contenido del transputer principal. Se usa al depurar programas que se están ejecutando en el transputer principal
ilibr	Biblioteca. Construye bibliotecas de código compilado
ilink	Linker. Crea referencias externas y liga código compilado por separado en un sólo archivo.
Ilist	Binary lister. Despliega información a nivel código fuente y código objeto.
Imakef	Makefile generator. Genera archivos Makefile para la construcción de código objeto y ejecutable. También crea bibliotecas de archivos muy usados.
Iserver	Host file server. Carga programas a los transputers y provee comunicación en el momento de la ejecución de programas con la computadora huésped
isim	Simulador para el transputer T414.
Iskip	Skip loader tool. Prepara redes de transputers para ejecutar programas sin el uso de un transputer principal o raíz.
Occam	Compilador occam. Compila código fuente para los transputers modelos IMS T212, M212, T222, T414, TT425, T800 y T805

En este ejemplo colocamos los procesos aplicación y enlace sobre dos transputers, así como los canales de comunicación necesarios para la comunicación entre ellos y la PC, como vemos en la Fig. 2.11.

Esta configuración así como el código para cada transputer es procesada por la herramienta de configuración **iconf** [6].

En la Tabla 2-1 se presenta una lista de las herramientas que existen para trabajar con la red de transputers.

En el siguiente capítulo describiremos más a detalle la arquitectura del transputer y la manera de programarlo. Posteriormente, en el capítulo 4, describiremos una herramienta más de software que utilizamos para verificar propiedades de los protocolos propuestos en esta tesis.

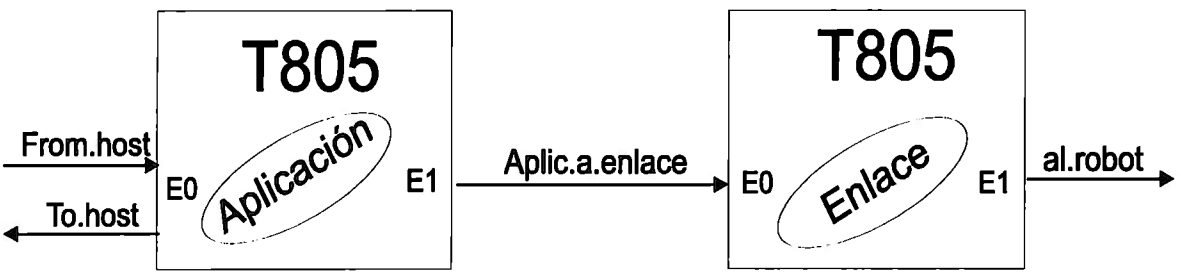


Fig. 2.11: Configuración de un programa en OCCAM

3. ARQUITECTURAS CON TRANSPUTERS Y SU LENGUAJE OCCAM

Los miembros de la familia de transputers -en particular, el transputer T805 utilizado- contienen generalmente un procesador (CPU) [11], una unidad de punto flotante (FPU), memoria (RAM) y un sistema de comunicación conectados por un bus de 32 bits. Este bus también se conecta a una interfaz de memoria externa para ampliar la memoria local, en caso de ser necesario.

Los transputers pueden ser programados en lenguajes de alto nivel como C, Pascal y Fortran; sin embargo es recomendado el uso de OCCAM, que es el lenguaje nativo de los transputers y provee la infraestructura necesaria para desarrollar sistemas concurrentes. El diseño de sistemas se facilita por la relación existente entre OCCAM y la arquitectura del transputer [8], pues este último fue diseñado teniendo en cuenta la filosofía de OCCAM.

La Fig. 3.1 muestra la forma en que los bloques más importantes de un transputer están interconectados.

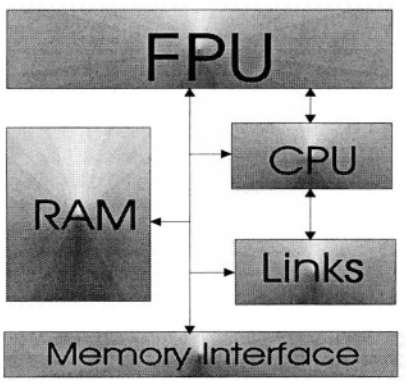


Fig. 3.1: Conexión de los bloques más importantes del transputer

3.1 Arquitectura del Transputer

El alto desempeño del transputer tiene su fundamento en la capacidad de trabajo concurrente entre su procesador central y su unidad de punto flotante (FPU). Ellos son capaces de ejecutar 10 Mips y 1.5 Mflops respectivamente, a una frecuencia de reloj de 20 Mhz [8].

3.1.1 FPU

Cada transputer cuenta con una unidad de punto flotante de 64 bits pequeña y *de* muy alto desempeño, a diferencia de los co-procesadores convencionales, los cuales normalmente ocupan un tamaño mayor que el del mismo procesador.

La *FPU permite* operaciones de longitud simple y doble de acuerdo al estándar ANSI-IEEE 754-1985 para aritmética de punto flotante [11].

3.1.2 CPU

El CPU del transputer contiene tres registros (A, B y C) usados para aritmética de enteros y de direcciones, mismos que están configurados en forma de pila [8].

Resultados sobre la eficiencia del CPU muestran que cerca de un 70% de las instrucciones ejecutadas son codificadas en un simple byte, por lo que muchas de estas instrucciones requieren de un solo ciclo del procesador. Ésto se debe a la filosofía RISC con que fue fabricado el procesador.

La representación de instrucciones es más compacta que en otros lenguajes de alto nivel. Dado que un programa requiere menos capacidad de almacenamiento, menos ancho de banda de memoria es requerido para cada ciclo de búsqueda de

instrucciones. Más aún, como los accesos a memoria son por palabras, el procesador puede recibir varias instrucciones en cada ciclo de búsqueda.

3.1.3 Memoria

El transputer tiene 4 Kbytes de memoria rápida estática interna [8] en la que puede almacenar programas y datos. Esta memoria es de acceso rápido pues sólo toma un ciclo de reloj. En caso de requerir una gran capacidad en memoria RAM, el procesador tiene acceso a 4 Gbytes o 64 Kbytes de memoria externa por medio de la interfaz de memoria externa EMI (External Memory Interface). Ambas memorias son parte del mismo espacio lineal de direcciones.

3.1.4 Enlaces de comunicación

La comunicación entre procesadores y hacia el exterior se realiza a través de cuatro enlaces de comunicación serial y bidireccional. Cada enlace está formado por un canal de entrada y otro de salida. Se puede implementar un enlace entre dos transputers conectando la interfaz de enlaces de un transputer con la interfaz de otro [8]. La velocidad de los enlaces puede ser de 5, 10 ó 20 Mbits/s.

3.2 Funcionamiento del software en OCCAM

OCCAM permite la ejecución de cualquier número de procesos en paralelo o secuencialmente.

El procesamiento paralelo en el transputer será eficiente si existe un buen desempeño en la comunicación entre procesos.

OCCAM cuenta con una serie de instrucciones propias del transputer y lenguajes paralelos [6] para:

- comunicación con la computadora huésped (capa de aplicación)
- control de flujo en los procesos
- manipulación de cadenas y números
- creación de procesos y ejecución de los mismos
- comunicación entre procesos
- etc.

3.2.1 Bibliotecas en OCCAM

OCCAM cuenta con varias bibliotecas que proveen funciones estándares matemáticas, procedimientos de administración de la comunicación con la computadora huésped y de los archivos, funciones para detección y corrección de errores en comunicación, etc. La Tabla 3-1 muestra una clasificación general de las bibliotecas según [6].

3.2.2 Pasos a seguir para compilar y ejecutar un programa en OCCAM

En OCCAM pueden crearse tantos procesos como sean necesarios. Estos pueden quedar integrados bajo un mismo o diferentes archivos [7].

Si el programa es un simple archivo que se va a ejecutar en un solo transputer, se tiene que:

1. compilar con la herramienta **occam** para después
2. ejecutarlo con la herramienta **iserver.**

Al tener varios archivos a ejecutar sobre una red de transputers, las herramientas de OCCAM se usan de la siguiente forma:

1.usar el compilador **occam** para cada módulo,

2.crear un archivo de configuración para determinar la forma de ejecución de los procesos y su comunicación,

3.usar la herramienta **ilink** para unir todos los módulos,

4.para que los programas puedan ejecutarse en una red de transputers se usa también la herramienta **imakef,** la cual agrega código bootstrap al programa ejecutable,

5.para ejecutar el código final se usa la herramienta **iserver.**

Tabla 3-1: Tabla de las bibliotecas de OCCAM

Biblioteca	Descripción
Bibliotecas del compilador	Aritmética de enteros de longitud múltiple Funciones de punto flotante Funciones aritméticas de 32 bits bajo los estándares IEEE Funciones aritméticas de 64 bits bajo los estándares IEEE Manipulación de bits y CRC Biblioteca de instrucciones aritméticas
snglmath.lib	Funciones matemáticas de longitud simple
dblmath.lib	Funciones matemáticas de longitud doble
tbmmaths.lib	Funciones matemáticas de optimización para transputers modelos T414/425
hostio.lib	Biblioteca de servicio con la computadora huésped
streamio.lib	Biblioteca de entrada y salida
string.lib	Biblioteca de manejo de strings
convert.lib	Biblioteca para conversión de tipos
crc.lib	Biblioteca para manejo de CRC
xlink.lib	Biblioteca para soporte extraordinario de los links
process.lib	Biblioteca para el soporte de procesos

3.2.3 Manejo de secuencialidad y paralelismo en OCCAM

OCCAM cuenta con una serie de instrucciones y bibliotecas que le dan una gran facilidad para expresar el paralelismo. OCCAM permite la ejecución concurrente de procesos por medio de una instrucción llamada **PAR** [7]. El bloque de instrucciones que se quiera ejecutar en paralelo se coloca bajo el mismo título PAR, por ejemplo:

```
PAR
    y := 0
    x := 20
    c := 100
```

En caso de querer realizar operaciones en forma secuencial lo único que debe cambiar es la instrucción PAR por **SEQ** [7], por ejemplo:

```
SEQ
    x := 1
    y := x * 15
    z := x + y
```

Estas instrucciones pueden ser anidadas y mezcladas de diferentes formas dependiendo de los requerimientos del programa.

La instrucción ALT es muy útil cuando se esperan datos de varios canales a la vez y/o se quiere prevenir el caso de no recibir dato alguno.

Su significado es similar al de la istrucción CASE implementada en diversos lenguajes de programaciónal como C: Sólo uno de los conjuntos de instrucciones en el ALT es ejecutada, y sólo es elegible si su condición es verdadera. Por ejemplo, la siguiente instrucción espera a que se reciba un dato por el canal uno o dos. El valor recibido determina el valor enviado a la salida.

```
ALT
    canal.uno ? x
        salida ! x
    canal.dos ? y
        salida ! y
```

3.2.4 Concurrencia en OCCAM

Cuando un proceso (secuencia de instrucciones) inicia, realiza un conjunto de acciones y entonces termina o se detiene. Un transputer puede ejecutar varios procesos en paralelo (concurrentemente). A cada proceso se le puede asignar alta o baja prioridad para su ejecución en caso de haber varios procesos esperando para su ejecución [8].

El transputer tiene un secuenciador en microcódigo para procesos en ejecución concurrente, lo que elimina la necesidad de algún sistema operativo que administre y despache estos procesos.

Los estados en los que puede estar un proceso son:

- Activo
  - ⇒ En ejecución o,
  - ⇒ en lista de espera para ser ejecutado.

- Inactivo
  - ⇒ Listo para iniciar o,
  - ⇒ listo para terminar o,
  - ⇒ esperando un tiempo específico.

El secuenciador trabaja de tal forma que ninguno de los procesos inactivos consumen tiempo del procesador. Los procesos activos en lista de espera están divididos en los de alta y los de baja prioridad.

3.2.5 Comunicación entre transputers

La comunicación es una parte esencial de los programas en OCCAM. Los valores son intercambiados entre los procesos concurrentes a través de **canales de**

comunicación [8]. Los canales proveen comunicación punto a punto unidireccional y sin buffer entre dos procesos concurrentes, ya sea que se encuentren en el mismo transputer o en diferentes.

Existen canales de comunicación internos (entre procesos de un mismo transputer) y canales externos (entre procesos en diferentes transputers).

El procesador cuenta con operaciones que permiten el intercambio de mensajes, entre las cuales las más importantes son:

⇒ Recepción de mensajes (?)

⇒ Envío de mensajes (!)

La Fig. 3.2 muestra gráficamente la comunicación de procesos por medio de canales.

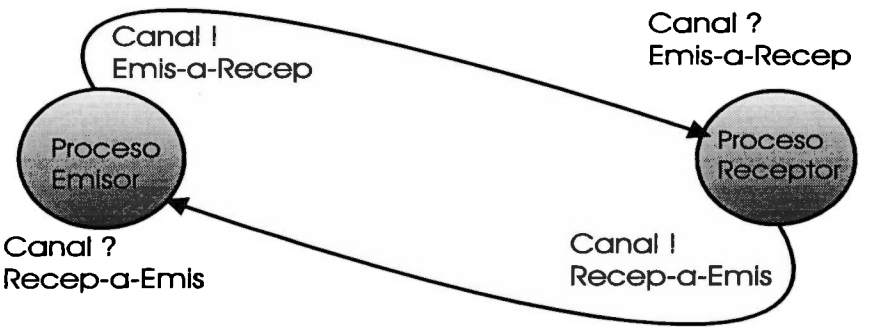


Fig. 3.2: Comunicación entre procesos.

Por ejemplo, para enviar un número del proceso emisor al receptor se puede realizar de la siguiente forma:

```
Proceso Emisor (CHAN OF NUMERO emisor.a.receptor)
  SEQ
    emisor.a.receptor ! 100
```

```
Proceso Receptor (CHAN OF NUMERO emisor.a.receptor)
INT    numero :
 SEQ
  emisor.a.receptor ? numero
```

Las instrucciones de recepción y envío de mensajes usan la dirección del canal para determinar si el canal es interno o externo, por lo que puede usarse la misma instrucción independientemente del tipo de canal [8]. Ésta es la razón por la cual existe la propiedad de reconfigurabilidad en OCCAM: un programa puede escribirse y compilarse sin preocuparse por la ubicación de los canales o procesos.

La comunicación entre procesos sigue el modelo **rendezvous** en la cual sólo pueden pasar datos sin buffer de almacenamiento. El proceso que quiere enviar datos espera hasta que esté listo el que recibe, o éste último espera hasta que tenga datos el que envía. El que no haya almacenamiento temporal permite establecer una sincronización en el intercambio de información.

La comunicación entre procesos se realiza colocando en la memoria de un canal los siguientes datos:

⇒ Apuntador al mensaje.

⇒ Dirección del canal.

⇒ Un contador de los bytes que se van a transmitir o recibir.

Una vez hecho esto el hardware realiza la operación de recepción o salida del mensaje a través del canal.

3.2.5.1 Comunicación entre procesos en diferentes transputers

Un canal entre dos procesos en diferentes transputers es implementado por medio de enlaces de conexión punto a punto, a velocidades de 20 Mbps, 10 Mpbs ó 5 Mbps, dependiendo de la configuración de hardware. Para esta tesis, se trabajó con transputers cuyos enlaces tienen una velocidad de 20 Mbps [8].

Para transmitir mensajes vía enlaces, el procesador delega este trabajo a una interfaz autónoma que los administra. Los procesos pasan a una lista de espera mientras la interfaz no termine su trabajo. Al finalizar el envío o recepción de mensajes, el proceso sale de la lista de espera y continúa su ejecución. La ventaja de este mecanismo es que pueden estar en comunicación ciertos procesos por medio de la interfaz de enlaces, mientras otros se encuentran activos y en ejecución. Lo anterior permite traslapar tiempo de ejecución del CPU con el tiempo de comunicación.

Cada interfaz de enlace usa tres registros:

- Apuntador al área de trabajo del proceso.
- Apuntador al mensaje.
- Contador de bytes del mensaje.

En la Fig. 3.3 se muestran el proceso **emisor** y el **receptor** que se encuentran en diferentes transputers y se comunican por medio de un canal implementado por un enlace serial.

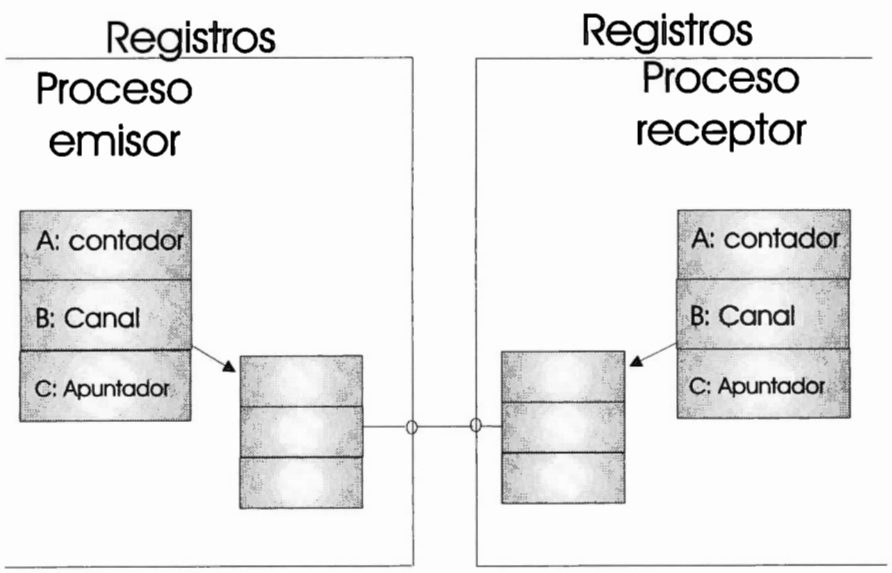


Fig. 3.3: Procesos emisor y receptor en diferentes transputers

Cuando el **emisor** quiere enviar un mensaje, los registros de la interfaz del enlace correspondiente al transputer que ejecuta este proceso, adquieren los valores de

dirección y número de bytes de dicho mensaje. Y de manera similar, cuando el **receptor** recibe el mensaje, los registros de la interfaz del enlace adquieren los valores del otro transputer. El protocolo usado por los dos procesos asegura que no importa quién solicita primero el servicio (receptor o emisor), pues de cualquier modo, el primero en llegar al "rendezvouz" coloca su interfaz de enlace en espera hasta que pueda ser atendida por su interfaz complemento en otro transputer.

3.2.6 Redes de transputers

Al conectar dos o más transputers en una tarjeta principal se pueden conformar diferentes arquitecturas, como vimos en la sección 2.3. En esta tesis se usará como base la arquitectura desarrollada por Alarcón [4] en la que los transputers son conectados de la manera más conveniente a la aplicación, a través de un conmutador.

Un canal entre diferentes transputers (implementado con un enlace) tiene instrucciones de operación iguales a las de canales internos en un mismo transputer, es decir que la ubicación de los canales es invisible para el protocolo de comunicación desarrollado.

El único caso importante que hay que considerar es cuando los procesos se encuentran en transputers que no están conectados directamente por un enlace. En ese caso es necesario implementar un proceso ruteador.

3.2.7 Control de errores en OCCAM

Los errores en programas de OCCAM son detectados por el compilador o pueden ser manejados durante el tiempo de ejecución en tres formas [6]:

1. Pueden causar que el proceso se detenga para que otro continúe.
2. Puede causar que se detenga todo el sistema.
3. Pueden tener un efecto arbitrario e indefinido.

El método 1 es el modo preferido para la ejecución de un programa terminado. El método 2 es usual durante el desarrollo de un sistema, mientras que el método 3 sólo es usado en programas que se sabe son correctos

Cuando un proceso para por algún error, el sistema de desarrollo de OCCAM permite realizar un análisis del estado del sistema.

Para desarrollar los protocolos en esta tesis se tomaron en cuenta estos métodos decidiéndonos por usar el segundo, el cual detiene la ejecución de todos los procesos al momento de encontrar un error que pueda ocasionar problemas posteriores. Este método saca de ejecución procesos que no hayan parado en forma natural al encontrar un error, desocupando así la red de transputers.

El método uno será usado cuando la tolerancia a fallas sea implementada en el controlador [1].

4. DISEÑO Y VALIDACIÓN DE PROTOCOLOS PARA CELDAS FLEXIBLES DE MANUFACTURA

El controlador de la CFM requiere de flexibilidad para permitir la reconfigurabilidad y tolerancia a fallas [2]. El sistema incluirá rutinas "inteligentes" en todas las capas de procesamiento para poder implementar esta flexibilidad.

Ésto implica que los requerimientos de procesamiento del controlador son grandes y la necesidad de respuestas rápidas sólo puede ser satisfecha por procesadores con las capacidades de procesamiento y comunicación con las que cuentan los transputers.

El protocolo que se presenta en esta tesis provee un servicio de comunicación rápido y confiable entre los elementos de la celda a diferentes niveles. La mayor parte de este protocolo se desarrolló en lenguaje OCCAM, pero es importante también conocer el funcionamiento de la capa física tanto en el transputer como en el controlador.

En este capítulo describiremos los requerimientos de comunicación de nuestra aplicación y las características de los protocolos con que contamos (capa física) en el transputer y el hardware del controlador. Hablaremos también de las herramientas que usamos para validar nuestro protocolo.

4.1 Qué es un protocolo de comunicación

"Un protocolo de comunicación es un conjunto de reglas que gobiernan las interacciones y la coordinación entre entidades en un sistema distribuido y redes de computadoras. Por lo tanto derivar protocolos libres de errores es crucial para asegurar que son confiables" [23].

Un protocolo involucra las tres partes siguientes:

•Define el formato preciso para los mensajes válidos (sintaxis).

•Define las reglas de procedimiento para el intercambio de datos (gramática).

•Y define un vocabulario de mensajes válidos que pueden ser intercambiados, así como su significado (semántica).

El protocolo desarrollado, la arquitectura y los componentes involucrados en conjunto, aseguran que se establezca una comunicación eficiente. En el caso de Celdas Flexibles de Manufactura se requiere que el sistema de comunicación sea confiable, correcto, seguro, en tiempo real, que permita una rápida sincronización entre sus elementos y que tenga un control concurrente y distribuido.

Nuestro protocolo incluye dentro de su especificación la detección de errores de comunicación, lo cual asegura la **confiabilidad**. En caso de haber errores, tiene mecanismos de reenvío de datos, por lo cual el sistema tiene la característica de **corrección**.

El protocolo permite atender en **tiempo real** tanto al huésped como a los elementos de la celda, respondiendo a sus peticiones de **comunicación** en ambos sentidos.

Cada módulo del sistema se encuentra **sincronizado** con el resto de los módulos concurrentes por medio de sus canales de comunicación. Los módulos se comunican entre sí para llevar el **control** de la celda, pero al mismo tiempo tienen la capacidad de detectar cuando algo no funciona para tomar las acciones pertinentes. Los módulos fueron diseñados para poder ejecutarlos en diferentes transputers y tener un **control distribuido**.

4.2 Niveles de un protocolo

En 1980 la Organización Internacional de Estándares (OSI), reconoció las ventajas de estandarizar jerárquicamente los servicios de un protocolo de comunicación y ponerlo a disposición de los diseñadores como un modelo de referencia. Las recomendaciones de la OSI definen siete niveles [12] como se muestra en la Fig. 4.1.

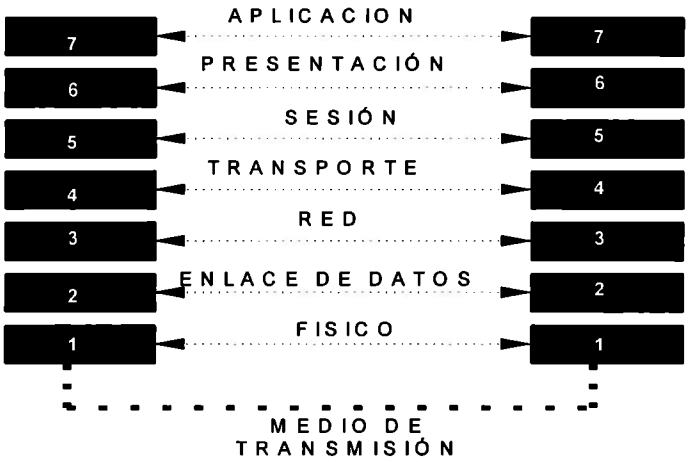


Fig. 4.1: . Modelo de referencia OSI para la interconexión de sistemas abiertos

1. Capa Física: Transmisión física de bits por un circuito físico.
2. Capa de Enlace de Datos: Detección y recuperación de errores en enlaces.
3. Capa de Red: Transferencia y ruteo de datos.
4. Capa de Transporte: Transferencia de datos usuario - usuario (alto nivel).
5. Capa de Sesión: Coordinación de sesiones del usuario.
6. Capa de Presentación: Interpretación de la sintaxis a nivel de usuario para encriptación y compresión de datos.
7. Capa de Aplicación: Procesos de aplicación e interfaz con usuarios.

Cabe aclarar que para nuestro fines particulares no es necesario el desarrollo de todas las capas del protocolo, como se describirá en la sección 4.4.

4.3 Validación de Protocolos

4.3.1 Antecedentes

Diseñar un protocolo confiable no es una tarea fácil. Problemas como abrazos mortales, esperas infinitas, mensajes duplicados, etc. pueden presentarse si el protocolo no es validado [22].

En el capítulo 6 se da la definición de nuestro protocolo incluyendo la especificación del servicio, la cual se puede definir por niveles. Una especificación es válida en el momento en que las reglas de cada proceso son consistentes.

Para especificar y validar las reglas de procedimiento del protocolo se utiliza un lenguaje que sirve para describirlo parcialmente con un **modelo de validación**. Un modelo de validación define la intercomunicación entre procesos de un sistema distribuido pero no resuelve detalles de implementación, por ejemplo, no especifica cómo va a ser transmitido, codificado o almacenado un mensaje. PROMELA [12] es el lenguaje que nos permitirá construir este modelo para probar la consistencia y buena estructura del protocolo.

4.3.1.1 Prueba de Propiedades Globales en Sistemas Distribuidos

Para validar un protocolo es necesario especificar exactamente las características que lo hacen correcto (criterios de corrección). Algunos de estos criterios son: ausencia de abrazos mortales, de esperas infinitas y de terminaciones impropias, los cuales se detallarán en la sección 4.3.2.

Se sabe [12] que verificar las propiedades más simples de un protocolo, como la ausencia de abrazos mortales, es un problema tipo PSPACE. Sin embargo, existen metodologías que permiten, con un pequeño margen de error, validar protocolos en tiempo razonable. Estas metodologías:

1. Utilizan un formalismo que especifica claramente los requerimientos de corrección.

2. Cuentan con un método para reducir la complejidad de ciertos modelos para que puedan ser validados en tiempo razonable.

LOTOS y ESTELLE [24] se enfocan principalmente en la primera actividad. Promela se enfoca más en la segunda, aunque también permite especificar criterios de corrección organizados en varios niveles de complejidad independientes. El nivel más simple y frecuente de requerimientos incluye aquellos como la ausencia de abrazos mortales. Otro nivel un poco más complicado son aquellos como la ausencia de esperas infinitas, pues llevan implícito un gran costo computacional para validarlos.

Para probar los requerimientos de corrección en un sistema distribuido, es necesario probar propiedades globales, lo cual no siempre resulta evidente.

El problema general es llamado "Global Predicate Evaluation" (GPE) [41], y no es fácil de resolver, pues en sistemas paralelos y distribuidos, puede no haber cotas superiores para los tiempos de comunicación.

Un estado global obtenido por observaciones remotas puede ser obsoleto, incompleto o inconsistente (ningún observador "ideal" externo podría verlo), lo cual nos llevaría a evaluar inadecuadamente una propiedad global.

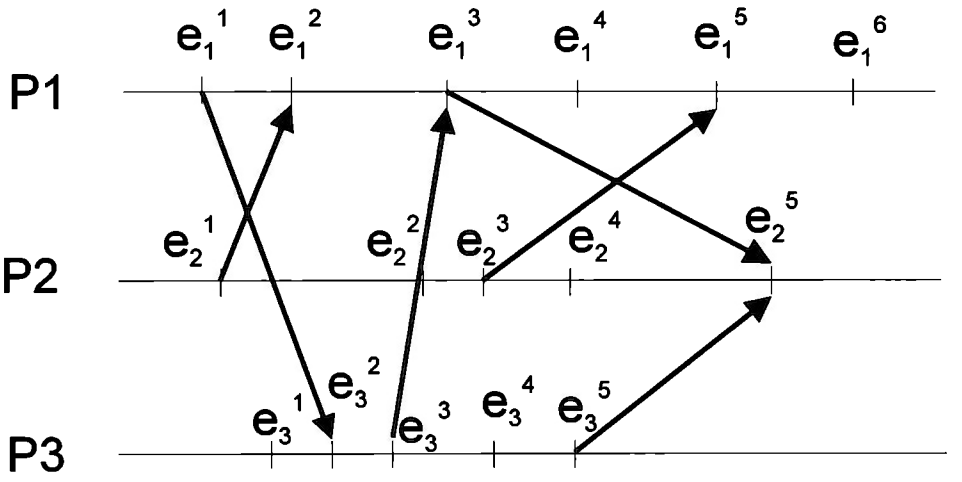


Fig. 4.2: Secuencia de eventos de los procesos P1, P2 y P3.

La historia global de un algoritmo distribuido es (Fig. 4.2):

$$H = h_1 \cup h_2 \cup \ldots \cup h_n$$

donde $h_i = e_i^1 \; e_i^2 \ldots e_i^m$ es la historia local (secuencia de eventos) de un proceso P_i.

Un algoritmo distribuido puede ser visto entonces como un conjunto ordenado de eventos (H, ->) donde -> es la relación de precedencia (orden parcial) entre eventos.

Las especificaciones de algoritmos distribuidos se dan con un orden parcial, pero la ejecución se da en orden total, por lo que puede haber varias corridas válidas diferentes para una misma especificación. En la Fig. 4.3, por ejemplo todos los caminos del estado global $e_1^0 e_2^0 e_3^0$ a $e_1^3 e_2^2 e_3^4$ representan corridas válidas del algoritmo.

Una secuencia de ejecución es un conjunto ordenado de estados posibles en la ejecución del modelo y puede ser: **terminal** o **cíclica.**

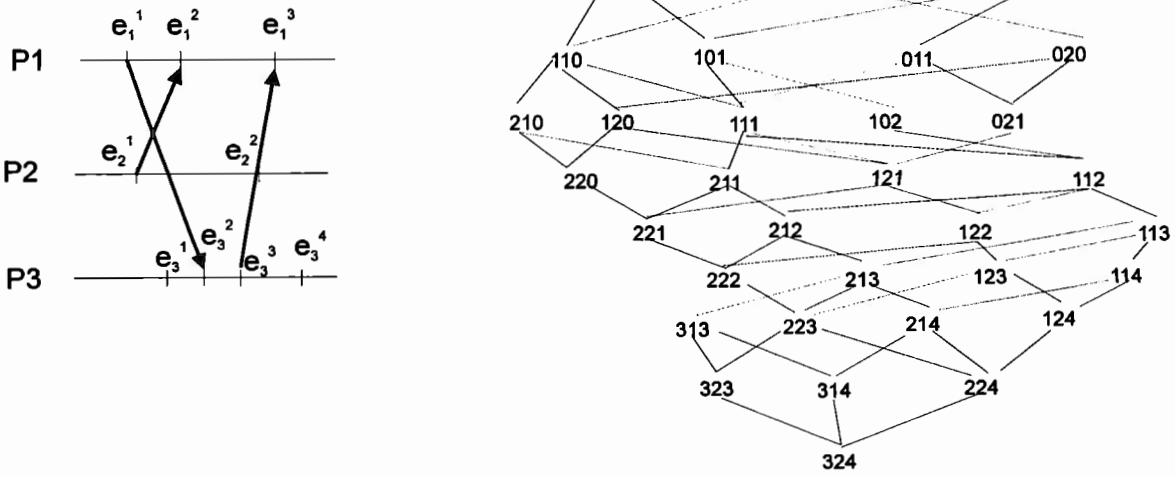


Fig. 4.3: Grafo de posibles secuencias de ejecución de los procesos P1, P2 y P3.

El número de secuencias de ejecución se vuelve exponencial aun para sistemas pequeños, por lo que validar un protocolo resulta difícil. En la sección 4.3.3 mencionaremos como afronta PROMELA este problema.

4.3.2 PROMELA: Criterios de corrección

En esta sección describiremos detalladamente las características de Promela que permiten validar criterios de corrección, mientras que en el apéndice L damos un breve resumen de su sintaxis. Para una descripción más completa puede consultarse a Holzmann [12].

El comportamiento de un modelo de validación [12] es el conjunto de todas las secuencias de ejecución del mismo. Un criterio de corrección se expresa en Promela como una restricción que indica que cierto comportamiento dentro del modelo es **inevitable** o **imposible**. Una es el complemento de la otra, por lo que en Promela sólo se utiliza la restricción que marca un comportamiento **imposible.** Por ejemplo, la siguiente porción de código:

never {do :: skip od -> P -> !Q }

indica que independientemente de la secuencia inicial de eventos, es imposible que para un estado en el que la propiedad P es verdadera, siga un estado en el cual la propiedad Q es falsa.

Entre los criterios de corrección y validación que Promela permite especificar podemos mencionar [42]:

- Aserciones.
- Invariantes del sistema.
- Ausencia de deadlocks (abrazo mortal).
- Ausencia de ciclos erróneos.
- Exigencias Temporales.

4.3.2.1.1 **Aserciones**

Las criterios de corrección para un modelo en Promela pueden expresarse como **proposiciones lógicas**, que pueden referirse a todos lo elementos del sistema en

cierto estado (variables, puntos de flujo, canales, etc.). Un ejemplo de proposición en Promela es una **aserción:**

assert { condición }

en la cual se afirma que la condición será siempre verdadera independientemente del orden de la secuencia de estados. Puede ser colocada en cualquier parte del modelo y si la condición es verdadera, este enunciado no tiene efecto alguno, si es falsa genera un error al violar la aserción. Por ejemplo:

assert { len (canalentrada) == 0 }

indica que el canal de entrada no debe de tener mensajes en espera en el momento en que el programa pasa por esta instrucción en cualquier secuencia de ejecución.

4.3.2.1.2 Invariantes del Sistema

La formalización de *invariantes del sistema* representa una aplicación más general del enunciado **assert**, ya que es una condición lógica que al ser verdadera en el estado inicial del sistema, permanece verdadera en todos los estados alcanzables del mismo sin importar la secuencia de estados seguidos.

Para expresar ésto, se coloca la invariante del sistema en un proceso monitor que es independiente a los demás módulos, y de ejecución concurrente a ellos y aleatoria. Por ejemplo:

proctype monitor ( ) { assert (invariante) }

4.3.2.1.3 Ausencia de deadlocks (abrazo mortal)

El estado final de una secuencia de ejecución terminal (aquélla en donde todos sus estados son diferentes) debe satisfacer los siguientes dos criterios para considerarse como un *estado terminal propio*:

1. Todos los procesos instanciados han terminado.
2. Todos los canales de comunicación están vacíos.

Sin embargo existen casos en que un proceso no necesariamente debe terminar, por ejemplo, si es un proceso servidor, por lo cual se debe identificar como un estado terminal propio. En Promela ésto se hace con el uso de etiquetas **end**. Por ejemplo:

```
proctype dijkstra ( ) {
end: do
        :: sema ! p -> sema ? v
od
}
```

4.3.2.1.4 Ausencia de ciclos erróneos

En la sección anterior hablamos de secuencias terminales; en ésta hablaremos sobre las *secuencias cíclicas*, las cuales son aquéllas en las que algunos de sus estados se repiten y deben cumplir con dos propiedades:

1. No existen ciclos infinitos conteniendo únicamente estados sin marcar.
2. No existen ciclos infinitos que incluyan estados marcados.

Para identificar ciclos sin progreso, es importante poder definir cuáles son los estados que denotan progreso en el sistema o protocolo y declarar los ciclos que los incluyen como *ciclos progresivos*. Por ejemplo:

```
proctype dijkstra ( ) {
end:    do
                :: sema ! p ->
progress:                       sema ? v
        od
}
```

Para identificar esperas infinitas, hay que formalizar los ciclos que no pueden suceder infinitamente usando la etiqueta **accept**, la cual marca estados que no pueden formar parte de ciclos que se repitan infinitamente. Por ejemplo:

```
proctype dijkstra ( ) {
end:    do
                :: sema ! p ->
accept:                         sema ? v
        od
}
```

En este ejemplo aseguramos que es imposible iterar sobre una secuencia de operaciones P y V, lo cual, en este caso particular, es falso.

4.3.2.1.5 Exigencias temporales

Las exigencias temporales definen ordenamientos en el tiempo de las propiedades de estados, para lo cual se usa la etiqueta **never**. Por ejemplo, para expresar que "Cada estado en el cual la propiedad P es verdadera, es seguido de un estado en el cual la propiedad Q es verdadera" se utilizaría el siguiente código:

```
never {
    do
        :: skip
    od -> P -> !Q
}
```

En donde estamos expresando que independientemente de la secuencia de eventos iniciales, es imposible que un estado en el que la propiedad P es verdadera, sea seguido por otro estado en el que la propiedad Q es falsa.

4.3.3 Validación automática de Protocolos

En [26] se menciona que los primeros validadores automáticos surgen a mediados de los años setenta. Entre los primeros en construir una de estas

herramientas está Jan Hajek en la Universidad Técnica Eindhoven , quien entre 1976 y 1978 desarrolló un verificador descrito en [27]. Aproximadamente al mismo tiempo, Colin West, en IBM, empezó a trabajar en la implementación de una herramienta que usaba una técnica de análisis llamada Matriz Duologue [28], la cual se usó para probar el protocolo CCITT X.21. Esta herramienta ayudó a demostrar por primera vez que las técnicas automatizadas de validación son factibles y pueden encontrar errores importantes en los protocolos [22].

Los primeros trabajos orientados a crear la herramienta que se usa en esta tesis (PROMELA) iniciaron en 1980 con el verificador **Pan**, el cual fue mejorado sucesivamente hasta llegar a **SPIN** [12] en 1989. Este verificador se basa en el paradigma **on-the-fly**: la verificación se realiza en un solo paso, colocando en memoria los datos necesarios sobre el modelo, para utilizarlos en la verificación de los criterios de corrección necesarios. Esta técnica ocupa poco espacio y tiempo, por lo que es la más usada en los verificadores actuales.

La versión 2.8 hace que SPIN sea portable a cualquier sistema que cuente con un compilador GNU de C, incluyendo sistemas PC corriendo bajo Windows95.

4.3.3.1 Análisis de Estados Alcanzables

PROMELA valida protocolos haciendo uso de la técnica llamada "análisis de estados alcanzables" [22].

La forma de trabajo de esta técnica consiste en contar con un estado inicial del protocolo. Se determinan iterativamente todos los estados alcanzables a partir del estado inicial y se analizan para verificar si en algún momento no se satisfacen los criterios de corrección.

El concepto de análisis de estados alcanzables fue manejado primeramente por Sunshine [37]. Más tarde C. H. West desarrolló una herramienta de validación automática basada en el análisis de estados alcanzables, misma que se aplicó por

primera vez al protocolo X.25, reportando los resultados a la CCITT en 1977. El primer reporte externo sobre esta herramienta [38] fue presentado en una conferencia en la que Bochmann reportó un análisis de estados alcanzables también sobre el protocolo X.25 [39]. Hajek [40] también reportó el uso de herramientas automatizadas usando esta filosofía.

El determinar sucesivamente todos los estados alcanzables implica crear un grafo como el de la Fig. 4.3, teniendo siempre cuidado en que el nuevo estado determinado no se repita con alguno anterior.

La base de datos utilizada para almacenar los datos del grafo puede variar dependiendo del tipo de recorrido que se haya elegido. Cada tipo de recorrido tiene sus ventajas en ahorro de espacio o tiempo, pero depende de las necesidades del diseño el determinar la forma de recorrer el grafo.

Los algoritmos que utilizan esta técnica pueden dividirse en tres categorías:

1. Búsqueda completa o exhaustiva (sistemas con hasta 10^5 estados).
2. Búsqueda parcial controlada o supertrace (sistemas con hasta 10^8 estados).
3. Simulación aleatoria (sistemas aun más grandes)

Normalmente se aplican en el orden en que se mencionaron, según la complejidad del sistema a validar.

4.3.3.1.1 Búsqueda completa o exhaustiva.

Este algoritmo realiza un análisis exhaustivo de todos los estados (alcanzables o no) del sistema haciendo una separación de éstos.

El problema principal consiste en la capacidad de memoria disponible así como en la velocidad de procesamiento.

4.3.3.1.2 Búsqueda Parcial Controlada o Supertrace

Si no es posible realizar el análisis anterior, se utiliza este algoritmo, basado en la premisa de que en la mayoría de los casos prácticos, el número máximo de estados que pueden ser analizados (A) es sólo una fracción del número total de los estados Alcanzables (R).

Esta búsqueda tiene los siguientes objetivos:

1. Analizar exactamente los estados alcanzables (A).
2. Seleccionar los estados A del conjunto completo de estados alcanzables R, de tal manera que la mayoría de las funcionalidades del protocolo sean probadas.
3. Seleccionar los estados A de manera que la probabilidad de una buena calidad de búsqueda (la manera de encontrar un error) sea mejor que la cobertura A/R.

La selección de los estados sucesores puede basarse en una heurística para favorecer ejecuciones que revelen, con una probabilidad alta, errores de diseño. Entre éstos métodos podemos mencionar: Limite de profundidad, Búsqueda dispersa, Orden Parcial y Selección aleatoria, mismos que se pueden consultar en [42].

4.3.3.1.3 Simulación Aleatoria.

Este algoritmo se utiliza para analizar diseños que fracasaron con la segunda técnica: el espacio de estados es tal que no puede seleccionarse el algoritmo de búsqueda parcial controlada. La búsqueda aleatoria es utilizada para sistemas con más de 10^8 estados; y explora el espacio de estados usando técnicas aleatorias que funcionan de manera independiente al tamaño y complejidad del sistema modelado (inclusive para sistemas de tamaño infinito).

El protocolo validado en esta tesis es de tamaño medio, por lo que las primeras dos técnicas fueron usadas. En el capítulo 6 describiremos los resultados de este análisis.

4.4 Características deseables de protocolos para CFM.

4.4.1 Antecedentes

En [24] se describe la aplicación de una técnica de descripción formal llamada ESTELLE que ayuda a especificar sistemas distribuidos y concurrentes permitiendo la creación del modelo de un protocolo para CFM. Los autores afirman que el protocolo debe permitir respuestas del sistema en tiempo real, y dividen el tráfico de mensajes entre los procesos de una red industrial en dos tipos:

1. Periódico: resultados de los sensores, actuadores, torno, fresadora, etc.

2. No periódico: alarmas, operaciones de administración entre los procesos, estadísticas, etc.

El primer tipo de mensajes es muy corto, y el segundo debe proveerse con poca frecuencia. El envío de mensajes se da generalmente a través de conexiones punto a punto (en el caso de transputers, implementadas con los enlaces).

En [14] Valenzano describe los estándares de comunicación TOP (Technical Office Protocol) y MAP (Manufacturing Automation Protocol) para la automatización de procesos en Celdas Flexibles de Manufactura, según el tipo de red:

1.Interconexión de mainframes, minicomputadoras, y estaciones de trabajo en una red que controle los procesos a grandes distancias.

2. Interconexión de celdas, cada una con su propio controlador, para la coordinación de diferentes procesos al mismo tiempo.

3. Interconexión de dispositivos como robots, máquinas de control numérico, bandas transportadoras, ... controlando un área de producción. El tráfico típico de

estas redes consiste generalmente en comandos para los elementos de la celda y para el control del proceso dentro del mismo controlador.

El tipo de red que nosotros estudiaremos es el tercero, para el cual Valenzano propone un servicio orientado a conexión, en donde cada mensaje enviado tiene como respuesta obligatoria un reconocimiento de recepción del mensaje.

En [20] Reza S. Raji expone el problema de tratar de controlar las operaciones de un proceso industrial por medio de un solo procesador. La falla de éste representaría la caída total del sistema. Por el contrario, al utilizar redes de control distribuidas se puede tener tolerancia a fallas.

En [25] se muestra que el concepto de paralelismo puede incrementar significativamente el desempeño de un sistema, lo cual puede ser bastante útil cuando se requieren respuestas en tiempo real. Este artículo habla en especial sobre un protocolo de la capa de red y muestra que planear las operaciones de esta capa con paralelismo permite un diseño modular y una ejecución eficiente.

4.4.2 Opciones seleccionadas

En nuestro controlador, las capas física, de enlace y de red [13] serán las responsables de transferir paquetes de datos entre dos estaciones de trabajo. En particular, la capa física incluye todos los mecanismos de hardware y de software requeridos para accesar el medio físico y para enviar y recibir bits de los elementos de la celda. La capa de enlace provee servicios de ensamble o desensamble y ayuda a detectar y recuperar errores de transmisión ocurridos entre procesadores adyacentes. Finalmente la capa de red transporta los mensajes entre procesos en transputers no adyacentes.

Tratándose de una red local, en la que los tiempos de respuesta deben ser pequeños, no se necesita toda la complejidad del modelo OSI. Por lo tanto, aparte de las 3 capas mencionadas, tendremos solamente la de aplicación (Fig. 4.4), la cual será implementada en este trabajo de manera muy básica.

En nuestro protocolo se resuelven los requerimientos de respuesta en tiempo real tomando en cuenta la naturaleza propia de nuestro problema. Primeramente, la arquitectura de control diseñada en [4] servirá para controlar celdas de manufactura que no sobrepasan una decena de metros. Por otro lado, dado que el controlador se implementa en una tarjeta para PC, la comunicación entre transputers será en una red de área muy pequeña.

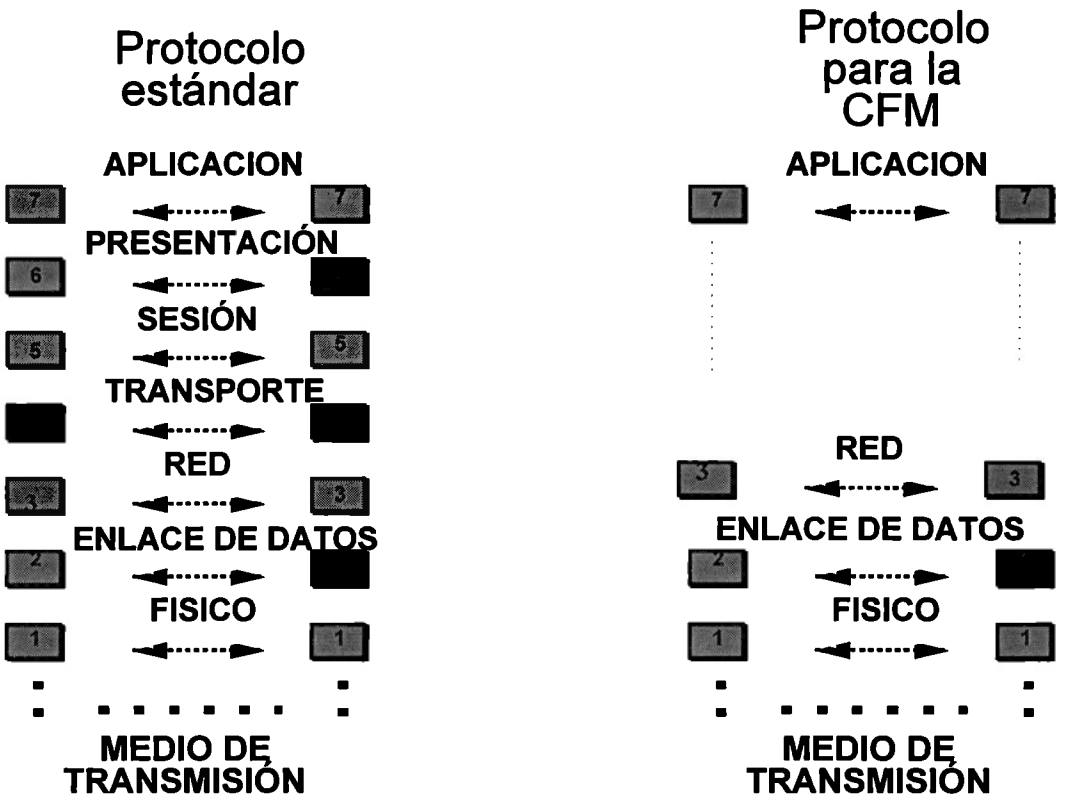


Fig. 4.4: Niveles de OSI aplicados a CFM

Dado que en nuestro sistema las conexiones son dedicadas, el emisor y receptor se conocen automáticamente, por lo cual no es necesario colocar en los mensajes un encabezado que incluya direcciones, pues sólo ocuparía más espacio y tiempo de transmisión.

En el capítulo siguiente describiremos la capa física del controlador para CFM que utilizaremos [4], para describir posteriormente, en el capítulo 6, el protocolo desarrollado.

5. PROTOCOLO DE COMUNICACIÓN Y FUNCIONAMIENTO DE LA CAPA FÍSICA Y DEL CONTROLADOR

La arquitectura del controlador de CFM diseñada por Alarcón [4] se muestra en la Fig. 5.1. Esta arquitectura esta formada por un conjunto de procesadores con interfaces seriales en los elementos de la celda.

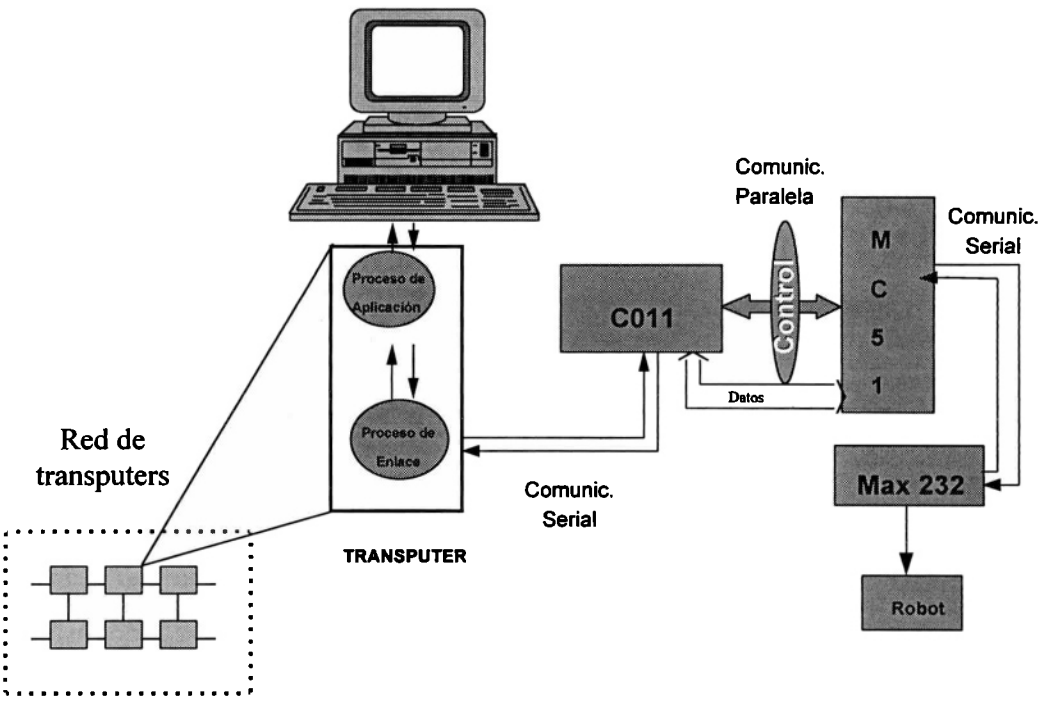


Fig. 5.1: Arquitectura del Controlador

En este capítulo describiremos las características de comunicación dadas por esta arquitectura, que servirán como base para el desarrollo del protocolo que mostraremos en el capítulo 6.

5.1 Protocolos de comunicación del transputer

El protocolo de comunicación entre procesos que corren sobre transputers (ver Fig. 5.2) varía dependiendo de la ubicación de los procesos, los cuales pueden estar en ejecución en el mismo transputer o en varios. Independientemente de la ubicación de los procesos, podemos observar en la Fig. 5.2 que existen dos canales dedicados a la comunicación entre procesos, un canal para envío y otro para recepción.

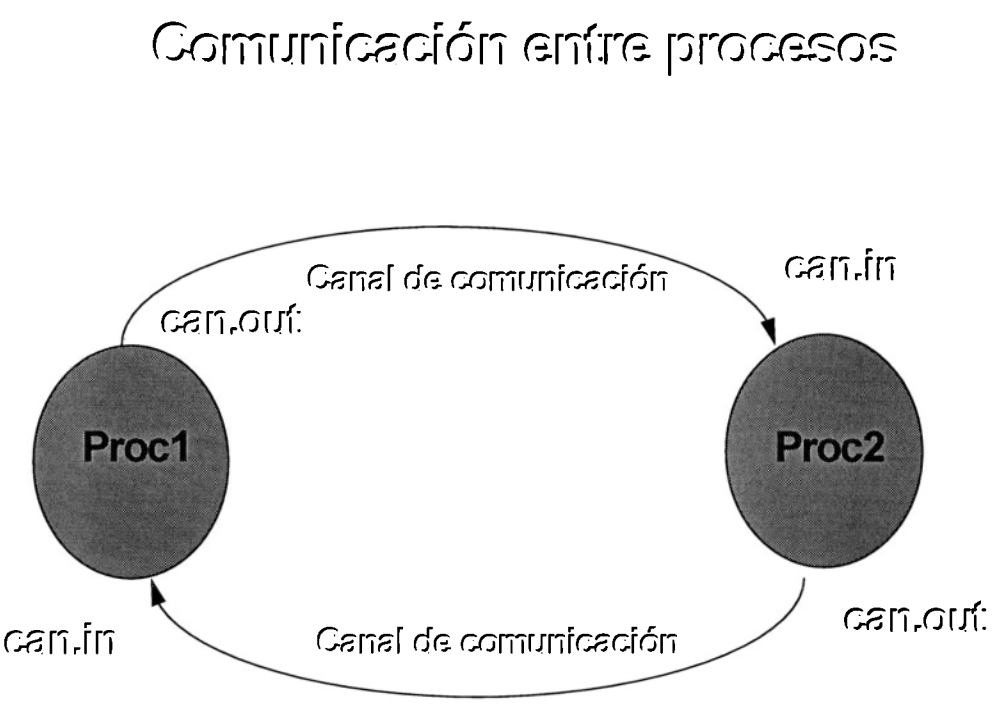


Fig. 5.2: Forma de comunicarse entre procesos

5.1.1 Protocolo entre procesos en el mismo transputer

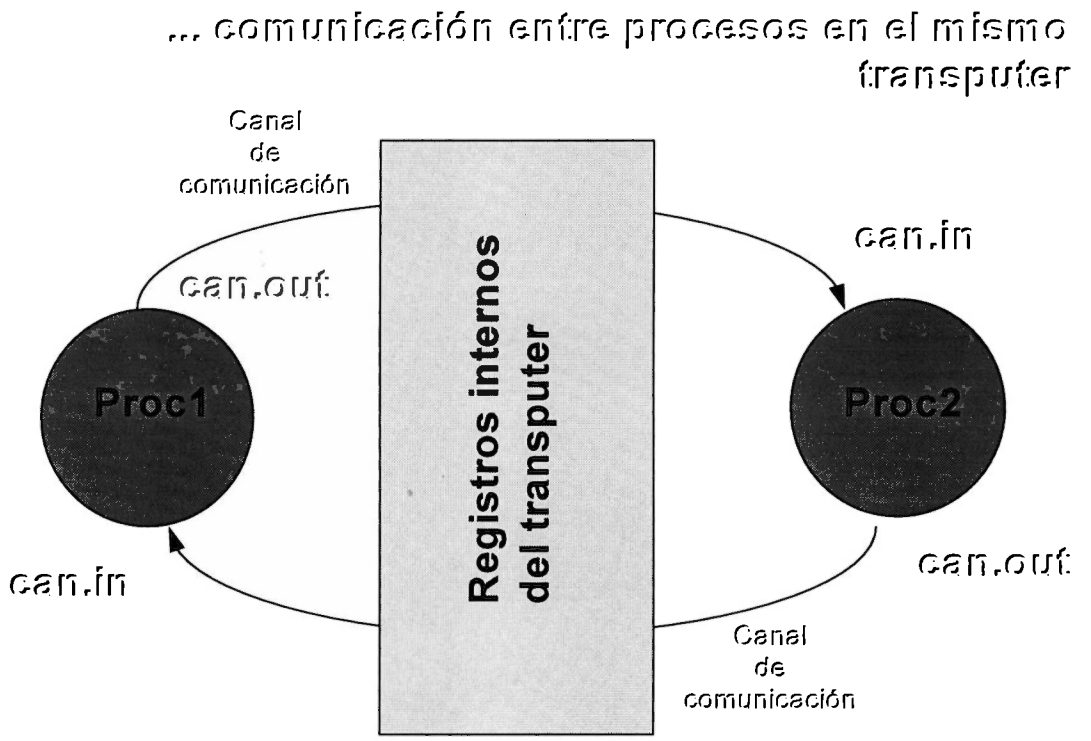


Fig. 5.3: Comunicación entre procesos en el mismo transputer

Si los dos procesos se encuentran en el mismo transputer, la comunicación se realiza internamente por medio de registros en memoria (ver Fig. 5.3). Los pasos necesarios para enviar datos en OCCAM son los siguientes:

1. Declaración del formato del mensaje que se enviará (PROTOCOLO en OCCAM). Por ejemplo, si voy a comunicar un proceso "A" con "EI" el formato de los mensajes se declara como:

```
PROTOCOL A.EI
  CASE
        ack.carac.e.ai
        final.a.ei
```

```
PROTOCOL E.AI
  CASE
        carac.e.ai; BYTE
        final.e.ai
  :
```

Esta declaración indica que se pueden intercambiar cuatro tipos de símbolos, ack.carac.e.ai, final.e.ai, carac.e.ai o final.a.ei.

2. Declaración de los canales que se van a comunicar, los cuales deben ser uno para cada sentido de la comunicación. En nuestro ejemplo tendríamos:

   CHAN OF A.EI aplic.a.enlacei

   CHAN OF E.AI enlace.a.aplici

3. Determinación de la forma de compartir los canales entre los procesos por medio de una conexión:

   aplicacion (aplic.a.enlacei, enlace.a.aplici)

   enlace (aplic.a.enlacei, enlace.a.aplici)

4. Declaración de cada uno de los procesos:

   Para el proceso de aplicación

```
. . .
--aplicacion.occ
#INCLUDE          "hostio.inc"
#INCLUDE          "proto.inc"

PROC aplicacion (   CHAN OF A.EI aplic.a.enlacei,
                    CHAN OF E.AI enlace.a.aplici )
. . .
```

   Para el proceso de enlace

```
. . .
--enlacei.occ
#INCLUDE          "hostio.inc"
#INCLUDE          "proto.inc"
```

PROC enlace (CHAN OF A.EI aplic.a.enlacei,
CHAN OF E.AI enlace.a.aplici ...)

. . .

5. Creación de un archivo de configuración en el que se declaran las conexiones de los procesos y la forma de ejecutarlos en uno o varios transputers. Un ejemplo de este archivo se muestra a continuación:

```
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
#INCLUDE "linkaddr.inc"
#USE "aplici.c8h"
#USE "enlacei.c8h"
CHAN OF SP from.host, to.host :
PLACED PAR
    PROCESSOR 0 T800
        PLACE from.host AT link0.in:
        PLACE to.host AT link0.out :
        CHAN OF A.EI a.a.ei :
        CHAN OF E.AI e.a.ai :
        aplici (from.host, to.host, a.a.ei, e.a.ai)
        enlacei (a.a.ai, e.a.ai)
```

6. Compilación de cada módulo con el uso de la herramienta **occam**:

   occam enlacei.occ

   occam aplic.occ

7. Unión de todos los módulos con el uso de un archivo de configuración (**conf.pgm**) con la instrucción:

   imakef conf.btl

8. Generación del archivo final ejecutable con:

   make -f conf

9. Ejecución del programa:

   iserver /se /sb conf.btl

5.1.2 Protocolo entre procesos en diferentes transputers

Una conexión entre dos transputers se implementa conectando las interfaces de los enlaces de dichos transputers por medio de dos cables unidireccionales por los que transitan datos tipo serial a 20 Mbps [8]. Los dos cables son dos canales de OCCAM (uno en cada dirección), los cuales siguen un protocolo simple para el envío y recepción de mensajes de información y control. Los mensajes son transmitidos como una secuencia de bytes, cada uno de los cuales debe ser confirmado con la recepción de un ACK antes de transmitir el siguiente byte. Cada byte de datos es transmitido como un bit de start (1), seguido por otro bit con valor de 1, los ocho bits de datos y finalmente un bit de stop (0).

El ACK que regresa el proceso receptor es transmitido como un bit de start seguido por un bit de stop como se ve en la Fig. 5.4. Al recibirlo el proceso emisor sabe que el proceso receptor:

1. Recibió el dato
2. Está listo para recibir el siguiente

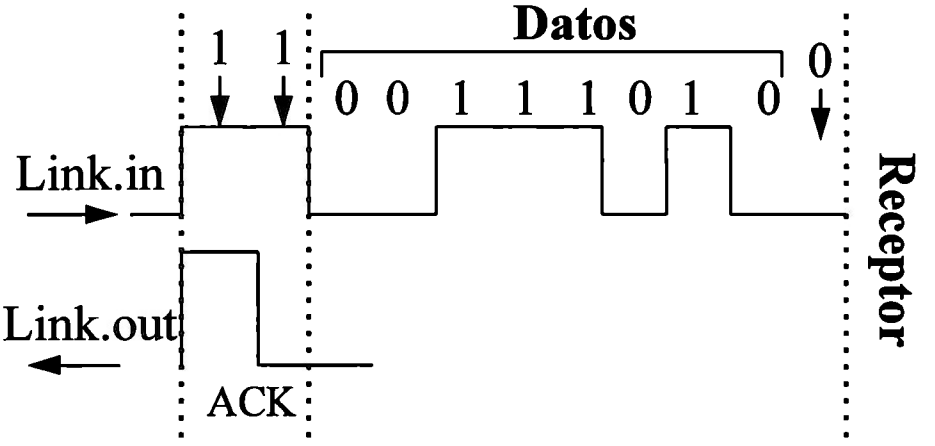


Fig. 5.4: Generación del ACK por el receptor

Este protocolo permite la generación de un ACK desde que el receptor ha identificado un paquete de datos, por lo que el ACK puede ser recibido por el emisor antes de que todo el paquete de datos sea transmitido.

Los pasos a seguir para comunicar procesos que están ubicado en diferentes transputers son iguales a los descritos en la sección 5.1.1, con una pequeña variación en el archivo de configuración que se mostró en el paso 6. En él tenemos que modificar la ubicación de los procesos con la instrucción **PLACED PAR,** colocándolos en procesadores diferentes. También tienen que colocarse los canales sobre los enlaces (link*x*.in, link*x*.out). Se muestra en seguida un ejemplo del archivo de configuración:

```
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
#INCLUDE "linkaddr.inc"
#USE "aplici.c8h"
#USE "enlacei.c8h"
CHAN OF SP from.host, to.host :
CHAN OF A.EI a.a.ei :
CHAN OF E.AI e.a.ai :
PLACED PAR
    PROCESSOR 0 T800
        PLACE from.host AT link0.in:
        PLACE to.host AT link0.out :
        PLACE a.a.ei AT link1.out :
        PLACE e.a.ai AT link1.in :
        aplici (from.host, to.host, a.a.ei, e.a.ai)
    PROCESSOR 1 T800
        PLACE a.a.ei AT link0.in :
        PLACE e.a.ai AT link0.out :
        enlacei (a.a.ai, e.a.ai)
```

5.2 Protocolo entre elementos del controlador y la CFM (capa Física)

Aquí se describe brevemente la capa física del controlador y la interfaz con robots utilizados en nuestro trabajo. Para más detalle puede consultarse [4].

5.2.1 Transputer - C011

Para realizar la comunicación desde un transputer hacia los robots se utiliza un adaptador de enlace llamado C011 [8]. Este es un dispositivo de alta velocidad que provee comunicación desde el enlace del transputer -de acuerdo al protocolo serial estándar de enlaces de INMOS explicado anteriormente- hasta algún microprocesador estándar como el MC8051, el cual recibe los datos del C011 en paralelo.

El IMS C011 puede recibir y enviar datos al transputer en cualquiera de las velocidades en que esté programado (5 Mbps, 10 Mbps ó 20 Mbps) y puede ser usado en dos modos:

MODO 1 (Interfaz periférica). Se hace uso de dos canales de comunicación, uno para entrada y otro para salida.

MODO 2 (Interfaz de BUS). Se utiliza un mismo bus de comunicación para lectura y escritura. Maneja varias señales de control para sincronizar la comunicación.

El Modo 1 fue el que se usó ya que no requiere de muchas señales de control. Puede haber dos direcciones de transmisión ya que el transputer puede enviar datos al robot por medio del C011 o recibir datos del robot. En ambos casos la comunicación usa un protocolo "handshake" que describiremos a continuación.

5.2.1.1 Modo escritura del C011

Cuando el transputer envía datos hacia el robot, el C011 recibe estos datos seriales por su enlace link.in (link.out de un transputer) y los coloca en forma paralela en sus líneas **D0** a **D7** para ser leídas por el microcontrolador MC51 (ver Fig. 5.5). El C011 genera además una señal **Qval** que indica que hay un dato listo para salida (escritura). El dispositivo externo (MC51) lee estos datos y contesta con una señal de **Qack.** Al mismo tiempo, el MC51 envía los datos leídos en forma serial hacia el

robot. Finalmente, el C011 contesta con un ACK al transputer por medio de su enlace link.out. La Fig. 5.5 muestra este protocolo marcando el orden de ejecución.

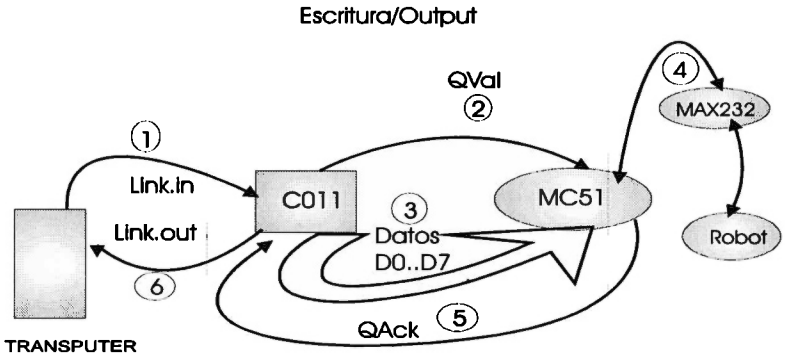


Fig. 5.5: Protocolo de escritura del IMS C011

5.2.1.2 Modo lectura del C011

Cuando el robot envía información hacia el Transputer (Fig. 5.6), el C011 recibe una señal del MC51 (**Ival**) que indica que hay un dato listo para recepción (lectura). El C011 lee estos datos en paralelo por medio de sus líneas **I0** hasta **I7,** los cuales posteriormente son enviados a través de su enlace **link.out** al transputer, mismo que contesta con un ACK. Finalmente el C011 contesta con un **Iack** al MC51 para confirmar la recepción.

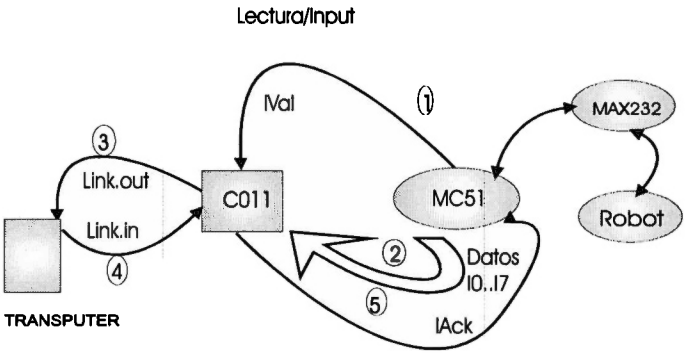


Fig. 5.6: Protocolo de lectura del IMS C011

5.2.2 Microcontrolador 8051 (MC51)

La velocidad máxima de las interfaces seriales de los robots se limita a 9.6 Kbps, por lo cual es necesario sincronizarlos con los enlaces de alta velocidad (20 Mbps) de los transputers. Ésto se logra con el microprocesador MC51 de Intel, el cual tiene cuatro puertos de comunicación paralelos [15], de 8 líneas de datos. El puerto 0 (Fig. 5.7) se usa para intercambiar datos de 8 bits en paralelo (provenientes del C011) con un bit del puerto 1 hacia el Robot (RS-232). La conversión serie-paralela (y viceversa) es hecha por un programa implementado en el MC51. Se usan algunas líneas del puerto 1 para control (**Qack, Iack, Qval, Ival**), y el resto de los puertos es usado para tolerancia a fallas [4].

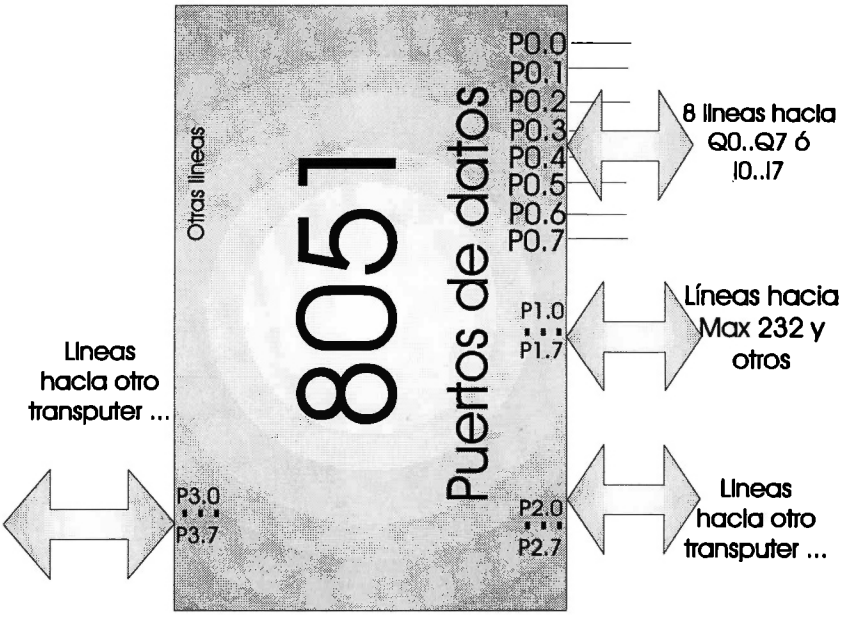


Fig. 5.7: Diagrama del microcontrolador 8051

5.2.3 Interfaz-RS232

Todos los elementos de la celda descrita en la sección 2.1 pueden comunicarse con otros dispositivos usando la interfaz RS232.

Este estándar surgió originalmente para la comunicación entre módems, utilizando las siguientes señales:

- Transmit Data
- Receive Data
- Data Set Ready
- Data Terminal Ready
- Request To Send
- Clear to Send
- Carrier Detect
- Ring Indicate

La mayor parte de los robots en una CFM usan esta interfaz en su forma más simple, realizando la comunicación con las señales que aparecen en la Fig. 5.8.

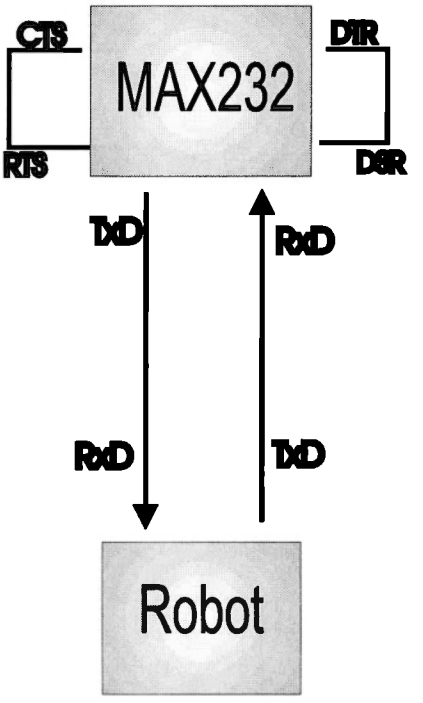


Fig. 5.8: Aplicación del Estándar RS-232

Obsérvese que el circuito MAX232 es el que se encarga del establecimiento de este protocolo, convirtiendo las señales que le llegan al MC51 de niveles de voltaje TTL a los niveles establecidos por el estándar RS232.

6. Protocolo diseñado para el controlador de la CFM

En el protocolo para el controlador de CFM consideramos algunos comportamientos que deben tomarse en cuenta en las capas de enlace y aplicación:

- El protocolo de envío de comandos al robot.
- El protocolo de recepción de información del robot.
- El reenvío limitado de caracteres que no se hayan recibido.
- Considerar que alguno de los módulos puede dejar de funcionar, en cuyo caso hay que avisar a quien sea necesario y dar una terminación correcta.

El trabajo de esta tesis puede aplicarse a la comunicación entre la interfaz diseñada por Alarcón [4] y cualquier robot que utilice el estándar RS232. Sin embargo, se requiere, como trabajo futuro, la implementación de un sistema en la capa de aplicación que se encargue de administrar tantos caminos de comunicación como sea necesario, así como un sistema en la capa de red que ayude al ruteo de mensajes. Consideramos que la investigación presentada en esta tesis es la base suficiente para continuar con este trabajo.

Considerando la dificultad de realizar las pruebas necesarias sobre una celda de manufactura durante el diseño del protocolo, se implementó un programa de prueba para simular el intercambio de información entre robots y el controlador.

El robot usado para nuestra prueba fue el Puma que se describió en la sección 2.1.1, y de aquí en adelante lo denotaremos genéricamente como "el robot". En esta primera versión, nuestro controlador prototipo envía comandos al robot desde la computadora huésped, caracter por caracter, hasta finalizar la entrada del comando. Aplicaciones más complejas serán definidas en trabajos futuros del proyecto [2].

En este capítulo describiremos la simulación que se hizo del robot y el protocolo diseñado, capa por capa. Para simplificar la descripción, ésta se hará según la dirección del flujo de datos (robot-controlador y viceversa). Sin embargo no hay que olvidar que los datos pueden fluir en ambas direcciones a la vez entre los dispositivos del controlador. En cada capa mostraremos los requerimientos de corrección del protocolo, los cuales fueron usados para construir el modelo de validación del mismo. Finalmente mostraremos los resultados de la validación usando Promela.

En el apéndice A mostramos el protocolo de envío de comandos sin tomar en cuenta la recepción, con el cual se realizaron pruebas para probar la comunicación unidireccional controlador-robot. En el apéndice B mostramos el protocolo implementado en sentido contrario, que usamos para probar la recepción de información del robot por medio de uno de los enlaces del transputer.

Finalmente, en el apéndice I se muestra la ejecución de los procesos en forma concurrente así como la forma de realizar la comunicación en ambos sentidos.

6.1 Diseño de protocolos

El diseño de protocolos es un problema de ingeniería cuya solución implica el uso de una gran variedad de técnicas conocidas. En seguida se menciona un conjunto general de principios para su diseño:

1. SIMPLICIDAD: Un protocolo bien estructurado debe componerse de piezas cuyas funciones son claras, simples y correctas, y su interacción está bien definida. Estos protocolos son fáciles de implementar, verificar y mantener.
2. MODULARIDAD: Un protocolo que realiza funciones complejas se construye de pequeñas piezas que interactúan de una forma sencilla y bien definida. Cada pieza es eficiente y fácil de verificar. Estos módulos se desarrollan como entidades independientes de tal forma que no hacen suposiciones complicadas sobre el trabajo o contenido de otros módulos.
3. ESTRUCTURA CORRECTA: No debe haber código inalcanzable; debe contemplar todos los casos posibles, como recepciones de mensajes o situaciones imprevistas. Debe tener límites bien definidos, como lo es el caso de sus espacios de almacenamiento, el tráfico de mensajes, etc. Los protocolos deben ser adaptables a las distintas velocidades de los medios con que se cuenta.
4. ROBUSTEZ: El protocolo debe estar preparado para responder a cualquier secuencia factible de acciones.
5. CONSISTENCIA: El protocolo debe estar preparado para detectar y responder a: Deadlocks (abrazos mortales), Livelocks (esperas infinitas) y Terminaciones Impropias.

Como respuesta a los requerimientos anteriores se han desarrollado diez reglas que ayudan a realizar un buen diseño de protocolos, mismas que se tomaron en cuenta para el nuestro:

1. Definir bien el problema.
2. Definir bien el servicio que proveerá.
3. Diseñar las funciones externas antes que las internas.
4. Buscar siempre simplicidad.
5. No relacionar lo que es independiente.
6. No incluir lo innecesario o validar lo que es irrelevante.

7. Antes de la implementación, diseñar un prototipo y verificarlo.
8. Implementar el diseño, probarlo y si es necesario optimizarlo.
9. Verificar que el diseño final se apegue al prototipo probado.
10. No evitar las reglas uno a la siete.

6.2 Estructura General del protocolo

En el protocolo de comunicación entre robot y controlador hay que permitir el flujo de información en ambos sentidos, lo que implica intercambio de mensajes diferentes dependiendo de la dirección en que estén corriendo los datos (hacia el robot o hacia el controlador).

Tomando en cuenta este modelo de intercambio de mensajes, se determinó la creación de dos caminos de comunicación paralelos que se muestran en la Fig. 6-1.

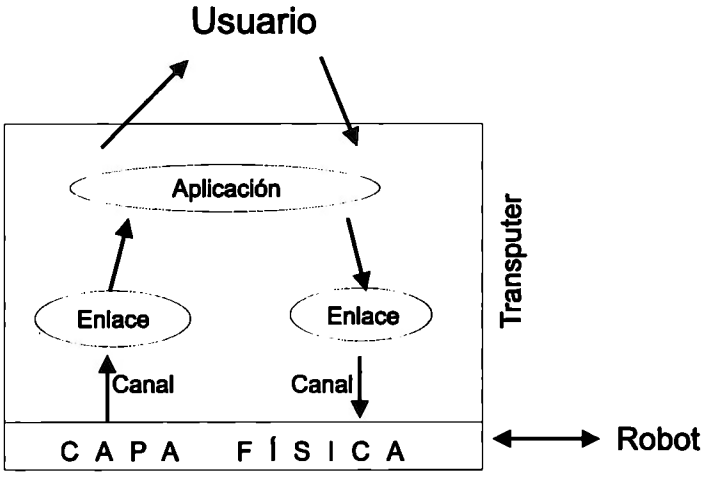


Fig. 6-1: Diagrama general del Protocolo diseñado

Desarrollamos las capas de protocolo del controlador adaptándonos al funcionamiento del robot y añadiendo mecanismos que permitan una comunicación rápida y confiable entre ambos.

La descripción del protocolo de comunicación se dará por módulos para comprender cada parte del mismo. Al finalizar mostraremos el comportamiento global.

Cada módulo se explicará enumerando las siguientes características:

1. Diagrama que muestra el funcionamiento del proceso, especificando el servicio que provee.
2. Requerimientos de corrección, que nos servirán para validar el protocolo.

Las siguientes características de cada módulo se presentan en los apéndices C al G.

3. Especificación del vocabulario del proceso y el formato de sus mensajes.
4. Medio Ambiente sobre el que se desenvuelve el protocolo.
5. Autómata del proceso y definición formal del mismo.
6. Listado del código del proceso en OCCAM.

Los errores que contempla nuestro protocolo no incluyen aquéllos de modificación o duplicado de datos, ya que los enlaces a corta distancia entre transputers proveen comunicación segura y sin error, es decir garantizan la transmisión correcta de cada byte. Aumentar un campo para enviar un dato como el CRC no se justifica, ya que los datos que intercambian los transputers y el robot son de tamaño pequeño. Por otro lado, para la comunicación con el robot no podemos agregar campos al formato del mensaje preestablecido por el fabricante.

El protocolo tiene instrucciones que se enfocan a detectar procesos que deben contestar y que después de un cierto tiempo no lo hacen (fallas de tiempo). Se detecta este tipo de fallas por medio de temporizadores implementados con la instrucción ALT de OCCAM (ver Capítulo 3). Este lenguaje cuenta además con la biblioteca "xlink.lib" que tiene cinco instrucciones para depositar y recibir datos de un enlace, así como reinicializarlo en caso de encontrar algún error. Esta reinicialización se hace en los dos extremos del canal, el del emisor y el receptor. De las cinco instrucciones de esta biblioteca, son dos las que se usan:

1. OutputOrFail: deposita un dato al enlace o detecta que éste no responde.
2. Reinitialize: Reinicializa el canal en ambos extremos.

Las otras instrucciones cubren requerimientos que no se presentan en nuestro protocolo.

Existen dos casos en los que no se cuenta con la información necesaria para detectar una falla de tiempo. Primeramente, cuando la capa de enlace está esperando datos del robot, pues no sabemos cuánto tiempo pasará antes de que el robot nos envíe información. El otro caso ocurre cuando la capa de aplicación está esperando la recepción de algún comando, pues no sabemos cuánto tiempo va a pasar antes de que el usuario provea alguno.

Por otro lado, cuando un proceso manda un dato a otro proceso y se espera su reconocimiento de recepción, sí podemos detectar una falla de tiempo, ya que sabemos que el receptor debe contestar muy rápidamente.

Para cada proceso del protocolo que describimos a continuación, se inicia la explicación con un diagrama que muestra su funcionamiento. La notación usada en estos diagramas es similar a la sintaxis de programación en lenguajes que utilizan canales, como OCCAM (capítulo 3) y Promela (sección 4.3 y apéndice L). Cuando se envía un mensaje a un proceso se usa la notación:

proceso ! mensaje

La recepción de mensajes se especifica como:

proceso ? mensaje

Esta comunicación se comporta de la misma manera que en OCCAM: es síncrona y punto a punto. Adicionalmente a las dos funciones anteriores, existe una tercera implementada con la instrucción OutputOrFail, que detecta cuando un enlace no responde. Esta instrucción la denotaremos como:

proceso !! mensaje

Cuando usamos esta instrucción, en caso de fallar la salida de un mensaje por un enlace, el proceso intenta reenviarlo un máximo de cuatro veces, reinicializando cada vez el canal antes de hacerlo. En el caso de enviar mensajes por canales internos al transputer no existe manera de hacer reinicialización, por lo cual ésta no se implementó. Se deja para trabajo futuro el detectar la falla en la transmisión de alguno de estos mensajes, así como la implementación de posibles soluciones.

En base a lo establecido anteriormente, a continuación explicaremos cada capa del protocolo, comenzando por la simulación del robot.

6.3 Simulación del Robot.

El proceso de control (capa de aplicación) que se implementó en esta tesis recibe comandos desde el teclado y despliega en pantalla los datos que envía el robot. Se usó una pantalla ASCII para ésto, por lo que el programa de aplicación provee exclusión mutua sobre el uso de este recurso.

El programa de aplicación se compone de los siguientes procesos: el que recibe datos del robot (se comunica con la capa de enlace) y el que espera comandos del usuario (se comunica con la computadora huésped).

Una vez que se probó el buen funcionamiento de la interfaz desarrollada en [4] para comunicar el robot con un transputer, se implementó un proceso de simulación del robot, que nos permitiera realizar pruebas y modificaciones al controlador sin la necesidad de estar conectado a la celda de manufactura.

En la Fig. 6.2 vemos que este proceso se coloca como receptor de los datos que vienen y van hacia la capa de enlace del controlador.

El protocolo de comunicación del proceso de simulación es el mismo con que se realiza la comunicación al robot, con la diferencia de que la capa de enlace no se

comunica con la capa física para enviar y recibir datos del robot, sino que se comunica con un canal Occam conectado al proceso de simulación.

El proceso de simulación puede encontrarse en el mismo o en otro transputer (Fig. 6.2). Ésto no requiere de modificaciones al programa, tan sólo del archivo de configuración de los procesos, donde se indica la ubicación del robot simulado.

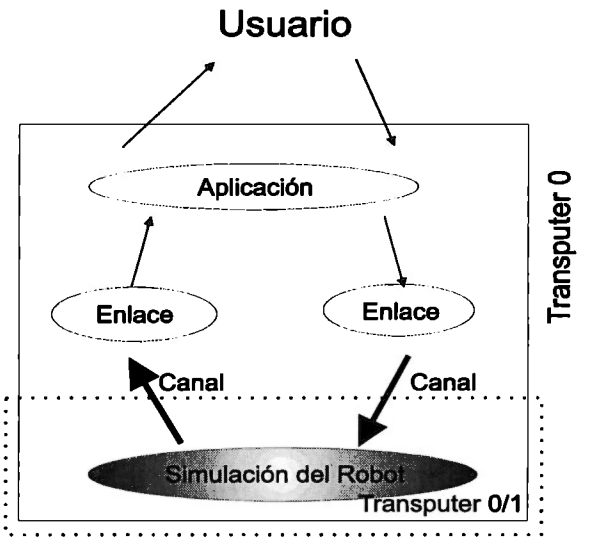


Fig. 6.2: Conexión del proceso de simulación del robot

6.3.1 Funcionamiento y servicio que provee

Cuando el robot inicia, pregunta al usuario si desea cargar el sistema operativo desde el disco flexible ("Load VALII from floppy (Y/N) ? "); después de recibir la respuesta, pregunta si se desea que inicialice sus parámetros ("Initialize (Y/N) ? "). El robot se dedica después a recibir comandos y a ejecutarlos, contestando al controlador con un ACK de ejecución de comando, representado por un caracter ".". En la Fig. 6.3 se muestra un diagrama de bloques que describe este proceso y que se explica a continuación:

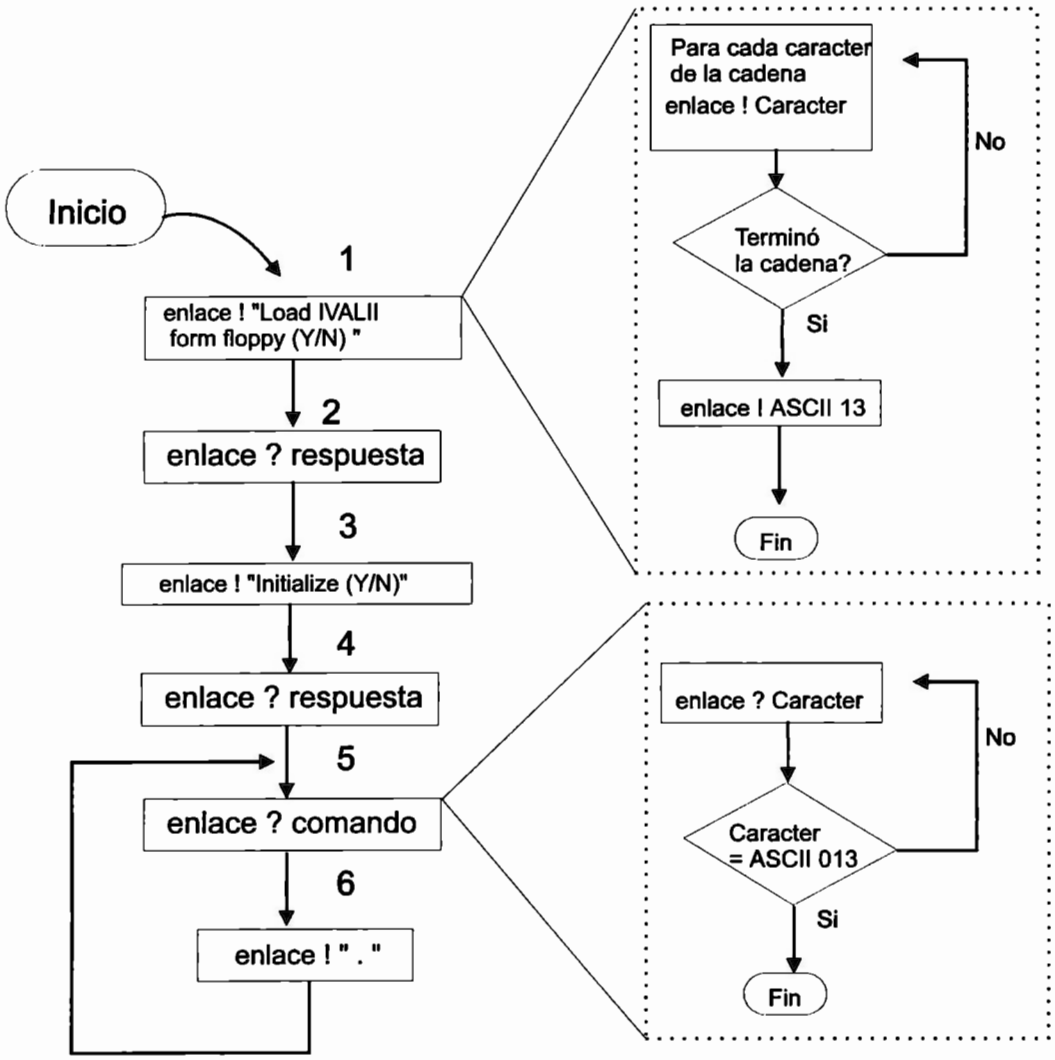


Fig. 6.3: Funcionamiento de la Simulación del Robot

1.- Se envía la primer pregunta al controlador (capa de enlace):

- Se envía caracter por caracter.
- Al final, se envía el código correspondiente al ASCII 013 (ENTER).

2.- Se recibe respuesta de la capa de enlace.

3.- Se envía la segunda pregunta al controlador (capa de enlace):

- Se envía caracter por caracter.
- Al final se envía el código correspondiente al ASCII 013 (ENTER)

4.- Se recibe respuesta.

Para cada comando:

5. - Se recibe el comando:
   - Se recibe caracter por caracter.
   - Se termina al recibir un ENTER.
6. Se envía un " . " a la capa de enlace (ACK de ejecución del comando) y regresa al paso 5.

Puede consultar en el APÉNDICE C la descripción completa del autómata de este proceso y su código.

6.3.2 Requerimientos de Corrección

Ya que este proceso se implementó como una simulación del robot, el protocolo no verifica su buen funcionamiento. Esta tarea corresponderá en un futuro a un módulo monitor en la capa de aplicación, no implementado en este trabajo de tesis. El proceso de simulación se dedica a recibir y enviar información al controlador, independientemente de que los comandos sean o no correctos.

6.4 Descripción de cada proceso del protocolo en la comunicación Robot -> Controlador

En esta sección se describen los procesos que permiten la comunicación del robot hacia el controlador (Fig. 6.4). Al hablar de controlador se incluyen tanto el proceso de la capa de enlace como el de la capa de aplicación.

6.4.1 Capa de enlace (Robot -> Controlador).

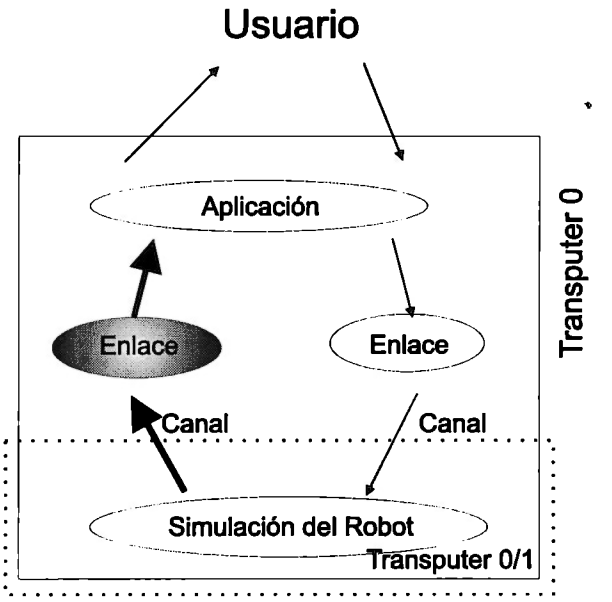


Fig. 6.4: Ubicación del módulo Enlace (Robot -> Controlador)

En esta capa realizamos la recepción de los datos que vienen de la capa física del controlador para posteriormente enviarlos a la capa de aplicación (Ver Fig. 6.4), donde se le da el tratamiento requerido.

El objetivo de esta capa es detectar posibles fallas de comunicación con el proceso de aplicación, por lo que se implementa un mecanismo de reenvío de mensajes. Este reenvío es implementado para asegurar que un caracter recibido por el controlador llega eventualmente a la capa de aplicación. En una etapa futura de este proyecto se requiere que la aplicación se recupere de fallas, por lo que este mecanismo de reenvío será muy útil.

No se intenta detectar fallas de comunicación con el robot pues la comunicación con éste es asíncrona. Se dejará al proceso de aplicación (en particular a un proceso monitor) la tarea de detectar estas fallas en implementaciones futuras.

6.4.1.1 Funcionamiento y servicio que provee

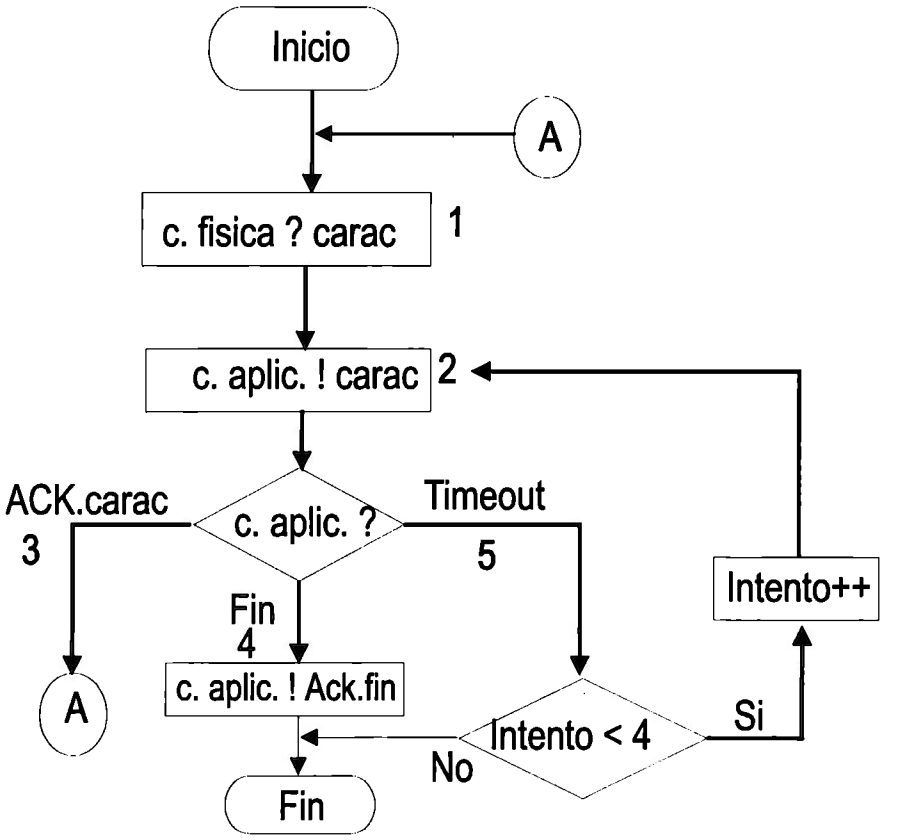


Fig. 6.5: Funcionamiento de la capa de Enlace (Robot -> Controlador)

En la Fig. 6.5 vemos el comportamiento del protocolo que se explica a continuación:

1. Se recibe un caracter de la capa física.
2. Se envía el caracter recibido a la capa de aplicación y cualquiera de los siguientes eventos puede ocurrir:
    3. Se recibe ACK del caracter enviado a la capa de aplicación. Regresamos entonces a esperar otro caracter.
    4. Se recibe una petición de final de la capa de aplicación. Se le envía entonces un Ack.fin y se termina.

5. Timeout: no llega el ACK de la capa de aplicación, por lo que se regresa al paso 2 para volver a enviar el caracter a la capa de aplicación. Este reenvío sucede máximo cuatro veces. En caso de no tener éxito, el proceso se da de baja.

Ver APÉNDICE D para la descripción completa del autómata de este proceso y su código.

6.4.1.2 Requerimientos de Corrección

1. En esta capa del protocolo es muy importante tener la seguridad de que la capa de aplicación está trabajando, para lo cual es básico que conteste con un ack a cada caracter que se le envía.
2. Necesitamos asegurarnos de que el ciclo de reenvío no sea infinito.

El error que podría tener este proceso es que la capa de aplicación, o el canal de comunicación entre estos dos procesos, no estén funcionando correctamente y por lo tanto no se reciba el mensaje que la capa de enlace está enviando. En este caso el proceso de la capa de enlace se quedaría esperando eternamente a que la capa de aplicación reciba el caracter, puesto que el canal es bloqueante.

Consideramos que este problema puede resolverse colocando un proceso intermedio que funcione como un buffer, es decir, que se encargue de recibir los datos que vienen de la capa de enlace (capa emisora) y de enviarlos a la capa de aplicación (capa receptora).

Este proceso podría almacenar algunos de los datos recibidos mientras no puedan ser enviados a la capa receptora, es decir implementando una comunicación no bloqueante. Si la capa receptora tiene alguna falla, ésta sería detectada por el proceso intermedio y la capa emisora no se quedaría bloqueada. El proceso intermedio debería ser desbloqueado en este caso por un monitor.

6.4.2 Capa de aplicación (Robot -> Controlador).

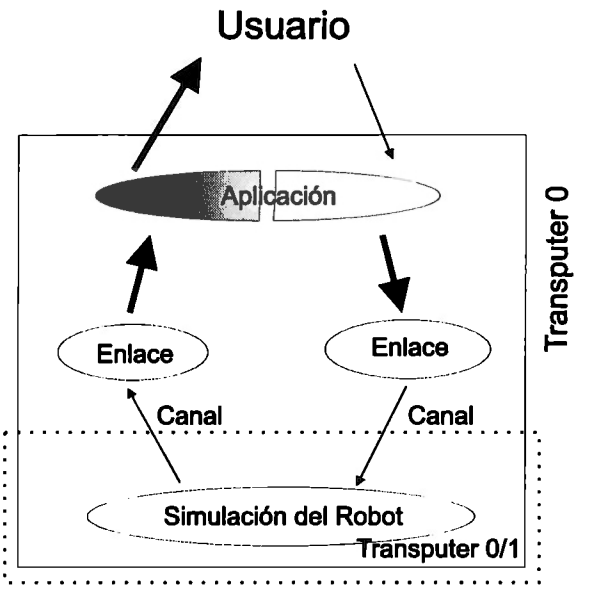


Fig. 6.6: Ubicación del módulo Aplicación (Robot -> Controlador)

Durante la ejecución normal de una CFM, esta capa será la encargada de planificar, supervisar y controlar la ejecución de todos los elementos de la celda (Fig. 6.6).

En esta sección describiremos el funcionamiento de la parte de esta capa que se encarga de recibir los datos que vienen de la capa de enlace, para ser desplegados en la pantalla de la computadora huésped. En cuanto inicia la recepción de caracteres provenientes del robot, este programa se dedica exclusivamente a atender a la capa de enlace. La recepción de información termina cuando se recibe un comando de finalización de parte de la capa de enlace.

6.4.2.1 Funcionamiento y servicio que provee

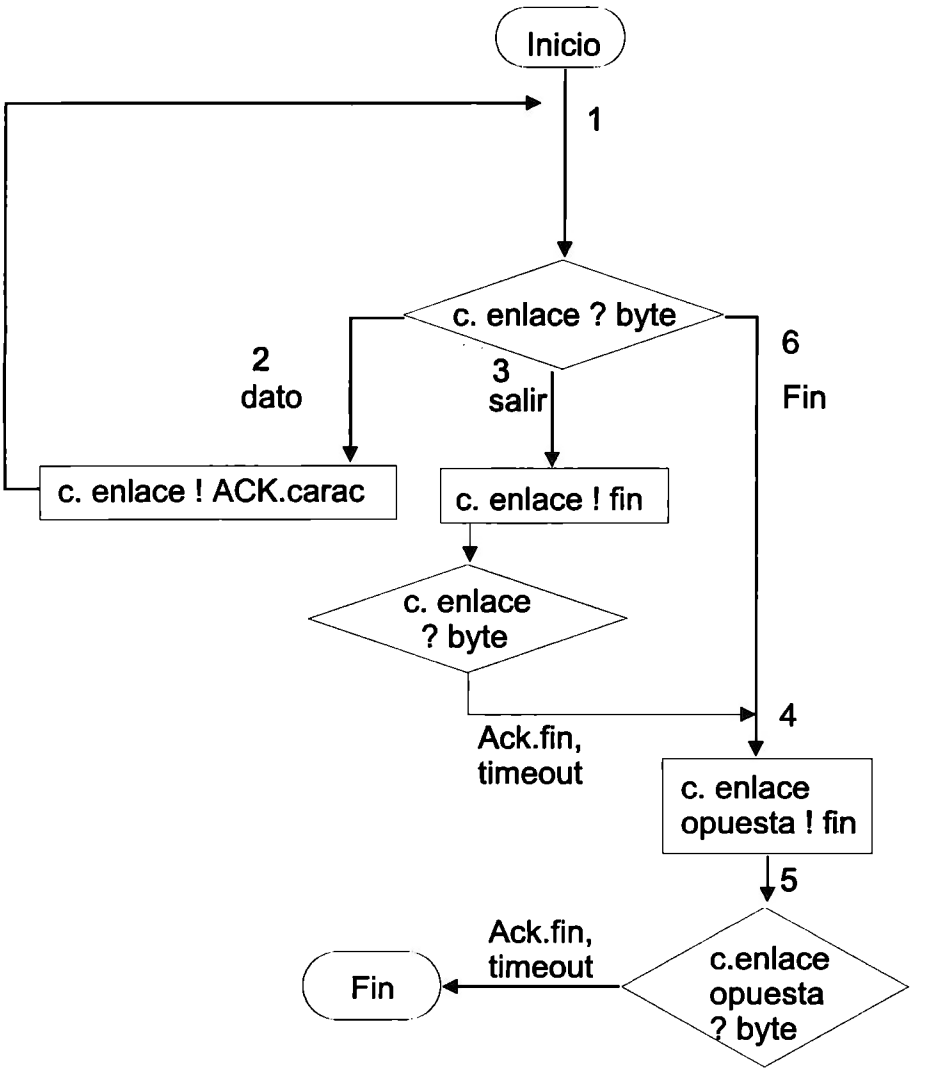


Fig. 6.7: Funcionamiento de la capa de Aplicación (Robot -> Controlador)

En la Fig. 6.7 vemos el funcionamiento de esta capa, que describiremos a continuación:

1. Se recibe un byte de la capa de enlace, con lo cual puede suceder lo siguiente:
    2. Si corresponde a un dato, se contesta con un ACK y se regresa al estado inicial.

3. Si corresponde a un código de salir por parte del robot que desea terminar la comunicación, y dado que la capa de enlace no sabe el contenido de este mensaje, se le avisa con un mensaje de fin. Después, se espera la confirmación de recepción de Fin. Si ésta no se recibe, se informa. Se continúa con los siguientes pasos:
    4. Se informa a la capa de enlace del camino opuesto (sección 6.5.2) que debe finalizar.
    5. Se espera su confirmación y se finaliza. En caso de que ésta no se reciba, se informa y se finaliza.
6. Si la capa de enlace está solicitando finalizar, se realizan los pasos 4 y 5.

Ver APÉNDICE E para la descripción completa del autómata de este proceso y su código.

6.4.2.2 Requerimientos de Corrección

1. En caso de que esta capa ya no pueda (o no quiera) seguir funcionando, se informa a las dos capas de enlace. Es importante asegurarnos de que estas capas han recibido el comando de fin, por medio de un caracter de reconocimiento que ellas deben enviar, llamado ack.fin.

Un error adicional que podría presentarse, es la falta de recepción de cualquiera de los tres mensajes que esta capa envía hacia la capa de enlace (c. enlace ! ACK.carac, c. enlace ! fin, c. enlace opuesta ! fin). La solución propuesta para la capa de enlace (sección 6.4.1.2) sobre la simulación de canales asíncronos, también podría aplicarse aquí.

Nótese además que aquí no se implementa ningún mecanismo de reenvío de mensajes, pues de ésto se ocupa la capa de enlace.

6.5 Descripción de cada proceso del protocolo en la comunicación Controlador -> Robot

Aquí se describirán los dos módulos que se ejecutan dentro del controlador para enviar información al robot. Éstos son el proceso de aplicación, que se encarga de recibir líneas de comando de la computadora huésped, y el proceso enlace, que es el intermediario entre las capas de aplicación y física.

6.5.1 Capa de aplicación e interfaz con el usuario (Controlador -> Robot).

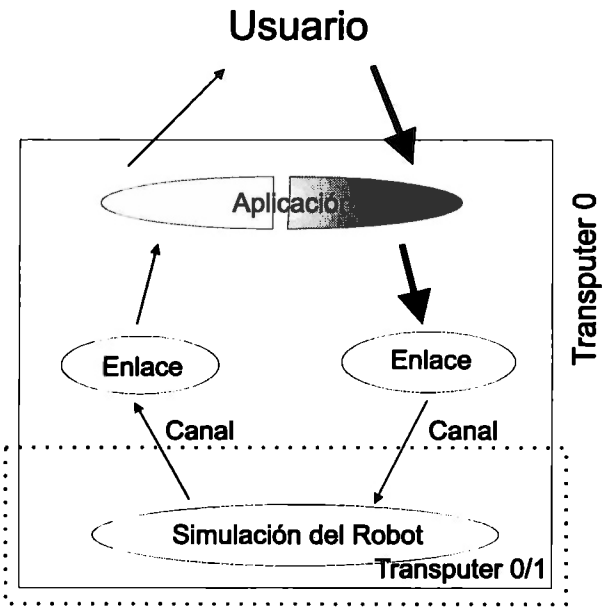


Fig. 6.8: Ubicación del módulo Aplicación (Controlador -> Robot)

Esta capa se encarga de recibir comandos de la computadora huésped para ser enviados al robot a través de las capas de enlace y física (Fig. 6.8). La recepción de cada comando termina cuando se recibe un ENTER (ASCII 013) que indica el fin. El

término de la captura de datos se da cuando el comando es un caracter "\" o éste forma parte de un comando (que ya no se envía).

6.5.1.1 Funcionamiento y servicio que provee

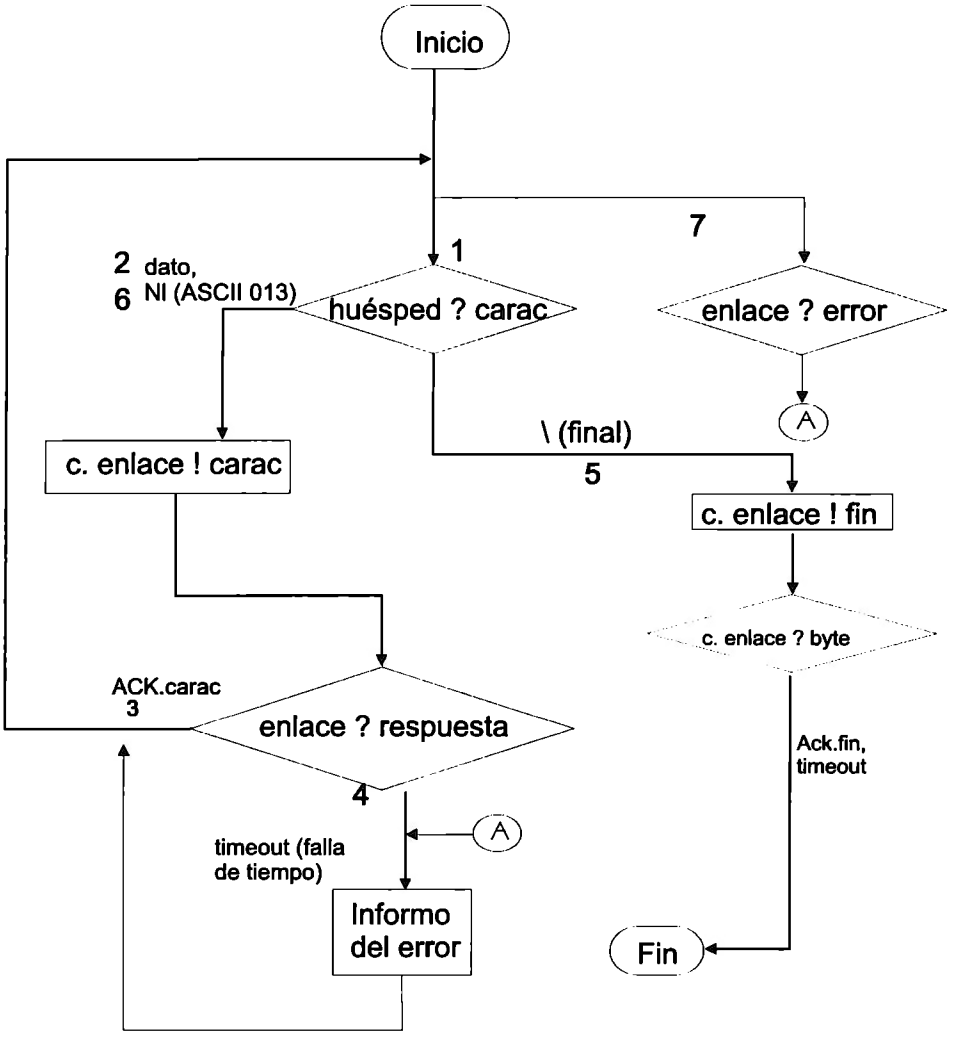


Fig. 6.9: Funcionamiento de la capa de Aplicación (Controlador -> Robot)

El funcionamiento de este módulo puede describirse con la ayuda de la Fig. 6.9. Los pasos en los que se puede estar son:

1. Recepción de un caracter de la computadora huésped o PC (el cual forma parte de un comando). El valor del caracter determina las siguientes opciones de ejecución:

    2. Si es un elemento del comando (un dato) entonces se envía ese caracter a la capa de enlace. Esta capa puede contestar:

        3. ACK de caracter recibido. Volvemos entonces al paso uno a esperar otro caracter del comando.

        4. Falla de tiempo (timeout) al esperar respuesta de la capa de enlace, se informa y se regresa al paso uno.

    5. En caso de que el caracter recibido del usuario en el paso uno sea un caracter de final \, se informa a la capa de enlace, se espera la confirmación de fin y este módulo también finaliza. En caso de no recibir confirmación, informa y finaliza.

    6. Al recibir un ASCII 013 (ENTER) del usuario, el proceso realiza los mismos pasos del 2 al 4, para después volver al estado inicial y esperar la recepción de un nuevo comando.

7. Error que indica falla de la capa física. Este error se detecta después de que la capa de enlace no puede comunicarse con la capa física. Se informa a la aplicación y se regresa al paso uno.

Note que este proceso y el de enlace *no* terminan aún después de ocurrir un error cuando están esperando un reconocimiento de sus procesos receptores. La finalización sucede cuando el usuario lo requiere explícitamente. Ésto permitirá que mecanismos de recuperación puedan ser efectuados en los robots (p. ej. Un RESET) o desde la capa de aplicación (p. ej. Un proceso monitor de recuperación) sin necesidad de finzalizar los procesos del controlador.

Ver APÉNDICE F para la descripción completa del autómata de este proceso y su código.

6.5.1.2 Requerimientos de Corrección

1. Para que esta capa funcione, es necesario saber que esté trabajando la capa de enlace, pasando información a la capa física. Requerimos entonces una respuesta de ack cada vez que se envíe un dato.

Un error que podría presentarse es la falta de recepción de cualquiera de los dos mensajes que esta capa envía hacia la capa de enlace (c. enlace ! carac, c. enlace ! fin), como en el caso de las capas de enlace y aplicación (Robot -> Controlador) anteriores. Para solucionar este error puede emplearse la misma técnica descrita en la sección 6.4.1.2.

6.5.2 Capa de enlace (Controlador -> Robot).

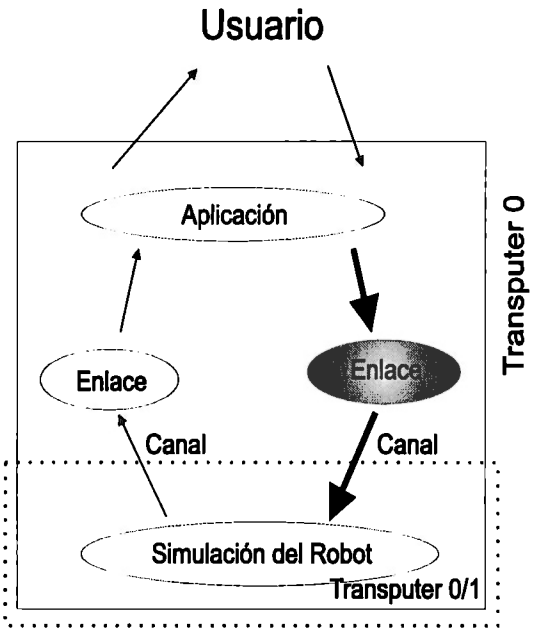


Fig. 6.10: Ubicación del módulo Enlace (Controlador -> Robot)

Esta capa se asegura de que los mensajes provenientes de la aplicación lleguen a la capa física, (Fig. 6.10). Para ello se implementa un mecanismo de reenvío que funciona cuando falla la recepción de datos por algún enlace del transputer. El reenvío funciona máximo cuatro veces y en caso de no tener éxito, se informa a la capa de aplicación sobre el error.

Note que aquí no se verifica que el robot regrese "." (Ack para reconocer un comando), pues ésto debe ser hecho por la aplicación, una vez que se envían todos los caracteres del comando.

6.5.2.1 Funcionamiento y servicio que provee

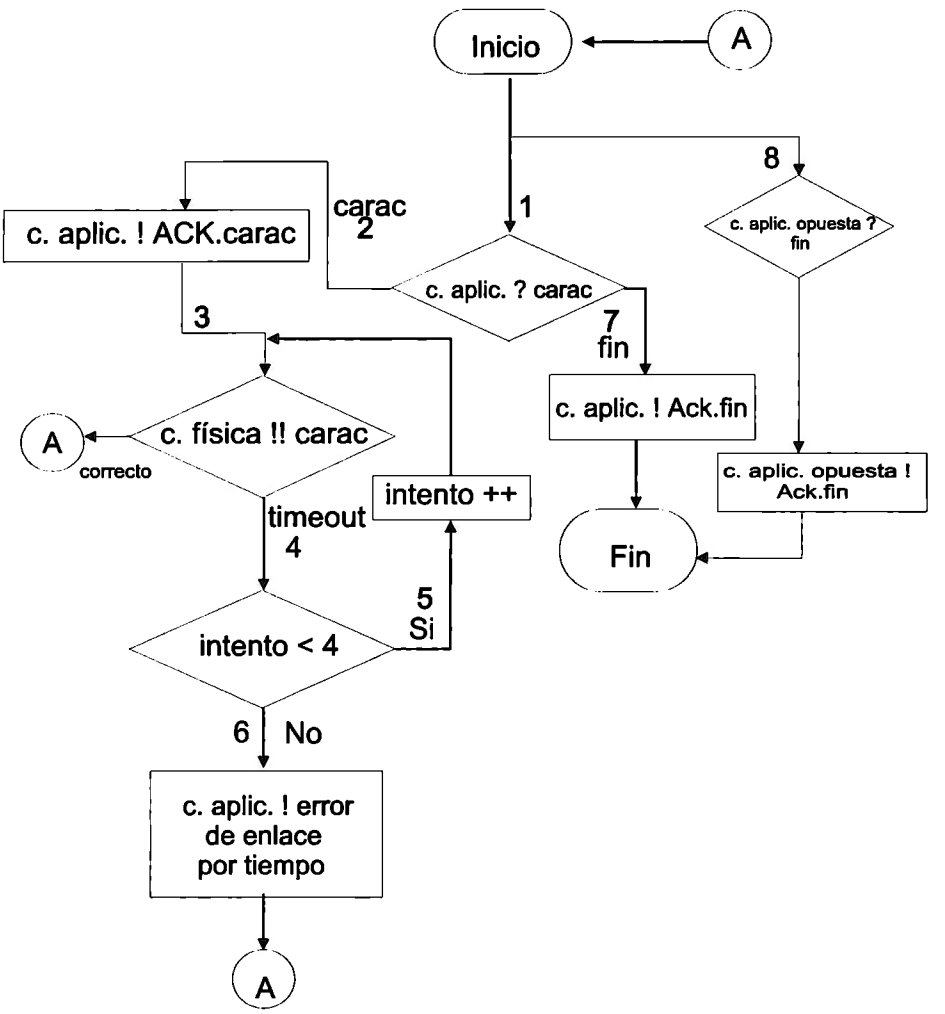


Fig. 6.11: Funcionamiento de la capa de Enlace (Controlador -> Robot)

El funcionamiento de este módulo puede describirse con la ayuda de la Fig. 6.11:

1. Se recibe un caracter de la capa de aplicación. Este caracter puede tener dos valores:
    2. Caracter normal, con lo cual se contesta un Ack de caracter recibido a la capa de aplicación y se realiza lo siguiente:
        3. Se envía al robot el caracter por un enlace del transputer.
        4. Si se detecta una falla de tiempo al enviar a la capa física, se realizan reenvíos (después de reinicializar el canal) de la siguiente forma:
            5. Si el número de intentos es menor a 4 volvemos a enviar.
            6. Si los intentos ya fueron 4, se informa a la capa de aplicación de un error en el enlace y se regresa al estado inicial.
    7. En caso de que el caracter enviado por la capa de aplicación sea una petición de final, ésta se confirma con un Ack.fin y se finaliza.
8. Se recibe una petición de finalizar de la capa de aplicación opuesta, se confirma con un Ack.fin y se finaliza.

Ver APÉNDICE G para la descripción completa del autómata de este proceso y su código.

6.5.2.2 Requerimientos de Corrección

1. En primer lugar necesitamos asegurar que la capa física esté recibiendo los datos que le estamos enviando.
2. Que el ciclo de reenvío no sea infinito.

Otro error que podría presentarse, es la falta de recepción de cualquiera de los tres mensajes que esta capa envía hacia la capa de aplicación (c. aplic ! Ack.carac, c. aplic ! Ack.fin, c. aplic ! error de enlace), para lo cual se propone la solución de envíos asíncronos como la mencionada en la sección 6.4.1.2.

6.6 Consideraciones generales

El protocolo implementado tiene la ventaja de ser rápido gracias al uso de mensajes cortos. Además cuenta con instrucciones que le ayudan a detectar la falta de respuestas y declarar fallas de tiempo, pudiendo así tomar acciones correctivas.

Por ser un protocolo que funciona en una área muy pequeña, no requiere detección de errores en cuando a la modificación, pérdida o duplicación del dato que viaja entre procesos. Sin embargo, es probable que ésto tenga que integrarse al sistema en un futuro, si se desea la propiedad de transparencia de red.

El uso del transputer es una gran ventaja ya que se ha podido observar su rapidez en la comunicación y el procesamiento de información (cuatro enlaces de comunicación concurrentes a 20 Mbps, procesador a 10 Mips y unidad de punto flotante a 1.5 Mflops). Por otro lado, haber realizado esta investigación con el uso del lenguaje OCCAM nos aporta muchas facilidades para el desarrollo de protocolos ya que, como se había mencionado, es de naturaleza paralela y garantiza la sincronización entre procesos. Una de las desventajas de OCCAM es que su sintaxis no es muy amigable, ya que obliga a indentar los programas en lugar de usar instrucciones de Inicio y Fin. Cuando se ha llegado a cierto nivel de anidación, ésto ya no resulta muy conveniente.

En la Fig. 6.12 mostramos un ejemplo del flujo de datos entre el robot y el controlador en el tiempo. Las líneas MAX 232, MC51 y C011 corresponden a la capa física, mientras que nuestro protocolo cubre la operación del T805.

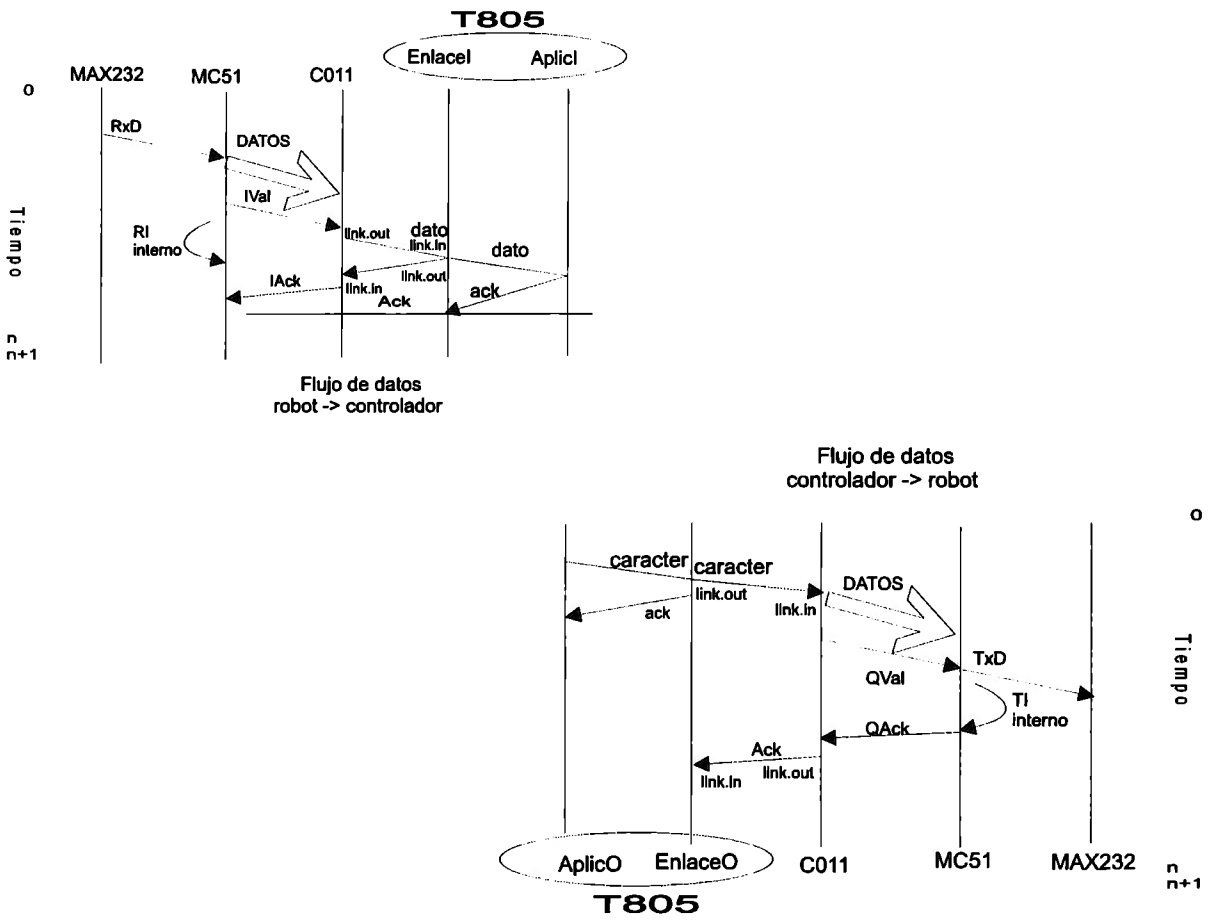


Fig. 6.12: Ejemplo del flujo de datos del protocolo en ambas direcciones

6.7 Validación del Protocolo usando el modelo en Promela

En esta sección se muestra el modelo implementado en Promela para efectos de prueba y validación, para lo cual se usan los requerimientos de corrección mencionados a lo largo de todo este capítulo. En esta validación no se pretende ser exhaustivo, sino mostrar sólo algunos ejemplos de requerimientos de corrección en nuestro protocolo, y la forma de validarlos. El modelo simula la generación de errores en forma aleatoria, de tal forma que se pueda mostrar el comportamiento del protocolo ante la presencia de éstos. De igual forma se prueba que los predicados establecidos para su validación no sean violados.

En la sección 6.7.1 explicaremos los pasos que se siguieron para la construcción del modelo, así como las herramientas que se usaron para validarlo.

Posteriormente, en la sección 6.7.2, se hablará del funcionamiento del modelo, y en la sección 6.7.3 se describirán los predicados que nos ayudaron a validar los requerimientos de corrección.

Finalmente, en las últimas secciones mostraremos los resultados sobre las variantes hechas para validar que el protocolo se comporte bien, en todas las ejecuciones posibles, ante la ocurrencia de diversos tipos de errores.

6.7.1 Pasos seguidos para la construcción y validación del modelo

Para realizar la construcción y validación del protocolo se siguieron los siguientes pasos:

1. Implementación en Promela de un modelo de validación del programa controlador de la celda de manufactura, implementado a su vez en OCCAM.
2. Uso de aserciones (assert) y estados marcados como finales (end:) para comprobar los requerimientos de corrección que se establecieron en cada una de las capas del protocolo.
3. Generación de un verificador en Promela con la instrucción:

spin -a <modelo>

La línea anterior crea una serie de archivos llamados ***pan*** con diferentes extensiones, mismos que servirán para la compilación y ejecución del modelo. Los archivos que se generan son:

pan.c:	Contiene casi en su totalidad el código en C para el análisis del protocolo.
pan.t:	Contiene una matriz de transición que codifica el control de flujo del protocolo.
pan.b y pan.m:	Contiene código en C para poder realizar transiciones hacia adelante y hacia atrás.
pan.h:	Es un archivo general con definiciones.

4. Para poder usar el verificador, el archivo pan.c debe compilarse por medio del siguiente comando:

gcc -o pan pan.c

En ocasiones no se puede realizar una búsqueda exhaustiva para la verificación (ver sección 4.3.3) dado el número exponencial de estados analizados. En ese caso, es necesario hacer una *búsqueda parcial controlada,* también llamada *supertrace,* en donde la compilación debe ser:

gcc -DBITSTATE -o pan pan.c

Cuando el análisis que realiza búsqueda exhaustiva falla, quiere decir que hay más estados alcanzables que los realmente calculados (por limitaciones de memoria) en el espacio de estados del sistema analizado. Spin ayuda a mejorar este problema informando que la búsqueda no fue exhaustiva, y permitiendo la compilación y ejecución de un análisis con búsqueda parcial.

En el caso de protocolos pequeños, el cambio entre los dos tipos de compilación no es representativo, ya que la verificación va a ser completa en ambos. En el caso de protocolos medianos y grandes, el primer tipo de verificación no es suficiente.

5. En el paso cuatro se obtiene un verificador llamado ***pan***, mismo que puede ejecutarse simplemente tecleando su nombre en DOS, o bien, tomando en cuenta las siguientes recomendaciones:

a). Se puede variar el tamaño de la tabla de estados con el parámetro -wN (que por default es 18), por ejemplo:

pan -w23

pan -w25,

Lo anterior debe hacerse tomando en cuenta que 2^N indica el número de estados esperados para el análisis y, por ende, la cantidad de memoria a ocupar. La ventaja es que no importa si la memoria se desborda, ya que el análisis simplemente se trunca y si sobra, no se usa.

Por ejemplo, si se indica -w26, quiere decir que usaremos 2^{26} bits (67,108,864 bits), es decir 8 Megabytes de RAM.

b). En caso de encontrar errores en la verificación, estos se almacenan en el archivo ***nombre.trail***, en donde ***nombre*** es el nombre del modelo en Promela. Este archivo puede usarse para seguir la ejecución de la verificación y encontrar posibles errores por medio de:

spin -t -p ***nombre***

Los parámetros con que se puede ejecutar el programa en Promela para tratar de seguir errores pueden verse con el comando:

spin --

c). La profundidad de búsqueda puede variarse para lograr que sea más completa por medio del parámetro -mN, por ejemplo:

pan -m100000

6. Para probar el modelo en Promela se introdujeron algunas instrucciones que simularán errores que podrían ocurrir durante la ejecución real del protocolo: caída de enlaces, procesos, transputer, etc. Esta simulación de errores nos permitió probar la tolerancia a fallas de nuestro protocolo.

6.7.2 Funcionamiento del Modelo del Protocolo

Como ya se había mencionado anteriormente, un modelo de validación no es un protocolo real, sino una simulación de su funcionamiento.

Este modelo realiza lo mismo que el protocolo en OCCAM:

1. El robot hace la primer pregunta al controlador.
2. El controlador responde.
3. El robot hace la segunda pregunta.
4. El controlador responde.
5. Repetir lo siguiente un número de veces aleatorio:
    6. El controlador envía un comando al robot.

7. El robot envía un Ack de comando ejecutado.

Sin embargo, para efectos de la validación se realizaron algunas variaciones que no modifican el correcto funcionamiento del protocolo:

1. El usuario no tiene control sobre la duración de la simulación, ya que ésta termina aleatoriamente.
2. Los mensajes consisten de una secuencia de números.
3. El Ack de correcta ejecución de un comando es un "0".

A lo largo de la compilación y verificación se hicieron algunas modificaciones para mostrar el correcto funcionamiento de todas las partes, resultando lo que se mostrará a continuación.

6.7.2.1 Ejecución directa del modelo

En la Fig 6.13 se muestra un ejemplo del funcionamiento del modelo de validación sin simulación de errores. El robot envía las preguntas iniciales al controlador, quien le responde para después generar una serie de comandos, que son contestados con un Ack por parte del robot.

```
Robot Pregunta1:
12345678910111213141516
Controlador Responde: 123456789
Robot Pregunta2:
12345678910111213
Controlador Responde: 123456789
Comando: 123456789
Ack_Comando: 0
Comando: 123456789
Ack_Comando: 0
Comando: 123456789
Ack_Comando: 0
```

```
Comando: 123456789
Ack_Comando: 0
Comando: 123456789
Ack_Comando: 0
Comando: 123456789
Ack_Comando: 0
Comando: Aplico termina
 Enlaceo termina
 I:Enlacei termina
 I:Aplici termina
 Robot termina
6 processes created
```

Fig. 6.13: Funcionamiento del modelo en Promela.

Note el orden de terminación de los procesos establecido por el protocolo: aplicación (generador de comandos), enlace, aplicación (recepción de Ack).

En la Fig. 6.14 se muestra una gráfica de ejecución del modelo generado por Promela. Las líneas verticales corresponden, de izquierda a derecha, a los procesos

de la capa de enlace (controlador -> robot), capa de enlace y capa de aplicación (robot -> controlador), c. de aplicación (controlador -> robot) y capa física.

En esta gráfica podemos observar el flujo de datos entre procesos, en particular vemos los mensajes generados al enviar los caracteres 1 y 2 de la capa física a la capa de aplicación (dirección robot -> controlador):

1. Envío del caracter 1 de la c. física a la c. de enlace (5 ! 4, 1).
2. Envío del caracter 1 de la c. de enlace a la c. de aplicación (7 ! 4, 1).
3. Envío de un Ack de la c. de aplicación a la c. de enlace (8 ! 3).
4. Envío de un Ack de la c. de enlace a la c. física (6 ! 3).
5. Se repiten los pasos anteriores para el envío del número 2.

La instrucción (5 !4, 1) que se muestra en la gráfica indica que se ha depositado un mensaje de tipo 4 en el canal 5 con el caracter 1, en el caso (6 ! 3) se está depositando un mensaje de tipo 3 en el canal 6, el cual es un mensaje de control y no lleva caracter incluido.

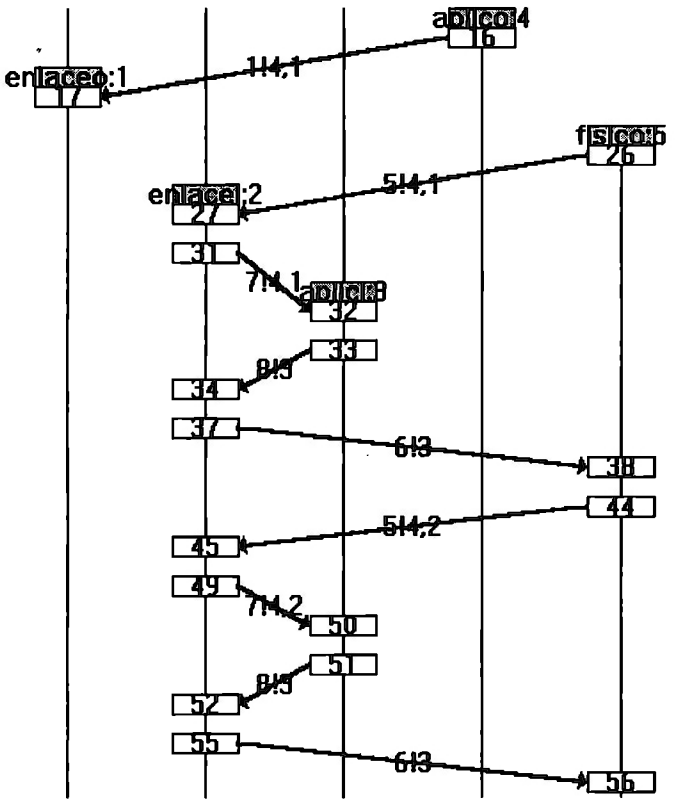


Fig. 6.14: Grafica de flujo de datos entre procesos del Modelo en Promela (usando la herramienta Xspin).

6.7.3 Predicados para la validación de los requerimientos establecidos

En esta sección mostraremos la validación de requerimientos de corrección de cada capa del protocolo, siguiendo el mismo orden en que se expusieron anteriormente en este capítulo. En cada módulo se verificará que el protocolo es resistente a cierto tipo de errores, y en la sección 6.7.4 se validará la ausencia de abrazos mortales y terminaciones impropias. El listado completo del modelo de validación puede verse en el apéndice K.

6.7.3.1 Requerimientos de Corrección en la Capa de Enlace (Robot -> Controlador)

En la sección 6.4.1.2 se definen los siguientes requerimientos de corrección:

1. *En esta capa del protocolo es muy importante tener la seguridad de que la capa de aplicación está trabajando, para lo cual es básico que conteste con un ack a cada caracter que se le envía.*

Ésto se verifica en el modelo, en la línea 17 de la Fig. 6.15. Un caracter que llega de la capa física a esta capa (línea 10), debe ser enviado a la capa de aplicación (línea 15) para después recibir un Ack.carac (línea 17) en confirmación de su recepción. En la presencia de una falla de tiempo se detecta en la línea 20 y se informa en la 21 de la misma figura.

2. *Necesitamos asegurarnos de que el ciclo de reenvío no sea infinito.*

Ésto se asegura colocando el reenvío dentro de un ciclo (líneas 13 a la 29 Fig. 6.15) con condición de ejecución intento < limite_de_intentos (línea 14).

Para validar una terminación correcta de este módulo, es decir que no existe deadlock, se usa el predicado en la línea 37 (ver Fig. 6.15), lo cual se validará en la sección 6.7.4.

En la línea 30 se recibe una petición de la capa física para finalizar, en cuyo caso se informa a la capa de aplicación (línea 31) y se finaliza (línea 33).

```
1. proctype enlacei (chan fisenl, enlfis, enlapl, aplenl)
2. {
3. byte x, intento;
4. bool continua, envia;
5.
6. continua = 1;
7. do
8. :: (continua == 1 ) ->
9.    if
10.   :: fisenl ? caracter, x ->
11.      envia = 1;
12.      intento = 1;
13.      do
14.      :: ((envia == 1) && (intento < li)) ->
15.         enlapl ! caracter, x;
16.         if
17.         :: aplenl ? ack_carac;
18.            enlfis ! ack_carac;
19.            envia = 0;
20.         :: timeout ->
21.            printf ("EI: No llega Ack.carac de AI,
reintentaré\n");
22.            intento = intento + 1;
23.         fi;
24.      :: (envia == 0) -> break;
25.      :: (intento >= li) ->
26.         enlfis ! error;
27.         break;
28.         printf("EI: error a FI\n ");
29.      od;
30.   :: fisenl ? fin ->
31.      enlapl ! fin;
32.      continua = 0;
33.      break;
```

```
34.  fi;
35.:: (continua == 0) -> break;
36.od;
37.end: printf("I:Enlacei termina\n ");
38.}
```

Fig. 6.15: Código del proceso de la capa de Enlace (Robot -> Controlador)

6.7.3.1.1 Resultados de la simulación de errores

Para probar que el protocolo funciona correctamente aún en presencia de error, se simula que no llega el Ack.carac de la capa de aplicación, lo que implica la modificación que se muestra y explica en la sección 6.7.3.2 en simulación de errores (ver Fig. 6.17, líneas 14 a la 17).

En el caso de no llegar esta confirmación, obtenemos la respuesta que mostramos en la Fig. 6.16:

```
Robot Pregunta1:
123456    timeout      timeout     EI: No llega Ack.carac de AI, reintentar,
678       timeout      timeout     EI: No llega Ack.carac de AI, reintentar,
8         timeout      timeout     EI: No llega Ack.carac de AI, reintentar,
8910      timeout ...
```

Fig. 6.16: Detección de falla de tiempo en la capa de Enlace (Robot-> Controlador)

Se puede observar que el robot está enviando su primer pregunta al controlador; los datos 1, 2, 3, 4 y 5 llegan correctamente, pues no se ve algún informe de falla de tiempo (timeout), sin embargo, al enviar el número 6 se presentan dos fallas de tiempo y el siguiente anuncio de la capa de enlace: **EI: No llega Ack.carac de AI, reintentar**,con lo cual la capa de enlace reintenta y logra la recepción de los números 6 y 7. El número 8, sin embargo, tiene que reenviarse dos veces más y así sucesivamente.

6.7.3.2 Requerimientos de Corrección en la Capa de Aplicación (Robot -> Controlador)

En la sección 6.4.2.2 se define el siguiente requerimiento de corrección:

1. *En caso de que esta capa ya no pueda (o no quiera) seguir funcionando, se informa a las dos capas de enlace. Es importante aseguramos de que estas capas han recibido el comando de fin, por medio de un caracter de reconocimiento que ellas deben enviar, llamado ack.fin.*

Le corresponde a la capa de aplicación informar al robot y a todas las capas del protocolo cuándo ha terminado, para que se pueda realizar una correcta terminación en todas las capas. El código de este módulo se muestra en la Fig. 6.17. En las líneas 9 a la 25 se encuentra el código que permite recibir información de la capa de enlace; en la línea 10 se reciben datos y en la 22 el aviso de fin de la simulación. En la línea 28 se ve el predicado que valida que no haya datos en espera de entrada a esta capa en el momento de la terminación. Finalmente, en la línea 29 se coloca el predicado que verifica la correcta terminación de éste módulo.

```
1. proctype aplici(chan enlapl, aplenl, aplapl)
2. {
3.   byte x;
4.   bool continua;
5.
6. continua = 1;
7. do
8. :: (continua == 1) ->
9.    if
10.   :: enlapl ? caracter, x ->
11.      if
12.      :: (x != 255) ->
13.         printf("%d", x);
14.         if
15.         :: aplenl ! ack_carac;
16.         :: skip
17.         fi;
18.      :: (x == 255) ->
```

```
19.        aplenl ! ack_carac;
20.        aplapl ! verde;
21.      fi;
22.    :: enlapl ? fin ->
23.      continua = 0;
24.      break;
25.   fi;
26.:: (continua == 0) -> break;
27.od;
28.assert (len(enlapl) == 0);
29.end:  printf("I:Aplici termina\n ");
30.}
```

Fig. 6.17: Código del proceso de la capa de Aplicación (Robot -> Controlador)

6.7.3.2.1 Simulación de errores

En las líneas 14 a la 17 de la Fig. 6.17, se muestra una estructura -if- que permite simular el hecho de que existan ocasiones en que no se envía el Ack.carac a la capa de enlace. Por semántica de la instrucción ***if*** en Promela (ver apéndice L), en algunas ocasiones se ejecuta el envío (instrucción 15) y en otras no (instrucción 16). En la Fig. 6.16 (sección anterior) se mostró el resultado de esta simulación.

6.7.3.3 Requerimientos de Corrección en la Capa de Aplicación (Controlador -> Robot)

En la sección 6.5.1.2 se define el siguiente requerimiento de corrección:

1. *Para que esta capa funcione, es necesario saber que esté trabajando la capa de enlace, pasando información a la capa física. Requerimos entonces una respuesta de ack cada vez que se envíe un dato.*

El código de éste módulo se muestra en la Fig. 6.18. De la línea 11 a la 50 se realiza el ciclo de envío de los caracteres de un comando a la capa de enlace así como la indicación de fin de comando (línea 16).

Después de enviar un caracter (línea 14) hay un mecanismo de recepción (líneas 18 a la 22) de cualquiera de las siguientes opciones:

- un reconocimiento (línea 19), que indica que el caracter llegó bien a la capa de enlace, o
- una falla de tiempo (timeout, línea 20), que indica que la capa de enlace no está funcionando correctamente ya que no está emitiendo reconocimiento alguno. En la línea 21 se informa: **AO: Error, capa de enlace no contesta**, como se muestra en la ejecución de la Fig. 6.19 (resultados de simulación de errores), pero se continúa el envío de datos. Esta acción puede modificarse de acuerdo a los requerimientos de la aplicación.

Con estas dos opciones aseguramos el cumplimiento del requerimiento de corrección establecido.

En las líneas 27 a la 40 de la Fig. 6.18 se espera recibir ya sea un Ack de la capa de enlace, o un aviso de error, el cual puede llegar después de que la capa de enlace haga varios intentos de comunicación con la capa física sin recibir a su vez un Ack. En caso de recibir el aviso de error, se activa la bandera ***berror*** (línea 33) y se informa (línea 43): **AO(F): Error, capa fisica no funciona**, como se muestra al final de la Fig. 6.19.

En la línea 51 del código de la Fig. 6.18 podemos observar la condición que finaliza la creación de comandos del controlador al robot. Al cumplirse esta condición, se envía una petición de fin a la capa de enlace (línea 53), para la cual se debe recibir una confirmación (línea 55). En caso de no recibir esta confirmación (línea 56) se informa (línea 57): **AO: No llega Ack.fin de EO** y se finaliza (línea 59) como se muestra en la Fig. 6.20.

En la línea 63 de la Fig. 6.18 se encuentra el predicado que valida la correcta terminación de éste módulo, llegue o no la confirmación de finalización.

```
1. proctype aplico (chan aplenl, enlapl, aplapl)
2. {
3.   byte  i;
4.   bool berror;
5.   int   vuelta, puede, sale;
6. puede = 0;
7. do /*ciclo que envia varios comandos al robot.*/
8. ::i = 0;
9.   puede = puede + 1;
10.  aplapl ? verde;
11.  do  /* ciclo que envia todos los caracteres de un comando al robot */
12.  :: (i <= max1) ->
13.    if
14.    :: (i < max1) -> i = i + 1; aplenl ! caracter, i;
15.    :: (i == max1) -> i = i + 1;
16.       aplenl ! caracter, 255;/* env¡o del 013 al robot */
17.    fi;
18.   if
19.    :: enlapl ? ack_carac;
20.    :: timeout ->
21.       printf("AO: Error, capa de enlace no contesta\n ");
22.    fi;
23.    if
24.    :: (i <= max1) ->
25.     berror = 0;
26.     sale = 0;
27.     do
28.     :: ((sale < limsale) && (berror == 0))->
29.       sale = sale + 1;
30.       if
31.       :: enlapl ? ack_carac;
32.       :: enlapl ?  error ->
33.          berror = 1;
34.          sale = limsale;
35.       :: timeout ->
36.          skip;
37.       fi;
38.     :: (sale >= limsale) ->
39.        break;
40.     od;
41.     if
42.     :: (berror == 1) ->
43.        printf("AO(F): Error, capa fisica no funciona\n ");
44.     :: (berror != 1) ->
45.       skip;
46.     fi;
47.   :: (i > max1) -> skip;
48.   fi;
```

```
49.  :: (i > max1) -> break;
50.  od;
51.:: (((vuelta == 0) && (puede > 3)) || (vuelta > licom)) ->
52.   aplapl ? verde;
53.   aplenl ! fin;
54.   if
55.   :: enlapl ? fin;
56.   :: timeout ->
57.      printf("AO: No llega Ack.fin de EO\n ");
58.   fi;
59.   break;
60.:: vuelta = vuelta + 1
61.:: vuelta = vuelta -1
62.od;
63.end:  printf("Aplico termina\n "); /* valida una terminaci¢n correcta */
64.}
```

Fig. 6.18: Proceso de la capa de Aplicación (Controlador -> Robot).

6.7.3.3.1 Resultados de la simulación de errores

1. **AO: Error, capa de enlace no contesta**: este error se genera por una falla de tiempo (ver Fig. 6.19), que indica que la capa de enlace no está funcionando correctamente ya que no está emitiendo reconocimiento alguno. La detección de este error implica simularlo en la capa de enlace, como se explicará en la siguiente sección (6.7.3.4).
2. **AO(F): Error, capa fisica no funciona**: corresponde a un aviso de error proveniente de la capa de enlace, indicando que la capa física no está funcionando, como se muestra al final de la Fig. 6.19. La recepción de este aviso de error implica simularlo en la capa física y detectarlo en la capa de enlace. La explicación de ésto se realizará en la siguiente sección (6.7.3.4).

```
Controlador Responde:
1timeout        EO: No llega Ack.carac de F, reintentar,
12      timeout         timeout         timeout         timeout
 timeout
AO: Error, capa de enlace no contesta
```

```
3timeout        timeout        timeout        timeout
EO: No llega Ack.carac de F, reintentar,
34       5      timeout
EO: No llega Ack.carac de F, reintentar,
5timeout
EO: No llega Ack.carac de F, reintentar,
5timeout
EO: No llega Ack.carac de F, reintentar,
5timeout
EO: No llega Ack.carac de F, reintentar,
timeout timeout        timeout
AO(F): Error, capa fisica no funciona
...
```

Fig. 6.19: La Capa de Aplicación (Controlador -> Robot) informa: falla de tiempo (capa de enlace no responde) y error (capa física no responde)..

3. **AO: No llega Ack.fin de EO**: Cuando se envía una petición de fin a la capa de enlace se debe recibir una contestación. La Fig. 6.20 muestra el anuncio **AO: No llega Ack.fin de EO**, que indica la falta de dicha contestación (Ack.fin) por parte de la capa de enlace. La respuesta a este error es: *El proceso Aplico termina.* Las instrucciones que permiten generar este error también se explican en la siguiente sección.

```
I:Enlacei termina
I:Aplici termina
Robot termina
timeout
AO: No llega Ack.fin de EO
Aplico termina
6 processes created
```

Fig. 6.20: Terminación del proceso de la capa de Aplicación aún sin la confirmación de otros procesos (Controlador -> Robot).

6.7.3.4 Requerimientos de Corrección en la Capa de Enlace (Controlador -> Robot)

En la sección 6.5.2.2 se definen los siguientes requerimientos de corrección para esta capa:

1. *En primer lugar necesitamos asegurar que la capa física esté recibiendo los datos que le estamos enviando.*

El cumplimiento de este requerimiento se verifica en las líneas 19 a la 25 del código de este módulo que se muestra en la Fig. 6.21, en donde se recibe la confirmación de recepción por parte de la capa física (línea 20), o no se recibe al presentarse una falla de tiempo (timeout, línea 22), lo que ocasiona un reenvío si no se ha excedido el límite establecido en la condición de la línea 28.

2. *Que el ciclo de reenvío no sea infinito*

En las líneas 16 a la 31 se muestra un ciclo que controla el reenvío de datos a la capa física. La salida de este ciclo se realiza al no cumplirse la condición de la línea 17 (máximo número de reintentos), en cuyo caso se cumplen las condiciones de las líneas 26 ó 28 y se termina el ciclo (líneas 27 ó 30) con lo que se asegura el cumplimiento de este requerimiento de corrección.

```
1. proctype enlaceo (chan aplenl, enlapl, enlfis, fisenl)
2. {
3.   byte x, intento;
4.   bool continua, envia;
5. continua = 1;
6. do
7. :: (continua == 1 ) ->
8.    if
9.    :: aplenl ? caracter, x ->
10.      if
11.      :: enlapl ! ack_carac;
12.      :: skip;
```

```
13.     fi;
14.     envia = 1;
15.     intento = 1;
16.     do /* ciclo de reenvio en caso de error en la recepcion */
17.     :: ((envia == 1) && (intento < li)) ->
18.       enlfis ! caracter, x;
19.       if
20.       :: fisenl ? ack_carac;
21.         envia = 0;
22.       :: timeout -> /* si c. fisica no est  funcionando bien*/
23.         printf("EO: No llega Ack.carac de F, reintentar,\n ");
24.         intento = intento + 1;
25.       fi;
26.     :: (envia == 0) ->
27.       break;
28.     :: (intento >= li) ->
29.       enlapl ! error;
30.       break;
31.     od;
32.   :: aplenl ? fin ->
33.     enlfis ! fin;
34.     if
35.     :: enlapl ! fin;
36.     :: skip;
37.     fi;
38.     continua = 0;
39.     break;
40.   :: timeout ->
41.     skip;
42.   fi;
43.:: (continua == 0) -> break;
44.od;
45.assert (len(fisenl) == 0);
46.end:   printf("Enlaceo termina\n "); /* valida una terminaci¢n
correcta */
47.}
```

Fig. 6.21: Código del proceso de la capa de Enlace (Controlador -> Robot).

El cuerpo principal de este módulo es un ciclo (líneas 6 a la 44) que se encarga de recibir (líneas 8 a la 42): un caracter (línea 9) o un aviso de fin (línea 32) de la capa de aplicación.

En las líneas 10 a la 13 se encuentra la estructura -*if*- que contesta un Ack de confirmación (línea 11) a la capa de enlace al recibir un caracter, o continúa sin contestar (línea 12) para simular una falla de la capa física.

En las líneas 32 a la 39 se recibe aviso de fin de la capa de Aplicación y se contesta con una confirmación (línea 35) o no se contesta (línea 36) para simular un error; en ambos casos se finaliza la ejecución (líneas 38 y 39).

Se valida la terminación correcta de éste módulo por medio de las líneas 45 y 46 de la Fig. 6.21. El predicado en la línea 45 verifica que no hay datos en espera de recepción por la capa de enlace. En la línea 46 se marca la instrucción printf como correspondiente a un estado final válido.

6.7.3.4.1 Simulación de errores y resultados de la simulación

1. Para simular que la capa de enlace tiene problemas al enviar sus datos a la capa física y realice reenvíos, es necesario que esta última no envíe su reconocimiento (Ack.carac) por lo menos cuatro veces continuas, lo cual se logra con un envío aleatorio de Ack.carac. El código que envía (línea 9, Fig. 6.22) o no (línea 10) el Ack.carac pertenece al módulo de la capa física que está simulando al robot, como se muestra en la Fig. 6.22.

```
proctype fisico (chan fisenl, enlfis, enlfiso, fisenlo)
{
1. if
2.    :: enlfiso ? caracter, x ->
3.       if
4.       :: (x == 255) ->
5.          fisenlo ! ack_carac;
6.          continua = 0;
7.       :: (x != 255 ) ->
8.          if  /* simulaci¢n de falla en emisi¢n del Ack.carac */
9.          :: fisenlo ! ack_carac;
10.         :: skip;
```

```
11.        fi;
12.        printf("%d", x);
13.     fi;
...
```

Fig. 6.22: Modificación para el envío del Ack.carac en la capa Física (Controlador->Robot)

En la línea 2 del código vemos que se recibe el caracter de la capa de enlace, al cual se contesta con una confirmación (líneas 5 y 9). En las líneas 3 a la 13 diferenciamos entre la recepción de un elemento de un comando (línea 7) o de una indicación de final de comando (línea 4). De las líneas 8 a la 11 se muestra la simulación del error. A continuación (Fig. 6.23) se muestra una parte de la ejecución en la que suceden estos errores:

```
Robot Pregunta1:
1234 ...
Controlador Responde:
...
3   4       5       timeout
EO: No llega Ack.carac de F, reintentar,
5   timeout
EO: No llega Ack.carac de F, reintentar,
5   timeout
EO: No llega Ack.carac de F, reintentar,
5   timeout
EO: No llega Ack.carac de F, reintentar,
timeout     timeout     timeout
AO(F): Error, capa fisica no funciona
...
```

Fig. 6.23: Ejecución del modelo al enviar datos con fallas de tiempo entre el Controlador y el Robot (capa de Aplicación).

En la sección 6.7.3.3, (Fig. 6.19), se explicó lo referente al aviso: *AO(F): Error, capa fisica no funciona*.

2. Para simular la falta de respuesta de la capa de enlace al recibir un caracter, se colocó la estructura en las líneas 10 a la 13 (ver Fig. 6.21) del proceso en la capa

de enlace. El resultado que presenta la capa de aplicación al no encontrar respuesta se explicó en la sección anterior (ver Fig. 6.19).

3. Cuando llega una petición de fin de la capa de aplicación se debe enviar una contestación o bien simular que no se envía, para que este error sea detectado por la capa de aplicación. Esta simulación se colocó en las líneas 34 a la 37 (ver Fig. 6.21), y el resultado que emite la capa de aplicación se mostró en la sección anterior (ver Fig. 6.20).

6.7.4 Búsqueda exhaustiva de estados inalcanzables y abrazos mortales.

De esta sección en adelante podremos ver los resultados después de haber generado (con el comando ***spin -a pro2***), compilado (con el comando ***gcc -o pan pan.c***) y ejecutado (con el comando ***pan***) el verificador. Éste contempla todas las posibilidades de ejecución del protocolo y verifica posibles errores de terminación, así como predicados (assert) que no se cumplan durante la ejecución.

Iniciamos por mostrar el caso de compilación con búsqueda exhaustiva y ejecución sin variación en los parámetros por default:

Compilación	gcc -o pan pan.c
Ejecución	pan

Al ejecutar el verificador nos da varias líneas de información como resultado, las cuales podemos dividir en las siguientes partes:

A) . Las líneas iniciales dan información general sobre el tipo de búsqueda, la versión de la herramienta usada y la existencia de algun error. Por ejemplo:

```
error: max search depth too small
pan: missing pars in receive (at depth 9882)
pan: wrote pro2.trail
(Spin Version 2.9.3 -- 5 October 1996)
Warning: Search not completed
        + Partial Order Reduction
```

El error que se muestra en las primeras dos líneas indica que la profundidad de búsqueda fue muy pequeña. Veremos cómo corregir ésto en la sección 6.7.4.2.

B). En las siguientes líneas se indica cuáles son los criterios de corrección verificados. En nuestro modelo son dos: violación de aserciones y estados finales inválidos.

```
Full statespace search for:
        never-claim         - (none specified)
        assertion violations +
        acceptance  cycles      - (not selected)
        invalid endstates   +
```

C). La linea que se muestra a continuación es una de las más importantes en la emisión de resultados:

```
State-vector 120 byte, depth reached 9999, errors: 1
```

La cual indica lo siguiente:

i). State-vector 120 byte: dice que la descripción de cada estado ocupa 120 bytes en memoria.

ii). depth reached 9999: indica la profundidad máxima alcanzada por las secuencias de ejecución, la cual debe ser menor al límite superior, que por default se restringe a 10,000 pasos. Es necesario asegurarse de que la verificación no llega a este límite ya que, de ser así, es muy probable que no se haya logrado una búsqueda completa.

iii). errors: 1, indica que el número de errores encontrados es de uno. En este ejemplo como ya se había mencionado, el error es debido a una profundidad de búsqueda pequeña.

D). En las últimas líneas vemos:

```
7935 states, stored
  1 states, matched
7936 transitions (= stored+matched)
  0 atomic steps
```

el número de estados (7,935) y transiciones generados (7,936) y analizados. Otra línea importante es:

```
(max size 2^18 states)
```

ya que indica el tamaño del espacio de estados ocupado en bits. En este caso vemos que es 2^18 bits, equivalente a 32 Kbytes de memoria, lo cual es muy poco. Por último, la siguiente línea:

```
2.10375e+06      memory usage (bytes)
```

indica la memoria usada en la validación.

Los resultados mostrados no informan de la violación de validaciones establecidas como un assert o un end. Sin embargo, dado que hubo un error en cuanto al tamaño del espacio de estados, no tenemos la seguridad de que se haya realizado una búsqueda completa, por lo que se decidió ejecutar este mismo modelo pero variando algunos parámetros.

6.7.4.1 Resultados al ejecutar el modelo cambiando el tamaño del espacio de estados (parámetro -w)

Compilación	gcc -o pan pan.c
Ejecución	pan -w21

Esta opción modifica el ancho de la tabla que contiene el espacio de estados alcanzables y debe ser mayor o igual al logaritmo base dos del número de estados generados por el verificador. Un número mayor del parámetro w permitirá que el verificador analice mas estados, lo cual puede ser útil en nuestro ejemplo.

En las ejecuciones que no usan éste parámetro el valor default es igual a 18; sin embargo, dado que en nuestro caso no se sabe cuántos estados se generan, se

probaron diferentes cantidades hasta encontrar un mejor resultado en cuanto al uso de memoria.

Al realizar pruebas encontramos que colocarlo en "-w21", es decir 2^{21} bits, aumentó el uso de memoria como se muestra:

```
9.44378e+06 memory usage (bytes)
```

Sin embargo, los demás resultados mostraron que se llegó al límite de profundidad permitido (9,999) y el número de estados y transiciones calculados no mejoró, por lo que decidimos probar variando la profundidad de búsqueda.

6.7.4.2 Resultados al ejecutar el modelo cambiando la profundidad de búsqueda (parámetro -m)

Compilación	gcc -o pan pan.c
Ejecución	pan -m100000

Por default, los verificadores tienen una profundidad de búsqueda de 10,000 pasos; si ésta no es suficiente, el programa indica que la búsqueda se truncó a 9,999 pasos. Ésto sucedió en nuestras pruebas descritas anteriormente, por lo que se decidió modificar esta profundidad para intentar una búsqueda exahustiva.

Esta ejecución reportó algunos cambios como:

```
State-vector 120 byte, depth reached 99999, errors: 1
```

La profundidad alcanzada se incrementó de 9,999 a 99,999, lo cual muestra que se requería de mayor profundidad pero, aún así, no fue suficiente para revisar todo el espacio de estados.

```
79344 states, stored
```

El número de estados ganerados aumentó de 7,935 a 79,344

```
(max size 2^18 states)
1.1358e+07      memory usage (bytes)
```

Y en las últimas líneas vemos que la utilización de memoria se incrementó pero el espacio de estados (2^18) sigue siendo muy pequeño, por lo que decidimos probar variando los dos parámetros "w" y "m".

6.7.4.3 Resultados al ejecutar el modelo cambiando el espacio de estados y la profundidad de búsqueda

Compilación	gcc -o pan pan.c
Ejecución	pan -m100000 -w21

Esta ejecución generó el mejor resultado, pues no se encontró ningún error:

```
State-vector 120 byte, depth reached 35277, errors: 0
```

La profundidad alcanzada fue de 35,277 pasos, menor a 100,000 (límite superior indicado con el parámetro "m"). Ésto muestra que se logró revisar todo el espacio de estados, por lo que no hubo errores como en las pruebas anteriores.

Los estados calculados y el hecho de que no se encontró ningún conflicto muestran que la verificación no tuvo problemas.

```
27987 states, stored
    0 states, matched
  27987 transitions (= stored+matched)
    0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^21 states)
```

6.7.5 Conclusiones sobre la validación

Al validar el protocolo, pudimos comprobar las diferencias que hay entre un lenguaje para la creación de modelos, como Promela y un lenguaje que funciona sobre un procesador como el Transputer y en forma paralela (Occam). Entre éstas podemos mencionar:

1. La sintaxis, aún cuando es muy parecida tiene sus diferencias, principalmente en el uso de las estructuras básicas de programación con que cuentan los dos lenguajes. Promela carece de algunos tipos de información e instrucciones usados en OCCAM, por lo que se tiene que buscar la forma de simularlas en Promela.
2. Los tiempos de ejecución también varían, dado que se trata de un lenguaje que solamente simula paralelismo.
3. El mecanismo de unión de los módulos es más simple en Promela que en OCCAM.

La estructura básica del protocolo que desarrollamos cuenta con dos caminos de comunicación. Cada camino cuenta con varios procesos en ejecución, los cuales fueron validados con Promela para asegurar su buen funcionamiento.

Una de las ventajas de Promela fue que nos permitió introducir errores durante la ejecución de los procesos para poder comprobar el correcto comportamiento del algoritmo ante éstos, además de la ausencia de deadlocks.

Se probó la ejecución del protocolo tomando en cuenta que se pueden presentar dos tipos de comunicación:

1. Síncrona, en el caso de enviar datos y esperar una confirmación inmediata y
2. Asíncrona, en el caso de esperar un comando del usuario o datos del robot. En este caso el protocolo espera lo necesario (no hay límite de tiempo) hasta recibir datos y realizar las acciones correspondientes a la información que está recibiendo.

En el caso de la comunicación síncrona, se simularon todos los posibles errores de sincronización (pérdidas de mensaje, caídas de proceso, etc.) y se comprobó que la respuesta obtenida es satisfactoria.

Comprobamos que el uso de Promela fue una buena opción ya que encontramos ventajas como:

1. El manejo del código es simple ya que Spin (herramienta usada para manejar Promela) permite agregar o quitar fácilmente código de programación.
2. El lenguaje es amigable ya que sus estructuras son muy parecidas a las que usan los lenguajes C y OCCAM, lo cual facilita la creación del modelo.
3. Encontramos que Promela es un lenguaje robusto, ya que permite la creación de modelos completos y pone a nuestra disposición las herramientas que lo validan. Además, la validación del modelos es completa cuando existe memoria suficiente. En caso de encontrar errores, Spin ayuda a su corrección.
4. Spin es un paquete de dominio público que se ha desarrollado para ejecutarse bajo distintos sistemas operativos, como Unix y Windows 95.

De igual forma podemos mencionar algunas desventajas:

1. Como el lenguaje de validación y el de implementación (Promela y OCCAM) son diferentes en cuanto a su sintaxis y manejo de sus estructuras de programación (a pesar de su similitud semántica), pudimos observar que no es automático el generar el programa en OCCAM partiendo del modelo en Promela.
2. OCCAM cuenta con algunas instrucciones y herramientas que no provee Promela en forma directa, como la instrucción ALT, la reinicialización de canales, etc., las cuales tuvimos que simular.

7. CONCLUSIONES Y TRABAJO FUTURO

7.1 Conclusiones

Al poner en uso nuestra arquitectura basada en transputers en el CIM del Campus Toluca, pudimos observar la confiabilidad en el intercambio de mensajes entre robots y controlador. Esto nos ayudó a comprobar el correcto funcionamiento tanto de la arquitectura como del protocolo de comunicación, el cual también fué validado en Promela.

Se mostró que el uso del transputer para el diseño del controlador de celda fue una buena decisición por varias razones, entre las cuales podemos mencionar:

1. La velocidad de procesamiento y su capacidad de paralelismo permite la creación de tantos procesos como sean necesarios para su ejecución, dando respuestas rápidas en tiempo real.
2. El lenguaje del transputer tiene la capacidad suficiente para nuestros requerimientos en la creación de programas, ya que cuenta con un conjunto de instrucciones de alto nivel, y permite un contacto directo con la arquitectura del transputer por medio de instrucciones adicionales como las instrucciones de paralelismo, las de intercambio de mensajes por medio de los enlaces del transputer, etc.

Una filosofía incorrecta en cuanto a las funciones básicas del protocolo podría degradar esta capacidad computacional con que se cuenta. La implementación de un protocolo tipo "*handshake*" da la simpleza de ejecución necesaria para garantizar respuestas rápidas en ambos sentidos, así como la confiabilidad requerida en el sistema de comunicación. Además, contar con una biblioteca en OCCAM para la administración de enlaces entre transputers (como *"xlink.lib"*, usada en esta tesis) fue básico para detectar y prevenir fallas que pueden detener la ejecución del protocolo en forma inesperada.

Este protocolo se dividió en dos sentidos de comunicación concurrentes: cuando el controlador envía información hacia el robot y el otro en sentido inverso. Nos pareció conveniente emplear esta técnica en nuestro protocolo por lo siguiente:

1. Permite una ejecución de tareas concurrentes en ambos sentidos. La concurrencia en OCCAM es implementada fácil y eficientemente por el transputer.
2. Al contar con módulos independientes con una función bien definida, es muy sencillo colocarlos en diferentes unidades de procesamiento (especialmente en el caso de la capa de aplicación) sin realizar modificaciones a éstos. La arquitectura usada cuenta con las herramientas necesarias para colocar los módulos en forma distribuida y comunicarlos.

Gracias a la naturaleza de los enlaces de comunicación del Transputer, nuestro protocolo no realiza las tareas de detección de errores en la integridad de los mensajes intercambiados, ya que los enlaces garantizan que no hay errores de este tipo. Nuestro protocolo, sin embargo, sí se encarga de asegurar que cada proceso esté realizando sus funciones adecuadamente.

Cuando se envía un dato a otro módulo se espera una confirmación (Ack) para asegurar que éste se recibió. En caso contrario, se genera un timeout (falla de tiempo).

Al diseñar el protocolo encontramos de gran utilidad el uso de Promela para crear un modelo del mismo que pudiera probarse y sugerir correcciones. Lo más importante en la creación de éste es la posibilidad que ofreció Promela de probarlo para asegurar que cumple con los requerimientos de corrección establecidos y que carece de errores como abrazos mortales, terminaciones impropias y esperas

infinitas. SPIN, que es el sistema de validación de potocolos que usa Promela, generó el código para realizar todas las posibles ejecuciones del modelo y asegurar que éste no tiene errores, resultados que se muestran al final del capítulo 6.

7.2 Trabajo Futuro

Trabajar bajo el contexto de un proyecto tan amplio como el de Conacyt-NSF [1], nos llevó a un buen nivel de investigación y al mismo tiempo difícil de delimitar, dados los requerimientos del proyecto. El área de trabajo es muy amplia y pueden proponerse varios trabajos de investigación con especialización en diferentes tópicos.

Tomando como base el protocolo desarrollado y la infraestructura de transputers con que cuenta el ITESM Campus Estado de México [13], podríamos visualizar y proponer como trabajo futuro la expansión y prueba de nuestro protocolo sobre una red de tamaño mediano y con la ejecución de varios procesos (quizá hasta redundantes, para tolerar fallas) en forma concurrente. Ésto implica la planeación de rutas de comunicación y la administración de la red, así como el diseño de un sistema que se enfoque a la comunicación y tolerancia a fallas.

De lo anterior podríamos proponer la formación de un equipo de trabajo orientado a las siguientes partes medulares:

1. Generación del modelo del protocolo de red, incluyendo tolerancia a fallas, y su validación rigurosa usando un lenguaje como Promela. Se propone un uso más extensivo de las herramientas con que cuenta Promela, tales como Spin y Xspin para la Generación y Validación de modelos. Spin fue la herramienta más usada en esta tesis, mientras que Xspin se utilizó para generar la gráfica de la Fig. 6.14 y realizar las ejecuciones del modelo.
2. Proponemos también que se estudien algunas de las facilidades de software dadas por los fabricantes de Transputers, como los canales virtuales de

comunicación administrados automáticamente, ayuda considerable en la administración de la comunicación y la tolerancia a fallas.

3. En conjunto con las investigaciones anteriores considero importante orientarse también al estudio de las posibles fallas que podría presentar dicho sistema para dar pauta a un diseño e implementación robustos.
4. Hablando del protocolo, creo que podrían desarrollarse procesos monitores que ayuden también a la detección y corrección de fallas en la comunicación de procesos. En particular, podríamos recordar un problema analizado en el capítulo 6: cuando un proceso emisor envía datos a un receptor de manera asíncrona, prodríamos prevenir fallas de comunicación implementando un proceso "*buffer*" que se encargue de recibir los datos y de enviarlos al receptor y en caso de falla se reporte a un proceso supervisor. De esta manera los bloqueos por fallas serían sólo en estos procesos, lo que facilitaría la labor de monitoreo.

En las propuestas anteriores habría que tomar en cuenta la importancia de la retroalimentación dentro del equipo, ya que los resultados parciales de la misma investigación podrían proponer nuevas modificaciones y mejoras a cada una de las partes.

8. BIBLIOGRAFÍA

[1] Jesús Sánchez et. al. Programa Conacyt-NSF. "TECNOLOGÍAS DE COMUNICACIÓN AVANZADAS EN ROBÓTICA Y CELDAS FLEXIBLES DE MANUFACTURA". Proyecto en el que participan el ITESM Campus Edo. de México-Campus Morelos y "UNIVERSITY OF TEXAS" en Austin. Enero 1996.

[2] Jesus Sánchez. "A flexible manufacturing cell distributed controller". 4th International conference on Applied Corporate Computing. Monterrey 1996.

[3] Peter Gray. "A FLEXIBLE MANUFACTURING CELL DESIGNED WITH PETRI NETS FOR TRANSPUTER CONTROL". Base de Datos IPO.

[4] J. Alarcón C. "Diseño de una arquitectura reconfigurable basada en Transputers". Tesis de maestría en Ciencias Computacionales, Jaime Alarcón Celis, ITESM Campus Estado de México y Campus Toluca, 1997.

[5] A. Bottle H. "Módulo de Interface programable". Tesis de maestría en Ciencias Computacionales presentada por Raúl Armando Bottle Hernández en el ITESM Campus Estado de México, 1992.

[6] INMOS. "OCCAM 2 Toolset. User Manual". CSA Computer System Architectures. Abril 1989.

[7] INMOS. A. E. Gore. "TRANSPUTER EDUCATION KIT. Occam and the Transputer. A Workbook". CSA Computer System Architects. 1990.

[8] Página en Internet "SGS-THOMSON MICROELECTRONICS" en la Red Internet. Dirección http://www.st.com/.

[9] AMATROL. "COMPUTER INTEGRATED MANUFACTURING LABORATORY MANUAL". Amatrol. Segunda Edición.

[10] "4 SLOT PC/ISA TRAM MOTHERBOARD". Engineering Specification. Información integrada a la tarjeta SMT004A.

[11] INMOS. "IMS T800 transputer". INMOS Group.

[12] Gerard J. Holzmann. "DESIGN AND VALIDATION OF COMPUTER PROTOCOLS. Prentice Hall Software Series. Primera Edición.

[13] M. Salmerón. "Diseño de la arquitectura de Hardware utilizando Transputers para el control de una celda de Manufactura Flexible". Propuesta de tesis en el ITESM CEM. 1996.

[14] A. Valenzano, C. Demartini, L. Ciminiera. "MAP AND TOP COMUNICATIONS Standars and Applications". Primera Edición.

[15] I. Scott MacKenzie. "THE 8051 MICROCONTROLLER". Prentice Hall. Segunda Edición.

[16] INMOS. "Comunicating processes architecture". INMOS Limited Prentice Hall International 1988.

[17] INMOS. "TRANSPUTER INSTRUCTION SET. A compiler writer's guide ". INMOS Limites Prentice Hall 1988.

[18] J. Alarcón, M. Rodríguez, S. Mota. "Diseño de una Arquitectura Reconfigurable basada en Transputers". 2° Congreso Nacional sobre Electrónica y Comunicaciones. Colegio de Ingenieros en Comunicaciones y Electrónica. EXPO-CICE '96.

[19] J. Alarcón, M. Rodríguez, S. Mota. "Arquitectura Reconfigurable basada en Transputers para el control de una CFM". 6° Simposium Internacional de sistemas Computacionales y Electrónicos. ITESM Campus Toluca 1996.

[20] Reza S. Raji. "SMART NETWORKS FOR CONTROL". Echelon Corp. IEEE Spectrum Junio 1994. pp. 49-55.

[21] J. N. Daigh. "Comunications for Manufacturing: An Overview". J. N. Daigh, A. Seidman, J.R. Pimente. IEEE Network. May 1988. Vol. 2, No. 3.

[22] Colin H. West. "PROTOCOL VALIDATION. PRINCIPLES AND APPLICATIONS". IBM Zurich Research Laboratory. Elsevier Science Publishers B.V.

[23] Chung-Ming Huang, "Integrated FDT-based protocol verification system". IEEE Software Engineering Journal. Noviembre, 1995. pp. 233-244.

[24] Victor G. García, E. Velázquez, J. Vinyes, "Modelling and formal specification of the Real-Time Manufacturing Automation Protocol". IEEE 1994. 0-7803-1772-6/94. pp. 1197-1200.

[25] T. Braun, C. Schmidt, "Implementation of a Parallel Transport Subsystem on a Multiprocessor Architecture". Institute of Telematics, University of Karlsruhe. Germany. IEEE 1993. 0-8186-3900-8/93. pp. 76-83

[26] G. J. Holzmann, "What's New in SPIN Version 2.0 (Updated for SPIN Version 2.9, June 10, 1996)". Bell Laboratories, Murray Hill, New Jersey 07974. Agosto, 1996.

[27] J. Hajek, "Automatically verified data transfer protocols", Proc. 4th ICCC, 1978, Kyoto, pp. 749-756

[28] C. H. West, "General Technique for Communications Protocol Validation", IBM Journal of Research and Development, Vol 22, No. 4, p 393, 1978.

[29] G. J. Holzmann, R. A. Beukers, "The PANDORA Protocol Development System", Proc. 3rd Int Conf. on Protocol Specification, Testing, and Verification, INWG/IFIP, Eds. H. Rudin and C. West, pp. 357-369, North Holland Publ. Co., June 1983.

[30] G. J. Holzmann, "Tracing Protocols", AT&T Techn. Journal, Vol 64, No. 10, Dec. 1985.

[31] G. J. Holzmann, "An improved reachability analysis technique", Software Practice and Experience, Vol 18, No. 2, pp. 137-161, Feb. 1988

[32] G. J. Holzmann, "An analysis of bitstate searching", Proc. IFIP/WG6.1 Symp. on Protocols Specification, Testing, and Verification, PSTV95, Warsaw, Poland, June 1995.

[33] G. J. Holzmann, J. Patti, "Validating SDL Specifications: An Experiment", Proc. 9th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP, De. C. Vissers and E. Brinksma, Twente, Neth., June, 1989

[34] D. A. Peled, "Combining partil order reductions with on-the-fly model checking", Proc. 6th Int. Conf. on Computer Aided Verification, CAV94, Stanford, Ca., June 1994.

[35] G. J. Holzmann and D. Peled "An improvement in formal verification", Proc. 7th Int. Conf. on Formal Description Thechniques, FORTE94, Berne, Switzerland. October 1994

[36] R. Gerth, D. Peled, M. Vardi, P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic", Proc. PSTV95 Conference, Warsaw Poland, 1995

[37] C. A. Sunshine, "Interprocess communication protocols for computer networks", Ph. D. Thesis, Computer Science Department, Stanford University, 1975

[38] H. Rudin, C. H. West y P. Zafiropulo, "Automated protocol validation - on the chain of development", Proc. Computer Networks Symposium. University of Liege, Belgium, Febrero 1978

[39] G. V. Bochmann, "Finite state description of communication protocols, Proc. Computer Networks Symposium, University of Liege, Belgium, Febrero 1978

[40] J. Hajek. Automatically verified data data transfer protocols", Proc. International Conference on Computer Communications, Hyoto, Japón, 1978, pp. 749-756

[41] Mullender, Sape. "Distributed Systems". Addison Wesley, 2nd. Edition (1993).

[42] L. Trejo. "Diseño y Validación de Protocolos de Comunicación". Notas del curso. Maestría en Ciencias Computacionales ITESM-CEM Enero 1997.

9. APÉNDICE A: PROGRAMA DEL PROTOCOLO QUE ENVÍA COMANDOS AL ROBOT

9.1 DECLARACIÓN DE PROTOCOLOS OCCAM PARA LOS CANALES

```
-- Pro10
-- Protoc1.inc
PROTOCOL ENTERO
  CASE
    nume; INT
    ack; INT
    final
:
PROTOCOL BITE IS BYTE :
```

9.2 PROGRAMA EN CAPA DE APLICACIÓN

```
-- Pro10
-- Liga2.occ
#INCLUDE "hostio.inc"
#INCLUDE "protoc1.inc"
PROC proceso.2 (CHAN OF SP ffs, tts, CHAN OF ENTERO p2.a.p1, p1.a.p2)
  #USE "hostio.lib"

  BOOL rerr, sigue :
  INT numero, numer :
  BYTE caracter :
  SEQ
    sigue := TRUE
    so.write.nl (ffs, tts)
    WHILE sigue
      SEQ
        p1.a.p2 ? CASE
          nume; numer
            SEQ
              caracter := BYTE numer
              so.write.char (ffs, tts, caracter)
              IF
                (numer = 13)
                  so.write.nl (ffs, tts)
                TRUE
                  SKIP
              -- fin del IF
              p2.a.p1 ! ack; numer
          final
            SEQ
              sigue := FALSE
```

```
          so.write.nl (ffs, tts)
          so.write.string (ffs, tts, "FINAL")
          so.write.nl (ffs, tts)
  so.exit (ffs, tts, sps.success )
```

9.3 PROGRAMA EN CAPA DE ENLACE

```
-- Pro10
-- Liga1.occ
#INCLUDE "hostio.inc"
#INCLUDE "protoc1.inc"
PROC proceso.1 (CHAN OF ENTERO p2.a.p1, p1.a.p2, CHAN OF BITE p1.a.l2, l2.a.p1)
 #USE "hostio.lib"
 INT numero, numer :
 BOOL sigue :
 BYTE sale, entra :
 VAL salida IS 27 :
 SEQ
  sigue := TRUE
  WHILE sigue
   SEQ
    l2.a.p1 ? entra
    numero := INT entra
    IF
     (numero = salida)
      SEQ
       p1.a.p2 ! nume; numero
       p2.a.p1 ? CASE
        ack; numer
         SKIP
       p1.a.p2 ! final
       sigue := FALSE
     (numero <> salida)
      SEQ
       p1.a.p2 ! nume; numero
       p2.a.p1 ? CASE
        ack; numer
         SKIP
    -- fin del IF

    -- instrucciones para que solo reciba un numero y termine
   -- fin del SEQ del WHILE
```

9.4 CONFIGURACIÓN DE LOS PROCESOS

```
-- Pro10
-- Confliga.pgm
#INCLUDE "hostio.inc"
#INCLUDE "protoc1.inc"
```

```
#INCLUDE "linkaddr.inc"
#USE "liga1.c8h"
#USE "liga2.c8h"
CHAN OF SP from.host, to.host :
CHAN OF BITE l2.a.p1, p1.a.l2 :
PLACED PAR
  PROCESSOR 0 T800
    PLACE from.host AT link0.in:
    PLACE to.host AT link0.out :
    PLACE p1.a.l2 AT link2.out :
    PLACE l2.a.p1 AT link2.in :
    CHAN OF ENTERO p2.a.p1, p1.a.p2 :
    PAR
      proceso.2 (from.host, to.host, p2.a.p1, p1.a.p2)
      proceso.1 (p2.a.p1, p1.a.p2, p1.a.l2, l2.a.p1)
```

10. APÉNDICE B: PROGRAMA DEL PROTOCOLO QUE RECIBE CARACTERES DEL ROBOT

10.1 DECLARACIÓN DE PROTOCOLOS OCCAM PARA LOS CANALES

```
-- Pro14
-- proto.inc
PROTOCOL A.E
  CASE
    comando; BYTE::[]BYTE
    final.a.e
:
PROTOCOL E.A
  CASE
    ack.com
    err.com
    ack.comando.e.a
    err.comando.e.a
    ack.com.inc.e.a
    err.com.inc.e.a
    err.com.trunc.e.a
    final.e.a
:
PROTOCOL E.F
  CASE
    carac; BYTE
    final.e.f
    cancela
:
PROTOCOL F.E
  CASE
    ack.carac
    err.carac
    ack.comando.f.e
    err.comando.f.e
    final.f.e
:
PROTOCOL F.L IS BYTE :
PROTOCOL L.F IS BYTE :
```

10.2 PROGRAMA EN CAPA DE ENLACE

```
-- Pro14
-- enlace.occ
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
PROC enlace (CHAN OF A.E a.a.e, CHAN OF E.A e.a.a, CHAN OF E.F e.a.f, CHAN OF F.E
f.a.e)
  #USE "hostio.lib"
  BOOL sigue, mas, continua :
```

```
BYTE largo, caracter :
INT longitud, intento1, intento2, i :
[255]BYTE linea :
SEQ
 sigue := TRUE
 WHILE sigue
  SEQ
   a.a.e ? CASE
    comando; largo::linea
     SEQ
      e.a.a ! ack.com
      longitud := INT largo
      mas := TRUE
      i:= 0
      intento1 := 0
      intento2 := 0
      continua := FALSE
      caracter := linea[i]
      WHILE mas              -- envio del comando por partes a F
       SEQ
        e.a.f ! carac; caracter

        f.a.e ? CASE         -- esp contestacion de  F
         ack.carac           -- espero ack.carac de F
          SKIP
         err.comando.f.e
          SEQ
           IF
            (intento1 <= 4)
             SEQ
              e.a.f ! cancela
              i := -1
              intento1 := intento1 + 1
            (intento1 > 4)
             SEQ
              i := i - 1
              e.a.a ! err.com.inc.e.a
              mas := FALSE     -- no reenviar el comando
         ack.comando.f.e
          SEQ
           i := i - 1
           e.a.a ! ack.com.inc.e.a
           mas := FALSE     -- no reenviar el comando
         err.carac
          SEQ
           IF
            (intento2 <= 4)
             SEQ
              i := i - 1
              intento2 := intento2 + 1
```

```
            (intento2 > 4)
              SEQ
                i := i - 1
                e.a.f ! cancela
                e.a.a ! err.com.trunc.e.a
                mas := FALSE     -- no reenviar el comando
          final.f.e
            SEQ
              e.a.a ! final.e.a
              i := i - 1
              mas := FALSE     -- no reenviar el comando
              sigue := FALSE
        -- fin del CASE
        i := i + 1
        IF
          ( i = longitud)
            SEQ
              caracter := 13 (BYTE)
              continua := TRUE
          ( i > longitud)
            SEQ
              mas := FALSE
          ( i < longitud )
            SEQ
              caracter := linea[i]
        -- fin del IF
      -- fin del SEQ del WHILE
    -- fin del WHILE para enviar el comando
    IF
      (continua = TRUE)
        SEQ
          f.a.e ? CASE
            ack.comando.f.e
              SEQ
                e.a.a ! ack.comando.e.a
              -- fin del SEQ
            ack.carac
              SKIP
            err.comando.f.e
              SKIP
            final.f.e
              e.a.a ! final.e.a
          -- fin del CASE
        -- fin del SEQ
      TRUE
        SKIP
    -- fin del IF
  -- fin del SEQ
```

```
      final.a.e -- Aplicaci¢[2]n quiere terminar
        SEQ
          e.a.a ! final.e.a
          e.a.f ! final.e.f
          sigue := FALSE
      -- fin del CASE
```

10.3 PROGRAMA EN CAPA DE APLICACIÓN (COMUNICACIÓN CON LA PC)

```
-- Pro 14
-- aplic.occ
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
PROC aplic (CHAN OF SP fs, ts, CHAN OF A.E a.a.e, CHAN OF E.A e.a.a)
 #USE "hostio.lib"
 BOOL rerr, sigue :
 BYTE char, result, largo :
 INT longitud :
 [255]BYTE linea :
 SEQ
  sigue := TRUE
  WHILE sigue
   SEQ
    so.write.string (fs, ts, "Comando: ")
    so.read.echo.line (fs, ts, longitud, linea, result)

    largo := BYTE longitud
    a.a.e ! comando; largo::linea

    e.a.a ? CASE   -- espera ack de comando recibido
     err.com      -- no sucede
       SKIP
     ack.com      -- esto esperamos
       SEQ
         so.write.string.nl (fs, ts, "ack_comando recibido")
       -- fin de SEQ
     final.e.a -- never
       SKIP
    -- fin de CASE
    e.a.a ? CASE
     err.com -- never
       SKIP
     ack.comando.e.a    -- esto esperamos
       SEQ
         so.write.string.nl (fs, ts, "ack_comando bien ejecutado")
       -- fin de SEQ
```

[2] Las palabras acentuadas dentro de un programa aparecerán como se muestra.

```
      final.e.a -- never
        SEQ
          so.write.string.nl (fs, ts, "E termino A -> E -> L2. E -> A")
    -- fin de CASE
    IF
      (linea[0] = '\')
        SEQ
          a.a.e ! final.a.e
          sigue := FALSE
      TRUE
        SKIP
    -- fin del IF
  -- fin SEQ

  e.a.a ? CASE
    err.com          -- never
      SKIP
    ack.comando.e.a    -- never
      SEQ
        so.write.string.nl (fs, ts, "ack_comando")
      -- fin de SEQ
    final.e.a        -- esto esperamos
      SEQ
        so.write.string.nl (fs, ts, "E termino A -> E -> L2. E -> A")
  -- fin de CASE

  so.exit (fs, ts, sps.success )
```

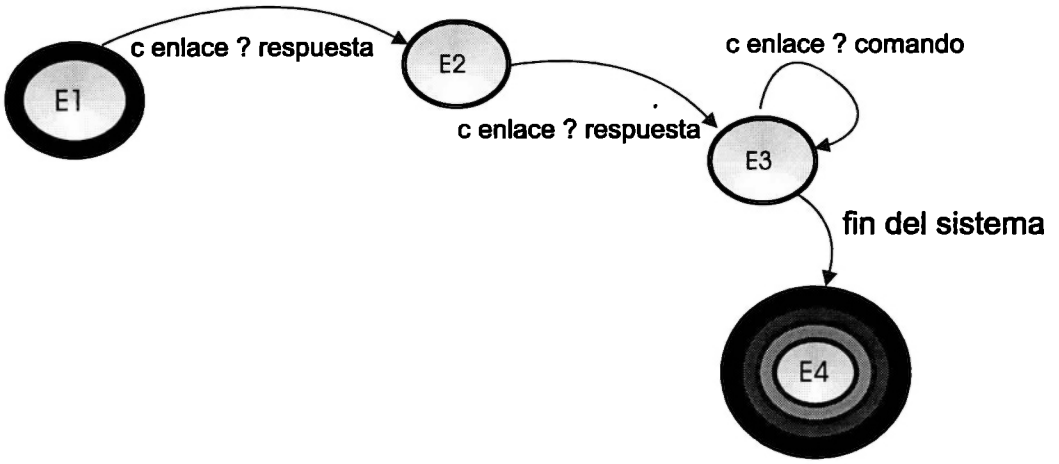
10.4 CONFIGURACIÓN DE LOS PROCESOS

```
-- Pro14
-- conf.pgm
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
#INCLUDE "linkaddr.inc"
#USE "enlace.c8h"
#USE "aplic.c8h"
#USE "fisico.c8h"
CHAN OF SP from.host, to.host :
CHAN OF F.L f.a.l2 :
PLACED PAR
  PROCESSOR 0 T800
    PLACE from.host AT link0.in:
    PLACE to.host AT link0.out :
    PLACE f.a.l2 AT link2.out :
    CHAN OF A.E a.a.e :
    CHAN OF E.A e.a.a :
    CHAN OF E.F e.a.f :
    CHAN OF F.E f.a.e :
    PAR
```

aplic (from.host, to.host, a.a.e, e.a.a)
enlace (a.a.e, e.a.a, e.a.f, f.a.e)
fisico (e.a.f, f.a.e, f.a.l2)

11. APÉNDICE C: SIMULACIÓN DEL ROBOT (ROBOT <-> CONTROLADOR)

11.1 AUTÓMATA



11.2 Vocabulario y formato de mensajes

El intercambio de mensajes entre el robot y el controlador es por medio de secuencias de caracteres (bytes) terminadas por un ENTER. Para enviar un comando al Robot se envían los caracteres que lo conforman (p.e. do move movim1) y en caso de recibir respuesta, también es en forma de bytes que representan un mensaje de status, alguna indicación del Robot, alguna pregunta, etc. (p.e. "Initalize (Y/N) ?").

Mensaje para el envío de caracteres

BYTE :

Generalmente lo único que se envía es un byte representando un caracter.

Mensaje para la recepción de caracteres

BYTE :

Un byte representando un caracter o código recibido.

11.3 Medio Ambiente de desarrollo

Este proceso de simulación se realizó en OCCAM y la comunicación se realiza por medio de canales de éste lenguaje (ver sección 5.1).

Los canales pueden ser internos a un transputer en caso de ejecutar todos los procesos en él. Sin embargo, es posible la ejecución del simulador en diferentes transputers. En este caso los canales se implementan por medio de los enlaces de los transputers.

11.4 Máquina de Estados

$M1 = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ E1, E2, E3, E4 \}$

E1: Se envía la primer pregunta al controlador y se espera su respuesta.

E2: Se envía la segunda pregunta al controlador y se espera su respuesta.

E3: Se espera un comando y se le contesta un ack de ejecución (.).

E4: Estado Final

$\Sigma = \{$ respuesta, comando, . $\}$

$F = \{ E4 \}$

Tabla de Transiciones

	respuesta. aplic	*comando. sistema*	*Fin de sistema*
E1	E2	Null	Null
E2	E3	Null	Null
E3	Null	E3	E4

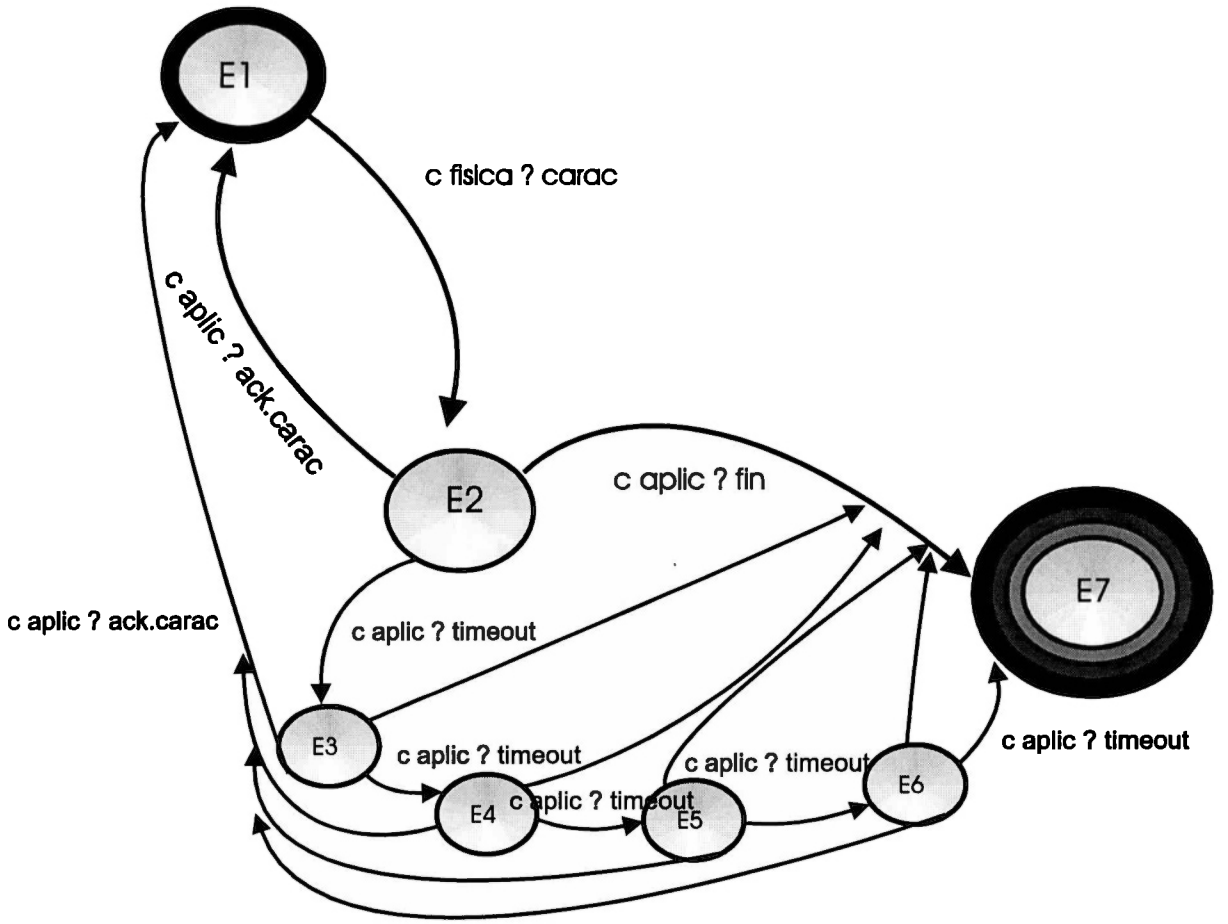
11.5 CODIGO DEL MÓDULO QUE SIMULA AL ROBOT

```
-- prorob3
-- robot.occ
#INCLUDE       "hostio.inc"
#INCLUDE       "proto.inc"

PROC robot (  CHAN OF ANY e.a.r,
              CHAN OF BYTE r.a.e )


 #USE          "hostio.lib"
 #INCLUDE      "const.inc"
 BOOL          sigue :
 BYTE          caracter :
 INT           longitud,
               vuelta :
 [31]BYTE      cadena :
 [19]BYTE      cadenac :

 SEQ
  vuelta := 0
  cadena := "Load VALII from floppy (Y/N) ? "
  longitud := 31
  WHILE (vuelta < longitud)
    SEQ
      r.a.e ! cadena[vuelta]      -- envio caracter salir E
      vuelta := vuelta + 1
    -- fin SEQ WHILE (vuelta ...
  -- fin WHILE (vuelta < longitud)
  r.a.e ! nl      -- envio caracter salir E

  sigue := TRUE
  WHILE sigue
    SEQ
      e.a.r ? caracter
      IF
        (caracter = nl)
          SEQ
            sigue := FALSE
        TRUE
          SKIP
      -- fin del IF
    -- fin de SEQ del WHILE
  -- fin del WHILE sigue

  vuelta := 0
  cadenac := "Initialize (Y/N) ? "
  longitud := 19
```

```
    WHILE (vuelta < longitud)
      SEQ
        r.a.e ! cadenac[vuelta]      -- envio caracter salir E
        vuelta := vuelta + 1
      -- fin SEQ WHILE (vuelta ...
    -- fin WHILE (vuelta < longitud)
    r.a.e ! nl      -- envio caracter salir E

    sigue := TRUE
    WHILE sigue
      SEQ
        e.a.r ? caracter
        IF
          (caracter = nl)
            r.a.e ! '.'
          TRUE
            SKIP
        -- fin del IF
      -- fin de SEQ del WHILE
    -- fin del WHILE sigue
  -- fin del SEQ
:
```

12. APÉNDICE D: PROCESO DE LA CAPA DE ENLACE (ROBOT -> CONTROLADOR)

12.1 AUTÓMATA



12.2 Vocabulario y formato de mensajes

Mensajes de Robot -> Enlace

BYTE :

Mensajes de Enlace -> Aplicación

1. *carac.e.ai;* **BYTE**
2. *final.e.ai*

Mensajes de Aplicación -> Enlace

1. *ACK.carac.e.ai*
2. *final.a.ei*

12.3 Medio Ambiente de desarrollo

La comunicación entre los procesos de la capa de simulación del robot, capa de enlace y capa de aplicación se realiza por medio de registros internos al transputer, lo que hace que el intercambio de información entre capas sea rápido.

12.4 Máquina de Estados

$M2 = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ E1, E2, E3, E4, E5, E6, E7 \}$

E1: Recepción de caracteres del Robot

E2: Envío de los caracteres a la capa de aplicación y espera de un ACK de recepción de caracter, falla de tiempo (timeout) o fin.

Los siguientes cuatro estados corresponden al mecanismo de reenvío en caso de que la capa de aplicación no esté contestando a la recepción de caracteres.

E3: Primer reenvío.

E4: Segundo reenvío.

E5: Tercer reenvío.

E6: Cuarto y último reenvío.

E7: Estado Final

Σ = { carac, ack.carac, fin, timeout }

F = { E7 }

Tabla de Transiciones

	carac	*ack.carac*	*fin*	*timeout*
E1	E2	Null	Null	Null
E2	Null	E1	E7	E3
E3	Null	E1	E7	E4
E4	Null	E1	E7	E5
E5	Null	E1	E7	E6
E6	Null	E1	E7	E7

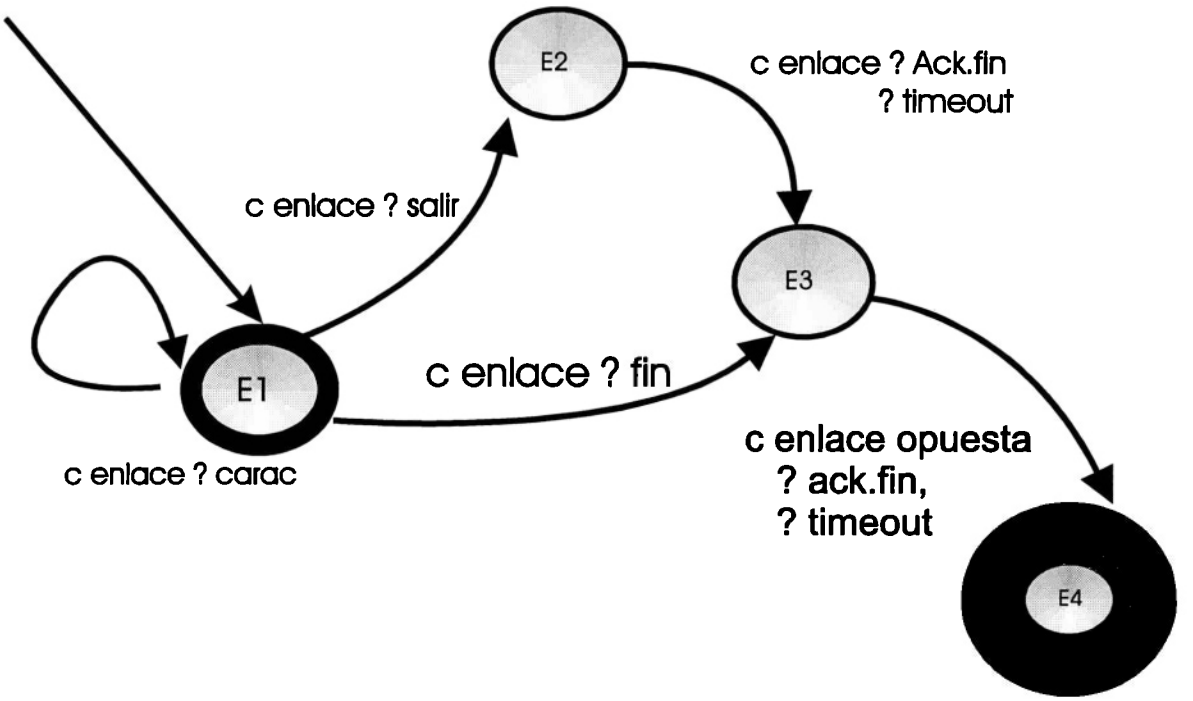
12.5 CODIGO DEL PROGRAMA

```
-- prorob3
-- enlacei.occ
#INCLUDE      "hostio.inc"
#INCLUDE      "proto.inc"

PROC enlacei ( CHAN OF A.EI a.a.ei, CHAN OF E.AI e.a.ai,
               CHAN OF BYTE r.a.e)

 #USE         "hostio.lib"
 #INCLUDE     "const.inc"
 BOOL         sigue :

 SEQ
   sigue := TRUE
   WHILE sigue
     BYTE       caracter :
     TIMER      reloj :
     INT        hora,
                vuelta :
     SEQ
       r.a.e ? caracter
       vuelta := 0
       WHILE (vuelta < limvuel)
         SEQ
           e.a.ai ! carac.e.ai; caracter      -- envio caracter A
           reloj ? hora
           ALT
             a.a.ei ? CASE                    -- recibo ack A
```

```
            ack.carac.e.ai
              SEQ
                vuelta := limvuel
            final.a.ei
              SEQ
                e.a.ai ! final.e.ai
                sigue := FALSE
                vuelta := limvuel
          -- fin CASE
          reloj ? AFTER (hora PLUS timeout)
            SEQ
              vuelta := vuelta + 1
              IF
                (vuelta >= limvuel)
                  sigue := FALSE
                TRUE
                  SKIP
              -- fin IF
            -- fin SEQ
        -- fin del reloj ...
      -- fin del ALT
    -- fin del SEQ WHILE
  -- fin WHILE (vuelta < limvuel)
-- fin SEQ del WHILE sigue
-- fin del WHILE sigue
-- fin del SEQ del procedimiento
:
```

13. APÉNDICE E: PROCESO DE LA CAPA DE APLICACIÓN (ROBOT -> CONTROLADOR)

13.1 AUTOMATA



13.2 Vocabulario y formato de mensajes

Mensajes de Aplicación -> Enlace

1. *ack.carac.e.ai*
2. *final.a.ei*

Mensajes de Enlace -> Aplicación

1. *carac.e.ai;* **BYTE**
2. *final.e.ai*

13.3 Medio Ambiente de desarrollo

La comunicación entre estos procesos se realiza por medio de los canales OCCAM declarados en los procesos involucrados.

13.4 Máquina de Estados

$M3 = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ E1, E2, E3, E4 \}$

E1: Estado de recepción de caracteres y envío del ACK.carac a la capa de enlace mientras no reciba un caracter de fin de la capa de enlace (pasando al estado E3) o de salir del robot (pasando al estado E2).

E2: Cuando el robot solicita finalizar, en este estado se informa a la capa de enlace que nos está dando servicio directamente (Robot->Controlador), se espera su confirmación de Ack.fin y pasa al estado E3.

E3: Si la capa de enlace solicita salir, en este estado se informa a la capa de enlace opuesta (Controlador -> Robot) y se espera su confirmación Ack.fin.

E4: Estado Final

$\Sigma = \{$ cacarac, fin, salir (ESC), ack.fin $\}$

$F = \{ E4 \}$

Tabla de Transiciones

	carac	*fin*	*salir (ESC)*	*ack.fin*
E1	E1	E3	E2	Null
E2	Null	Null	Null	E3
E3	Null	Null	Null	E4

13.5 CÓDIGO DEL PROGRAMA

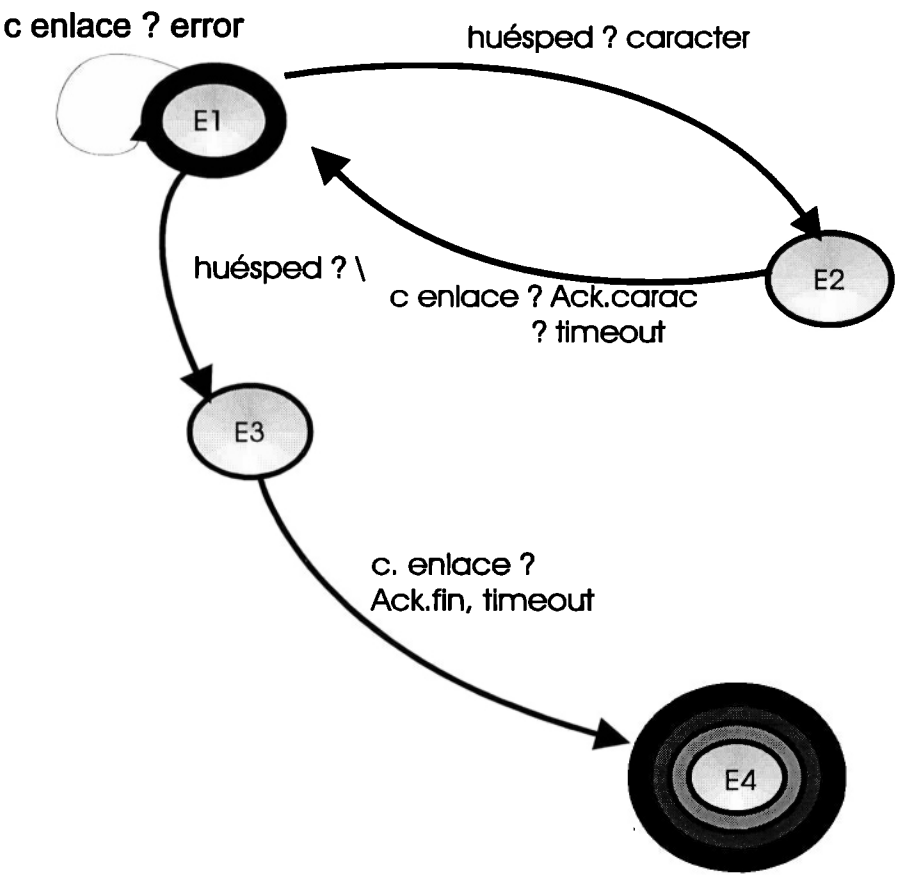
```
-- prorob3
-- aplic.occ
#INCLUDE      "hostio.inc"
#INCLUDE      "proto.inc"

PROC aplic    (CHAN OF SP fs, ts,
          CHAN OF A.E a.a.e, CHAN OF E.A e.a.a,
          CHAN OF A.EI a.a.ei, CHAN OF E.AI e.a.ai )
 #USE        "hostio.lib"
 #INCLUDE    "const.inc"
 BOOL        sigue,
          finin :
 INT         hora,
          momento :
 TIMER       reloj :

 SEQ
  sigue     := TRUE
  finin     := TRUE
  WHILE     sigue

   BYTE     cara :
   SEQ
    reloj ? hora
    ALT  -- recibo datos del Robot Enlace e Input a aplicacion
     e.a.ai ? CASE        -- recepcion de datos
      carac.e.ai; cara      -- si recibo el primer caracter

       BOOL     continua :
       SEQ
        --so.write.string (fs, ts, "robot: ")
        continua := TRUE
        WHILE continua -- mientras haya mas caracteres
         SEQ
          so.write.char (fs, ts, cara)
          IF
           (cara <> salir)
            SEQ
             a.a.ei ! ack.carac.e.ai
           TRUE
            SEQ
             a.a.ei ! final.a.ei
             reloj ? momento
             ALT
              e.a.ai ? CASE        -- espera un Ack.fin
               carac.e.ai; cara
                SKIP
               final.e.ai
```

```
                    SEQ
                      SKIP
                  -- fin del CASE
                  reloj ? AFTER (momento PLUS limrobot )
                    SEQ
                      so.write.string.nl (fs, ts, "Enlace no contesta Ack.fin")
                -- fin ALT
                so.write.string.nl (fs, ts, "Termino")
                continua := FALSE
                sigue := FALSE
              -- fin TRUE
            -- fin IF
            IF
              (cara = nl)
                SEQ
                  so.write.nl (fs, ts)
                  continua := FALSE
              (cara = salir)
                SKIP
              TRUE
                SEQ
                  reloj ? momento
                  ALT
                    e.a.ai ? CASE        -- recibe otro caracter
                      carac.e.ai; cara
                        SKIP
                      final.e.ai
                        SEQ
                          sigue := FALSE
                          continua := FALSE
                          so.write.nl (fs, ts)
                    -- fin del CASE
                    reloj ? AFTER (momento PLUS limrobot )
                      SEQ
                        continua := FALSE
                -- fin del SEQ de TRUE
              -- fin TRUE
            -- fin del IF
          -- fin SEQ WHILE continua
        -- fin WHILE continua
      -- fin SEQ
      final.e.ai
        SEQ
          so.write.string.nl (fs, ts, "Enlace quiere terminar")
          sigue := FALSE
    -- fin del CASE e.a.ai

IF
  finin
    TIMER  reloj :
```

```
      INT    tiempo :
      SEQ
       a.a.e ! final.a.e
       reloj ? tiempo
       ALT
        e.a.a ? CASE    -- espera contestacion de E fin
         final.e.a      -- esto esperamos
          SEQ
           so.write.string.nl (fs, ts, "E termino A -> E -> L2. E -> A")
        -- fin de CASE
        reloj ? AFTER (tiempo PLUS timeout)
         SEQ
          so.write.string.nl (fs, ts, "Enlace Output no contesto Aplic termina")
       -- fin ALT
     -- fin SEQ finin
    TRUE
     SKIP
  -- fin IF
  so.exit (fs, ts, sps.success )
 -- fin SEQ principal
:
```

14. APÉNDICE F: PROCESO DE LA CAPA DE APLICACIÓN (CONTROLADOR -> ROBOT)

14.1 AUTÓMATA



14.2 Vocabulario y formato de mensajes

Mensajes de Aplicación -> Enlace

1. *carac.a.e;* **BYTE**
2. *final.a.e*

Mensajes de Enlace -> Aplicación

1. *ack.carac.a.e*
2. *error;* **INT**
3. *final.e.a (Ack)*

14.3 Medio Ambiente de desarrollo

Este proceso utiliza uno de los enlaces del transputer conectado al bus de la computadora huésped para la comunicación. El protocolo entre ellos es implementado con el uso de bibliotecas OCCAM (ver capítulo cuatro), y es confiable. Por otro lado, la comunicación entre los procesos de la capa de aplicación y de enlace es, como en los módulos anteriores, a través de canales de OCCAM.

14.4 Máquina de Estados

$M4 = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ E1, E2, E3, E4 \}$

E1: Estado inicial (espera la recepción de comandos). Al recibir el primer caracter de la computadora huésped, lo envía a la capa de enlace y pasa al estado E2. Al recibir un caracter "\" que solicita un fin, se pasa al estado E3. Si se recibe un aviso de error de la capa de enlace se informa y se regresa a este estado.

E2: Espera contestación de reconocimiento (ack.carac) de la capa de enlace para ir nuevamente al estado E1. En caso de que no conteste (timeout), se informa de esto en la aplicación y se vuelve a la espera inicial de un comando (estado E1).

E3: Se pide finalizar a la capa de enlace, se espera su confirmación Ack.fin y se termina la ejecución de éste módulo en el estado E4.

E4: Estado Final

$\Sigma = \{$ caracter, \, error, ack.carac, timeout, Ack.fin $\}$

F = { E4 }

Tabla de Transiciones

	carac	\	*error*	*ack.carac*	*timeout*	*Ack.fin*
E1	E2	E3	E1	Null	Null	Null
E2	Null	Null	Null	E1	E1	Null
E3	Null	Null	Null	Null	E4	E4

14.5 CÓDIGO DEL PROGRAMA

```
-- prorob3
-- aplic.occ
#INCLUDE       "hostio.inc"
#INCLUDE       "proto.inc"

PROC aplic     (CHAN OF SP fs, ts,
            CHAN OF A.E a.a.e, CHAN OF E.A e.a.a,
            CHAN OF A.EI a.a.ei, CHAN OF E.AI e.a.ai )
 #USE          "hostio.lib"
 #INCLUDE      "const.inc"
 BOOL          sigue,
            finin :
 INT           hora,
            momento :
 TIMER         reloj :

 SEQ
  sigue       := TRUE
  finin       := TRUE
  WHILE       sigue

    BYTE      cara :
    SEQ
     reloj ? hora
     ALT  -- recibo datos del Robot Enlace e Input a aplicacion

-- recibo comnando del HOST y Output a Enlace
       reloj ? AFTER (hora PLUS limite )

         BYTE       resultado,
                 caracter:

         SEQ
          reloj ? hora
          ALT
```

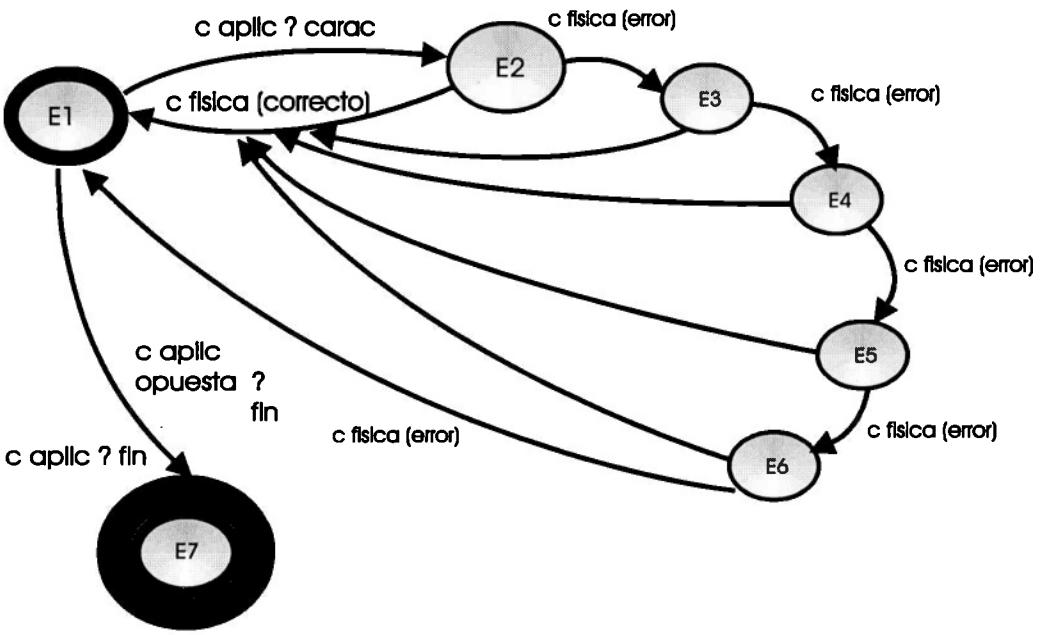
```
e.a.a ? CASE
 error
  SEQ
   so.write.string.nl (fs, ts, "Error: Capa fisica no funciona")
-- fin CASE
reloj ? AFTER (hora PLUS lim2)
 SEQ
  so.pollkey (fs, ts, caracter, resultado)
  IF
   (resultado <> 0 (BYTE)) -- no hay comando HOST
    SKIP
   TRUE                  -- si hay comando
    BOOL continua :
    SEQ
     continua := TRUE
     WHILE continua
      SEQ
       IF
        (resultado <> 0 (BYTE))
         SKIP
        TRUE
         SEQ
          so.write.char (fs, ts, caracter )
          IF
           (caracter = nl )
            SEQ
             so.write.nl ( fs, ts )
             continua := FALSE
           (caracter = '\')
            SEQ
             sigue := FALSE
             continua:= FALSE
             finin := FALSE
           TRUE
            SKIP
          -- fin del IF
          IF
           (caracter<>'\')
            TIMER    clock :
            INT      tiempo :
            SEQ
             a.a.e ! carac.a.e; caracter
             clock ? tiempo
             ALT
              e.a.a ? CASE
               error
                SEQ        -- no sucede
                 so.write.string.nl (fs, ts, "Enlace no pudo usar el link")
               ack.carac.a.e -- esto esperamos
                SEQ
```

```
                SKIP
              -- fin del CASE
              clock ? AFTER (tiempo PLUS timeout)
                SEQ
                  so.write.string.nl (fs, ts, "Enlace no contesta")
                  continua := FALSE
                  sigue := FALSE
                  finin := FALSE
                -- fin SEQ
              -- fin clock ? AFTER ...timeout
            -- fin ALT
          --fin SEQ si caracter <> '\'
        --fin caracter <> '\'
        TRUE
          TIMER    clock :
          INT      tiempo :
          SEQ
            a.a.e ! final.a.e
            clock ? tiempo
            ALT
              e.a.a ? CASE
                error
                  SEQ        -- no sucede
                    so.write.string.nl (fs, ts, "Enlace no pudo usar el link")
                ack.carac.a.e -- no sucede
                  SEQ
                    SKIP
                final.e.a -- Ack.fin
                  SEQ
                    SKIP
                    -- so.write.string.nl (fs, ts, "Enlace contesto ack.fin")
              -- fin del CASE
              clock ? AFTER (tiempo PLUS timeout)
                SEQ
                  so.write.string.nl (fs, ts, "No llega Ack.fin de EO")
                  continua := FALSE
                  sigue := FALSE
                  finin := FALSE
                -- fin SEQ
              -- fin clock ? AFTER ...timeout
            -- fin ALT
          -- fin SEQ
        -- fin TRUE
      -- fin IF
    -- fin del SEQ TRUE
  -- fin de TRUE
-- fin del IF
so.pollkey (fs, ts, caracter, resultado)
-- fin SEQ WHILE ...
-- fin WHILE continua
```

```
                -- fin SEQ TRUE
              -- fin TRUE si Host comenzo un comando
            -- fin IF teclearon algo
          -- fin de SEQ
        -- fin de reloj ? AFTER (hora PLUS lim2)
      -- fin de ALT
    -- fin SEQ si se cumplio el timeout
  -- fin de reloj ? AFTER para timeout
 -- fin de ALT
-- fin SEQ del WHILE
-- fin WHILE sigue
so.exit (fs, ts, sps.success )
-- fin SEQ principal
:
```

15. APÉNDICE G: PROCESO DE LA CAPA DE ENLACE (CONTROLADOR -> ROBOT)

15.1 AUTÓMATA



15.2 Vocabulario y formato de mensajes

Mensajes de Aplicación -> Enlace

1. *carac.a.e;* **BYTE**
2. *final.a.e*

Mensajes de Enlace -> Aplicación

1. *ack.carac.a.e*
2. *error;* **INT**
3. *final.e.a*

Mensajes de C. Enlace -> C. Física

1. **BYTE**

15.3 Medio Ambiente de desarrollo

Los procesos se comunican de igual forma que en los módulos anteriores, por medio de canales implementados en OCCAM. En la comunición con la capa física también se usan canales, pero conectados a enlaces (links) de comunicación del transputer. En este caso, el envío de un mensaje es por medio de la instrucción:

canal !! *mensaje*

descrita en la sección 6.2.

15.4 Máquina de Estados

M4 = (Q, Σ, δ, q_0, F)

Q = { E1, E2, E3, E4, E5, E6, E7}

E1: Espera caracter de la capa de aplicación, al recibirlo contesta Ack.carac a la capa de aplicación y lo envía al robot. Si el caracter que envía la capa de aplicación es una solicitud para finalizar este proceso se va al estado final E7. Si se recibe una solicitud de fin de la capa de enlace opuesta le confirma y este proceso termina (Edo. E7).

E2: Espera la recepción correcta para regresar al estado E1, en caso de que el robot no esté recibiendo correctamente el caracter pasa al estado E2.

Los siguientes cuatro estados corresponden al mecanismo de reenvío de caracteres al robot en caso de que éste no esté respondiendo correctamente. En cualquiera de los cuatro podemos llegar al estado E1 en caso de una recepción correcta del reenvío al robot.

E3: Primer reenvío.

E4: Segundo reenvío.

E5: Tercer reenvío.

E6: Cuarto y último reenvío. En caso de que en este estado el reenvío fracase se informa de esto a la capa de aplicación y la ejecución continúa.

E7: Estado Final.

Σ = { caraca, fin, recep.correc.robot, no.recep.robot }

F = { E7 }

Tabla de Transiciones

	carac	fin	*recep.correc.robot*	no.recep.robot
E1	E2	E7	Null	Null
E2	Null	Null	E1	E3
E3	Null	Null	E1	E4
E4	Null	Null	E1	E5
E5	Null	Null	E1	E6
E6	Null	Null	E1	E1

15.5 CÓDIGO DEL PROGRAMA

```
-- prorob
-- enlaceo.occ
#INCLUDE      "hostio.inc"
#INCLUDE      "proto.inc"

PROC enlaceo ( CHAN OF A.E a.a.e, CHAN OF E.A e.a.a,
               CHAN OF ANY e.a.r)

  #USE        "hostio.lib"
  #USE        "xlink.lib"
  #INCLUDE    "const.inc"
  BOOL        sigue :

  SEQ
    sigue := TRUE
    WHILE sigue

      [1]BYTE    caracter :
      BOOL    banderror, abortado :
      TIMER   reloj :
      INT     hora, t :
```

```
    SEQ
     a.a.e ? CASE          -- recibo comandos A
      carac.a.e; caracter[0]
       INT vuelta :
       SEQ
        e.a.a ! ack.carac.a.e
        vuelta := 0
        WHILE (vuelta < limvuel)
         SEQ
          --OutputOrFail.t(e.a.r, caracter, reloj, t, abortado)
          e.a.r ! caracter
          abortado := FALSE
          IF
           abortado
            vuelta := vuelta + 1
           TRUE
            vuelta := limvuel
          -- fin IF
         -- fin SEQ
        -- fin WHILE
        IF
         abortado
          SEQ
           e.a.a ! error
           Reinitialise (e.a.r)
         TRUE
          SKIP
        -- fin IF
       -- fin SEQ WHILE continua
      final.a.e -- Aplicaci¢n quiere terminar
       SEQ
        e.a.a ! final.e.a  -- Enlace termina avisa F
        sigue := FALSE
       -- fin SEQ final.a.e
      -- fin final.a.e
     -- fin del CASE a.a.e ? ...
    -- fin de SEQ del WHILE
   -- fin de WHILE sigue
  -- fin SEQ
:
```

16. APÉNDICE H: DEFINICIÓN DE PROTOCOLOS OCCAM

```
-- Prorob3
-- proto.inc
PROTOCOL A.E
  CASE
    carac.a.e; BYTE
    final.a.e
:
PROTOCOL E.A
  CASE
    ack.carac.a.e
    error
    final.e.a
:
PROTOCOL A.EI
  CASE
    ack.carac.e.ai
    final.a.ei
:
PROTOCOL E.AI
  CASE
    carac.e.ai; BYTE
    final.e.ai
:
PROTOCOL F.EI IS BYTE :
PROTOCOL E.L IS BYTE :
PROTOCOL L.E IS BYTE :
PROTOCOL ROBOT IS BYTE::[]BYTE :
PROTOCOL SIGA IS BYTE :
```

17. APÉNDICE I: CONFIGURACIÓN Y EJECUCIÓN DE LOS PROCESOS DEL PROTOCOLO EN FORMA CONCURRENTE.

17.1 Código de la configuración y ejecución

```
-- Prorob3
-- conf.pgm
#INCLUDE "hostio.inc"
#INCLUDE "proto.inc"
#INCLUDE "linkaddr.inc"
#USE "aplic.c8h"
#USE "enlaceo.c8h"
#USE "enlacei.c8h"
#USE "robot.c8h"
CHAN OF SP from.host, to.host :
PLACED PAR
  PROCESSOR 0 T800
    PLACE from.host AT link0.in:
    PLACE to.host AT link0.out :
    CHAN OF A.E a.a.e :
    CHAN OF E.A e.a.a :
    CHAN OF A.EI a.a.ei :
    CHAN OF E.AI e.a.ai :
    CHAN OF ANY e.a.r :
    CHAN OF BYTE r.a.e :
    PAR
      aplic (from.host, to.host, a.a.e, e.a.a, a.a.ei, e.a.ai )
      enlaceo (a.a.e, e.a.a, e.a.r )
      enlacei (a.a.ei, e.a.ai, r.a.e )
      robot ( e.a.r, r.a.e )
```

17.2 Unión de los procesos en la capa de aplicación

```
-- prorob3
-- aplic.occ
#INCLUDE      "hostio.inc"
#INCLUDE      "proto.inc"

PROC aplic    (CHAN OF SP fs, ts,
               CHAN OF A.E a.a.e, CHAN OF E.A e.a.a,
               CHAN OF A.EI a.a.ei, CHAN OF E.AI e.a.ai )
  #USE        "hostio.lib"
  #INCLUDE    "const.inc"
```

```
  BOOL        sigue,
              finin :
  INT         hora,
              momento :
  TIMER       reloj :

  SEQ
    sigue     := TRUE
    finin     := TRUE
    WHILE     sigue

      BYTE    cara :
      SEQ
        reloj ? hora
        ALT  -- recibo datos del Robot
          e.a.ai ? CASE       -- recepcion de datos
            carac.e.ai; cara      -- si recibo el primer caracter
              BOOL    continua :
              SEQ
                continua := TRUE
                WHILE continua -- mientras haya mas caracteres

                -- Procesamiento de los caracteres que vienen del robot

                -- fin WHILE continua
              -- fin SEQ
            final.e.ai
              SEQ
                so.write.string.nl (fs, ts, "Enlace quiere terminar")
                sigue := FALSE
          -- fin del CASE e.a.ai
-- recibo comnando del HOST y Output a Enlace
          reloj ? AFTER (hora PLUS limite )
            BYTE      resultado,
                      caracter:
            SEQ
              reloj ? hora
              ALT

              -- Estructura que espera un dato de la computadora huésped o un error de
              -- la capa de enlace

                e.a.a ? CASE
                -- En caso de que llegue un error de la capa de enlace

                  error
                    SEQ
                      so.write.string.nl (fs, ts, "Error: Capa fisica no funciona")
                -- fin CASE
                reloj ? AFTER (hora PLUS lim2)
```

```
            -- Revisar si hay un comando de la computadora huésped

            SEQ
              so.pollkey (fs, ts, caracter, resultado)
              IF
                (resultado <> 0 (BYTE)) -- no hay comando HOST
                  SKIP
                TRUE
                -- si hay comando de la computadora huésped
                  BOOL continua :
                  SEQ
                    continua := TRUE
                    WHILE continua

                      -- Procesamiento del comando que viene de la computadora huésped

                      -- fin WHILE continua
                  -- fin SEQ TRUE
                -- fin TRUE si Host comenzo un comando
              -- fin IF teclearon algo
            -- fin de SEQ
          -- fin de reloj ? AFTER (hora PLUS lim2)
        -- fin de ALT
      -- fin SEQ si se cumplio el timeout
    -- fin de reloj ? AFTER para timeout
  -- fin de ALT
 -- fin SEQ del WHILE
-- fin WHILE sigue

IF
    finin
    -- Si el robot pidió finalizar:
     TIMER  reloj :
     INT    tiempo :
     SEQ
      a.a.e ! final.a.e
      reloj ? tiempo
      ALT
        e.a.a ? CASE    -- espera contestacion de E fin
          final.e.a     -- esto esperamos
            SEQ
              so.write.string.nl (fs, ts, "E termino A -> E -> L2. E -> A")
        -- fin de CASE
        reloj ? AFTER (tiempo PLUS timeout)
          SEQ
            so.write.string.nl (fs, ts, "Enlace Output no contesto Aplic termina")
      -- fin ALT
    -- fin SEQ finin
    TRUE
     SKIP
```

```
   -- fin IF
   -- terminación del proceso
   so.exit (fs, ts, sps.success )
  -- fin SEQ principal
:
```

18. APÉNDICE J: DEFINICIÓN DE CONSTANTES

```
-- prorob3
-- const.inc
VAL    enter   IS     '*c' :
VAL    salir   IS     27 (BYTE) :
VAL    limite  IS     200 :
VAL    lim2    IS     200 :
VAL    spc     IS     ' ' :
VAL    timeout IS     15000 :
VAL    limrobot IS    8000 :
VAL    nl      IS     13 (BYTE) :
VAL    limvuel IS     4 :
```

19. APÉNDICE K: Listado del Modelo para la validación del protocolo en Promela

```
mtype = {caracter, ack_carac, fin, error, verde}

#define p          0
#define v          1
#define max1       9
#define maxpre1    16
#define maxpre2    13
#define max3       3
#define li2        20
#define li         5 /* limite m ximo de intentos */
#define licom      10 /* limite m ximo de comandos enviados al robot */
#define limsale    8 /* limite m ximo de espera del error en AO */


  chan A = [0] of {byte, byte};
  chan B = [0] of {byte, byte};
  chan C = [0] of {byte};
  chan D = [0] of {byte};
  chan AI = [0] of {byte, byte};
  chan BI = [0] of {byte, byte};
  chan CI = [0] of {byte};
  chan DI = [0] of {byte};
  chan APLI = [0] of {byte};

/*****************************************/
/*          COMANDOS DE APLIC. AL ROBOT  */

proctype aplico (chan aplenl, enlapl, aplapl)
{
  byte  i;
  bool berror;
  int   vuelta, puede, sale;
puede = 0;
do /*ciclo que envia varios comandos al robot */
::i = 0;
  puede = puede + 1;
  aplapl ? verde;
  do  /* ciclo que envia todos los caracteres de un comando al robot */
  :: (i <= max1) ->
     if
     :: (i < max1) -> i = i + 1; aplenl ! caracter, i;
     :: (i == max1) -> i = i + 1;
         aplenl ! caracter, 255;/* env¡o del 013 al robot */
     fi;
     if
```

```
    :: enlapl ? ack_carac;
    :: timeout ->
       printf("AO: Error, capa de enlace no contesta\n ");
    fi;
    if
    :: (i <= max1) ->
     berror = 0;
     sale = 0;
     do
     :: ((sale < limsale) && (berror == 0))->
       sale = sale + 1;
       if
       :: enlapl ? ack_carac;
       :: enlapl ?  error ->
         berror = 1;
         sale = limsale;
       :: timeout ->
         skip;
       fi;
     :: (sale >= limsale) ->
        break;
     od;
     if
     :: (berror == 1) ->
        printf("AO(F): Error, capa fisica no funciona\n ");
     :: (berror != 1) ->
       skip;
     fi;
   :: (i > max1) -> skip;
   fi;
 :: (i > max1) -> break;
 od;

:: (((vuelta == 0) && (puede > 3)) || (vuelta > licom)) ->
  aplapl ? verde;
  aplenl ! fin;
  if
  :: enlapl ? fin;
  :: timeout ->
    printf("AO: No llega Ack.fin de EO\n ");
  fi;
  break;
:: vuelta = vuelta + 1
:: vuelta = vuelta -1
od;
end:  printf("Aplico termina\n "); /* valida una terminaci¢n correcta */
}

proctype enlaceo (chan aplenl, enlapl, enlfis, fisenl)
{
```

```
  byte x, intento;
  bool continua, envia;

continua = 1;
do
:: (continua == 1 ) ->
   if
   :: aplenl ? caracter, x ->
      if
      :: enlapl ! ack_carac;
/*         printf("EO: Deposite Ack.carac\n ");*/
      :: skip;
/*         printf("EO: No deposit, Ack.carac\n ");*/
      fi;
      envia = 1;
      intento = 1;
      do /* ciclo de reenvio en caso de error en la recepcion */
      :: ((envia == 1) && (intento < li)) ->
         enlfis ! caracter, x;
         if
         :: fisenl ? ack_carac;
            envia = 0;
         :: timeout -> /* si c. fisica no est  funcionando bien*/
            printf("EO: No llega Ack.carac de F, reintentar,\n ");
            intento = intento + 1;
         fi;
      :: (envia == 0) ->
         break;
      :: (intento >= li) ->
         enlapl ! error;
         break;
      od;
   :: aplenl ? fin ->
      enlfis ! fin;
      if
      :: enlapl ! fin;
      :: skip;
      fi;
      continua = 0;
      break;
   :: timeout ->
/*      printf("EO: Espero carac\n ");*/
      skip;
   fi;
:: (continua == 0) -> break;
od;
assert (len(fisenl) == 0);
end:  printf("Enlaceo termina\n "); /* valida una terminación correcta */
}
```

```
/*****************************************/
/*        DATOS DEL ROBOT A APLICACION */

proctype fisico (chan fisenl, enlfis, enlfiso, fisenlo, aplapl)
{
  byte  i, x;
  int   continua;

/* Robot -> controlador */
printf("\nRobot Pregunta1: ");
i = 0;
do
:: (i <= maxpre1) ->
   if
   :: (i < maxpre1) -> i = i + 1; fisenl ! caracter, i;
   :: (i == maxpre1) -> i = i + 1;
/*      printf("F: 255 %d\n ", i);*/
      fisenl ! caracter, 255;/* env¡o del 013 a apli */
   fi;
   if
   :: enlfis ? ack_carac;
   :: enlfis ? error;
      printf("F: Controlador no recibe dato\n ");
   :: timeout ->
        if
        :: enlfis ? ack_carac;
        :: enlfis ? error;
           printf("F:Controlador no recibe dato\n");
        fi;
       fi;
:: (i > maxpre1) -> break;
od;

/* controlador -> robot */
printf("\nControlador Responde: ");
continua = 1;
do
:: (continua == 1) ->
   if
   :: enlfiso ? caracter, x ->
      if
      :: (x == 255) ->
         fisenlo ! ack_carac;
         continua = 0;
      :: (x != 255 ) ->
         if  /* simulaci¢n de falla en emisi¢n del Ack.carac */
         :: fisenlo ! ack_carac;
/*            printf("FO: Deposite Ack.carac\n");*/
         :: skip;
```

```
/*          printf("FO: No dep. Ack.carac\n");*/
        fi;
        printf("%d", x);
      fi;
    :: enlfiso ? fin ->
      continua = 0;
      break;
    :: timeout ->
/*      printf("FO: Espero carac\n");*/
      skip;
  fi;
:: (continua == 0) -> break;
od;
x = 0;

/* Robot -> controlador */
printf("\nRobot Pregunta2: ");
i = 0;
do
:: (i <= maxpre2) ->
  if
  :: (i < maxpre2) -> i = i + 1; fisenl ! caracter, i;
  :: (i == maxpre2) -> i = i + 1;
    fisenl ! caracter, 255;/* envío del 013 a apli */
  fi;
  if
  :: enlfis ? ack_carac;
  :: enlfis ? error;
    printf("F: controlador no recibe dato\n ");
  :: timeout ->
    if
    :: enlfis ? ack_carac;
    :: enlfis ? error;
      printf("F:Controlador no recibe dato\n ");
    fi;
  fi;
:: (i > maxpre2) -> break;
od;

/* controlador -> robot */
printf("\nControlador Responde: ");
continua = 1;
do
:: (continua == 1) ->
  if
  :: enlfiso ? caracter, x ->
    fisenlo ! ack_carac;
    if
    :: (x == 255) ->
      continua = 0;
```

```
      :: (x != 255 ) ->
         printf("%d", x);
      fi;
   :: enlfiso ? fin ->
      continua = 0;
      break;
   :: timeout -> skip;
   fi;
:: (continua == 0) -> break;
od;
aplapl ! verde;
continua = 0;
do
:: (continua == 0) ->
   printf("\nComando: ");
   continua = 1;
   do
   :: (continua == 1) ->
      if
      :: enlfiso ? caracter, x ->
         fisenlo ! ack_carac;
         if
         :: (x == 255) ->
            continua = 0;
         :: (x != 255 ) ->
            printf("%d", x);
         fi;
      :: enlfiso ? fin ->
         continua = -1;
         fisenl ! fin;
         break;
      :: timeout -> skip;
      fi;
   :: (continua <= 0) -> break;
   od;
   if
   :: (continua == 0) ->
      printf("\nAck_Comando: ");
      fisenl ! caracter, 0;
      if
      :: enlfis ? ack_carac;
      :: enlfis ? error;
         printf("F: Controlador no recibe dato\n ");
      :: timeout ->
         if
         :: enlfis ? ack_carac;
         :: enlfis ? error;
            printf("F: Controlador no recibe dato\n ");
         fi;
      fi;
```

```
      fisenl ! caracter, 255;
      if
      :: enlfis ? ack_carac;
      :: enlfis ? error;
        printf("F: Controlador no recibe dato\n ");
      :: timeout ->
        if
        :: enlfis ? ack_carac;
        :: enlfis ? error;
          printf("F: Controlador no recibe dato\n ");
        fi;
      fi;
  :: (continua != 0) -> skip;
  fi;
:: (continua < 0) -> break;
od;
end:  printf("Robot termina\n "); /*Valida correcta terminaci¢n de proceso*/
}

proctype enlacei (chan fisenl, enlfis, enlapl, aplenl)
{
  byte x, intento;
  bool continua, envia;

continua = 1;
do
:: (continua == 1 ) ->
  if
  :: fisenl ? caracter, x ->
    envia = 1;
    intento = 1;
    do
    :: ((envia == 1) && (intento < li)) ->
      enlapl ! caracter, x;
      if
      :: aplenl ? ack_carac;
        enlfis ! ack_carac;
        envia = 0;
      :: timeout ->
        printf("El: No llega Ack.carac de Al, reintentar,\n ");
        intento = intento + 1;
      fi;
    :: (envia == 0) -> break;
    :: (intento >= li) ->
      enlfis ! error;
      break;
      printf("El: error a Fl\n ");
    od;
  :: fisenl ? fin ->
    enlapl ! fin;
```

```
      continua = 0;
      break;
   fi;
:: (continua == 0) -> break;
od;
end: printf("I:Enlacei termina\n ");
}

proctype aplici(chan enlapl, aplenl, aplapl)
{
  byte x;
  bool continua;

continua = 1;
do
:: (continua == 1) ->
   if
   :: enlapl ? caracter, x ->
      if
      :: (x != 255) ->
         printf("%d", x);
         if
         :: aplenl ! ack_carac;
         :: skip
         fi;
      :: (x == 255) ->
         aplenl ! ack_carac;
         aplapl ! verde;
      fi;
   :: enlapl ? fin ->
      continua = 0;
      break;
   fi;
:: (continua == 0) -> break;
od;
assert (len(enlapl) == 0);
end:  printf("I:Aplici termina\n ");
}

init
{
  byte vuelta;

run enlaceo (A, C, B, D);
run enlacei (AI, CI, BI, DI);
run aplici (BI, DI, APLI);
run aplico (A, C, APLI);
run fisico (AI, CI, B, D, APLI);
}
```

20. APÉNDICE L: Breve resumen de la sintaxis de Promela

Hay cinco clases de tokens: identificadores, palabras reservadas, constantes, operadores y separadores de sentencias o comandos.

Comentarios

Cualquier cadena que inicie con /* y termine con */

Identificadores

Una letra, punto o guión bajo seguido por cero o más letras, dígitos, puntos o guiones bajos.

Palabras reservadas

active	assert	atomic	bit
bool	break	byte	chan
d_step	do	else	empty
enabled	fi	full	got
hidden	if	init	int
len	mtype	nempty	never
nfull	od	of	pc_value
printf	proctype	run	short
skip	timeout	typedef	unless
xr	xs	_	

Constantes

Representación de un entero:

#define NOMBRE valor

Expresiones

Se pueden usar los siguientes operadores:

```
+       -       *       /       %
>       >=      <       <=      ==      !=      !
&&      ||
&       |       >>      <<
++
```

Operadores aplicados a canales

```
len  empty      nempty    nfull      full
```

Operador para instanciar procesos

```
run
```

Operadores para envío y recepción de mensajes

```
!    ?
```

Declaraciones

Los procesos, canales y variables deben ser declarados antes de usarlos.

VARIABLES

Inicia con una palabra reservada que indica el tipo y continúa con el identificador. Los tipos son: **bit**, **bool** (lógico 0 ó 1), **byte**, **mtype**, **short** (entero de 16 bits) e **int** (entero de 32 bits). **mtype** permite definir tipos por el usuario.

CANALES DE MENSAJES

El formato para la definición de un canal es:

chan *nombre* = [*N*] **of** **{int, short, ...}**

PROCESOS

La declaración de un proceso es como sigue:

```
proctype nombre ( parámetros )
{
    /* comandos */
}
```

Estatutos

assertion (*assert*:)	asignación (=)	atomic
break	declaraciones	expresiones
goto	receive (canal ? variable)	selección (*if ... fi*;)
repetición (*do ... od*;)	send (canal ! variable o cte.)	timeout
unless	sorted_send	random_receive

Estatuto IF

Este estatuto puede ser usado para probar diversas condiciones, si ninguna se cumple, la bandera de error del transputer se activa y se detiene el programa. La prueba de condiciones se realiza en el orden en que se colocaron y la primera que se cumpla realizará las acciones que tenga. Por ejemplo:

```
IF
        a > 1
                ... acciones uno
        b = 0
                ... acciones dos
        TRUE
                ... acción por default
-- fin del IF
```