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

"ESPECIFICACIÓN FORMAL DEL MANEJO Y ADMINISTRACIÓN DE UNA MEMORIA COMPARTIDA DISTRIBUIDA : ADMINISTRADOR DE MEMORIA EXTERNO "

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

MARCOS RUBÉN DE ALBA ROSANO

BIBLIOTECA

Asesor: **DR. HÉCTOR RUIZ BARRADAS**

Comité de tesis: **Dr. JESÚS VAZQUEZ**
Dr. JESÚS SANCHEZ
Dr URIEL TIRADO

Jurado:	**Dr. JESÚS SÁNCHEZ**	**Presidente**
	Dr. JESÚS VÁZQUEZ	**Secretario**
	Dr. URIEL TIRADO	**Vocal**
	Dr. HÉCTOR RUIZ	**Vocal**

Atizapán de Zaragoza, Estado de México, Noviembre de 1996.

INDICE DEL CONTENIDO

Prefacio **8**

Introducción **8**

Agradecimientos **11**

1. Memoria Compartida Distribuida (MCD) **13**

1.1 Introducción 13
1.2 Características de la MCD 14
1.3 Objetivos y formas de construcción de una MCD 15
1.3.1 MCD basada en bloques 15
1.3.2 MCD basada en estructura 16
1.4 Capas de construcción de una MCD 17
1.5 Aplicaciones de la MCD 18
1.6 Estado del Arte de la MCD 18

2. El Sistema Operativo Mach **20**

Introducción **21**

2.1 Arquitectura del modelo de Mach 23
2.2 Hilos y tareas 25
2.2.1 Creación de una tarea 26
2.2.2 Invocación de operaciones del núcleo 26
2.2.3 Procesamiento de excepciones 27
2.2.4 Administración de tareas e hilos 27
2.3 Modelo de comunicación 28
2.4 Realización de la comunicación 30
2.5 Administración de memoria 31
2.6 Administración de memoria virtual 32
2.6.1 Conceptos básicos del modelo de memoria virtual en Mach 32
2.6.2 Rangos y tamaños de página 34
2.6.3 Espacio de direcciones virtuales 36
2.6.4 Objetos de memoria 39
2.6.5 Manejo básico 40
2.7 Administradores de memoria y paginadores 42
2.7.1 Administrador de memoria estándar 43
2.7.2 Paginadores de memoria externos 44
2.8 Memoria compartida distribuida en Mach 45
2.9 Consistencia o coherencia de páginas 46

3. LOTOS: Language Of Temporal Ordering Specification **49**

3.1 Introducción 50
3.2 El lenguaje LOTOS 51
3.3 Expresiones de comportamiento en LOTOS y su semántica 52
3.3.1 Operadores 52
3.4 Representación de procesos como árboles 59

3.5 Tipos de datos 65
3.5.1 Conceptos básicos 65
3.5.2 Tipos de datos básicos 67
3.6 Full LOTOS 74
3.6.1 Eventos estructurados 74
3.6.2 Conducta condicional en Full LOTOS 75
3.6.3 Procesos parametrizados 76
3.6.4 Funcionalidad 76
3.6.5 Definición de valor local 77
3.6.6 Alcance ("Scope") 78
3.7 Especificación en LOTOS 78
3.7.1 Definición de tipo global 79
3.7.2 Estructuras de eventos 80
3.7.3 Predicados de selección 81
3.7.4 Parámetros y guardias en Full LOTOS 81

4. Descripción informal de una MCD enfocada al sistema operativo Mach 84

4.1 Administrador de Memoria Externo, AME 85
4.2 Elementos básicos de un administrador de memoria externo 85
4.3 Relaciones entre los elementos del AME 86
4.4 Funcionamiento y estructura del paginador 88
4.5 Operaciones en el uso de memoria 92
4.5.1 Obtención de un objeto de memoria 92
4.5.2 Mapeo de un Objeto de Memoria 95
4.5.3 Acceso a un Objeto de Memoria 97
4.5.4 Manejo del mecanismo de salida de páginas 99
4.5.5 Consistencia y fallo de protección de páginas 99
4.5.6 Eliminación de un objeto de memoria 100

5. Especificación Formal del Administrador de Memoria Externo 102

5.1 Principios de diseño de la especificación 103
5.2 Especificación inicial 104
5.3 Primer refinamiento 109
5.3.1 Justificación del primer refinamiento 109
5.3.2 Especificación resultante del primer refinamiento 110
5.3.3 Descripción de la especificación resultante del primer refinamiento 112
5. 4 Segundo refinamiento 114
5.4.1 Justificación del segundo refinamiento 114
5.4.2 Especificación resultante del segundo refinamiento 115
5.4.3 Descripción de la especificación resultante del segundo refinamiento 115
5.5 Tercer refinamiento 126
5.5.1 Justificación del tercer refinamiento 126
5.5.2 Especificación resultante del tercer refinamiento 126
5.5.3 Descripción del tercer refinamiento 127
5.5.4 Descripción de la especificación resultante del tercer refinamiento 128
5.6 Cuarto refinamiento 131
5.6.1 Justificación del cuarto refinamiento 131
5.6.2 Especificación resultante del cuarto refinamiento 131
5.6.3 Descripción del cuarto refinamiento 131
5.6.4 Comprobación de comportamientos equivalentes entre AME4 y AME5 133
5.7 Quinto Refinamiento 136

5.7.1 Justificación del quinto refinamiento 136
5.7.2 Especificación resultante del quinto refinamiento 136
5.7.3 Descripción del quinto refinamiento 136

6. Conclusiones 148

Bibliografía 152

Apéndices 155

INDICE DE FIGURAS

FIG. 1.1 ESQUEMA DE LA MEMORIA COMPARTIDA DISTRIBUIDA. 14
FIG. 1.2 PROYECTOS DE MCD. 19
FIG. 2.1 MODELO DE MEMORIA DE UN PROGRAMA EN MACH. 33
FIG. 2.2 MACH CONVIERTE RANGOS DE MEMORIA A RANGOS DE PÁGINAS 34
FIG. 2.3 MEMORIA VIRTUAL 35
FIG. 2.4 ABSTRACCIONES DE MEMORIA VIRTUAL 36
FIG. 2.5 ESPACIO DE DIRECCIONES 36
FIG. 2.6 REGIONES DE UNA MEMORIA VIRTUAL 37
FIG. 2.7 RANGOS DE MEMORIA DE UNA TAREA 38
FIG. 2.8 PAGINADOR SIN MEMORIA COMPARTIDA DISTRIBUIDA. 46
FIG. 2.9 PAGINADOR CON MEMORIA COMPARTIDA DISTRIBUIDA. 48
FIG. 3.1 PARTES DE LOTOS. 51
FIG. 3.2 REPRESENTACIÓN DE PROCESOS COMO ÁRBOLES. 60
FIG. 3.3 ARBOL DE COMPORTAMIENTO DEL SISTEMA DE DOS CLAVES 61
FIG 3.4 ARBOL DE COMPORTAMIENTO UTILIZANDO RECURSIVIDAD. 62
FIG. 3.5 ARBOL DE COMPORTAMIENTO CON EL OPERADOR DE INTERPAGINACIÓN. 63
FIG. 3.6 ARBOL DE COMPORTAMIENTO CON EL OPERADOR DE SINCRONIZACIÓN. 64
FIG. 3.7 ARBOL DE COMPORTAMIENTO UTILIZANDO OCULTACIÓN. 64
FIG. 3.8 CUADRO DE COMPARACIÓN DE VALORES Y VARIABLES OFERTADAS 74
FIG. 3.9. REGLAS DE FUNCIONALIDAD. 77
FIG. 3.10 CUADRO DE DIFERENCIAS ENTRE ESPECIFICACIÓN Y PROCESO. 79
FIG. 4.1 RELACIONES ENTRE LOS ELEMENTOS DEL AME. 88
FIG. 4.2 ARBOL SIMBÓLICO DE COMPORTAMIENTO DEL PAGINADOR. 88
FIG. 4.3 ARBOL DE COMPORTAMIENTO DEL PAGINADOR. 90
FIG. 4.4 ARBOL SIMBÓLICO DE LA OPERACIÓN DE OBTENCIÓN DE UN OM. 94
FIG. 4.5 ARBOL SIMBÓLICO DE LA OPERACIÓN DE MAPEO 96
FIG. 4.6 ARBOL DE ACCESO A UN OBJETO DE MEMORIA 98
FIG. 4.7 ARBOL DE LA ELIMINACIÓN DE UN OBJETO DE MEMORIA 100
FIG. 5.1 COMPORTAMIENTOS EQUIVALENTES 105
FIG. 5.2 COMPUERTAS DE COMUNICACIÓN 105
FIG. 5.3 ESPECIFICACIÓN INICIAL, AME1. 106
FIG. 5.4 COMPUERTAS DEL PRIMER REFINAMIENTO. 110
FIG. 5.5 PRIMER REFINAMIENTO, AME2. 111
FIG. 5.6 PROCESO USA_OBM DE LA ESPECIFICACIÓN AME1. 112
FIG. 5.7 PROCESO USA_OBM DE LA ESPECIFICACIÓN AME2. 113
FIG. 5.8 FUNCIÓN DE RENOMBRAMIENTO ENTRE ESPECIFICACIONES. 113
FIG. 5.9 CAMBIOS EN COMPUERTAS DEL SEGUNDO REFINAMIENTO. 115
FIG. 5.10 PROCESO USA_OBM DE LA ESPECIFICACIÓN AME3. 116
FIG. 5.11 ARBOLES DE CONDUCTA EQUIVALENTES 118
FIG. 5.12 FUNCIONES DE RENOMBRAMIENTO FORMAL. 119
FIG. 5.13 SUBÁRBOL L'. 120
FIG. 5.14 ARBOL B, CON LA SUSTITUCIÓN DE L POR L'. 121
FIG. 5.15 SUBARBOLES MAPEA Y TERMINA DE LAS FUNCIONES F_M Y F_F. 121
FIG. 5.16 ARBOL B DESPUÉS DE SUSTITUIR M' Y F'. 122
FIG. 5.17 ARBOL B REDUCIDO. 122
FIG. 5.18 FUNCIONES DE COMPORTAMIENTOS DE ESCRIBE Y USA_OBM. 123
FIG. 5.19 SUBARBOLES DEL PROCESO ESCRIBE DESPUÉS DEL RENOMBRAMIENTO. 124

FIG. 5.20 ARBÓL DEL PROCESO USA_OBM DE LA ESPECIFICACIÓN AME3. ..... *125*
FIG. 5.21 TABLA DE TIPOS DE DATOS EN LA ESPECIFICACIÓN AME4. ..... *127*
FIG. 5.22 COMPUERTAS DE COMUNICACIÓN ENTRE PARES DE ENTIDADES..... *128*
FIG. 5.23 ANÁLISIS DE SINCRONÍA ENTRE EVENTOS *129*
FIG. 5.24 ARBOLES DE COMPORTAMIENTO EQUIVALENTES. *130*
FIG. 5.25 PROCESOS MUTUOS. *132*
FIG. 5.26 ARBOLES DE COMPORTAMIENTO EQUIVALENTES *134*
FIG. 5.27 TIPOS DE DATOS DEL PAGINADOR. *137*
FIG. 5.28 TIPOS DE DATOS DE LOS OBJETOS DE MEMORIA. *138*
FIG. 5.29 DEFINICIÓN DE PUERTOS *139*
FIG. 5.30 DEFINICIÓN DE BYTE *139*
FIG. 5.31 DEFINICIÓN DE PERMISOS Y EST_COPIA *140*
FIG. 5.32 DEFINICIÓN DE TUPLA_OBM. *141*
FIG. 5.33 DEFINICIÓN DE TUPLA_OBM_NCL. *143*
FIG. 5.34 DEFINICIÓN DEL TIPO DE DATO ESTRUCTURA. *145*

INDICE DE EJEMPLOS

EJEMPLO 1 PREFIJO DE ACCIÓN. *53*
EJEMPLO 2 ELECCIÓN. *53*
EJEMPLO 3 LLAMADA DE PROCESOS. *54*
EJEMPLO 4 DEFINICIÓN LOCAL. *54*
EJEMPLO 5 SUBSTITUCIÓN EN COMPUERTAS. *55*
EJEMPLO 6 COMPOSICIÓN PARALELA *56*
EJEMPLO 7 COMPOSICIÓN PARALELA *56*
EJEMPLO 8 CONSTRUCTOR PARALELO GENERALIZADO *57*
EJEMPLO 9 OCULTAMIENTO DE EXPRESIONES. *57*
EJEMPLO 10 TERMINACIÓN SATISFACTORIA. *58*
EJEMPLO 11 EXPRESIÓN DE HABILITACIÓN. *58*
EJEMPLO 12 EXPRESIÓN DE HAILITACIÓN CON PASO DE VALOR. *58*
EJEMPLO 13 EXPRESIÓN DE DESHABILITACIÓN *59*
EJEMPLO 14 SISTEMA DE INSERCIÓN DE DOS LLAVES *61*
EJEMPLO 15 SISTEMA DE INSERCIÓN DE DOS LLAVES CON RECURSIÓN. *62*
EJEMPLO 16 SISTEMA DE DOS LLAVES CON EL OPERADOR DE INTERCALACIÓN *63*
EJEMPLO 17 SISTEMA UTILIZANDO SINCRONIZACIÓN EN UNA COMPUERTA *63*
EJEMPLO 18 SISTEMA DE UNA CLAVE UTILIZANDO OCULTAMIENTO *64*
EJEMPLO 19 TIPO DE DATO BOOLEANO. *67*
EJEMPLO 20 TIPO DE DATO NUMERO NATURAL *67*
EJEMPLO 21 EXTENSIÓN DE LA DEFINICIÓN DE LOS NÚMEROS NATURALES *68*
EJEMPLO 22 DEFINICIÓN DEL TIPO NATURALSTACK UTILIZANDO COMBINACIÓN *69*
EJEMPLO 23 DEFINICIÓN DEL TIPO NATURAL MAXIMUM *70*
EJEMPLO 24 NOMBRAMIENTO DE TIPOS DE DATOS. *71*
EJEMPLO 25 PARAMETRIZACIÓN DE TIPOS DE DATOS. *73*
EJEMPLO 26 ACTUALIZACIÓN DE TIPOS DE DATOS. *73*
EJEMPLO 27 EVENTOS ESTRUCTURADOS *75*
EJEMPLO 28 COMPORTAMIENTO CONDICIONAL *76*
EJEMPLO 29 DEFINICIÓN DE TIPO DE DATO GLOBAL. *79*
EJEMPLO 30 SISTEMA DE DOS CLAVES CON FULL LOTOS. *80*
EJEMPLO 31 PREDICADOS DE SELECCIÓN EN FULL LOTOS. *82*
EJEMPLO 32 PARÁMETROS Y GUARDIAS EN FULL LOTOS. *83*

Prefacio

Introducción

Uno de los puntos de gran interés para considerar el estudio de la memoria compartida distribuida radica en las nuevas tendencias de los modelos de programación. Los métodos tradicionales de programación están basados en el paradigma de envío de mensajes como PVM (Parallel Virtual Machine) o abstracciones de éste, como los RPC (Remote Procedure Calls). En los últimos años, se ha visto un gran interés en métodos de programación basados en mecanismos de compartición de memoria para la programación de sistemas distribuidos. Respecto a esto, se han estudiado nuevos paradigmas de programación como lo son GAMMA [11] y Linda [12], los cuales son métodos basados en un área de datos compartidos cuyos fundamentos descansan en la implantación de una memoria compartida distribuida.

Por otro lado, la descripción de sistemas distribuidos es otra área de estudio de gran interés. Los lenguajes de programación tradicionales no contemplan herramientas suficientes para la evaluación y prueba de los sistemas desarrollados con ellos; además, las herramientas actuales de diseño de sistemas no garantizan un proyecto completo, correcto y sin ambigüedades. Los errores de diseño y la carencia de herramientas que permitan comprobar el funcionamiento deseado de un sistema antes de su implantación, ocasionan un incremento considerable en los costos causados por la corrección de errores, modificaciones y anexos a un sistema en desarrollo. Este trabajo se enfoca particularmente al estudio de técnicas de descripción de sistemas distribuidos que suministran herramientas útiles en los procesos de diseño e implantación, las cuales proporcionan seguridad y confiabilidad en el funcionamiento del sistema a desarrollar, permiten la comprobación previa a la implantación, del funcionamiento del mismo y, ayudan a la verificación satisfactoria de los requerimientos deseados durante el diseño. Las técnicas que proporcionan tales ventajas en el desarrollo de sistemas distribuidos están fundamentadas en la lógica matemática y son llamadas técnicas de descripción formal. Existe una gran variedad de lenguajes de descripción formal, tal es el caso de UNITY [31], Z[29], B[30], ESTELLE[4], SDL[4], entre otros. En el presente trabajo se utiliza un lenguaje que se realizó como norma ISO. Este lenguaje

es LOTOS y está definido en la norma ISO8807:1989 [20]. Los desarrollos con este lenguaje abarcan la especificación de protocolos de la capa de transporte del modelo OSI, descripción de sistemas telefónicos, de telemática distribuida y sistemas de información de gran escala.
Tomando en consideración todo lo anterior, el objetivo e interés de este trabajo se enfoca a la especificación formal de los mecanismos del manejo y administración de una memoria compartida distribuida, en específico, la descripción formal en LOTOS de un administrador de memoria externo para el sistema operativo Mach.

Organización del documento

El trabajo está desarrollado en cinco capítulos, un apartado para las conclusiones y cuatro apéndices para incluir los refinamientos de la especificación.

En el capítulo I se introduce al lector en el contexto de la memoria compartida distribuida, objetivos, características, estado del arte de sistemas que implantan este paradigma y aplicaciones.

En el capítulo II se presenta un panorama general del sistema operativo Mach, arquitectura general, abstracciones del micronúcleo y, se profundiza en lo referente a la administración de la memoria en Mach.

En el capítulo III se realiza un análisis y se presenta un tutorial del lenguaje de especificación formal LOTOS. En éste se muestran los operadores de LOTOS básico, su función y uso. Se explica lo concerniente a los tipos de datos abstractos y su integración con LOTOS básico para dar origen a la técnica conocida como "FullLOTOS".

En los primeros tres capítulos se dan al lector los fundamentos necesarios para analizar la especificación del sistema administrador de memoria externo.

En el capítulo IV se describe informalmente el comportamiento del sistema, esto es, se explica como está constituido, las funciones de cada uno de sus elementos y las interacciones entre ellos.

En el capítulo V se realiza la especificación formal del administrador de memoria externo, diversos refinamientos, así como los análisis correspondientes.

Posteriormente, se presenta una sección de conclusiones, en la cual se efectúan observaciones del proyecto, se destacan los puntos más importantes del documento y se describen detalles relevantes referentes al proceso evolutivo del presente trabajo.

Los cuatro apéndices contienen los refinamientos de la especificación inicial del sistema.

CAPITULO 1

MEMORIA COMPARTIDA DISTRIBUIDA

1. Memoria Compartida Distribuida (MCD)

1.1 Introducción

Un sistema MCD está construido por un conjunto de memorias aisladas trabajando como una memoria única, con el objetivo de compartir datos entre computadoras que no tienen memoria física compartida. Como se aprecia en la figura 1.1, unidades con memoria física independiente pueden hacer uso de la memoria de otras unidades agregando un nivel lógico que proporciona el servicio de administración de este recurso.

El sistema de MCD debe ofrecer todas las características de una memoria compartida tradicional, incluidas la consistencia e integridad de los datos, con la finalidad de que todos los procesos, compartiendo memoria en el sistema y ejecutándose en diferentes computadoras, observen las modificaciones realizadas por alguno de ellos.

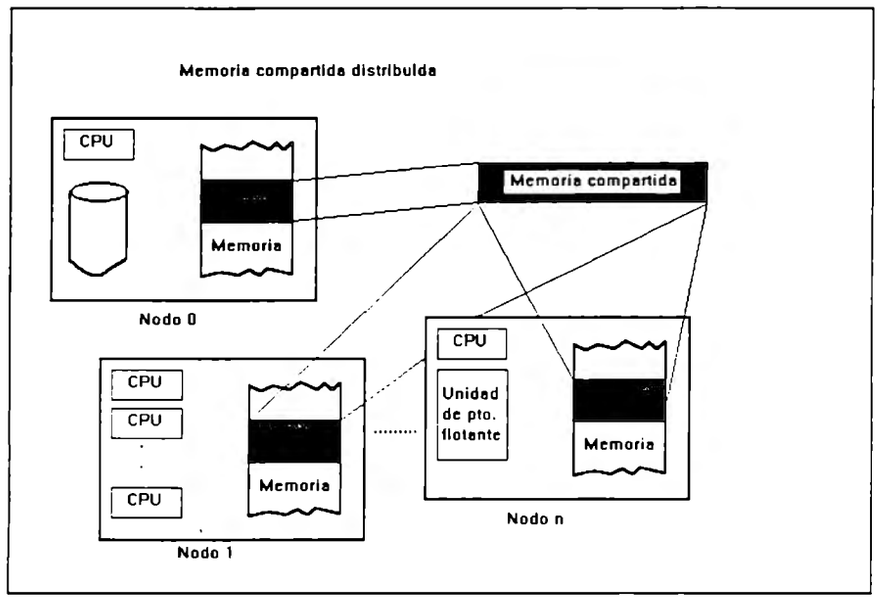


Fig. 1.1 Esquema de la memoria compartida distribuida.

1.2 Características de la MCD

Un sistema de MCD tiene las siguientes características:

a) No existe paso de mensajes explícitamente, abstracción. Los programadores no se involucran con el paso de mensajes de sus aplicaciones.

b) La MCD es una herramienta útil para aplicaciones paralelas y distribuidas en las cuales pueden ser accesados de forma individual los datos compartidos por diferentes aplicaciones.

c) A nivel de hardware se maneja la replicación de los datos para mejorar la velocidad de acceso (manipulado por el sistema operativo).

d) La información puede tener mayor duración que el tiempo de ejecución de cualquier proceso o grupo de procesos que la accesen, y puede ser compartida por diferentes grupos de procesos al mismo tiempo, persistencia.

1.3 Objetivos y formas de construcción de una MCD

Un sistema de MCD debe proporcionar en su forma más general cuatro funciones básicas constituidas principalmente por primitivas que tienen como objetivos :

a) Crear áreas de memoria compartida.
b) Escribir datos compartidos.
c) Leer datos compartidos.
d) Liberar regiones de memoria.

y su construcción puede estar basada en dos diferentes aproximaciones: en bloques o en estructura. A continuación se describe cada una de ellas.

1.3.1 MCD basada en bloques

La memoria compartida distribuida basada en bloques está caracterizada por ser una memoria particionada en bloques de tamaño fijo, es decir, el espacio de direcciones de memoria compartida se encuentra dividido en páginas lógicas. Tales bloques se distribuyen en el sistema y son controlados por uno o más manejadores.

La MCD se puede construir como una extensión de los sistemas tradicionales de memoria virtual y se diseña al mismo nivel de las capas de hardware y/o sistema operativo.

El tamaño de los bloques es un párametro vital, ya que éste determinará la cantidad, la frecuencia y el costo de las transmisiones requeridas para lograr la compartición de memoria a través de una red. además, repercute en retardos sobre el tiempo de respuesta.

Este tipo de construcción proporciona transparencia en el uso de la memoria. sin embargo, el costo es muy elevado debido a que presenta un alto grado de complejidad en el diseño y requiere la modificación del sistema operativo existente, lo que involucra un gran esfuerzo. Además, el mantenimiento para estos sistemas resulta muy costoso.

Otro inconveniente para este tipo de construcción es la dependencia de ésta con respecto al sistema. Diferentes computadoras puden tener diferentes tamaños de páginas, ordenación de bytes y métodos de manejo de memoria. y llegan a ser muy difíciles de manejar sobre sistemas heterogéneos.

1.3.2 MCD basada en estructura

En el modelo basado en estructura, la memoria compartida es particionada en pequeñas unidades, donde cada una de ellas contiene datos relacionados lógicamente.

Esta aproximación puede ser implementada como capa de lenguaje/compilador o capa de biblioteca, lo que provoca que usualmente requiera modificaciones y ajustes para el usuario final (como el modelo de programación, lenguaje o estilo).

Aunque este tipo de enfoque no tiene transparencia absoluta, es una implementación independiente y flexible. y puede ser diseñada sobre subsistemas de comunicación genéricos existentes, reduciendo los esfuerzos de desarrollo y permitiendo portabilidad a diferentes arquitecturas, soportando así, sistemas heterogéneos.

En este esquema, los programadores pueden tener la posibilidad de determinar parámetros importantes respecto a la memoria como el tamaño de bloque que deseen utilizar, el método de acceso, etc.

1.4 Capas de construcción de una MCD

El diseño de un sistema de MCD puede ser hecho a diferentes capas o niveles de abstracción como son: nivel de hardware, nivel de lenguaje de programación y nivel de biblioteca .

Una MCD diseñada como una capa de hardware o sistema operativo suministra gran transparencia y eficiencia en el uso de la memoria, sin embargo, son sistemas dependientes de la arquitectura de la máquina, con una gran complejidad de desarrollo por lo que resultan muy costosos, además, son sistemas poco portables que no soportan sistemas heterogéneos.

Una MCD modelada como una capa de lenguaje de programación requiere del desarrollo de nuevos compiladores y debe considerar puntos como la partición y distribución automática de datos, la extracción del paralelismo, así como la optimización de comunicación. Sin embargo, éstos son temas que aún requieren mucho tiempo de estudio y grandes esfuerzos de investigación para utilizarlos en el desarrollo de la MCD.

Proyectar una MCD como una capa de biblioteca tiene como inconveniente un desempeño bajo por encontrarse en niveles superiores. La ventaja es que los programadores utilizan los lenguajes tradicionales de programación que son equiparados con bibliotecas para el uso de MCD. Este tipo de sistemas son independientes, altamente portables y flexibles en lo referente a la determinación de parámetros del sistema para la afinación del desempeño. Su implantación es más fácil de desarrollar, tienen bajos costos de mantenimiento y pueden ser construidos sobre ambientes heterogéneos.

1.5 Aplicaciones de la MCD

Las aplicaciones del uso de una MCD son muy variadas. Proporcionar este servicio no sólo involucra adicionar una capa de software donde los programadores pueden usar nuevos paradigmas de programación de sistemas distribuidos, sino también la posibilidad de desarrollar servidores de archivos o bases de datos distribuidas haciendo uso de MCD.

1.6 Estado del Arte de la MCD

El desarrollo de sistemas de memoria compartida distribuida ha sido un tema de interés en diferentes universidades y centros de investigación de gran reconocimiento internacional. entre los principales proyectos se encuentran los mostrados en la figura 1.1; en ésta se muestra el nombre, el lugar de origen, el tiempo de desarrollo del proyecto, y sus características más importantes. Se puede observa que cada sistema presenta características particulares que lo hacen destacar de los demás. también se aprecia que el tiempo de investigación invertido es muy significativo. por lo tanto, el presente proyecto es un buen comienzo para el desarrollo de estos sistemas en nuestro país y puede ser una estructura fundamental sobre la cual se basen trabajos posteriores.

Agora	Carnegie Mellon University	1987-	Sistema de MCD heterogéneo, permite la compartición de estructuras de datos entre máquinas.
Amber	University of Washington	1988-	Basado en objetos, compartición lograda migrando datos a los procesos.
Capnet	University of Delaware	1990-	Extensión de una MCD a una WAN.
Choices	University of Illinois	1988-	Sistema de MCD incorporado dentro de un un SOD orientado a objetos jerárquico.
Clouds	Georgia Institue of Technology	1987-	SOD orientado a objetos en el que éstos pueden migrar
Dash	Stanford University	1988-	Implantación de una MCD en Hardware con un protocolo de coherencia basado en directorios.
Emerald	of Washington University	1986-1988	Lenguaje y sistema orientado a objetos que soporta indirectamente MCD a través de la movilidad de objetos.
Ivy	Yale University	1984-1986	Sistema pionero de MCD en una red de estaciones de trabajo Apollo.
Linda	Yale University	1982-	Memoria asociativa de objetos compartidos con funciones de accesos. Puede ser suplementada para diferentes lenguajes y máquinas.
Memnet	University of Delaware	1986-1988	Implantación en hardware de una MCD en una red token-ring de 200Mbps usada para comunicar señales inválidas y requerimientos de lectura.
Mermaid	University of Toronto and Princeton University	1988-1991	Sistema de MCD heterogéneo en el que el compilador forza las páginas compartidas para incluir sólo un tipo de dato.
Mether	Supercomputing Research Center	1990	MCD transparente construida en SunOs 4.0 Permite aplicaciones para accesar un estado inconsistente para eficiencia.
Mirage	UCLA	1987-1990	Construcción a nivel del núcleo de una MCD. Reduce la pérdida de datos prohibiendo el robo de páginas antes de cumplir una cantidad de tiempo minima.
Munin	Rice University	1989	Sistema de MCD basada en objetos que investiga protocolos de coherencia específicos.
Modelo Objeto-de datos compartidos	University, Vrije Amsterdam	1988	Implementación de una MCD en la parte más alta del sistema operativo AMOEBA
Shiva	Princeton University	1988-	Sistema de MCD parecido a Ivy para el hipercubo de intel iPSC/2.

Fig. 1.2 Proyectos de MCD.

CAPITULO 2

EL SISTEMA OPERATIVO MACH

BIBLIOTECA

78990

2. El Sistema Operativo Mach

Introducción

Desde el desarrollo de UNIX en los laboratorios Bell no ha habido un sistema operativo que llame tanto la atención de la industria de la computación como el sistema operativo Mach. Este es el resultado de investigaciones en la Universidad de Carnegie Mellon (CMU) que empezaron a mediados de los 80´s y actualmente continuan. El objetivo de dichas investigaciones fue desarrollar un nuevo sistema operativo que permitiera a los programadores de computadoras explotar las modernas arquitecturas de computadoras producidas en compañías, universidades y laboratorios de investigación. Este sistema operativo fue diseñado para reducir el número de características en el núcleo y, por ello, a reducir el tamaño del mismo, lo cual se opone a la realización de sistemas operativos con muchas características y complejidad excesiva.

Mach fue desarrollado originalmente como una variante de UNIX BSD (Berkeley Software Distribuition) para proveer a los usuarios un manejador de memoria mejorado, múltiples puntos de control (hilos) y una extensa facilidad de comunicación interprocesos (IPC). Los objetivos incluían:

- Explotar el paralelismo tanto en sistema operativo como en las aplicaciones de usuario.
- Soportar grandes espacios de direcciones esparcidas con una flexibilidad para compartirlas.
- Permitir el acceso transparente a los recursos de la red.
- Adaptarse con el ambiente de software existente (BSD).
- Ser portable.

El sistema operativo Mach provee la base sobre la cual un sistema operativo puede ser construido. El núcleo de Mach sigue la siguiente filosofía:

- Un núcleo simple con amplios mecanismos de comunicación.
- Estructura basada en objetos, con canales de comunicación (puertos) como objetos de referencia.
- Un modelo de programación cliente-servidor usando comunicaciones interprocesos tanto asíncronas como síncronas .
- Tareas en modo usuario desempeñando muchas funciones tradicionales del sistema operativo. (por ejemplo, sistema de archivos, accesos a la red, etc).

El núcleo de Mach provee el soporte de la comunicación entre tareas, particularmente en las que se muestran a continuación:

- Administración de puntos de control (hilos).
- Asignación de recursos (tareas).
- Soporte del espacio de direcciones para las tareas.
- Administración de los recursos físicos (memoria física, procesadores, canales de dispositivos).

2.1 Arquitectura del modelo de Mach

Aunque es un objetivo del núcleo de Mach minimizar las abstracciones provistas por el núcleo, no es un objetivo reducir la semántica asociada con dichas abstracciones. De tal forma, cada una de las abstracciones tiene asignada una poderosa semántica asociada y un conjunto complejo de interacciones con las demás abstracciones. Las principales abstracciones del núcleo son las siguientes:

- Hilo. Es un punto de control de flujo en una tarea, tiene acceso a todos los elementos de la tarea que lo contiene, potencialmente se ejecuta en paralelo con otros hilos, aún con hilos de la misma tarea.

- Tarea. Es una colección de recursos del sistema. Estos recursos, con la excepción del espacio de direcciones de la tarea, son referenciados por medio de puertos. Estos recursos pueden ser compartidos con otras tareas si se distribuyen derechos a los puertos. Una tarea proporciona un espacio de direcciones potencial no contiguo, referenciado por las direcciones de la máquina. Las porciones de este espacio pueden ser compartidas por medio de herencia o administración de memoria externa. Además, cada tarea tiene un número dado de hilos.

- Puerto. Es un canal de comunicación unidireccional entre tareas.

- Conjunto de puertos. Es un conjunto de puertos que puede ser considerado como una unidad simple cuando recibe un mensaje.

- Derecho de puerto. Capacidad que permite derechos específicos de acceso a un puerto.

- Espacio de nombres de puertos. Es una colección indexada de nombres de puertos cada uno de los cuales nombra a un derecho de puerto en particular.

- Mensaje. Es una colección de datos pasados entre dos tareas.

- Cola de mensajes asociada a un puerto simple.

- Espacio de dirección virtual. Es un conjunto indexado de páginas de memoria (no contiguas) que pueden ser referenciadas por los hilos de una tarea. Los rangos de páginas pueden tener atributos arbitrarios y una semántica asociada realizada por medio de mecanismos del núcleo y por manejadores externos de memoria.

- Objeto abstracto de memoria. Un objeto de memoria contiene un rango de páginas cuyas propiedades son controladas por un paginador simple. Es una abstracción fundamental que es manipulada por un paginador.

- Objeto en memoria caché. Es un objeto del núcleo que contiene el estado residente de los rangos de memoria utilizados por un objeto abstracto de memoria.

- Procesador. Es un procesador físico capaz de ejecutar hilos.

- Conjunto de procesadores. Es un conjunto de procesadores, cada uno de los cuales puede ser usado para ejecutar los hilos asignados al conjunto de procesadores.

- Nodo. Es un punto individual perteneciente a una multicomputadora.

- Host. Es la computadora que ofrece el servicio.

- Dispositivo. Es un dispositivo físico accesible por tareas de modo usuario.

- Evento. Es un dispositivo del núcleo que mantiene la señalización de cuenta de eventos.

Por supuesto, el núcleo provee parte de la administración de memoria. La memoria está asociada con las tareas. Los objetos de memoria son los medios por los cuales las tareas accesan la

memoria y son la unidad básica para la administración de la memoria. Este punto se analiza en párrafos posteriores.

2.2 Hilos y tareas

Una de las principales características ofrecidas por el sistema operativo Mach es su ambiente de ejecución de programas. Este ambiente es muy avanzado y permite el desarrollo de aplicaciones para optimizar el desempeño a través de programas concurrentes denominados hilos. Un hilo pertenece exclusivamente a una tarea, la cual define su espacio de direcciones virtuales. Para explotar la concurrencia se comparten datos y otros recursos pertenecientes a los hilos de una manera eficiente y conveniente. Mach logra esta compartición agrupando recursos en una tarea. En una misma tarea se puede asociar cualquier número de hilos. Los hilos y las tareas permiten el diseño de aplicaciones que toman sin ningún cambio y de forma automática las ventajas de procesadores múltiples mientras corren en máquinas multiprocesador.

El procesamiento paralelo de información se realiza de una manera más natural en Mach con tareas e hilos que en el sistema operativo UNIX con procesos. La creación de hilos en Mach tiene un menor costo de tiempo de CPU que la creación de procesos en UNIX.

Una tarea es un ambiente de ejecución, las tareas por sí mismas no desempeñan acciones, sólo los hilos dentro de ellas lo pueden hacer. Los principales recursos asociados a una tarea son su espacio de direcciones, sus hilos, sus derechos de puerto, conjuntos de puertos y el nombre de espacio local en el cual se buscan los derechos de puerto y los conjuntos de puertos. A continuación se describen las operaciones para crear una tarea, las características necesarias para manipularla y la ejecución de los hilos que la componen.

2.2.1 Creación de una tarea

La creación de una tarea en Mach está basada en el mecanismo de creación de un proceso en UNIX, pero a diferencia de éste, las tareas en Mach no están relacionadas a la tarea que las crea. Por conveniencia, se refiere a la tarea creada como la tarea *hijo* y a la tarea que la creó como la tarea *padre*.

Contrariamente a UNIX, no existen relaciones de control prioritarias entre las tareas de Mach, aún entre tareas *padre* e *hijo*. Cuando coloquialmente se refiere a una tarea como si fuera un programa en ejecución, es decir, que una tarea *padre* crea una tarea *hijo*. Lo que realmente significa es que un hilo en ejecución en la tarea *padre* creó una tarea hijo. Una tarea recién creada no tiene hilos. En vez de esto, la tarea creadora realiza la petición de un hilo junto con la tarea *hijo*. Para la creación de una tarea y de un hilo, se utilizan llamadas al sistema que no se analizan en este documento, sin embargo pueden ser consultadas en [15],[26] y [27].

2.2.2 Invocación de operaciones del núcleo

Después de que se creó una tarea o un hilo, se asigna a un puerto nombrado *puerto de núcleo*. Las llamadas al sistema de Mach se dividen en dos: en las que son ejecutadas como "trampas" en el núcleo y en las que son ejecutadas por medio de paso de mensajes a puertos del núcleo. Este último grupo de llamdas permiten mantener la transparencia de las operaciones en la red entre tareas remotas, así como en las locales. Un servicio del núcleo administra los recursos del núcleo de la misma manera que un servidor a nivel usuario administra otros recursos. Cada tarea tiene derechos de envío a su puerto de núcleo, el cual la habilita para llamar a nuevas operaciones por sí misma, como la creación de un nuevo hilo. El acceso a los servicios del núcleo se realiza a través del paso de mensajes a sus diferentes interfases de servicio. Las tareas pueden accesar dichos servicios por medio de procedimientos fragmentados que están generados conforme a sus interfases de definición y pueden ser consultadas en [28].

2.2.3 Procesamiento de excepciones

Además del puerto del núcleo asociado a una tarea o a un hilo. puede existir un puerto denominado *puerto de excepciones.* Cuando existe un tipo de excepción, el núcleo responde intentando el envío de un mensaje descriptor del tipo de excepción a un puerto de excepción dado. Si no existe ningún puerto de excepción asociado al hilo, entonces el núcleo busca un puerto asociado a la tarea. El hilo que recibe el mensaje puede intentar resolver el problema, y regresar el valor del estado en un mensaje de respuesta. Si el núcleo encuentra un puerto de excepción y recibe una respuesta exitosa. entonces reinicia al hilo que generó la excepción. De otra forma lo termina. Ejemplos del manejo de excepciones pueden ser: cuando una tarea intenta un acceso de dirección de memoria no válida o cuando se presente una división por cero en alguna operación aritmética. El propietario del puerto de excepción podría ser una tarea de compilación ejecutada en cualquier punto de la red. debido a la característica de Mach de comunicación independiente del lugar. Los fallos de página se administran por paginadores externos.

En la presente tesis se desarrolla la especificación formal de un administrador de memoria externo, el cual incluye entre uno de sus elementos a un paginador. Esto será descrito en los capítulos 4 y 5.

2.2.4 Administración de tareas e hilos

En el sistema operativo Mach existen un gran número de procedimientos relacionados con la creación y administración de tareas e hilos [15],[26] y [27]. El primer argumento de cada procedimiento es un derecho de envío al puerto del núcleo correspondiente, y las llamadas al sistema de paso de mensajes son usadas para generar requerimientos al núcleo destino. Las prioridades de planeación de hilos pueden ser activadas de forma individual, las tareas y los hilos pueden ser suspendidos, restaurados y terminados; y el estado de ejecución de los hilos puede ser activado, leído o modificado externamente. Esta última característica es importante para la compilación, también para la configuración y la atención a interrupciones de software. Además,

compilación, también para la configuración y la atención a interrupciones de software. Además, algunas llamadas de interfaz al núcleo están relacionadas con la asignación de los hilos de una tarea a conjuntos de procesadores específicos. Mediante la asignación de los hilos a conjuntos de procesadores, se pueden dividir los recursos computacionales disponibles entre diferentes tipos de actividades.

2.3 Modelo de comunicación

El sistema operativo Mach cuenta con una llamada al sistema, denominada *mach_msg*, para llevar a cabo el paso de mensajes. Para comprender claramente como se utiliza dicha llamada es necesario explicar antes algunas características de los elementos que se manejan en ella, es decir, mensajes, puertos y conjuntos de puertos.

Los mensajes en Mach consisten de un encabezado de tamaño fijo y de múltiples bloques de datos de tamaños variables, algunos de los cuales pueden ser *datos fuera de línea* (es decir, no contiguos). Cuando llega un mensaje de aviso de que se han enviado datos no contiguos, el núcleo escoge de forma automática la localidad en el espacio de direcciones de la tarea dentro de la cual almacena dichos datos. Este es un efecto lateral de la técnica de copia en escritura usada para transferir los datos. Las regiones extra de memoria virtual que son recibidas en un mensaje deben ser desalojadas de la tarea receptora cuando ya no son requeridas.

La ventaja de permitir distintos tipos de datos en mensajes es que facilitan al programador asignar memoria separada para datos y "metadatos". Por ejemplo, un servidor de archivos puede asignar un bloque de disco requerido de su caché. En lugar de copiar el bloque en el buffer de un mensaje, se buscan los datos directamente junto con la información del encabezado del mensaje, proporcionando un apuntador en el mensaje de respuesta, -sólo si el cliente está localizado en la misma máquina-. Esta es una forma de lo que se conoce como "*scatter-gather I/O*" (E/S dispersa-

conjunta), en la que los datos se leen o escriben de múltiples áreas del espacio de direcciones del solicitante en una llamada al sistema.

Un puerto tiene una cola de mensajes que cambia su tamaño dinámicamente. Sólo la tarea que tiene los derechos de recepción puede retirar los mensajes de un punto. Esta característica habilita que los receptores puedan efectuar una forma de control de flujo.

Cuando se utiliza un derecho de envío normal, el hilo que está intentando enviar un mensaje a un puerto cuya cola de mensajes está llena, se bloquea hasta que exista espacio disponible. Cuando se utiliza el derecho de envío de un mensaje de "sólo una vez", se forma el mensaje en la cola aunque esté llena. En el instante en que se forma un mensaje en una cola llena, no pueden provenir más mensajes de esa fuente. Para evitar el bloqueo, los hilos del servidor envían mensajes de "sólo una vez" al instante de mandar sus respuestas a los clientes. Cuando los derechos a puertos son enviados dentro de un mensaje, el receptor adquiere los derechos de envío al mismo puerto. Cuando se transmiten los derechos de envío, éstos son despojados de la tarea transmisora automáticamente. Esto se debe a que dos tareas no pueden tener los derechos de envío simultáneamente. Todos los mensajes encolados en el puerto y todos los posteriores mensajes transmitidos pueden ser recibidos por el nuevo dueño de los derechos, de tal forma que esto sea transparente a las tareas que enviaron información al puerto. La transferencia transparente de los derechos de envío es fácil de interpretar en una sola computadora. Esta característica es simplemente un apuntador a la cola de mensajes local. Sin embargo, cuando se trata el caso de varias computadoras, se requiere el diseño de algoritmos complejos.

El núcleo está diseñado para informar a los transmisores y receptores cuando llegan las condiciones bajo las cuales se facilite el envío y recepción de mensajes. Para este propósito, el núcleo mantiene información acerca del número de derechos de envío y recepción correspondientes a un puerto dado. Si ninguna tarea tiene derechos de recepción en un puerto particular (por ejemplo, cuando la tarea que tiene los derechos falle), entonces, todos los derechos de envío en espacios de nombres de puertos de tareas locales se convierten en "nombres muertos". Cuando un transmisor intenta usar un nombre referido a un puerto en el cual no existen derechos de recepción, el núcleo cambia el nombre en un nombre muerto y regresa una indicación

de error. Similarmente, las tareas pueden pedir al núcleo que les notifique asíncronamente la confirmación de que no existen derechos de envío para un puerto específico. El núcleo realiza esta notificación enviando un mensaje al hilo solicitante, usando los derechos de envío dados a él.

Podría afirmarse que las condiciones de *no-transmisores/no-receptores* pueden ser atacadas en el dominio de un solo núcleo a un relativo bajo costo. Sin embargo, para aquellas condiciones que se presentan en un sistema distribuido, es una operación compleja y cara.
Transferir los derechos se logra con el intercambio de mensajes; los derechos de envío y recepción para un puerto dado pueden estar en cualquier tarea, o aún en algún mensaje, encolados en un puerto o tránsito entre computadoras.

2.4 Realización de la comunicación

Uno de los aspectos más interesantes del desarrollo de comunicaciones en Mach es la utilización de servidores de red a nivel usuario. Los servidores de red (nombrados servidores de mensajes de red "*netmsgservers*"), uno por computadora, son responsables de extender a través de la red la semántica de la comunicación local; esto incluye la preservación, tanto como sea posible, garantía de liberación y transparencia de la comunicación de la red. También contiene los efectos y el monitoreo de la transferencia de derechos de puerto. En particular, los servidores de red son responsables de proteger los puertos contra accesos ilegales, y de mantener la privacidad de los mensajes de datos a través de la red.

Para obtener transparencia en la entrega de mensajes es necesario agregar una abstracción externa denominada puerto de red, por medio de la cual pueden ser direccionados los mensajes a través de la red. Un puerto de red es un identificador único de canal que sólo es manejado por los servidores de red, y es asociado por ellos con un sólo puerto en cualquier instante. Los servidores

de red poseen derechos de envío y recepción a los puertos de red, de la misma forma que las tareas poseen derechos de envío y recepción a puertos de Mach.

Otra característica buscada para mejorar la transparencia de migración es como sincronizar la entrega de mensajes. En este sistema operativo se garantiza que cuando se envían dos mensajes, provenientes del mismo hilo, éstos son entregados en el orden de envío: en el caso en el que los derechos de recepción también están siendo transmitidos se podría presentar alguna dificultad -la pérdida de mensajes o el orden de los mismos- si no se tiene cuidado. debido a que un mensaje podría ser entregado por un servidor de mensajes de red antes de que sea atendido un mensaje de mayor prioridad encolado en la computadora original. El servidor de red logra esta garantía deteniendo la liberación de todos los menajes de la computadora original hasta que hayan sido transferidos todos los mensajes encolados de la computadora destino. La entrega de mensajes también puede ser reenrutada de forma segura, y la subsecuente dirección regresada a los transmisores.

2.5 Administración de memoria

El núcleo de Mach provee los mecanismos para soportar grandes espacios de direcciones virtuales. Cada tarea tiene asociada una tabla de direcciones (mantenida por el núcleo). la cual controla la traslación de direcciones virtuales al espacio de direcciones de una tarea dentro del direccionamiento físico. Los mecanismos como el caché, para usar la memoria física, no están desarrollados por el núcleo de Mach, sino que el núcleo permite a las tareas de modo usuario la habilidad de participar en estos mecanismos.

La memoria virtual es referenciada mediante direccionamiento virtual; las direcciones virtuales pueden verse como índices dentro de un espacio de direcciones de una tarea en particular. La memoria de una tarea puede ser parcial o totalmente compartida con otras tareas. En Mach la

implementación de memoria compartida por varias tareas relacionadas se establece mediante el mecanismo de copia sobre escritura -*copy on write* (COW)-.

A una tarea, se le pueden asignar, desasignar y cambiar las protecciones sobre los rangos de memoria de su espacio de direcciones.

Cuando una región de memoria está asignada significa que está ocupada por un objeto de memoria. Un manejador de memoria provee las políticas que gobiernan la relación entre la imagen de un conjunto de páginas mientras se encuentren en memoria real y la imagen de un conjunto de páginas cuando no lo están. Mach provee la facilidad de desarrollar una aplicación a nivel programador capaz de controlar operaciones de paginación de memoria virtual, tradicionalmente reservadas al sistema operativo. El usuario puede crear un paginador que controle el uso de memoria perteneciente a porciones del espacio de direcciones de una tarea. Un paginador puede controlar el uso de memoria a través de múltiples máquinas, proporcionando memoria virtual compartida a través de la red. El control de operaciones de paginación de una tarea a nivel usuario se denomina administrador de memoria externo (AME).

En este trabajo se analiza el comportamiento de un administrador de memoria externo para proveer una memoria compartida distribuida.

2.6 Administración de memoria virtual

2.6.1 Conceptos básicos del modelo de memoria virtual en Mach

Cada tarea en Mach tiene su propio espacio de direcciones que puede ser compartido completamente por todos los hilos pertenecientes a ella. Las localidades de memoria

pertenecientes a un espacio de direcciones están numeradas secuencialmente de un límite bajo a uno alto impuestos por el sistema operativo o por el hardware. Cada localidad contiene un byte.

La mayoría de lo microprocesadores más populares ofrecen un espacio de direcciones de 32 bits, es decir, 2^{32} bytes direccionables. Mach fue diseñado para manejar grandes espacios de memoria eficientemente. Típicamente, estos espacios son dispersos, lo que significa que puede haber memoria asignada separada por grandes huecos entre localides. El programador puede fácilmente asignar cantidades de memoria de tamaños pequeños hasta muy grandes, sin la necesidad de que sean contiguas.

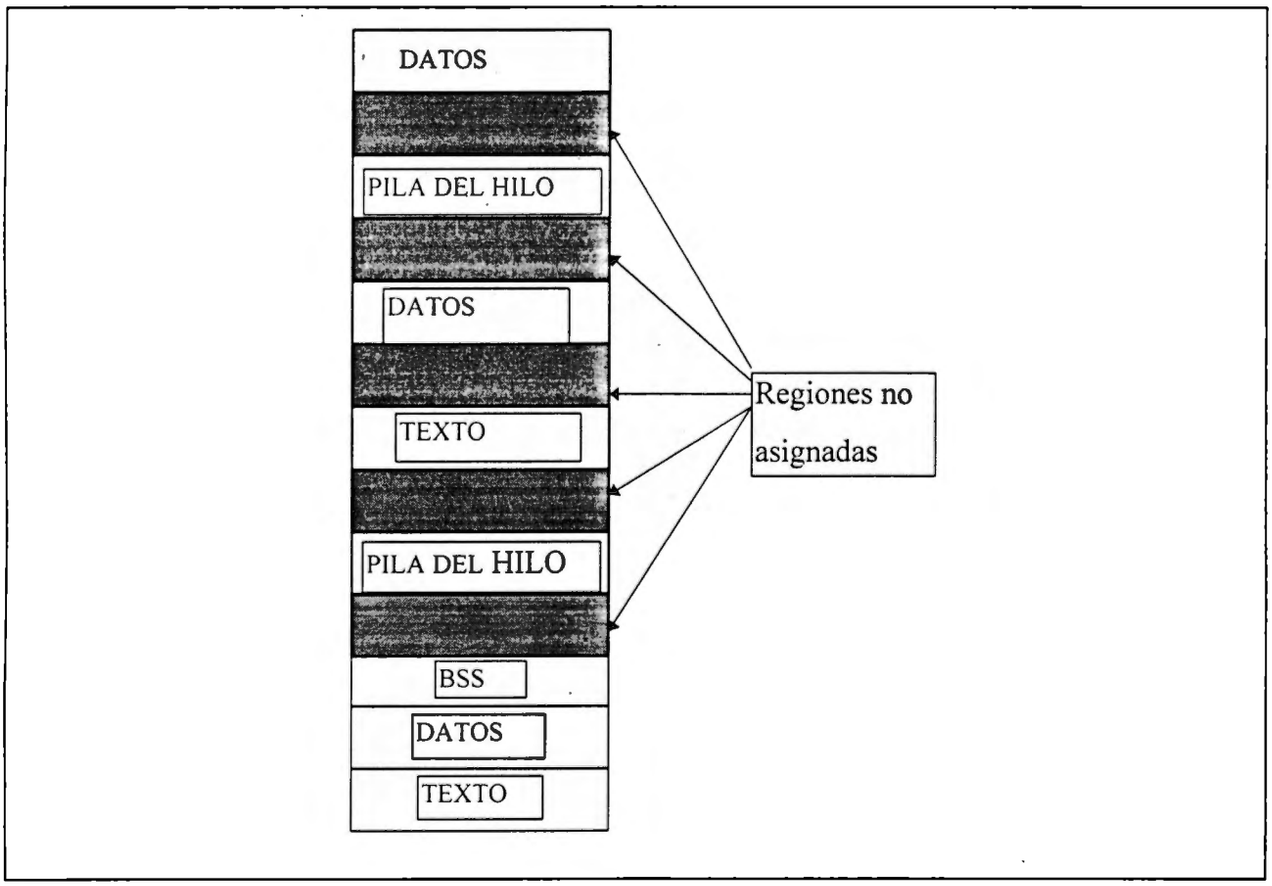


Fig. 2.1 *Modelo de memoria de un programa en Mach.*

En la figura 2.1 se muestra un modelo de memoria de un programa en Mach, en ella se aprecia como existen regiones de memoria no asignadas o dispersas en la memoria de un programa. La memoria de un programa está ocupada por los siguientes elementos: (indicando la numeración de las direcciones de forma ascendente de abajo hacia arriba) el texto del programa (típicamente de

sólo lectura, para que pueda ser compartido), el área de datos inicializados, el área de datos no inicializados, la pila del hilo, texto extra, datos extra, pila del hilo y datos extra, respectivamente. Mientras que los hilos de una tarea comparten todo el espacio de direcciones de la misma, diferentes tareas también pueden compartir porciones de sus espacios de direcciones. Normalmente, esta compartición se lleva a cabo de una tarea padre a una o más tareas hijas.

2.6.2 Rangos y tamaños de página

Las primitvas de memoria virtual en Mach toman como argumentos un rango del espacio de direcciones, especificado por una dirección inicial y una longitud en bytes. Por esto, se ve a la memoria como un arreglo lineal de bytes en el que se pueden ejecutar operaciones en cualquier secuencia de bytes. El núcleo de Mach siempre manipula la memoria en unidades de página. Una operación en un rango de memoria arbitrario se convierte a una operación en un rango de página. Por ejemplo, considérese que se tiene un requerimiento que empieza en alguna *dirección* y se extiende en un tamaño de *n bytes*, como se presenta en la figura 2.2. Mach convierte la *dirección* dada a una dirección alineada en páginas, truncándola a una cota de página. El sistema operativo Mach opera con rangos de página porque el hardware no puede operar con unidades más pequeñas que una página.

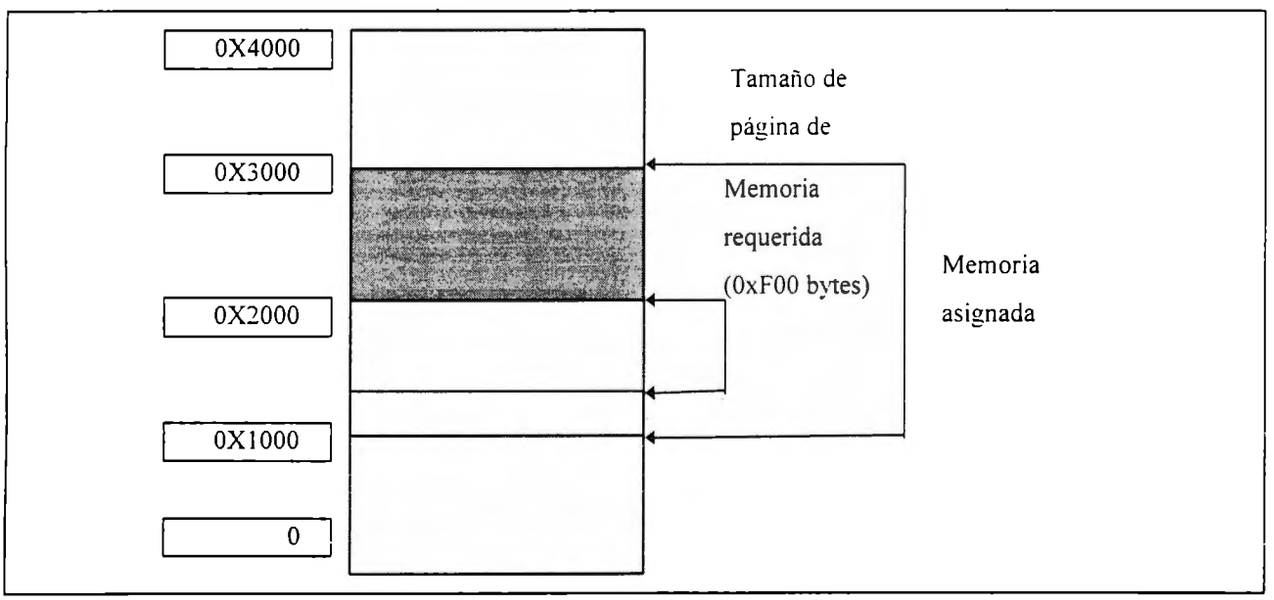


Fig. 2.2 Mach convierte rangos de memoria a rangos de páginas

El diseño de memoria virtual dentro de Mach se hace en dos capas: una dentro de la parte dependiente de la máquina (ver la figura 2.3) y otra dentro de la independiente. La parte dependiente provee una interfaz simple, validando, invalidando y determinando los derechos de acceso para páginas de memoria virtual (manteniendo las tablas de direcciones de hardware). La parte independiente de la máquina provee soporte para tablas de direcciones lógicas (mapeo de espacio de direcciones virtuales), rangos de memoria dentro de esta tabla y la interfaz de recuperación de almacenamiento (objetos de memoria) para esos rangos de memoria, mediante la interfaz del administrador de memoria externo.

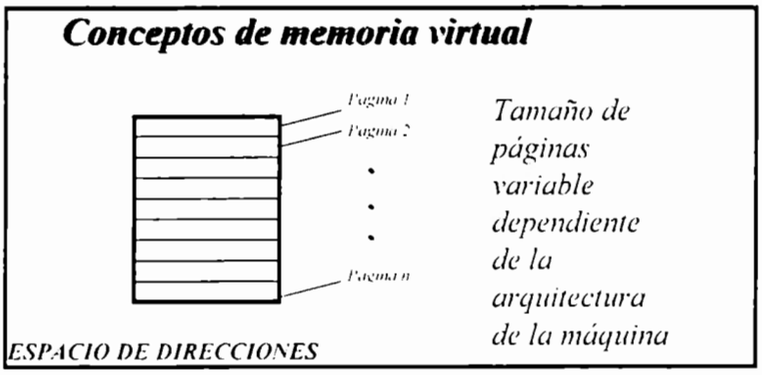


Fig. 2.3 Memoria virtual

El sistema de memoria virtual está diseñado para un procesador y un número moderado de procesadores de memoria compartida. Esto ha sido transportado a una arquitectura de memoria de acceso no uniforme, aunque el soporte óptimo para estas arquitecturas es tan bueno como uno con mapeo de hardware más complejo, como el caché virtualmente direccionado.

El alto rendimiento es una característica del diseño de memoria virtual de Mach, obtenido gracias a que soporta espacios grandes y esparcidos de direcciones, (ver en la figura 2.4) aprovechando esta virtud para ofrecer memoria compartida y optimización de la capa de memoria virtual.

En la figura 2.4 se muestran dos abstracciones de la memoria virtual. En el bloque A se presenta una memoria virtual, la cual se construye de diferentes regiones de memoria que pueden ser contiguas o no. En el bloque B, se observa una región de memoria virtual constituida, en este caso, por direcciones de memoria contiguas.

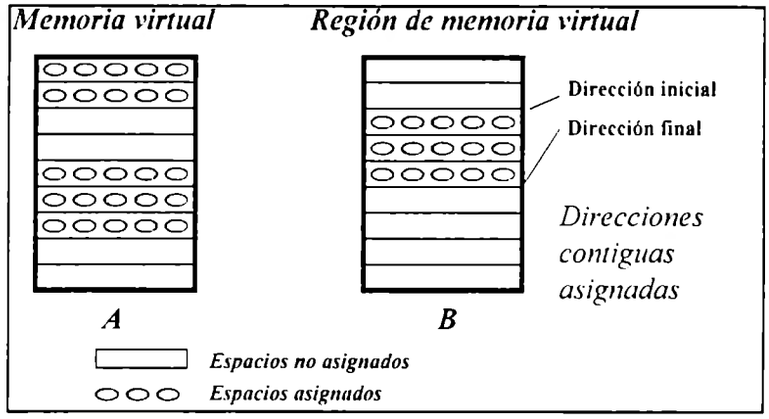


Fig. 2.4 Abstracciones de memoria virtual.

Otra característica del sistema de memoria virtual es permitir a los clientes la liberación de rangos de memoria.

2.6.3 Espacio de direcciones virtuales

Un espacio de direcciones virtuales define el conjunto de direcciones virtuales válidas a las que un hilo de una tarea en ejecución puede hacer referencia. Un espacio de direcciones virtuales es nombrado por su propia tarea.

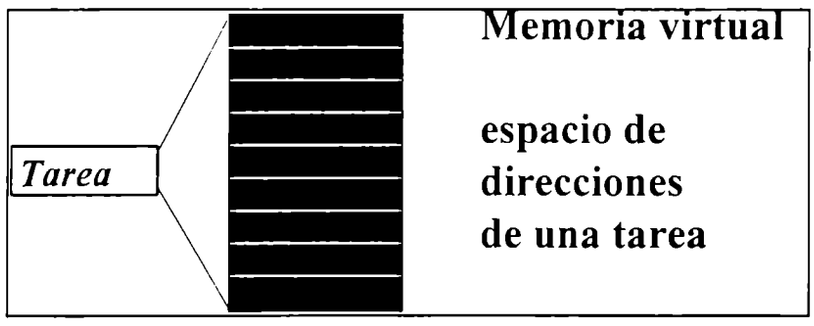


Fig. 2.5 Espacio de direcciones

En la figura 2.5, se representa el espacio de direcciones de una tarea, la memoria virtual se constituye del conjunto de éstos.

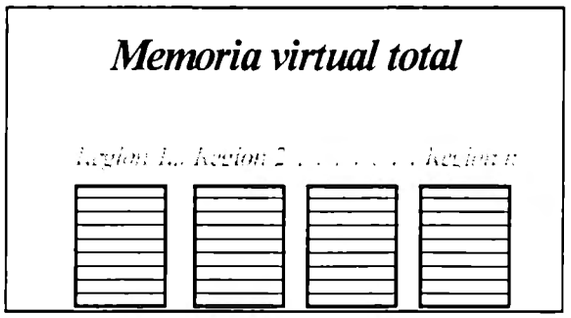


Fig. 2.6 Regiones de una memoria virtual

Un espacio de direcciones virtuales consiste de un conjunto indexado de páginas de memoria, éste puede ser continuo o no. Una región de un espacio de direcciones es la memoria asociada a un rango continuo de direcciones, tiene una dirección inicial y una dirección final, ver figura 2.6. El núcleo en forma interna agrupa virtualmente conjuntos contiguos de páginas que tienen los mismos atributos (objetos de memoria abstractos, herencia, protección y otras propiedades) dentro de regiones de memoria. Estas son nombradas sólo por sus rangos de direcciones virtuales junto con sus espacios de direcciones. Varias operaciones y mecanismos del sistema están dedicados a la identificación de regiones de memoria, a la protección de los accesos de usuarios y a la prestación de servicios que permitan a las tareas cliente expandir dichas regiones. El núcleo es libre para dividir y proporcionar regiones de memoria.

Un espacio de direcciones virtuales es creado cuando se genera una tarea y es destruido cuando la tarea termina. Cuando una nueva tarea es creada (ignorando la herencia), su espacio de direcciones está vacío y debe de ser construido a través de la manipulación del espacio de direcciones virtuales.

Un rango de memoria puede tener asociadas distintas semánticas a través de un manejador de memoria. Cuando es establecido un nuevo rango de memoria en un espacio de direcciones virtuales, se especifica un objeto abstracto de memoria (posiblemente de manera automática) que representa la semántica del rango de memoria, siendo asociado con una tarea (un manejador de memoria) la cual provee dicha semántica. Para una tarea no existen llamadas al núcleo que afecten directamente la semántica asociada con su rango de memoria. Una tarea tiene dicho control sólo por la virtud de escoger los manejadores de memoria que proporcionen la

funcionalidad deseada, o directamente por el envío de mensajes a sus manejadores de memoria para dirigir sus acciones.

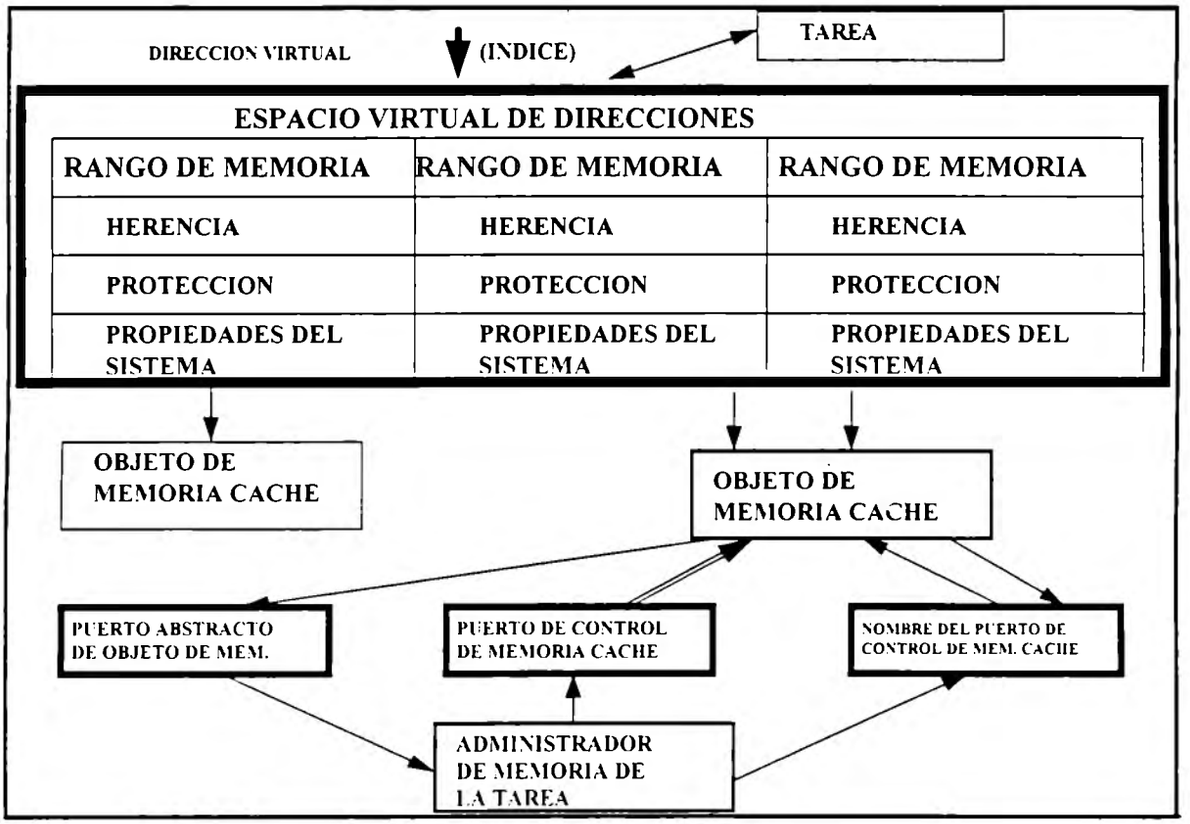


Fig.2.7 Rangos de memoria de una tarea.

En la figura 2.7 una tarea especifica una dirección virtual dentro de su espacio de direcciones por medio de un registro índice. Dentro de su espacio virtual de direcciones existen diferentes rangos de memoria asignados a objetos de memoria caché; en cada uno de ellos se obtienen características particulares como herencia, protección y propiedades del sistema (estrategias de copia, por ejemplo). Se representa a un objeto de memoria caché que tiene asociado un rango de memoria y a otro con dos rangos de memoria (para expresar que pueden tener tamaños variables). Así mismo, se expresa el mecanismo para la obtención del objeto de memoria caché. Un administrador de memoria externo accesa a un puerto de control de un objeto de memoria caché (proporcionando el número de puerto y su nombre), con la finalidad de obtener el puerto del objeto, y recibe el puerto abstracto asignado al objeto de memoria.

Junto con los accesos elementales a memoria permitidos por el hardware, el núcleo también permite manipulaciones de memoria abstractas. Las operaciones permitidas para la manipulación de un espacio de direcciones virtuales son las siguientes [26]:

- Copiar un rango de memoria de un lado a otro en un espacio de direcciones virtual.

- Activar propiedades de hardware (específicas de la máquina) para el rango de memoria.

- Activar los accesos permitidos para un rango de memoria. Cada rango de memoria tiene una máscara de protección actual y máxima. El manejador de memoria de un rango puede especificar las protecciones máximas para todos los usuarios de un rango de páginas: cada tarea tiene su propio valor de protección máximo para restringir aún más los permisos válidos. así como su máscara de protección actual.

- Copiar un rango de memoria para lectura.

- Asignar como fijo un rango de memoria. La habilidad de activar la paginación de memoria es una operación privilegiada y, por lo tanto requiere el puerto de control del servidor.

- Copiar un rango de memoria para escritura.

2.6.4 Objetos de memoria

Un objeto de memoria es un elemento de almacenamiento secundario para el cual existe una correspondencia con el espacio de direcciones de una tarea. Un ejemplo es un archivo administrado por un servidor, pero un objeto de memoria puede ser cualquier objeto para el cual puedan manejarse solicitudes de lectura y escritura. Un objeto de memoria está representado por un puerto (o varios), y los mensajes de comunicación entre procesos se envían al puerto para solicitar operaciones sobre el objeto. Puesto que Mach usa comunicación interprocesos. los objetos de memoria pueden residir en sistemas remotos y lograr el acceso a ellos de manera transparente.

2.6.5 Manejo básico

Mach ofrece a los programas de usuario mayor flexibilidad en la administración de memoria.

El manejo de un espacio de direcciones virtuales por una tarea en modo usuario toma la forma básica siguiente :

- Una tarea reserva un nuevo rango de memoria dentro de su espacio de direcciones. Mach asigna un puerto de nombre a este rango, el cual representa al objeto de memoria.

- La tarea intenta hacer referencia a una porción de este rango de memoria y dado que esta porción todavía no está asignada en memoria, causa un fallo de página no residente. El núcleo envía un mensaje al paginador responsable del objeto abstracto de memoria requiriendo la asignación de los datos en memoria. La respuesta del paginador indica la solución del fallo de página causado por la tarea.

- Eventualmente, las páginas residentes del rango de memoria, con valores posiblemente modificados por tareas cliente, son "desalojadas" de la memoria. Las páginas son enviadas en mensajes al paginador del objeto de memoria para su disposición.

- La tarea cliente libera el rango de memoria. Cuando todos los mapas de este objeto de memoria son destruidos, éste se termina.

El núcleo puede ser visto como un consumidor de memoria principal o como un consumidor de memoria caché para el almacenamiento de varios objetos de memoria. La porción de esta memoria caché, que contiene páginas residentes de un objeto de memoria, es conocida como objeto de memoria caché. El núcleo mantiene un diálogo con varios manejadores de memoria para que exista disponibilidad en cualquier región de la misma. En general, este diálogo consiste de mensajes asíncronos, dado que el núcleo no puede ser bloqueado por un manejador de memoria, y éste último desea la máxima concurrencia posible en sus operaciones.

El puerto de un objeto abstracto de memoria del cliente está asociado a un manejador de memoria que administra el objeto abstracto de memoria. Cada uno de los puertos tiene asociado un objeto de memoria caché en el núcleo en que reside, el cual representa las páginas caché residentes del objeto de memoria. Este objeto tiene un puerto de control asociado, el cual es suministrado por el manejador de memoria de manera que permite controlarlo (para responder a las solicitudes del núcleo en nombre del objeto). El núcleo también genera un puerto de nombre para este objeto de memoria, para uso de la tarea cliente y para referirse al objeto. Si un objeto abstracto de memoria es buscado por la tarea en más de un servidor, éste tendrá muchos puertos de nombre y de control, uno para cada objeto de memoria caché en cada servidor.

Cada página en un objeto de memoria caché representa un desplazamiento perteneciente al objeto abstracto de memoria. Un objeto de memoria puede tener copias múltiples de sus datos en memoria para valores de desplazamiento diferentes que sean especificados por sus clientes.

2.6.5.1 Inicialización de administración de memoria

Cuando se mapea por primera vez un objeto abstracto de memoria en un servidor, el núcleo envía un mensaje de inicialización a su objeto de memoria. Este mensaje informa al manejador de memoria que el objeto está siendo mapeado en un servidor nuevo. El mensaje transporta los nombres del puerto de control del objeto de memoria caché generado por el núcleo y el nombre del puerto del objeto de memoria. En seguida, responde el manejador de memoria al núcleo indicándole el tipo de manejador del que se trate (antiguo o nuevo, conforme a las adiciones de mecanismos que se han hecho en las nuevas versiones de Mach, [26]), envía los atributos del objeto que lo identifican como caché y define las estrategias de copia. Estas llamadas de inicialización informan al núcleo que el manejador de memoria se encuentra listo para responder a los requerimientos en nombre de algún objeto de memoria.

2.6.5.2 Finalización del objeto de memoria

Cuando todas las tareas remueven un objeto de memoria de sus espacios de direcciones, el núcleo envía al manejador de memoria un mensaje para la terminación del objeto. Alternadamente, el manejador de memoria realiza las funciones necesarias para dar de baja al objeto de memoria. Descarta todas las páginas de memoria residentes y no permite más actividad para el objeto de memoria. El hecho de liberar el puerto del objeto abstracto de memoria tiene también el mismo efecto. En cualquiera de los casos el núcleo descarta todas las páginas.

La terminación de un objeto de memoria también depende de los atributos asignados al objeto en el momento de inicialización, cuando éste fue iniciado con los atributos de objeto caché, entonces el núcleo debe esperar para terminarlo, en caso contrario puede eliminarlo inmediatamente.

2.7 Administradores de memoria y paginadores

Tradicionalmente un programador no debe preocuparse por la manipulación de los espacios de direcciones de las tareas generadas por las operaciones de asignación y liberación de páginas. Con esta consideración, el sistema operativo Mach funciona como un sistema operativo tradicional, por ejemplo UNIX, manipulando dichas operaciones basado en una política que ofrezca transparencia al usuario. Mach tiene dos paginadores internos denominados "*default*" y "*unode*" [15] para manejar los requerimientos de paginación. Sin embargo, a diferencia de UNIX, Mach también permite paginadores a nivel usuario (conocidos como externos) para manipular las regiones de memoria. Los paginadores internos están construidos dentro de Mach; los paginadores externos son provistos por programas a nivel usuario. Un espacio de direcciones de una tarea puede tener cualquier número de paginadores internos o externos que manipulan piezas dispersas de él.

Mach soporta distintos paginadores por varias razones; una de ellas es proveer interfases generales disponibles al programador de aplicaciones. Mach evita reducir la necesidad de ofrecer más características de propósito general de compartición de memoria o de control de paginación. El usuario puede controlar directamente las operaciones de compartición de memoria y de paginación. La memoria principal puede ser utilizada como caché para objetos a nivel usuario, tales como bases de datos o archivos. En lugar de forzar a alguna aplicación para manejar cachés estáticos con los contenidos del objeto, el objeto completo puede estar disponible como un rango de páginas. El núcleo está dedicado a resolver el problema de obtener los datos cuando son accesadas las páginas y de guardarlos cuando ya no se utilizan las mismas.

El núcleo de Mach puede distinguir que tipo de paginador utilizar en una operación gracias al intercambio de mensajes de identificación y autentificación entre un paginador y un núcleo, respectivamente.

2.7.1 Administrador de memoria estándar

El administrador de memoria estándar es, en términos generales, una entidad que provee almacenamiento de memoria. Una propiedad significativa de este administrador es que representa el último recurso para la administración de la memoria, es decir, no puede fallar en caso de que los demás administradores no estén operando. La ruta de datos de escritura de este administrador está conectada directamente a la memoria, y toda la memoria que maneja también lo está, esto significa que es un administrador de memoria primaria.

Para inicializar un administrador de memoria estándar en Mach, se debe generar un mensaje solicitándolo.

La memoria utilizada por el paginador estándar puede ser creada de diversas maneras. La creación no lo involucra directamente, de tal forma que el núcleo debe informarle explícitamente acerca de la creación de nuevos objetos de memoria estándar. Esto se logra por medio del intercambio de mensajes de creación de un objeto abstracto de memoria.

2.7.2 Paginadores de memoria externos

La filosofía del micronúcleo de Mach no soporta archivos y otras abstracciones de almacenamiento externo directamente. En lugar de ello, esos recursos son controlados por paginadores externos. Mach tiene un modelo de acceso mapeado para recursos de memoria. Otra ventaja de Mach es que ofrece facilidades para el desarrollo de aspectos distribuidos en la realización de una memoria virtual. Estas facilidades consisten principalmente en el protocolo que existe entre el núcleo y un paginador externo, necesario para administrar el mapeo de los datos almacenados por éste último.

Las principales funciones de un paginador externo son:

1. Almacenar datos que son controlados por el núcleo en sus páginas caché.
2. Suministrar datos de páginas según sean requeridos por algún núcleo, e
3. Implantar las restricciones de consistencia, pertenecientes a la capa que representa la abstracción de recursos de memoria, en el caso en el que el recurso de memoria sea compartido y muchos núcleos puedan tener simultáneamente objetos de memoria caché del mismo objeto de memoria.

2.8 Memoria compartida distribuida en Mach

Se concluye la discusión de los administradores de memoria enfocándose en el desarrollo de una memoria compartida distribuida por medio de un administrador de memoria externo. Como se ha mencionado continuamente, un objeto de memoria puede ser utilizado por varias máquinas simultáneamente. Este es uno de los principales beneficios de un administrador de memoria externo. Los clientes distribuidos alrededor de la red, así como los clientes compartiendo la misma máquina, pueden tener la misma vista de una memoria compartida. Existen diversas formas para hacer que un objeto de memoria esté disponible en cualquier máquina, una forma de hacerlo es registrar un paginador en un servidor de mensajes de red (parte del núcleo), el cual puede responder a requerimientos de cualquier máquina en una red de área local. Desafortunadamente, las interfases del administrador de memoria externo no proveen por sí mismas una vista de memoria compartida. En la figura 2.8 se muestra un ejemplo que corresponde a la situación en la cual se construye un paginador que no soporta memoria compartida distribuida. Cualquier paginador que permite accesos a las páginas de sus objetos, sin recordar que núcleo está utilizando dichos objetos, no permite memoria compartida distribuida.

En la figura 2.8, el núcleo A obtiene una página de escritura en los pasos 1 y 2, la cual es modificada en el paso 3. Mientras tanto, el núcleo B intenta accesar la misma página para acceso de lectura. Debido a que el paginador no sabe quien tiene sus páginas, permite el requerimiento del núcleo B en el paso 5. Los clientes del núcleo B ahora tienen una copia inválida de la página, debido a que su copia de la página no incluye las modificaciones hechas en el paso 3. Eventualmente, el núcleo A envía de regreso la página modificada al paginador en el paso 7, pero en ese instante el daño ya fué ocasionado. Sin embargo, el daño podría ser mayor: Si el núcleo B modifica subsecuentemente su página, la liberación de la página del núcleo B sobreescribirá los resultados de la liberación de la página del núcleo A.

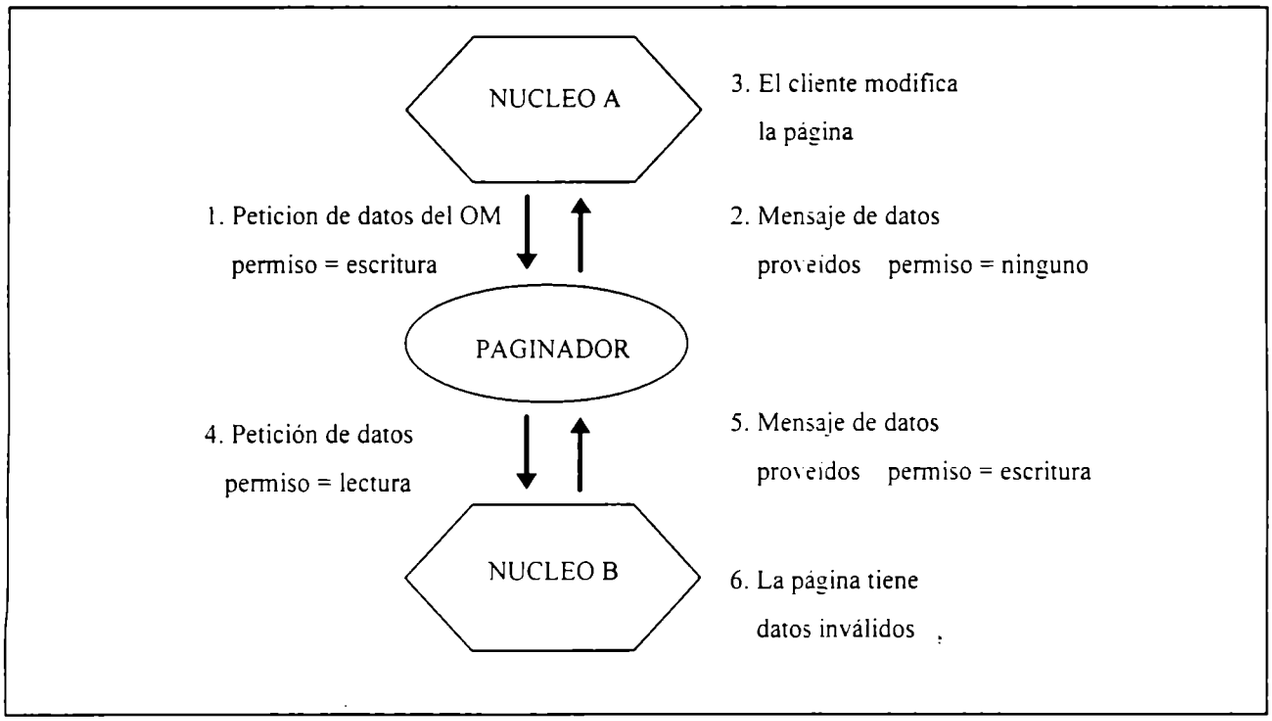


Fig. 2.8 Paginador sin memoria compartida distribuida.

2.9 Consistencia o coherencia de páginas

El mantener una vista predecible de la memoria a través de las máquinas se denomina consistencia de página o coherencia de página. Si un paginador realiza coherencia *completa (full)*, los datos modificados por una máquina serán visibles inmediatamente a todas las otras máquinas que comparten la página. Alternadamente, un paginador puede realizar consistencia *incompleta (loose)*, en la cual las actualizaciones de página son detenidas hasta que sean absolutamente necesarias, o hasta que sean inicializadas por un mecanismo especial. Por ejemplo, el uso de una instrucción de candado de hardware especial puede inicializar un mecanismo de coherencia. En tanto, como todos los clientes usen instrucciones de candado apropiadamente, la memoria aparecerá consistente. Unicamente los clientes que cometan errores al utilizar las instrucciones de candado podrían ver valores inconsistentes en la memoria compartida. Un paginador que no

desempeña coherencia puede perder algunas o todas las modificaciones hechas en los datos por diferentes máquinas. como se muestra en la figura 2.8.

El principal mecanismo usado en la coherencia de páginas es la manipulación de páginas de escritura. Las páginas de sólo lectura no pueden ser modificadas. de tal forma que su uso no causa un problema en la integridad de la memoria. Sin embargo. cuando un cliente tiene acceso de escritura a una página que otro está leyendo o cuando dos clientes tienen acceso de escritura a una página. un paginador de coherencia. debe tener alguna forma de prevenir resultados inconsistentes.

Un método popular para mantener la coherencia de páginas es el de *múltiples lectores/un sólo escritor*. Con este protocolo. se puede garantizar la integridad de memoria. La base del algoritmo es como sigue: solamente un núcleo a la vez puede tener una copia de escritura. Antes de que se de un acceso en escritura a una página en algún núcleo. el núcleo que actualmente está escribiendo la página debe negar su acceso. Aún. si un núcleo intenta adquirir una página para acceso de lectura. el núcleo con derechos de escritura (si es que existe) debe negar la página. Unicamente se permiten lecturas múltiples para utilizar una página en un mismo instante.

En la figura 2.9 se ilustra una parte del protocolo de coherencia *múltiples lectores/un solo escritor*. Este inicia suponiendo que los núcleos A y B tienen copias de sólo lectura de la misma página. Lo cual es legal. debido a que no hay más núcleos que tengan copias de la página ni clientes de ninguno de los núcleos que puedan modificar la página sin haber pedido primero un incremento en los permisos de acceso al paginador. En algún instante un tercer núcleo. C. requiere acceso de escritura a la página (paso 1). Antes de que el paginador permita la solicitud del núcleo C. debe negar las copias de la página de los núcleos A y B. El paginador envía un mensaje de anulación para liberar la página en cada núcleo que tenga una copia (paso 2). El paginador utiliza un puerto para recibir una respuesta complementaria a estos mensajes. Si el paginador no espera la respuesta a los mensajes de los dos núcleos, podría ser posible que un cliente del núcleo C modificara la página mientras un cliente del núcleo A o B estuviera examinando una copia (inválida) de la página. Después de que los núcleos A y B responden en el

paso 3, el paginador permite al núcleo C acceso de escritura a la página en el paso 4. Las propiedades del protocolo de coherencia se preservan y sólo un escritor tiene acceso a la página.

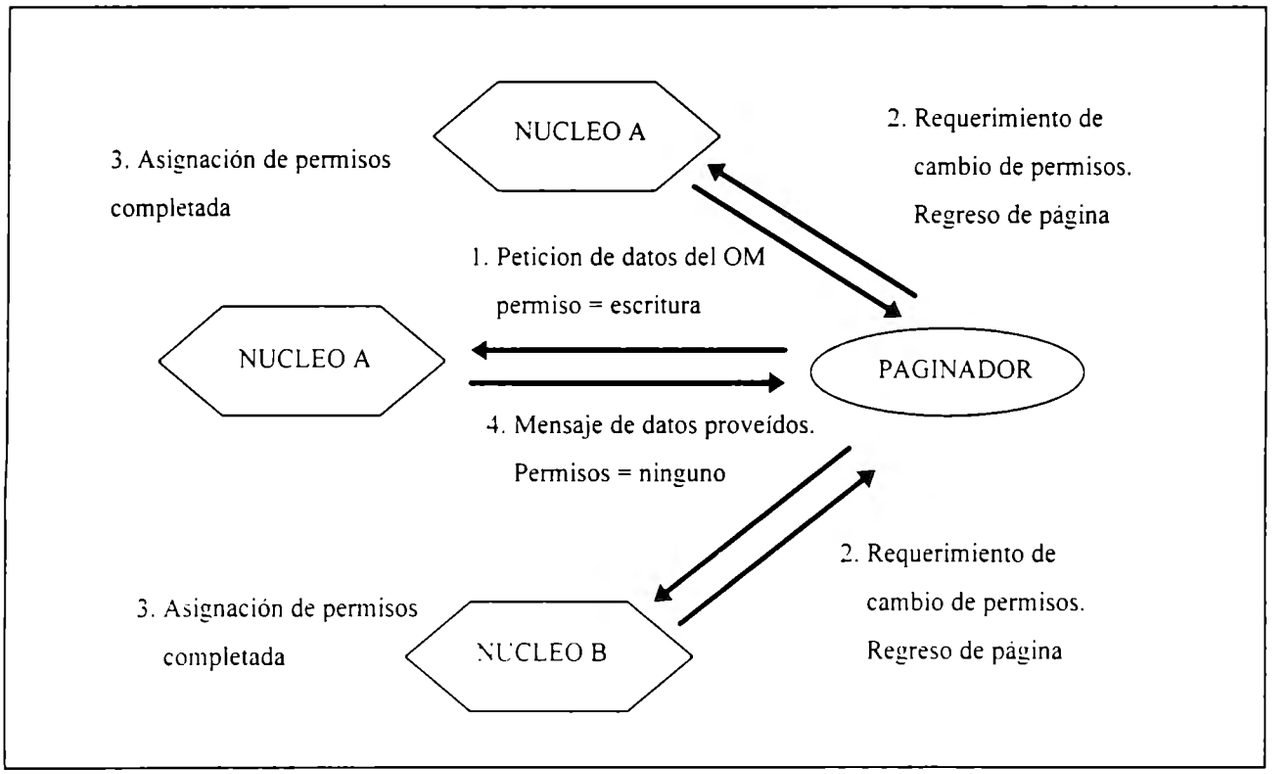


Fig. 2.9 Paginador con memoria compartida distribuida.

CAPITULO 3

LOTOS: Language Of Temporal Ordering Specification

3. LOTOS: Language Of Temporal Ordering Specification

3.1 Introducción

LOTOS es una técnica de especificación formal desarrollada para definir estándares formales independientes de la implementación de servicios y protocolos del modelo estandarizado por ISO. OSI. LOTOS es usado para modelar el orden en el que los eventos de un sistema se llevan a cabo.

LOTOS se divide en 2 partes específicas, la primera ofrece un modelo de conducta derivado de las álgebras de proceso, principalmente de CCS (Calculus of Communicating Systems) [22] así como de CSP (Communicating Sequential Processes) [23]. La segunda parte permite a los especificadores describir tipos de datos abstractos y valores. Esta parte está basada en el lenguaje de tipos de datos abstractos ACT ONE [21]. La siguiente figura muestra esta división de LOTOS (Figura 3.1).

Cada una de las partes que constituyen el lenguaje LOTOS tienen una función específica. El primer componente describe la parte de control, mientras que el segundo describe la parte de datos.

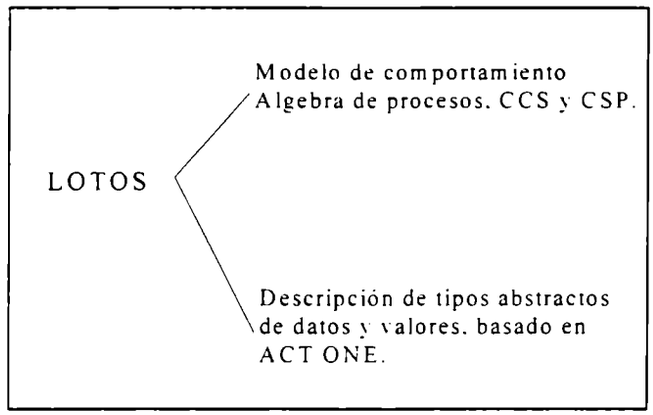


Fig. 3.1 Partes de LOTOS.

3.2 El lenguaje LOTOS

El componente de control describe, mediante una combinación de operadores de LOTOS, el comportamiento de los procesos de una especificación. Un proceso es una entidad capaz de desempeñar acciones internas no observables, e interactuar con su ambiente a través de puntos de interacción común, denominados compuertas. La forma atómica de interacción es un evento. Un evento es una unidad de sincronización que puede existir entre dos o más procesos que participan en ese evento. Una especificación en LOTOS tiene la siguiente forma:

specification *nombre-especificación [g1,...,gn] (v1,...,vm) : funcionalidad*
behaviour
    <expresión de comportamiento>
where
    <definición de procesos>
endspec

La definición de proceso tiene la forma:

process *nombre-proceso [g1,...,gn] (v1,...,vm) : funcionalidad*
behaviour
    <expresión de comportamiento>
where
    <definición de procesos>
endspec

donde

nombre-especificación	es el nombre de la especificación,
nombre-proceso	es el nombre del proceso,
g1,...,gn	es la lista de compuertas,
v1,...,vm	es la lista de parámetros formales.

El componente de datos de LOTOS utiliza el lenguaje de tipos de datos abstractos ACT ONE. describe la estructura de datos manipulada por la componente de control. Un tipo de dato abstracto es una representación de los datos de una especificación independiente de la implementación; ésta es una combinación de *sorts*. operaciones y ecuaciones. Los *sorts* son nombres de conjuntos de elementos, las operaciones son funciones en los elementos. y las ecuaciones describen axiomas del *sort*.

3.3 Expresiones de comportamiento en LOTOS y su semántica

En esta sección se explican los operadores principales utilizados para construir las expresiones de comportamiento.

3.3.1 Operadores

Ninguna acción (Inaction) *B = stop*

La conducta *B* no puede ofrecer ninguna acción al medio ambiente ni puede ejecutar eventos internos. ***stop*** es equivalente a un candado mortal.

Prefijo de acción (Action Prefix) *B = aE1...En[P];B2*

El comportamiento *B* ofrece la participación en el evento *aE1...En[P];B2,* si la interacción ocurre el comportamiento resultante es *B2*. *a* es el nombre de una compuerta, y *Ei* puede ser cualquier acción de transmisión o recepción de un valor *!V*, o *?x:s*, respectivamente. *P* es una lista de predicados opcionales, la cual debe ser satisfecha para que *B* participe en el evento. Los *Ei* se

denominan *ofertantes* de valores ("*value offers*"), y [P] es conocido como el conjunto de los *predicados de selección*. Estos dos últimos pueden estar ausentes. Haciendo la substitución *e=aE1...En[P]* la expresión *B = aE1...En[P];B2* es equivalente a *B = e;B2* , en este caso, la terminación con éxito del evento *e* genera el comportamiento *B2*, nótese que *e* es equivalente al ofrecimiento y/o recepción de un valor a través de una compuerta, es decir, una acción atómica.

```
process sale-entra [sale, entra] :=
    entra?x:Nat [x > 0 and x < 6 ];
    sale!x ;
endproc
```

Ejemplo 1 Prefijo de acción.

Este proceso acepta un número natural entre 0 y 6 en la compuerta *entra*, ofrece el mismo valor en la compuerta *sale* y después se detiene.

Prefijo de acción interno (Internal Action Prefix) *B = i ; B1*

La conducta *B* ejecuta una acción interna no observable, denotada por ***i***, y se transforma en *B1*.

Elección (Choice) *B = B1 [] B2*

El comportamiento *B* puede ofrecer tanto *B1* como *B2*. La elección puede depender de la acción ofrecida por el medio ambiente.

```
process entra1_entra2 [entra1, entra2, sale] :=
    entra1?x:Nat ;
    sale!x ;
    stop
 []
    entra2?x:Nat ;
    sale!x ;
    stop
endproc
```

Ejemplo 2 Elección.

Este proceso recibe cualquier número natural en la compuerta *entra1* o en la compuerta *entra2* y ofrece el mismo valor en la compuerta *sale*. Después se detiene.

Comportamiento con guardia (guarded behaviour) *B = [P] → B1*

El comportamiento *B* se conducirá como *B1* si *P* es verdadera. de otra forma, se comportará como ***stop***.

Llamada de proceso *B = P[g1, ... , gn](f1, ... , fm)*

El comportamiento *B* está dado por la definición del proceso *P*. sustituyendo sus parámetros formales con los parámetros actuales *f1, ... , fm* y sus compuertas por *g1, ... gn*.

B = entra1_entra2[a, b, c]

Ejemplo 3 llamada de procesos.

donde el proceso *entra1_entra2* está definido en el ejemplo 2. La interpretación de la conducta *B* es como sigue:

a?x:Nat ; *c!x* ; ***stop***

[]

b?x:Nat ; *c!x* ; ***stop***

Una llamada recursiva a un proceso genera una conducta infinita.

Definición local *B =* ***let*** *x1 = t1, ... , xn = tn* ***in*** *B1*

B se comporta como *B1* con todos los identificadores de valor *x1, ... xn* substituidos por las expresiones de valor *t1, ... ,tn* respectivamente.

B = ***let*** *x:Nat = 0 , y:Nat = Succ(0)* ***in***   *a!x* ; *b!y* ; ***stop***

Ejemplo 4 Definición local.

La interpretación de este ejemplo es:

a!0 ; *b!Succ(0)* ; ***stop***

Substitución de compuertas *B = **choice** g **in** [g1, ... ,gn] [] B1*

B se comporta como *B1[g/g1] [] ... [] Bn[g/gn]*, donde *[g/gi]* denota todas las ocurrencias de la compuerta *g* substituidas por la compuerta *gi*.

```
B = choice a in [entra1, entra2] []
      a?x:Nat  ;  sale!x  ;  stop
```

Ejemplo 5 Substitución en compuertas.

La interpretación del ejemplo es como sigue.

```
    entra1?x:Nat  ;  sale!x  ;  stop
[]
    entra2?x:Nat  ;  sale!x  ;  stop
```

Substitución de valores *B = **choice** x:s [] B1*

B se comporta como *B1[x/t1] [ ] ... [ ] B1[x/tn]*, donde *t1, ... ,tn* son todos los posibles valores de la expresión de tipo *s*.

Composición paralela *B = B1| [g1,...,gn] | B2*

Comúnmente al llevar a cabo una especificación es necesario dar una estructura a las especificaciones o representar algunas estructuras de diseño. Estas estructuras, en el caso de sistemas distribuidos, son generalmente instancias paralelas de funcionalidad. En LOTOS, las expresiones de comportamiento pueden ser combinadas utilizando el constructor paralelo.

Considérese la expresión de comportamiento general *B = B1| [g1,...,gn] | B2*, en la cual *B1* y *B2* son expresiones de comportamiento y *g1...gn* es una lista identificadora de compuertas. En esta expresión de comportamiento, los eventos en las compuertas que pertenecen a la lista pueden ocurrir solamente con la participación de ambas expresiones *B1* y *B2*. Los eventos que no pertenecen a la lista identificadora de compuertas pueden ocurrir con la participación de *B1* o *B2* de manera aislada.

Casos especiales de composición paralela

Existen dos casos especiales del operador paralelo: intercalado (*pure interleaving*) y sincronización total (*full synchronisation*).

En el caso de intercalado, las operaciones de dos expresiones de comportamiento están completamente intercaladas, es decir, siempre ocurren con la participación de sólo una de las expresiones de conducta. Esto corresponde al caso en el cual la lista identificadora de compuertas está vacía; la representación de este tipo de interacción se representa con el operador |||. En el ejemplo 6 se muestra una expresión de comportamiento de este tipo.

B = (a; b; ***stop****)* ||| *(c;* ***stop****)*

Ejemplo 6 Composición paralela

La equivalencia de este ejemplo, en términos de los operadores prefijo de acción y de elección, está dada por la ocurrencia de alguna de las siguientes conductas:

a; b; c; ***stop***

[]

a; c; b; ***stop***

[]

c; a; b; ***stop***

En el caso de sincronización total, las interacciones en cualquier compuerta de las dos expresiones de conducta están sincronizadas completamente, es decir, ocurren con la participación de las dos expresiones de comportamiento. Sintácticamente, es el caso en el cual la lista identificadora de compuertas contiene todos los identificadores de compuerta que aparecen en las dos expresiones de comportamiento; el operador usado para expresar la sincronización total es ||. En el ejemplo 7 se presenta una expresión de comportamiento con sincronización total.

B = (a; b; c; stop) || (a; b; d; c; stop)

Ejemplo 7 Composición paralela

La equivalencia de este ejemplo en términos de los operadores de prefijo de acción y de elección es :

a; b; ***stop***

Constructor paralelo generalizado

El constructor paralelo generalizado es una notación que sirve para realizar especificaciones más compactas. Es posible interpretar el operador paralelo generalizado en términos de composiciones paralelas de expresiones de comportamiento. Esto se presenta en el ejemplo 8.

par *g1* ***in*** *[a1, a2,a3]* ***parop*** *B [g1,h1]*

Ejemplo 8 Constructor paralelo generalizado

En el ejemplo anterior, *B [g1,h1]* es una expresión de comportamiento y *parop* es un operador paralelo como || o |||. La expresión de comportamiento correspondiente en términos del operador paralelo aparece como:

B [a1,h1] ***parop*** *B [a2,h1]* ***parop*** *B [a3,h1]*

Expresión de ocultamiento *B =* ***hide*** *g1, ..., gn* ***in*** *B1*

B se comporta como *B1*, sin embargo, cualquier acción que ocurra en cualquiera de las compuertas *g1, ..., gn* es transformada en una acción interna. Esto significa que el medio ambiente de B1 no puede participar en estas acciones.

B = ***hide*** *a,b* ***in***
(a; b; d; ***stop****)* ***|[b,d]|*** *(c; b; f; d;* ***stop****)*

Ejemplo 9 Ocultamiento de expresiones.

La interpretación de este ejemplo es :

i; c; i; f; d; ***stop***
[]
c; i; i; f; d; ***stop***

Terminación satisfactoria *B = **exit**(V1, ..., Vn)*

Denota la terminación satisfactoria del comportamiento *B*, donde *V1*, ... *Vn* son los casos de los parámetros de ***exit***. El comportamiento resultante de *B* después de ***exit*** es ***stop***.

B = g?x:Nat; ***exit*** *(x)*

Ejemplo 10 Terminación satisfactoria.

La expresión de comportamiento *B* acepta un valor para *x* en la compuerta *g*, después ofrece el valor de *x* en una compuerta denominada δ (al ocurrir un evento interno correspondiente a un ***exit*** satisfactorio).

Expresión de habilitación *B = B1 >> B2*

B se comporta como *B1* hasta que termina satisfactoriamente, entonces *B* se comporta como *B2*. Si *B1* no termina satisfactoriamente, entonces *B2* no se ejecuta.

B = (a; b; c; ***stop[]*** *d; e;* ***exit****) >> (f;* ***stop****)*

Ejemplo 11 Expresión de habilitación

La interpretación de este ejemplo es :

a; b; c; ***stop***
[]
d; e; i; f; ***stop***

Expresión de habilitación con paso de valor *B = B1 >>* ***accept*** *x1:s1,....xn:sn* ***in*** *B2*

B se comporta como *B1* hasta que termina satisfactoriamente, después *B* se comporta como *B2*, durante *B2* se reciben los valores enviados por *B1*, *x1....,xn* de tipo *s1,.... sn*, respectivamente.

B = a?cont:Bool; ***exit****(cont) >>* ***accept*** *respuesta:Bool* ***in*** *(b!respuesta;* ***stop****)*

Ejemplo 12 Expresión de habilitación con paso de valor

La interpretación de este ejemplo es :

a?cont:Bool; ***i;*** *b!cont;* ***stop***

Expresión de deshabilitación *B = B1 [> B2*

B se comporta como *B1*, sin embargo durante la ejecución de *B1*, *B2* puede deshabilitar a *B1* en cualquier instante y empezar su ejecución. Si *B1* termina satisfactoriamente, *B2* no puede ocurrir.

B = (a; b; c; ***stop****) [> (e; f;* ***stop****)*

Ejemplo 13 Expresión de deshabilitación

La interpretación de este ejemplo es :

e; f; ***stop***

[]

a; e; f; ***stop***

[]

a; b; e; f; ***stop***

[]

a; b; c; e; f; ***stop***

3.4 Representación de procesos como árboles

El comportamiento de un proceso puede ser representado como una estructura en forma de árbol. La raíz del árbol representa el estado inicial del proceso. Las ramas del árbol son etiquetadas por los nombres de los eventos. De esta forma, las etiquetas en las ramas de cada nodo representan los siguientes pasos posibles del proceso. La estructura de árbol representa el comportamiento del proceso como una secuencia de posibles operadores de opción, ordenados en el tiempo de acuerdo a la profundidad de los nodos en el árbol. En el caso finito, tal árbol coincide fácilmente

con la estructura de una expresión de forma normal[1]. Es importante enfatizar que esta estructura de árbol no puede ser observada como una máquina de estados disfrazada bajo la interpretación usual de un aceptador de cadenas. En el caso de LOTOS, un árbol no es sólo la caracterización de un número de rutas o trazas. Por ejemplo, los dos árboles en la figura 3.2 modelan conductas diferentes a pesar de que ambos tienen el mismo conjunto de ramas, es decir, {ab,ac}.

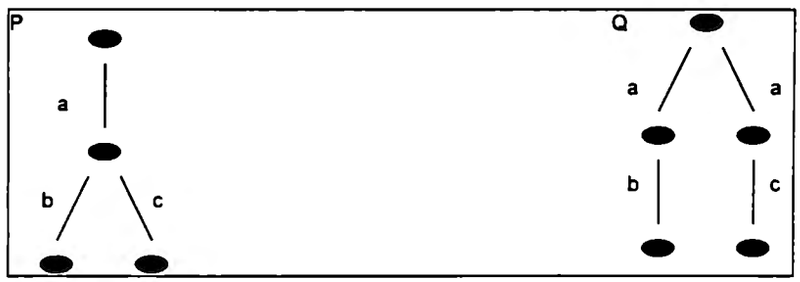


Fig. 3.2 Representación de procesos como árboles

La esencia de la interpretación como una estructura de árbol reside en el hecho de que la elección que es representada por nodos de ramas de un árbol se resuelve en la interacción entre los procesos y su ambiente. La diferencia entre ***P*** y ***Q***, anteriormente ilustrada, es como sigue: Se asume que ***P*** y ***Q*** están puestos en un ambiente que desempeña el experimento "*a*" seguido de "*b*". Con el proceso ***P***, esto podría ser posible bajo cualquier circunstancia. Para la interacción con ***P*** en el nodo decisivo, el resultado será sólo un experimento que sea posible para ambos, ***P*** y el ambiente, esto es "*b*". En el caso de ***Q***, la elección se realiza en el estado inicial. La acción de que el ambiente quiera desempeñar "*a*" no resuelve la elección. Una elección no determinista se hace entre "*a*" con un posible futuro de "*b*" y "*a*" con un posible futuro de "*c*" bajo la influencia del ambiente, por lo tanto, es posible que se escoja la ruta "*c*" no deseada según el experimento propuesto. El hecho de que exista un posible comportamiento de ***Q*** que lo distingue de ***P*** forza a aceptar la diferencia entre ellos.

A continuación se proporcionan una serie de ejemplos más completos del uso de los operadores descritos, presentando su especificación en LOTOS básico y su representación equivalente en estructuras con forma de árbol.

Ejemplo 14 Aquí se describe el comportamiento de un sistema de control de acceso que requiere la inserción de dos claves antes de permitir el acceso. Las claves pueden ser extraídas sólo después de que el acceso haya ocurrido.

[1] La forma normal, también conocida como forma canónica, tiene las siguientes propiedades: no puede ser reescrita y dos expresiones que son equivalentes llegan a la misma forma canónica después de aplicar tantas reglas de escritura como sea posible [4].

```
In1;
In2;
Access;
(Out1; Out2; stop [] Out2; Out1; stop)
[]
In2;
In1;
Access;
(Out1; Out2; stop  Out2; Out1; stop)
```

Ejemplo 14 Sistema de inserción de dos llaves

El primer paso en el ejemplo 14 es la definición de la interfaz observable del sistema. Existen cinco eventos abstractos para la creación de un modelo del sistema de dos claves: *In1, In2, Access, Out1, Out2*. Los dos primeros representan la inserción de las claves, el tercero representa el acceso y los últimos dos representan la extracción de las claves. La especificación de este ejemplo hace uso de los siguientes operadores: prefijo de acción (;) , elección (*[]*), y de no acción (***stop***).

La figura 3.3 muestra el árbol de comportamiento descrito por la especificación previa.

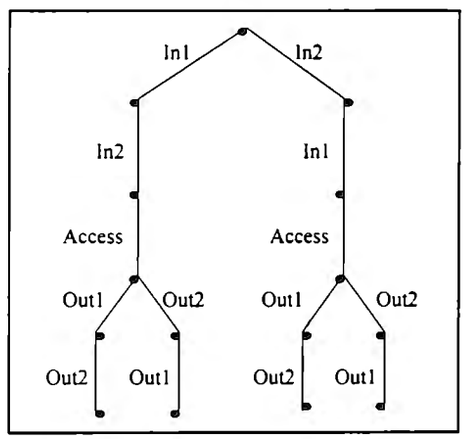


Fig. 3.3 Árbol de comportamiento del sistema de dos claves

La interpretación de este árbol facilita la comprensión del comportamiento del sistema. El sistema se encuentra en un estado inicial en espera de la inserción de las claves; existen dos posibilidades que son representadas con dos ramas complementarias, es decir, puede existir un caso en el que se inserte la clave 1, después la clave 2, se permita el acceso y ocurra alguna de las dos siguientes posibilidades: que se extraiga la clave 1 y después la clave 2, o que se extraiga la clave 2 y después la clave 1. En el otro caso se proporciona la clave 2 seguida de la clave 1, en seguida se proporciona el acceso y se presentan las dos opciones de extracción de claves como en el caso anterior.

Ejemplo 15 Representación del sistema de dos claves utilizando recursión. Se utiliza la propiedad denominada recursión, que permite la creación de comportamientos infinitos. Esta especificación describe el comportamiento del sistema de dos claves ejecutándose de manera infinita. Esto se hace llamando al proceso *Acc* dentro del proceso *TwoKey* y llamando este último dentro del proceso *Acc*. Además, se maneja la reetiquetación de compuertas, renombrando los eventos definidos en el ejemplo anterior como compuertas del proceso principal. El árbol que representa este comportamiento se aprecia en la figura 3.4.

```
TwoKey [In1, In2, Access, Out1, Out2]

where
process TwoKey [I1, I2, Ac, O1, O2] : noexit :=
  K1; K2; Acc [I1, I2, Ac, O1, O2]
[]
  K2; K1; Acc [I1, I2, Ac, O1, O2]
endproc (*TwoKey*)

process Acc[I1, I2, Ac, O1, O2] : noexit :=
  Ac;
  (
  O1; O2; TwoKey [I1, I2, Ac, O1, O2]
[]
  O2; O1; TwoKey [I1, I2, Ac, O1, O2]
  )
endproc (*Acc*)
```

Ejemplo 15 Sistema de inserción de dos llaves con recursión

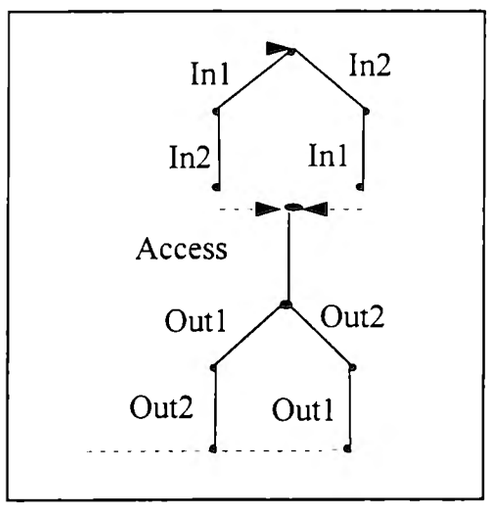


Fig 3.4 Árbol de comportamiento del sistema de dos claves utilizando recursividad

Ejemplo 16 En la primer fase se introducen las claves, en ella se modela el comportamiento intercalado que trata con la inserción de las claves en cualquier orden.

```
In1; exit
|||
In2; exit
```

Ejemplo 16 Sistema de dos llaves con el operador de intercalación

En la parte alta de la figura 3.5 se muestran los árboles de los dos comportamientos que se intercalan. El árbol resultante de la composición intercalada se muestra en la parte baja de la figura. Ambos comportamientos se sincronizan en el evento δ (al ejecutarse ***exit***) a pesar de que están compuestos por intercalación.

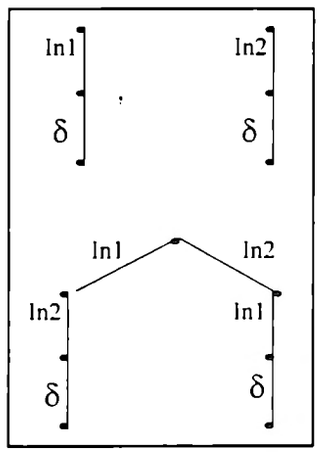


Fig. 3.5 Árbol de comportamiento del sistema de dos claves con el operador de interpaginación

Ejemplo 17 Sincronización. Una especificación alternativa para el sistema de dos claves utiliza el operador paralelo que incluye una lista de identificadores de compuertas. El sistema puede ser especificado como un número dado de restricciones independientes que cuentan con secuencias válidas de comportamiento. Las restricciones independientes se construyen con el operador paralelo para definir el sistema.

```
(
   In1; Access; stop
  |[Access]|
   In2; Access; stop
)
|[Access]|
(
   Access; Out1; stop
  |[Access]|
   Access; Out2; stop
)
```

Ejemplo 17 Sistema de dos claves utilizando sincronización en una compuerta

La figura 3.6 muestra el comportamiento del sistema.

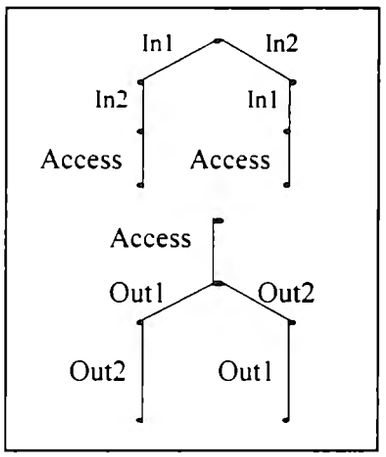


Fig. 3.6 Árbol de comportamiento del sistema de dos claves con el operador de sincronización

Ejemplo 18 Ocultamiento. El ocultamiento provee un mecanismo de abstracción poderosa. Puede ser usada con el sistema de dos claves para obtener un nuevo sistema de una clave a partir de alguna de las especificaciones previas. Sólo es necesario esconder las compuertas *In2* y *Out2*.

```
hide In2, Out2 in
        In1;
        In2;
        Access;
        (Out1; Out2; stop   Out2; Out1; stop)
    []
        In2;
        In1;
        Access;
        (Out1; Out2; stop   Out2; Out1; stop)
```

Ejemplo 18 Sistema de una clave utilizando ocultamiento

En la figura 3.7 se muestra el ocultamiento de las compuertas *In2* y *Out2*. Sin embargo hay eventos internos compuestos con el operador de elección, el comportamiento del sistema es determinista.

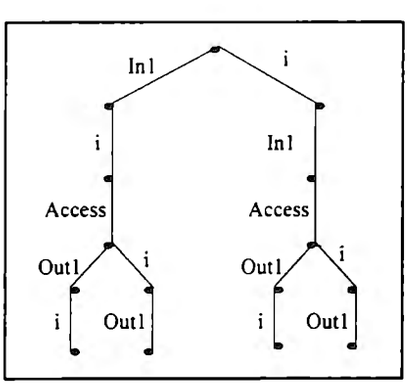


Fig. 3.7 Árbol de comportamiento del sistema de una claves utilizando ocultación

3.5 Tipos de datos

El lenguaje basado en ACT ONE, modela los datos como tipos de datos abstractos (TDA). La teoría fundamental consiste en modelos por ecuaciones. En seguida se presenta una revisión breve de tipos de datos.

3.5.1 Conceptos básicos

No existen tipos de datos predefinidos. Una especificación en LOTOS inicia sin la definición interna de tipos de datos. Por ejemplo, cada valor de verdad (expresión booleana) requerida explícitamente por algunos constructores del lenguaje como guardias y predicados, debe ser definido en cada especificación de LOTOS. A la parte de la especificación en donde se definen los tipos de datos se le conoce como definición de tipos.

Existe una biblioteca estándar en LOTOS donde se encuentran definidos los tipos de datos comúnmente requeridos, esto ayuda en la especificación ahorrando tiempo y espacio. El uso de bibliotecas en LOTOS, por ejemplo es:

```
library
        Boolean, Set, NaturalNumber
endlib
```

Los tipos de LOTOS no se deben confundir con los tipos usados en los lenguajes de programación. En un lenguaje de programación, un tipo está determinado como un conjunto de valores con operaciones predefinidas. En LOTOS, un tipo está determinado como un conjunto de valores y operaciones que deben ser definidas incluyendo la especificación de las propiedades, de las operaciones y de los valores. Concretamente, en LOTOS se deben definir en forma precisa el *sort*, las operaciones (*operations*) y las ecuaciones (*equations*) para cada tipo de dato.

Los *sorts* son conjuntos distintos de valores de datos. Una variable debe tener sólo un *sort* específico. Como se puede observar, el concepto de *sort* es equivalente a la acepción del término tipo en muchos lenguajes de programación.

Las operaciones en LOTOS corresponden a las funciones que se definen para el manejo de objetos. Por medio de estas operaciones se pueden combinar valores de un mismo tipo o de tipos diferentes en valores agregados, o establecer relaciones entre ellos.

Finalmente, las ecuaciones utilizan variables universalmente cuantificadas. Por ejemplo, la expresión:

$$\forall x: Nat.\ x + 0 = 0$$

denota que el resultado de la suma de la variable x y el valor entero 0 es 0, para toda x en el conjunto de los número naturales. Las ecuaciones establecen las propiedades que deben ser satisfechas por los objetos del tipo. El razonamiento de las ecuaciones debe ser aplicado a las expresiones de valor para encontrar si son iguales o no.

En la práctica, se escriben ecuaciones que pueden ser tratadas como reglas de reescritura, las cuales manejan transformaciones de expresiones. La ecuación:

LI = LD (Lado Izquierdo igual a Lado Derecho)

puede ser tratada como una regla de reescritura:

LI => LD

La interpretación de la regla de reescritura es como sigue:

Una expresión que cumpla el patrón LI, puede ser reescrita de acuerdo al esquema LD.

El objetivo de la reescritura es reducir una expresión a su forma canónica. Esta es única y en ocasiones es denominada como forma normal. Las propiedades de la forma canónica son las siguientes:

- No puede ser reescrita otra vez.
- Dos expresiones que puedan ser probadas deben satisfacer la misma forma canónica después de aplicarles tantas reescrituras como sea posible.

3.5.2 Tipos de datos básicos

A continuación se presentan algunos tipos básicos mostrando la sintaxis utilizada para su definición.

En el siguiente ejemplo se define el tipo de dato *Boolean.*

```
type Boolean is
  sort Bool
  opns
        true, false:     -> Bool
        not : Bool       -> Bool
  eqns
        ofsort Bool
                not (true)  = false;
                not (false) = true;
endtype ( *Boolean*)
```

Ejemplo 19 Tipo de dato booleano.

En esta especificación de tipo ha sido definido el tipo *Boolean* con el tipo *Bool.* Para éste, se utilizan las constantes usuales *true* y *false* y la operación de negación *not* que complementa un valor booleano. Una constante en LOTOS es sencillamente una operación sin argumentos, como es el caso de *true* y *false.* Debido a que una operación siempre devuelve el mismo resultado cuando se usan los mismos argumentos, una operación sin argumentos siempre devuelve un valor constante, para este ejemplo *true* o *false.*

En el siguiente ejemplo se define el tipo de dato para números naturales.

```
type NaturalNumber is
  sort Nat
  opns
        0 :              -> Nat
        Succ :Nat        -> Nat
endtype ( *NaturalNumber*)
```

Ejemplo 20 Tipo de dato numero natural

La especificación del tipo *NaturalNumber* define todos los números positivos incluyendo el cero. Esta especificación está basada en el enfoque matemático usual, más que en la notación arábiga convencional para los números. La idea es tener una sola constante *0*, y una operación de

sucesores *Succ* que determina el siguiente valor. Los términos para este tipo, tomando en cuenta lo anterior son:

0, *Succ* (0), *Succ* (*Succ* (0)),*Succ* (*Succ* (*Succ* (0))),
Succ (*Succ* (*Succ* (*Succ* (0)))), ...

Una vez que se definen los tipos, cualquier variable del tipo definido puede ser usada en la especificación de LOTOS. En el caso de los ejemplos 19 y 20, respectivamente, cualquier expresión booleana que involucre valores de verdad *true* o *false*, o números naturales que involucren el uso del *0* o sus sucesores, puede ser escrita y utilizada dentro de guardias o predicados de la especificación.

3.5.2.1 Extensión

Un tipo de dato definido puede ser extendido con tipos, operaciones y ecuaciones nuevas. La especificación de nuevos tipos suministra más conjuntos de valores de datos. Añadiendo nuevas operaciones se amplia el rango de expresiones permitidas. La introducción de nuevas ecuaciones da por resultado nuevas propiedades de operaciones y nuevas formas de reescritura. En la siguiente especificación de tipos se presenta una extensión de los números naturales.

```
type NaturalExtend is NaturalNumber
  opns
        _ + _  :Nat, Nat        -> Nat
  eqns
       forall x, y : Nat
         ofsort Nat
               x + 0 = x;
               x + Succ(y) = Succ (x + y);
endtype ( * NaturalExtended *)
```

Ejemplo 21 Extensión de la definición de los números naturales

Esta especificación enriquece el tipo *NaturalNumber* con una operación "+" para la suma. La operación toma dos números naturales y devuelve el número natural correspondiente a la suma. Los guiones bajos antes y después de la operación indican que no es una operación fija.

Las ecuaciones permiten que los términos nuevos que pueden ser escritos con "+" sean evaluados. Por ejemplo, para la evaluación de 3 + 2 = 5 sería como sigue:

Succ (Succ (Succ (0))) + Succ (Succ (0))
⇒ Succ (Succ (Succ (Succ (0))) + Succ (0))
⇒ Succ (Succ (Succ (Succ (Succ (0))) + 0))
⇒ Succ (Succ (Succ (Succ (Succ (0)))))

3.5.2.2 Combinación

Los tipos pueden ser combinados para construir tipos más complejos. A continuación se presenta un ejemplo de combinación de tipos:

```
type NaturalStack is  NaturalNumber, Boolean
  sort Stack
  opns
        empty:                          -> Stack
        push:         Nat, Stack        -> Stack
        top:          Stack             -> Nat
        pop:          Stack             -> Stack
        IsEmpty:      Stack             -> Bool
  eqns
        forall n : Nat, s : Stack
          ofsort Nat
            top ( push (n, s) ) = n;
          ofsort Stack
            pop ( push (n, s) ) = s;
          ofsort Bool
            IsEmpty (empty) = true;
            IsEmpty (push ( n, s ) ) = false;
endtype ( * NaturalStack *)
```

Ejemplo 22 Definición del tipo NaturalStack utilizando combinación

En esta especificación se define un nuevo ***sort***, *Stack*. La base es la pila vacía (*empty:-> Stack*). Esta operación está definida para dar como resultado una pila sin usar ningún argumento. Las operaciones *push, pop, top* e *IsEmpty* trabajan sobre valores exsitentes de tipo *Stack*.

Las ecuaciones establecen las propiedades esperadas: la cima de la pila es el último elemento introducido a la pila; realizar el *pop* a la pila devuelve la misma estructura sin el último elemento introducido en ella; encontrar si la pila está vacía o no, requiere verificar si cualquier elemento ha sido introducido en ella.

3.5.2.3 Ecuaciones condicionales

La aplicación de una ecuación puede ser hecha a partir de una condición booleana, es decir, una premisa. Si la premisa pertenece a un conjunto dado de valores, la ecuación es aplicable para estos valores, de lo contrario, la ecuación no es aplicable para ellos. Como ejemplo se presenta la especificación con la operación que regresa el valor máximo entre dos números naturales.

```
type NaturalMaximum is Natural Number
 opns max : Nat, Nat                    -> Nat
                    eqns
                 forall x, y : Nat
                    ofsort Nat
                     x ge y =>
                         max (x, y) = x;
                     y ge x =>
                         max (x, y) = y;
      endtype ( * NaturalMaximum *)
```

Ejemplo 23 Definición del tipo Natural Maximum utilizando ecuaciones condicionales

3.5.2.4 Renombramiento

El renombramiento permite la especificación de un nuevo tipo de dato completamente independiente, cambiando los nombres de un tipo existente. Se usa antes de extender un tipo y principalmente cuando se quiere un nuevo tipo de dato con características adicionales a uno ya existente. El renombramiento se muestra en el ejemplo 24.

```
type Bit is  Boolean
  sort Bit
  opns
        0,1:                          -> Bit
        _ + _:          Bit, Bit      -> Bit
        _ eq _:         Bit, Bit      -> Bool
  eqns
        forall b: Bit
          ofsort Bit
            0 + 0 = 0; 0 + 1 = 1;
            1 + 0 = 1; 1 + 1 = 0;
          ofsort Bool
            b eq b = true;
            0 eq 1 = false; 1 eq 0 = false;
endtype ( * Bit *)


type Parity is Bit renamedby
  sortnames
  Parity for Bit
  opnnames
        even for 0
        odd  for 1
endtype ( * Parity *)
```

Ejemplo 24 Nombramiento de tipos de datos.

Los nombres de las operaciones y de los tipos pueden ser cambiados. Si no se cambian, se utiliza el mismo nombre con el tipo nuevo.

En el ejemplo 24 se tiene el tipo nuevo *Parity* que tiene los mismos valores *Boolean* que el tipo *Bit*, donde éste define las operaciones *0*, *1*, '+' y *eq*. Las operaciones *0* y *1* son renombradas explícitamente como *even* y *odd*, respectivamente. Las operaciones '+' y *eq* no fueron renombradas, pero como sus argumentos si lo fueron (*Bit* reemplazado por *Parity*) se convierten en operaciones nuevas de *Parity* conservando el mismo nombre (a este mecanismo se le conoce como *Operation Overloading*).

3.5.2.5 Parametrización

Los tipos de datos pueden ser especificados parcialmente, dejando espacios para la adición de información posterior. Una situación similar se presenta con las funciones y procedimientos en un lenguaje de programación donde un algoritmo puede hacer referencia a argumentos formales que son suministrados posteriormente. Cuando los argumentos actuales son suministrados en una llamada, el algoritmo se ejecuta con estos valores definitivos. Esta técnica representa la eliminación de programación repetida, reduciendo así los errores y simplificando el mantenimiento de los módulos.

Los espacios en los tipos de datos se pueden dejar para los tipos y para las operaciones, éstas actúan como una interfaz al tipo genérico. Los requerimientos pueden ser puestos en esta interfaz por la especificación de ecuaciones que permanecen sólo si los parámetros actuales llenan los espacios.

El ejemplo 25 indica la especificación de una pila parametrizada por algún elemento de cierto tipo.

3.5.2.6 Actualización

Los tipos parametrizados pueden ser actualizados en instancias con parámetros actuales para llenar los espacios. La actualización invoca un renombramiento de los componentes del tipo de dato parametrizado, de manera que los *sorts* y las operaciones del tipo parametrizado son renombradas a los *sorts* y operaciones del tipo actual.

Los *sorts* y las operaciones que no son formales en el tipo parametrizado pueden ser renombrados de la forma usual. En el ejemplo 26 se actualiza el tipo de dato *GenericStack* para definir una pila que opere con números naturales, *NatStack*; y otra que opere con valores booleanos, *BoolStack*.

```
type GenericStack is  Boolean
  formalsorts Element
  sort Stack
  opns
        empty:                          -> Stack
        push:          Element, Stack-> Stack
        top:           Stack            -> Element
        pop:           Stack            -> Stack
        IsEmpty:       Stack            -> Bool
  eqns
        forall e : Element, s : Stack
          ofsort Element
            top ( push (e, s) ) = e;
          ofsort Stack
            pop ( push (e, s) ) = s;
          ofsort Bool
            IsEmpty (empty) = true;
            IsEmpty (push ( e, s ) ) = false;
endtype ( * ElementStack *)
```

Ejemplo 25 Parametrización de tipos de datos.

```
type NatStack is GenericStack actualizedby NaturalNumber using
  sortnames
    Nat for Element
    NatStack for Stack
endtype ( * NatStack *)

type BoolStack is GenericStack actualizedby Boolean using
  sortnames
    Bool for Element
    BoolStack for Stack
endtype ( * BoolStack *)
```

Ejemplo 26 Actualización de tipos de datos.

3.6 Full LOTOS

LOTOS básico permite la representación de sincronización, pero no permite la transferencia de datos, debido a que el concepto de datos no está definido en él. Partiendo del álgebra de procesos definida por LOTOS básico y el lenguaje de tipos de datos abstractos, se permite la sincronización y la transferencia de datos, necesarios en sistemas distribuidos. La unión de estas dos propiedades definen un nuevo tipo de LOTOS, nombrado full LOTOS.

3.6.1 Eventos estructurados

En full LOTOS un evento tiene un identificador de compuerta y una lista de parámetros de interacción nombrada "*experiment offers*" los cuales se clasifican en:

a) Ofertante de valor ("*Value offer*") y tiene la forma ***! v*** donde ***v*** es una expresión de valor.

b) Ofertante de variable ("*Variable offer*") y tiene la forma ***? x : s***, donde ***x*** es una variable de tipo ***s***.

Un evento puede llevarse a cabo en una compuerta sólo cuando el ofertante del experimento iguala el tipo, el valor y el orden. En la figura 3.8 se indican las posibilidades de interacción.

Posibilidades de Interacción.			
Expresión de conducta A	**Expresión de conducta B**	**Condición**	**Tipo de valor**
g ! E1	**g ! E2**	**valor(E1) =valor(E2)**	**valor equivalente**
g ! E1	**g ? x : t**	**sort(E1) = t**	**paso de valor**
g ? x : t	**g ? y : u**	**t = u**	**generación de valor**

Fig. 3.8 Cuadro de comparación de valores y variables ofertadas

En el ejemplo 27 se presenta la utilización de eventos estructurados con ofertas de valores.

g ! Succ (0) ? x : Bool ! false

satisface :

g ? x : Nat ! true ! false

pero no satisface:

g ! Succ (0) ? x : Bool

ni :

g ? x : Nat ! true ! true

Ejemplo 27 Eventos estructurados

Valor equivalente, representa la sincronización alcanzando los valores esperados. Un ejemplo de este tipo de interacción es la verificación del "password", en el cual ocurre una sincronización sólo si se proporciona el valor correcto para "password".

Paso de valor, representa operaciones convencionales de entrada/salida. Una entidad suministra un valor (***! E1*** en la figura 3.8) y la otra recibe un valor (***g ? x:t*** en la figura 3.8).

Generación de valor, representa la selección no determinista de un valor para la variable de interacción. Modela el concepto de negociación, debido a que ambos participantes de la interacción deben acordar en un valor para que la interacción tenga lugar. Después de la sincronización, las variables de ambas partes tienen el mismo valor ($x = y$ en la figura 3.8).

3.6.2 Conducta condicional en Full LOTOS

La conducta de un sistema depende de ciertas condiciones. Las condiciones pueden depender de eventos pasados y de los datos almacenados por el sistema. Estas condiciones se representan en LOTOS como operaciones del tipo "*Boolean*". La condición impuesta en un evento puede ser aplicada antes de que ocurra, o ser impuesta durante su ejecución. La primera se representa en LOTOS por guardias, y la segunda por predicados de selección. En el ejemplo 28 se calcula el número mayor entre dos números naturales, usando comportamiento condicional con guardias.

```
process CalcMayor [entrada, mayor] : exit :=
        entrada ? x: Nat ? y:Nat;
        (
         [x ge y] ->
                mayor !x; exit
        []
         [y ge x] ->
                mayor !y; exit
        )
endproc (* CalcMayor *)
```

Ejemplo 28 Comportamiento condicional

3.6.3 Procesos parametrizados

Los procesos pueden ser parametrizados con el conjunto de valores de los datos. Los parámetros de los procesos se declaran en la definición de un proceso como parámetros formales. En las llamadas de los procesos, dichos parámetros deben ser asignados a sus valores correspondientes, es decir, expresiones de valor que evalúan la equivalencia del tipo.

3.6.4 Funcionalidad

Cada expresión de comportamiento válida en LOTOS tiene asignada una característica estática nombrada funcionalidad. La funcionalidad de una expresión de comportamiento es una lista ordenada de tipos, que indica la lista de parámetros de salida asociada con esa expresión de comportamiento. Las reglas para la evaluación de la funcionalidad se muestran en la figura 3.9. En ella, ***noexit*** indica que la expresión de comportamiento no termina satisfactoriamente, mientras que <..> indica una lista de parámetros de salida.

La funcionalidad de un proceso debe alcanzar la funcionalidad dada por su expresión de comportamiento.

	Reglas de funcionalidad	
Operador	**Condiciones**	**Funcionalidad**
stop	-	*noexit*
exit	-	<>
exit(v1...vn)	*sort(v1)=s1,....,sort(vn)=sn*	*<s1,...sn>*
act;B	-	*func(B)*
B1[ ] B2	*func(B1)=func(B2)*	*func(B1)*
	func(B1)=noexit	*func(B2)*
	func(B2)=noexit	*func(B1)*
	otherwise	*inválido*
B1>>B2	-	*func(B2)*
B1[>B2	*same as B1 [ ] B2*	*same as B1 [ ] B2*
B1 parop B2	*func(B1) = func(B2)*	*func(B1)*
	func(B1)=noexit	*noexit*
	func(B2)=noexit	*noexit*
	otherwise	*inválido*

Fig. 3.9 Reglas de funcionalidad.

3.6.5 Definición de valor local

Una definición de valor local asocia valores con variables libres en expresiones de comportamiento. Este operador permite más consistencia y mayor legibilidad en las especificaciones, debido a que permite que las expresiones de valor puedan ser reemplazadas por un identificador simple.

En el siguiente ejemplo se considera una parte de una especificación de la entidad de un protocolo en el que su comportamiento depende del contenido de una unidad de datos del protocolo. La unidad de datos del protocolo se deriva de los parámetros de datos una primitiva de servicio. La especificación formal se puede expresar como :

```
g ? sp : PrimServ;
(
  [CampoDatos (PDU(Datos Usuario(sp))) eq <>] -> ...
[]
  [CampoDatos (PDU(DatosUsuario(sp))) ne <>] -> ...
)
```

Usando una definición de valor local la especificación equivalente podría ser:

```
g ? sp : PrimServ;
(
  let datos : Datos = CampoDatos (PDU(DatosUsuario(sp)))   in
    (
      [datos eq <>] -> ...
    []
      [datos ne <>] -> ...
    )
)
```

En este caso la oración con ***let*** asocia un valor con la variable *datos*. Esto hace más legible y compacta la especificación, particularmente si la variable *datos* es utilizada numerosas veces.

3.6.6 Alcance ("Scope")

Los procesos pueden tener definiciones locales (después de la palabra ***where***), en la que pueden ser definidos los tipos de datos u otros procesos. El operador ***scope*** determina reglas para accesar a definiciones de tipos y procesos por otros tipos y procesos, es parte del texto de LOTOS que controla el acceso de algún elemento: un tipo de dato, un *sort*, una operación, una variable, un valor, o un proceso. Las reglas impuestas por ***scope*** son similares a los lenguajes de programación estructurada.

3.7 Especificación en LOTOS

Una especificación en LOTOS es el proceso que representa el sistema completo que se está especificando. Sin embargo, existen algunas diferencias sintácticas entre una especificación y un proceso, como se muestra en la figura 3.10.

Cuadro de diferencias sintácticas entre "specification" y "process"

Elemento de sintaxis	especificación	proceso
inicio de la definición	specification	process
definición de tipo global	si	no
inicio de conducta	behaviour	:=
fin de la definición	endspec	endproc

Fig. 3.10 Cuadro de diferencias sintácticas entre especificación y proceso.

3.7.1 Definición de tipo global

La definición de tipo global es una definición que es accesible a todas las expresiones de conducta, las definiciones de tipos y los procesos en la especificación. Las definiciones de tipo global aparecen antes de la palabra reservada ***behaviour*** en una especificación y no existen en las definiciones de procesos. En el ejemplo 29 se realiza la definición de un tipo de dato global.

```
specification S [a, b, c, d]: noexit

  type SomeType is              (* Definición de un tipo global*)
    sorts SomeSort
    opns SomeOpns : SomeSort -> SomeSort
  endtype    (* SomeType *)

  behaviour
    P[a, b, c] |[b]| Q[b,

  where

  process P[a, b, c] ...
  process Q[b, d] ...
endspec ( * S *)
```

Ejemplo 29 Definición de tipo de dato global.

Es importante enfatizar que los operadores, funciones y palabras reservadas de LOTOS básico son aplicables a Full LOTOS. A continuación se presentan ejemplos completos de especificaciones con Full LOTOS.

```
specification TwoKeySys [KI, Access]  :noexit

  type   KeyOps is
    sorts KeyOps
    opns In1, In2, Out1,Out2 : -> KeyOps
  endtype          (* KeyOps *)

  behaviour
      KI ! In1; KI ! In2;
      Access;
      (
        KI ! Out1, KI ! Out2; stop
      []
        KI ! Out2, KI ! Out1;  stop
      )
      []
      KI ! In2; KI ! In1;
      Access;
      (
        KI ! Out2, KI ! Out1;  stop
      []
        KI ! Out1, KI ! Out2;  stop
      )

endspec      (* TwoKeySys *)
```

Ejemplo 30 Sistema de dos claves con Full LOTOS

En el ejemplo 30 se expresa el sistema de dos claves reescrito como una especificación en Full LOTOS.

3.7.2 Estructuras de eventos

En Full LOTOS la definición de las interfaces del sistema tiene que tomar en cuenta los valores de los datos intercambiados en las compuertas. La combinación de compuertas y sus valores de parámetros normalmente se denomina como estructura de eventos. La primera consideración a tomar cuando se describe un sistema con Full LOTOS es la definición de la estructura de eventos de las interfaces

3.7.3 Predicados de selección

Los predicados sobre valores de variables en una especificación sirven para seleccionar el orden apropiado de un conjunto de operaciones. En el ejemplo 31 los predicados sobre *Keyops* sirven para seleccionar el orden adecuado sobre las operaciones de bloqueo (*lock*). *IsKeyIn* verifica cuando una clave está siendo insertada, mientras que I*sKeyOut* verifica cuando una clave está siendo extraída. *IsOtherIn* prueba por la otra clave del par que se está insertando, mientras que I*sOtherOut* similarmente verifica por la extracción. Nótese que a pesar de no haber parámetros, el sistema tiene memoria: cada identificador de valor que sigue un *?* es como una variable que almacena un valor para un uso posterior.

3.7.4 Parámetros y guardias en Full LOTOS

Los parámetros de procesos son usados como variables de estado, mientras que los guardias establecen condiciones en variables de estado para transiciones que están por ocurrir. El ejemplo 32 es de una máquina de estados extendida.

```
library
  Boolean
endlib

type  KeyOps ...

type  MoreKeyOps  is  KeyOps, Boolean
  opns
    IsKeyIn, IsKeyOut:    KeyOps                -> Bool
    IsOtherIn, IsOtherOutKeyOps, KeyOps         -> Bool
  eqns
    forall  x, y : KeyOps
       ofsort  Bool
         IsKeyIn (In1)      = true; IsKeyIn (In2)      = true;
         IsKeyIn (Out1)     = false; IsKeyIn (Out2) = false;
         IsKeyOut (Out1) = true; IsKeyOut(Out2) = true;
         IsKeyOut (In1)     = false; IsKeyOut(In2)     = false;
         not (IsKeyIn (x) and IsKeyIn (y)) =>
           IsOtherIn (x,y) = false;

         IsOtherIn (In1, In1) = false;
         IsOtherIn (In2, In2) = false;
         IsOtherIn (In1, In2) = true;
         IsOtherIn (In2, In1) = true;
         not (IskeyOut (x) and IsKeyOut (y) ) =>
             IsOtherOut (x,y) = false;
         IsOtherOut (Out1, Out1) = false;
         IsOtherOut (Out2, Out2) = false;
         IsOtherOut (Out1, Out2) = true;
         IsOtherOut (Out2, Out1) = true;

endtype     (* MoreKeyOps *)

behaviour
    KI ? op1 : KeyOps [IsKeyIn (op1)];
    KI ? op2 : KeyOps [IsOtherIn (op1, op2)];
    Access;
    KI ? op1 : KeyOps [IsKeyOut (op1)];
    KI ? op2 : KeyOps [IsOtherOut (op1, op2)];
    stop
```

Ejemplo 31 Predicados de selección en Full LOTOS.

```
library...
type KeyOps ...
behaviour BeforeAccess [KI, Access] (false,false)
where
process BeforeAccess
  [KI, Access] (InKey1, InKey2 : Bool ): noexit :=
    (
      [not (InKey1)] ->
        KI ! In1 ;
        BeforeAccess [KI, Access] (true, InKey2)
    []
      [not (InKey2)] ->
        KI ! In2 ;
        BeforeAccess [KI, Access] (InKey1, true)
    []
      [InKey1 and InKey2] ->
        Access;
        AfterAccess [KI] (InKey1, InKey2)
    )
endproc (* BeforeAccess *)

process AfterAccess [KI] (InKey1, InKey2 : Bool ) : noexit :=
    (
      [InKey1] ->
        KI ! Out1 ;
        AfterAccess [KI] (false, InKey2)
    []
      [InKey2] ->
        KI ! Out2 ;
        AfterAccess [KI] (InKey1, false)
    )
endproc (* AfterAccess *)
```

Ejemplo 32 Parámetros y guardias en Full LOTOS.

CAPITULO 4

DESCRIPCIÓN INFORMAL DE UNA MCD ENFOCADA AL SISTEMA OPERATIVO MACH

4. Descripción informal de una Memoria Compartida Distribuida enfocada al sistema operativo Mach

4.1 Administrador de Memoria Externo, AME

El sistema operativo Mach cuenta con un manejador de memoria que permite la compartición de objetos de memoria de tareas que están relacionadas, es decir, de procesos que se encuentran realizando tareas comunes dentro de un núcleo. Sin embargo, éste tipo de mecanismo no permite la compartición de recursos de memoria entre tareas no relacionadas localizadas en diferentes máquinas; para poder lograrlo se debe crear un administrador de memoria externo, esto es, un administrador diseñado por el usuario que si ofrezca dicha posibilidad. Este administrador es una aplicación especializada a nivel usuario que debe realizar las funciones de paginación y uso de memoria, proporcionando integridad, coherencia y seguridad en los objetos de memoria compartidos. A continuación, se detalla de manera informal el modo de operación del AME y se describen las instrucciones de memoria que debe incluir para poder desarrollar la memoria compartida distribuida.

4.2 Elementos básicos de un administrador de memoria externo

Un AME debe considerar intercambios de información entre tareas que solicitan el uso de memoria a los diferentes núcleos de Mach y los mecanismos necesarios para realizar funciones de administración de la memoria. Existen tres elementos principales que intervienen en este sistema:

a) El cliente o tarea que solicita los servicios de manejo de memoria,

b) El paginador dedicado a controlar las operaciones de paginación y consistencia, y,

c) El núcleo de Mach, con características propias en su arquitectura, permite que las operaciones de paginación de la memoria virtual (tradicionalmente reservadas al sistema operativo) puedan ser manipuladas a nivel usuario.

4.3 Relaciones entre los elementos del AME

Cuando una tarea requiere utilizar áreas de memoria para manipular datos o estructuras, solicita los servicios del paginador. Cada tarea puede crear o terminar una o varias de estas áreas de memoria, denominadas en Mach objetos de memoria. Un objeto de memoria puede estar constituido por una o más páginas y sus propiedades están controladas por el paginador.

Los tres elementos participantes en el AME interactúan mediante interfaces asíncronas. Las interfaces usadas entre el paginador y los diferentes núcleos son transparentes al usuario (tarea cliente). Para que un cliente solicite los servicios de un paginador debe conocer su dirección o puerto de servicio para comunicarse con él. Bajo el contexto de Mach, el paginador debe registrar sus servicios en una entidad externa (proporcionada por Mach) la cual pone a disposición de los diferentes clientes, los servicios registrados en ella, entre los cuales se puede encontrar, el

servicio del paginador una vez registrado. Esta entidad es conocida como servidor de mensajes de red, SMR.

La relación entre los tres elementos básicos y la entidad externa de registro se muestra en la figura 4.1.

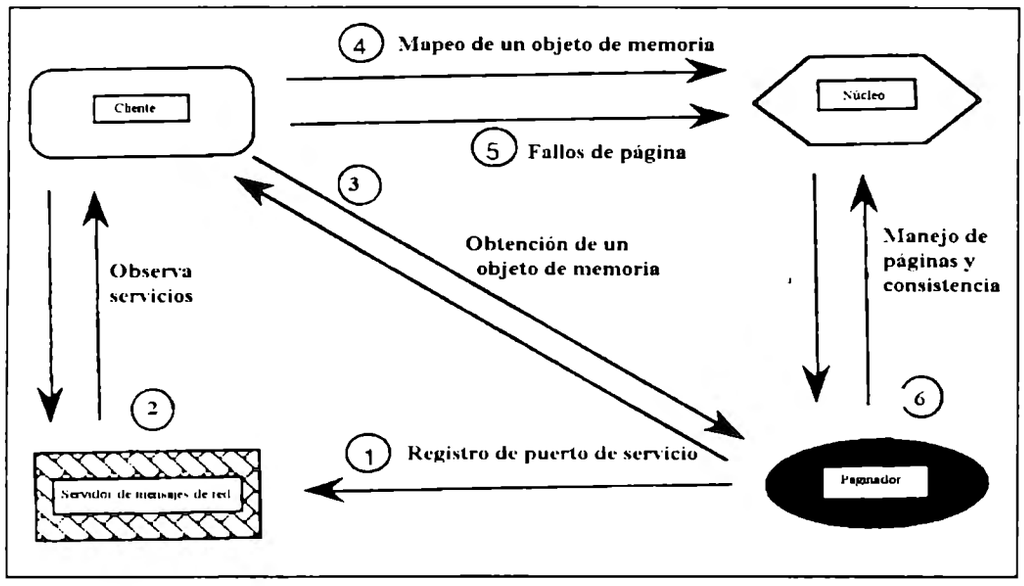


Fig. 4.1 Relaciones entre los elementos del AME.

Los pasos de la figura 4.1 se describen a continuación:

1.- El paginador ofrece servicios de manejo de objetos de memoria compartidos registrándose en la entidad externa SMR.

2.- El cliente obtiene el puerto de servicio del paginador por medio del SMR.

3.- El cliente se comunica directamente con el paginador, mediante el puerto de servicio obtenido en el paso anterior, para solicitar un objeto de memoria.

4 y 5 .- La tarea cliente pide al núcleo mapear todo o sólo una parte del objeto de memoria en su espacio de direcciones, generando posibles fallos de página.

6.- Para mapear y resolver los fallos de página generados en los pasos anteriores. el paginador y el núcleo intercambian mensajes asíncronos transparentes a la tarea cliente.

Los pasos anteriores representan cualquier operación que se realice sobre la memoria; conforme se avanza en el presente proyecto. cada operación es caracterizada y analizada con el objeto de integrar la especificación formal completa.

En la figura 4.1, es de interés señalar que la entidad más importante desde el punto de vista de la manipulación de los objetos de memoria es el paginador, cualquier decisión relacionada con el inicio. acceso. mapeo o terminación de algún objeto de memoria, involucra a dicho elemento. por lo cual se desarrolla una descripción profunda del mismo.

4.4 Funcionamiento y estructura del paginador

El funcionamiento general del paginador puede ser representada gráficamente utilizando un árbol simbólico de comportamiento. Esto se muestra en la figura 4.2.

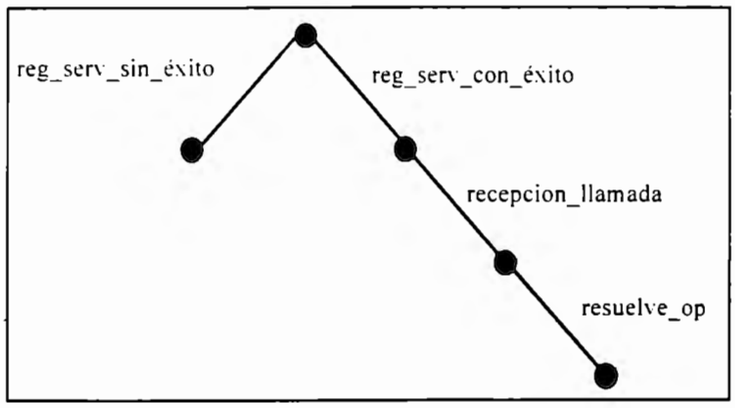


Fig. 4.2 *Árbol simbólico de comportamiento del paginador.*

En la figura 4.2, cada nodo del árbol representa un estado y cada una de las ramas el evento que se lleva a cabo entre los estados. En el árbol se muestra que el primer evento que debe suceder es

el registro de los servicios del paginador en el núcleo. Si no se logra esta operación, entonces se detiene el paginador. En el caso contrario, los eventos subsecuentes son la recepción de una llamada *-recepcion_llamada-* y en seguida la solución a la misma *-resuelve_op-*. Este árbol representa un comportamiento monolítico y su aproximación en términos de LOTOS es:

reg_ser >> recepcion_llamada; resuelve_op; stop

Donde el operador >> proporciona la habilitación de una secuencia de eventos. Si el comportamiento definido antes del operador es exitoso, se llevan a cabo los eventos posteriores a él. Si el resultado de dicho comportamiento no es exitoso, se detiene el comportamiento del sistema.

En el árbol de la figura 4.2 no se especifica el tipo de llamada que recibe el paginador. En la figura 4.3 se muestra un árbol en el que se indica a mayor detalle el tipo de llamadas que puede recibir el paginador.

El paginador puede recibir llamadas de un cliente solicitando algún objeto de memoria, o de algún núcleo requiriendo la solución a una operación relacionada con la manipulación de un objeto de memoria, como lectura, mapeo, etc. En la figura 4.3 el árbol expone los dos tipos de llamas. Además, describe los diferentes tipos de solución que el paginador puede realizar. Si un cliente realiza una llamada, la única posibilidad de solución es que éste obtenga el objeto de memoria requerido *-obtencion-*. Si la llamada es realizada por el núcleo, existen cuatro posibilidades de solución *-lectura, escritura, mapeo* y *terminación-*. Al finalizar cualquier tipo de solución, como se puede apreciar en el árbol, se aclara que la transición de estado del sistema consiste en regresar al estado de recepción de llamadas.

Los dos árboles de comportamiento de las figuras 4.2 y 4.3 expresan la conducta del paginador. En cada uno de ellos se representan los diferentes conjuntos de eventos que ocurren de acuerdo a las condiciones internas del sistema (que aún no se detallan), sin embargo, la diferencia principal entre ambos radica en que en el primero se representa la conducta más general del paginador sin entrar en detalles específicos como preguntarse ¿de quién recibe la llamada del paginador? o ¿qué

tipo de solución debe dar a la llamada?, mientras que en el segundo comportamiento se aclaran estas preguntas describiendo ciertas peculiaridades del mismo. De esta forma, en el segundo árbol se desarrolla un refinamiento del primero.

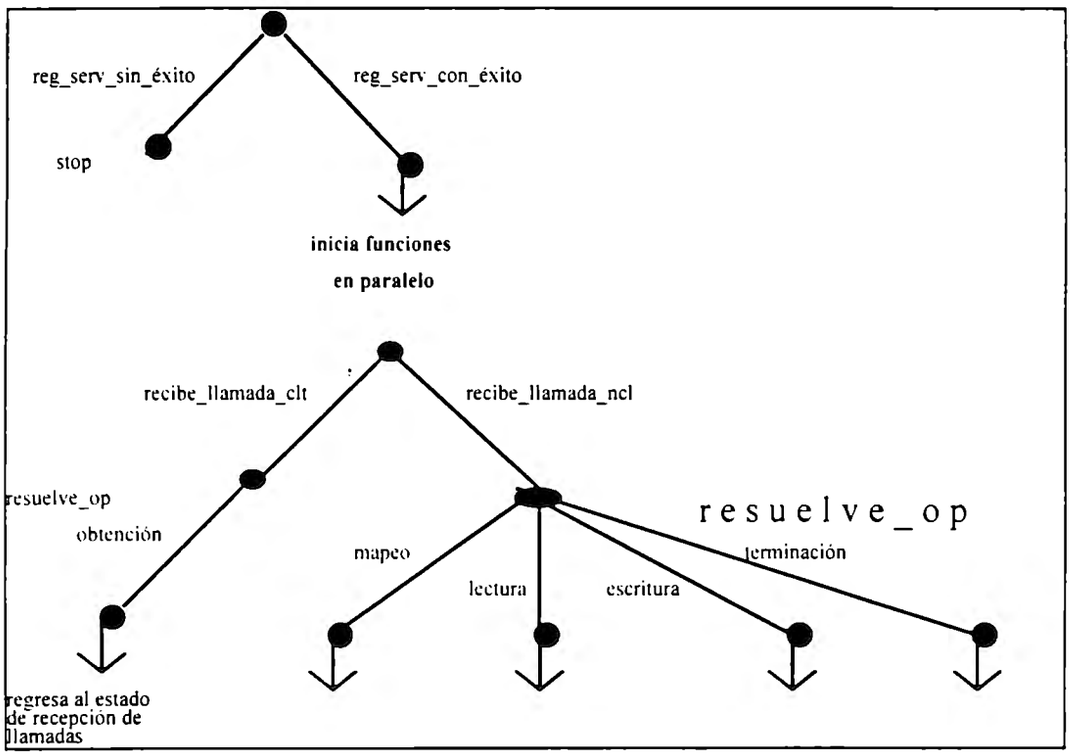


Fig. 4.3 Árbol de comportamiento del paginador

Otra observación de esas dos figuras es su construcción. La figura 4.2 muestra un diagrama monolítico del sistema. No existen eventos o acciones que se realicen en paralelo, el comportamiento es netamente secuencial. Por el contrario, la figura 4.3 presenta un diagrama ramificado en el que se indican ciertas acciones realizadas en paralelo, las cuales al ser terminadas, regresan a un estado intermedio, y pueden ser solicitadas de forma recursiva.

La estructura del paginador, desde el punto de vista de un programador, está constituida por tres módulos principales:

a) Módulo de iniciación y de monitoreo.

Este módulo lleva a cabo operaciones necesarias para la inicialización del programa, realiza las operaciones de registro de los servicios de memoria dentro del servidor de mensajes de red y monitorea el puerto de servicio, generado con el registro para la atención de llamadas. Es responsable de las operaciones de iniciación y destrucción de conexiones con los clientes (solicitudes de obtención de un objeto de memoria). También tiene la responsabilidad de generar los procesos necesarios para el funcionamiento del paginador, así como de la terminación de los mismos.

b) Módulo de control de objetos.

Implanta las operaciones del administrador de memoria externo manipulando las solicitudes de fallos de página y de liberación de objetos de memoria, generadas por los clientes o por el núcleo.

c) Módulo de control y administración de la memoria compartida distribuida.

Es responsable del acceso y compartición de objetos remotos de manera que un cliente, ubicado en cualquier parte dentro de la red, pueda accesar objetos compartidos.

Partiendo del conocimiento general de un administrador de memoria externo, sus elementos y sus relaciones principales, se continua con una descripción más profunda del mismo.

4.5 Operaciones en el uso de memoria

Las operaciones para el uso de memoria se pueden dividir en :

a) Creación de áreas de memoria. Involucra la creación, obtención o inicialización del objeto de memoria.

b) Lectura de datos. En esta operación se requiere un posible mapeo del objeto de memoria dentro del espacio de direcciones de la tarea que solicita la lectura, ocasionando fallos de páginas y por lo tanto, el manejo de las operaciones, denominadas de *page-in*, correspondientes.

c) Escritura de datos. Para ésta, se necesita un modelo de protección para mantener una coherencia entre los datos actualizados y sus diferentes copias compartidas.

d) Liberación de regiones de memoria. Terminación o destrucción del objeto de memoria.

Además, se deben considerar las operaciones requeridas por los núcleos, denominadas de *page-out*, cuando su memoria se encuentra saturada y requieren liberar áreas de memoria para otras aplicaciones o usos.

4.5.1 Obtención de un objeto de memoria

El mecanismo para que una tarea cliente pueda manipular una región de memoria se inicia con la obtención de un objeto de memoria. Este es el proceso inicial y cumple con la siguiente secuencia de pasos.

Primero, el paginador debe registrar sus servicios en el servidor de mensajes de red para que las tareas que los requieran, los puedan identificar en el puerto respectivo. En seguida, el cliente puede observar los puertos de los diferentes paginadores registrados en la entidad mencionada y obtener uno para hacer uso de sus servicios. Posteriormente, el cliente puede solicitar directamente al paginador un objeto de memoria.

El paginador, por su parte, una vez que el cliente solicitó el objeto de memoria, debe validar si éste existe o debe ser creado. En caso de que sea creado, el paginador genera una estructura interna para manejar al OM y un puerto conocido como puerto del objeto (el nombre del puerto del objeto de memoria está determinado según el espacio de nombres de puertos del paginador), que es enviado al cliente. Además, el núcleo asociado con la tarea solicitante genera dos puertos más, uno de control y otro de nombre y avisa al paginador de esta operación. La estructura interna mencionada está constituida por los siguientes campos:

a) Un contador de usuarios,

b) el nombre del archivo que contiene todas las páginas del objeto y su identificador (archivo 1, conocido como archivo contenedor de páginas),

c) el nombre del archivo verificador de páginas válidas (contenidas en el archivo 1) y su identificador (archivo 2, también conocido como archivo de mapa de páginas),

d) un mapa de páginas en memoria que inicialmente se encuentra vacío. Este es un arreglo de bits y cada página tiene asociado un bit: el valor de éste indica el estado de la página, será *V* cuando el paginador guarda la página en el archivo contenedor de páginas y *F* cuando la página ha sido modificada por el cliente y el paginador aún no ha registrado los cambios en el archivo correspondiente. Este arreglo se guarda eventualmente en el archivo de mapa de páginas (archivo 2) cuando el objeto ya no es usado.

Si el OM existe, el paginador ya tiene registrados sus archivos (de contenido de páginas y de páginas válidas) y sólo genera las estructuras necesarias para su control con la información existente, creando así un nuevo objeto. La inicialización de este objeto es similar a la descrita

anteriormente e incluye la creación y regreso al cliente de un puerto de objeto, además de la creación de los puertos respectivos por el núcleo.

Si el OM existe y tiene todas las estructuras necesarias para su manejo (archivos, mapa en memoria, puertos, etc.) el paginador únicamente registra al nuevo cliente incrementando el contador de usuarios.

Lo descrito anteriormente se expresa en la figura 4.4.

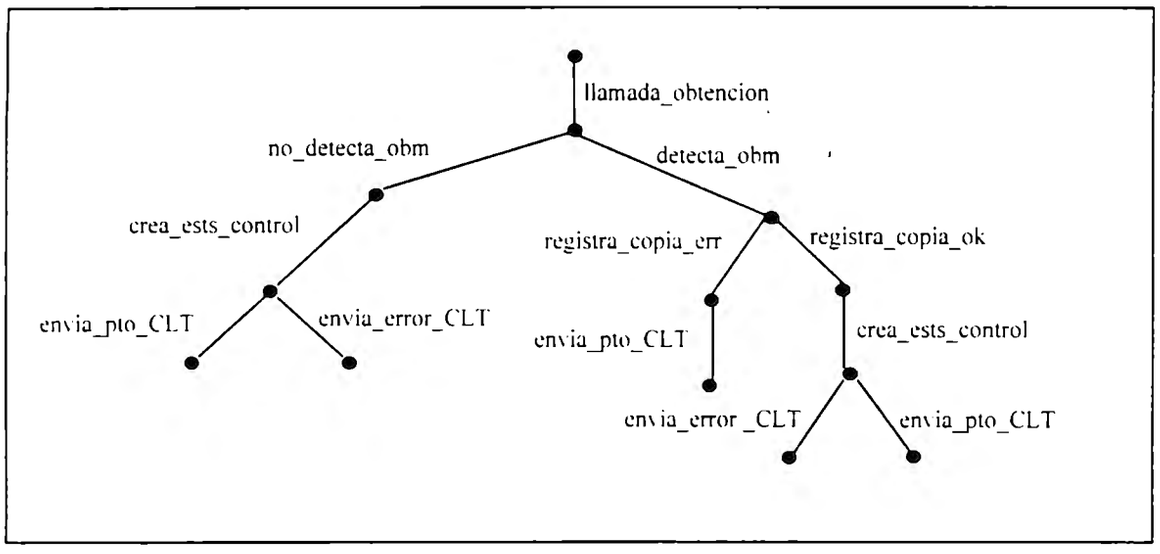


Fig. 4.4 Árbol simbólico de comportamiento de la operación de obtención de un OM.

El árbol inicia con el evento *llamada_obtencion*, el cual detecta un mensaje de petición de obtención de un OM enviado por algún cliente. El estado resultante tiene dos alternativas de ejecución: *no_detecta_obm* y *detecta_obm*. Estos eventos están relacionados con la verificación de la existencia del objeto de memoria en cuestión. El primero de ellos representa la situación en la que el OM no existe, por lo que es necesario crearlo, y el segundo expresa la verificación de existencia del OM.

Si no existe el OM, entonces deben de ser creadas las estructuras de control para su manejo -*crea_ests_control*-. El siguiente estado tiene dos opciones de ejecución: *envia_pto_CLT* y *envía_error_CLT*. Ambos envían resultados al cliente, uno exitoso y el otro erróneo.

Si el OM existe, el estado resultante puede llevar a cabo *reg_copia_err* o *reg_copia_ok*. Estos eventos efectúan la tarea de registro de copia con la finalidad de llevar un control de los clientes que tienen derechos de uso sobre el OM correspondiente. Si este registro no fue satisfactorio, el estado resultante sólo tiene la opción de ejecutar *envía_error_CLT* indicándole al cliente que no puede obtener el OM. Si el registro es exitoso, el evento que se ejecuta a continuación es: *crea_ests_control*. Este es responsable de crear las estructuras utilizadas para el control del OM, así como de los puertos correspondientes. Una vez más, el estado resultante tiene dos alternativas de ejecución: *envía_error_CLT* o *envía_pto_CLT* indicando al cliente solicitante la conclusión exitosa o errónea de la obtención del OM.

4.5.2 Mapeo de un Objeto de Memoria

Una vez que fue obtenido satisfactoriamente un OM, el cliente solicita acceso a porciones del mismo. Este proceso se inicia con la solicitud de mapeo del OM por parte del cliente, parcial o total, dicha petición está dirigida al núcleo. El núcleo debe realizar las operaciones necesarias para mapear el OM dentro del espacio de direcciones del cliente. Si un cliente pide un acceso a un OM y éste ya ha sido mapeado, el núcleo se encarga directamente de suministrar los datos. Si el OM no ha sido mapeado, para que el núcleo pueda completar estas operaciones, envía llamadas hacia el paginador y espera una respuesta de notificación de inicialización del mismo, la cual incluye el conjunto de atributos correspondientes al objeto. Es importante hacer notar que también existe la inicialización del OM por parte del cliente en el proceso de obtención del mismo. Esta inicialización se refiere a la realizada por el cliente cuando solicita el objeto al paginador. La notificación de inicialización que espera el núcleo tiene que ver con el tipo de llamada generada por éste cuando aún no tiene ninguna referencia del OM que le ha sido solicitado, esto es, cuando el cliente solicita mapearlo, automáticamente el núcleo interpreta esta llamada. Como el núcleo desconoce las características del OM, verifica el paginador que lo administra, establece una comunicación de inicialización de OM con él y espera la respuesta. El paginador genera como respuesta un mensaje para indicar los atributos del OM al núcleo.

Cada núcleo que mapea un objeto de memoria, almacena un conjunto de atributos para ese objeto. Estos atributos incluyen una bandera que indica si el objeto puede o no aceptar solicitudes de paginación aún más profundas y valores de especificación para las estrategias de copia y manejo del caché. Sin embargo, los atributos son específicos a un núcleo individual. Cuando múltiples núcleos mapean un objeto, cada uno tiene su propio conjunto de atributos para el objeto. Existe un tipo de llamada para escribir los atributos de un objeto de memoria, ésta se envía del paginador a un núcleo en específico y existe otro tipo de llamada paralela para leer dichos atributos.

En la figura 4.5 se presenta el árbol de comportamiento que indica este intercambio de información.

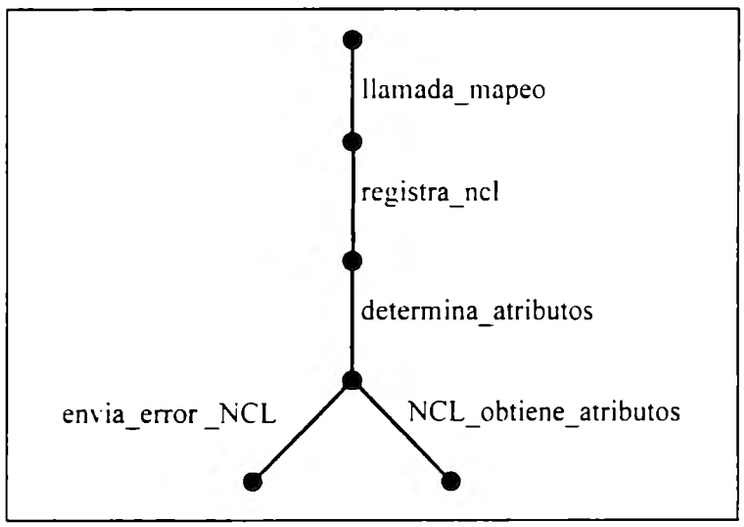


Fig. 4.5 Árbol simbólico de comportamiento de la operación de mapeo

La interpretación del esquema anterior es como sigue:

Cuando un núcleo detecta la petición de mapeo de un objeto de memoria por parte de un cliente, evalúa si ya lo inicializó con el paginador, si no es así, envía una llamada para tal propósito. El paginador recibe la solicitud del núcleo - evento *llamada_mapeo*-, en seguida, guarda la información acerca del núcleo generador del mensaje -*registra_ncl*-, que incluye el nombre del objeto y el tamaño de las páginas que soporta. A continuación, el paginador asigna los atributos

del objeto - *determina_atributos* -; si existe algún problema de consistencia, envía al núcleo un mensaje de error - *envia_error_ncl* - de lo contrario, le envía los atributos del objeto - *NCL_obtiene_atributos* -.

4.5.3 Acceso a un Objeto de Memoria

Una vez que fue mapeado satisfactoriamente un objeto, un programa cliente puede accesarlo. Si un cliente intenta utilizar una página de un objeto que no está en memoria, el núcleo del cliente transforma el fallo de página resultante en una solicitud enviada al paginador del objeto. El paginador toma la responsabilidad de suministrar los datos pedidos o regresar un mensaje de error. La respuesta de un paginador a una petición de datos puede caer en uno de los tres casos siguientes:

Primero, que suministre los datos, caso típico. Normalmente un paginador responde al llamado de requerimiento de páginas generado por el núcleo y le envía las páginas requeridas de regreso. El paginador tiene el control completo sobre este proceso. El paginador puede almacenar las páginas en disco, en cinta o buscarlas sobre la red. Una buena forma es almacenar las páginas en el disco en forma encriptada. Es recomendable utilizar el mismo tamaño de páginas para el paginador y para el núcleo, ya que si varían los tamaños se tendrían que hacer llamadas adicionales para manipular todos los datos, además de que se puede perder la integridad en la información.

Segundo, el paginador indica que la región es válida pero que no hay datos disponibles para ella. Este caso normalmente se presenta cuando se utiliza por primera vez una página en un objeto. El paginador tiene dos opciones: enviar un mensaje indicando que los datos del OM fueron suministrados y regresar al núcleo solicitante la memoria llena de ceros, o enviar un mensaje de datos no disponibles causando que el núcleo produzca páginas llenas de ceros para dicha región. El segundo mensaje es más eficiente que el primero. Mach tiene un código especial que optimiza el uso de páginas llenas de ceros, además en la segunda opción el paginador concede todos los posibles accesos a la región, no sólo los pedidos por el núcleo en su mensaje original.

Tercero, un paginador puede indicar que la solicitud no pudo ser resuelta, debido a un error. El paginador tiene también la opción implícita de no responder, aunque ésta acción no es deseable porque los hilos se quedan esperando por un fallo de página que quizá no sea realizado. Es benéfico utilizar códigos de error en el paginador que no generen conflictos con los mensajes de error del núcleo, por ejemplo :

1.- Error debido a offset.

2.- Error de búsqueda.

3.- Error de lectura.

En la figura 4.6 se presenta el árbol que representa el comportamiento para accesar un objeto de memoria.

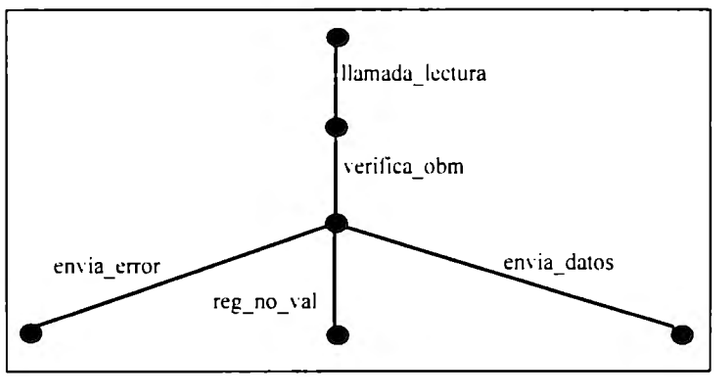


Fig. 4.6 Árbol de comportamiento para el acceso a un objeto de memoria

Una vez que el núcleo transformó el fallo de página en una solicitud de datos al paginador, éste la recibe - *llamada_lectura* -, verifica las condiciones del objeto - *verifica_obm* - y envía una respuesta al núcleo. La respuesta puede ser de tres tipos:

- *envia_error* cuando existe cualquier tipo de error en la lectura del objeto,
- *reg_no_val* indicando que la región del objeto no es válida y,
- *envia_datos* cuando los datos fueron encontrados y enviados satisfactoriamente.

4.5.4 Manejo del mecanismo de salida de páginas

Cuando un núcleo necesita memoria de páginas, modificadas por una aplicación para uso de otra, se deben remover esas páginas de los espacios de direcciones de los clientes que las estén utilizando, de tal forma que el cliente no pueda usarlas de nuevo hasta que no se haya realizado un fallo de página. El núcleo envía la página al paginador del objeto, entonces él decide que hacer con ella. Usualmente la almacena en disco.

4.5.5 Consistencia y fallo de protección de páginas

Una vez que el núcleo ha obtenido las páginas necesarias, la presencia de restricciones de protección puede prevenir que un programa cliente accese datos que ha mapeado. Un núcleo debe tener alguna forma de indicar a un paginador su necesidad de utilizar acceso menos restringido, y debe haber una forma en la cual el paginador responda a dicha solicitud. Aún más, debe haber un método por el cual un paginador pueda forzar a un núcleo a enviar de regreso las páginas modificadas o negar completamente las páginas del objeto. Esta habilidad es un prerrequisito pedido a un paginador para que provea consistencia entre diferentes clientes o entre diferentes núcleos que mapean el mismo objeto de memoria.

La solicitud de un núcleo para incrementar permisos en una página se envía al paginador a través de un mensaje. Si la página no existe en memoria, se envía un mensaje de solicitud de datos; pero si la página existe sin los permisos suficientes para completar la manipulación, entonces se utiliza un mensaje de fallo en el cambio de protección de página. Al realizarse una falla en el intento de incrementar los permisos de una página se le conoce como falla de protección.

El paginador responde a un mensaje de solicitud de cambio de permisos con otro mensaje de reconocimiento, poniendo un nuevo valor a los permisos de la región. Este mensaje también puede ser utilizado para controlar las operaciones de consistencia en páginas.

Los algoritmos relacionados con el manejo de consistencia de páginas de los objetos de memoria, constituyentes de un sistema de memoria compartida distribuida son especificados en un proyecto de tesis que se desarrolla en paralelo con este trabajo debido a su grado de complejidad [33].

4.5.6 Eliminación de un objeto de memoria

Cuando un objeto de memoria deja de ser usado en un núcleo, entonces debe ser eliminado. Para eliminar un objeto, el núcleo debe de tomar la decisión de negar a los usuarios algunos o todos los accesos al objeto. Esta decisión puede ser hecha de dos maneras. Si el objeto fue inicializado sin el atributo de ser objeto de memoria caché, entonces todos los accesos al objeto pueden ser negados inmediatamente. Si el objeto fue inicializado con el atributo de ser objeto de memoria caché, entonces el núcleo negará únicamente los accesos a ese objeto cuando no tenga memoria disponible y la memoria del objeto sea utilizada para otro propósito. También un objeto puede ser eliminado de manera inmediata si el núcleo le envía al paginador un mensaje para destrucción de un objeto dado, si ese objeto tuviera datos importantes, entonces el núcleo debió haber enviado previamente un mensaje de cambio en las protecciones del objeto para que de esa manera sea protegido y actualizado antes de ser destruido.

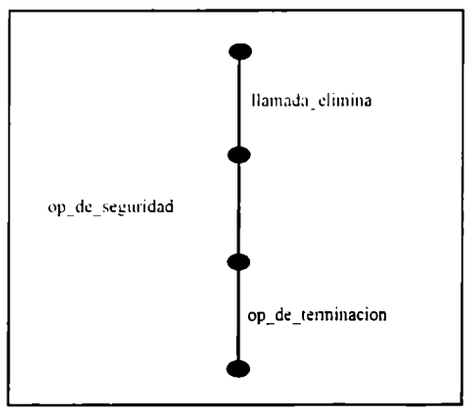


Fig. 4.7 Árbol de comportamiento para la eliminación de un objeto de memoria

En la figura 4.7 se presenta la ejecución de los eventos para la terminación de un OM. Esta secuencia de eventos inicia con la recepción de la llamada del núcleo para tal propósito - *llamada_elimina* -. El paginador realiza entonces las operaciones necesarias para la protección de archivos - *op_de_seguridad* - y finalmente realiza la operación de terminación del objeto - *op_de_terminacion* -.

CAPITULO 5

ESPECIFICACIÓN FORMAL DEL ADMINISTRADOR DE MEMORIA EXTERNO

5. Especificación Formal del Administrador de Memoria Externo

En el capítulo anterior se describe de manera informal el funcionamiento del administrador de memoria externo y la relación de sus elementos: en el presente capítulo se desarrolla la especificación del sistema de manera formal.

5.1 Principios de diseño de la especificación

La descripción formal de un sistema puede tener como su principal prioridad la descripción del comportamiento observable como una caja negra, o la descripción del comportamiento interno de ese sistema como una caja blanca. La primera descripción es nombrada extensional, mientras que la segunda es nombrada intencional. En la descripción extensional de un sistema sólo se muestran las interacciones observables y la ordenación temporal de sus relaciones está definida en términos de alternativas de secuencias o composición paralela de restricciones. En la descripción intencional son presentadas ambas interacciones del sistema, observables e internas.

El proceso de especificación puede llevarse a cabo a través de pasos, donde cada paso representa un nivel de abstracción del sistema. Primero se describe al sistema en términos de un conjunto de procesos que representan diferentes objetos, después cada proceso resultante se descompone en términos de subprocesos. La tarea de refinamiento del sistema se repite hasta haber terminado con las descripciones más simples, esto es, el punto donde no se puedan obtener más descomposiciones y los procesos descritos contengan los detalles suficientes para su implantación.

En la especificación del AME inicialmente se usa la descripción extensional y aplicando la técnica de refinamiento, los procesos se descomponen de tal forma que se realiza una descripción intencional del sistema.

Al iniciar con la especificación de este sistema se ha observado que es extremadamente difícil, por no asentar imposible, tratar de describir el comportamiento detallado del AME en una sola especificación, razón por la cual, se procede utilizando refinamientos.

5.2 Especificación inicial

En la especificación inicial del administrador de memoria externo (AME), se describe el comportamiento general del sistema constituido por tres procesos principales: *registro*, *pedir_pto* y *pedir_obm*. Primero se lleva a cabo el proceso *registro*. Posteriormente, si éste concluye exitosamente se inicia un comportamiento paralelo de los procesos *pedir_pto* y *pedir_obm*. Para poder representar la existencia de varios clientes simultáneos (unos intentado solicitar el puerto de servicio del paginador y otros deseando accesar los objetos de memoria) se utiliza un operador que permita la ejecución independiente y paralela de dichos procesos, ||| (ver figura 5.1 A). Esta conducta también puede ser especificada de forma equivalente utilizando los operadores **[]** y **;**,

según se muestra en la figura 5.1 B.

A	*B*
pedir_pto ||| *pedir_obm*	*pedir_pto* [] *pedir_obm* [] *pedir_pto; pedir_obm* [] *pedir_obm; pedir_pto*

Fig. 5.1 Comportamientos equivalentes

Como se aprecia en la figura 5.1 B, el desarrollo de la especificación con los operadores de selección requiere de un esfuerzo mayor no necesario. La conducta es no determinista y está regulada por las acciones efectuadas en el ambiente externo al sistema.

En la especificación inicial se utilizan las compuertas necesarias para describir las peticiones requeridas en cada uno de los procesos y son las siguientes: *pn, np, cn, nc, pc, cp*. Cada una de ellas representa alguna comunicación llevada a cabo entre las entidades del AME (cliente, núcleo y paginador). El nombre de cada una de estas compuertas está determinado por las iniciales de los elementos que se comunican y por el sentido de dicha comunicación, como se muestra en la figura 5.2.

pn : Para establecer las comunicaciones que se llevan a cabo del paginador al núcleo.
np : Para establecer las comunicaciones que se efectuan del núcleo al paginador.
cn : Para determinar las comunicaciones que se realizan del cliente al núcleo.
nc : Para establecer las comunicaciones que tienen lugar del núcleo al cliente.
pc : Para aquellas comunicaciones generadas por el paginador hacia el cliente.
cp : Para establecer las comunicaciones que se realizan del cliente al paginador.

Fig. 5.2 Compuertas de comunicación

La especificación inicial se nombra *AME1* utilizando la palabra reservada ***specification*** como lo muestra la figura 5.3 que contiene la especificación inicial completa.

Es importante hacer notar que las compuertas representan los eventos llevados a cabo por los procesos del AME, así mismo, el orden temporal de estos eventos equivalen al orden de ejecución real del sistema, es decir, su comportamiento.

Esta especificación precisa la estructura principal del AME y sirve de modelo para los refinamientos posteriores.

```
01    specification AME1 [pn, np, cn, nc, pc, cp] : noexit :=
02    behaviour
03            registro[pn, np]
04            >>
05            (
06            pedir_pto[cn, nc]
07            |||
08            pedir_obm[cp, pc, cn, nc, np, pn]
09            )
10    where
11    process registro[pn, np]: exit :=
12            pn; np; exit
13    endproc (* registro *)
14    process pedir_pto[cn, nc]: noexit :=
15            cn; nc; stop
16            |||
17            i; pedir_pto[cn, nc]
18    endproc (* pedir_pto *)
19    process pedir_obm[cp, pc, cn, nc, np, pn]: noexit :=
20        hide op_ini in
21            cp; pc; activa_obm[cn, nc, np, pn, op_ini]
22            |||
23            i; pedir_obm[cp, pc, cn, nc, np, pn]
24            where
25              process activa_obm[cn, nc, np, pn, op_ini]: exit :=
26                hide op_usa, t in
27                cn; np; op_ini; pn; nc; usa_obm[cn, nc, np, pn, op_usa,t]; exit
28                 where
29                  process usa_obm[cn, nc, np, pn, op_usa, t]: noexit :=
30                    (cn; (
31                          nc; usa_obm[cn, nc, np, pn,op_usa, t]
32                         []
33                         np; op_usa; pn; nc; usa_obm[cn, nc, np, pn,op_usa,t]
34                         []
35                         np; op_usa; usa_obm[cn, nc, np, pn,op_usa,t]
36                         ))
37                         [> t; stop
38                  endproc (* usa_obm *)
39            endproc (* activa_obm *)
40    endproc (* pedir_obm *)
41    endspec
```

Fig. 5.3 Especificación inicial, AME1.

La descripción del comportamiento del sistema inicia con la palabra reservada de LOTOS ***behaviour*** (línea 02), en ella se señala el proceso *registro*. Después del símbolo >> se expresa la conducta a ejecutar si el registro es exitoso, caso en el que los procesos *pedir_pto* (línea 06) y *pedir_obm* (línea 08) se pueden llevar a cabo; en los cuales, uno o más clientes pueden pedir al núcleo el puerto de servicio del paginador, así como requerirle objetos de memoria para su uso.

El proceso *registro* (línea 11) representa el envío de un mensaje del paginador al núcleo solicitándole el registro de sus servicios en un puerto para que sea dado a conocer a clientes que requieran algún tipo de servicio de memoria compartida. En la descripción del proceso *registro* existen dos eventos, *pn* y *np*, es decir, un evento de comunicación paginador-núcleo, para describir la solicitud y otro de comunicación núcleo-paginador representando la respuesta a dicha solicitud.

En la línea 14 se define el proceso *pedir_pto*, éste tiene dos compuertas, *cn* y *nc*, que representan la comunicación cliente-núcleo equivalente a la solicitud del puerto de servicio asignado al paginador, y una comunicación del tipo núcleo-cliente como su respuesta. Una vez ejecutadas estas dos comunicaciones se termina la secuencia de eventos para pedir un puerto de servicio por un cliente (*stop*). Mediante el operador ||| (línea 16) se expresa la generación simultánea e independiente de nuevas peticiones del puerto de servicio del paginador; la presencia del evento ***i*** señala una acción interna del sistema para generar dichas solicitudes.

Una vez que un cliente ha adquirido el puerto de servicio del paginador puede entonces realizar peticiones de objetos de memoria, este comportamiento se expresa en el proceso *pedir_obm* (líneas 19 a 40). En este proceso se tiene un evento denominado *op_ini* que es utilizado para establecer las operaciones de inicio de un objeto de memoria. Este evento permanece oculto en esta especificación inicial para no incluir detalles relacionados con la implantación.

Al inicio, un cliente envía un mensaje al paginador pidiendo un objeto de memoria (evento *cp*, línea 21), inmediatamente después recibe la respuesta del paginador a esta solicitud (evento *pc*, línea 21), en seguida se realizan operaciones para activar un objeto de memoria (proceso *activa_obm*, líneas 25 a 39), las cuales se describen más adelante. De forma análoga a la

generación de múltiples peticiones del puerto de servicio del paginador mencionadas en el párrafo anterior, por medio del operador |||, en este proceso también se desarrollan múltiples peticiones de objetos de memoria independientes y paralelas.

En el proceso *activa_obm* se describe el comportamiento general del uso de objetos de memoria. En este proceso también se incluye un evento denominado *op_usa*, el cual establece las operaciones realizadas sobre un objeto de memoria durante su uso y se detalla en los refinamientos subsecuentes. El primer evento que se ejecuta en *activa_obm* es una comunicación cliente-núcleo (evento *cn*, línea 27) para representar cualquier petición de operación de inicio realizada por un cliente sobre un objeto. Dicho mensaje es detectado por el núcleo y, por ser la primera operación a ejecutar sobre el objeto, el núcleo debe realizar una comunicación con el paginador (evento *np*) para inicializar el objeto de memoria (evento *op_ini*), el paginador manda atributos del objeto al núcleo (evento *pn*), el núcleo responde al cliente (evento *nc*), se invoca al proceso *usa_obm*, y se termina.

El proceso *usa_obm* (líneas 29 a 38) sirve para especificar las operaciones de uso genérico sobre el objeto (lectura, escritura, mapeo y terminación). Existe un evento *cn*, que expresa cualquier petición de algún cliente para hacer uso de un objeto de memoria, seguido a este evento hay tres comportamientos posibles; en el primero (secuencia de eventos de la línea 31), el núcleo contesta directamente al cliente la solución, *nc* y el proceso *usa_obm* se invoca recursivamente permitiendo nuevas operaciones sobre el objeto; en el segundo comportamiento (secuencia de eventos de la línea 33), el núcleo no puede resolver la petición en forma inmediata por lo que envía un mensaje al paginador pidiéndole la solución, *np*, este último desempeña operaciones de uso sobre el objeto, *op_usa*, responde al núcleo, *pn*, el núcleo responde al cliente, *nc*, y es invocado recursivamente el proceso *usa_obm*. En la tercera alternativa (secuencia de eventos de la línea 35) el núcleo envía un mensaje de petición al paginador, *np*, el paginador efectúa operaciones de uso sobre el objeto, *op_usa*, y se llama al proceso *usa_obm*. Este comportamiento recursivo finaliza si existe algún error en el sistema (línea 37), en este caso, sin importar que evento se ejecuta en un instante, se termina de manera abrupta el proceso *usa_obm*, como consecuencia de una falla de lectura, escritura o mapeo en algún objeto de memoria.

En esta especificación se describe de la forma más general el comportamiento del AME, para ello se utilizan compuertas que representan los intercambios de información entre las diferentes entidades, pero no el tipo de mensaje que se envía entre ellas. En este punto es necesario describir dichas compuertas, de manera que se precise el tipo de comunicación que se realiza entre las entidades.

5.3 Primer refinamiento

5.3.1 Justificación del primer refinamiento

En este primer refinamiento se describen nuevas compuertas equivalentes a las de la especificación inicial. La descripción consiste en descomponer cada una de las compuertas anteriores en diversas compuertas que se obtienen de acuerdo a las entidades que se comunican y al tipo de mensaje que se transfiere entre ellas.

Tomando como ejemplo la compuerta *pn* de la especificación inicial, se realiza su refinamiento, que consiste en la descomposición de *pn* en dos nuevas compuertas; *reg_pag* que representa el mensaje de solicitud de registro de servicios del paginador en el núcleo y *sol_pag* que es el mensaje de respuesta del núcleo al paginador. De manera similar se obtienen las demás compuertas, esto se presenta en la figura 5.4.

Compuertas especificación inicial	Compuertas primer refinamiento	Descripción
pn : ->	reg_pag	Mensaje que envía el paginador al núcleo para solicitar el registro del paginador en el núcleo
	sol_pag	Mensajes enviados del paginador al núcleo con soluciones a peticiones realizadas por el núcleo
cn : ->	pide_pto	Mensaje del cliente al núcleo para enviar la solicitud del puerto de servicio del paginador
	c_n	Mensajes de solicitudes de los clientes al núcleo
nc : ->	n_c	Mensajes de respuestas del núcleo hacia un cliente
	asigna_pto	Mensaje del núcleo al cliente para comunicar el puerto de servicio del paginador a un cliente
cp : ->	pide_obm	Mensaje del cliente al paginador para solicitar un objeto de memoria
pc : ->	asigna_obm	Mensaje que envia el paginador al cliente para poder accesar el objeto de memoria solicitado
np : ->	pet_nuc	Mensajes del núcleo para realizar peticiones al paginador
	not_nuc	Mensaje enviado del núcleo al paginador para notificar su registro

Fig. 5.4 Compuertas del primer refinamiento.

5.3.2 Especificación resultante del primer refinamiento

Este refinamiento, nombrando ahora la especificación como *AME2*, tiene un comportamiento equivalente al de la especificación inicial. La especificación resultante del primer refinamiento se muestra en la figura 5.5.

```
01 specification AME2[reg_pag, not_nuc, pide_pto, asigna_pto, pide_obm, asig_obm,
02                    pet_nuc, sol_pag, c_n, n_c, ] : noexit :=
03 behaviour
04     registro[reg_pag, not_nuc]
05     >>
06     (
07      pedir_pto[pide_pto, asigna_pto]
08      |||
09     pedir_obm[pide_obm, asig_obm, pet_nuc, sol_pag, c_n, n_c]
10     )
11  where
12     process registro[reg_pag, not_nuc]: exit :=
13             reg_pag; not_nuc; exit
14     endproc (* registro *)
15     process pedir_pto [pide_pto, asigna_pto] : noexit :=
16             pide_pto; asigna_pto; stop
17             |||
18             i; pedir_pto[pide_pto, asigna_pto]
19     endproc (* pedir_pto *)
20     process pedir_obm[pide_obm, asig_obm, pet_nuc, sol_pag, c_n, n_c] : noexit :=
21       hide op_ini in
22        pide_obm; asigna_obm; activa_obm[c_n, n_c, pet_nuc, sol_pag, op_ini]
23          |||
24        i; pedir_obm [pide_obm, asig_obm, pet_nuc, sol_pag, c_n, n_c]
25           where
26            process activa_obm[c_n, n_c, pet_nuc, sol_pag,op_ini] : exit :=
27              hide op_usa, t in
28                c_n; pet_nuc; op_ini; sol_pag; n_c;
29                usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa,t]
30             where
31              process usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t] : noexit :=
32               (c_n;(
33                     n_c; usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
34                  []
35                     pet_nuc; op_usa; sol_pag; n_c;
36                       usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
37                  []
38                     pet_nuc; op_usa;  usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
39                  ))[> t; stop
40             endproc (* usa_obm *)
41          endproc (* activa_obm *)
42       endproc (* pedir_obm *)
43 endspec
```

Fig. 5.5 Primer refinamiento, AME2.

5.3.3 Descripción de la especificación resultante del primer refinamiento

El refinamiento *AME2* inicia de forma análoga a la especificación inicial, tienen el mismo comportamiento indicado por los procesos *registro*, *pedir_pto* y *pedir_obm*, ya descritos anteriormente.

Uno de los principales cambios hechos en este refinamiento es el ya indicado en lo referente a la descomposición de la compuertas. Por ejemplo, en el proceso *registro* (línea 12) el paginador le solicita al núcleo el registro de sus servicios, indicado con el evento *reg_pag* y el núcleo le responde enviándole el puerto, lo cual se expresa con el evento *not_nuc*. Todos los cambios en los procesos que se incluyen en este refinamiento están relacionados con sus nuevas compuertas.

La equivalencia entre la especificación *AME1* y *AME2* se puede comprobar informalmente comparando un proceso específico en cada una de ellas. Tomando el proceso *usa_obm* para esta comparación, podemos observar en las figuras 5.6 y 5.7 que el comportamiento es el mismo y las diferencias respecto de la primera con la segunda radican en que en la última las compuertas son más representativas de las operaciones que se realizan en el sistema.

```
29    process usa_obm[cn, nc, np, pn, op_usa, t]: noexit :=
30      (cn; (
31          nc; usa_obm[cn, nc, np, pn,op_usa, t]
32          []
33          np; op_usa; pn; nc; usa_obm[cn, nc, np, pn,op_usa,t]
34          []
35          np; op_usa; usa_obm[cn, nc, np, pn,op_usa,t]
36          ))
37          [> t; stop
38    endproc (* usa_obm *)
```

Fig. 5.6 Proceso usa_obm *de la especificación* AME1.

```
31      process usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t] : noexit :=
32        (c_n;(
33              n_c; usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
34            []
35              pet_nuc; op_usa; sol_pag; n_c;
36                usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
37            []
38              pet_nuc; op_usa;  usa_obm[c_n, n_c, pet_nuc, sol_pag, op_usa, t]
39            ))[> t; stop
40      endproc (* usa_obm *)
```

Fig. 5.7 Proceso usa_obm *de la especificación* AME2.

Para comprobar la equivalencia de una manera formal se efectúa el siguiente procedimiento.

Sea *usa_o* el comportamiento del proceso *usa_obm* del refinamiento AME2 y *usa_o'* el comportamiento del mismo proceso después de aplicar la función de renombramiento *f_usa_o* definida de la manera siguiente:

$$f_usa_o(x) = \begin{cases} cn & \text{if } x = c_n \\ nc & \text{if } x = n_c \\ np & \text{if } x = pet_nuc \\ pn & \text{if } x = sol_pag \\ op_usa & \text{if } x = op_usa \\ t & \text{if } x = t \end{cases}$$

Fig. 5.8 Función de renombramiento para formalizar equivalencia entre especificaciones.

Aplicando al comportamiento *usa_o* la función *f_usa_o(x)*, para todo evento *x*, se obtiene el comportamiento *usa_o'*, que tiene un comportamiento idéntico a *usa_obm* de la especificación *AME1*, con lo cual se comprueba que son equivalentes los procesos *usa_obm* en ambas especificaciones. Si se aplica el mismo procedimiento a cada uno de los procesos de la especificación puede demostrarse la equivalencia entre los refinamientos AME1 y AME2.

En este refinamiento, con la ayuda de las nuevas compuertas, se expresa de una forma más clara el comportamiento del sistema, debido a que se dan a conocer los diferentes eventos que se desarrollan en el AME para ofrecer los servicios de compartición de objetos de memoria. Se observa también que es necesario profundizar en el refinamiento de las compuertas descritas en la presente especificación con la finalidad de representar ahora las funciones que se llevan a cabo sobre los objetos de memoria. Además, el proceso *usa_obm* debe ser detallado para expresar las diversas operaciones que desean realizar los clientes en dichos objetos.

5. 4 Segundo refinamiento

5.4.1 Justificación del segundo refinamiento

Para describir con mayor detalle el comportamiento del AME, se realiza el segundo refinamiento con el objetivo de representar las funciones particulares que se efectúan sobre los objetos de memoria. Para ello, se desarrolla una descomposición de las compuertas de la especificación anterior de manera que un evento general se precisa en dos o más eventos específicos, estos cambios pueden ser analizados en la figura 5.9.

En la figura 5.4 se puede observar que en el refinamiento anterior, la acción correspondiente a un cliente que genera una petición de operación sobre algún objeto de memoria, se representa con el evento *c_n*, el cual es una comunicación cliente-núcleo, es decir, el cliente necesita efectuar alguna tarea sobre un objeto; en cambio en el presente refinamiento (figura 5.9) dicho evento se descompone en tres eventos correspondientes a las diferentes funciones que puede pedir el cliente al núcleo, mapeo, lectura y escritura, *c_m_o*, *c_l_o* y *c_e_o*, respectivamente.

Compuertas segunda especificación	Compuertas tercera especificación	Descripción
c_n :	c_l_o	Cliente lee objeto de memoria
	c_e_o	Cliente escribe objeto de memoria
	c_m_o	Cliente mapea objeto de memoria
n_c :	n_s_l	Núcleo soluciona lectura
	n_s_e	Núcleo soluciona escritura
	n_s_m	Nnúcleo soluciona mapeo
pet_nuc :	n_i_o	Núcleo inicializa objeto
	n_a_o	Núcleo accesa objeto
	n_e_o	Núcleo escribe objeto
	n_t_o	Núcleo termina objeto
sol_pag :	p_da_o	Paginador determina atributos del objeto
	p_datos	Paginador envía datos a nucleo

Fig. 5.9 Cambios en compuertas del segundo refinamiento.

5.4.2 Especificación resultante del segundo refinamiento

La especificación obtenida en el segundo refinamiento es extensa y compleja, razón por la cual no se incluye de forma completa en este capítulo, sin embargo, ésta puede ser consultada en el apéndice **A1**.

5.4.3 Descripción de la especificación resultante del segundo refinamiento

En la especificación obtenida con el segundo refinamiento, además de las descomposiciones de las compuertas anteriormente descritas, se incluye la adición de los procesos que desarrollan las operaciones sobre objetos de memoria dentro del proceso *usa_obm* (líneas 35 a 77), *lee*, *escribe*, *mapea* y *termina*, respectivamente. La figura 5.10 presenta dichos procesos.

```
35        process usa_obm[c_l_o, n_s_l, n_a_o, p_datos, c_e_o, n_s_e, n_e_o,
36                        c_m_o, n_s_m, n_t_o, t] : noexit :=
37          hide op_usa in
38          (
39          lee [c_l_o, n_s_l, n_a_o, p_datos, op_usa]
40          []
41          escribe [c_e_o, n_s_e, n_e_o, op_usa, n_a_o, p_datos]
42          []
43          mapea [c_m_o, n_s_m, n_a_o, p_datos, op_usa]
44          []
45          termina [n_t_o, op_usa]
46          )       [> t; stop
47          where
48          process lee [c_l_o, n_s_l, n_a_o, p_datos, op_usa] :exit :=
49            c_l_o;(
50                  n_s_l; exit
51                  []
52                  n_a_o; op_usa; p_datos; n_s_l;exit )
53          endproc (* lee *)
54          process escribe [c_e_o, n_s_e, op_usa, n_e_o, n_a_o, p_datos, p_sol_esc]
55                  :exit :=
56                  c_e_o;
57                  solucion_esc[n_s_e, op_usa n_e_o, n_a_o, p_datos]
58                where
59                  process solucion_esc [n_s_e, op_usa, n_e_o, n_a_o, p_datos] :exit:=
60                          n_s_e; exit
61                          []
62                          n_e_o; op_usa; n_s_e; exit
63                          []
64                          n_a_o; op_usa; p_datos;
65                          solucion_esc [n_s_e, op_usa, n_e_o, n_a_o, p_datos]
66                  endproc (* solucion_esc *)
67          endproc (* escribe *)
68          process mapea [c_m_o, n_s_m, n_a_o, op_usa, p_datos] :exit :=
69                  c_m_o; (
70                          n_s_m; exit
71                          []
72                          c_m_o; op_usa; p_datos; n_s_m; exit )
73          endproc (* mapea *)
74          process termina [n_t_o, op_usa] :exit :=
75                  i; n_t_o; op_usa; exit
76          endproc (* termina *)
77        endproc (* usa_obm *)
```

Fig. 5.10 Proceso usa_obm *de la especificación* AME3.

El proceso *lee* (líneas 48 a 53) puede tener dos comportamientos después de que un cliente pide

leer un objeto de memoria -acción *c_l_o*-. El primero, se presenta cuando un cliente ya tiene asignado dentro de su espacio de direcciones el objeto de memoria, el núcleo soluciona la operación de lectura mediante el evento *n_s_l*, y sale -*exit*-. El segundo comportamiento es cuando el cliente todavía no tiene asignado dentro de su espacio de direcciones el objeto de memoria, en este caso, el núcleo le pide al paginador los datos solicitados por el cliente -evento *n_a_o*-, el paginador realiza operaciones de protección sobre la consistencia del objeto -acción *op_usa*- (no definidas en esta especificación), le proporciona los datos al núcleo -evento *p_datos*-, el núcleo soluciona al cliente la operación de lectura -*n_s_l*-, y termina -*exit*-.

En el proceso *escribe* (líneas 54 a 67), un cliente pide escribir un objeto -*c_e_o*-, y se ejecuta el proceso *solucion_esc* (líneas 59 a 66); este proceso tiene tres casos en los que un núcleo soluciona una operación de escritura. En el primero, el núcleo tiene los permisos de escritura sobre ese objeto de memoria, por lo que puede resolver la petición inmediatamente -*n_s_e*- y termina -*exit*-. En el segundo caso, el núcleo envía al paginador un mensaje de petición de escritura, -evento *n_e_o*-, el paginador realiza operaciones de protección sobre la consistencia del objeto -*op_usa*-, el núcleo soluciona la petición de escritura hecha por el cliente -*n_s_e*- y se concluye -*exit*-. En el tercer caso, se genera un fallo de página, el núcleo pide datos al paginador -evento *n_a_o*-, el paginador realiza operaciones de protección sobre el objeto -*op_usa*-, el paginador le envía los datos del objeto al núcleo -*p_datos*-, y se ejecuta nuevamente el proceso para realizar alguna de las otras dos opciones, según sea el caso.

En el proceso *mapea* (líneas 68 a 73) el cliente pide el mapeo de un objeto de memoria -evento *c_m_o*-, y en seguida se presentan dos opciones para solucionar la petición de mapeo de un objeto de memoria: en la primera, el núcleo soluciona el mapeo -*n_s_m*- y termina -*exit*-; en la segunda, el núcleo pide los datos del objeto de memoria al paginador -*n_m_o*-, éste realiza operaciones de utilización para mantener la consistencia del objeto -*op_usa*-, proporciona los datos del objeto al núcleo -*p_datos*-, el núcleo soluciona el mapeo (pone el objeto de memoria en el espacio de direcciones del cliente) -*n_s_m*-, y termina su ejecución -*exit*-.

El proceso *termina* (líneas 74 a 76) expresa el comportamiento de terminación de un objeto de memoria por parte del núcleo, en él se presenta un evento interno ***i*** que indica una posible

terminación. en seguida, el evento *n_t_o* (línea 75) es la petición del núcleo al paginador para la terminación de un objeto, el núcleo realiza las últimas operaciones sobre el objeto *-op_usa-* y finaliza el proceso *-exit-*.

Para dar una idea de la equivalencia de los refinamientos *AME3* y *AME2*, se analiza el mismo proceso en ambas especificaciones. En este caso se toma el proceso *usa_obm* para realizar dicha comparación. Para facilitar la comprensión de equivalencia de comportamientos se muestran los diferentes árboles de conducta en ambos casos.

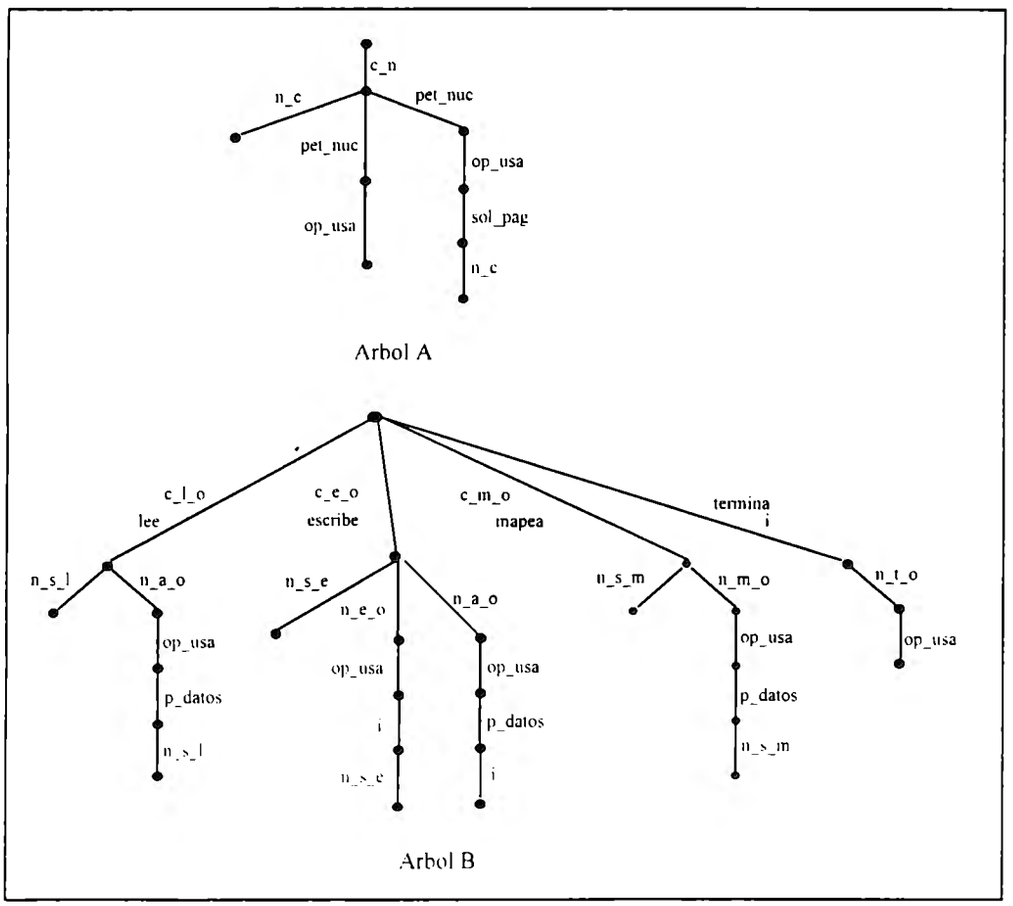


Fig. 5.11 Arboles de conducta equivalentes

El árbol A de la figura 5.11 representa el proceso *usa_obm* correspondiente a la especificación *AME2*, y el árbol B expresa el mismo proceso en la especificación *AME3*. En el árbol A se tiene un evento denominado *c_n* para expresar las diferentes operaciones que son solicitadas por un cliente a un núcleo para utilizar un objeto de memoria, mientras que en el árbol B dicho evento se

descompone en las cuatro funciones ofrecidas por el paginador: *lee -c_l_o-*, *escribe -c_e_o -*, *mapea -c_m_o-*, y *termina* -evento *i* -. En cada uno de los arboles, el evento inicial es una petición de un cliente hacia un núcleo. En seguida, en el árbol A se presenta un punto de decisión entre tres comportamientos (cabe mencionar que se asume la ejecución de sólo una operación en el árboles B en un instante, es decir, *lee*, *mapea*, *escribe* o *termina*, para entender la equivalencia). Analizando los arboles A y B, se puede observar que cada una de las funciones del árbol B tiene el mismo comportamiento que el árbol A; esto puede ser demostrado de una manera formal aplicando funciones de renombramiento, según se explica a continuación.

Sean *l*, *m* y *f* los comportamientos de los procesos *lee*, *mapea* y *termina* respectivamente, y sean *l'*, *m'* y *f'* los mismos comportamientos aplicándoles las funciones de renombramiento *f_l*, *f_m* y *f_f*, definidas como lo indica la figura 5.12.

```
                 c_n          if x = c_l_o
                 n_c          if x = n_s_l
f_l(x) =         pet_nuc      if x = n_a_o
                 op_usa       if x = op_usa
                 sol_pag      if x = p_datos

                 c_n          if x = c_m_o
                 n_c          if x = n_s_m
f_m(x) =         pet_nuc      if x = n_m_o
                 op_usa       if x = op_usa
                 sol_pag      if x = p_datos

                 c_n          if x = i
f_f(x) =         pet_nuc      if x = n_t_o
                 op_usa       if x = op_usa
```

Fig. 5.12 Funciones de renombramiento formal.

Aplicando la función *f_l* a los distintos eventos del comportamiento *l* se obtienen los iguientes renombramientos:

f_l(c_l_o) = c_n

f_l(n_s_l) = n_c

f_l(n_a_o) = pet_nuc

f_l(op_usa) = op_usa

f_l(p_datos) = sol_pag

Sustituyendo estos eventos en la rama correspondiente al proceso *lee* del árbol B de la figura 5.11 se obtiene un nuevo árbol de comportamiento indicado en la figura 5.13.

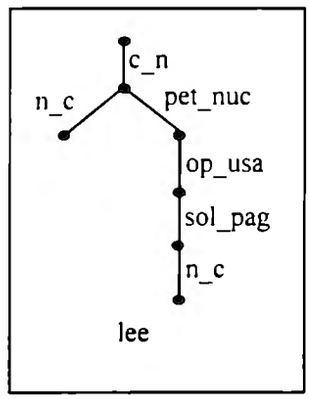


Fig. 5.13 Subárbol l'.

Reemplazando el subárbol *l'* en el árbol B, anteriormente descrito, se obtiene el proceso *usa_obm* de la especificación *AME3* dado en la figura 5.14.

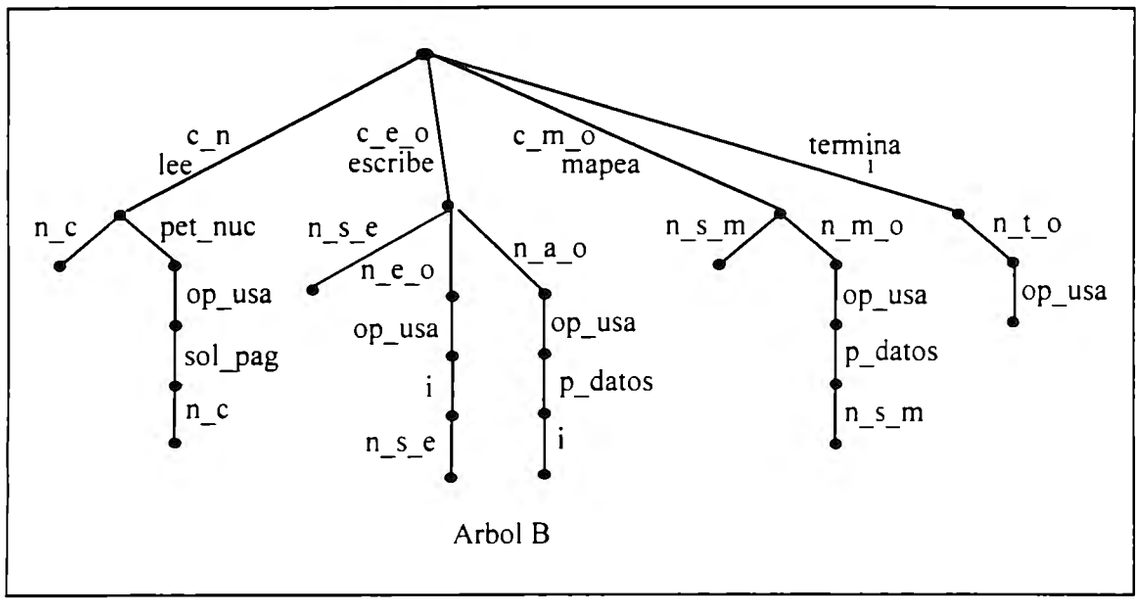


Fig. 5.14 Árbol B, con la sustitución de l por l'.

Ahora, si se aplican las funciones de renombramiento *f_m* y *f_f* a los eventos de los procesos *mapea* y *termina*, respectivamente, se obtienen los siguientes renombramientos:

f_m(c_m_o) =c_n

f_m(n_s_m) =n_c

f_m(n_m_o) =pet_nuc

f_m(op_usa) =op_usa

f_m(p_datos) =sol_pag

f_f(i) =c_n

f_f(n_t_o) =pet_nuc

f_f(op_usa) =op_usa

Los subarboles de cada proceso después de aplicar las funciones de renombramiento se ilustran en la figura 5.15.

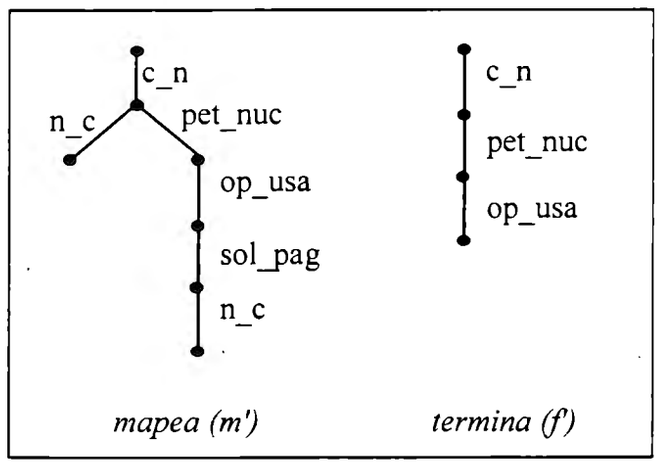


Fig. 5.15 Subarboles correspondientes a mapea y termina después de la aplicación de las funciones de renombramiento f_m y f_f.

Sustituyendo *m'* y *f'* en el árbol B original se obtiene el comportamiento mostrado en la figura 5.16.

Como se puede observar en la figura 5.16, las ramas correspondientes a *lee* y a *mapea* tiene un comportamiento idéntico, por lo que se puede reducir el árbol según se indica en la figura 5.17.

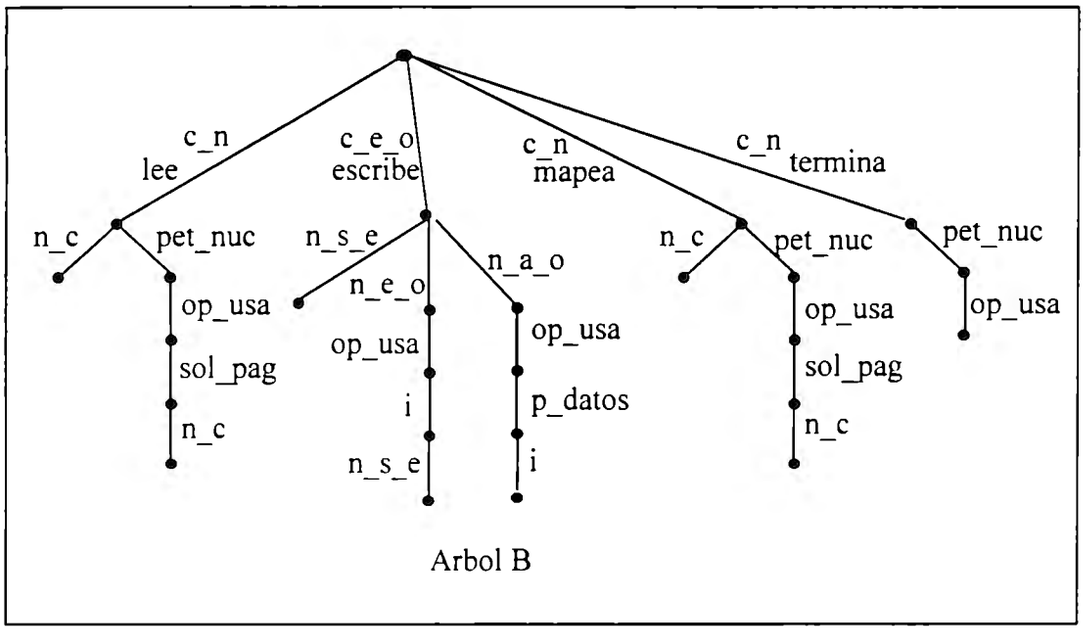


Fig. 5.16 Arbol B después de sustituir m' y f'.

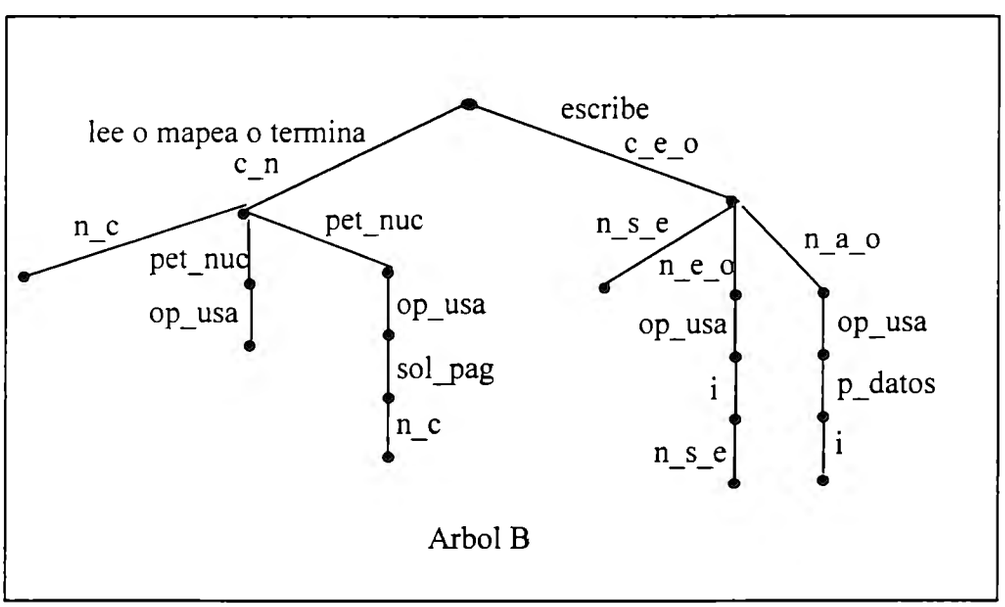


Fig. 5.17 Arbol B reducido.

En este punto, para demostrar la equivalencia del proceso *escribe* se define a *e* como el

comportamiento del proceso *escribe* de la siguiente forma:

e=c_e_o; (e1 [] e2 [] e3)

donde:

e1=n_s_e; ***exit***

e2=n_e_o; op_usa; ***i;*** *n_s_e;* ***exit***

e3=n_a_o; op_usa; p_datos; ***i;*** ***exit***

Sean las funciones *f_e1*, *f_e2* y *f_e3* definidas en la figura 5.18. Aplicando dichas funciones a los comportamientos *e1*, *e2* y *e3*, respectivamente, se obtienen los comportamientos:

e1'=n_c; ***exit***

e2'=pet_nuc; op_usa; sol_pag; n_c; ***exit***

e3'=pet_nuc; op_usa; sol_pag; n_c; ***exit***

$$f_e1(x) = n_c \quad \text{if } x = n_s_e$$

$$f_e2(x) = \begin{cases} pet_nuc & \text{if } x = n_e_o \\ op_usa & \text{if } x = op_usa \\ sol_pag & \text{if } x = i \\ n_c & \text{if } x = n_s_e \end{cases}$$

$$f_e3.(x) = \begin{cases} pet_nuc & \text{if } x = n_a_o \\ op_usa & \text{if } x = op_usa \\ sol_pag & \text{if } x = p_datos \\ n_c & \text{if } x = i \end{cases}$$

Fig. 5.18 Funciones para demostrar la equivalencia de comportamientos de escribe y usa_obm.

El subárbol de comportamiento del proceso *escribe* que se obtiene después de aplicar las

funciones anteriormente descritas se ilustra en la figura 5.19 a), así mismo se aprecia que las ramas correspondientes a las secuencias de eventos *e2'* y *e3'* son iguales, razón por la cual dicho subárbol puede ser reducido a dos ramas (figura 5.19 b).

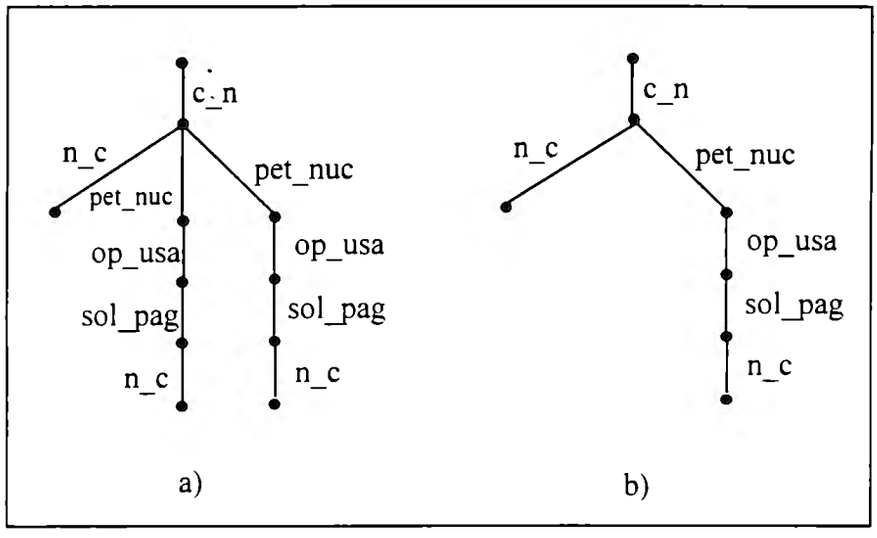


Fig. 5.19 Subarboles de comportamiento del proceso escribe después del renombramiento.

Sea e' = *c_e_o; (e1' ∏ e2' ∏ e3')* , el comportamiento resultante del renombramiento de los procesos *e1*, *e2* y *e3*. Dado que los comportamientos *e2'* y *e3'* son los mismos, *e'* puede reescribirse como:

e' = *c_e_o; (e1' ∏ e2')*

Sea la función de renombramiento:

$$f_e1(x) = \begin{cases} c_n & \text{if } x = c_e_o \\ x & \text{if } x \neq c_e_o \end{cases}$$

Aplicando *f_e* a los eventos de e', se obtiene el comportamiento equivalente a los procesos *lee* y *mapea*, como se muestra en la figura 5.20a y dado que son iguales, pueden reducirse a un solo comportamiento, quedadno el árbol resultante de la figura 5.20 b. Este último es equivalente al árbol A de la figura 5.11, con lo cual, queda demostrada la equivalencia de comportamientos

entre los procesos *usa_obm* de la especificación *AME2* y *usa_obm* de la especificación *AME3*.

De forma análoga, puede ser comprobada la equivalencia de comportamientos en el resto de los procesos refinados.

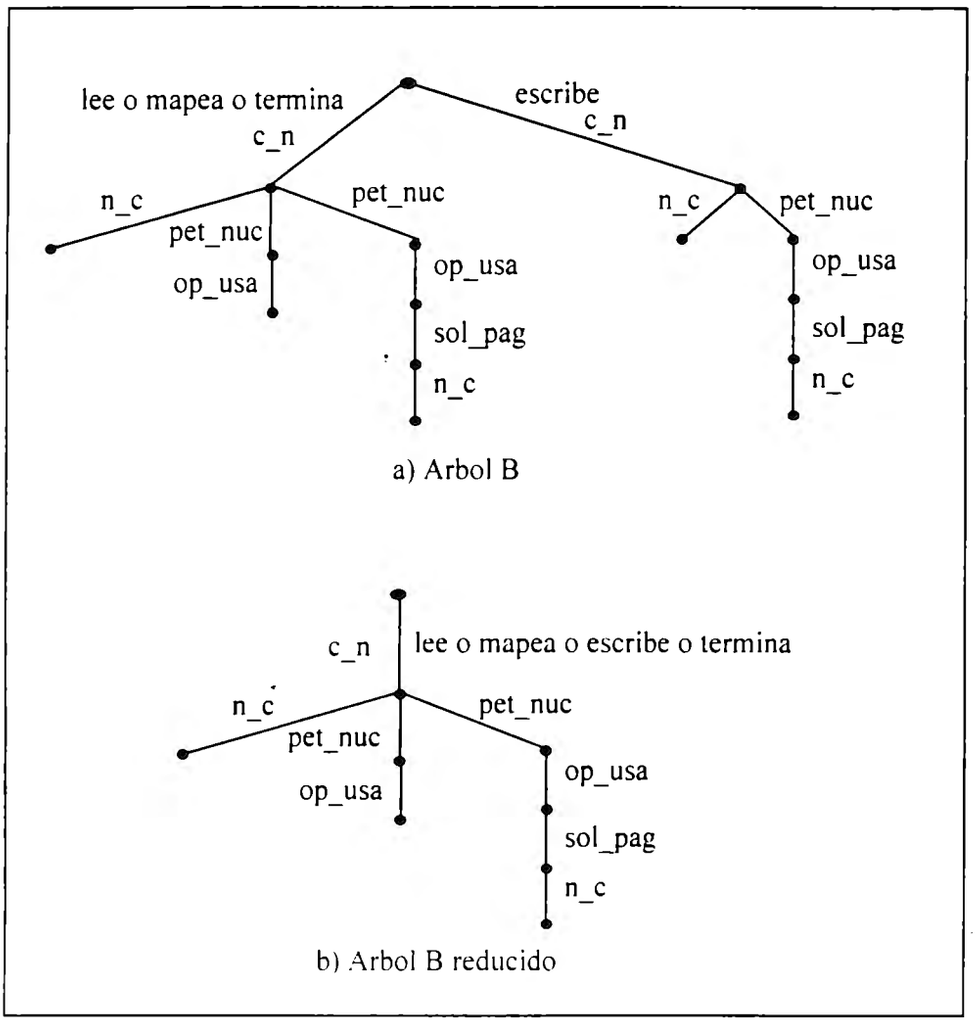


Fig. 5.20 Arbol del proceso usa_obm de la especificación AME3.

En este segundo refinamiento se establecen compuertas para denotar el tipo de mensaje intercambiado entre los elementos del AME. Estas facilitan el entendimiento de las distintas funciones del sistema. También, se expresan como procesos algunos eventos del proceso *usa_obm* del refinamiento anterior, lo cual ayuda a definir las operaciones particulares sobre los objetos de memoria.

Ahora, es importante definir una descomposición de los eventos de comunicación especificados, de tal forma que se separen en dos partes, la transmisora y la receptora; además, se debe establecer sincronización en el envío y recepción de dichos mensajes.

5.5 Tercer refinamiento

5.5.1 Justificación del tercer refinamiento

En la especificación del segundo refinamiento, la comunicación de los procesos se define de forma extensional: sólo se indica por medio de una compuerta representativa el tipo de comunicación que se efectua entre los distintos elementos. En este tercer refinamiento se realiza una descripción intencional de las comunicaciones entre procesos, concretando la sincronización entre ellos. Para lograrlo, se utilizan operadores de sincronización, operadores de interacción, tipos de datos abstractos y nuevas compuertas a través de las cuales se transfieren valores y variables. Se incluyen guardias para verificar que los valores de los datos transferidos sean recibidos correctamente y para generar la conducta deseada en los diferentes procesos del sistema.

5.5.2 Especificación resultante del tercer refinamiento

La especificación que se obtiene como resultado del tercer refinamiento se presenta completa en el apéndice **A2**. A continuación se proporciona una explicación del refinamiento realizado para su obtención.

5.5.3 Descripción del tercer refinamiento

La definición de interfases del sistema tiene que tomar en cuenta los valores de los datos intercambiados en las compuertas. La combinación de las compuertas y sus parámetros de valor normalmente se denomina **estructura de eventos** [4].

Los valores de los datos intercambiados en las compuertas son necesarios para conocer la información que se intercambia en el sistema AME y el sentido en que viaja. Para determinar ésto, se renombran las compuertas que representan eventos de comunicación como tipos de datos que cumplan con las características necesarias de cada operación. Además, la utilización de estos tipos de datos permite incluir un conjunto de compuertas en donde se intercambian los eventos entre los diferentes procesos. Los tipos de datos definidos a partir de las compuertas del refinamiento anterior se muestran en la figura 5.21.

Tipo de dato		Nombre de compuertas del refinamiento anterior
com_cn_nc	->	*c_l_o, c_c_o, c_m_o, n_s_l, n_s_e, n_s_m, pide_pto, asigna_pto*
com_np_pn	->	*n_i_o, n_a_o, n_e_o, n_t_o, reg_pag, p_da_o, p_datos, not_nuc*
com_pc_cp	->	*pide_obm, asigna_obm*

Fig. 5.21 Tabla de tipos de datos en la especificación AME4.

En la figura 5.21 se puede analizar que el tipo de dato *com_cn_nc* representa cualquier mensaje intercambiado entre un cliente y un núcleo, por lo que incluye todos los eventos usados en la especificación *AME3* relacionados con alguna comunicación cliente-núcleo o núcleo-cliente. También, se puede observar que el tipo de dato *com_np_pn* agrupa cualquier comunicación llevada a cabo entre un núcleo y un paginador, razón por la cual contiene todos los posibles mensajes que se intercambian entre ellos. Así mismo, el tipo *com_pc_cp* contempla todas las comunicaciones relacionadas entre un cliente y un paginador.

Para que el intercambio de estos tipos de datos se efectúe eficientemente se definen tres compuertas, *comp_cn, comp_np* y *comp_pc*, que representan el medio de comunicación entre un par de entidades dado, esto se muestra en la figura 5.22.

Compuerta	Descripción
comp_cn ->	*Compuerta para los eventos entre el cliente y el núcleo*
comp_np ->	*Compuerta para los eventos entre el núcleo y el paginador*
comp_pc ->	*Compuerta para los eventos entre el paginador y el cliente*

Fig. 5.22 Compuertas de comunicación entre pares de entidades

En cuanto a las operaciones de inicialización, manipulación y terminación de los objetos de memoria se continúan utilizando las compuertas *op_ini*, *op_usa* y *t*, descritas en párrafos anteriores.

La sustitución de eventos por tipos de datos en este refinamiento, nos permite expresar el paso de diferentes mensajes por la misma compuerta, facilitando así el análisis del comportamiento del sistema AME.

5.5.4 Descripción de la especificación resultante del tercer refinamiento

La especificación *AME4*, inicia con la definición de los tipos de datos. En las definiciones de cada uno de ellos se especifica un conjunto de operaciones, correspondiente a las acciones que se presentan entre cada uno de los elementos del AME.

Los tipos de datos facilitan el entendimiento del funcionamiento del sistema. Se escribe un tipo de dato y haciendo uso de los operadores de manejo de sincronía y de interacción (||, |//|, |||, ! y ?), se expresa de manera formal el tipo de comunicación presente. Se realiza el refinamiento que consiste en la aplicación de los tipos de datos para transmisión y recepción de información entre las entidades existentes.

Para facilitar el análisis de sincronía entre eventos se utiliza como ejemplo el proceso *lee*, estos se presentan en la figura 5.23.

Cliente	Compuerta |[comp_cn]|	Núcleo	Compuerta |[comp_np]|	Paginador	Operador	Línea en la especificación
!c_l_o	→	*?c_l_o*			;	75
					//	81
?n_s_l	←	*!n_s_l*			;	77-79
?n_s_l	←	*!n_a_o* *?p_datos* *!n_s_l*	→ ←	*?n_a_o* *op_usa* *!p_datos*	; ; ;	83-87

Fig. 5.23 Análisis de sincronía entre eventos

Las columnas de dicha tabla indican: la operación que realiza el cliente, el sentido en que viaja la información a través de la compuerta *comp_cn*, la operación realizada por núcleo, el sentido de la transferencia de información por la compuerta *comp_np*, la función que efectúa el paginador, el operador por medio del cual se controla la secuencia de eventos y el número de línea en el cual se encuentra el código dentro de la especificación *AME4*. El comportamiento está constituido por una secuencia de eventos sincronizados con el operador |[*x*]|, donde *x* es la compuerta mediante la cual se establece la sincronía. A continuación se explica dicha conducta: el cliente pide leer un objeto por medio de la compuerta *comp_cn* -evento *!c_l_o*- y el núcleo recibe la petición -evento ?c_l_o-; después, se tiene un comportamiento no determinista entre dos situaciones que depende de la localización del objeto de memoria. En la primera, cuando el objeto se encuentra en la memoria de la máquina en la que está situado el proceso cliente, el núcleo soluciona la operación de lectura enviando los datos al cliente a través de la compuerta *comp_cn* -evento *!n_s_l*- y el cliente los recibe -evento *?n_s_l*-; en la segunda, cuando el objeto no está en la memoria de la máquina cliente, el núcleo envía la petición de los datos al paginador usando la compuerta *comp_np* -evento *!n_a_o*-, el paginador recibe la solicitud -evento *?n_a_o*-, realiza las operaciones de protección sobre el objeto -evento *op_usa*-, envía los datos solicitados al núcleo mediante *comp_np* -evento *!p_datos*-, el núcleo los recibe -evento *?p_datos*-, éste los envía al cliente por la compuerta *comp_cn* -evento *!n_s_l*- y el cliente los recibe -evento *?n_s_l*.

El refinamiento de los demás procesos de la especificación se llevo a cabo utilizando la metodología explicada previamente.

metodología explicada previamente.

Para comprobar la equivalencia entre esta especificación y la anterior se pueden utilizar arboles de conducta como los mostrados en la figura 5.24, los cuales expresan el comportamiento del proceso *lee* en ambas especificaciones, respectivamente.

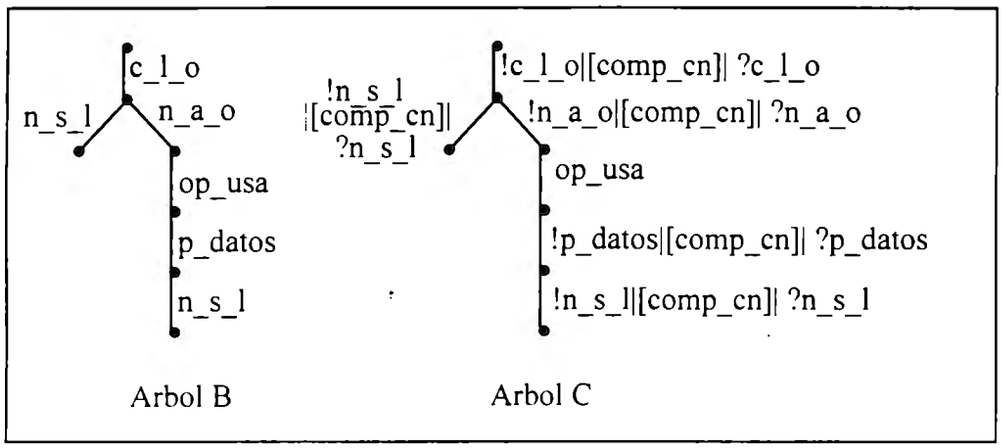


Fig. 5.24 Arboles de comportamiento equivalentes.

La figura 5.24 muestra dos arboles de comportamiento del proceso *lee*. El árbol B de esta figura representa el comportamiento del proceso *lee* de la especificación *AME3*. El árbol C expresa la misma serie de eventos que el árbol B, sin embargo, en éste último dichos eventos se descomponen en la parte emisora y en la parte receptora sincronizadas por medio de su compuerta correspondiente, mientras que en el árbol B dichos eventos expresaban a las comunicaciones como operaciones atómicas.

En este refinamiento se incorporan a la especificación los operadores de sincronía y paso de mensajes en los procesos con el objeto de definir una secuencia lógica del comportamiento del sistema, así mismo, se adiciona tipos de datos para clasificar los diferentes mensajes que se transfieren.

Conforme se refina la especificación crece el número de eventos derivados de las descomposiciones de eventos anteriores y por lo tanto, surge la necesidad de separar las funciones que corresponden a cada entidad del AME (paginador, núcleo y cliente). En el siguiente

refinamiento se deben definir las operaciones para cada entidad en relación a los servicios que proporcionen o soliciten.

5.6 Cuarto refinamiento

5.6.1 Justificación del cuarto refinamiento

En el presente refinamiento se definen tres nuevos procesos: *PGR*, *NCL* y *CLT* para clasificar las funciones correspondientes a cada uno de ellos. La forma en la cual se desarrolla la clasificación de funciones está íntimamente relacionada con las prestaciones o solicitudes de servicios de dichas entidades. Esta separación de procesos no altera el comportamiento del sistema, proporciona un acercamiento a la implantación del mismo y hace crecer el tamaño de la especificación considerablemente.

5.6.2 Especificación resultante del cuarto refinamiento

La especificación resultante del cuarto refinamiento se presenta en el apéndice **A3**. A continuación se proporciona una explicación del refinamiento desarrollado para la obtención de la misma.

5.6.3 Descripción del cuarto refinamiento

Al inicio de la definición del comportamiento de la especificación (líneas 17- 21) se fijan los

procesos *PGR*, *NCL* y *CLT*, representando los tres elementos principales del AME, paginador, núcleo y cliente, respectivamente. En la definición de cada uno de estos procesos se respeta el comportamiento establecido en las especificaciones previas y se incluye un conjunto de operaciones referentes al uso de los objetos de memoria, clasificadas de acuerdo al comportamiento de cada uno de los elementos del sistema, es decir, el grupo de operaciones relacionadas con la entidad *CLT* están referidas a peticiones de servicio de objetos de memoria, las operaciones de *NCL* están directamente ligadas a la coordinación y control de las peticiones de los clientes y respuestas del paginador, y las funciones del proceso *PGR* contienen todos las acciones de manipulación, liberación, asignación y terminación de objetos de memoria.

Para poder apreciar la aproximación a la implantación del sistema obtenida con este refinamiento, es necesario analizar procesos cooperantes entre cada una de las partes. Esto se logra haciendo una identificación del elemento que genera una petición y del que la resuelve en un proceso mutuo. Con ayuda de la figura 5.25 se puede analizar el proceso para realizar una lectura a un objeto de memoria entre las tres entidades.

CLT	Linea	*comp_cn*	NCL	Linea	*comp_np*	PGR	Línea
!c_l_o	*141*	→	*?c_l_o*	*204*			
?n_s_l	*142*	←	*!n_s_l*	*206*			
			!n_a_o	*211*	→	*?n_a_o*	*59*
						op_usa	*60*
			?p_datos	*212*	←	*!p_datos*	*61*
?n_s_l	*142*	←	*!n_s_l*	*213*			

Fig. 5.25 Procesos mutuos.

En la figura 5.25 se presenta el comportamiento del proceso *lee* desde el punto de vista de cada una de las entidades. En la tabla se indican las acciones de la entidad *CLT*, su número de línea correspondiente en el código de la especificación, el sentido de la transferencia en la compuerta por la cual se comunica con el núcleo - *comp_cn* -, en seguida los eventos que realiza el proceso *NCL*, el número de la línea en la especificación, el sentido de la transferencia de la compuerta - *comp_np* -, las acciones del paginador y las líneas de dichas acciones en la especificación. El funcionamiento del sistema para resolver una petición de solución de lectura por un cliente es como sigue. Un cliente genera una petición de lectura enviando el mensaje *c_l_o* mediante la compuerta *comp_cn* (línea 141), síncronamente, el núcleo recibe dicha petición (línea 204), en

seguida el núcleo decide resolver inmediatamente la lectura (línea 206) o pedir los datos del objeto de memoria al paginador (línea 211) dependiendo si tiene o no los datos del objeto de memoria. Si se presenta el primer caso, el cliente recibe la solución de su petición (línea 142) y si sucede el segundo, el paginador recibe la llamada del núcleo (línea 59), realiza operaciones de protección sobre el objeto (línea 60), soluciona la llamada del núcleo (línea 61), el núcleo recibe la solución (línea 212), resuleve la petición del cliente (línea 213) y el cliente recibe la respuesta (línea 142). En este análisis puede observarse que el comportamiento del sistema es equivalente al establecido en los refinamientos previos, la única diferencia que existe es la clasificación de los procesos para cada entidad según sea el caso.

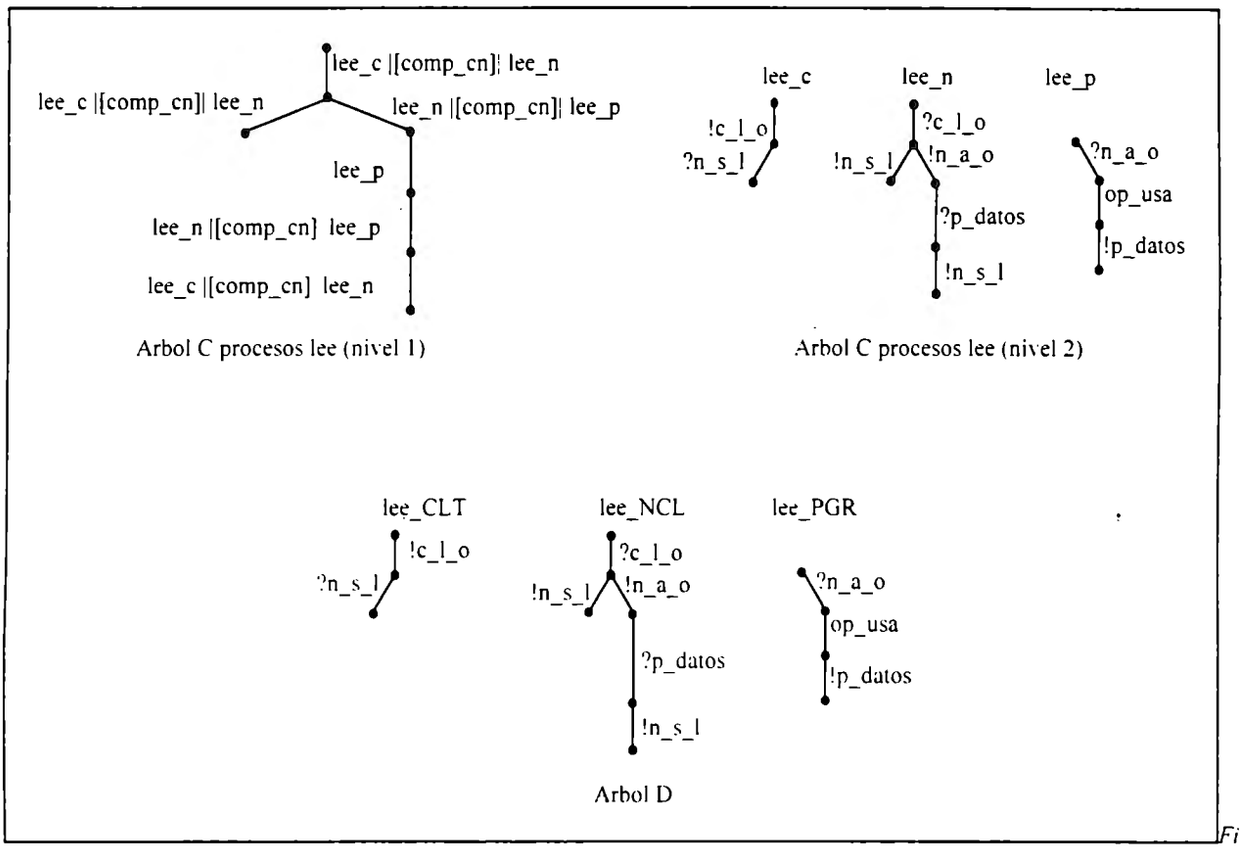
5.6.4 Comprobación de comportamientos equivalentes entre *AME4* y *AME5*

Para comprobar la equivalencia de comportamientos entre las especificaciones *AME4* y *AME5* se desarrolla un análisis similar al efectuado para la comprobación de comportamientos entre las especificaciones *AME3* y *AME4*. En este caso, se utiliza el proceso *lee* del tercer y cuarto refinamientos.

La interpretación de la especificación *AME5* debe llevarse a cabo analizando en paralelo los tres procesos principales e identificando aquellos eventos en los cuales efectúan intercambio de información. La figura 5.26 presenta nuevamente el árbol de comportamiento del proceso *lee* con el propósito de comprobar la equivalencia de comportamientos respecto a la especificación *AME4* y mostrar la ejecución paralela en cada uno de los elementos.

En la figura 5.26 el árbol C corresponde a la conducta que se desarrolla al realizar una operación de lectura en la especificación *AME4*. El árbol de conducta para dicha operación (proceso *lee*) puede ser visto desde dos niveles. El primer nivel es el más abstracto y representa sólo la sincronización entre los procesos *lee_c*, *lee_n* y *lee_p* mediante el operador $|[]|$ utilizando las compuertas correspondientes (líneas 105 a 111 en *AME4*). El segundo nivel representa el detalle de cada uno de los procesos involucrados en el nivel anterior, es decir, las secuencias de eventos específicas que se llevan a cabo en *lee_c*, *lee_n* y *lee_p* (líneas 113 a 135). El árbol D representa

la conducta de la operación de lectura en la especificación *AME5*.



g. 5.26 Arboles de comportamiento equivalentes

En el árbol C de nivel 1, no se observa de manera directa la equivalencia con respecto al árbol D, sin embargo, si se analiza el árbol C de nivel 2, la equivalencia salta a la vista. La diferencia que existe entre ambas especificaciones radica en que en *AME4* sólo hay un proceso *lee* que incluye las diferentes conductas a desarrollar por cada entidad - *lee_c*, *lee_n* y *lee_p*- y; en *AME5* estas conductas se separan en los procesos *lee* de cada elemento del AME -*lee_CLT*, *lee_NCL* y *lee_PGR*-. La explicación de esta conducta equivalente se desarrolla en seguida.

En ambos arboles (C nivel 2 y D), la lectura de un objeto de memoria inicia con la petición de un cliente ejecutando el evento *!c_l_o*; el núcleo recibe la petición representado con el evento *?c_l_o* y para resolverla decide entre dos opciones: responder a la llamada de forma inmediata por medio del evento *!n_s_l*, o pedir al paginador los datos del objeto de memoria mediante el evento *!n_a_o*. En la segunda opción, el paginador recibe la solicitud con el evento *?n_a_o*, realiza las

operaciones necesarias de protección sobre el objeto de memoria llevando a cabo el evento *op_usa* y envía la respuesta al núcleo utilizando el evento *!p_datos*. De forma paralela, el núcleo recibe la respuesta del paginador por medio del evento *?p_datos* y le envía los datos al cliente generando el evento *!n_s_l*. Finalmente, el cliente recibe los datos solicitados del núcleo con el evento *?n_s_l*.

Con las explicaciones dadas a lo largo de los refinamientos se asume que el lector tiene los fundamentos para poder realizar el análisis del comportamiento de toda la especificación por sí mismo, debido a que éste es demasiado extenso para incluirlo en este documento. Para ello se sugiere observar el comportamiento de los mismos consultando dicha especificación en el apéndice **A3**.

En este cuarto refinamiento quedan establecidas de manera separada las tres entidades del AME, así como las diversas definiciones para lograr la comunicación síncrona entre ellas, sin embargo, los tipos de datos no han sido refinados. Estos representan mensajes indivisibles específicos, por ejemplo, el mensaje para que un cliente realice la operación de una lectura está representado por *c_l_o* o la solución a la misma *n_s_l*, y ambos están definidos como un mismo tipo de dato que representa un mensaje intercambiado entre el núcleo y el cliente. En este momento es necesario detallar cada uno de los mensajes en la especificación dividiéndolos en datos que expresen información más precisa. Esta información es necesaria para que el proceso que representa al paginador tenga mayores elementos para llevar a cabo el control y la administración de los objetos de memoria.

5.7 Quinto Refinamiento

5.7.1 Justificación del quinto refinamiento

En la especificación obtenida con el cuarto refinamiento, las entidades principales del AME están separadas y los tipos de datos intercambiados son generales. En esta especificación se detallan los tipos de datos y se definen estructuras de datos específicas para la parametrización de los procesos. Las descripciones de los tipos de datos para la comunicación entre procesos y para las estructuras de control son más complejas en el paginador que en las otras dos entidades. Los principales cambios en la especificación de los procesos consisten en incluir la transferencia de los nuevos tipos de datos derivados como resultado de un análisis profundo de los tipos de datos del refinamiento anterior.

Además, se desarrolla el quinto refinamiento de la especificación considerando las llamadas utilizadas en el sistema operativo Mach para la implantación de un administrador de memoria externo a nivel usuario.

5.7.2 Especificación resultante del quinto refinamiento

La especificación resultante del quinto refinamiento se presenta en el apéndice A4. A continuación se proporciona una explicación del refinamiento realizado para su obtención.

5.7.3 Descripción del quinto refinamiento

La descripción de este refinamiento se divide en tres partes. En la primera, se explica como se obtienen los tipos de datos y las compuertas. En la segunda, se describe la definición de los tipos

de datos. En la última, se expresan los detalles efectuados en los diferentes procesos para lograr la aproximación a la implantación de la MCD en el sistema operativo Mach.

5.7.3.1 Refinamiento de tipos de datos y uso de compuertas

En general, cada tipo de dato de la especificación *AME5* representa un tipo de comunicación, por ejemplo, la comunicación núcleo-paginador está representada por el tipo *com_np_pn* y cada operación dentro de este tipo representa una llamada en específico, por ejemplo, la llamada en donde el núcleo mapea un objeto corresponde a la operación *c_m_o*. Ahora, esta llamada, aproximándola a la implantación para Mach y considerando la información mínima necesaria para que el paginador pueda ejercer control sobre el uso de los objetos de memoria, debe contener ciertos datos fundamentales para este sistema.

Fue estudiado cada mensaje definido en los tipos de datos anteriores; para cada uno de ellos se determinan los elementos requeridos que deben constituirlo para aproximar la estructura del mensaje a la estructura de la llamada real en el núcleo de Mach. El análisis, en la mayoría de los casos, arroja elementos repetitivos, como el nombre del objeto, el puerto del mismo o el puerto de control; por lo que todos estos elementos son clasificados de acuerdo a tipos específicos, por ejemplo, el puerto de control y el puerto del objeto se clasifican dentro de un mismo tipo: *puertos*. De la misma forma se clasifican el resto de los elementos, resultando en 10 tipos de datos, mostrados en la figura 5.27.

Tipo de dato	Descripción
puertos	Contiene todos los elementos llamados en el S.O. Mach "puertos"
nombres	Representa cadenas de caracteres utilizadas en el S.O. Mach para identificar con un nombre específico algún objeto
banderas	Expresa valores booleanos, falso o verdadero
Error	Identifica tipos de errores
Datos	Representa conjuntos de bytes para simbolizar los datos del objeto de memoria
tam_pag	Establece un tamaño de página estándar y está definido como un número natural (4096)
identificador	Establece las características necesarias para asegurar identificadores únicos
direccion	Representa un desplazamiento dentro del objeto de memoria.
permisos	Determina los permisos posibles sobre los objetos de memoria (lectura, escritura o ejecución)
est_copia	Describe los tipos de copia que puede usar un paginador sobre un objeto de memoria

Fig. 5.27 Tipos de datos resultantes del análisis de cada tipo de mensaje relacionado directamente con el paginador.

Los tipos de datos que no se encuentran relacionados exclusivamente con el núcleo y el cliente no fueron detallados, es el caso del tipo de dato *com_cn_nc*.

Además de los tipos de datos anteriores se utilizan el tipo de dato *Byte* para definir posteriormente a *Datos*, el tipo *consis* representa las operaciones de control de integridad de los objetos de memoria, y algunos tipos de datos que representa conjuntos y estructuras de control como: *Conj_ptos, Conj_nom, Conj_obm, Conj_obm_ncl, tupla_obm, tupla_obm_ncl, est_obm, est_obm_ncl* y *datos_obm*. Todos estos son mostrados en la figura 5.28.

Tipo de dato	Descripción
Byte	Describe conjutos de ocho bits utilizando el tipo de dato formal *Octet* tomado de la biblioteca estándar.
consis	Auxiliar en las operaciones de protección de los objetos de memoria.
Conj_ptos	Auxiliar en la validación de puertos únicos.
Conj_nom	Auxiliar en la validación de nombres de objeto únicos.
Conj_obm	Auxiliar en el control de objetos de memoria.
Conj_obm_ncl	Auxiliar en el control de objetos de memoria activos en algún núcleo.
tupla_obm	Representa un elemento de la estructura *est_obm*. Contiene elementos para administrar a los objetos de memoria.
tupla_obm_ncl	Representa un elemento de la estructura *est_obm_ncl*. Contiene elementos para la administración de los objetos de memoria activos en algún núcleo.
est_obm	Define una estructura compuesta de un conjunto de elementos del tipo *tupla_obm*. Contiene operaciones para el control de los objetos de memoria manipulados por el paginador.
datos_obm	Auxiliar en la verificación de los datos recibidos.
est_obm_ncl	Define una estructura compuesta de un conjunto de elementos del tipo *tupla_obm_ncl*. Contiene operaciones para el control de los objetos de memoria manipulados por el paginador activos en algún núcleo.

Fig. 5.28. Tipos de datos auxiliares en el control de los objetos de memoria.

Las compuertas para intercambiar los tipos de datos entre los procesos son tres: *comp_cn, comp_np* y *comp_pc*. Cada una representa un canal de comunicación entre dos entidades, como ya ha sido explicado.

5.7.3.2 Descripción de tipos de datos

Los tipos de datos *puertos, identificador, nombres* y *banderas,* se definen como un

renombramiento [4] de los tipos de datos predefinidos en la biblioteca estándar[20]; los dos primeros, como números naturales, el siguiente, como cadena de caracteres, y el último, como valor booleano. El renombramiento que se realiza se ejemplifica tomando como muestra el tipo de dato *puertos* en la figura 5.29.

```
17 Type puertos is NaturalNumber
18 sorts pto
19 opns
20   pto_SMR, pto_PGR, pto_obm, pto_control          : ->     pto
21 endtype
```

Fig. 5.29 Definición de puertos

En dicha figura, la definición de *puertos* es un renombramiento de *NaturalNumber*, indicando que *puertos* toma todas las características preestablecidas en la biblioteca estándar. La definición de *identificador*, *nombres* y *banderas* se realiza de manera similar a *puertos*, con la diferencia de que en el lugar de *NaturalNumber* en la definición inicial se coloca el nombre del tipo estándar correspondiente.

Los tipos de datos *Byte*, *Conj_ptos* y *Conj_nom* se definen como **actualizaciones** [4]; el primero, al tipo formal *Octet*, también definido en la biblioteca estándar, y los tipos *Conj_ptos* y *Conj_nom* al tipo formal *Set*, que representa conjuntos. Para *Conj_ptos*, los elementos del conjunto son de tipo *pto* y para *Conj_nom*, son de tipo *nom*. Un ejemplo de **actualización** se muestra en la figura 5.30.

```
37 Type Byte is String actualizedby Octet using
38 sortnames
39                  Octet for Element
40                  Bool for FBool
41                  Byte for String
42 opnames          Octet for String
43 endtype
```

Fig. 5.30. Definición de Byte

Octet es un tipo definido con parámetros formales en la biblioteca estándar, cuando se realiza la actualización, el tipo que se indica, en este caso *Byte*, toma el lugar de *Octet*, es decir, se comporta como el parámetro actual. Los parámteros formales de *Octet* son *Element*, *FBool* y

String, los cuales son actualizados con los parámetros actuales *Octet*, *Bool* y *Byte*, respectivamente. *Conj_ptos* y *Conj_nom* son definidos de manera similar con el tipo formal *Set* (definido en la biblioteca estándar).

El tamaño de página *tam_pag*, está definido como una constante, un número natural con valor 4096. Se establece el mismo tamaño de página para todos los objetos de memoria con la finalidad de facilitar la especificación del sistema y con el mismo propósito, se determina el tamaño de los objetos igual a una página.

El tipo de dato para describir los permisos que tiene un usuario sobre un objeto de memoria -permisos- y la estrategia de copia para los objetos -est_copia- son definidos con tres constantes como se muestra en la figura 5.31.

```
100 Type permisos is NaturalNumber
101   opns
102      lectura, escritura, ejecucion          : ->     permisos
103   eqns
104      ofsort Nat
105      succ(0) = lectura;
106      succ(1) = escritura;
107      succ(2) = ejecucion;
108 endtype
109 Type est_copia is NaturalNumber
110   opns
111      copia_inmed, copia_aut_PGR, copia_COW      : ->     est_copia
112   eqns
113      ofsort Nat
114      succ(0) = copia_inmed;
115      succ(1) = copia_aut_PGR;
116      succ(2) = copia_COW;
117 endtype
```

Fig. 5.31 Definición de permisos *y* est_copia

Como se puede apreciar, para *permisos*, 1 (succ(0)) representa la posibilidad de lectura de un objeto de memoria - *lectura* -, 2 (succ(1)) la escritura - *escritura* - y 3 (succ(2)) para tener la autorización de ejecutarlo - *ejecucion* -; y para *est_copia*, 1 (succ(0)) representa la característica del paginador de permitir al núcleo realizar las copias de objetos de manera inmediata - *copia_inmed* -, 2 (succ(1)) limita la posibilidad de copia de los objetos y el núcleo debe solicitar primero la autorización del paginador - *copia_aut_PGR* - y 3 (succ(2)) representa la estrategia de copia sobre escritura - *copia_COW* -.

Se definen además dos estructuras de control y dos conjuntos auxiliares a éstas. Una, para controlar los objetos y otra, para controlar los objetos y núcleos que los comparten. Cada estructura está constituida por un conjunto de elementos **tupla** y cada **tupla** contiene un número de tipos de datos que la constituye. Es decir, el tipo de dato de las estructuras de control está constituido por el conjunto **estructuras tupla**. Por ejemplo, la **tupla** para la estructura de control de objetos de memoria se constituye de cuatro elementos o tipos de datos: puerto del objeto -*pto_obm*-, nombre -*nom_obm*-, puerto de control del núcleo que tiene permiso de escritura sobre el objeto, que llamamos escritor activo -*escritor*- y una bandera para indicar el estado del objeto -*válido*-. Esta bandera contiene el valor de verdadero (*true*) cuando el objeto en memoria no tiene diferencias contra aquel que se encuentra en su archivo y tiene el valor de falso (*false*) para indicar que existen tales diferencias. Para apreciar mejor la definición de las **tuplas** se muestra en la figura 5.32 la definición de *tupla_obm*.

```
110 Type tupla_obm is pto, nom, ban, Boolean
111 sorts
112  opns
113      tupo: pto, nom, pto, ban                                   :-> tupla_obm
114      _eq_: tupla_obm, tupla_obm                                 :-> Bool
115      comp_obm: pto, tupla_obm                                   :-> Bool
116      comp_esc: pto, pto, tupla_obm                              :-> Bool
117      obm_val: pto, tupla_obm                                    :-> Bool
118      act_esc: pto, pto, tupla_obm                               :-> tupla_obm
119      limp_obm: pto, pto, tupla_obm                              :-> tupla_obm
120  eqns
121   forall
122   o, c, p, pp, e, ee: pto, n, nn: nom, v, vv: ban
123   ofsort Bool
124      tupo(p, n, e, v) eq tupo(pp, nn, ee, vv) = (p eq pp) and (n eq nn) and
125                                                 (e eq ee) and (v eq vv);
126      (o eq p) => comp_obm(o, tupo(p, n, e, v)) = true;
127      (o ne p) => comp_obm(o, tupo(p, n, e, v)) = false;
128      (o eq p) and (c eq e) => comp_esc(o, c, tupo(p, n, e, v)) = true;
129      (o ne p) or (c ne e) => comp_esc(o, c, tupo(p, n, e, v)) = false;
130      (o eq p) and (v eq true) => obm_val(o, tupo(p, n, e, v)) = true;
131      (o eq p) and (v eq false) => obm_val(o, tupo(p, n, e, v)) = false;
132 ofsort tupla_obm
133      (o eq p) => act_esc(o, c, tupo(p, n, e, v)) = tupo(p, n, c, false);
134      (o eq p) and (c eq e) => limp_obm(o, c, tupo(p, n, e, v)) = tupo(p, n, e, true);
135 endtype
```

Fig. 5.32. Definición de tupla_obm.

En el tipo *tupla_obm* se definen siete operaciones *tupo, _eq_, comp_obm, comp_esc, obm_val, act_esc, limp_obm* - líneas 113 a 119 - . *tupo* es la operación que expresa la construcción de la tupla. Sus parámetros son los cuatro elementos ya descritos que debe contener la misma - *pto_obm, nom_obm, escritor, válido* - representados por *pto, nom, pto* y *ban*, línea 113. La operación *_eq_* expresa la comparación entre dos **tuplas** y regresa un valor booleano: *true*, cuando las **tuplas** son iguales y *false* cuando no lo son. Las ecuaciones de esta operación están descritas en las líneas 124 y 125 -. Las operaciones de *comp_obm* y *comp_esc* realizan la comparación de uno o dos elementos para verificar si estos se encuentran como integrantes de una **tupla**. *comp_obm* sólo compara el puerto del objeto con una tupla dada y *comp_esc*, compara el puerto del objeto y del escritor (puerto del núcleo escritor). Ambas operaciones regresan un valor booleano, *true* para indicar si los elementos en el parámetro se encuentran en la tupla y *false* para indicar que no lo están. Las ecuaciones de estas operaciones están expresadas en las líneas 126 a la 129 y establecen una implicación. En la operación *comp_obm*, si el parámetro dado es igual al primer elemento de la tupla, entonces, el resultado es verdadero; si es diferente, es falso. En la operación *comp_esc*, si los dos parámetros dados son iguales al primer y al tercer elemento respectivamente, entonces, el resultado de la operación es verdadero; si cualquiera de los dos es diferente, entonces el resultado es falso. La interpretación del resultado de estas operaciones es como sigue. Si *comp_obm* es verdadero, entonces el objeto de memoria está activo en el paginador; si su resultado es falso, entonces, el objeto no está activo. Si *comp_esc* es verdadero, entonces el puerto del núcleo dado es el escritor activo, de lo contrario, no lo es.

La operación *obm_val* se define para obtener el estado de *ban* para un puerto de objeto dado dentro de la tupla. Las ecuaciones están expresadas en las líneas 130 y 131 por una implicación. Si el puerto de objeto dado es igual al primer elemento de la tupla y el valor de la bandera *valido* es *true*, entonces, el resultado de la operación es verdadero -*true*-. Si en la comparación *valido* es igual a *false*, entonces, el resultado de la operación es falso -*false*-.

Las operaciones *act_esc* y *limp_obm* son operaciones definidas para modificar los valores de *escritor* y *valido*, respectivamente. La primera, para actualizar el escritor activo del objeto de memoria y la segunda, para establecer a un objeto de memoria como limpio. *act_esc* -línea 133, tiene como parámetros un puerto de objeto de memoria -*o*- y un puerto de control -*c*-. Si el puerto de objeto dado -*o*- es igual al primer elemento de la tupla -*p*-, entonces, la operación da por

resultado una tupla con los mismos valores en los dos primeros elementos pero con el tercer elemento actualizado con el puerto de control dado -*c*- y con el último elemento actualizado con el valor *false*. *limp_obm* -línea 134- también tiene como parámetros un puerto de objeto de memoria -*o*- y un puerto de control -*c*-. Si el puerto de objeto y el puerto de control son iguales al primer y al tercer elemento respectivamente, entonces, el resultado de la operación es la **tupla** con los mismos valores en los tres primeros elementos pero con el valor de la bandera *valido* actualizado con *true*.

El tipo *tupla_obm_ncl* -líneas 136 a 157- tiene operaciones similares a *tupla_obm* por lo que sólo se explican las diferencias. En la figura 5.33 se muestran las operaciones diferentes de este tipo.

```
141   obten_p: pto, pto, tupla_obm_ncl                :->   permisos
142   obten_est: pto, pto, tupla_obm_ncl              :->   est_copia
143   act_per: pto, pto, permisos, tupla_obm_ncl :->   tupla_obm_ncl

.
.
150 ofsort permisos
151   (co eq c) and (ob eq o)  => obten_p(co, ob, tupon(c, o, p, s)) = p;
152 ofsort est_copia
153   (co eq c) and (ob eq o)  => obten_est(co, ob, tupon(c, o, p, s)) = s;
154 ofsort tupla_obm_ncl
155   (co eq c) and (ob eq o)  => act_per(co, ob, pp, tupon(c, o, p, s)) =
156                                   tupon(c, o, pp, s);
```

Fig. 5.33.Definición de tupla_obm_ncl.

Las operaciones *obten_p* -línea 141 y *obten_est* - línea 142- están definidas para obtener una, los permisos que tiene un núcleo sobre el objeto y la otra, para obtener el tipo de estrategia de copia que realiza el núcleo sobre el objeto. Otra operación diferente que tiene el tipo *tupla_obm_ncl* es *act_per*. Esta permite modificar el valor de los permisos que tiene un núcleo sobre un objeto de memoria dado. Las ecuaciones de las tres operaciones se encuentran expresadas en la especificación en las líneas 151 a la 156.

Los tipos de datos **tupla** se utilizan para la manipulación de los objetos de memoria, como ya fue

explicado. Uniendo un conjunto de elementos **tupla** se obtiene un tipo de dato *estructura* con la finalidad de manejar las **tuplas** como elementos de un conjunto con operaciones definidas. El tipo de dato *estructura* está definido en la especificación en las líneas 158-178. A continuación se muestra este tipo de dato en la figura 5.34.

estructura se define como un tipo de dato formal que puede ser redefinido (**actualizado**) con cualquier otro, en este caso, se utiliza para definir *est_obm* y *est_obm_ncl*; la actualización de la definición se realiza utilizando los parámetros actuales *tupla_obm* y *tupla_obm_ncl*, respectivamente. Los parámetros formales son *tupla* y *estructura* y las operaciones especificadas dentro de este tipo son las siguientes: *vacio* para expresar una *estructura* vacía, *agrega* para adicionar una *tupla* a la estructura, *remueve* para eliminar una *tupla* de la estructura, y *esta_en* para identificar si una *tupla* pertenece a la estructura. Las ecuaciones para estas operaciones están definidas en las líneas 166 a la 177 y se clasifican en dos grupos, las de tipo *estructura* y las de tipo booleano. La función para agregar una *tupla* a la estructura consiste en definir una *x* de tipo *tupla*, la cual se agrega a *est* de tipo estructura y da como resultado una estructura con una *tupla* nueva. Para remover un elemento de la estructura se utiliza la operación *remueve*, la cual toma a la *tupla x* de la estructura y regresa la estructura sin dicha *tupla*, es importante observar que esta operación se define como la remoción de una *tupla* que fue agregada previamente, lo que da como resultado la estructura original, este comportamiento es lógico, ya que para poder remover una *tupla* primero es necesario haberla insertado. La operación *esta_en* regresa un valor de tipo booleano, *true* cuando la **tupla** se encuentra en la estructura y *false* en caso contrario.

```
158 Type estructura is
159     formalsorts tupla
160     sorts estructura
161 opns
162     vacio:                                  :-> estructura
163     agrega: tupla, estructura               :-> estructura
164     remueve: tupla, estructura              :-> estructura
165     esta_en: tupla, estructura              :-> Bool
166 eqns
167     forall x,y: tupla, est: estructura
168 ofsort estructura
169     agrega(x, agrega(x, est))= agrega(x,est);
170     remueve(x, agrega(x, est))= est;
171     not(x eq y) => remueve(x, agrega(y, est))=agrega(y,est);
172 ofsort Bool
173     esta_en(x, vacio)=false;
174     esta_en(x, agrega(x,est))=true;
175     esta_en(x, remueve(x,est))=false;
176     not(x eq y)=> esta_en(x, agrega(y,(remueve(x,est)))=false;
177     not(x eq y)=> esta_en(x, remueve(y,est))=esta_en(x,est);
178 endtype
```

Fig. 5.34. Definición del tipo de dato estructura.

Como se mencionó, los tipos de datos *est_obm* y *est_obm_ncl* se definen como una actualización al tipo *estructura*. Las operaciones definidas en *estructura* son las mismas para las actualizaciones, en las cuales se reemplazan los parámetros formales por los tipos indicados en la redefinición.

5.7.3.2 Descripción de detalles de implantación en la especificación

Los cambios efectuados en este refinamiento, referentes al detalle de los procesos, se centran en la verificación de la actualización de los parámetros que se pasan entre las diferentes entidades del AME. Esto se realiza con la inclusión de guardias en todos los procesos en los que se reciben datos generados por la comunicación de los elementos CLT, NCL y PGR. Dichos guardias tienen

el objeto de identificar los parámetros actuales en tránsito para generar una secuencia de eventos que cumpla con las características pedidas al sistema. La especificación que se obtiene en este refinamiento es bastante extensa y su explicación es compleja, sin embargo, a pesar de su extensión, el comportamiento del sistema es el mismo, únicamente se detallan los procesos con la finalidad de aproximarlos al funcionamiento de Mach.

CAPITULO 6

CONCLUSIONES

6. Conclusiones

Se ha aprendido que para diseñar un sistema distribuido fiable se requiere de un estudio y un análisis profundo de las necesidades del sistema a desarrollar. Esto implica adquirir una sólida base teórica que se logra con una vasta investigación documental.

El diseño de sistemas distribuidos eficientes y confiables no es una tarea fácil. Para lograrlo es necesario utilizar técnicas de diseño y algoritmos que ofrezcan: protección de la información, administración óptima de los recursos, transparencia de acceso, migración, portabilidad, modularidad y fácil mantenimiento.

El conocimiento de los conceptos fundamentales de sistemas operativos, sistemas distribuidos, así como de técnicas de manejo y administración de memoria, real, virtual y compartida distribuida, fortalecen la comprensión del modelo desarrollado en este trabajo.

Una vez conocido el funcionamiento del sistema a desarrollar, la acción siguiente radica en la elección de la técnica a utilizar para el diseño del mismo. En este caso, se eligió la técnica de descripción formal, con el objeto de llevar a cabo la construcción de una memoria compartida distribuida a programarse sobre el sistema operativo Mach.

Antes de diseñar el sistema usando la técnica formal, se describe de manera informal el comportamiento del mismo, con el objeto de aclarar cualquier duda referente al funcionamiento de alguna de las partes que lo constituyen. Se presentan arboles de comportamiento que expresan paso a paso las diferentes situaciones que pueden presentarse durante el procesamiento del sistema y una vez establecidos todos los arboles necesarios para cada una de las diferentes operaciones del sistema. entonces se procede con la utilización de la técnica formal.

Adquirir experiencia en el uso de las técnicas formales exige un gran esfuerzo por parte del diseñador de un sistema. debido a que éste debe tener amplio conocimiento del problema a resolver. así como de una profunda comprensión de dichas técnicas.

La técnica utilizada en este proyecto, LOTOS, está basada en el álgebra de procesos. Esto es. se expresa el comportamiento de un sistema como un conjunto de procesos. los cuales están constituidos por secuencias de eventos que cumplen las acciones del sistema. Se exige el dominio de los operadores básicos de LOTOS para poder llevar a cabo la especificación del sistema de memoria compartida distribuida.

Una memoria compartida distribuida es implantada en el sistema operativo Mach utilizando un administrador de memoria externo a nivel usuario, constituido por tres elementos principales: el cliente. el núcleo y el paginador. La expresión del comportamiento de estos elementos en LOTOS es bastante compleja y se deben tomar en cuenta todas las situaciones posibles que puedan dar origen a algún conflicto. con la finalidad de ofrecer seguridad en el funcionamiento del sistema antes de ser implantado. Los operadores y constructores del lenguaje LOTOS ayudan a la descripción del sistema. es decir. su especificación.

Una vez establecida la especificación inicial del sistema. esto es, la que describe su comportamiento de la forma más general, se procede a desarrollar refinamientos de la misma con los que se van detallando paso a paso y de manera menos abstracta, las descripciones de los distintos procesos representativos del sistema.

Para demostrar que el refinamiento de una especificación se efectúa de manera correcta es necesario comprobarlo. Sin embargo, la comprobación de un refinamiento en este documento se lleva a cabo de manera semiformal utilizando los arboles de comportamiento que expresan dos procesos *equivalentes* , un proceso dado en una especificación y ese mismo proceso después de aplicar el refinamiento (proceso dado de la especificación siguiente), y funciones de renombramiento que se aplican al proceso *refinado* que dan como resultado el proceso original. Aunque en este trabajo no se desarrolla una verificación formal se realizan refinamientos que dan un sentido original al mismo, lo cual, tiene un gran valor desde el punto de vista de las técnicas formales.

La verificación del sistema AME descrito en las diferentes especificaciones se lleva a cabo a lápiz y papel, lo que implicó gran dedicación. Actualmente, existen herramientas de software para realizar esta función, sin embargo, dada la complejidad de este proyecto dicha función se limitó a pruebas manuales.

Con la experiencia adquirida en este proyecto, se establecen las bases para ejecutar verificaciones de especificaciones con la ayuda de herramientas de software. Tomando en consideración esto, se encuentra en proceso de desarrollo una investigación que tiene como finalidad la explotación de estas herramientas.

La utilización de tipos de datos en LOTOS para la transferencia de información entre procesos, a diferencia de los lenguajes tradicionales de programación, requieren que el diseñador exprese el tipo de dato como un conjunto de operaciones y ecuaciones que representen de manera formal las acciones deseadas para el mismo. Existe una biblioteca estándar en LOTOS que cuenta con los tipos de datos básicos a partir de los que se diseñan nuevos tipos.

La aplicación de las técnicas formales en el diseño de sistemas aseguran el buen funcionamiento de los mismos cuando son implantados, reducen su costo real, evitan ambigüedades y facilitan el mantenimiento, entre otras ventajas. Sin embargo, su uso no es trivial y requieren que el diseñador tenga una profunda base matemática.

A lo largo de este trabajo se adquirió experiencia en el diseño de sistemas distribuidos, también se aprendió a utilizar el lenguaje de especificación formal LOTOS como una técnica de diseño.

Bibliografía

[1] Tanenbaum, Andrew Sistemas Operativos Modernos 2a. Ed. Prentice Hall Hispanoamericana, 1992.

[2] Deitel, H.M. Sistemas Operativos 2a. Ed. Addison Wesley, U.S.A. 1993.

[3] Coulouris. G.; Dollimore, J.; Kindberg. T. Distributed Systems, Concepts and Design. 2a. Ed. Addison-Wesley, U.S.A., 1994.

[4] Turner Kenneth J. Using formal techniques an introduction: to ESTELLE, LOTOS, SDL Ed. Wiley Publishers, U.S.A., 1993.

[5] Butler. R.; Johnson Formal methods for life-critical software NASA Langley Research Center. Hampton, Virginia 1993.

[6] Wing, J. "A Specifier's Introduction to Formal Methods" Computer, 1990 p. 8 - 20.

[7] Azcorra. A.; Quemada, J.; Pavón, S. "An Introduction to LOTOS" Depto. Ingeniería de Sistemas Telemáticos, Universidad de Madrid, 1993.
M: http://wwwtios.cs.utwente.nl/lotos/lotos.html.

[8] Logrippo. L.; Faci, M.; Haj-Hussein, M. "An Introduction to LOTOS: Learning by Examples" University of Ottawa, Computer Science Department.

[9] Ford. Neville; Ford. Judith Introducing Formal Methods 1a. Ed. Allis Horwood Limited. 1993.

[10] Nutt, Gary J. Centralized and Distributed Operating Systems 1a. Ed Prentice-Hall 1992.

[11] Banâtre (J.P) et le Métayer "Programing by multiset transformation" Com. of ACM Vol. 36, No. 1 Enero 1993 p 98 - 111.

[12] Carriero (N) et Gelernter (D) " Linda in context" Com. of ACM Vol 32, No. 4 Abril 1989 p 444 - 458.

[13] Tanenbaum, Andrew Distributed Operating Systems Prentice Hall Hispanoamericana, 1995.

[14] Bustard, David W. and Winstanley .Adam C. "Making Changes to formal Specifications : Requirements and an example ".IEEE Transactions On Software Engineering, Vol. 20. No. 8. August 1994. pp 562 - 568.

[15] Boykin. Kirschen. Langerman, LoVerso Programming Under Mach 1st. Ed. Addison Wesley,1993.

[16] "Design.Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach".

[17] Liang Wen-Yew "ADSMITH : "A Structured-Based Heterogeneous Distributed Shared Memory on PVM". Institute of Computer Science National Tsing Hua University, June 1994.

[18] Boumezbeur. Rezki; Loggripo Luigi "Specificaction and Validation of Telephone Systems in LOTOS" . Computer Science Department, University of Ottawa, May 1991.

[19] Faci Mohammed; Loggripo Luigi; Stepien Bernard " Formal Specification of Telephone Systems in LOTOS : The Constraint-Oriented Stye Approach". Computer Science Department, University of Ottawa. June 1989.

[20] ISO 8807:1989 Information processing systems- Open Systems Interconnection - LOTOS- A formal description technique based on the temporal ordering of observational behaviour.

[21] Ehrig. H. and Mahr. B. Fundamentals of Algebraic Specification 1, EATCS Monograhps on Theoretical Computer Science. 6, Springler- Verlag. Berlin. 1985.

[22] Milner. A.J.R.G. Communication and Concurrency. Addison-Wesley . Reading Massachusetts. 1989.

[23] Hoare. C.A.R Communicating Sequential Processes. Prentice-Hall. International. Englewood Cliffs. New Jersey. 1985.

[24] Ashkar. Pierre. Symbolic Execution of LOTOS Specifications, Computer Science Department. University of Ottawa. Ontario. Canada. January 1992.

[25] De Nicola Rocco, Inverardi Paola and Nesi Mónica. "Equational Reasoning about LOTOS Specifications A Rewriting Approach" , Università La Sapienza. Roma; IEI-CNR, Pisa; University of Cambridge, UK. 1991.

[26] Loepere Keith, "Mach 3 Kernel Principles", Open Software Foundation and Carnegie Mellon University, NORMA-MK12, July 15, 1992.

[27]Baron Robert, Black D., *et al*, "Mach Kernel Interface Manual", Department of Computer Science, Carnegie Mellon University, August 23, 1990.

[28]Draves, *et al*, "MIG The Mach Interface Generator", Department of Computer Science, Carnegie Mellon University.

[29]J.M Spivey, Introducing Z: A Specification Language and its Formal Semantics, Cambridge Univeristy Press,1988.

[30] J.-R. Abrial, The B-Book, Cambridge University Press, 1996. ISBN 0-521-49619-5.

[31] J. Misra and K. M. Chandy, Parallel Program Design: A Foundation, Addison-Wesley, 1988.

[32] Silberschatz A. *et al* Sistemas Operativos: Conceptos fundamentales 3a. Ed Addison-Wesley, 1994.

[33] Flores Pérez Gabriela "Especificación formal de una memoria compartida distribuida: Algoritmos de coherencia", Tesis de Maestria en Ciencias de la Computación, ITESM-CEM, 1996.

[34] Ross Kenneth A. y Wright R.B. Matemáticas Discretas 2a. Ed Prentice Hall, México, 1990.

Apéndices

Esta parte del documento contiene las diferentes especificaciones correspondientes al sistema AME.

El apéndice A1 describe la especificación obtenida con el segundo refinamiento, AME3.

El apéndice A2 describe la especificación obtenida con el tercer refinamiento, AME4.

El apéndice A3 describe la especificación obtenida con el cuarto refinamiento, AME5.

El apéndice A4 describe la especificación obtenida con el quinto refinamiento, AME6.

```
1   specification AME3[reg_pag, not_nuc, p_da_o, p_datos, op_usa, pide_pto, c_l_o,
2                     c_e_o, c_m_o, n_s_l, n_s_e, n_s_m, asigna_pto, pide_obm,
3                     asig_obm, n_i_o, n_a_o, n_e_o, n_t_o, t, op_ini]:noexit :=
4   behaviour
5         registro[reg_pag, not_nuc]
6         >>
7         (
8          pedir_pto[pide_pto, asigna_pto]
9          |||
10        pedir_obm[pide_obm, asig_obm, c_l_o, c_m_o, c_e_o, n_s_l, n_s_e, n_s_m, n_i_o,
11  n_a_o, n_e_o, n_t_o, p_da_o, p_datos]
12  where
13        process registro[reg_pag, not_nuc]: exit :=
14              reg_pag; not_nuc; exit
15        endproc (* registro *)
16        process pedir_pto [pide_pto, asigna_pto] : noexit :=
17              pide_pto; asigna_pto; stop
18              |||
19              i; pedir_pto[pide_pto, asigna_pto]
20        endproc (* pedir_pto *)
21        process pedir_obm[pide_obm, asig_obm, c_l_o, c_m_o, c_e_o, n_s_l,
22            n_s_e, n_s_m, n_i_o, n_a_o, n_e_o, n_t_o, p_da_o, p_datos] : noexit :=
23            hide op_ini in
24            pide_obm; asigna_obm; activa_obm[c_m_o, n_i_o, op_ini, p_da_o, n_s_m]
25            |||
26            i; pedir_obm[pide_obm, asig_obm, c_l_o, c_m_o, c_e_o, n_s_l, n_s_e, n_s_m,
27                         n_i_o, n_a_o, n_e_o, n_t_o, p_da_o, p_datos]
28             where
29             process activa_obm[c_m_o, n_i_o, op_ini, p_da_o, n_s_m] : exit :=
30                 hide t in
31                 c_m_o; n_i_o; op_ini; p_da_o; n_s_m;
32                 usa_obm[c_l_o, n_s_l, n_a_o, p_datos, c_e_o, n_s_e, n_e_o, c_m_o,
33                         n_s_m, n_t_o, t]
34              where
35              process usa_obm[c_l_o, n_s_l, n_a_o, p_datos, c_e_o, n_s_e, n_e_o,
36                              c_m_o, n_s_m, n_t_o, t] : noexit :=
37               hide op_usa in
38               (
39               lee [c_l_o, n_s_l, n_a_o, p_datos, op_usa]
40               []
41               escribe [c_e_o, n_s_e, n_e_o, op_usa, n_a_o, p_datos]
42               []
43               mapea [c_m_o, n_s_m, n_a_o, p_datos, op_usa]
44               []
45               termina [n_t_o, op_usa]
46               )        [> t; stop
47               where
```

```
48              process lee [c_l_o, n_s_l, n_a_o, p_datos, op_usa] :exit :=
49                      c_l_o; (
50                              n_s_l; exit
51                              []
52                              n_a_o; op_usa; p_datos; n_s_l; exit
53              endproc (* lee *)
54              process escribe [c_e_o, n_s_e, op_usa, n_e_o, n_a_o, p_datos, p_sol_esc]
55                      :exit :=
56                      c_e_o;(
57                              solucion_esc[n_s_e, op_usa n_e_o, n_a_o, p_datos])
58                    where
59                      process solucion_esc [n_s_e, op_usa, n_e_o, n_a_o, p_datos] :exit:=
60                              n_s_e; exit
61                              []
62                              n_e_o; op_usa; n_s_e; exit
63                              []
64                              n_a_o; op_usa; p_datos;
65                              solucion_esc [n_s_e, op_usa, n_e_o, n_a_o, p_datos]
66                      endproc (* solucion_esc *)
67              endproc (* escribe *)
68              process mapea [c_m_o, n_s_m, n_a_o, op_usa, p_datos] :exit :=
69                      c_m_o; (
70                              n_s_m; exit
71                              []
72                              n_m_o; op_usa; p_datos; n_s_m; exit )
73              endproc (* mapea *)
74              process termina [n_t_o, op_usa] :exit :=
75                      i; n_t_o; op_usa; exit
76              endproc (* termina *)
77             endproc (* usa_obm *)
78          endproc (* activa_obm *)
79          endproc (* pedir_obm *)
80  endspec( *AME3*)
```

```
1   specification AME4[comp_cn, comp_np, comp_pc]: noexit :=
2    Type com_cn_nc is
3     sorts      com_cn_nc
4     opns       c_l_o, c_e_o, c_m_o, n_s_l, n_s_e, n_s_m, pide_pto,
5                asigna_pto : ->                    com_cn_nc
6    endtype (* com_cn_nc *)
7    Type com_np_pn is
8     sorts      com_np_pn
9     opns       n_i_o, n_a_o, n_e_o, n_t_o, reg_pag, p_da_o
10               p_datos, p_e_o, not_nuc : ->              com_np_pn
11   endtype (* com_np_pn *)
12   Type com_pc_cp is
13    sorts      com_pc_cp
14    opns       pide_obm, asigna_obm        : ->          com_pc_cp
15   endtype (* com_pc_cp *)
16  behaviour
17   registro[comp_np]
18    >>
19    (
20        pedir_pto[comp_cn]
21     |||
22        pedir_obm[comp_pc, comp_np, comp_cn]
23    )
24  where
25        process registro[comp_np]: exit :=
26                comp_np! reg_pag; comp_np? not_nuc': com_np_pn;
27                [not_nuc'=not_nuc]-> exit
28                 ||
29                 comp_np? reg_pag' : com_np_pn;
30                [reg_pag'=reg_pag]-> comp_np! not_nuc; exit
31        endproc (* registro *)
32        process pedir_pto[comp_cn]: noexit :=
33                (
34                        comp_cn! pide_pto; comp_cn ? asigna_pto': com_cn_nc; stop
35                        ||
36                        comp_cn? pide_pto': com_cn_nc;
37                        [pide_pto' = pide_pto]-> comp_cn! asigna_pto; stop
38                )
39                 |||
40                i; pedir_pto[comp_cn]
41         endproc (* pedir_pto *)
42        process pedir_obm[comp_cn, comp_np, comp_pc] : noexit :=
43        hide op_ini in
44                (
45                pedir_obm_c[comp_pc] | [comp_pc]| pedir_obm_p[comp_pc]
46                 >>
```

```
47              activa_obm[comp_np, comp_cn, op_ini]
48              )
49              ||| i; pedir_obm[comp_cn, comp_np, comp_pc]
50          where
51              process pedir_obm_c [comp_pc]: exit :=
52                      comp_pc! pide_obm;
53                      comp_pc?asigna_obm': com_pc_cp;[asigna_obm'= asigna_obm]->
54                      exit
55              endproc (*pedir_obm_c*)
56              process pedir_obm_p [comp_pc]: exit :=
57                      comp_pc? pide_obm': com_pc_cp;
58                      [pide_obm'=pide_obm]-> comp_pc! asigna_obm;
59                      exit
60              endproc (*pedir_obm_p*)
61              process activa_obm [comp_np, comp_cn, op_ini]: exit :=
62                hide t in
63                      activa_obm_c [comp_cn] | [comp_cn]|
64                              (
65                                activa_obm_n [comp_cn, comp_np]
66                                | [comp_np]|
67                                activa_obm_p [comp_np]
68                              )
69                              >>
70                              usa_obm [comp_cn, comp_np, t]
71              where
72                      process activa_obm_c [comp_cn] : exit :=
73                              comp_cn! c_m_o;
74                              comp_cn? n_s_m': com_cn_nc;
75                              exit
76                      endproc (*activa_obm_c*)
77                      process activa_obm_n [comp_cn, comp_np] : exit :=
78                              comp_cn?c_m_o': com_cn_nc;
79                              [c_m_o'= c_m_o]->
80                              comp_np! n_i_o;
81                              comp_np? p_da_o': com_np_pn;
82                              [p_da_o'= p_da_o]->
83                              comp_cn! n_s_m;
84                              exit
85                      endproc (*activa_obm_n*)
86                      process activa_obm_p [comp_np] : exit :=
87                              comp_np? n_i_o': com_np_pn;
88                              [n_i_o'= n_i_o]->
89                              op_ini;
90                              comp_np! p_da_o
91                              exit
92                      endproc (*activa_obm_p*)
```

```
93      process usa_obm[comp_cn, comp_np, t] : noexit :=
94          hide op_usa in
95          (
96              lee [comp_cn, comp_np, op_usa]
97              []
98              escribe [comp_cn, comp_np, op_usa]
99              []
100             mapea [comp_cn, comp_np, op_usa]
101             []
102             termina [comp_cn, comp_np, op_usa]
103         ) [> t; stop
104     where
105         process lee[comp_cn, comp_np, op_usa] :exit :=
106             lee_c[comp_cn]
107             |[comp_cn]|
108             (
109                 lee_n[comp_cn, comp_np]
110                 |[comp_np]|
111                 lee_p[comp_np, op_usa])
112         where
113             process lee_c[comp_cn ]:exit:=
114                 comp_cn!c_l_o;
115                 comp_cn?n_s_l':com_cn_nc; [n_s_l'= n_s_l] ->
116                 exit
117             endproc (* lee_c *)
118             process lee_n[comp_cn, comp_np]:exit:=
119                 comp_cn? c_l_o':com_cn_nc;
120                     [c_l_o'=c_l_o] ->
121                     (comp_cn! n_s_l; exit)
122                 []
123                     (comp_np! n_a_o;
124                     comp_np? p_datos':com_np_np;
125                     [p_datos'=p_datos] ->
126                     comp_cn! n_s_l;
127                     exit)
128             endproc (* lee_n *)
129             process lee_p[comp_np, op_usa]:exit:=
130                 comp_np? n_a_o':com_np_pn;
131                 [n_a_o'=n_a_o] ->
132                 op_usa;
133                 comp_np! p_datos;
134                 exit
135             endproc (* lee_p *)
136         endproc (* lee *)
137         process escribe [comp_cn, comp_np, op_usa ]:exit:=
138             escribe_c [comp_cn]
```

```
139                 |[ comp_cn]|
140                         ( escribe_n [comp_cn, comp_np]
141                           |[comp_np]|
142                           escribe_p [comp_np, op_usa]
143                         )
144         where
145                 process escribe_c[comp_cn]:exit:=
146                         comp_cn! c_e_o;
147                         comp_cn? n_s_e':com_cn_nc; [n_s_e'= n_s_e] ->
148                         exit
149                 endproc (* escribe_c *)
150                 process escribe_n[comp_cn, comp_np]:exit:=
151                         comp_cn? c_e_o':com_cn_nc;
152                         [c_e_o'=c_e_o] ->
153                         solucion_esc_n[comp_cn, comp_np ]
154                 where
155                         process solucion_esc_n[comp_cn, comp_np]:exit:=
156                                 comp_cn! n_s_e;exit
157                                 []
158                                 comp_np! n_e_o;
159                                 comp_cn! n_s_e;exit
160                                 []
161                                 comp_np! n_a_o;
162                                 comp_np? p_datos':com_np_pn;
163                                 [p_datos'= p_datos]->
164                                 solucion_esc[comp_cn, comp_np]
165                         endproc (* solucion_esc *)
166                 endproc (* escribe_n *)
167                 process escribe_p[comp_np, op_usa]:exit:=
168                                 comp_np? n_e_o':com_np_pn;
169                                 [n_e_o'=n_e_o] ->
170                                 op_usa;exit
171                         []
172                                 comp_np? n_a_o':com_np_pn;
173                                 [n_a_o'=n_a_o] ->
174                                 op_usa;
175                                 comp_np! p_datos;
176                                 exit
177                 endproc (* escribe_p *)
178         endproc (* escribe *)
179         process mapea [comp_cn, comp_np, op_usa]:exit:=
180                 mapea_c[comp_cn]
181                 |[comp_cn]|
182                 (   mapea_n [comp_cn, comp_np]
183                     |[ comp_np]|
184                    mapea_p[comp_np, op_usa]
```

```
185                         )
186                 where
187                         process mapea_c[comp_cn]:exit:=
188                                 comp_cn!c_m_o;
189                                 comp_cn?n_s_m':com_cn_nc; [n_s_m'= n_s_m] ->
190                                 exit
191                         endproc (* mapea_c *)
192                         process mapea_n[comp_cn, comp_np]:exit:=
193                                 comp_cn?c_m_o':com_cn_nc;
194                                 [c_m_o'=c_m_o] ->
195                                         ( comp_cn! n_s_m; exit)
196                                   []
197                                         (comp_np! n_m_o;
198                                         comp_np? p_datos':com_np_np;
199                                         [p_datos'=p_datos] ->
200                                         comp_cn! n_s_m;
201                                         exit)
202                         endproc (* mapea_n *)
203                         process mapea_p[comp_np, op_usa]:exit:=
204                                 comp_np? n_m_o':com_np_pn;
205                                 [n_m_o'=n_m_o] ->
206                                 op_usa;
207                                 comp_np! p_datos;
208                                 exit
209                         endproc (* mapea_p *)
210                 endproc (* mapea *)
211                 process termina[comp_np, op_usa]: exit:=
212                         i;
213                         (
214                                 comp_np! n_t_o; exit
215                                 |[comp_np]|
216                                 comp_np? n_t_o': com_np_pn; op_usa; exit
217                         )
218                 endproc (* termina *)
219             endproc (* usa_obm *)
220         endproc (* activa_obm *)
221     endproc (* pedir_obm *)
222 endspec (* AME4 *)
```

```
1   specification AME5[comp_cn, comp_np, comp_pc]: noexit :=
2    Type com_cn_nc is
3     sorts      com_cn_nc
4     opns       c_l_o, c_e_o, c_m_o, n_s_l, n_s_e, n_s_m, pide_pto,
5                asigna_pto  : ->            com_cn_nc
6    endtype (* com_cn_nc *)
7    Type com_np_pn is
8     sorts      com_np_pn
9     opns       n_i_o, n_a_o, n_e_o, n_t_o, reg_pag, p_da_o
10               p_datos, op_usa, p_e_o, not_nuc : ->      com_np_pn
11   endtype (* com_np_pn *)
12  Type com_pc_cp is
13    sorts  com_pc_cp
14    opns pide_obm, asigna_obm              : ->   com_pc_cp
15   endtype (* com_pc_cp *)
16  behaviour
17    PGR[comp_np, comp_pc]
18    |||
19    NCL[comp_cn, comp_np]
20    |||
21    CLT[comp_cn, comp_pc]
22  where
23    process PGR[comp_np, comp_pc] : noexit :=
24        registro_PGR[comp_np]
25        >>
26        pedir_obm_PGR[comp_pc]
27    where
28      process registro_PGR[comp_np] : exit :=
29              comp_np! reg_pag;
30              comp_np? not_nuc': com_np_pn; [not_nuc'=not_nuc]-> exit
31      endproc (* registro_PGR*)
32      process pedir_obm_PGR[comp_pc]: noexit :=
33            hide op_ini  in
34              comp_pc? pide_obm': com_pc_cp; [pide_obm'= pide_obm]->
35              comp_pc! asigna_obm; activa_obm_PGR[comp_np, op_ini]
36               |||
37              i; pedir_obm_PGR[comp_pc]
38      where
39        process activa_obm_PGR[comp_np, op_ini]: exit :=
40            hide t in
41              comp_np? n_i_o': com_np_pn; [n_i_o'= n_i_o]->
42              op_ini;
43              comp_np! p_da_o; usa_obm_PGR[comp_np, t]
44        where
45              process usa_obm_PGR[comp_np, t]: noexit :=
46              hide op_usa in
47              (
```

```
48          lee_PGR[comp_np, op_usa]
49          []
50          escribe_PGR[comp_np, op_usa]
51          []
52          mapea_PGR[comp_np, op_usa]
53          []
54          termina_PGR[comp_np, op_usa]
55          )
56          [> t; stop
57          where
58            process lee_PGR[comp_np, op_usa] :exit :=
59                  comp_np? n_a_o': comp_np_pn; [n_a_o' = n_a_o]->
60                  op_usa;
61                  comp_np! p_datos;
62                  exit
63          endproc (* lee_PGR*)
64          process escribe_PGR[comp_np, op_usa] :exit :=
65                  (
66                  comp_np? n_e_o': comp_np_pn; [n_e_o' = n_e_o]->
67                  op_usa; exit
68                  )
69                  []
70                  (
71                  comp_np? n_a_o': comp_np_pn; [n_a_o' = n_a_o]->
72                  op_usa;
73                  comp_np! p_datos;
74                  exit
75                  )
76          endproc (* escribe_PGR*)
77          process mapea_PGR[comp_np, op_usa] :exit :=
78                  (
79                  comp_np? n_a_o: com_np_pn;
80                  op_usa;
81                  comp_np! p_datos;
82                  exit
83                  )
84          endproc (* mapea *)
85          process termina_PGR[comp_np, op_usa] :exit :=
86                  (
87                  comp_np? n_t_o: com_np_pn;
88                  op_usa;
89                  exit
90                  )
91          endproc (* termina_PGR *)
92        endproc (* usa_obm_PGR*)
93      endproc (* activa_obm_PGR*)
94    endproc (* pedir_obm_PGR*)
```

```
95   endproc (* PGR*)
96  process CLT[comp_cn, comp_pc] : noexit :=
97          pedir_pto_CLT[comp_cn]
98          |||
99          pedir_obm_CLT[comp_pc, comp_cn]
100       where
101        process pedir_pto_CLT[comp_cn] : noexit :=
102                (
103                comp_cn! pide_pto;
104                comp_cn? asigna_pto': com_cn_nc; stop
105                )
106                |||
107                i; pedir_pto_CLT[comp_cn]
108        endproc (* pedir_pto_CLT*)
109        process pedir_obm_CLT[comp_pc]: noexit :=
110                (pide_o_c[comp_pc]  >> activa_obm_CLT[comp_cn]
111                )
112                |||
113                i: pedir_obm_CLT[comp_pc]
114        where
115                process pide_o_c[comp_pc] : exit :=
116                        comp_pc! pide_obm;
117                        comp_pc? asigna_obm': com_pc_cp; [asigna_obm'=asigna_obm]->
118                        exit
119                endproc
120                process activa_obm_CLT[comp_cn]: exit :=
121                        hide t  in
122                        act_o_c [comp_cn] >>
123                        usa_obm_CLT[comp_cn, t]
124                    where
125                        process act_o_c [comp_cn]: exit :=
126                                comp_pc! c_m_o; comp_pc? n_s_m': com_cn_nc;
127                        [n_s_m'= n_s_m]->
128                                exit
129                        endproc
130                        process usa_obm_CLT[comp_cn, t]: noexit :=
131                                (
132                                lee_CLT[comp_cn]
133                                []
134                                escribe_CLT[comp_cn]
135                                []
136                                mapea_CLT[comp_cn]
137                                )
138                                [> t; stop
139                        where
140                        process lee_CLT[comp_cn] :exit :=
141                                comp_cn! c_l_o;
```

```
142                         comp_cn? n_s_l': com_cn_nc;
143                     [n_s_l'= n_s_l]->exit
144                     endproc (* lee_CLT*)
145                     process escribe_CLT[comp_cn] :exit :=
146                         comp_cn! c_e_o; comp_cn ? n_s_e': com_cn_nc ;
147                         [n_s_e'= n_s_e] -> exit
148                     endproc (* escribe_CLT*)
149                     process mapea_CLT[comp_cn] :exit :=
150                         comp_cn! c_m_o;
151                         comp_cn? n_s_m': com_cn_nc; [n_s_m'= n_s_m] ->
152                         exit
153                     endproc (* mapea_CLT *)
154                     endproc (* usa_obm_CLT*)
155                 endproc (* activa_obm_CLT*)
156             endproc (* pedir_obm_CLT*)
157 endproc (* CLT*)
158 process NCL[comp_cn, comp_np] : noexit :=
159         registro_NCL[comp_np]
160         >>
161         (
162             pedir_pto_NCL[comp_cn]
163             |||
164             activa_obm_NCL[comp_cn, comp_np]
165         )
166 where
167         process registro_NCL[comp_np] : exit :=
168                 comp_np? reg_pag': com_np_pn; [reg_pag'= reg_pag] ->
169                 comp_np! not_nuc; exit
170         endproc (* registro_NCL*)
171         process pedir_pto_NCL[comp_cn] : noexit :=
172                 (
173                 comp_cn? pide_pto': com_cn_nc; [pide_pto'= pide_pto] ->
174                 comp_cn! asigna_pto; stop
175                 )
176                 |||
177                 i; pedir_pto_NCL[comp_cn]
178         endproc (* pedir_pto_NCL*)
179         process activa_obm_NCL[comp_cn, comp_np]: exit :=
180         hide t in
181                 usa_o_n[comp_cn, comp_np] >>
182                 usa_obm_NCL[comp_cn, comp_np, t]
183         where
184                 process usa_o_n[comp_cn, comp_np, t]: exit :=
185                         comp_cn? c_m_o': com_cn_nc; [c_m_o'= c_m_o] ->
186                         comp_np! n_i_o;
187                         comp_np? p_da_o': com_np_pn; [p_da_o'= p_da_o] ->
```

```
188                 comp_cn! n_s_m;
189                 exit
190         endproc
191         process usa_obm_NCL[comp_cn, comp_np, t]: noexit :=
192                 (
193                 lee_NCL[comp_cn, comp_np]
194                 []
195                 escribe_NCL[comp_cn, comp_np]
196                 []
197                 mapea_NCL[comp_cn, comp_np]
198                 []
199                 termina_NCL[comp_cn, comp_np]
200                 )
201                 [> t; stop
202         where
203                 process lee_NCL[comp_cn, comp_np] :exit :=
204                         comp_cn? c_l_o': comp_cn_nc: [c_l_o'= c_l_o] ->
205                                 (
206                                 comp_cn! n_s_l;
207                                 exit
208                                 )
209                                 []
210                                 (
211                                 comp_np! n_a_o;
212                                 comp_np? p_datos': com_np_pn;
213                                 [p_datos'= p_datos] -> comp_cn! n_s_l;
214                                 exit
215                                 )
216                 endproc (* lee_NCL*)
217                 process escribe_NCL[comp_cn, comp_np,t] :exit :=
218                         (
219                         comp_cn? c_e_o': com_cn_nc: [c_e_o'= c_e_o] ->
220                         solucion_esc_NCL[comp_cn, comp_np]
221                         )
222                 where
223                         process solucion_esc[comp_cn, comp_np] : exit :=
224                                 comp_cn! n_s_e: exit
225                             []
226                                 (
227                                 comp_np! n_e_o;
228                                 comp_cn! n_s_e;
229                                 exit
230                                 )
231                             []
232                                 (
233                                 comp_np! n_a_o;
```

```
234                                 comp_np? p_datos': com_np_pn;
235                                 [p_datos'= p_datos] ->
236                                 solucion_esc_NCL[comp_cn, comp_np]
237                                 )
238                         endproc (* solucion_esc_NCL *)
239                 endproc (* escribe_NCL*)
240                 process mapea_NCL[comp_cn, comp_np] :exit :=
241                         comp_cn? c_m_o': com_cn_nc; [c_m_o'= c_m_o] ->
242                                 (
243                                 comp_cn! n_s_m;
244                                 exit
245                                 )
246                         []
247                                 (
248                                 comp_np ! n_m_o;
249                                 comp_np ? p_datos' : com_np_pn;
250                                 [p_datos'= p_datos] -> comp_cn ! n_s_m;
251                                 exit
252                                 )
253                 endproc (* mapea_NCL *)
254         process termina_NCL[comp_cn, comp_np] :exit :=
255                         i;
256                         (
257                         comp_np! n_t_o;
258                         exit
259                         )
260                 endproc (* termina_NCL *)
261         endproc (* usa_obm_NCL*)
262 endproc (* activa_obm_NCL*)
263 endproc (* NCL*)
264 endspec (* AME5*)
```

```
1   specification AME6 [comp_cn, comp_np, comp_pc]: noexit :=
2    library NaturalNumber, String, Boolean, Set, Octet endlib
3     Type com_cn_nc is
4       sorts com_cn_nc
5       opns
6             c_l_o, c_e_o, c_m_o, n_s_l, n_s_e, n_s_m : ->        com_cn_nc
7    endtype (* com_cn_nc *)
8   Type consis is NaturalNumber
9   opns
10            consis_datos, consis_pag_ceros, consis_error: ->    consis
11    eqns
12            ofsort Nat
13            succ(0) = consis_datos;
14            succ(1) = consis_pag_ceros;
15            succ(2) = consis_error;
16  endtype
17  Type puertos is NaturalNumber
18   sorts pto
19    opns
20            pto_SMR, pto_PGR, pto_obm, pto_control : ->        pto
21  endtype
22  Type nombres is String
23   sorts nom
24    opns
25            nom_pto, nom_obm, nom_host : ->            nom
26  endtype
27  Type banderas is Boolean
28   sorts ban
29    opns
30            crear_obm, obm_ini, obm_listo, cache, status, termina_obm: ->        ban
31  endtype
32  Type Error is NaturalNumber
33   sorts err
34    opns
35            nom_pto_no_existe, nom_host_desconocido, nom_host_no_encont, error_l, error_m: -> err
36  endtype
37  Type Byte is String actualizedby Octet using
38   sortnames
39                    Octet for Element
40                    Bool for FBool
41                    Byte for String
42  opnames           Octet for String
43  endtype
44  Type Datos is Byte
45   sorts
46    opns
47            datos, datos_m, pags_ceros : ->   Datos
48  endtype
49
50  Type Conj_ptos is Set actualizedby puertos, Boolean using
51   sortnames
52                    pto for Element
53                    Bool for FBool
54                    Conj_ptos for Set
55  endtype
```

```
56   Type Conj_nom is Set actualizedby nombres, Boolean using
57    sortnames
58                    nom for Element
59                    Bool for FBool
60                    Conj_nom for Set
61  endtype
62   Type Conj_obm is Set actualizedby pto, Boolean using
63    sortnames
64                    pto for Element
65                    Bool for FBool
66                    Conj_obm for Set
67   endtype
68   Type Conj_obm_ncl is Set actualizedby pto, Boolean using
69    sortnames
70                    pto+pto for Element
71                    Bool for FBool
72                    Conj_obm_ncl for Set
73   endtype
74   Type tam_pag is Natural_Number
75    sorts tam_pag
76    opns
77          4096    : ->     tam_pag
78    eqns
79          ofsort Nat
80                    succ(4095) = 4096;
81  endtype
82   Type identificador is NaturalNumber
83    sorts identificador
84    opns
85          id_PGR : ->     identificador
86  endtype
87  Type direccion is NaturalNumber
88   sorts
89    opns
90          dir     : ->     direccion
91  endtype
92  Type permisos is NaturalNumber
93    opns
94          lectura, escritura, ejecucion, ->     permisos
95    eqns
96          ofsort Nat
97          succ(0) = lectura;
98          succ(1) = escritura;
99          succ(2) = ejecucion;
100 endtype
101 Type est_copia is NaturalNumber
102   opns
103         copia_inmed, copia_aut_PGR, copia_COW          : ->     est_copia
104    eqns
105         ofsort Nat
106         succ(0) = copia_inmed;
107         succ(1) = copia_aut_PGR;
108         succ(2) = copia_COW;
109 endtype
110 Type tupla_obm is pto, nom, ban, Boolean
111   sorts
```

```
112     opns
113             tupo: pto, nom, pto, ban    (* pto_objeto, nombre_obm, escritor, válido*)    :-> tupla_obm
114             _eq_: tupla_obm, tupla_obm                                                     :-> Bool
115             comp_obm: pto, tupla_obm                                                       :-> Bool
116             comp_esc: pto, pto, tupla_obm                                                  :-> Bool
117             obm_val: pto, tupla_obm                                                        :-> Bool
118             act_esc: pto, pto, tupla_obm                                                   :-> tupla_obm
119             limp_obm: pto, pto, tupla_obm                                                  :-> tupla_obm
120     eqns
121     forall
122      o, c, p, pp, e, ee: pto, n, nn: nom, v, vv: ban
123      ofsort Bool
124             tupo(p, n, e, v) eq tupo(pp, nn, ee, vv) = (p eq pp) and (n eq nn) and
125                                                        (e eq ee) and (v eq vv);
126             (o eq p) => comp_obm(o, tupo(p, n, e, v)) = true;
127             (o ne p) => comp_obm(o, tupo(p, n, e, v)) = false;
128             (o eq p) and (c eq e) => comp_esc(o, c, tupo(p, n, e, v)) = true;
129             (o ne p) or (c ne e) => comp_esc(o, c, tupo(p, n, e, v)) = false;
130             (o eq p) and (v eq true) => obm_val(o, tupo(p, n, e, v)) = true;
131             (o eq p) and (v eq false) => obm_val(o, tupo(p, n, e, v)) = false;
132     ofsort tupla_obm
133             (o eq p) => act_esc(o, c, tupo(p, n, e, v)) = tupo(p, n, c, false);
134             (o eq p) and (c eq e) => limp_obm(o, c, tupo(p, n, e, v)) = tupo(p, n, e, true);
135 endtype
136 Type tupla_obm_ncl is pto, permisos, est_copia, Boolean
137 sorts
138 opns            (* pto_obm, nombre_obm, escritor, válido*)
139             tupon: pto, pto, permisos, est_copia          :->    tupla_obm_ncl
140             _eq_: tupla_obm_ncl, tupla_obm_ncl            :->    Bool
141             obten_p: pto, pto, tupla_obm_ncl              :->    permisos
142             obten_est: pto, pto, tupla_obm_ncl            :->    est_copia
143             act_per: pto, pto, permisos, tupla_obm_ncl    :->    tupla_obm_ncl
144 eqns
145             forall
146             ob, co, c, cc, o, oo: pto, p, pp: permisos, s, ss: est_copia
147 ofsort Bool
148             tupon(c, o, p, s) eq tupon(cc, oo, pp, ss) = (c eq cc) and (o eq oo)
149                                                          and (p eq pp) and (s eq ss);
150 ofsort permisos
151             (co eq c) and (ob eq o)  => obten_p(co, ob, tupon(c, o, p, s)) = p;
152 ofsort est_copia
153             (co eq c) and (ob eq o)  => obten_est(co, ob, tupon(c, o, p, s)) = s;
154 ofsort tupla_obm_ncl
155             (co eq c) and (ob eq o)  => act_per(co, ob, pp, tupon(c, o, p, s)) =
156                                                        tupon(c, o, pp, s);
157 endtype
158 Type estructura is
159             formalsorts tupla
160             sorts estructura
161 opns
162             vacio:                                        :-> estructura
163             agrega: tupla, estructura                     :-> estructura
164             remueve: tupla, estructura                    :-> estructura
165             esta_en: tupla, estructura                    :-> Bool
166 eqns
167             forall x,y: tupla, est: estructura
```

```
168   ofsort estructura
169           agrega(x, agrega(x, est))= agrega(x,est);
170           remueve(x, agrega(x, est))= est;
171           not(x eq y) => remueve(x, agrega(y, est))=agrega(y, est);
172   ofsort Bool
173           esta_en(x, vacio)=false;
174           esta_en(x, agrega(x,est))=true;
175           esta_en(x, remueve(x,est))=false;
176           not(x eq y)=> esta_en(x, agrega(y,remueve(x,est)))=false;
177           not(x eq y)=> esta_en(x, remueve(y,est))=esta_en(x,est);
178   endtype
179   Type est_obm_ncl is estructura actualizedby tupla_obm_ncl
180           using
181   sortnames
182           tupla_obm_ncl for tupla;
183           est_obm_ncl for estructura;
184   endtype
185   Type est_obm is estructura actualizedby tupla_obm
186           using
187   sortnames
188           tupla_obm for tupla;
189           est_obm for estructura;
190   endtype
191   Type datos_obm  is pto,nom, ban, Boolean
192   sorts
193   opns
194           lista: pto, nom, ban, pto, pto                    :->     datos_obm
195           _eq_: datos_obm, datos_obm                        :->     Bool
196   eqns
197           forall
198           pgr, pgr',obm,obm', cont, cont': pto,nomb, nomb': nom, crea, crea': ban
199   ofsort Bool
200           lista(pgr, nomb, crea, obm, cont) eq lista(pgr', nomb', crea', obm', cont') =
201                                   (pgr'=pgr) and ( nomb'= nomb) and ( crea'= crea) and
202                                           (obm'= obm) and (cont'=cont);
203           not(pgr'=pgr) or not( nomb'= nomb) or not( crea'= crea) or
204                                   not (obm'= obm) or not(cont'=cont)=>
205                                   lista(pgr, nomb, crea, obm) eq lista(pgr', nomb', crea', obm') =false;
206
207   endtype
208   behaviour
209     (PGR[comp_np, comp_pc, consistencia] (vacio, vacio, vacio, vacio) |[consistencia]|
210     Control_Consistencia [consistencia]( vacio, vacio, vacio, vacio)
211     |||
212    NCL[comp_cn, comp_np](vacio, vacio)
213     |||
214     CLT[comp_cn, comp_pc]
215   where
216     process Control_Consistencia [consistencia]( objetos: est_obm, obm_ncl: est_obm_ncl,
217                           c_ob: conj_obm, c_on: conj_obm_ncl): noexit :=
218                   ... (*Definición del proceso para el control de la consistencia*)
219     endproc(*Control_Consistencia*)
220     process PGR[comp_np, comp_pc] (objetos: est_obm, obm_ncl: est_obm_ncl,
221      c_ob: conj_obm, c_on: conj_obm_ncl): noexit :=
222      registro_PGR[comp_np] (false, vacio, vacio, vacio, vacio)
223           >>
```

```
224     pedir_obm_PGR[comp_pc]
225     where
226         process registro_PGR[comp_np] (not_nuc: Bool, objetos: est_obm, obm_ncl:est_obm_ncl,
227     c_ob: conj_obm, c_on: conj_obm_ncl): exit :=
228         comp_np! pto_SMR ! nom_pto ! id_PGR ! pto_PGR;
229         comp_np? not_nuc': Bool;
230         [not_nuc'= not_nuc] - > exit (est_obm, est_obm_ncl, c_ob, c_on)
231         endproc (* registro_PGR*)
232         process pedir_obm_PGR[comp_pc]: noexit :=
233         hide op_ini, t in
234         accept est_obm, est_obm_ncl, c_ob, c_on in
235         comp_pc? pto_PGR': pto ? nom_obm': nom ?crear_obm': ban? pto_control': pto ? pto_obm': pto;
236         [lista(pto_PGR , nom_obm, crear_obm, pto_control) eq
237                 lista(pto_PGR, nom_obm, crear_obm, pto_control)] - >
238         comp_np! pto_PGR ! pto_obm ! nom_obm ! obm_ini;
239         [not(esta_en(pto_obm, c_ob ))] -> (
240                 activa_obm_PGR[comp_np, op_ini] (agrega((tupo(pto_obm, nom_obm, 0, true)), objetos),
241         agrega(tupon(pto_obm, pto_control, 0, succ(2)), obm_ncl), agrega (pto_obm,
242                 c_ob), agrega(pto_obm+pto_control, c_on) ) )
243         [] ;
244         [(esta_en(pto_obm, c_ob ) and not(esta_en(pto_obm+ pto_control, c_on )] -> (
245                 activa_obm_PGR[comp_np, op_ini]
246                 (objetos, agrega(tupon(pto_obm, pto_control, 0, succ(2)), obm_ncl), c_ob,
247         agrega(pto_obm+pto_control, c_on) ) )
248         []
249         [(esta_en(pto_obm, c_ob ) and (esta_en(pto_obm+ pto_control, c_on )] -> (
250                         usa_obm_PGR[comp_np, t] (objetos, obm_ncl, c_ob, c_on ) )
251     |||
252     i: pedir_obm_PGR[comp_pc, op_ini]
253     where
254                 process activa_obm_PGR[comp_np, op_ini] (objetos: est_obm, obm_ncl:
255                 est_obm_ncl, c_ob: conj_obm, c_on: conj_obm_ncl): exit :=
256                   comp_np? pto_obm': pto? pto_control': pto? nom_obm': nom? tam_pag': tam_pag
257                   [(pto_obm'= pto_obm) and (pto_control'=pto_control)] -> (
258                           consistencia ! pto_obm' ! pto_control' ? ver_consis: consis;
259                           [ver_consis = consis_ini] ->
260                                   (op_ini;
261                                   comp_np! pto_control' !obm_listo !cache !est_copia;
262                                   usa_obm_PGR[comp_np] (objetos, obm_ncl, c_ob, c_on )))
263                 where
264                         proces usa_obm_PGR[comp_np, t] (objetos: est_obm, obm_ncl:
265                         est_obm_ncl, c_ob: conj_obm, c_on: conj_obm_ncl): noexit :=
266                                 hide op_usa in
267                                 (
268                                 lee_PGR[comp_np, consistencia]
269                                 []
270                                 mapea_PGR[comp_np, consistencia]
271                                 []
272                                 escribe_PGR[comp_np, consistencia, op_usa]
273                                 []
274                                 termina_PGR[comp_np, consistencia]
275                                 )
276                                 [> t; stop
277                         where
278                         process lee_PGR[comp_np, consistencia] :exit :=
279                         comp_np? pto_obm': pto ? pto_control': pto ? tipo_acceso': permiso;
```

```
280                 [(pto_obm'= pto_obm) and (pto_control'=pto_control) and
281                 (tipo_acceso=Succ(0))] ->
282                  consistencia ! pto_obm' ! pto_control' ! tipo_acceso' ? ver_consis: consis;
283                    (
284                     [ ver_consis = consis_datos] -> comp_np ! pto_control ! datos ! permisos; exit
285                    []
286                    [ ver_consis = consis_pag_ceros ] ->
287                                 comp_np ! pto_control ! pag_ceros ! permisos; exit
288                    []
289                    [ ver_consis = consis_error ] -> comp_np ! pto_control ! error_l; exit
290                    )
291                 endproc (* lee_PGR*)
292                 process mapea_PGR[comp_np, consistencia] :exit :=
293                 comp_np? pto_obm': pto ? pto_control': pto ? tipo_acceso': permiso
294                        ?direc': direccion;
295                 [(pto_obm'= pto_obm) and (pto_control'=pto_control) and
296                 (tipo_acceso=Succ(0))] ->
297                  consistencia ! pto_obm' ! pto_control' ! tipo_acceso' !direc' ? ver_consis: consis;
298                    (
299                     [ ver_consis = consis_datos] -> comp_np ! pto_control ! datos_m ! permisos;
300          exit
301                    []
302                    [ ver_consis = consis_error ] -> comp_np ! pto_control ! error_m; exit
303                    )
304                 endproc (* mapea_PGR*)
305                 process escribe_PGR[comp_np, consistencia, op_usa] :exit :=
306                 (
307                 comp_np? pto_control': pto ? datos': Datos ?permisos': permisos
308                 [(pto_control'= pto_control) and (datos'=datos) and
309                        (permisos=Succ(1))] ->
310                        (
311                        consistencia! pto_control' ! pto_obm' ! permisos' ? ver_consis: consis
312                        [ ver_consis=consis_escribe ] -> op_usa: exit
313                        )
314                 )
315                 []
316                 (
317                 comp_np? pto_obm': pto ? pto_control': pto ? tipo_acceso': permiso;
318                 [(pto_obm'= pto_obm) and (pto_control'=pto_control) and
319                 (tipo_acceso=Succ(1))] ->
320                  consistencia ! pto_obm' ! pto_control' ! tipo_acceso' ? ver_consis: consis;
321                    (
322                     [ ver_consis = consis_datos] -> comp_np ! pto_control ! datos ! permisos; exit
323                    []
324                    [ ver_consis = consis_pag_ceros ] ->
325                                 comp_np ! pto_control ! pag_ceros ! permisos; exit
326                    []
327                    [ ver_consis = consis_error ] -> comp_np ! pto_control ! error_l; exit
328                    )
329                 )
330                 endproc (* escribe_PGR*)
331                 process termina_PGR[comp_np, consistencia] :exit :=
332                 comp_np? pto_obm' : pto ? pto_control' : pto
333                        ? nom_obm' : nom ?termina_obm':ban;
334                 [(pto_obm'= pto_obm) and (pto_control'=pto_control) and
335                 (nom_obm'=nom_obm) and (termina_obm'=true)] ->
```

```
336                         consistencia! pto_obm' ! pto_control'; exit
337                         endproc (* termina_PGR *)
338         endproc (* usa_obm_PGR*)
339     endproc (* activa_obm_PGR*)
340    endproc (* pedir_obm_PGR*)
341  endproc (* PGR*)
342  process CLT[comp_cn, comp_pc]
343         pedir_pto_CLT[comp_cn]
344         |||
345         pedir_obm_CLT[comp_pc, comp_cn]
346      where
347       process pedir_pto_CLT[comp_cn, comp_pc] : noexit :=
348                 (
349                 comp_cn! pto_SMR ! nom_host ! nom_pto ! pto_PGR;
350                 (
351                    comp_cn? pto_PGR': pto; pedir_obm_CLT[comp_cn, comp_pc]
352                   []
353                   comp_cn?nom_pto_no_existe': err;stop
354                    []
355                   comp_cn? nom_host_desconocido': err;stop
356                    []
357                   comp_cn? nom_host_no_encont': err;stop
358                 )
359                 |||
360                 i; pedir_pto_CLT[comp_cn, comp_pc]
361      endproc (* pedir_pto_CLT*)
362      where
363         process pedir_obm_CLT[comp_pc, comp_cn]: noexit :=
364                         pide_o_c [comp_pc] >> activa_obm_CLT[comp_cn, comp_pc]
365                         |||
366                         i; pedir_obm_CLT[comp_pc, comp_cn]
367         where
368         process pide_o_c [comp_pc]: exit :=
369                         comp_pc! pto_PGR ! nom_obm ! crear_obm ! pto_control ! pto_obm;
370                         comp_pc?pto_PGR': pto ? pto_obm': pto ? nom_obm': nom ? status': ban;
371                         [status'=true]-> exit
372         endproc(*pide_o_c*)
373         process activa_obm_CLT[comp_pc, comp_cn]: exit :=
374                         act_o_c [comp_pc] >>
375                         usa_obm_CLT[comp_cn, comp_np]
376                 where
377                         process act_o_c [comp_pc]: exit :=
378                           comp_pc! c_m_o; comp_pc? n_s_m  com_cn_nc;
379                           [n_s_m'= n_s_m]-> exit
380                         endproc(*act_o_c *)
381                         process usa_obm_CLT[comp_cn, comp_np]: noexit :=
382                                 (
383                                    lee_CLT[comp_cn]
384                                 []
385                                    escribe_CLT[comp_cn]
386                                 []
387                                    mapea_CLT[comp_cn]
388                                 )
389                                 [> t; stop
390                         where
391                                 process lee_CLT[comp_cn] :exit :=
```

```
392                                     comp_cn! c_l_o;
393                                     comp_cn? n_s_l': com_cn_nc;
394                                     [n_s_l'= n_s_l]->
395                                     exit
396                             endproc (* lee_CLT*)
397                             process escribe_CLT[comp_cn] :exit :=
398                                     comp_cn! c_e_o;
399                                     comp_cn ? n_s_e': com_cn_nc ;
400                                     [n_s_e'= n_s_e]->
401                                     exit
402                             endproc (* escribe_CLT*)
403                             process mapea_CLT[comp_cn] :exit :=
404                                     comp_cn! c_m_o;
405                                     comp_cn? n_s_m': com_cn_nc;
406                                     [n_s_m'= n_s_m]->
407                                     exit
408                             endproc (* mapea_CLT *)
409                     endproc (* usa_obm_CLT*)
410             endproc (* activa_obm_CLT*)
411             endproc (* pedir_obm_CLT*)
412     endproc (* CLT*)
413     process NCL[comp_cn, comp_np] (conj_nombres:Conj_nom, conj_puertos: Conj_ptos):exit :=
414             registro_NCL[comp_np](conj_nom, conj_puertos)
415             >>
416             (
417                 pedir_pto_NCL[comp_cn]
418                 |||
419                 activa_obm_NCL[comp_cn, comp_np]
420             )
421     where
422         process registro_NCL[comp_np] (conj_nombres:Conj_nom, conj_puertos: Conj_ptos): exit :=
423                     comp_np? pto_SMR': pto ? nom_pto': nom ? id_PGR': identificador
424                     ? pto_PGR': pto;
425                     [(pto_SMR'= pto_SMR) and (nom_pto'= nom_pto) and (id_PGR'= id_PGR) and
426             (pto_PGR'= pto_PGR)]->
427                     comp_np! not_nuc; exit(Add(pto_PGR', conj_puertos), Add(nom_pto', conj_nombres))
428         endproc (* registro_NCL* )
429         process pedir_pto_NCL[comp_cn] : noexit :=
430                     accept conj_puertos, conj_nombres in
431
432                             comp_cn? pto_SMR': pto ? nom_hos:  nom ?nom_pto': nom ? pto_PGR': pto;
433                             (
434                             [not(IsIn(nom_pto',conj_nombres))]-> comp_cn ! nom_pto_no_existe; stop
435                             )
436                     []
437                             (
438                             [not(IsIn(nom_host',conj_nombres))]->comp_cn ! nom_host_desconocido; stop
439                             )
440                     []
441                             (
442                             [not(pto_SMR'=pto_SMR)]->comp_cn ! nom_host_no_encont; stop
443                             )
444                     []
445                             (
446                             [(pto_SMR'=pto_SMR) and (nom_host'=nom_host) and
447                             (nom_pto'=nom_pto) and (pto_PGR'=pto_PGR)]->comp_cn ! pto_PGR
```

```
448                     )
449                )
450                |||
451                i; pedir_pto_NCL[comp_cn]
452        endproc (* pedir_pto_NCL*)
453        process activa_obm_NCL[comp_cn, comp_np]: exit :=
454                hide t in
455                usa_o_n[comp_cn, comp_np] >>
456                usa_obm_NCL[comp_cn, comp_np, t]
457            where
458                process usa_o_n [comp_cn, comp_np]: exit :=
459                  comp_cn ? c_m_o': com_cn_nc;
460                 [c_m_o'=c_m_o]->
461                    comp_np ! pto_obm ! pto_control ! nom_obm ! tam_pag ;
462                   comp_np ? pto_control: pto ? obm_listo': ban ? cache': ban ? est_copia': est_copia;
463                   [(pto_control=pto_control) and (obm_listo'=true) and
464                         (cache'= cache) and (est_copia'= est_copia)]->
465                         comp_cn ! n_s_m;
466                         exit
467                endproc(*usa_o_n*)
468                process usa_obm_NCL[comp_cn, comp_np, t]: noexit :=
469                         (
470                         lee_NCL[comp_cn, comp_np]
471                         []
472                         escribe_NCL[comp_cn, comp_np]
473                         []
474                         mapea_NCL[comp_cn, comp_np]
475                         []
476                         termina_NCL[comp_cn, comp_np]
477                         )
478                         [> t; stop
479                where
480                         process lee_NCL[comp_cn, comp_np] :exit :=
481                                 comp_cn ? c_l_o': comp_cn_nc;
482                                 [c_l_o'= c_l_o] ->
483                                 (
484                                 comp_cn ! n_s_l;
485                                 exit
486                                 )
487                             []
488                                 (
489                                 comp_np ! pto_obm ! pto_control ! permisos;
490                                 (
491                                     comp_np ? pto_control: pto ? datos': Datos ? permisos: permisos;
492                                 []
493                                     comp_np ? pto_control: pto ? pag_ceros': pag_ceros
494                                             ? permisos: permisos;
495                                 []
496                                     comp_np ? pto_control: pto ? error_l': err ;
497                                 )
498                                 [(datos'=datos) or (pag_ceros'=pag_ceros) or (error_l'=error_l)] ->
499                                 comp_cn ! n_s_l;
500                                 exit
501                                 )
502                         endproc (* lee_NCL*)
503                         process escribe_NCL[comp_cn, comp_np] :exit :=
```

```
504                     (
505                     comp_cn ? c_e_o': com_cn_nc;
506                     [c_e_o'= c_e_o] ->
507                     solucion_esc_NCL[comp_cn, comp_np]
508                     )
509             where
510                     process solucion_esc_NCL[comp_cn, comp_np] : exit :=
511                             comp_cn ! n_s_e; exit
512                       []
513                         (
514                             comp_np ! pto_control ! datos ! permisos ;
515                             comp_cn ! n_s_e;
516                             exit
517                         )
518                       []
519                         (
520                             comp_np ! pto_obm ! pto_control ! permisos;
521                             (
522                               comp_np ? pto_control: pto ? datos': Datos
523                                     ? permisos: permisos;
524                             []
525                               comp_np ? pto_control: pto ? pag_ceros': pag_ceros
526                                     ? permisos: permisos;
527                             []
528                               comp_np ? pto_control: pto ? error_l': err ;
529                             )
530                             [(datos'=datos) or (pag_ceros'=pag_ceros) or
531                                     (error_l'=error_l)] ->
532                             solucion_esc_NCL[comp_cn, comp_np]
533                         )
534                     endproc (* solucion_esc_NCL *)
535             endproc (* escribe_NCL*)
536             process mapea_NCL[comp_cn, comp_np] :exit :=
537                 comp_cn ? c_m_o': com_cn_nc;
538               [c_m_o'= c_m_o] ->
539                     (
540                     comp_cn ! n_s_m;
541                     exit
542                     )
543               []
544                     (
545                     comp_np ! pto_obm ! pto_control ! permisos! direc;
546                     (
547                       comp_np ? pto_control':pto ? datos_m': Datos ? permisos':permisos;
548                    []
549                       comp_np ? pto_control: pto ? error_m': err;
550                     )
551                     [(datos_m'=datos_m) or (error_m'=errorm)] ->
552                     comp_cn ! n_s_m;
553                     exit
554                     )
555             endproc (* mapea_NCL *)
556             process termina_NCL[comp_cn, comp_np] :exit :=
557                     i;
558                     comp_np ! pto_obm ! pto_control ! nom_obm ! true;
559                     exit
```

```
                    endproc (* termina_NCL *)
                endproc (* usa_obm_NCL*)
            endproc (* activa_obm_NCL*)
        endproc (* NCL*)
    endspec (* AME6 *)
```