UNIVERSITAT AUTONOMA DE BARCELONA

FACULTAT DE CIENCIES

DEPARTAMENT D'INFORMATICA

SIMULACION DE ARQUITECTURAS COMPUTACIONALES

Memoria presentada por Miguel Angel Mayosky para
optar al grado de Doctor en Informática

Barcelona, Diciembre 1989.

UNIVERSITAT AUTONOMA

DE

BARCELONA

FACULTAT DE CIENCIES

DEPARTAMENT D'INFORMATICA

SIMULACION DE ARQUITECTURAS COMPUTACIONALES

. Memoria presentada por

Miguel Angel Mayosky para optar al

grado de Doctor en Informática

Barcelona, Diciembre 1989.

SIMULACION DE ARQUITECTURAS COMPUTACIONALES

Memoria presentada por Miguel Angel Mayosky para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona. Trabajo realizado en el departamento de Informática de la Facultad de Ciencias de la Universidad Autónoma de Barcelona, bajo la dirección del Dr. Emilio Luque Fadón.

Bellaterra, Diciembre de 1989.

Vo. Bo. Director de Tesis

Emilio Luque Fadón.

Agradecimientos.

Muchas personas han sido importantes en el desarrollo de esta Tesis. La lista es larga, y seguramente estará incompleta. Gracias a Emilio Luque Fadón por su acertada dirección y consejo. A José María Catalfo, por su confianza y por hacer posible mi estancia en España. A mis amigos de la Universidad Autónoma: Ana Ripoll, Lola Rexachs, Joan Sorribes, Porfidio Hernandez, Tomás Margalef, Miquel Angel Senar, Tomás Diez, William Fleischman y John Vaughan. A su paciencia, entusiasmo y buen humor se debe gran parte de este trabajo.

Gracias finalmente a mi familia y mis amigos argentinos, por la fe que en mí depositaron. La distancia nunca existió realmente.

Indice

ANEXO A. Listados del programa de simulación

Capítulo 1.

Introducción.

Las estructuras paralelas de cómputo existen, como una posibilidad teórica, desde los inicios del cálculo automático. Ya en 1840 Babbage diseñó instrucciones para la Máquina Analítica en las que la aritmética de indexado y las multiplicaciones podían hacerse simultáneamente. A lo largo de la corta pero vertiginosa historia de las computadoras digitales, se observa una incesante búsqueda de incremento en las prestaciones mediante soluciones descentralizadas y un progresivo aumento de la concurrencia a todo nivel [1]. Fundamentalmente existen cuatro formas de introducir paralelismo en una arquitectura:

-Paralelismo funcional, mediante unidades independientes para llevar a cabo diferentes operaciones artitmético-lógicas sobre distintos datos.
-Paralelismo en forma de línea de montaje o "pipelining", en donde las operaciones matemáticas y de control de la unidad de procesamiento se dividen en "pasos" encadenados que operan de manera concurrente. Este paralelismo "vertical" permite en teoría tener simultáneamente tantas instrucciones como pasos posee el pipeline en algún estado de procesamiento.
-Paralelismo en forma de "baterias" de procesadores idénticos, todos bajo un único control, que realizan la misma operación sobre diferentes datos.
-Paralelismo vía multiprocesamiento, donde unidades de cómputo independientes, cada una de ellas con su propio programa, cooperan en la resolución de un problema.

Las dos primeras tendencias son las responsables de los mayores avances producidos en los monoprocesadores hasta nuestros días. Las dos últimas son motivo de creciente interés desde que la tecnología de fabricación de alta escala de integración las hizo económicamente factibles.

La búsqueda de paralelismo mediante el uso de múltiples unidades funcionales está presente desde los albores del cálculo digital. De hecho, la primera computadora digital de propósito general, la ENIAC, poseía 25 unidades de cómputo independientes, cada una con su propia secuencia de instrucciones, que operaban en paralelo. Los módulos se comunicaban mediante un panel de interconexiones, de forma tal que la estructura de la máquina podía modificarse para cada aplicación. Sin embargo, cuando ENIAC fue operativa, en 1946, comenzaban a aparecer los primeros diseños de ordenadores "de programa almacenado". Los operadores de ENIAC observaron que la máquina podía ser configurada para trabajar en esta forma, sacrificando paralelismo por facilidad de programación. Por lo tanto el panel de interconexiones fue dejado fijo, y así la primera máquina con arquitectura concurrente trabajó toda su vida útil... como una estructura centralizada.

Los progresos en lo que a arquitectura y organización de computadoras se refiere son muy a menudo influenciados por la tecnología disponible en un dado momento y su costo. Observemos por un instante el estado del arte alrededor del año 1960: La arquitectura de Von Neumann ha sido adoptada masivamente, ya que ha demostrado su generalidad y permite una implementación eficiente. La tecnología de válvulas de vacío esta madura, y la memoria de núcleos magnéticos (que ha reemplazado a las líneas de retardo de mercurio) permite "sorprendentes" tiempos de acceso del orden de los microsegundos. Aparecen las arquitecturas tipo "pipeline" y las memorias cache (ATLAS, 1961). Los primeros intentos de descentralización han llegado a las máquinas comerciales con la introducción de los canales de entrada/salida (IBM 704, 1958), para soportar lectoras/perforadoras de tarjetas y cintas magnéticas. Algunas ideas de multiprocesamiento (Unger, 1958 [2]; Holland, 1959 [3]), esperan aún una tecnología que les permita salir del papel. Ha aparecido el transistor.

Un hito importante en la historia de las computadoras paralelas es el artículo de Slotnick et alt. (1962) titulado "The Solomon Computer" [4]. Este artículo describe un array bidimensional de 32 x 32 elementos de procesamiento, cada uno con 128 posiciones de me-

moria de 32 bits y una unidad aritmética de tipo bit-serie. A pesar que Solomon nunca fue construída, ejerció una fuerte influencia en la comunidad científica, y dió lugar a desarrollos tan importantes como ILLIAC IV y PEPE.

El diseño de ILLIAC IV [5] comenzó en 1966 en la Universidad de Illinois con el propósito de implementar las ideas de Solomon. La arquitectura original comprendía cuatro "cuadrantes" de 64 procesadores conectados en malla (8x8). Cada cuadrante poseía una unidad de control global, que generaba las instrucciones para todos los procesadores. Los cuadrantes se comunicaban mediante un bus paralelo de entrada/salida con una unidad de almacenamiento masivo, desde donde se leían las tareas a ejecutar y se escribían los resultados. La "performance" de cada elemento de procesamiento (Pe) debía ser, según el diseño, de una operación de punto flotante cada 240 nanosegundos, con un rendimiento máximo global de 1 Gigaflop/segundo. Aunque sólo fue construído un cuadrante y el rendimiento medido no superó los 50 Mflops/segundo, la máquina fue ciertamente revolucionaria, y la experiencia a que dió lugar afectó profundamente a los desarrollos subsecuentes, no sólo a nivel hardware, sino también en la forma de expresar paralelismo en lenguajes de alto nivel. Tecnológicamente, ésta fue la primera máquina que utilizó dispositivos semiconductores para todo su almacenamiento principal, y circuitos de mediana escala de integración (MSI). El tortuoso camino hacia la construcción física de ILLIAC IV culminó con el primer sistema operando en la Nasa en 1972, a un costo cuatro veces superior al estimado.

ILLIAC IV fue originalmente diseñada para la resolución de sistemas de ecuaciones diferenciales en derivadas parciales. Es interesante ver cómo en este ejemplo temprano de computación paralela ya aparece la idea de adaptación de la arquitectura a la aplicación, un concepto importante en multiprocesamiento. Una línea diferente de trabajo en procesamiento paralelo comienza con el artículo "Parallel computing with vertical data" de Shooman (1962) [6]. Este describe la organización de un ordenador "ortogonal", donde la memoria puede accederse en forma bidimensional, es decir, tanto en el sentido "horizontal" habitual (una palabra a la vez), como verticalmente, en forma de "bit slices", que se procesan en paralelo. Esta idea de efectuar tests en varias palabras simultáneamente dió lugar al concepto de memoria asociativa o direccionable por contenido. Un diseño que utiliza el concepto de ordenador ortogonal es el STARAN [7], originalmente concebido en 1962, y completado en 1972 (Goodyear aerospace). El sistema comprende cuatro módulos de 256 procesadores de un bit

y 64 Mbits de almacenamiento en su configuración máxima. La memoria, a diferencia de Solomon, no es privada de los Pe's, sino que es accedida a través de una red como "slices" de longitud 256. Como muchas computadoras orientadas a bit, STARAN trabaja muy eficientemente en aplicaciones tales como procesamiento digital de imágenes.

A pesar de la utilización de múltiples unidades de cómputo, muchos diseños conservan una organización heredada de la máquina de Von Neumann. Según la misma, la tarea del programa (dictada por uno o varios registros "contadores de programa") es alterar los contenidos de un espacio de memoria unidimensional, que es en todo momento, el "estado" de la máquina. El acceso a esta memoria sigue siendo el aspecto conflictivo, al igual que en monoprocesadores.

Los trabajos en máquinas de flujo de datos y reducción [8-9] intentan paliar este problema mediante el uso de una arquitectura altamente distribuída y de granularidad muy fina, sin memoria global. En estas arquitecturas la ejecución de las operaciones del programa es disparada por la disponibilidad de sus datos de entrada (en el caso de flujo de datos), o por la necesidad de sus resultados (reducción). La primera máquina basada en estos principios (MDFM, Manchester) data de 1975. Desde esa fecha se han realizado numerosos proyectos, que incluyen tanto la especificación a nivel de arquitectura como el desarrollo de lenguajes de programación de alto nivel para la misma.

Los dispositivos de alta escala de integración han permitido la implementación eficiente de muchas ideas sobre multiprocesamiento, dando lugar a la aparición de numerosas máquinas con arquitecturas radicalmente diferentes. La masiva aplicación de microprocesadores, a partir de las segunda mitad de los 70's, ha permitido alcanzar niveles de "performance" inimaginables sólo unos años atrás. Actualmente asistimos a una sorprendente diversificación de las aplicaciones de los sistemas de cálculo (procesamiento digital de imágenes, inteligencia artificial, sistemas espacialmente distribuídos, etc.), que imponen características a la arquitectura ciertamente diferentes de las tradicionales tareas de procesamiento numérico.

1.1.Clasificación

El notable avance en la investigación y desarrollo de sistemas paralelos de cómputo está impulsado por varios factores: Por una parte, la complejidad de cálculo requerida por ciertas aplicaciones (por ejemplo, problemas de física, pronóstico del clima, defensa, etc.), que impide a los actuales sistemas su procesamiento en tiempos aceptables. Además, la tecnología actual se acerca al límite teórico de utilización de los elementos semiconductores, esto es, es cada vez mas difícil incrementar la potencia de cálculo aumentando la velocidad de un único procesador. Finalmente, los costos del hardware son monótonamente decrecientes [10].

Los sistemas paralelos de cómputo proponen la solución a la demanda de potencia de cálculo repartiendo el trabajo entre un número de procesadores operando concurrentemente. Sin embargo, a diferencia de las máquinas tradicionales, basadas en el esquema de Von Neumann, no existe un único "modelo" de arquitectura paralela que englobe a todas las posibles configuraciones [11]. Muchas y muy diversas estructuras han sido propuestas en la literatura, pero ninguna parece ofrecer ventajas sobre otras en un amplio espectro de aplicaciones. Varios son los aspectos a tener en cuenta para caracterizar un sistema de cómputo paralelo [12]; factores fundamentales son:

Forma de manipular los datos.
Granularidad de la arquitectura.
Esquema de interconexión.
Asignación de tareas a los procesadores.

1.1.1.Forma de manipulación de los datos.

La clasificación tradicional (FLYNN) [1] según este criterio es:

1) sistemas SISD (Single Instruction stream, Single Data stream). Esta es la organización tradicional serie de Von Neumann, en la cual un único procesador ejecuta un único programa sobre un conjunto de datos. Es importante notar aquí que en esta categoría quedan englobadas aquellas máquinas que utilizan estructuras tipo pipeline (que es efectivamente una forma de paralelismo).

2) sistemas SIMD (Single Instruction stream, Multiple Data stream). Aquí un único programa controla la operación de un conjunto de unidades que operan en forma síncrona (lock step). En esta clase quedan encuadradas tanto arquitecturas del tipo "array processor" como ILLIAC IV o ICL DAP, como monoprocesadores del tipo vectorial (CRAY-1)

3) sistemas MISD (Multiple Instruction stream, Single Data stream). No se conocen arquitecturas de este tipo.

4) sistemas MIMD (Multiple Instruction stream, Multiple Data stream). Esta clase engloba aquellas máquinas donde cada procesador ejecuta su propio programa en forma autónoma sobre un conjunto de datos. Aquí caben estructuras tan disímiles como redes de "mainframes" hasta conjuntos de microprocesadores fuertemente acoplados.

La clasificación de Flynn es en cierta forma demasiado amplia: dentro de una clase pueden quedar asociadas arquitecturas totalmente diferentes. Un intento de solucionar esta falta de detalle es la taxonomía de Shore (1973) [1]. Esta clasificación está basada en la forma en que se organizan las diferentes partes del ordenador. Como puede verse en la figura 1.1, Shore propone 6 tipos de máquinas:

Máquina I (figura 1.1 (a)): La organización tradicional de Von Neumann con una única unidad de control (CU), de procesamiento (PU), memoria de instrucciones (IM) y de datos (DM). La DM se accede por palabras. La unidad de procesamiento puede contener paralelismo en forma de múltiples unidades funcionales y/o pipelines, por lo que esta máquina engloba tanto a los monoprocesadores escalares como a los vectoriales.

Máquina II (figura 1.1 (b)): Es una máquina similar a la I salvo que la DM se accede en forma de "bit slices" de toda la memoria; la PU está organizada para procesarlos en forma bit-serie. Esta clasificación engloba máquinas tales como STARAN e ICL DAP.

Máquina III (figura 1.1 (c)): Es una combinación de las máquinas I y II, en el sentido de que la memoria puede accederse en forma bidimensional (horizontalmente o en parabras, y verticalmente o en bit slices). Es la máquina ortogonal de Shooman. Diseños como STARAN pueden programarse para trabajar de este modo.

Máquina IV (figura 1.1 (d)): Esta máquina se obtiene replicando los módulos PU y DM de la máquina I, con una única CU. No existe comunicación directa entre PU's, sino que ésta se realiza a través de la unidad de control. Un ejemplo típico de esta máquina es PEPE. Esta falta de canales directos de comunicación limita las aplicaciones de esta arquitectura, pero por otra parte simplifica la adición de nuevas unidades.

Máquina V (figura 1.1 (e)): Es la máquina IV con el agregado de canales de comunicación entre los vecinos mas próximos. El ejemplo típico es ILLIAC IV.

Máquina VI (figura 1.1 (f)): Esta máquina, denominada "logic-in- memory array" (Lima), provee la alternativa de distribuir la lógica de procesamiento sobre toda la memoria. Aquí se encuadran los procesadores asociativos.

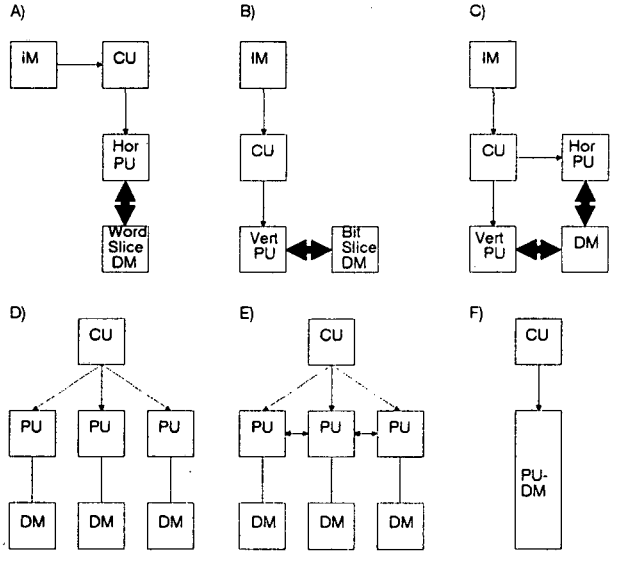


figura 1.1. Taxonomía de Shore

1.1.2. Granularidad.

Una decisión crucial en el diseño de sistemas paralelos es la granularidad o, en otras palabras, el tamaño de las sub-tareas en que es dividida la tarea original. En uno de los extremos tenemos las máquinas de flujo de datos, donde cada sub-tarea es una única operación. Las máquinas de flujo de datos [8-9] presentan el inconveniente del excesivo volumen de co-

municaciones ("overhead") involucrado en el paso de datos entre unidades de cómputo. Por otro lado, el uso de grandes subtareas crea el problema de la partición de la tarea original en forma razonable, y el "mapeo" de éstas en la arquitectura [13].

1.1.3.Esquema de interconexión.

Este aspecto considera tanto la topología de la red que une los procesadores como el mecanismo de comunicación (léase comunicación por paso de mensajes o mediante memorias compartidas), y el acceso a recursos comunes.

Cuando varios procesadores trabajan concurrentemente en una tarea, el problema de las comunicaciones se vuelve fundamental [10]. La cantidad de datos, la frecuencia con que se transmiten, la velocidad de transmisión, y la ruta que recorren son factores importantes que afectan la eficiencia y viabilidad de un determinado esquema de interconexión. Los primeros dos factores dependen, fundamentalmente, del algoritmo utilizado y de la forma en que éste ha sido particionado. El tercer factor depende del hardware utilizado. A continuación nos ocuparemos del último de los factores.

Idealmente, la comunicación entre dos procesadores cualesquiera de la máquina debería realizarse mediante un canal dedicado que directamente uniera los elementos en cuestión. Un canal entre cada posible par de procesadores daría lugar al sistema mas versátil. Además, con un número suficientemente elevado de unidades de cómputo, el "scheduling" sería trivial. Desafortunadamente, tal sistema presenta dos inconvenientes insalvables: complejidad y costo.

Evidentemente, dados n procesadores, esta estrategia dá lugar a $O(n^2)$ canales de comunicación. Para n suficientemente alto, el costo del sistema de interconexión sería intolerable. Tenemos pues, una situación de compromiso entre costo y versatilidad. Este compromiso involucra el "ruteo" de los datos entre dos procesadores no directamente conectados a través de procesadores intermedios. Este esquema tiene varias consecuencias: Primero, existe un retardo adicional en la comunicación debido a los pasos del ruteo. Segundo, debemos agregar a cada procesador las primitivas que permiten hacer este ruteo en forma "inteligente". Cada unidad debe reconocer cuándo el bloque de datos que ha recibido está destinado a

ella o debe ser retransmitido a otra (tal vez, un segundo intermediario). Además, cuando se requieren datos de otro procesador, debe conocerse cómo y cuándo estos datos están disponibles. En resumen, cada procesador debe manejar ciertas "reglas" de ruteo y sincronización.

En términos generales, una estrategia de comunicaciones válida debe tener un número reducido de canales de comunicación, y reglas de ruteo relativamente sencillas. Existen, ciertamente, otras consideraciones importantes a tener en cuenta como, por ejemplo, la tolerancia a fallos (cómo redistribuir los datos y tareas sin perder operatividad en caso de falla en una CPU o canal).

Una primera clasificación de estructuras de comunicación divide a las mismas en dos "clases": Esquemas tipo "link", en donde existe un canal dedicado uniendo dos procesadores, y esquemas tipo "bus", constituídos por un cierto número de buses, cada uno de ellos compartido por varias CPUs.

Ahora bien, ¿cómo comparar dos esquemas de interconexión?. Todas las topologías se proponen con una clase de aplicaciones en mente, para las cuales (teóricamente) presentan alguna ventaja. Esto hace que las comparaciones basadas sólo en la topología (sin tener en cuenta la aplicación) posean una validez relativa.

Los parámetros mas frecuentemente utilizados son:

-Distancia media: es la distancia (medida en cantidad de pasos de ruteo) que, en promedio, debe recorrer un mensaje en la red.
-Número de canales de comunicación.
-Algoritmo de ruteo.
-Tolerancia a fallos.
-Capacidad de expansión.

1.1.4.Asignación de tareas y administración de recursos.

Aquí consideramos la política mediante la cual un problema a resolver es dividido en subproblemas asignados a los distintos procesadores. Este punto es de vital importancia cuando la granularidad deja de ser fina [14].

Supongamos que tenemos un problema a resolver y que poseemos un algoritmo adecuado a tal efecto. Además, este algoritmo ha sido dividido en sub-tareas de modo tal que algunas de ellas pueden operar concurrentemente. A medida que estas sub-tareas son ejecutadas (en diferentes procesadores), pueden requerir intercambio de datos y sincronización. Si podemos estimar el volumen de este intercambio y sus dependencias, estaremos en condiciones de evaluar el comportamiento del algoritmo en la arquitectura.

El problema del "scheduling" puede enunciarse como sigue: ¿cómo hallar la asignación óptima de las sub-tareas a los procesadores de forma tal de maximizar una figura de mérito dada? (por ejemplo, mínimo tiempo de ejecución o máximo aprovechamiento de los recursos). Por un lado tenemos varias sub-tareas con ciertas interrelaciones en lo que hace a sus dependencias de datos (grafo del algoritmo), y por el otro, varios elementos de procesamiento (no necesariamente idénticos) interconectados de alguna forma (grafo de la arquitectura). Debemos "mapear" un grafo en el otro. Esta operación pertenece a la clase de problemas "NP completos", que no poseen solución algorítmica en tiempo polinomial. El parámetro a ser optimizado puede ser el tiempo de ejecución (costo de procesamiento), el número de mensajes intercambiados (costo de comunicación), o una combinación "ponderada" de ambos factores, que constituirían la función de costo del sistema [10] [14-18]. Las técnicas utilizadas habitualmente están basadas en teoría de grafos, o en estrategias heurísticas.

1.2.Implicaciones de la arquitectura en el lenguaje y el estilo de programación.

Todos los lenguajes de programación suponen un cierto "modelo" mas o menos abstracto del sistema de cómputo [19]. Los lenguajes convencionales, basados en la estructura de Von Neumann son, de hecho, versiones de "alto nivel" de la misma. En su forma mas simple, una máquina de Von Neumann tiene 3 partes: una unidad central de proceso, una memoria, y un canal que se utiliza para el intercambio de información entre las dos primeras partes. Esta información consiste fundamentalmente en datos o nombres de datos (direcciones). La tarea del programa es alterar los contenidos de la memoria en cierta forma, moviendo datos en ambos sentidos a través del canal ("Von Neumann Bottleneck").

Los lenguajes tradicionales usan variables para imitar las celdas de memoria, sentencias de control para imitar las instrucciones de salto y test, y operaciones de asignación pa-

ra imitar la decodificación, almacenamiento y aritmética de la máquina de Von Neumann. Aunque esta característica hace parecer a todos los lenguajes convencionales como equivalentes (y de hecho lo son en el sentido de Turing) es bien conocido que cada lenguaje favorece un diferente "estilo" de programación, y una diferente metodología para resolver los problemas (Fortran para aplicaciones científicas y Cobol en aplicaciones comerciales, por ejemplo).

De lo anteriormente dicho, resulta claro que los lenguajes convencionales no proveen un soporte adecuado para el propósito que nos ocupa, esto es, la síntesis de arquitecturas paralelas. El problema se complica aún mas ya que no parece clara la existencia de un único "modelo" abstracto del multiprocesador (el equivalente de la máquina de Von Neumann). Es decir, no existe un conjunto ampliamente aceptado de "facilidades" que una computadora paralela debería proveer en forma eficiente.

Se aprecian varias tendencias bien diferenciadas en los lenguajes de programación para sistemas multiprocesadores. La primera es utilizar lenguajes convencionales [20], dejando al compilador la tarea de buscar el paralelismo subyacente en el algoritmo. Esto no parece muy eficiente, ya que el código de un programa escrito en un lenguaje secuencial suele exhibir dependencias de datos "artificiales" (por ejemplo, al permitir usar la misma variable temporal en distintos lugares del programa para distintos fines, se está generando una dependencia que en realidad no es tal). Sin embargo, esta propuesta permite utilizar gran cantidad del software ya existente, desarrollado para monoprocesadores, en sistemas paralelos (una ventaja de indudable valor económico).

La segunda tendencia es "ampliar" los lenguajes tradicionales de programación con sentencias que permitan expresar el paralelismo en forma explícita [12]. La principal ventaja de esta solución es que la herramienta es familiar al programador, y provee un medio natural para comenzar a "pensar en paralelo". Esta estrategia ha sido utilizada en la mayoría de los sistemas comerciales. Su principal desventaja es la presencia de "efectos laterales" (producidos porque, de hecho, el modelo subyacente sigue siendo el de Von Neumann)[21], lo que complica la depuración y puesta a punto del programa. Los efectos laterales aparecen fundamentalmente en las dos situaciones siguientes:

a)Procedimientos que modifican variables de un nivel superior, no pasados como parámetros.

b)"Aliasing", producido por el paso de parámetros por referencia.

A la tercera tendencia pertenecen los lenguajes funcionales [22- 23] y de flujo de datos [21]. Estos lenguajes presentan una serie de características interesantes, que los hacen atractivos para su uso en multiprocesadores:

-Ausencia de efectos laterales, lo que asegura que la secuencia de instrucciones coincide con la dependencia de datos. En este sentido, los lenguajes funcionales poseen las siguientes características:
-no existen variables globales o Common.
-un procedimiento no puede modificar ni siquiera sus propios argumentos.

Los lenguajes funcionales solo permiten el paso por valor (en muchos de estos lenguajes, por ejemplo FP [22], ha desaparecido el concepto de nombre del dato). Este hecho, si bien provee facilidades para la verificación del algoritmo, genera un aumento considerable del tráfico entre procesadores (tal vez "el" problema de las máquinas de flujo de datos).

-Localidad de efecto: las instrucciones no tienen dependencias de datos artificiales. Esto se logra con estrictas definiciones del "alcance" de las variables, utilización de "bloques" con claras especificaciones de entrada-salida, ausencia de GOTOs y asignación única de variables dentro de un bloque. Todas estas características simplifican en gran medida el proceso de compilación y búsqueda de paralelismo.

Una cuarta tendencia la constituirían los lenguajes específicamente ideados para estructuras paralelas de cómputo. Tal vez el ejemplo mas importante de este tipo de lenguajes sea Occam, desarrollado para la familia Transputer de Inmos, y basado en las ideas de Hoare sobre procesos concurrentes [22-23]. En Occam puede verse claramente el concepto que enunciabamos anteriormente: Las facilidades del lenguaje y del hardware coinciden perfectamente. Obviamente, el problema con Occam es la portabilidad hacia otros sistemas.

La figura 1.2 presenta una clasificación debida a Treleaven (1988) [24]. Esta divide los sistemas paralelos en siete categorías de acuerdo al "estilo" de programación paralela.

Programming Languages						
Procedural Languages	Objet-Oriented Languages	Single Assignment languages	Applicative languages	Predicate Logic Languages	Production System Languages	Semantic Nets Languages
ADA, OCCAM	SMALLTALK	ID, VAL	Pure LISP	PROLOG	OP5	NETL

Computer architectures						
Control Flow	Actor	Data Flow	Reduction	Logic	Rule-Based	Connectionist
TRANSPUTER	DOOM	Manchester DFM	GRIP	ICOT PSI	Non-Von	CONNECTION MACHINE

figura 1.2. Clasificación según el lenguaje

En primer lugar tenemos los lenguajes procedurales y las máquinas de flujo de control. En ellas la secuencia de instrucciones viene dada explícitamente (Sequent Balance, Transputer). En los lenguajes procedurales (Ada, Occam) la operación fundamental es la asignación, y las estructuras de control son secuenciales.

Luego tenemos los lenguajes orientados a objeto (Smalltalk, etc.) y las computadoras basadas en el concepto de "actor". En estas máquinas (APIARY, DOOM) la ejecución de una instrucción es disparada por la llegada de un mensaje o paquete de datos. En lenguajes como Smalltalk, las estructuras de datos y los procedimientos que las manipulan están encapsulados en "objetos" que se comunican mediante el paso de mensajes.

En tercer lugar están las máquinas de flujo de datos y los lenguajes de asignación única, ya descritos. En cuarto lugar están los lenguajes aplicativos (Pure Lisp, etc) y las máquinas de reducción (Alice, Grip). En ellas, la necesidad de un resultado dispara la ejecución de la instrucción que lo genera.

En quinto lugar están las máquinas diseñadas para programación lógica en lenguajes como Prolog. Aqui encontramos proyectos como el ICOT y la ya famosa (y un tanto difusa) quinta generación de ordenadores japoneses.

El sexto lugar lo ocupan los "production system languages" (OP5, etc) y los sistemas basados en reglas. En ellos, las sentencias son básicamente del tipo if..then.., que son ejecutadas repetidamente hasta que ninguna se verifica.

Finalmente tenemos los lenguajes basados en redes semánticas (Netl, etc.) y los modelos conexionistas de computación (Connection machine). En las redes semánticas los conceptos están representados por nodos de un grafo, y las relaciones entre conceptos por los arcos que los vinculan. Los modelos coneccionistas intentan reflejar la organización y mecanismos de aprendizaje de los sistemas neuronales biológicos.

De esta discusión se desprende que las consideraciones acerca de la arquitectura aparecen desde el principio: en cierta forma, el enunciar un algoritmo para la resolución de un problema ya está presuponiendo algunas características de la máquina. Cualquier intento de síntesis de arquitecturas debe comenzar pues, con una especificación clara del "modelo de programación" elegido [11]. Esto quiere decir que el resultado del proceso de síntesis no será otra cosa que la generación de una instancia de esa "máquina virtual", sintonizada a las necesidades de la aplicación. El modelo debe ser lo suficientemente flexible como para ser aplicable en una gama amplia de problemas, y a la vez, debe reflejar claramente los compromisos del diseño.

1.3.Simulación: herramienta de análisis y verificación.

Ante la variedad de factores que intervienen en un sistema de cómputo, surge inevitablemente la cuestión de cómo evaluar las prestaciones de una arquitectura dada (análisis) o cómo decidir entre las diversas alternativas posibles, en el caso del diseño. El rendimiento de un sistema paralelo puede ser caracterizado mediante diferentes índices [25]:

a) Tiempo total de ejecución de un programa o grupo de programas (benchmarks).
b) Porcentaje de cómputo útil para cada procesador del sistema (utilización).
c) Número de mensajes retransmitidos entre procesadores (confiabilidad).
d) Capacidad del sistema de comunicaciones para soportar tráficos elevados. Tráfico máximo admisible.
e) Velocidad de la interconexión (retardo medio o latencia de mensajes).
f) Número medio de pasos de ruteo.

Aunque todos estas medidas son útiles, la información que suministran es sólo parcial. Los programas de benchmark, por ejemplo, suelen proporcionar resultados engañosos, ya que en realidad solo están midiendo la adaptación de un algoritmo a la arquitectura. Este efecto, que ya aparece en sistemas monoprocesadores, se acentúa aún mas en sistemas paralelos, donde las diferencias entre arquitecturas suelen ser mayores. Aquí factores tales como la elección del algoritmo y la forma en que éste ha sido paralelizado, particionado en tareas y mapeado en la arquitectura pueden distorsionar los resultados notablemente.

La evaluación de "performance" sobre un prototipo puede ser reemplazada por la emulación. Esta herramienta es utilizada a menudo en el diseño de una nueva máquina. Requiere la simulación a nivel de código de todo el software de la misma, y de sus funciones hardware de bajo nivel. Los tiempos de CPU involucrados en una emulación detallada son altos, y los resultados sólo aplicables a una máquina específica. Aunque esta metodología provee los elementos de juicio mas detallados para el diseño, no es apropiada para comparar un gran número de posibles implementaciones.

La simulación presenta una solución de compromiso entre precisión en los resultados y eficiencia computacional. Constituye una forma efectiva de estudiar las alternativas existentes a nivel de diseño, siempre que el ambiente de simulación presente adecuadas facilidades de medida y versatilidad de modelado. Este ambiente debe proveer una forma de estudiar la interacción entre algoritmos y arquitecturas, los efectos de los cambios en los recursos sobre la "performance" del sistema, y el análisis de las diferentes estrategias de mapeo, asignación de recursos y ruteo. A los efectos de mantener los tiempos de simulación en niveles aceptables, ésta debería concentrarse en los aspectos de comunicaciones, scheduling y cambio de

contexto del sistema bajo esudio, esto es, aquellas facetas críticas de la operación de multiprocesadores.

1.3.1.Antecedentes

Varios proyectos han enfocado el problema del modelado de sistemas multiprocesadores y el desarrollo de ambientes de programación paralela. Obviamente, todo sistema de cómputo puede ser estudiado mediante el uso de lenguajes de simulación de propósito general, tales como GPSS, SIMULA o SIMSCRIPT. Sin embargo en este apartado nos ocuparemos de aquellos simuladores específicamente diseñados para el análisis de sistemas paralelos. La razón que justifica el desarrollo de una herramienta de este tipo es, fundamentalmente, rendimiento. Al limitar la capacidad de modelado a una clase específica de objetos, el simulador puede ser optimizado, tanto en lo que se refiere a velocidad como al uso de memoria. Además, la interfaz con el operador puede "sintonizarse" a la aplicación de forma tal de agilizar el desarrollo de un modelo correcto. Estas características hacen que la simulación pueda realizarse en forma interactiva, a menudo mediante una interfaz gráfica, permitiendo así al usuario el desarrollo de "intuición" acerca del sistema bajo estudio.

Muchos de estos sistemas han sido diseñados con el propósito de proveer un ambiente para el desarrollo y verificación de algoritmos paralelos en arquitecturas específicas. Tal es el caso de POKER [26]. Este sistema fue construído para un multiprocesador con red de interconexiones variable (CHiP: configurable highly parallel processor), y posteriormente extendido para el Cubo Cósmico. POKER presenta un ambiente integrado, con una potente interfaz gráfica, donde el usuario es guiado a través de las diferentes fases de especificación del sistema: algoritmo, asignación de tareas y generación de la estructura de interconexión. Debemos notar que POKER no es una herramienta de propósito general, sino que está exclusivamente desarrollado para CHiP.

EUCLID [27] está basado en un modelo denominado "red de procesamiento" (processing network). Esta red está especificada mediante una tupla [T,f,P], donde T es un conjunto de elementos "terminales" (memoria, I/O, etc.), P es un conjunto de procesadores , y f es un mapeo de P en T. Las entradas al sistema son topología, algoritmo a ejecutar, e información sobre prioridades de acceso a memoria. EUCLID permite obtener tanto medidas so-

bre la arquitectura como los resultados del programa ejecutado. A pesar de que EUCLID ha sido utlizado con éxito para el estudio de ciertas estructuras basadas en buses múltiples, la "red de procesamiento" no parece ser un formalismo de representación de gran flexibilidad.

Otro desarrollo similar es PARET [25]. Este sistema utiliza grafos dirigidos para modelar el software del usuario, las funciones del sistema (ruteo, etc.) y la red de interconexión. Una arquitectura es aquí modelada por su topología y aquellas funciones requeridas para el paso de mensajes. PARET está orientado hacia estructuras de tipo "link", sin memoria compartida. Tal vez la característica mas interesante de PARET sea su interfaz hombre-máquina, basada en gráficos animados, ventanas y selección mediante ratón. Esta interfaz es una evolución de PAW (Performance Análisis Workstation)[28], una herramienta para la simulación de modelos de colas.

Un enfoque diferente puede apreciarse en SADAN [29]. Este es un lenguaje de simulación específicamente diseñado para el análisis de sistemas paralelos, e implementado como un superset de Pascal [30]. SADAN posee facilidades para la creación y eliminación de "módulos" que pueden representar tanto hardware como software, y tiene predefinidas estructuras tales como buses, "links" con características "threestate", colas de mensajes, etc. Los resultados se obtienen en forma de estadísticas generales e histogramas. SADAN carece de interfaz gráfica.

Finalmente, debemos citar a los lenguajes de especificación de hardware (HDL's)[31-34], desarrollados fundamentalmente para la descripción de dispositivos VLSI. Estas herramientas, cuyo objetivo es automatizar las fases de "bajo nivel" del diseño, pueden asimismo utilizarse para una simulación a nivel de sistema. Sin embargo, el alto grado de detalle que requieren, supone tiempos de simulación excesivamente altos, cuando la arquitectura deja de ser trivial.

1.4.Vista general del trabajo.

Este trabajo analiza la utilización de herramientas de modelado y simulación para el estudio y diseño de sistemas multiprocesadores. En tal sentido se ha desarrollado una metodología de representación formal de los elementos fundamentales que afectan la "perfor-

mance" de los ordenadores paralelos: algoritmos, estructuras de interconexión, políticas de asignación de tareas y comunicaciones. De esta forma, se pretende dar una visión totalizadora del problema, en la que se evidencien las múltiples alternativas posibles en la búsqueda de la adecuada combinación hardware/software para una aplicación.

En este primer capítulo damos una visión general de los ordenadores paralelos. Comenzamos con una breve reseña histórica que pone en evidencia la rápida evolución de estos sistemas, desde la máquina de Von Neumann hasta las soluciones altamente descentralizadas de nuestros días. Analizamos luego algunas propuestas de clasificación, y finalmente citamos los problemas clásicos de las máquinas paralelas: asignación de tareas en estructuras de granularidad alta, lenguajes y modelos de programación, adaptación del algoritmo a la arquitectura, etc.

El capítulo dos es un breve resumen de la teoría de simulación, con un especial énfasis en el formalismo de eventos discretos. Este capítulo provee una notación uniforme, que será utilizada posteriormente para la especificación del sistema. Se definen también los importantes conceptos de validación del modelo y verificación del simulador. Asimismo se describe el algoritmo elemental de simulación.

El capítulo tres resume las principales tendencias de modelado para los diferentes subsistemas presentes en un ordenador paralelo, y se hace una propuesta original al respecto.

El modelado de algoritmos paralelos se describe mediante redes de Petri y diversos tipos de grafos. En cada caso se detallan los pros y contras de cada formalismo, indicando su grado de detalle, complejidad y aplicación. Se propone un esquema de representación formal (WBG), que permite expresar, en forma simple y concisa, tanto el paralelismo "estático" del algoritmo, como sus características dinámicas: lazos, procedimientos recursivos y activación de copias por demanda, de acuerdo a la disponibilidad de sus señales de entrada. Este formalismo pretende aunar las ventajas de la representación utilizada en los grafos de flujo de datos, con una reducción del volumen de comunicaciones, merced al aumento de granularidad. El WBG posee asimismo parámetros para modelar los costos de procesamiento y comunicación del algoritmo.

Con respecto al esquema de interconexión, se describen las técnicas de modelado basadas en lenguajes de especificación de hardware, modelos de colas y grafos. El modelo propuesto tiene como principal característica su flexibilidad: cada módulo se define por su comportamiento entrada/salida, eludiendo la referencia a detalles tecnológicos. Este último aspecto, si bien disminuye la precisión cuantitativa de los resultados, permite comparar cualitativamente diferentes esquemas de interconexión sobre una base común. La idea no es aquí probar las bondades de una arquitectura dada sino verificar los efectos producidos por cambios en los diferentes elementos del sistema sobre la "performance" total.

La asignación de tareas en sistemas multiprocesadores se resume en las dos tendencias principales observadas en la literatura: métodos basados en teoría de grafos y métodos heurísticos. Finalmente comentamos los diferentes algoritmos de ruteo de mensajes en la arquitectura, detallando las estrategias estáticas y dinámicas o reconfigurables.

El capítulo cuatro describe el diseño e implementación del programa simulador. Su estructura es altamente modular, y muy eficiente en cuanto a tiempo de ejecución y utilización de recursos. El simulador se describe a nivel de su diagrama de bloques, especificando la funcionalidad de cada unidad y su interfaz con los restantes elementos del sistema. La implementación del simulador en Pascal se provee en el anexo A.

El capítulo cinco presenta el ambiente integrado de simulación de arquitecturas paralelas. El objetivo de diseño aqui ha sido la simplicidad de utilización, de forma tal de reducir el período de aprendizaje del usuario, así como el tiempo de desarrollo de un modelo correcto, por lo que se ha implementado una potente interfaz gráfica. La misma involucra la especificación del sistema de edición de grafos y de las rutinas de soporte gráfico en tiempo de simulación. Aquí se ha puesto énfasis en una operación simple, basada en ventanas múltiples, y dispositivos de selección gráfica. La presentación de resultados "off line" se realiza a partir de la definición de un Instrumento, lo que permite administrar en forma legible y eficiente un gran volumen de información simultánea.

El capítulo 6 presenta un ejemplo de utilización del sistema. Se describe el desarrollo de un algoritmo de scheduling estático para un hipercubo de tres dimensiones, utilizando el procedimiento Simulated Annealing. En particular, el sistema ha permitido la sintonización

de los coeficientes de la ecuación de costo del algoritmo, en función de las características del grafo del algoritmo paralelo, sujetas a las restricciones de la arquitectura.

Finalmente, el capítulo 7 presenta las conclusiones y líneas abiertas de este proyecto. En un área de tal efervescencia como la del cómputo paralelo, es difícil hacer un "pronóstico", aún del futuro inmediato. Las posibilidades son ciertamente excitantes, y hacen compleja la decisión del rumbo a seguir.

Capítulo 2.

Algunos conceptos de simulación

El principal atractivo del uso de modelos es el aumento de la capacidad de decisión sobre un sistema real. Como vemos en la figura 2.1, podemos suponer que la realidad está compuesta por características controlables y no controlables [35]. La parte controlable está referida a todos aquellos aspectos del sistema que podemos modificar, alterar, transformar, aumentar, reemplazar, etc. Obviamente esta división no es de modo alguno absoluta, sino que depende de nuestro punto de vista y objetivos.

Podemos distinguir tres niveles de intervención sobre el sistema: administración, control y diseño.

La **administración** implica el menor nivel de intervención. En forma general, consiste en la especificación de objetivos y la determinación de cursos generales de acción.

En el contexto del **control**, por el contrario, la acción está determinísticamente asociada a una política. Las restricciones aparecen aquí en la selección de las diferentes alternativas posibles de ser implementadas.

El **diseño** presenta el mayor espectro de posibilidades de selección. Es un proceso complejo e infrecuente, mientras administración y control son actividades permanentes.

Modelado y simulación pueden interpretarse como partes de un sistema asistido por ordenador para contribuir a la toma de decisiones en administración, control y diseño. Estos

tres objetivos orientan el proceso de construcción del modelo al definir su interfaz con el mundo exterior, sus componentes, y su nivel de detalle.

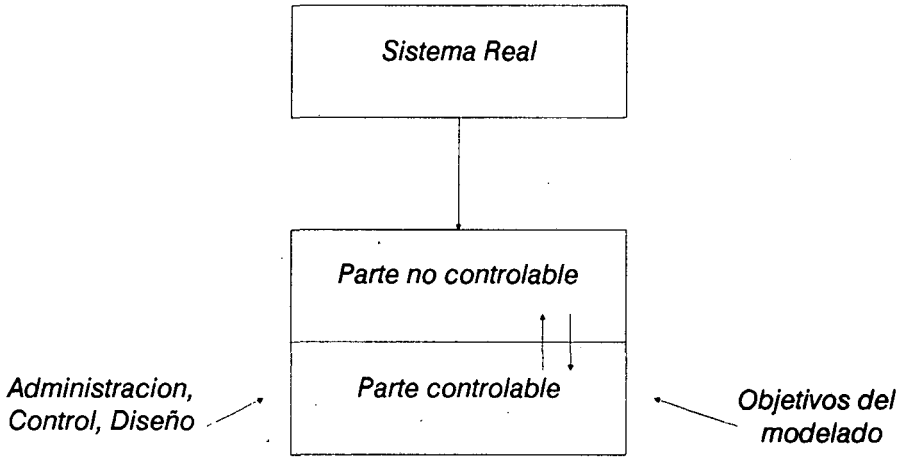


figura 2.1. Componentes de un sistema real

La interfaz define el límite entre el sistema bajo estudio y el resto del mundo, el cual sólo se conocerá a través de las variables de entrada al modelo.

Los componentes de un modelo se especifican como una estructura de procesos, definidos mediante abstracciones que describen su comportamiento. Los mensajes entre procesos se denominan eventos, y su consecuencia es la modificación del estado del sistema. El grado de detalle que se utiliza en el modelo está obviamente influenciado por los objetivos arriba citados.

El efecto de la intervención sobre el sistema es en cierta forma incierto debido a la presencia de la parte no controlable del mismo y su interacción con la parte controlable. Un mayor conocimiento del sistema, obviamente, implica una mayor certeza en los efectos que nuestra acción producirá. Los modelos codifican este conocimiento, siendo su característica mas importante permitir la extrapolación sobre un comportamiento mas allá del conjunto de datos de entrada-salida conocido.

El proceso de modelado y simulación es, en sí mismo, un sistema dinámico (figura 2.2). La síntesis del modelo es disparada por el arribo de nuevos objetivos al sistema, o por una falta de exactitud en los resultados observados. El conocimiento disponible de la activi-

dad de modelado proviene de la base de modelos y de la base de datos del sistema, que almacena y organiza el conocimiento del sistema real. La síntesis del modelo es seguida de las fases de simulación y validación. Esto, a su vez, puede dar lugar a una nueva experimentación sobre el sistema real, y a una eventual modificación del modelo.

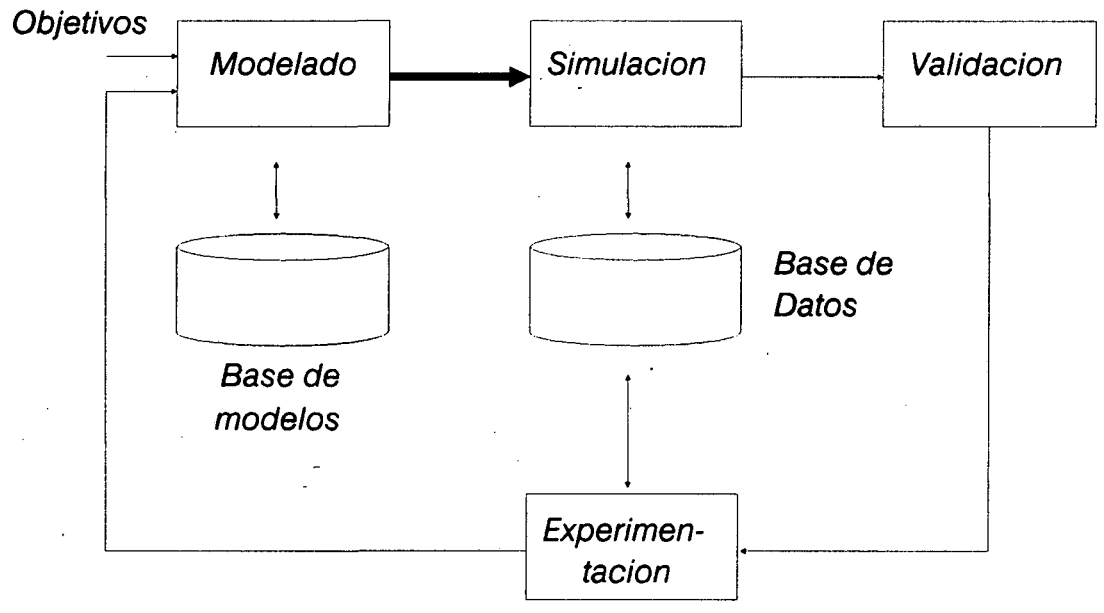


figura 2.2. Modelado y simulación: proceso dinámico

Este proceso iterativo puede no terminar en un único modelo. Los modelos son **competitivos** cuando implican hipótesis diferentes y mutuamente excluyentes acerca del funcionamiento del sistema real. Por el contrario, son **complementarios** cuando poseen las mismas hipótesis de base, pero difieren en la forma de representación. Las dos posibilidades pueden existir en una base de modelos, debido a que nunca poseemos un conocimiento exacto del sistema real, y nuestras observaciones sobre el mismo sólo existen sobre un período de tiempo acotado (problema conocido como validación parcial). La presencia de modelos competitivos limita el grado de confiabilidad de las decisiones emanadas de la simulación. Es importante entonces poder evaluar la sensibilidad de las decisiones (a nivel de administración, control y diseño), con respecto a los modelos alternativos. Los modelos complementarios se basan en las mismas hipótesis, pero una forma de representación puede ser mejor que otra para una aplicación dada(por ejemplo, la representación de un sistema lineal en el dominio del tiempo o de la frecuencia).

2.1.Fundamentos y formalismos.

La discusión sobre metodología de modelado se facilita en gran medida mediante el uso de convenciones de comunicación denominadas formalismos. Un formalismo especifica una clase de objetos en estudio en forma general y no ambigua. Su principal ventaja radica en que permite enfocar sólo aquellas características relevantes del objeto en estudio, proceso denominado abstracción. La base para la manipulación de abstracciones es la teoría de conjuntos.

La operación fundamental entre conjuntos es el producto cartesiano, definido como

$$A \times B = \{ (a,b) \mid a \ni A, b \ni B \}$$

es decir, el conjunto de todos los pares ordenados de A y B. En forma general podemos definir n-tuplas (a,....,n).

Generamos **estructuras** al prohibir ciertos pares de un producto cruzado. La estructura R verifica

$$R \subseteq (A \times B)$$

Una estructura útil es el **mapeo** de A en B: $F \subseteq (A \times B)$ tal que por lo menos existe un par en F para cada elemento de A. El mapeo suele también escribirse como $F : B \leftarrow A$.

La especificación de una Teoría comienza entonces con abstracciones y crea estructuras compuestas a partir de las mismas. Eventualmente, para aplicar esas estructuras al mundo real debemos conectar las abstracciones con los objetos reales. Avanzamos hacia los objetos reales agregando mas y mas detalles a las abstracciones, proceso conocido como **concretización**. La concretización es pues, el proceso iterativo de reemplazo de una abstracción por una estructura.

Cada "clase" de objeto está representado por un formalismo que describe sus parámetros y las restricciones que los gobiernan. Para especificar un determinado objeto den-

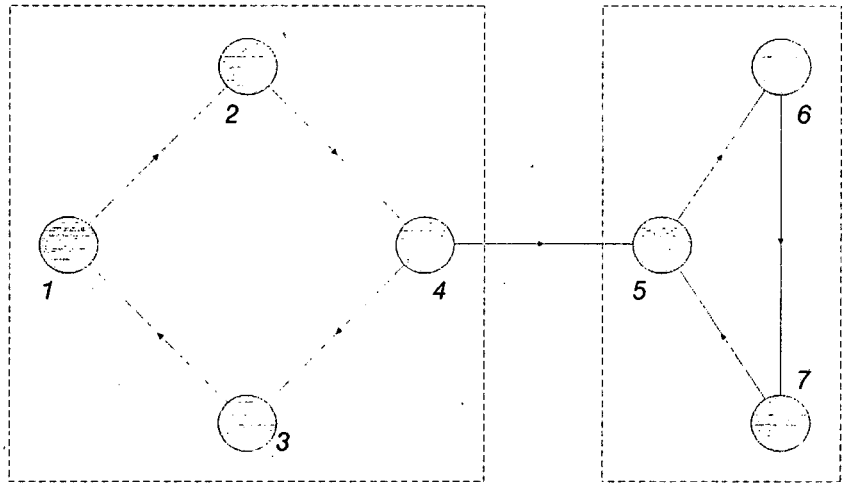
tro de una clase, se asignan valores a los parámetros que satisfacen las restricciones. Algunos ejemplos de formalismos servirán para clarificar estos conceptos:

a) Máquinas de Estados finitos:

Parámetros: entradas, estados, salidas, función de transición de estados, función de salida.
Restricciones: entradas, estados y salidas son conjuntos finitos; la función de transición mapea estados y entradas en estados; la función de salida mapea estados en salidas (Moore) o estados y entradas en salidas (Mealy).

b) Grafo dirigido (figura 2.3):

Parámetros: vértices, arcos dirigidos.
Restricciones: Los vértices y arcos son conjuntos finitos; los arcos dirigidos son pares ordenados de vértices.



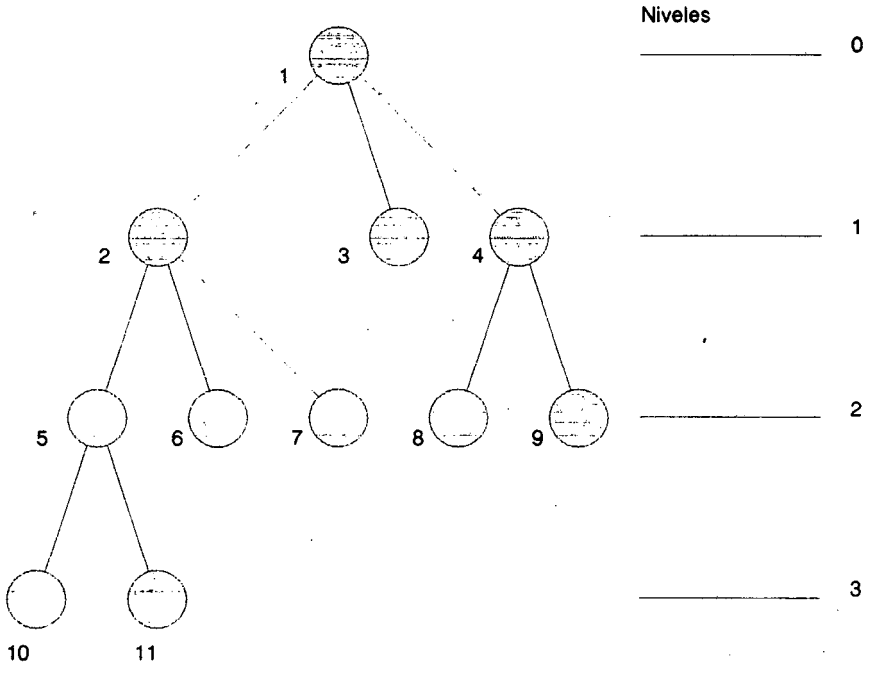
Vertices = { 1, 2, 3, 4, 5, 6, 7 }
Arcos = { (1, 2), (2, 4), (4, 3), (3, 1), (5, 6), (6, 7), (7, 5) }

figura 2.3. Grafo dirigido

c) Arbol (figura 2.4):

Parámetros: nodos, raíz, función sucesor.

Restricciones: nodos es un conjunto finito; raíz es un miembro de nodos; La función sucesor asigna a cada nodo un subconjunto de nodos denominado sucesores; raíz no es sucesor de ningún nodo; todo otro nodo es sucesor de algun miembro de nodos; los sucesores de distintos nodos son disjuntos; ningún nodo pertenece a su propio conjunto sucesor.



Nodos = { 1,...., 11 }
Raiz = 1

Nodo	Sucesores
1	2,3,4
2	5,6,7
4	8,9
5	10,11

figura 2.4.

La estructura de un formalismo provee una forma simple de medir la complejidad de los objetos que representa. Se denomina complejidad formal a aquella que se obtiene directamente de la estructura o topología del modelo. Por ejemplo, en el caso de máquinas de estados finitos, pueden usarse como medidas de complejidad el número de entradas/salidas y/o el número de estados.

2.2.Abstracción, asociación y especificación.

Las clases de objetos están frecuentemente relacionadas entre sí mediante una operación de mapeo de una clase en otra. Son particularmente interesantes para nuestros propósitos tres tipos de mapeo: abstracción, asociación y especificación.

Abstracción es el proceso mediante el cual algunos detalles del objeto en estudio son ignorados en beneficio de una visión mas clara de su estructura. Está pues caracterizada por el mapeo de los objetos de una clase dentro de otra, presumiblemente menos compleja. Es importante notar que el proceso es irreversible, dado que muchos objetos de la clase fuente se ven representados por un único objeto de la clase destino.

Asociación es la operación que permite reunir varias especificaciones parciales para formar una nueva, por ejemplo, en el caso de una estructura formada por una jerarquía de estructuras menores. Esta operación no es biunívoca: el proceso inverso a la asociación puede dar diferentes soluciones alternativas, denominadas implementaciones o realizaciones.

Muchas veces es útil definir un subgrupo dentro de una clase mediante un nuevo formalismo. Se hace necesario entonces definir un mapeo de la nueva clase dentro de la mas general, para poder relacionar ambas entre sí. Tal proceso recibe el nombre de **Especificación.**

En un sistema de simulación, el modelo está expresado mediante un tipo específico de formalismo tal como ecuaciones diferenciales, autómatas o eventos discretos. Cada uno de esos formalismos puede interpretarse como una selección de una clase especial, dentro del conjunto de todos los objetos posibles. La especificación del modelo dentro del formalismo consiste, simplemente, en proporcionar la información necesaria para distinguirlo del resto de los componentes de la clase.

La tabla 2.1 ilustra las subclases de sistemas correspondientes a los principales formalismos de modelado, caracterizando las restricciones que implican en los elementos estáticos y dinámicos del sistema.

	Ecuaciones Diferenciales	Eventos Discretos	Tiempo Discreto
Base de tiempo, T	Reales continuos	Reales continuos	Enteros discretos
Conjuntos basicos:	Espacio de vectores reales	Arbitrario	Arbitrario
Segmentos de entrada	Segmentos contínuos	Segmentos de eventos discretos	Secuencias
Trayectorias de estados y salida	Segmentos contínuos	Segmentos constantes	Secuencias

Tabla 2.1. Formalismos fundamentales de modelado

Obsérvese que la característica distintiva de los sistemas de **eventos discretos** [36] es que los cambios de estados ocurren en forma no contínua, y no uniformemente distribuída en el tiempo. Los cambios de estado internos, junto con los externos o exógenos forman la base para la especificación de la estructura del modelo y su comportamiento.

En el formalismo de **tiempo discreto**, se supone que los cambios de estado se producen cada paso (magnitud constante) de tiempo. Esto da lugar a una especificación relativamente sencilla del modelo : una máquina secuencial o autómata. Sin embargo, la suposición de la presencia de eventos en cada paso de tiempo implica serias desventajas en lo que a eficiencia del simulador se refiere.

A diferencia de los dos formalismos antes citados, una **ecuación diferencial** no prescribe los estados futuros directamente, sino que especifica la variación de la derivada de la trayectoria de estados a partir de un tiempo contínuo t. Dicha trayectoria se obtiene, pues, resolviendo una ecuación. En la práctica, los modelos de cierta complejidad no pueden resolverse en forma analítica, de modo tal que las ecuaciones son normalmente calculadas por métodos numéricos aproximados, usualmente como sistemas de tiempo discreto.

2.3.Formalismos para sistemas de eventos discretos.

La especificación de un sistema de eventos discretos es una estructura

$$M = < X, S, \delta, ta >$$

donde

X es el conjunto de los tipos de eventos externos.
S es el conjunto de los estados secuenciales.
δ es la funcion de especificación de transiciones.
ta es la función de avance del tiempo.

con las siguientes restricciones:

(a) *ta* es un mapeo de S en los reales no negativos con infinito:

$$ta : S \rightarrow R^{+}0, \infty$$

[*ta* es el tiempo que el sistema permanece en el estado s si no ocurren eventos externos]

(b) El estado total del sistema especificado por M es

$$Q = \{(s,e) \mid s \ni S, 0 \leq e \leq ta(s)\}$$

[el estado secuencial s y el tiempo e transcurrido en ese estado son variables de estado significativas]

(c) La especificación de transición δ consiste de dos partes:

(1) La función de transición interna

$$\delta_{\phi} : S \rightarrow S$$

[si no hay eventos externos, el sistema evolucionará del estado s a $\delta_{\phi}(s)$ luego de *ta(s)* unidades de tiempo].

(2) La función de transición externa

$$\delta_{ex} : Q \times X \rightarrow S$$

[si ocurre un evento $x \ni X$, y el sistema ha estado en el estado s por un tiempo e, evoluciona inmediatamente a $\delta_{ex}(s,e,x)$].

La cronología de eventos se representa en este formalismo mediante el concepto de **segmento** de tiempo. Asi, si X representa el conjunto de los posibles eventos externos, los segmentos sobre X se refieren a las posibles secuencias temporales de eventos externos.

Un estado $s \ni S$ para el cual $ta(s)$ es infinito se denomina **pasivo.** El sistema permanecerá en ese estado indefinidamente, y solo podrá evolucionar por efecto de un evento externo. Si $ta(s)$ es finito el estado es activo; en particular, un estado s para el cual $ta(s) = 0$ se denomina transitorio. Las secuencias de estados transitorios expresan cómputo intermedio del modelo, en el sentido de que no consumen tiempo de simulación.

Sea $x \ni X$ un evento externo y (s,s') un par de estados secuenciales tales que $\delta(s,e,x) = s'$ para algún tiempo transcurrido $e \ni [0, ta(s)]$. Si s es pasivo y s' activo, entonces x ha **activado** al modelo; si s' es transitorio, x ha activado **inmediatamente** al modelo. Si s es activo y s' pasivo, x ha **pasivado** al modelo.

La misma terminología se aplica a las transiciones causadas por eventos internos. Obviamente, por definicion es imposible que un modelo en estado pasivo se active a sí mismo sin la presencia de un evento externo.

Si $x \ni X$ se produce cuando el modelo se encuentra en el estado (s, σ) para el tiempo e, y produce una transicion tal que s'(el nuevo estado) $= s$, entonces el evento ha sido **ignorado** por el modelo. El único resultado ha sido la actualización del tiempo de simulación. Si el evento no es ignorado se dice que ha producido una **interrupcion**, generando un cambio de estado no trivial, y una nueva búsqueda de la próxima transición (rescheduling) del modelo.

2.4.Clases especiales de modelos de Eventos Discretos.

Los conceptos de generador, receptor y trasductor pueden generalizarse en el contexto de eventos discretos a partir de su definición mediante teoría de autómatas.

Un modelo es **pasivo** si su función de avance de tiempo es idénticamente infinita. Este tipo de modelos es incapaz de iniciar una actividad por sí mismo, ya que el tiempo entre eventos es infinito. Sin embargo, puede responder a eventos externos y generar una salida.

Los modelos no pasivos se denominan **activos**. Poseen por lo menos un estado en el cual pueden iniciar actividad propia.

Un **generador** es pues un modelo activo que no posee entradas. Es por lo tanto capaz de generar sus propias transiciones de estado internas. Su función de salida está definida de forma tal que es un segmento de tiempo en el sentido del formalismo de eventos discretos. Un generador es útil para modelar situaciones tales como el arribo de procesos a una cola, etc.

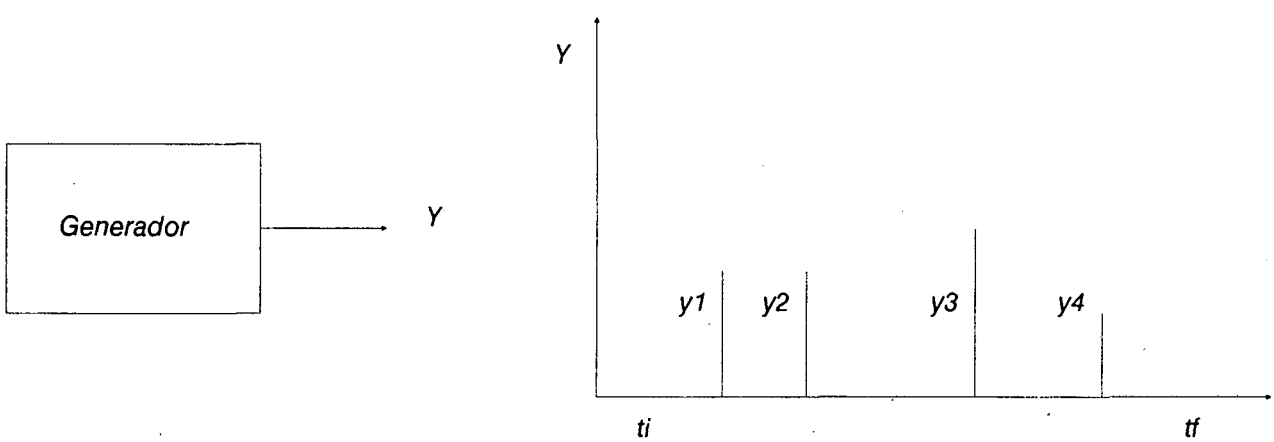


figura 2.5. Generador

Formalmente, un generador está definido como

$$G = < S, \delta_\phi, ta, Y, \lambda >$$

donde los símbolos poseen el significado standard en el formalismo de eventos discretos. X no existe (no hay eventos externos), Y es el conjunto de eventos de salida y λ, la función de salida verifica la siguiente restricción

$$\lambda(s,e) = \phi \quad \text{excepto cuando } e = ta(s)$$

entonces los eventos de salida ocurrirán solo en los tiempos de los eventos internos del modelo (figura 2.5).

Un receptor o aceptor es un modelo pasivo cuyo espacio de estados está compuesto por dos conjuntos, denominados conjuntos aceptor y no aceptor. Además el aceptor designa un estado en el cual el sistema es siempre inicializado. Un segmento de entrada es aceptado por el receptor si éste causa que alcance un estado final perteneciente al conjunto aceptor.

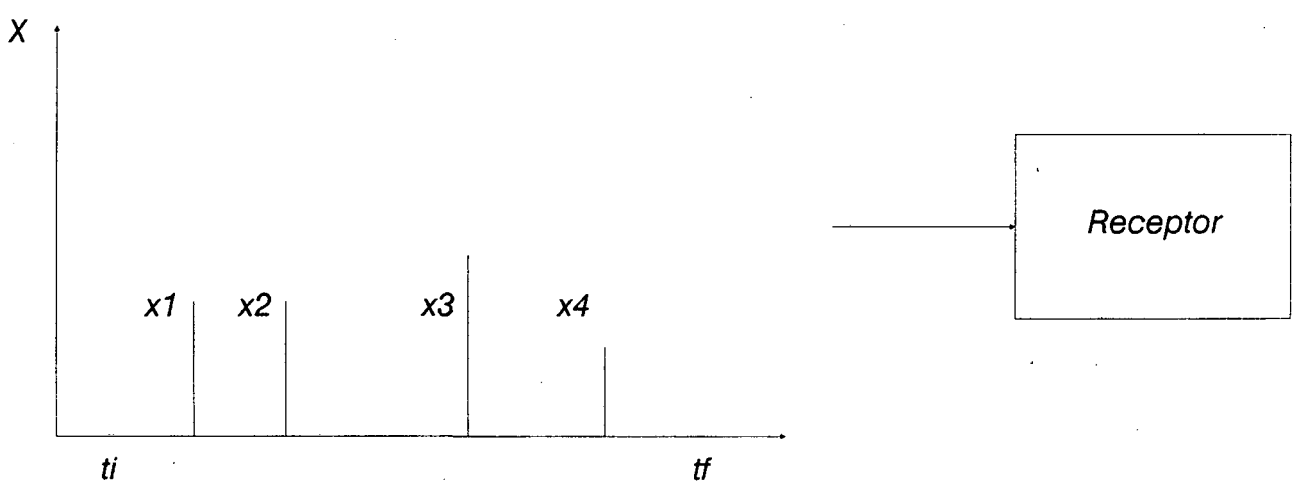


figura 2.6. Receptor o aceptor

Formalmente, un aceptor está definido como

$$A = < X, S, \delta_{ex}, q_0, F >$$

donde, aparte de los simbolos usuales tenemos que

q_0, el estado inicial, y
F, el conjunto de estados aceptores,

verifican las siguientes restricciones:

$$q_0 \ni Q, \text{ y}$$

F es un subconjunto de Q (donde Q es el conjunto total de estados).

Obsérvese que la función de avance de tiempo no se requiere en este caso pues es idénticamente infinita. Un ejemplo de aceptor se ilustra en la figura 2.6.

Un trasductor es un modelo pasivo con un estado inicial fijo. El trasductor mapea sus segmentos de entrada en segmentos de salida. Se denomina trasductor de valor final a aquel cuya salida es nula hasta el final del segmento de entrada. Los trasductores pueden utilizarse para colectar estadísticas sobre las trayectorias del modelo a la manera del mecanismo ACCUMULATE en Simscript 11.5.

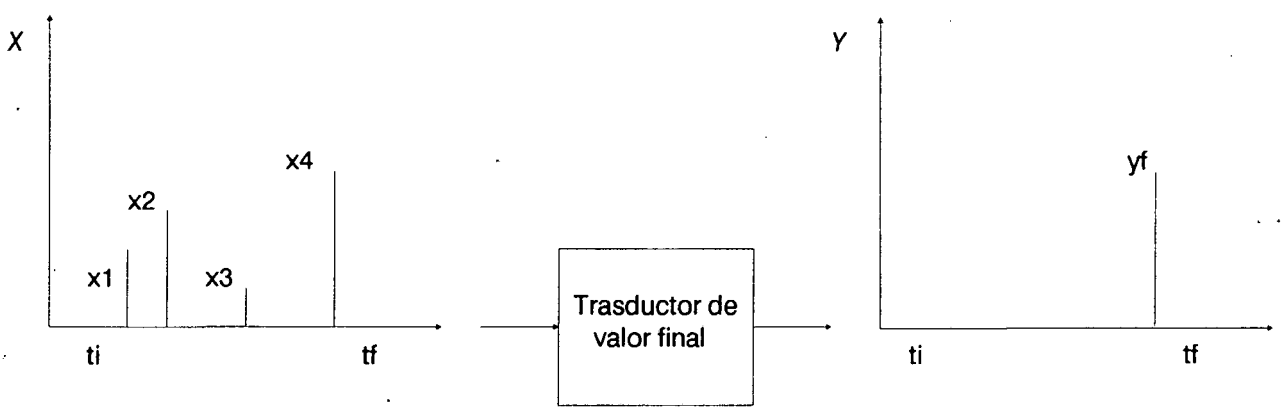


figura 2.7. Trasductor de valor final

Formalmente, un trasductor se define como:

$$T = < X, S, \delta_{ex}, Y, \lambda, q0 >$$

donde $q0$ es el estado inicial.

La función computada por T es el mapeo

$$f_t: \Omega \leftarrow Y$$

definida por

$$f_t(\omega) = \lambda(\delta(q0,\omega))$$

donde Ω es el conjunto de segmentos sobre X y δ es la función de transiciones del sistema especificada por T (figura 2.7).

2.5.Verificación del simulador y validación del modelo.

Básicamente, modelado y simulación involucran 3 tipos de entidades: el sistema real, el modelo y el simulador. Estas entidades deben entenderse en función de sus interrelaciones. Como puede verse en la figura 2.8, existen dos clases fundamentales de operaciones: el modelado, que se refiere a las relaciones entre los sistemas reales y los modelos, y la simulación, como relación entre el modelo y la computadora. Podemos ver entonces al sistema real como generador de datos, al modelo como un conjunto de instrucciones para producir datos, y al simulador como el dispositivo capaz de ejecutar esas instrucciones.

El proceso que lleva a la coincidencia entre datos obtenidos del sistema real y datos del modelo se denomina validación. En lo que sigue trataremos a las 3 entidades arriba citadas como sistemas, empleando los formalismos desarrollados en las secciones anteriores.

El sistema real es una fuente de datos (segmento entrada- salida, R_{sr}). Obviamente, al ser dicho sistema una entidad que evoluciona en el tiempo, jamás estaremos en conocimiento de la totalidad del segmento R_{sr}. Por lo tanto, llamaremos R^t_{sr} a la información conocida del sistema real en el instante t.

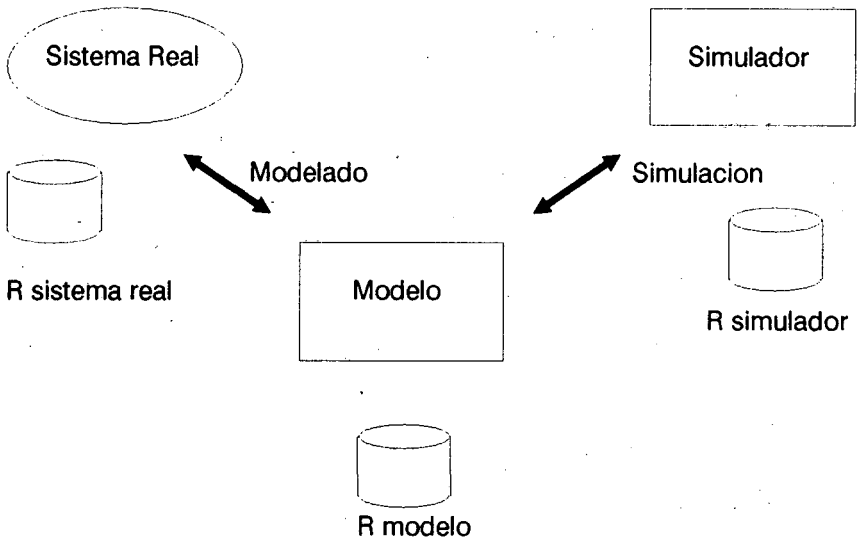


figura 2.8. Entidades de la simulación y el modelado

El modelo es, asimismo, fuente de datos. Mediante simulación deseamos obtener R_m. El proceso que lleva a asegurar que el simulador es correcto se denomina verificación. En todo caso, lo que en realidad obtenemos es R_s, la salida del simulador. Si el simulador es correcto,

$$R_s = R_m \qquad (1)$$

y si el modelo es válido

$$R_m = R_{sr} \qquad (2)$$

Por lo tanto, una correcta combinación [sistema real, modelo, simulador] verifica

$$R_{sr} = R_m = R_s \qquad (3)$$

Mientras (1) es siempre realizable, ya que tenemos control sobre el modelo y el simulador, (2) es utópica, pues del sistema real sólo conocemos $R^t_{sr} \subseteq R_{sr}$, Por lo tanto, sólo podemos asegurar

$$R^t_{sr} = R^t_m \qquad (2')$$

Mas aún, la expresión (2') solo será válida para un subconjunto de datos de entrada-salida ("experimental frame"), que hayamos elegido con algun criterio. Llamemos a este conjunto *E*. (2') queda ahora reducida a

$$R^t_{sr}|_E = R^t_m|_E \qquad (2'')$$

que nos indica que el modelo es válido solo para el conjunto *E* de datos de validación. El modelo puede ser incorrecto para otro conjunto *E'*. *E* se elige en función de los objetivos del estudio en cuestión. Asimismo, *E* limita el intervalo de confianza del modelo, esto es, el entorno dentro del cual el modelo es capaz de contestar correctamente preguntas acerca del sistema real.

2.6.Sistemas multicomponentes

De acuerdo a las ideas que hemos desarrollado en las secciones anteriores, desde el punto de vista del formalismo de eventos discretos los sistemas se presentan como jerarquías o estructuras de procesos elementales. Cada uno de estos procesos representa algún componente del sistema real a ser simulado. Podemos interpretar los eventos que el modelo genera por el **paso de mensajes** entre procesos. Cada evento tiene un tiempo de ocurrencia asociado. La dependencia entre eventos captura nuestro conocimiento intuitivo del ordenamiento temporal causa-efecto. Todos los sistemas físicos modelados mediante el paso de mensajes entre procesos elementales deben verificar dos propiedades: **realizabilidad** y **predecibilidad** [36].

Un modelo es **realizable** si los mensajes generados en el instante *t* solo son función del estado inicial S_0, el tiempo *t* y los mensajes generados hasta el instante *t* (inclusive). Esto simplemente significa que el modelo no puede responder en función de eventos futuros.

Supongamos que el sistema real presenta ciclos, es decir, un conjunto de procesos $pp_0,....,pp_{n-1}$, donde pp_i envía mensajes a pp_{i+1} (y tal vez a otros *pp*) y recibe mensajes desde pp_{i-1} (y tal vez desde otros). Si el mensaje generado por pp_i en el instante *t* depende del mensaje que pp_i recibe en ese mismo instante *t*, para todo *i*, entonces tenemos una definición circular en la que todos los *pp* en todos los intantes *t* dependen de sí mismos (deadlock). Para evitar este tipo de situaciones, se requiere que para todo ciclo y todo *t* exista un *pp* y un núme-

ro real γ mayor que 0 tal que los mensajes enviados por el *pp* a lo largo del ciclo puedan ser determinados en $t + \gamma$. Esto garantiza que el sistema está "bien definido" (predecible), en el sentido que la salida de todo *pp* en el instante *t* puede ser calculada a partir del estado inicial del sistema.

2.7.Algoritmo elemental de simulación.

Las dos principales estructuras de datos utilizadas por el algoritmo de simulación secuencial son el reloj de simulación y la lista de eventos.

Reloj: variable tipo real. Proporciona el tiempo de generación de eventos.

Lista de eventos: Un conjunto de tuplas del tipo *(t,m)*, donde *t* es el valor del tiempo de producción del evento (mayor o igual que reloj) y *m* es el evento (mensaje) asociado.

En cada paso de simulación, el evento con menor tiempo asociado es retirado de la lista, y se transmite el correspondiente mensaje. El envío del mensaje puede, a su vez, generar nuevas inserciones en la lista. El reloj es luego actualizado en el tiempo del evento simulado. Los eventos son pues simulados en su orden cronológico y en orden estrictamente secuencial. Si existen varios eventos con el mismo tiempo asociado, pueden (de acuerdo a lo expuesto en la sección anterior) simularse en orden arbitrario. El algoritmo elemental de simulación sería el siguiente:

Inicialización::

Reloj : = 0;

Lista_de_eventos : = {*(ti,mi)* | el mensaje *mi* será enviado en el instante *ti*}.

Iteración::

Mientras el criterio de fin no se verifique **hacer**

retirar la tupla (t,m) con menor tiempo asociado de la lista de eventos;

simular el efecto de transmitir m en el instante t;

Reloj : = t;

fin { mientras }

2.8.Resumen.

Este capítulo resume los formalismos utilizados habitualmente para la manipulación de modelos. La técnica fundamental de representación es la utilización de jerarquías de procesos elementales, lo que permite una estrategia estructurada de modelización. Se establecen los tres principales estilos de simulación: ecuaciones diferenciales, tiempo discreto y eventos discretos. Este último formalismo es descrito detalladamente utilizando la teoría de conjuntos y analizando sus características dinámicas. Se describen algunos tipos especiales de modelos tales como generadores, receptores y trasductores, que serán útiles mas adelante al especificar el sistema de simulación. Asimismo, se comentan los factores involucrados en la verificación del simulador y la validación del modelo, que delimitan el intervalo de confianza de las respuestas obtenidas del sistema. Por último se establecen las estructuras de datos básicas de un simulador abstracto, y su algoritmo elemental de funcionamiento.

Capítulo 3.

Modelado de sistemas multiprocesadores

Los sistemas de cómputo paralelo basan su funcionamiento en la teórica posibilidad de ejecutar simultáneamente múltiples tareas en forma coordinada. En el logro de este objetivo intervienen un sinnúmero de factores, comprendiendo estructuras de interconexión, factores tecnológicos, granularidad, políticas de sistema (asignación de recursos, sincronización y comunicación), algoritmos, etc.

El uso de herramientas de simulación para la evaluación del rendimiento de estos complejos sistemas supone la especificación de una estrategia de modelado para cada uno de los elementos en juego. Una decisión crucial en este sentido es la elección del grado de detalle con que el modelo describirá al sistema real. Obviamente, la precisión del modelo y la performance del simulador son factores antagónicos que obligan a una solución de compromiso. El modelo debe pues reflejar aquellos efectos críticos del funcionamiento del sistema en su conjunto, conservando a su vez una generalidad tal que permita comparar soluciones radicalmente diferentes sobre una base común, y permitiendo una implementación eficiente del simulador.

Un aspecto importante a tener en cuenta al encarar el diseño de una herramienta de simulación es la coherencia de la filosofía de modelado. Aquí nos referimos a la posibilidad de razonar sobre el modelo y someterlo a transformaciones algorítmicas. De fundamen-

tal importancia es entonces establecer una estructura jerárquica de representación, donde la complejidad se alcance mediante asociación de módulos elementales.

En este capítulo se analizarán los formalismos de modelización para algoritmos, estructuras de interconexión y políticas de sistema. Cada una de las secciones comienza con un análisis de las estrategias utilizadas habitualmente en la literatura, y finaliza con una descripción de la solución adoptada en nuestro caso.

3.1.Algoritmos paralelos.

3.1.1.Modelos utilizados en la literatura.

3.1.1.1.Redes de Petri.

La utilización de esquemas formales para describir algoritmos paralelos está fuertemente influenciada por la aplicación en cuestión. Así, por ejemplo, cuando el objetivo es el estudio de los mecanismos de sincronización y acceso a recursos comunes, la técnica tradicionalmente utilizada está basada en el empleo de Redes de Petri o alguna de sus múltiples variantes. Aquí el problema es enteramente similar al de los sistemas de multiprogramación en monoprocesadores. Las redes de Petri son útiles para el análisis de sistemas concurrentes y protocolos de comunicación, ya que permiten expresar el problema con gran detalle, en forma clara y legible. Este nivel de detalle hace que la complejidad del modelo sea alta: el número de nodos de una red de Petri crece explosivamente con el tamaño del problema.

Una red de Petri (figura 3.1) [37-40] consta, fundamentalmente, de dos tipos de componentes: **estáticos**, que describen la topología del sistema en estudio, y **dinámicos**, que proveen información sobre las condiciones iniciales y la evolución del modelo en el tiempo.

Los componentes estáticos (nodos) son de dos tipos: **lugares** (círculos en la figura 3.1), que representan condiciones, y **transiciones** (barras), que modelan eventos. Círculos y barras se conectan entre sí a través de **arcos** dirigidos, que describen el flujo de información en el sistema.

Las condiciones iniciales, y en general el estado del sistema modelado es representado mediante **tokens**, puntos que se alojan en los lugares e indican que la condición asociada a ese lugar se verifica. La distribución de tokens en los lugares de la red representa el estado del sistema, y se denomina "marking".

Una red se ejecuta mediante el disparo de sus transiciones. En una red "pura", las transiciones se disparan de acuerdo a las siguientes dos reglas:

1) Una transición debe estar habilitada para poder ser disparada, esto es, debe tener por lo menos un token en cada uno de sus lugares de entrada.

2) El disparo de una transición se produce retirando un token de cada uno de sus lugares de entrada y depositando un token en cada uno de sus lugares de salida.

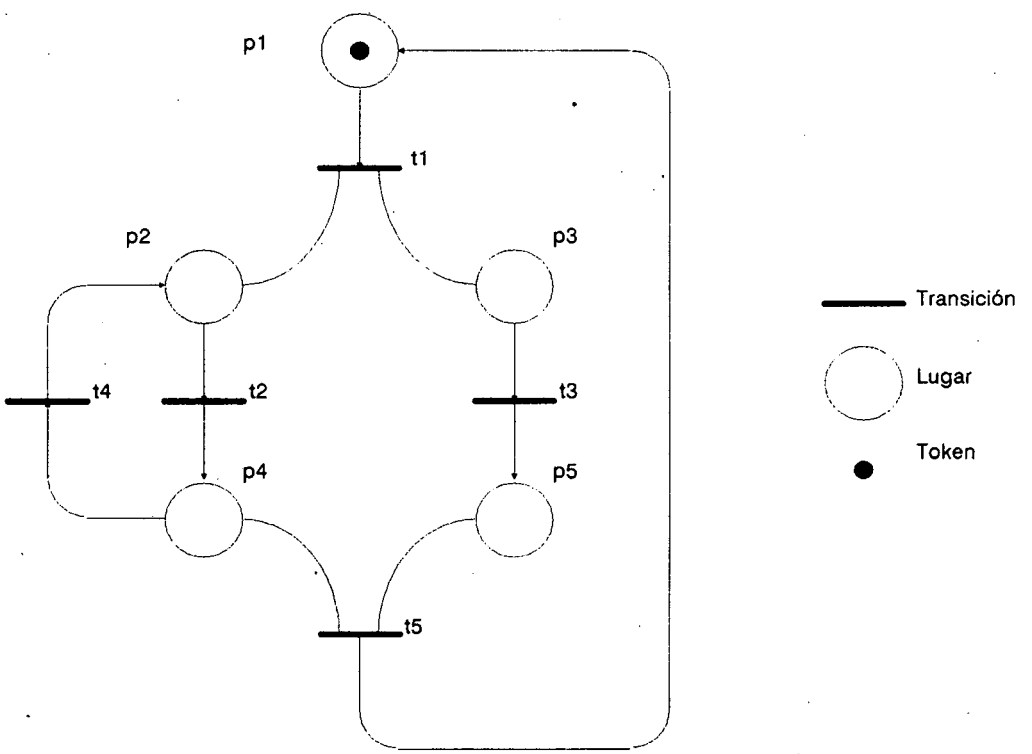


figura 3.1. Red de Petri.

3.1.1.2.Grafos de precedencia. Flujo de datos.

Los grafos de precedencia [41-43] han sido utilizados ampliamente en el diseño de compiladores y optimizadores desde los años '70. En estos grafos, los nodos representan instrucciones o bloques de instrucciones, y los arcos representan las relaciones de dependencia de datos entre los nodos (figura 3.2). Al expresar las dependencias claramente, el optimizador puede operar sobre el código para mejorar alguna figura de mérito tal como memoria utilizada, tiempo de ejecución, etc.

La técnica también ha sido usada con éxito al recompilar programas originalmente desarrollados en máquinas monoprocesadores para su utilización en sistemas paralelos [20]. Aquí el grafo de precedencia se utiliza para buscar la posible reconfiguración de las estructuras repetitivas (lazos) a fin de permitir su ejecución paralela.

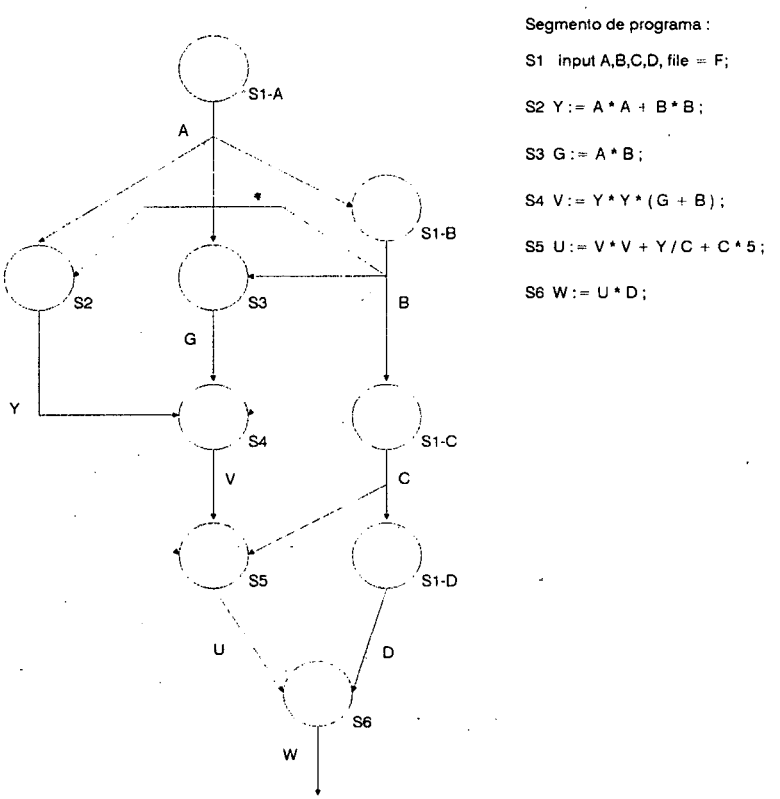
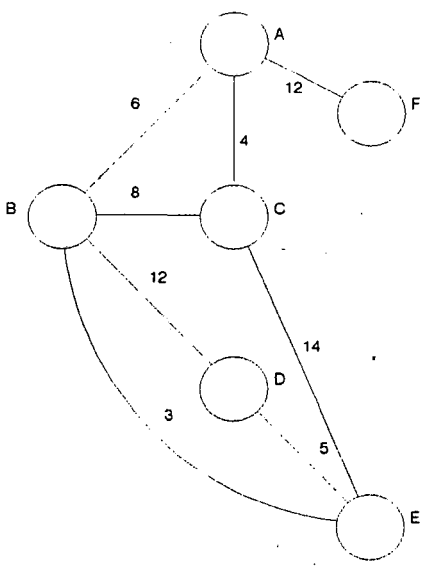


figura 3.2. Grafo de precedencia.

El interesante potencial de los grafos de precedencia ha sido reconocido por diversos investigadores que los han utilizado como lenguaje de programación de un nuevo estilo de máquina, radicalmente diferente de la estructura clásica de Von Neumann. Estas máquinas, denominadas genéricamente de Flujo de Datos, carecen de estructuras de control (contador de programa, etc), y soportan un paralelismo masivo, de granularidad muy fina, donde la sincronización se produce por la presencia de los datos de entrada de cada instrucción. Algunas características interesantes de los grafos de flujo de datos, como por ejemplo su ausencia de efectos laterales, han inspirado el desarrollo de toda una generación de lenguajes de programación de tipo funcional o aplicativo (Lucid, Val, Id, Linda, etc), semánticamente equivalentes a un grafo [44,45]. En estos lenguajes, no existe forma de expresar efectos laterales globales. Este desacoplamiento entre subprogramas hace también posible el desacoplamiento en su ejecución, es decir, dos nodos no conectados entre sí pueden ejecutarse concurrentemente.

3.1.1.3.Grafos de Comunicación.

Una tendencia de modelado diferente se ha utilizado para el desarrollo de políticas de asignación de recursos en computadoras paralelas [13, 15-18]. Obviamente, en estas aplicaciones se desconoce la naturaleza del algoritmo o su propósito (es decir, no interesa saber qué hace), sino que se requiere un conocimiento mas o menos preciso de la topología del grafo (eso es, su patrón de comunicaciones). Ahora los parámetros importantes a conocer son el costo de ejecución de cada nodo sobre cada procesador, el costo de comunicación de cada arco, sus probabilidades,etc. Estos grafos suelen ser no dirigidos (figura 3.3). Obviamente, al ser un algoritmo paralelo un sistema dinámico, cuyo comportamiento depende a menudo de los datos suministrados (su naturaleza, cantidad y valor), esta información es a lo sumo aproximada (cuando no desconocida en la práctica). En función de la técnica de scheduling utilizada (estática-dinámica, diferentes heurísticas, etc), los requerimientos al grafo de entrada suelen variar. A menudo las estrategias se basan en la minimización de alguna función de costo (una sumatoria ponderada por los costos de comunicación y procesamiento, ver capítulo 6).



Costos de intercomunicación

mod.	A	B	C	D	E	F
A		6	4	0	0	12
B			8	12	3	0
C				0	11	0
D					5	0
E						0
F						

Costos de procesamiento

mod.	A	B	C	D	E	F
	5	2	4	6	5	10

figura 3.3. Grafo de tareas a asignar.

3.1.2.Solución adoptada.

En el ambiente de simulación que proponemos, es obvio que se requiere considerar características de los tres tipos de estrategias antes mencionadas. Ya que el sistema debe permitir la simulación del comportamiento del algoritmo, así como su asignación a los diferentes elementos de procesamiento de la arquitectura paralela, y estimar la performance de las distintas alternativas posibles, se ha optado por una solución que exhibe tanto características lógicas como de evaluación de rendimiento. En lo que sigue llamaremos a este formalismo WBG (Weighted Behavioral Graph), y describiremos sus componentes.

Un WBG es, en esencia, un grafo dirigido. Los nodos de este grafo modelan segmentos secuenciales de código, mientras que los arcos entre nodos representan relaciones de dependencia (datos o control). Sin embargo, la definición "estática" del programa paralelo como un grafo es incapaz de reflejar todo el posible paralelismo del mismo debido a la posibilidad de activar varias copias o "instancias" de un mismo nodo durante la ejecución. Por tal motivo se han definido ciertas "reglas de ejecución" para el WBG, ciertamente inspiradas en las máquinas de Flujo de Datos dinámicas. Estas reglas hacen del WBG una herramienta poderosa de modelado, como veremos en las secciones siguientes.

3.1.2.1.Nodos.

Los nodos de un WBG están caracterizados estáticamente por un **volumen** de cómputo, y una **clase**, que describe su comportamiento.

El parámetro de volumen especifica la granularidad del nodo. Este valor es directamente proporcional al número de instrucciones que el nodo modela. Ajustando el valor del volumen de cómputo, el nodo puede modelar desde esquemas de granularidad fina (por ejemplo, una instrucción) o alta (toda una tarea). Es importante notar aquí que el volumen no modela el tiempo de ejecución del nodo, ya que éste dependerá de la performance relativa del procesador al cual se asigne, así como del tiempo que éste le dedique, en el caso de existir "multitasking".

El encapsulado del "estado" de un componente junto con los procedimientos que lo manipulan (esto es, que definen su "personalidad") es una forma eficiente de administrar la evolución de un diseño. Tal es el sentido de la división de los nodos en "clases". Como veremos mas adelante, cada clase lleva asociado un comportamiento específico en tiempo de simulación. En general, las diversas clases definidas (reseñadas en la figura 3.4 y que serán explicadas en las siguientes secciones) pertenecen a tres categorías:

(a) Clases relacionadas con el simple modelado de secciones secuenciales de código y lazos no paralelos. Aquí figuran los nodos "STANDARD" de un modelo WBG.

(b) Clases relacionadas con el comportamiento dinámico del WBG: Las mismas permiten la activación de copias de nodos por demanda y la correcta terminación (comportamiento determinístico) del grafo. A esta categoría pertenecen las clases "CALL", "RETURN" y "JOIN", detalladas en las próximas secciones.

(c) Clases relacionadas con la definición jerárquica del WBG y la administración de su complejidad. Estas clases permiten la estructuración del grafo, y el crecimiento ordenado del grado de detalle del modelo WBG a medida que el conocimiento del usuario acerca del sistema real aumenta. En esta categoría se encuadran las clases "MACRO", "INPUT" y "OUTPUT", analizadas con detalle en el capítulo 4. En esencia, los nodos MACRO poseen subgrafos "internos", que permiten establecer una jerarquia en el modelo, lo que mejora su legibilidad.Pa-

CLASES ESTANDARD	
CLASE	PROPOSITO
STANDARD	Código secuencial o lazo no paralelo.
MACRO	Nodo que contiene otro grafo.
INPUT	Comienzo de un subgrafo MACRO.
OUTPUT	Fin de un subgrafo MACRO.
CALL	Generador de un nuevo espacio de tag.
RETURN	Restauración del espacio de tags.
JOIN	Fin de procedimiento recursivo o FORALL.

figura 3.4. Clases estándar

ra preservar la filosofía de funcionamiento del WBG, todo subgrafo interno de un nodo MACRO debe comenzar con un único nodo INPUT y terminar con un único nodo OUTPUT, lo que preserva el comportamiento macroscópico del nodo.

3.1.2.2.Políticas de entrada y salida.

Generalizando el concepto de redes de Petri, todo nodo posee una política de entrada, que determina las condiciones de disparo del mismo, y una política de salida, que especifica el flujo de información en los arcos que abandonan el nodo. Al igual que en las redes de Petri, la información circula por el grafo en forma de "tokens", cuya distribución en un instante dado constituye el estado del algoritmo.

Dos políticas típicas son las de unión e intersección. Una política de intersección o "and" especifica que el nodo se activará cuando exista al menos un token en cada entrada. Es la regla de disparo de las redes de Petri "puras", y de los grafos de flujo de datos. La política de unión u "or" dispara al nodo cuando existe un token en alguna de las entradas. El comportamiento en las salidas es enteramente similar. Obviamente, son posibles asimismo políticas combinadas. En el caso de la política de salida tipo or, cada arco de salida del nodo lleva también asociada una cierta "probabilidad" de token. Esto es especialmente útil al modelar estructuras alternativas del tipo if...then...else y lazos.

Formalmente, un nodo es una tupla

n_i : < Id, Vp, Class, Pin, Pout, [inputs], [outputs] >

donde

Id : identificación (label o etiqueta) del nodo.

Vp : Volumen de procesamiento.

Class : clase de nodo.

Pin : Política de entrada.

Pout : Política de salida.

[inputs] : conjunto de nodos de entrada.

[outputs] : conjunto de nodos de salida, con su correspondiente probabilidad.

3.1.2.3.Arcos.

Los arcos de WBG se definen por su volumen de comunicaciones, es decir, el "costo" relativo de paso de un token. Por ejemplo, si el token modela el paso de datos, este valor depende de la estrategia de paso de variables (por valor o por referencia). Al igual que en el caso de los nodos, el volumen del arco no modela el tiempo real de comunicación, ya que éste se verá afectado por la asignación de los nodos fuente y destino en la arquitectura, el número de pasos de ruteo y el algoritmo empleado a tal efecto, la performance relativa de los canales de comunicación, etc.

Los arcos, pues, se definen como

a_i : < Ids, Idd, Vc >

donde

Ids : identificación del nodo fuente (el grafo es dirigido).

Idd : identificación del nodo destino.

Vc : volumen de comunicaciones.

La figura 3.5 ilustra un típico WBG. En ella pueden verse estructuras típicas de control tales como Forall, if...then...else, lazos no paralelizables, etc. Esta figura correspondería aproximadamente, a la "versión paralela" de un programa como el siguiente:

```
BEGIN
    FOR I: = 1 TO N DO READ(B,C); { nodo 1 }
REPEAT
    A: = FUNCTION1 (B,C); { nodo 2 }
    IF (CONDITION1(A))  { nodo 3 }
        THEN B: = FUNCTION2(A) { nodo 9 }
        ELSE B: = FUNCTION3(A); { nodo 10 }
    WHILE (CONDITION2(A)) { nodo 4 }
    DO  BEGIN
        FORALL I IN 1..N DO C[I] : = FUNCTION4(A,I) { nodo 5, 6..8, 11 }
        END;
UNTIL (CONDITION3(B,C)); { nodo 12 }
END. { nodo 13 }
```

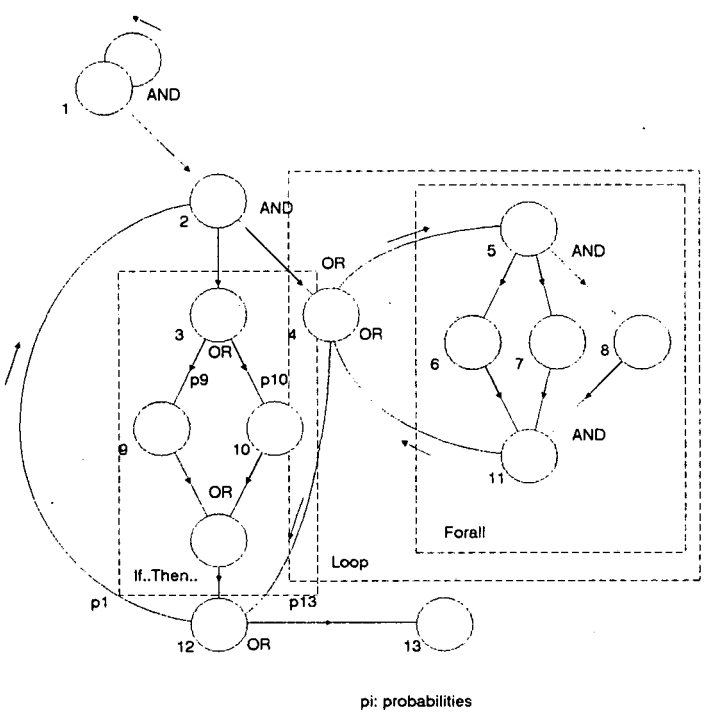


figura 3.5. Weighted Behavioral Graph.

Como veremos en el capítulo 4, nodos tales como CONDITIONi y FUNCTIONj pueden ser modelados como simples retardos, o como nuevos subgrafos. Los mismos se definen mediante la clase MACRO.

La pregunta obvia a estas alturas es ¿Es posible conocer "a priori" el valor de los volúmenes de cómputo y de comunicaciones?. Mientras la duración de ciertas porciones de código y el tamaño de algunas estructuras de datos (variables simples, arrays) pueden determinarse en tiempo de compilación, esto no es la regla. Muchos procedimientos iterativos y recursivos dependen de la naturaleza, valor y cantidad de sus datos de entrada. El formalismo adoptado para el programa paralelo debe ser capaz de modelar adecuadamente esta incertidumbre. Las siguientes secciones reflejan la forma en la que el WBG modela este tipo de situaciones.

3.1.2.4. Características dinámicas del WBG.

La especificación de un algoritmo mediante un WBG como el de la figura 3.5 solo refleja los aspectos estáticos del mismo, esto es, su topología o patrón de comunicaciones. Sin embargo esta representación es incapaz de exhibir todo el paralelismo subyacente en el algoritmo debido a los datos (su valor y cantidad). En tal sentido un grafo como el de la figura 3.5 es sólo una de las posibles "versiones" del algoritmo. Obviamente, si se dispone de suficientes datos en las entradas de un nodo como para activar varias copias o "instancias" del mismo, el paralelismo puede aumentarse notablemente. Evidentemente es menester suministrar un mecanismo que permita asegurar la coherencia en el algoritmo de forma tal de conservar un comportamiento determinístico.

Una solución simple para preservar la coherencia en la operación del grafo es la utilización de tags o colores, una técnica empleada en las máquinas de flujo de datos de tipo dinámico [8,9]. Cada token, además de la información de sus nodos fuente y destino, posee una etiqueta que identifica su número de "instancia", es decir, la identificación del número de activación (color) del nodo fuente. Ahora para habilitar un nodo se requiere que los tokens de entrada cumplan con la regla de disparo y además, posean idénticos tags (figura 3.6). De esta forma se asegura que los datos de entrada a un segmento de programa pertenecen a una misma copia del subgrafo.

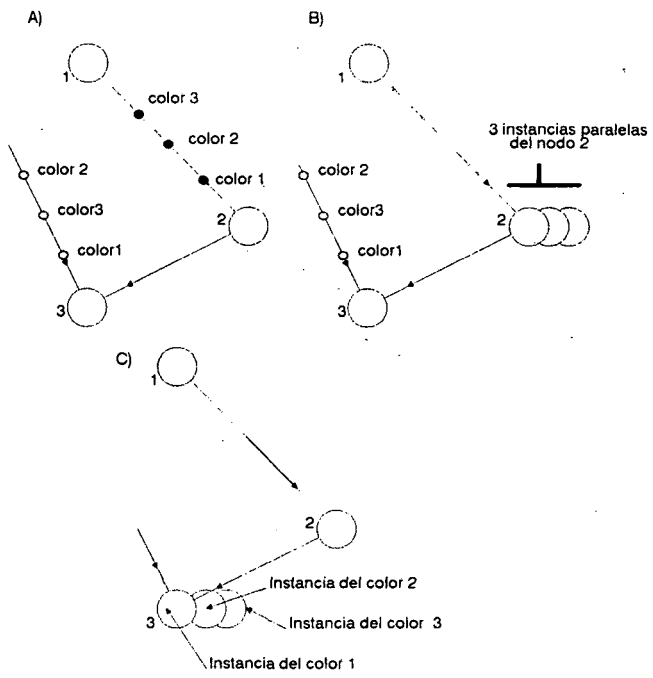


figura 3.6. Utilización de tags.

Pero el uso de un único tag no soluciona todos los problemas si el grafo presenta ciclos o "vueltas hacia atrás". Veamos, por ejemplo, el grafo de la figura 3.7. La ejecución de la instancia 1 del nodo 1 deposita un token con un número de instancia, digamos 1, en las entradas de los nodos 2 y 3. Este último, al tener entrada and, permanece a la espera de un token proviniente del nodo 2.

El nodo 2 se activa inmediatamente debido a su entrada or. Su salida, también or, producirá tokens en alguno de sus dos arcos, de acuerdo a las probabilidades p1 y p2. Imaginemos que luego de ejecutar n veces el cuerpo del lazo se produce un token en el arco 2-3. Debido a que el nodo 2 se ha ejecutado n veces, su tag será n. Ya que los tags no coinciden a la entrada del nodo 3 de la figura 3.7, éste no se disparará nunca.

El problema se agrava aún mas si el nodo 1 genera tokens tan rápidamente que es posible activar varias copias paralelas del lazo (figura 3.8). Ahora también pueden aparecer problemas de coherencia en los datos, ya que por ejemplo, la instancia 2 del lazo puede presentar menos iteraciones que la 1 y terminar antes.

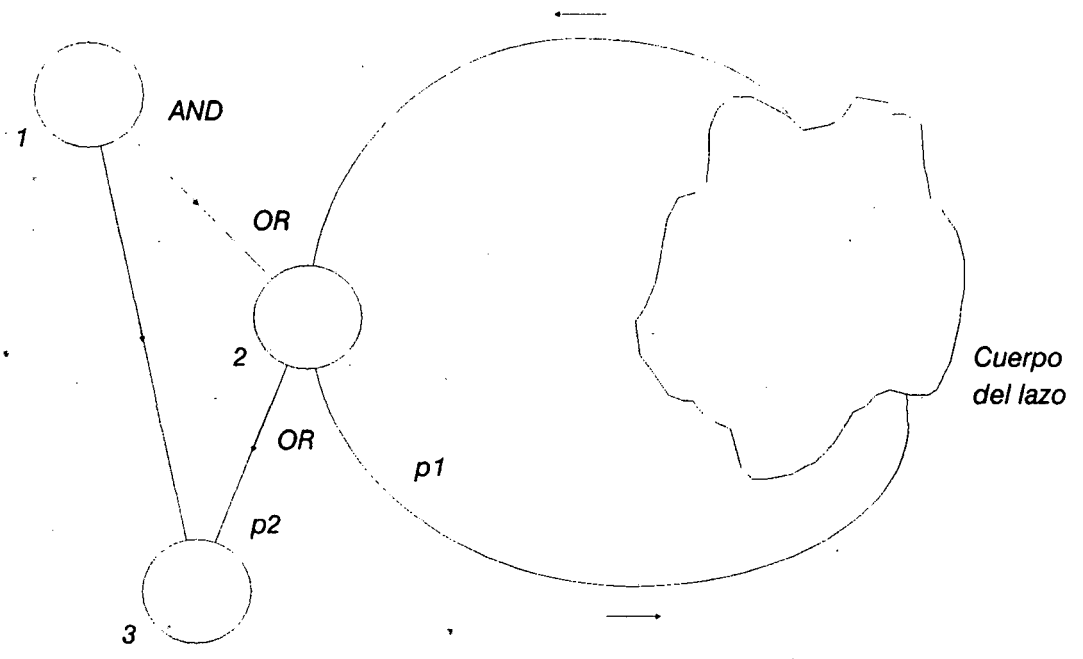


figura 3.7. grafo cíclico

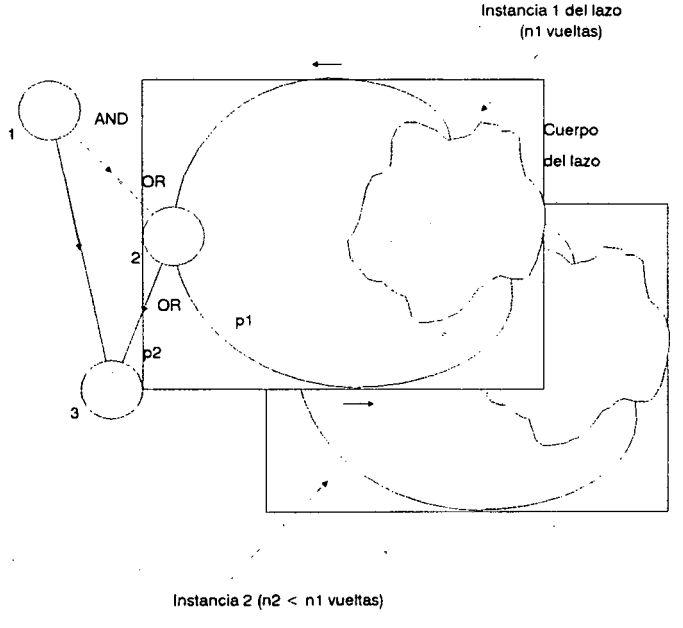


figura 3.8. Instancias paralelas de un lazo

La solución que proponemos supone generar un nuevo "espacio de tag" para cada lazo o procedimiento reentrante. El método se expone en la figura 3.9. El nodo clase "call" genera un nuevo tag "fresco" que es utilizado por cada instancia del lazo. Al salir, un nodo tipo "return" elimina el tag creado anteriormente y recupera el espacio de tags anterior al call. De esta forma se preserva la secuencia correcta de tokens a la entrada del nodo 3 en la figura 3.7, sin limitar en modo alguno el paralelismo del algoritmo.

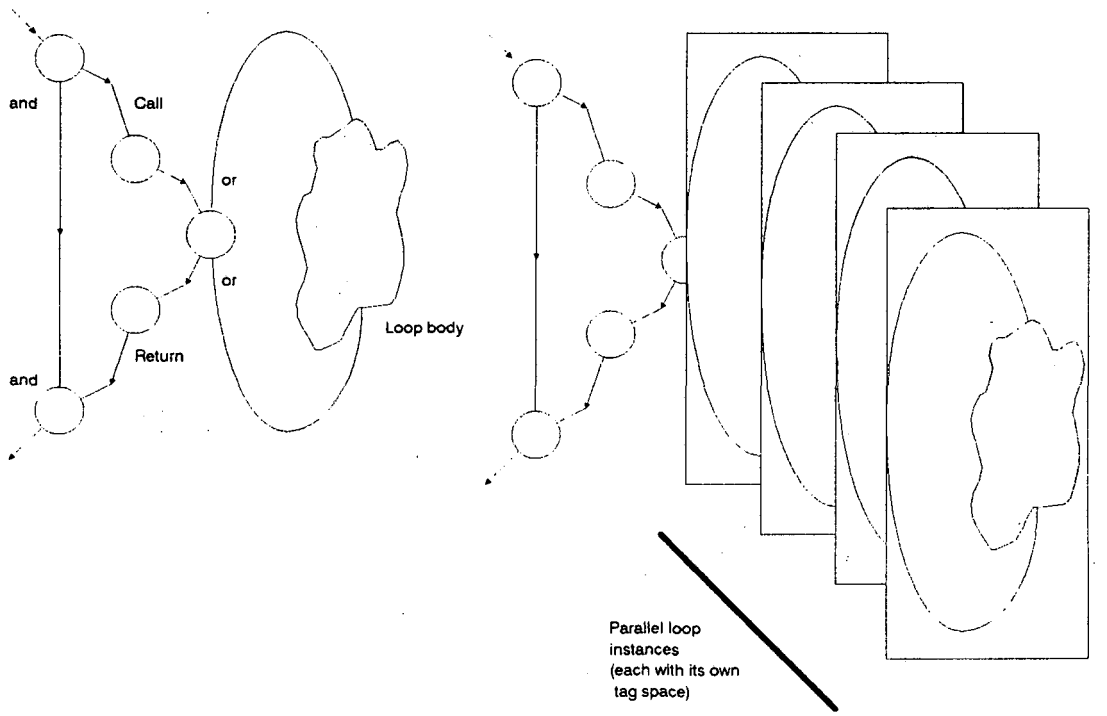


figura 3.9. Uso de Call y Return.

Es interesante comentar que esta solución es enteramente similar a la generación de un nuevo espacio de stack en las rutinas reentrantes de los sistemas monoprocesadores.

Conviene detenerse en este punto en el significado real del concepto de token. Como hemos visto, el token es utilizado solamente para decidir la habilitación de un nodo para ser ejecutado. En ningun momento hemos hecho consideraciones sobre su naturaleza. Es una decisión enteramente librada al usuario el interpretar un token como una señal de secuenciamiento o control (algo así como un contador de programa), o meramente como un dato. Como siempre, esta estrategia, si bien quita detalle al modelo, le otorga gran versatilidad.

3.1.3.Algunos Ejemplos de representación.

En esta sección describiremos algunos ejemplos típicos de estructuras paralelas de programación, y detallaremos su modelado mediante un WBG. En primer lugar nos referiremos a los lazos de Livermore, y luego a los procedimientos "divide and conquer". Los ejemplos seleccionados permiten poner en relieve las posibilidades del formalismo propuesto y ponen en evidencia sus características de modelización dinámica.

3.1.3.1.Los lazos de Livermore.

Los lazos de Livermore (Livermore loops benchmark, LLB) [46] son un conjunto de fragmentos de código escrito en Fortran, denominados genéricamente "Kernels". Cada kernel es pequeño (8 líneas en promedio), y contiene por lo menos un lazo. Estos han sido diseñados en el Lawrence Livermore National Laboratory (LLNL) para representar estructuras de dependencia clásicas. Aunque inicialmente desarrollados con fines internos, han ganado rápidamente popularidad como un medio para medir la performance de ordenadores paralelos. En esta sección analizaremos la estructura de estos lazos, y veremos su representación como WBG.

3.1.3.2.Tipos de dependencias.

Aunque los LLB son 24, sólo presentan cinco clases diferentes de dependencias:

-Independiente.
-Arbol.
-Recurrencia lineal.
-Equivalence set.
-Secuencial.

La estructura independiente es el caso típico del FORALL. Cada iteración es independiente de las otras. Un ejemplo de esta estructura es la suma de dos vectores:

```
FORALL K IN 1,N DO 10
10 A(K) = B(K) + C(K)
```

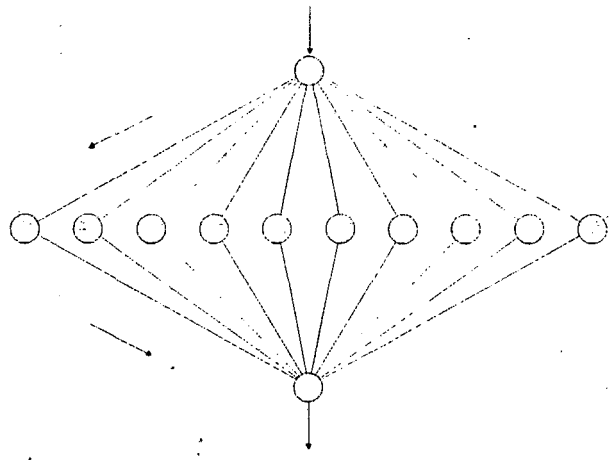
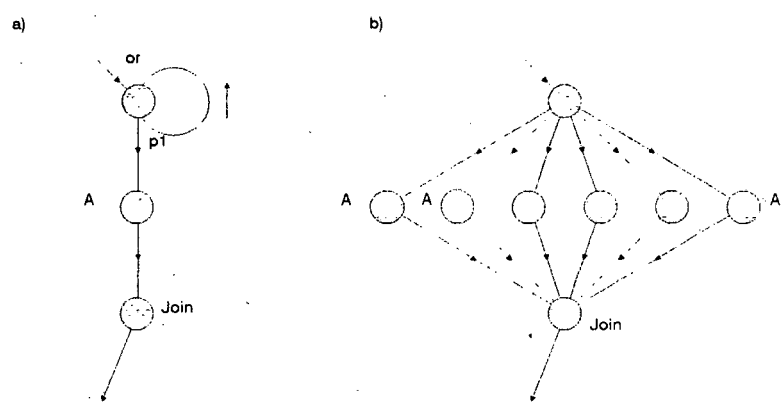


figura 3.10. Lazo independiente (forall estático).

El WBG correspondiente puede apreciarse en la figura 3.10. Aquí el volumen de comunicaciones y cómputo sería idéntico para todos los nodos del lazo.

Esta forma de expresar el forall sólo es válida cuando N es conocido en tiempo de compilación. Si no se conoce, la forma correcta de representarlo mediante un WBG se expone en la figura 3.11 (a). El nodo de entrada, con un volumen de cómputo nulo (para no consumir tiempo de simulación) genera tokens en sus salidas con probabilidades p1 y (1-p1). El lazo de realimentación hace que, para cada token que ingresa del exterior, se generen múltiples nodos de salida, que a su vez activan copias del nodo A. Ajustando, pues, el valor de p1 puede controlarse el número promedio de instancias del nodo A generadas. La figura 3.11 (b) ilustra una posible traza de ejecución.



figra 3.11. forall dinámico

Esta estrategia para construir la estructura hace necesaria la definición de una nueva "clase" de nodo, que denominamos Join. La particularidad de esta clase es que no se conoce "a priori" la cantidad de arcos de entrada que posee, al ignorar el número de copias simultáneas del nodo A que serán producidas durante la ejecución. La política de activación del nodo Join es, obviamente, "and". La implementación de esta clase se detallará en el capítulo 4.

La segunda estructura (árbol), no aparece como tal en la codificación original de los lazos de Livermore. Sin embargo puede obtenerse mediante transformaciones sobre alguno de los kernels. Por ejemplo, considérese el siguiente fragmento:

```
Q = 0
DO 10 K = 1,10
10 Q = Q + X(K)
```

La salida de cada iteración es usada como entrada a la próxima. Es, como veremos mas adelante, una dependencia estrictamente secuencial. Sin embargo, considerando que la operación de suma es asociativa, podríamos reescribir el lazo como

$$Q = ((((((A+B)+C)+D)+E)+F)+G)+H)+I)+J$$

donde llamamos A a X(1), B a X(2), etc. Esto es equivalente a

$$Q = ((A+B)+(C+D))+((E+F)+(G+H))+(I+J)$$

que posee la dependencia en árbol ilustrada en la figura 3.12. Obviamente las dos estructuras son semánticamente equivalentes, al igual que sus combinaciones. Si el número de sumas es mucho mayor que el de procesadores, cada procesador usará un algoritmo secuencial para hacer las sumas parciales, y la suma total se efectuará con una estructura árbol. Ahora bien, la forma en que se divide el problema en secciones secuenciales y no secuenciales está fuertemente influenciada por las características de la arquitectura. Por ejemplo, en estructuras tipo "link", tal como los hipercubos de la primera generación, las comunicaciones están fuertemente penalizadas. Este tipo de topologias propician el agrupamiento de tareas en bloques gran-

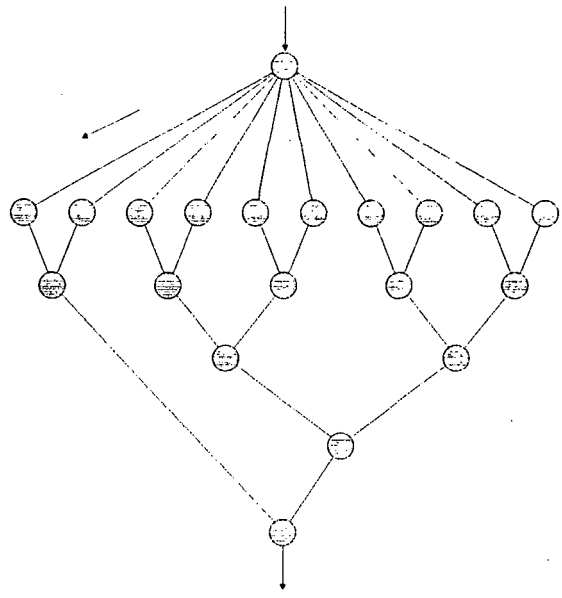


figura 3.12. dependencia en árbol

des no frecuentemente comunicados. En estos casos, la simulación puede proveer una forma sencilla de evaluar la performance de las diferentes alternativas.

La tercera estructura es la recurrencia lineal. Siguiendo con el ejemplo anterior, supongamos ahora que necesitamos todas las sumas parciales A de un vector B, desde el comienzo del mismo hasta la posición 8. El código secuencial sería

```
A[0] : = B[0];
FOR I : = 1 TO 7 DO A[I] : = B[I] + A[I-1]
```

donde para calcular el valor i-ésimo necesitamos todos los valores anteriores. Este código puede ser transfomado de la siguiente manera:

```
FOR J : = 1 TO 3 { log 8 pasos. Inicialmente A = B }
        FORALL I IN 0,7 DO BEGIN
                N1 : = I MOD 2**J;
                N2 : = 2**(J-1);
                IF N1 > = N2 THEN B[I] : = B[I] + B[I-(I MOD N2)-1];
        END;
END;
```

que posee una estructura de la forma expuesta en la figura 3.13. Obsérvese que para n datos de entrada, el algoritmo realiza el trabajo en log n pasos, y en cada paso hay n/2 procesadores ocupados.

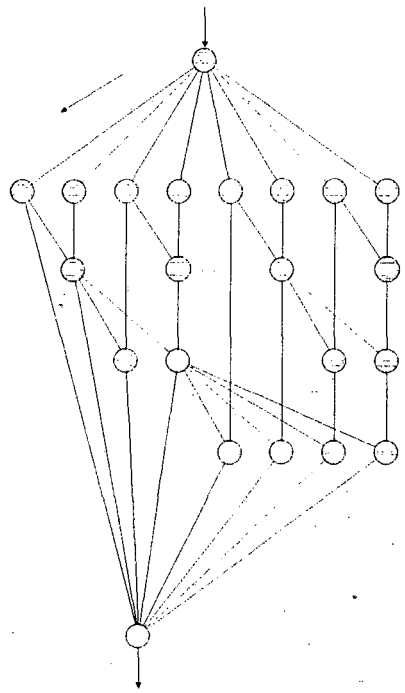


figura 3.13. recurrencia lineal

La siguiente clase de dependencia se denomina "equivalence set". Las estructuras arriba citadas son estáticas, en el sentido que no dependen del valor de los datos. En esta clase, por el contrario, cada iteración se utiliza para determinar un patrón de acceso a los datos (en un array, por ejemplo, el índice se calcularía en función de los datos actuales). El dato obtenido es luego combinado con el resto de los miembros del conjunto. Esto es, la topología exacta del algoritmo sólo puede conocerse en tiempo de ejecución. Su representación según un WBG y una posible traza de ejecución se aprecian en la figura 3.14.

La última clase es la secuencial. En esencia es una estructura del tipo

IF X(K-1) < 0

THEN X(K) = f(X(K-1))

ELSE X(K) = g(X(K-1))

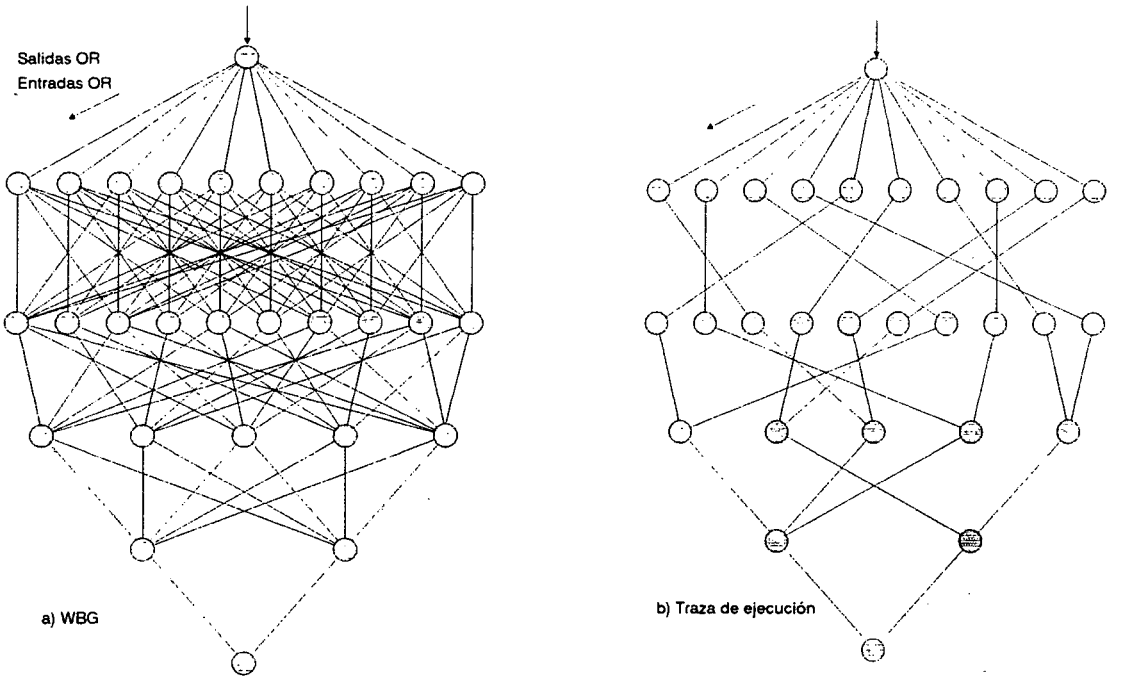


figura 3.14. equivalence set

es decir, la función (f o g) es aplicada a x dependiendo del valor computado para x(k-1). Estas estructuras presentan un WBG similar al ilustrado en la figura 3.15. En general, este tipo de recurrencia no lineal no puede ser transformada para obtener mayor paralelismo.

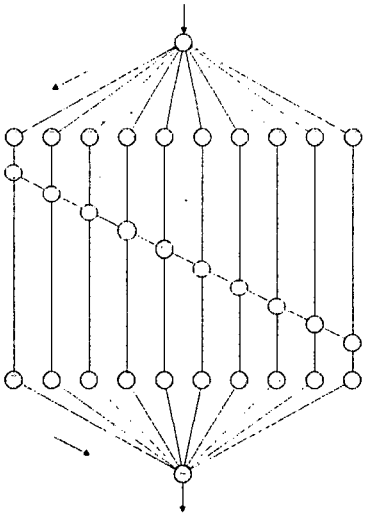


figura 3.15. dependencia secuencial

3.1.3.3.Procedimientos del tipo "divide and conquer".

La estrategia de "dividir para conquistar" consiste esencialmente, en una primera fase de partición del problema a resolver en subproblemas (usualmente en forma recursiva), seguida por una fase de "unificación" de los resultados parciales para llegar a la solución final[12]. Un ejemplo clásico es Quicksort.

Una característica importante de este tipo de algoritmos es que se desconoce a priori la profundidad del proceso de partición. El modelo correspondiente debe reflejar este hecho. Obsérvese que una ejecución eficiente de este tipo de algoritmos en un multiprocesador obliga a una estrategia dinámica de asignación.

La representación del "divide and conquer" mediante un WBG se observa en la figura 3.16 (a). Como el grafo presenta vueltas hacia atrás deben incluirse los nodos Call y Return, como discutimos anteriormente. Básicamente tenemos un nodo que representa el código del procedimiento, seguido por un nodo que modela la evaluación de la condición de finalización. Una salida del mismo (tipo or con probabilidades p1 y p2) se dirige a un nodo que, al bifurcarse , genera dos nuevas instancias del nodo original. La otra salida se dirige a un nodo tipo Join, que modela el proceso de unificación. La profundidad del árbol se controla ajustando los valores de p1 y p2.La traza típica de este grafo puede verse en la figura 3.16 (b).

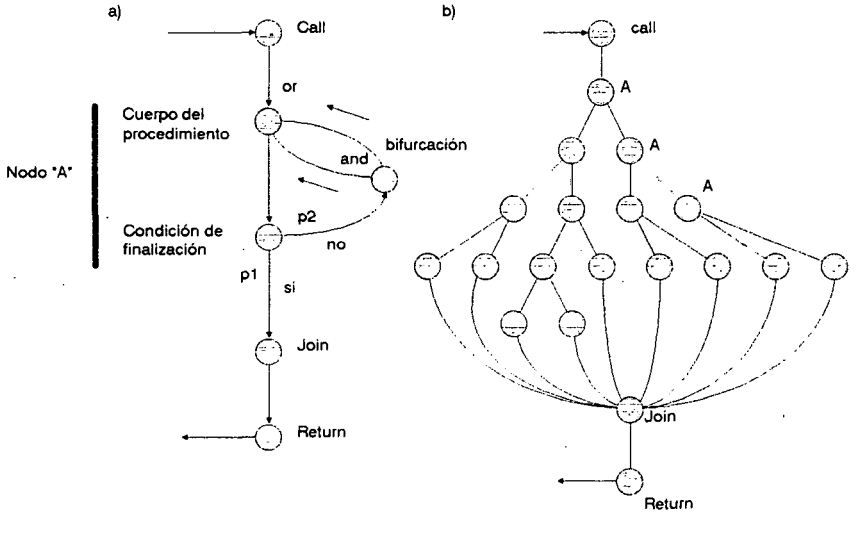


figura 3.16. procedimiento "divide and conquer"

3.2.Estructuras de interconexión.

3.2.1.Modelos utilizados en la literatura.

Resulta interesante notar el relativamente escaso interés que ha despertado el modelado de estructuras de interconexión en forma general. La mayor parte del trabajo se ha realizado para arquitecturas específicas, usualmente a fin de probar los efectos de modificaciones sobre una estructura base. Como en el caso de algoritmos, la estrategia escogida depende fuertemente de la aplicación.

3.2.1.1.Lenguajes de especificación de hardware.

Con el advenimiento de herramientas automáticas para el diseño e implementacion de circuitos integrados de alta escala de integración (VLSI), un interesante esfuerzo se ha llevado a cabo en el desarrollo de lenguajes para la especificación de circuitos digitales (figura 3.17) [31,33,34]. El objetivo aqui es desarrollar una especificación tal que permita probar formalmente que un diseño es correcto, de forma tal que el mismo puede hacerse con un nivel mayor de abstracción, ya que los detalles de implementación son resueltos por el sistema de CAD. Esta es un área de gran efervescencia actualmente, donde se han producido enormes avances en relativamente poco tiempo. Podríamos decir en este caso que el grado de detalle de la descripción es alto, ya que trabajamos a nivel de transferencias entre registros y operaciones lógicas.

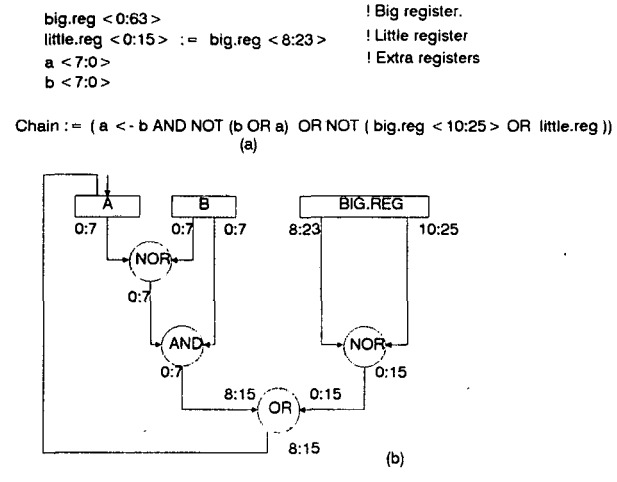


figura 3.17. Lenguajes de especificación (ISP)

Un sistema típico de diseño de arquitecturas basado en HDL's puede apreciarse en la figura 3.18. Su principal característica es la posibilidad de compilar "descripciones" de módulos hardware y almacenarlos en una base de datos relacional. Las descripciones pueden hacerse en tres niveles : estructural (netlist), de flujo de datos, o como una "caja negra". El sistema es jerárquico en el sentido que un módulo puede definirse como una combinación de subsistemas elementales. Un "linker" reúne los módulos en una especificación final, usualmente verificada mediante simulación.

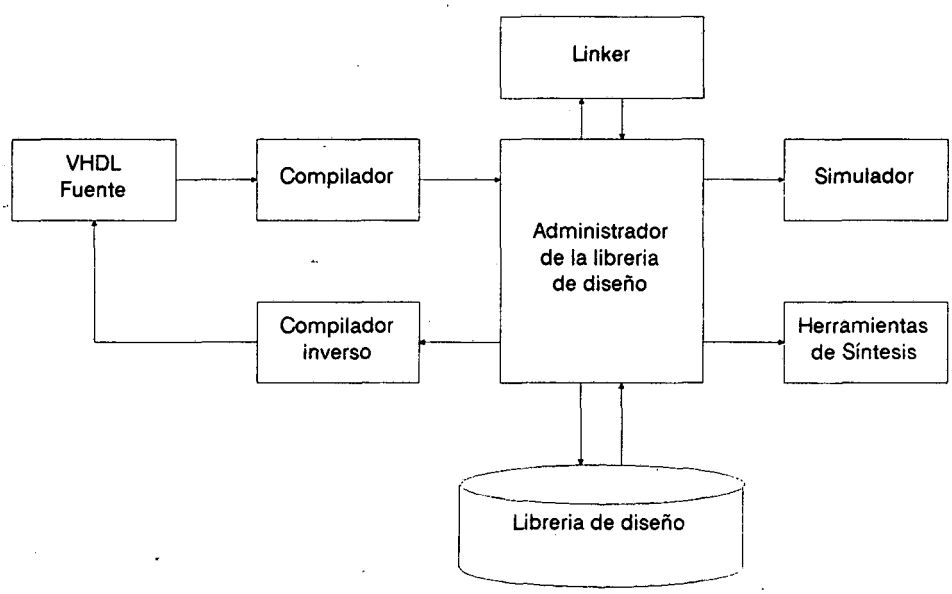


figura 3.18. Sistema de desarrollo VHDL

3.2.1.2.Modelo de colas.

Esta es la herramienta tradicionalmente utilizada para el modelado de sistemas operativos [47]. En este caso, el sistema se especifica a un muy alto nivel, y el software se modela como simples retardos, eventos que ingresan a una cola con cierta probabilidad.

Un modelo de colas es una colección de "estaciones de servicio" (dispositivos, en un sistema de cómputo), en la cual los usuarios (tareas) pasan de una estación a otra para satisfacer sus requerimientos de atención. La red es cerrada si el número de tareas permanece fijo. Un ejemplo puede apreciarse en la fig 3.19. Las flechas indican los posibles caminos entre

dispositivos. Debe existir al menos una flecha abandonando cada dispositivo. Las tareas se mueven en forma instantánea de una estación a la otra; cuando arriban a un dispositivo libre comienzan inmediatamente a ejecutarse. En caso contrario, las tareas entran en una cola de espera, donde permanecen hasta que son atendidas.

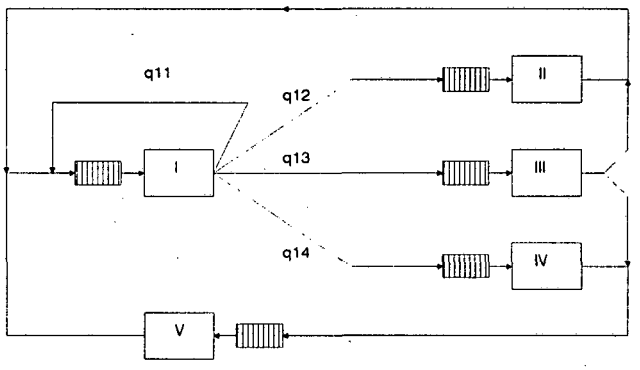


figura 3.19. Modelo de colas

El estado del sistema en un dado momento, es un vector con el número de tareas pendientes en cada cola. El tiempo de servicio de cada tarea es un valor aleatorio, al igual que el camino que recorren dentro de la red. Por lo tanto, de este tipo de modelos sólo puede obtenerse un descripción probabilística del comportamiento del sistema. Esta descripción cuantifica la probabilidad de que se esté en cierto estado en algun instante de tiempo. En la figura 3.19, qij denota la probabilidad de que una tarea se dirija al dispositivo j luego de ser procesada por el dispositivo i. El tiempo de atención se expresa mediante funciones que especifican la probabilidad de que el tiempo de servicio Ti de una tarea en un dispositivo i no exceda un dado t:

$$\mathbf{F_i\,(t) = Pr\,(T_i \le t)\,,\, 0 \le t}$$

Fi se denomina habitualmente función de distribución del tiempo de servicio.

3.2.1.3.Grafos de interconexión.

La estrategia mas utilizada para la especificación de estructuras de interconexión en el estudio de políticas de asignación en multiprocesadores,es el empleo de alguna forma de grafo no dirigido similar a los ya vistos al analizar los algoritmos. El tipo de información requerida al grafo nuevamente depende del tipo de estrategia de scheduling utilizada. El espectro observado en la literatura abarca desde estructuras homogéneas (es decir, con todos los elementos de procesamiento iguales) totalmente interconectadas, hasta sistemas heterogéneos y estructuras de interconexión con caminos alternativos y/o redundantes.

3.2.2.Modelo propuesto.

El formalismo adoptado es enteramente similar al propuesto en el caso de algoritmos (grafos dirigidos ponderados). Cada nodo posee un parámetro que establece su rendimiento relativo al resto de los elementos de su clase. La "clase" del nodo especifica un tipo especial de estructura de la arquitectura en estudio (procesador, memoria, switch, etc.). El comportamiento de cada nodo está especificado por los procedimientos correspondientes a cada clase, que manipulan su estado interno. No se hace referencia alguna a la implementación práctica del nodo. Así pues, definimos un nodo de la arquitectura como una tupla

n : < Id, Cp, Class, [inputs], [outputs] >

donde:
Id : identificación del nodo.
Cp : Capacidad o performance relativa.
Class : clase de nodo.
[inputs] : conjunto de nodos de entrada.
[outputs] : conjunto de nodos de salida.

Así, por ejemplo, para la clase Procesador, la performance relativa modela la potencia de cómputo del nodo; para la clase Memoria, el tiempo de acceso y para el Switch, el tiempo de conmutación y tránsito. Los arcos poseen, además, una capacidad de almacenamiento ("buffer") de tokens, definible por el usuario y organizado como un FIFO. Este parámetro es importante al analizar el tráfico en una topología dada, o al diseñar una política de ruteo de

mensajes, ya que la posible congestión de un link de la arquitectura puede decidir la utilización de un camino alternativo. Un arco o link es una tupla:

l : < Ids, Idd, Cc, Bs >

donde:
Ids: identificación del nodo fuente.
Idd: identificación del nodo destino.
Cc: performance relativa de comunicación.
Bs: Tamaño del buffer.

La figura 3.20 ilustra algunos esquemas de interconexión clásicos: una malla de 16 nodos (a), una estructura tipo bus (b) y una red butterfly (c).

Esta estrategia de modelado es, ciertamente, muy general. De hecho, en muchos casos, lo es en demasía. Pero es importante resaltar que el objetivo que nos guía no es el diseño detallado de una máquina en particular (para lo cual este modelo es indudablemente insuficiente), sino que nos interesa poner de manifiesto cuáles son los factores que determinan macroscópicamente el comportamiento de una arquitectura. En tal sentido este esquema de modelado permite tomar una decisión "temprana" sobre el conjunto *algoritmo - estructura de interconexión - políticas de sistema* mas adecuado para una aplicación específica. Estos factores no son independientes entre sí, y su interrelación es a menudo desconocida.

a)

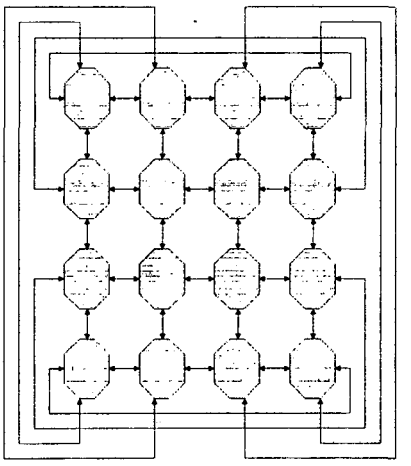
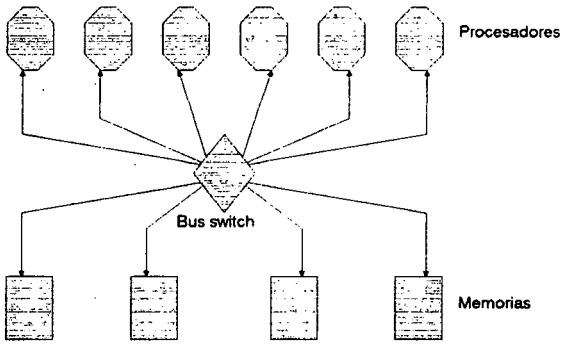


figura 3.20 a). Ejemplos de representación.

b)



c)

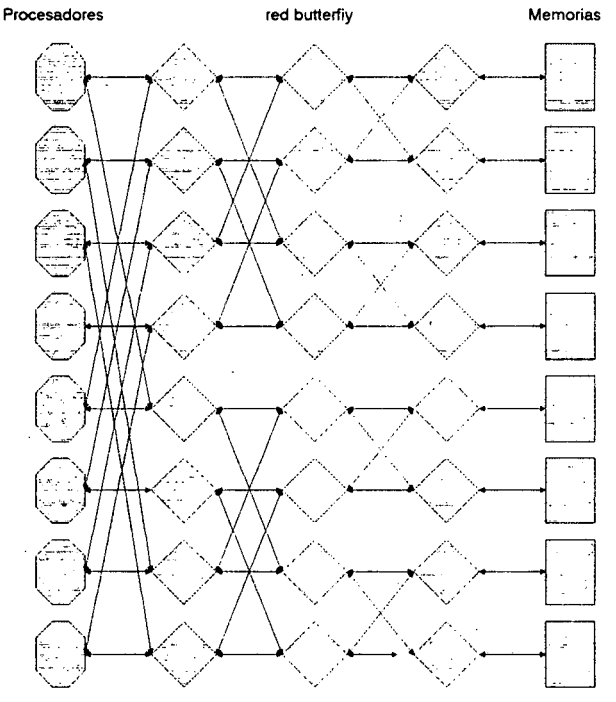


figura 3.20. Ejemplos de representación

3.3.Políticas de Asignación de tareas.

El propósito de la política de asignación de tareas a un conjunto de procesadores es, en última instancia, reducir el tiempo de procesamiento total. Esto puede lograrse maximizando la utilización de los recursos, al tiempo que se minimiza la comunicación entre procesadores [10,13-18, 48-52].

Varios problemas deben resolverse para alcanzar este objetivo. Un comportamiento típico de los sistemas paralelos de cómputo es el denominado "efecto de saturación" (fig. 3.21), que prevee una disminución en el rendimiento del sistema con un aumento en el número de procesadores. Las posibles causas de este efecto son los retardos debidos a las comunicaciones, cargas desbalanceadas en los procesadores, relaciones de dependencia entre tareas y falta de suficiente paralelismo en el algoritmo como para mantener la máquina ocupada todo el tiempo a su máxima potencia.

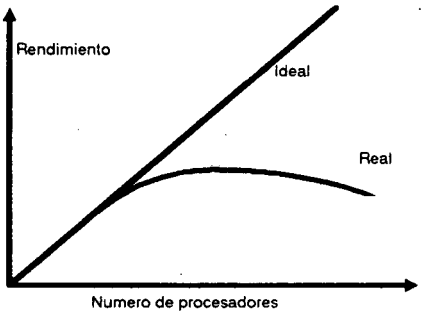


figura 3.21. el efecto de saturación

Observemos además, que los objetivos arriba citados imponen requerimientos conflictivos al sistema de cómputo:

a) Mientras la minimización de las comunicaciones tiende a asignar todas las tareas a un sólo procesador, el objetivo del balance de carga es distribuir las tareas en forma homogénea entre todos.

b) Las consideraciones de tiempo real (mínimo tiempo de respuesta), suponen utilizar el máximo número de procesadores posible, pero esto es imposible debido a las inevitables relaciones de dependencia entre tareas.

c) La curva de la figura 3.19 sugiere que, a partir de cierto umbral, la eficiencia decrece con el número de procesadores. Esto es debido a que, si existe un elevado costo de comunicaciones, es mas eficiente agrupar las tareas en grupos de mayor granularidad que distribuir la carga homogéneamente.

La figura 3.22 muestra un modelo general del proceso de asignación. Los elementos claves en esta figura son la cola de tareas a procesar y el scheduler S. Aquí se consideran las tareas como módulos indivisibles. La figura 3.23 ilustra la política que produce el mejor balance de carga (lo que puede implicar un alto costo de comunicación si las tareas están fuertemente acopladas), y la 3.24 la estrategia que minimiza dicho costo.

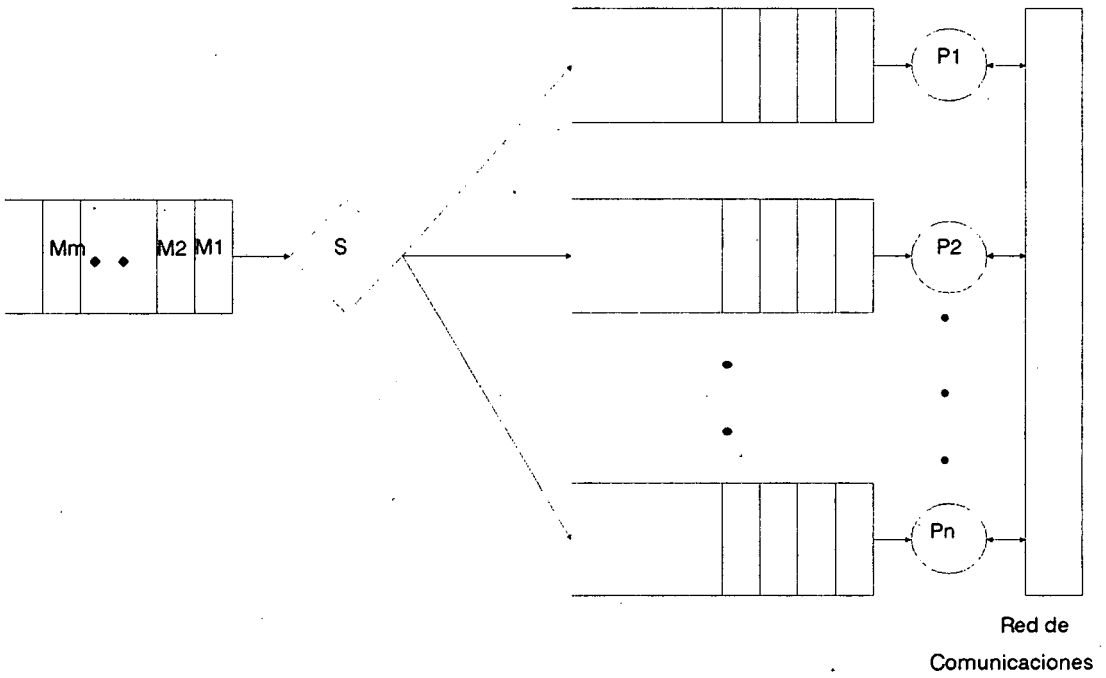


figura 3.22. modelo del proceso de asignación

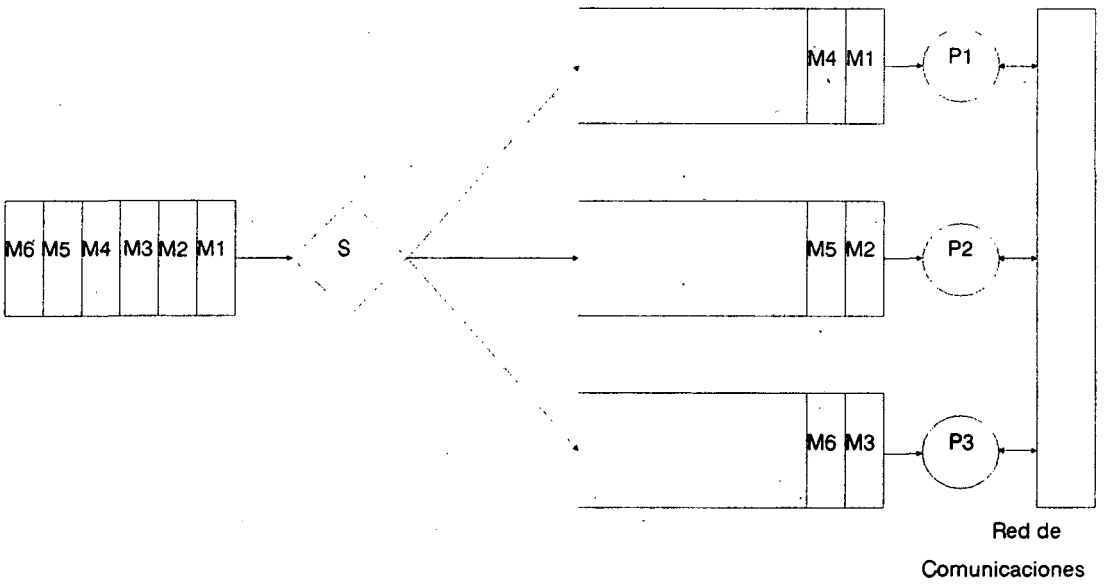


figura 3.23. estrategia para carga balanceada

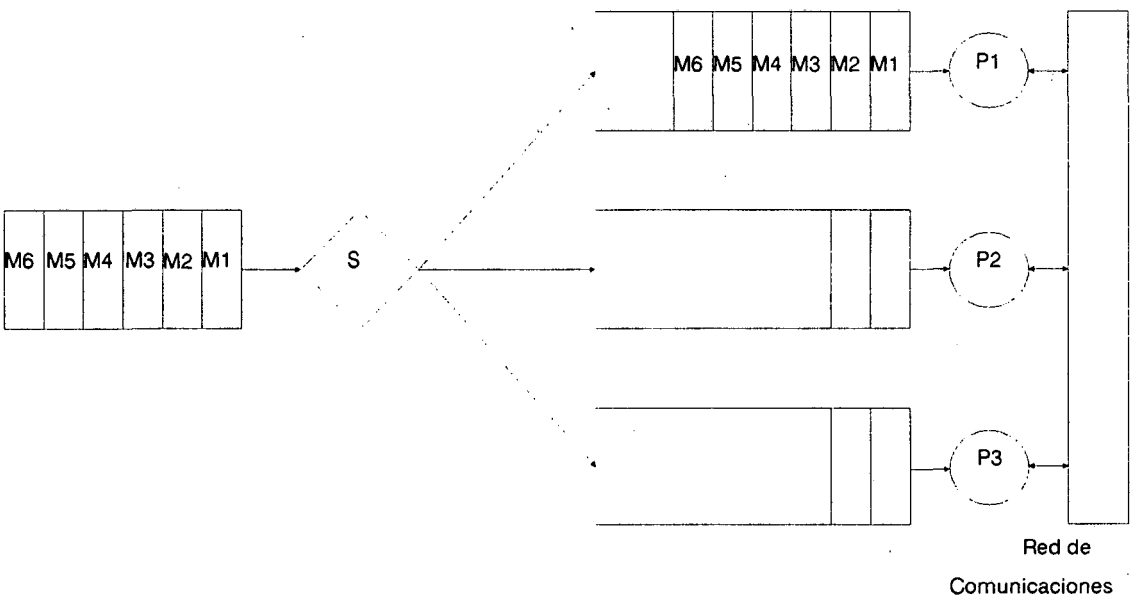


figura 3.24. mínimo costo de comunicación

Obviamente, es imposible satisfacer todos los requerimientos simultáneamente. Diferentes técnicas se han utilizado para resolver el problema de asignación "offline"; podemos clasificarlas en forma general dentro de dos categorías:

-Métodos basados en teoría de grafos y optimización matemática.
-Métodos heuristicos.

3.3.1.Métodos basados en teoría de grafos

Estas estrategias utilizan un algoritmo de corte mínimo para particionar un grafo de tareas no dirigido, minimizando el costo de comunicación (IPC: Inter Processor communication). Se supone conocido el volumen de comunicaciones entre tareas (IMC: Inter Module communication). Un costo cero indica que las tareas no se comunican, mientras que un valor infinito indica que las dos tareas deben asignarse al mismo procesador. Dos tareas asignadas a un mismo procesador tienen un IPC nulo.

Para representar la asignación de tareas a procesadores se define una matriz X tal que

$x_{i,k} = 1$ si la tarea M_i está asignada al procesador P_k

0 si no lo está

El costo de procesamiento esta dado por la matriz Q:

$q = \{q_{i,k}\}, i = 1,...,m, k = 1,...n$

donde $q_{i,k}$ representa el costo de procesamiento del módulo M_i en el procesador P_k.

Sea $c_{i,j}$ el volumen de comunicaciones entre las tareas M_i y M_j. El costo total de ejecución de un grafo puede ser definido como una función de la asignación X:

$$\text{Cost}(X) = \Sigma_k \Sigma_i \{q_{i,k}\, x_{i,k} + \Sigma_l \Sigma_j c_{i,j}\, x_{i,k}\, x_{j,l}\} \qquad (1)$$

donde el primer término representa el costo de procesamiento, y el segundo el costo de comunicación entre módulos no corresidentes. El algoritmo de corte mínimo busca un X tal que la ecuación se minimice.

Este método tiene serias limitaciones. Con un número de procesadores n, se requiere un algoritmo de corte mínimo n- dimensional que, a medida que n crece, se hace computacionalmente intratable.

Una segunda limitación del método es que no provee ninguna forma de balancear la carga entre los procesadores, ni tiene en cuenta las dependencias de datos entre módulos. El balance de carga puede obtenerse mediante la imposición de restricciones a la ecuación (1). Estas restricciones tienen en cuenta factores como memoria limitada en los procesadores, velocidad de los mismos y consideraciones de tiempo real.

La restricción de memoria se expresa como

$$\Sigma\, s_i\, x_{i,k} \leq R_k,\ k = 1...n \qquad (2)$$

donde s_i representa la cantidad de memoria requerida por el módulo i y R_k representa capacidad de memoria del procesador P_k. Esta ecuación simplemente establece que la suma de la memoria requerida por los módulos asignados a P_k no debe exceder su capacidad.

La limitación de tiempo real puede establecerse como

$$\Sigma\, u_i\, x_{i,k} \leq T_k,\ k = 1..n \qquad (3)$$

donde u_i representa el tiempo de procesamiento requerido por el módulo M_i, y T_k representa el tiempo límite para procesar los módulos que residen en P_k para un dado grafo. Ahora la ecuación no lineal (1) debe ser resuelta sujeta a las restricciones (2) y (3). Mediante la adición de mas restricciones es posible linealizar (1), acelerando así su resolución.

3.3.1.1.Métodos heurísticos.

A diferencia de los citados anteriormente, los métodos heurísticos no proveen una solución óptima, pero son mas simples y computacionalmente eficientes. De hecho, para problemas complejos son las únicas estrategias utilizables.

La idea fundamental detrás de los procedimientos heurísticos de asignación es reducir el espacio de búsqueda de la solución de la ecuación (1). Conocidos algoritmos de tipo "simulated annealing" (ver capítulo 6), así como técnicas de programación dinámica o basadas en el camino crítico pueden utilizarse para guiar la búsqueda.

Tanto las estrategias basadas en teoría de grafos como en técnicas de optimización suponen un conocimiento bastante detallado a priori de los tiempos de cómputo de las tareas y su volumen de comunicaciones. Sin embargo, en un sistema real estos datos suelen ser desconocidos. No siempre es posible realizar una "ejecución de prueba", para estimar costos. Mas aún, veíamos en secciones anteriores que en función de los datos el grafo del algoritmo puede variar sustancialmente. Se imponen entonces técnicas de asignación dinámicas, esto es, que operen durante la ejecución del algoritmo. Este es un tema escasamente tratado en la literatura. En esencia, todos los métodos se basan en la existencia, en alguna memoria globalmente accesible, de una o varias colas de tareas listas para su ejecución, que los procesadores consultan cuando están libres.

Por supuesto, existen también estrategias híbridas, que realizan una asignación estática off-line y utilizan, durante la ejecución, algún algoritmo para el balance de carga en tiempo real. Una política inteligente en ese sentido debe tener en cuenta entonces, el tiempo utilizado para migrar el código de la tarea de un procesador al otro, así como la forma de medir en tiempo real la carga de cada procesador.

3.3.2.Solución adoptada.

En los puntos anteriores hemos visto la diversidad de propuestas existentes para el problema de asignación de tareas. Esto hace difícil la especificación de una estrategia de modelado general que cubra todos los casos. Por lo tanto, hemos adoptado la solución de especificar el scheduler mediante un algoritmo escrito en un lenguaje de alto nivel. Esta rutina

debe concatenarse con las rutinas de simulación antes de efectuar una sesión. De esta forma, se permite la creación de una "biblioteca" paramétrica de procedimientos de asignación, con lo que se preserva la flexibilidad de modelado.

Los parámetros de entrada a la rutina de asignación son, si se trata de algoritmos estáticos, los grafos del algoritmo (WBG) y de la arquitectura. Cuando la política de asignación es dinámica, debe agregarse el estado de ocupación de la arquitectura, para poder estimar el balance de carga, y la cola de tareas no asignadas pero listas para su ejecución. Mas adelante, cuando hagamos un esbozo del sistema de simulación, se hará una definición mas precisa de las características de este programa.

3.4.Ruteo de mensajes en la arquitectura.

Los nodos del WBG se comunican mediante el paso de "mensajes", modelados como tokens que fluyen por el grafo. Ahora bien, al ser asignados a una arquitectura específica, dichos tokens deben ser almacenados en una zona de memoria común, en el caso de sistemas de memoria compartida, o transmitidos a través de los links de la máquina, en sistemas débilmente acoplados. Cuando la arquitectura no está totalmente interconectada, la comunicación entre dos nodos no vinculados directamente debe realizarse a través de procesadores intermedios. La elección de la ruta de comunicaciones mas adecuada en un instante dado es lo que denominamos "política de ruteo de mensajes" de un sistema.

Se ha desarrollado una amplia experiencia en este tipo de operaciones debido a la gran difusión de los sistemas de redes locales y WAN (wide-area-networks). Mucha de esta experiencia ha sido utilizada en el mejoramiento de las comunicaciones en sistemas paralelos de cómputo.

Las estrategias y técnicas para el manejo de mensajes son los dos factores importantes al determinar la eficiencia de las comunicaciones interprocesadores. Fundamentalmente existen dos parámetros en juego en el diseño de un algoritmo de ruteo: tiempo y tráfico. El tiempo es normalmente medido en pasos de ruteo, donde cada paso corresponde al tiempo que se necesita para enviar una unidad de información (token) desde un nodo a alguno de sus vecinos (nodos directamente conectados). El tráfico se cuantifica por el número de mensajes

que atraviesan los links utilizados en el trayecto fuente-destino. Obsérvese que el número de pasos requeridos para realizar una comunicación es como mínimo igual a la distancia fuente-destino (número mínimo de "hops", o estaciones intermedias).

Obviamente es deseable una estrategia tal que minimice los pasos de ruteo y el tráfico. Sin embargo, los dos factores no son independientes, y es usual tener que establecer una solución de compromiso.

En forma general, podemos clasificar las comunicaciones en un multiprocesador en tres categorías [53]:

Uno-a-uno (unicast) es la forma mas primitiva de comunicación punto a punto.
Uno-a-todos (broadcast) supone el envío del mismo mensaje a todos los nodos de la arquitectura.
Uno-a-varios (multicast) involucra el envío de un mismo mensaje a un grupo de procesadores.

Un importante factor a medir para evaluar la performance de un sistema paralelo de cómputo es la relación computación/comunicación. Un valor bajo de esta relación significa que los procesadores de la máquina están utilizando la mayor parte del tiempo para comunicarse. Por lo tanto, un mecanismo eficiente de manejo de las comunicaciones es una necesidad vital que afecta el rendimiento de todo el sistema.

Otra propiedad deseable en una política de ruteo es la tolerancia a fallos. Esto supone la posibilidad de establecer caminos alternativos ante la detección de una falla en las comunicaciones, que posibilite que el sistema siga funcionando, aunque probablemente con una degradación en su rendimiento. Esta característica es de capital importancia en sistemas de tiempo real.

Podemos encontrar, en general, tres tipos de políticas ampliamente utilizadas. En arquitecturas muy regulares (hipercubos, mallas, etc.[54]), es posible asignar a cada procesador un número de identificación tal que el ruteo puede calcularse en forma automática. En estos esquemas de interconexión la distancia máxima es normalmente una función exacta del número de procesadores.

En estructuras no regulares se advierten dos tendencias [53], ambas heredadas de las redes locales. La primera es enviar información sobre ruteo en la misma especificación del nodo destino. La segunda consiste en utilizar una política "inteligente", donde el camino que sigue el mensaje es calculado por un "router" distribuido que administra información sobre el tráfico y establece, con algún criterio, el camino mas adecuado en cada momento.

En las secciones siguientes describiremos los métodos mas utilizados para el ruteo de mensajes en sistemas paralelos. Se analizarán los procedimientos tipo "Store and forward" y haremos un sumario de las políticas estáticas y dinámicas mas frecuentes.

3.4.1.Procedimientos tipo "store and forward".

En este tipo de políticas, cada nodo recibe el mensaje en su totalidad, almacenándolo en su memoria local, y luego decide quién será el próximo intermediario. Esta es la estrategia mas ampliamente utilizada. Si el canal está ocupado, el mensaje continúa almacenado hasta que se libere. El ruteo puede ser calculado por el propio procesador (por ejemplo, en el cubo cósmico de CalTech y en general, en todos los hipercubos de la primera generación), pero también existen sistemas con coprocesadores especializados, que operan en paralelo con el procesador central (segunda generación de hipercubos, Intel iPSC [54]).

3.4.2.Algoritmos no adaptivos.

El algoritmo mas simple (y probablemente el menos utilizado) es el denominado "flooding" [55]. Cada mensaje de entrada a un nodo es enviado por todos los links, a excepción del link por el cual llegó. Este método genera múltiples copias del mensaje, elevando considerablemente el tráfico. Además requiere algún método para anular los mensajes duplicados una vez que se ha llegado a destino. Sin embargo, su evidente robustez lo hace útil para sistemas que requieren gran confiabilidad. Además, este algoritmo siempre obtiene el camino mas corto, ya que recorre todas las posibles trayectorias en paralelo.

Una alternativa al flooding es la denominada "random walk". Aquí el mensaje se envía a un link seleccionado aleatoriamente. Si la arquitectura está densamente interconectada, random walk tiene la característica de utilizar eficientemente las trayectorias alternativas, y ser robusto.

3.4.3.Ruteo estático.

Este algoritmo [55], también denominado ruteo por directorio es uno de los mas utilizados. En él, cada nodo posee una tabla con una fila para cada posible destino. Cada fila proporciona una lista de los nodos vecinos ordenados según el número de pasos de ruteo hasta el destino (mejor elección, segunda mejor, tercera, etc.)(figura 3.25). La elección se hace generando localmente un número aleatorio. Si este número es menor que el asignado a la mejor solución, se envía por ésta. Si no es así, se prueba con la siguiente, etc. Las tablas son generadas estáticamente.

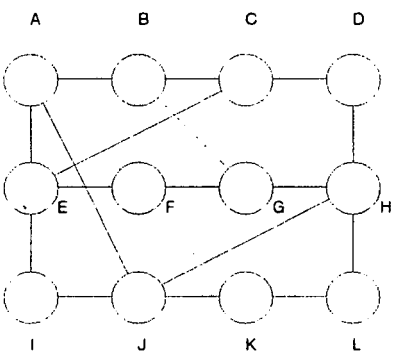


Tabla para el nodo J

Destino	eleccion1		eleccion 2		eleccion 3	
A	A	0.63	I	0.21	H	0.16
B	A	0.46	H	0.31	I	0.23
C	A	0.34	I	0.33	H	0.33
D	H	0.50	A	0.25	I	0.25
E	A	0.40	I	0.40	H	0.20
F	A	0.34	H	0.33	I	0.33
G	H	0.46	A	0.31	K	0.23
H	H	0.63	K	0.21	A	0.16
I	I	0.65	A	0.22	H	0.13
-						
K	K	0.67	H	0.22	A	0.11
L	K	0.42	H	0.42	A	0.16

figura 3.25. ruteo estático o por directorio.

3.4.4.Ruteo adaptivo.

En estos algoritmos la decisión de ruteo está influenciada por la historia reciente de la arquitectura [55]. Podemos diferenciar dos tipos de estrategias: aquellas que no suponen un conocimiento del estado de la máquina (tráfico global), y aquellas que sí lo hacen.

Un ejemplo simple de la primera estrategia es el algoritmo del "hot potato". El principal objetivo de este algoritmo es rutear el mensaje lo antes posible. El nodo mantiene una cola de mensajes para cada uno de los links de salida. Al recibir un mensaje se almacena en la cola mas corta.

Una estrategia algo mas interesante es la denominada "backward learning" [55]. Para implementearla se requiere, además de los nodos destino y fuente, un contador con los pasos de ruteo que tiene hasta el momento. Si un nodo recibe un mensaje por su link "k" de la fuente H con el contador en un valor 4, por ejemplo, sabe que el nodo H no se encuentra a mas de 4 pasos de ruteo en dirección k. Si su mejor distancia hasta H estaba estimada en mas de 4, se elige a k como el mejor ruteo hacia H, y se almacena 4 como su nueva distancia. Luego de operar un tiempo, el sistema encontrará una situación de equilibrio en la que cada nodo conocerá las distancias mínimas a todos los demás.

El algoritmo posee algunos problemas. Como los cambios siempre se hacen hacia la solución mejor, cuando un link falla o se encuentra saturado no existe mecanismo de recuperación. Esto se soluciona mediante un "factor de olvido", que periódicamente regenera el proceso de aprendizaje.

En el segundo grupo de algoritmos, se envían periódicamente mensajes con información de ruteo, tal como nuevos nodos, reconfiguración, etc [55]. Estas estrategias, si bien ampliamente utilizadas en redes locales, no encuentran aplicación en computadoras paralelas, donde las topologías y costos de los nodos son normalmente conocidas "a priori", y solo es necesario una sintonización durante la operación.

3.4.5.Solución adoptada.

En los puntos anteriores hemos visto que el espectro de posibles soluciones al problema del ruteo es amplio. Por tal motivo se ha optado para su modelado por una estrategia similar a la utilizada para el caso de las políticas de asignación, esto es, la especificación de cada estrategia en estudio mediante rutinas en lenguaje de alto nivel, que constituyen la biblioteca de procedimientos de ruteo. Los parámetros de entrada a estas rutinas son, para el caso estático, la información de los nodos fuente y destino, y del esquema de interconexión. Para el caso dinámico, se requiere además información sobre el estado de las colas de mensajes de cada link. La salida de la rutina es una lista de los pasos de ruteo que el mensaje debe recorrer para llegar a destino. Cuando veamos la especificación concreta del simulador estos parámetros serán convenientemente detallados.

3.5.Resumen.

Este capítulo analiza las diferentes soluciones propuestas para el modelado de los factores claves en la operación de los sistemas paralelos de cómputo digital: algoritmos, estructuras de interconexión, y políticas de asignación de tareas y ruteo. Para cada una de estas facetas, se propone un esquema de modelado que, conservando un nivel de abstracción razonable, permite evidenciar claramente los compromisos de diseño y los factores que afectan la performance global. Esta estrategia será utilizada en capítulos subsiguientes para la especificación del ambiente de simulación.

Capítulo 4.

El Simulador.

En el capítulo 3 se ha especificado la estrategia de modelado adoptada para los diferentes subsistemas presentes en un computador paralelo. El propósito de este capítulo es describir la implementación práctica del programa simulador. El mismo permite realizar el análisis en el dominio del tiempo del sistema, especificado como una red de módulos funcionales interconectados. Este programa acepta como entrada los grafos de programa y arquitectura, y las políticas de asignación de tareas a procesadores y ruteo de mensajes. El estado del sistema simulado es utilizado por el ambiente interactivo (analizado en el próximo capítulo) para la presentación de resultados durante la simulación.

4.1.Diagrama de bloques del simulador.

En la figura 4.1 puede apreciarse un diagrama de bloques del simulador. Su estructura es funcionalmente similar a una máquina de flujo de datos de tipo dinámico. Esto se debe a las particulares reglas de funcionamiento del WBG. Dos premisas han guiado el diseño del simulador: modularidad y eficiencia. La primera se ha obtenido mediante una precisa y rigurosa especificación funcional de cada elemento y de su interfaz con el resto de los componentes del sistema. La segunda se debe, fundamentalmente, a las características derivadas de la filosofía de modelado. A continuación explicaremos las funciones de cada uno de los bloques de la figura 4.1, a nivel de su comportamiento interno, y de su interfaz con los restantes módulos.

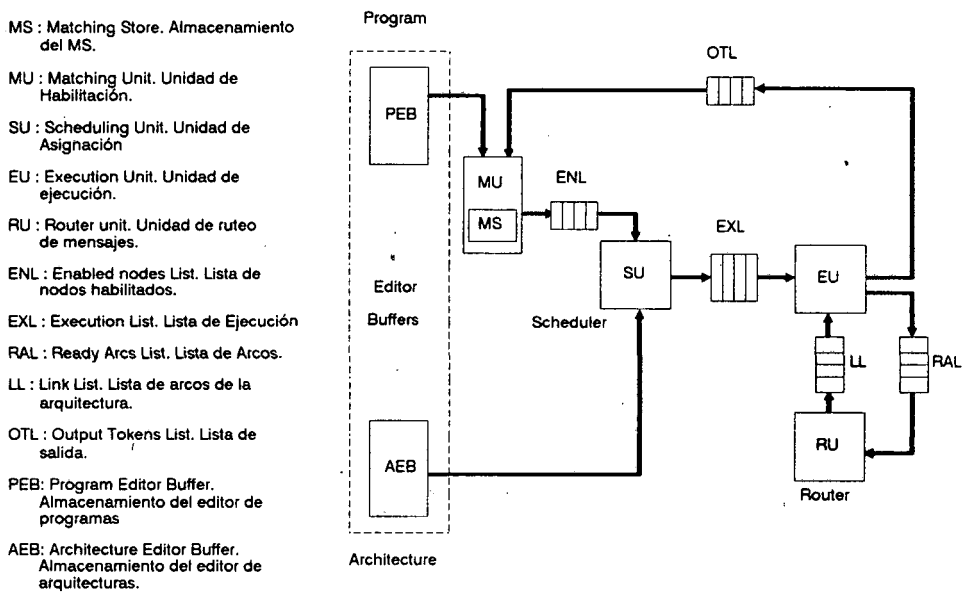


figura 4.1. Diagrama en bloques del simulador

4.1.1. Unidad de habilitación.

Este bloque (Matching Unit, MU) es el encargado de verificar la habilitación de nodos para su ejecución. La organización de la MU puede apreciarse en la figura 4.2. Su estructura de datos fundamental es el "Matching Store" (MS), organizada como una lista encadenada de nodos. Esta lista contiene los nodos parcialmente habilitados del grafo del programa. Para cada token de la lista de entrada (Output Token List, OTL), la MU inspecciona el MS, buscando el nodo destino asociado (recordar que el token posee información de los nodos fuente-destino, así como un "tag" que identifica su número de activación, capítulo 3). Esta operación es, conceptualmente, una búsqueda asociativa del elemento [idd,tag] en el MS.

Si el nodo destino es hallado, su entrada correspondiente al nodo fuente se actualiza, y se verifica la condición de habilitación, de acuerdo con la política de entrada asociada. Si el nodo no se encuentra en el MS, es leído desde el espacio de almacenamiento del Editor de Programas. Los nodos parcialmente habilitados quedan almacenados temporalmente en el MS. Si la condición de disparo se verifica, el nodo se extrae del MS y se deposita en la Ena-

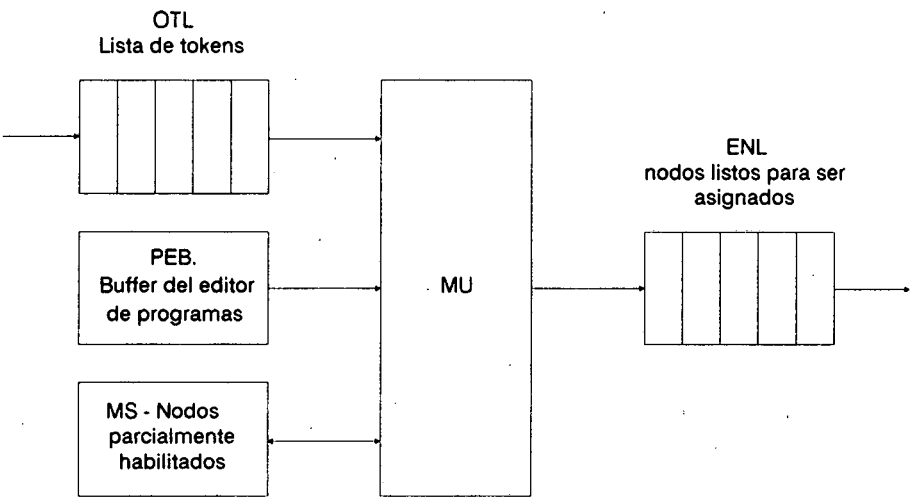


figura 4.2. Organización de la MU.

bled Node List (ENL), salida de la MU. Todos los nodos de la ENL están habilitados (son lógicamente ejecutables), y a la espera de ser asignados a algún procesador.

La MU tambien realiza las operaciones de actualización del número de ejecución del tag de cada nodo, asi como la generación de un nuevo espàcio de tags para los nodos de la clase "CALL", y la recuperación del espacio de tags anterior para los nodos de la clase "RETURN". El espacio de tags de cada nodo es administrado como un "stack" o pila, en una forma enteramente similar al "U- Interpreter" de Arvind [56].

El tratamiento de los nodos de la clase "JOIN" es un tanto mas complejo. La condición de habilitación involucra la verificación de que no existen mas instancias posibles de su nodo predecesor con el mismo espacio de tags. Esto se complica aún mas debido a la eventual presencia de lazos en el grafo. En este caso la MU debe examinar, por tanto, no sólo el MS sino todas las listas del simulador para asegurar un disparo correcto.

La figura 4.3 presenta el algoritmo fundamental de la MU. Al igual que en las máquinas de flujo de datos dinámicas, la performance de esta unidad está influenciada en gran me-

dida por la forma en que se administra el MS. El tamaño del MS varía dinámicamente y depende fuertemente de la aplicación. La presencia, en un momento dado, de gran cantidad de nodos en el MS (nodos parcialmente habilitados) indica la posibilidad de algún "cuello de botella" secuencial en el programa paralelo bajo estudio.

```
Matching_Unit::
Mientras existan tokens en la OTL hacer
        Buscar_nodo_destino_en_MS (identificación, tag);
        Si no existe entonces leer_nodo_destino (PEB));
        Actualizar_entradas;
        Verificar_condición_de_disparo;
        Si verifica entonces depositar_nodo_en_ENL;
        fin;
fin;
```

figura 4.3. Algoritmo fundamental de la MU.

4.1.2. Unidad de asignación.

Esta unidad (Scheduling Unit, SU) es la encargada de generar los pares [tarea, procesador], de acuerdo a la política de asignación especificada, la disponibilidad de tareas habilitadas (presentes en la ENL), y el estado global de la arquitectura. Su estructura básica se presenta en la figura 4.4.

Como veíamos en el capítulo anterior, existe una gran diversidad de estrategias posibles para la asignación de tareas en sistemas paralelos. En nuestro simulador, el usuario especifica su política mediante una rutina en alto nivel (Pascal en la actual implementación), o bien la extrae de la biblioteca de politicas de asignación del sistema. En el caso de estrategias de asignación estáticas (off-line), la única información necesaria para estas rutinas es la especificación de los grafos de programa y arquitectura. Esta información se encuentra en los

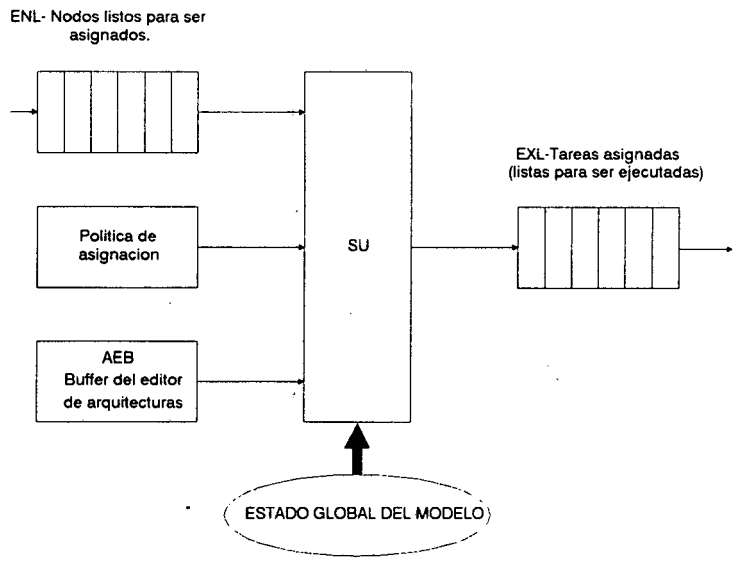


figura 4.4. Organización de la SU.

"buffers" del editor de programas y arquitecturas (PEB-AEB), estructuras de datos globales del simulador.

Para las políticas dinámicas e híbridas, se requiere información adicional acerca de la dinámica del sistema bajo estudio. El estado de la arquitectura está determinado, en todo momento, por las listas del simulador, que asimismo son variables globales del sistema. En todos los casos, la salida de la rutina de asignación es una estructura de datos con los pares tarea-procesador.

Las tareas asignadas son depositadas por la SU en la Execution List (EXL), entrada a la unidad de ejecución (EU), previo cálculo de su tiempo de ejecución a partir del volumen de cómputo del nodo de programa y la performance relativa del nodo de arquitectura. Los elementos de la EXL están agrupados según su procesador. Todas las tareas de la EXL se encuentran en ejecución o listas para ser ejecutadas.

4.1.3.Unidad de ejecución.

Este módulo (Execution Unit, EU) es el encargado de efectivamente simular la ejecución de nodos y su intercomunicación, actualizando el tiempo de simulación. Su estructura se aprecia en la figura 4.5. Las entradas a la EU son la lista de ejecución EXL, ya comentada,

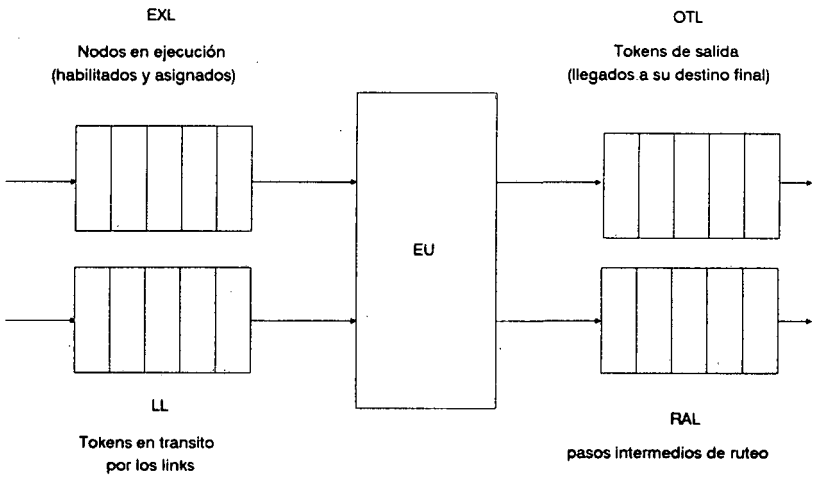


figura 4.5. Organización de la EU.

y la Link List (LL), donde residen los tokens actualmente en tránsito. La organización de la LL puede verse en la figura 4.6. Existe un ítem en la lista para cada link activo en la arquitectura. Cada elemento posee, asimismo, una sublista, organizada como un FIFO, donde se almacenan los mensajes en tránsito por ese link. El tamaño de este FIFO permite evaluar la congestión en la arquitectura, y puede ser usado por políticas "inteligentes" de ruteo, para optar por caminos alternativos. Los links, como vimos en el capítulo anterior, poseen un parámetro que especifica el tamaño máximo permitido del FIFO.

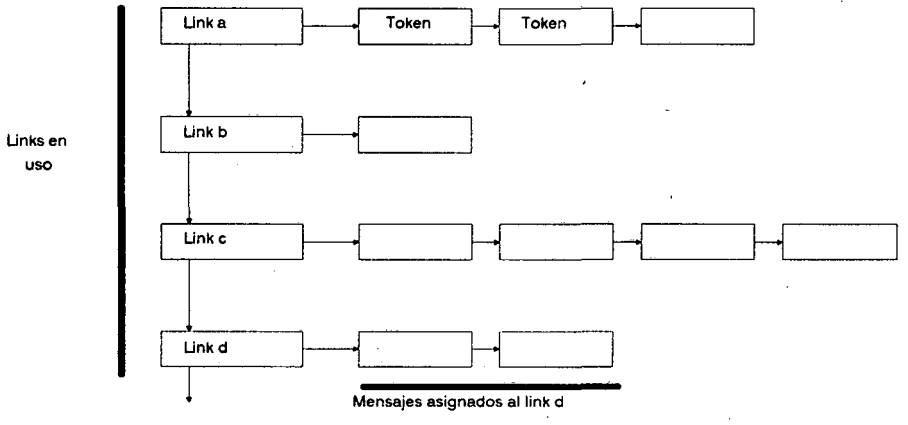


figura 4.6. Organización de la LL.

En la figura 4.7 puede apreciarse el algoritmo fundamental de la EU. Su principal actividad es decidir cuál es el próximo evento del modelo. Recordemos que este simulador es del tipo "event driven", es decir, el tiempo de simulación es siempre actualizado al tiempo acumulado del próximo evento (capítulo 2). En nuestro simulador, existen fundamentalmente dos tipos de eventos: fin de la ejecución de una tarea (comienzo de una comunicación), y fin de un paso de ruteo o de una comunicación entre nodos. La EU, pues, inspecciona la Execution List (tareas en ejecución) y la Link list (mensajes en curso) buscando el elemento con el menor tiempo asociado.

Si el próximo evento es el fin de la ejecución de un nodo, la EU extrae el elemento de la EXL, genera los tokens de salida (de acuerdo a la política de salida especificada por el nodo), y los deposita en la lista de arcos (Ready Arc List (RAL)). Esta lista es la entrada a la unidad de ruteo de mensajes. Si el evento seleccionado por la EU es un paso intermedio de ruteo, el token se retira de la LL, y se vuelve a ingresar en la lista de arcos, para simular el próximo paso de comunicación. Por el contrario, si la EU verifica que el token ha llegado al procesador destino, lo deposita en la lista de tokens de salida (Output token list, OTL), que cierra el ciclo ("circular pipeline", en la terminología de flujo de datos) al ingresar nuevamente en la unidad de habilitación.

Por último, en todos los casos, el tiempo del evento se resta de todos los elementos de la EXL y LL, y el tiempo de simulación se incrementa en el mismo valor.

```
Exec_unit::

Buscar_próximo_evento (EXL,LL);

Si evento = fin_ejecución entonces depositar_tokens_de_salida_en_RAL;

Si evento = Paso_de_ruteo entonces depositar_token_en_RAL;

Si evento = Fin_de_comunicación entonces depositar_token_en_OTL;

Actualizar_tiempo_de_simulación;

fin;
```

figura 4.7. Algoritmo fundamental de la EU.

4.1.4. Unidad de ruteo de mensajes.

Esta unidad (Router Unit (RU)) es la encargada de administrar el tránsito de tokens por los links de la arquitectura. Como vimos en el capítulo anterior, el usuario debe especificar su política de ruteo mediante una rutina de alto nivel, que es concatenada con las rutinas de simulación previamente a la sesión, o bien seleccionando una estrategia de ruteo de la biblioteca del sistema. De todas maneras, existe la posibilidad de trabajar en forma "manual", especificando el ruteo interactivamente, en tiempo de simulación. Este modo de operación se describirá detalladamente al discutir el soporte gráfico en tiempo de simulación. La organización de la RU es ilustrada en la figura 4.8.

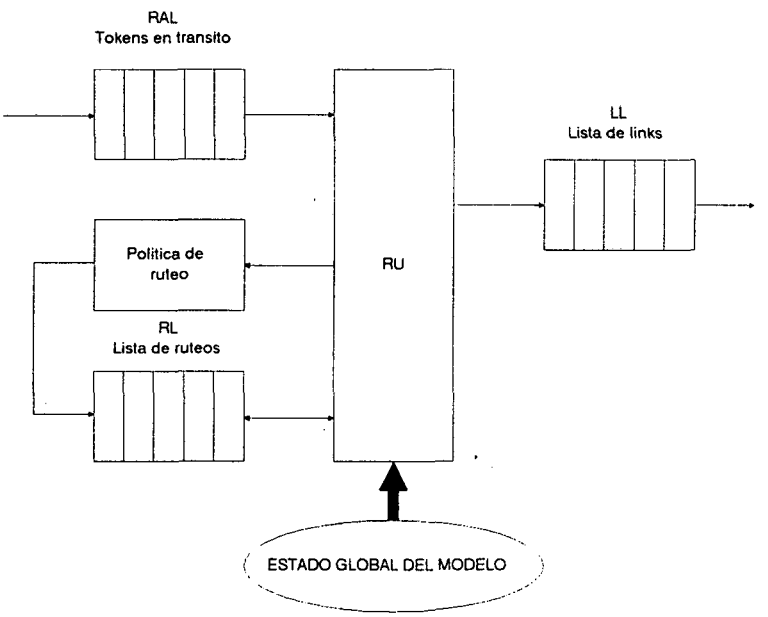


figura 4.8. Organización de la RU.

La RU tiene como entrada la lista de arcos RAL, donde residen los mensajes que aún no tienen ruta asignada, y como salida la lista de links LL, con los tokens en tránsito. La estructura de datos interna mas importante de la RU es la lista de ruteos (Router list, RL), donde se almacena (estática o dinámicamente) el ruteo para cada arco del grafo de programa.

La RL está organizada como puede apreciarse en la figura 4.9. Cada elemento de la RL está identificado por el token del grafo del programa (con su correspondiente tag), y posee una sublista con los sucesivos pasos de ruteo, organizados como un FIFO.

Para cada token de la RAL, la unidad de ruteo inspecciona la RL buscando su ruteo asociado. Si no lo encuentra (es decir, estamos en el primer paso de ruteo) el mismo se genera de acuerdo a la política especificada por el usuario, o bien interactivamente durante la simulación.

La política de ruteo puede también especificar el tiempo de conmutación del elemento intermedio, así como la operación en modo conmutado. Un token "circula" por la unidad de ruteo tantas veces como pasos de ruteo tenga asignados. En cada paso se retira un elemento del FIFO correspondiente en la RL, y el arco vuelve a ingresar en la LL, previo cálculo del tiempo de tránsito del paso de ruteo, a partir de la información del volumen de comunicaciones del nodo del programa y la performance relativa del link de la arquitectura. Cuando el FIFO de pasos de ruteo se ha vaciado, el mensaje ha llegado a destino, y es depositado por la unidad de ejecución en la OTL.

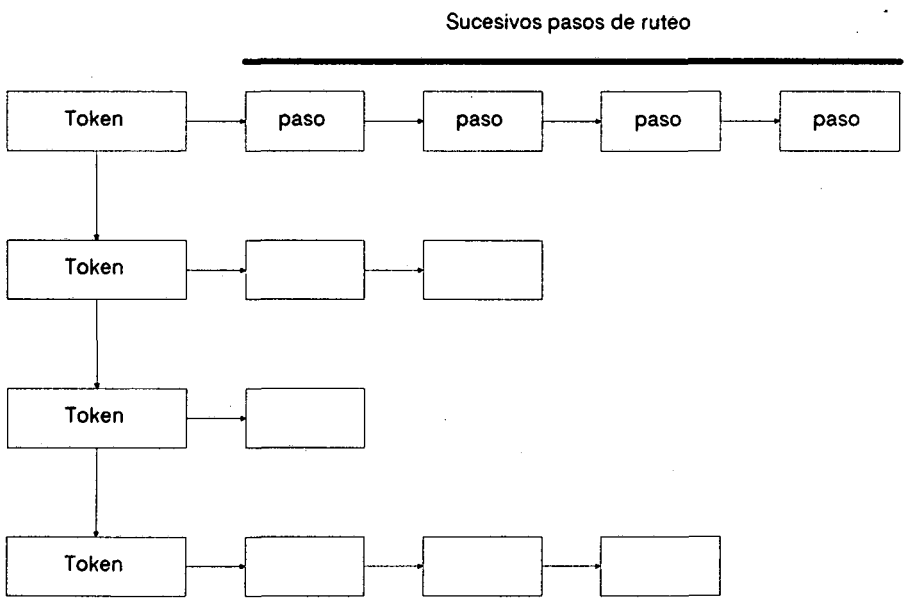


figura 4.9. Estructura de la RL.

4.2.Modos de operación del simulador.

El simulador puede ser configurado para simular sólo el grafo de programa, es decir, sin realizar asignación alguna en una arquitectura. Esta opción es útil para verificar la correcta operación del grafo del programa. Para trabajar de este modo, simplemente se "cortocircuitan" las unidades de asignación y ruteo de mensajes (ver figura 4.10).

El sistema interactivo del ambiente de simulación, que presentaremos en el capítulo siguiente, utiliza la información contenida en las listas del simulador para realizar la animación de los grafos durante tiempo de simulación. Asimismo, a través de la interfaz gráfica puede controlarse la operación del simulador en modo contínuo o paso a paso, activar/desactivar el soporte gráfico, y modificar la velocidad de simulación.

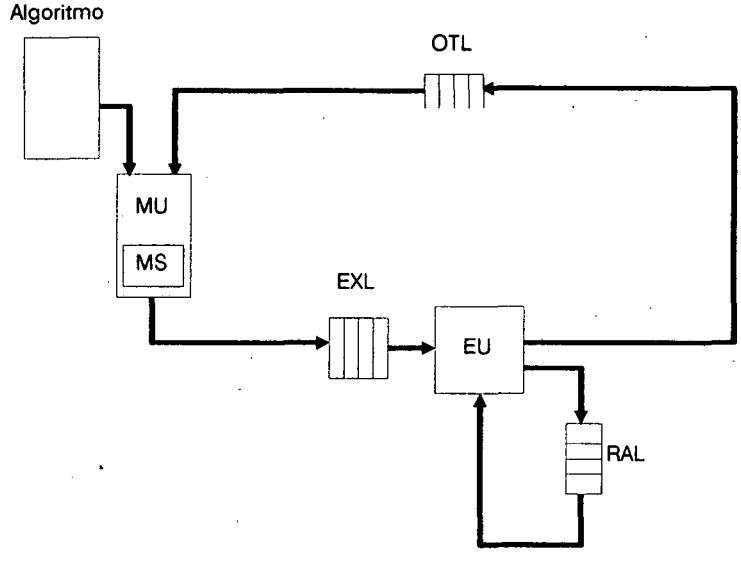


figura 4.10. Simulando sólo el algoritmo.

4.3.Arquitecturas paralelas para el simulador.

Dada la estructura modular en anillo de la máquina de simulación, es interesante analizar su posible implementación paralela. Esto permitiría acelerar los tiempos de simulación, y utilizar modelos mas detallados. El propósito de esta sección es poner en evidencia los factores involucrados en el diseño de tal sistema. La evaluación de la influencia de estos fac-

tores en el rendimiento global de la implementación constituye una primera aplicación del simulador, como herramienta de guía en el proceso de síntesis.

La primer organización posible es, obviamente, un anillo. Los procesadores del mismo realizarían las funciones de los bloques de la figura 4.1, es decir, las unidades de habilitación, asignación, ejecución, ruteo de mensajes y almacenamiento de los grafos. De hecho, la organización en anillo es la utilizada en muchas máquinas de flujo de datos dinámicas. Un posible esquema puede verse en la figura 4.11. Aquí hemos agrupado la unidad de habilitación (MU) con la de asignaciones (SU) en el MSP (Matching/scheduling processor), la unidad de ejecución (EU) con la de ruteo de tokens (RU) en el ERP (Execution/routing processor), y el buffer de arquitecturas y programas junto al procesador frontal que comunica al simulador con el sistema de desarrollo, FNP (Front end/node storage processor). En esta organización el simulador actúa como un acelerador esclavo del sistema de desarrollo de modelos, en la misma forma que los conocidos aceleradores para sistemas CAD [33][34]. La comunicación entre el sistema de desarrollo de modelos y el acelerador se realiza a través de un almacenamiento compartido en el FNP.

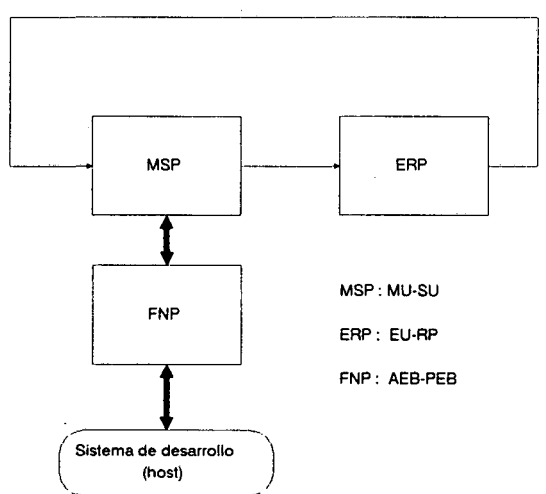


figura 4.11. Organización en anillo.

Es fácil de ver que el cuello de botella del "pipeline circular" es la unidad de habilitación MU. En realidad, esta unidad es la que controla la ejecución de instrucciones, al ser quien decide qué nodo verifica la condición de disparo. Obsérvese en la figura 4.1 que todos los nodos de la ENL están listos para ser asignados, y podrían ser procesados en paralelo. Por lo tanto, el paralelismo del simulador en sí mismo podría explotarse en dos sentidos:

1) Mediante múltiples unidades de habilitación. La operación de búsqueda asociativa de la MU, es ciertamente lenta, por lo que es lógico esperar un fuerte incremento en las prestaciones al replicar la MU, y brindar apoyo hardware a la operación de "match" (soporte en hardware para la operación de "hashing", por ejemplo).

2) Otra forma de introducir paralelismo en el simulador consiste en proveer varias unidades de ejecución, de modo tal de procesar simultáneamente todos los nodos generados por los MSP's. La segunda propuesta es, entonces, una estructura en anillos múltiples como la que se exhibe en la figura 4.12. Esta estructura parece ser un tanto rígida, en el sentido de que cada MSP está fuertemente acoplado a un único ERP y FNP, y la comunicación entre anillos es costosa. Por otra parte, hemos visto que el tamaño del MS (en la implementación paralela, el

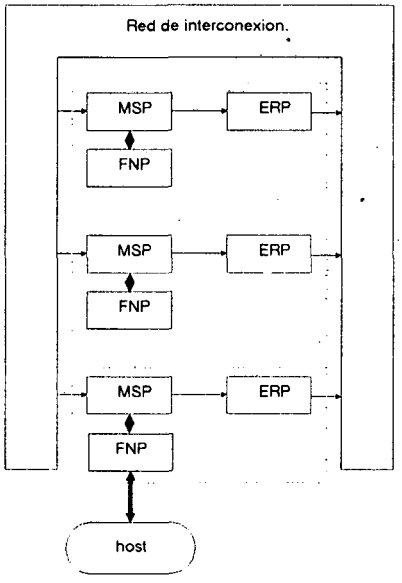


figura 4.12. Estructura en anillos múltiples.

número de MSP's) depende fuertemente de la aplicación, y puede no guardar relación alguna con el número óptimo de ERP's. El tercer problema de la estructura en anillos múltiples es su difícil escalabilidad.

4.3.1.Partición del WBG para su simulación paralela.

Como detallábamos en el capítulo 3, un WBG exhibe paralelismo en tres niveles: En un sentido estático, dos nodos de un WBG no directamente conectados entre sí (es decir, sin dependencias directas) pueden ser ejecutados en paralelo. En segundo lugar, pueden activarse varias copias de un mismo nodo mediante el uso de colores o tags. Por otra parte, las características dinámicas del WBG permiten la ejecución concurrente de copias de subgrafos, mediante la utilización de diferentes espacios de tag, a través de las clases Call y Return.

Una primera forma simple, pues, de particionar el WBG para su simulación paralela consiste en asignar un unidad de habilitación a todo nuevo espacio de tag para cada subgrafo. Obviamente, debido a que los recursos son siempre finitos, si tenemos n unidades de habilitación, la asignación debería hacerse " módulo n ".

En términos de almacenamiento de nodos, es obvio que un único FNP puede introducir un cuello de botella. Podemos utilizar dos estrategias para replicar el FNP:

a)analizar el WBG para reconocer sus zonas de ejecución paralelas. Estas secciones son asignadas a diferentes FNP's. Esto es eficiente en lo que respecta a utilización de memoria, pero puede generar intenso tráfico entre unidades en grafos fuertemente comunicados.

b)copiar todo el grafo en cada FNP. Cuando un MSP necesita leer un nodo, siempre lo encuentra en el FNP mas cercano. Obviamente, el tráfico se reduce y la utilización de memoria es ineficiente, aunque tal vez sea la estrategia mas simple de implementar.

La arquitectura ilustrada en la figura 4.13 puede ser un interesante compromiso. En ella tenemos dos tipos de procesadores: Aquellos encargados de realizar la operación de habilitación - asignación de nodos y el almacenamiento del grafo (Node store, matching and scheduling processor, NMS) y aquellos encargados de la ejecución de nodos, simulacion de los

pasos de ruteo y actualización del tiempo de simulación (ERP: execution - router processor). Todos los procesadores se comunican mediante una red de interconexión IN.

El ruteo de mensajes desde los ERP (tokens de salida de los nodos del WBG) a los NMS es relativamente simple, ya que la NMS destino es identificable a partir de la información [tag modulo n], del token. Esto no es así para los mensajes desde los NMS a los ERP (que son todos idénticos), ya que es necesario establecer algún criterio para mantener adecuadamente balanceada la carga de estos últimos. Probablemente una simple estrategia de tipo "round robin" sea suficiente.

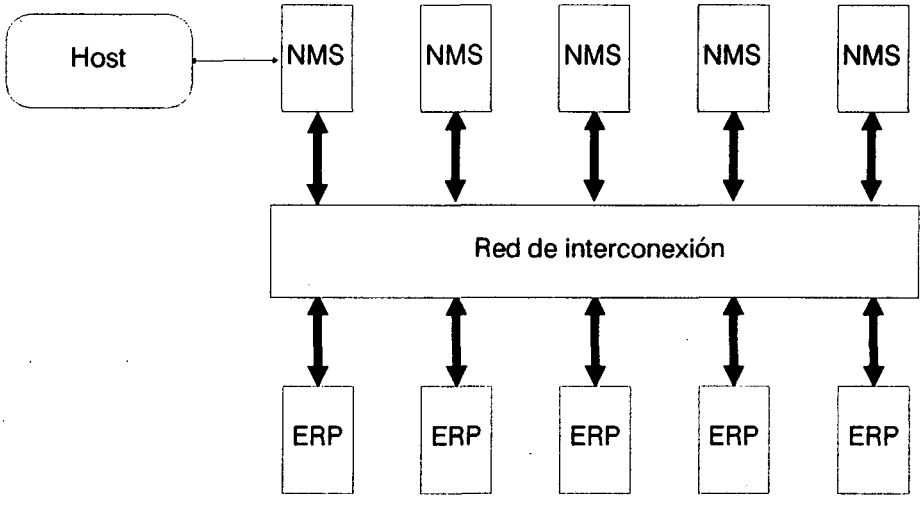


figura 4.13. Estructura propuesta.

4.4.Resumen.

En este capítulo se detalla la implementación de la máquina de simulación del ambiente integrado propuesto. Esta constituye un simulador de eventos discretos modular, cuya estructura recuerda a la de las máquinas de flujo de datos dinámicas. La descripción se realiza a partir del comportamiento interno de cada módulo, sus estructuras de datos fundamentales, y la interfaz entrada/salida. Las unidades que constituyen la máquina son altamente independientes entre sí, lo que permitiría su eficiente paralelización. Los listados de la actual implementación del sistema, escrito en Pascal, figuran en el Apéndice A.

Capítulo 5.

El ambiente integrado.

En este capítulo se detallarán las facilidades provistas en el ambiente interactivo de simulación. El sistema ha sido diseñado teniendo como principal objetivo una interfaz altamente interactiva, basada en gráficas animadas e iconos, con selección mediante ratón y múltiples ventanas simultáneas. Se pretende de esta forma reducir el período de aprendizaje del operador, al tiempo que se agiliza el proceso de desarrollo de un modelo correcto.

5.1.Diagrama en bloques del sistema.

La figura 5.1 ilustra el diagrama en bloques del ambiente implementado. El sistema permite la creación y modificación de los grafos de programa y arquitectura mediante un simple editor gráfico. Las facilidades del sistema de edición se detallan en las siguientes secciones. La interfaz gráfica permite asimismo editar la asignación de tareas a procesadores, y definir las diferentes opciones de simulación. Durante la simulación los gráficos del programa se animan indicando en diferentes colores los nodos no habilitados, listos para ser ejecutados y en ejecución, así como el número de instancias activas de cada nodo. La figura 5.2 ilustra la organización del software del sistema.

5.2.Convenciones y notación.

Con el término "click" nos referimos a la acción de presionar y soltar un botón del ratón. Es la operación habitual de selección para este tipo de dispositivos.

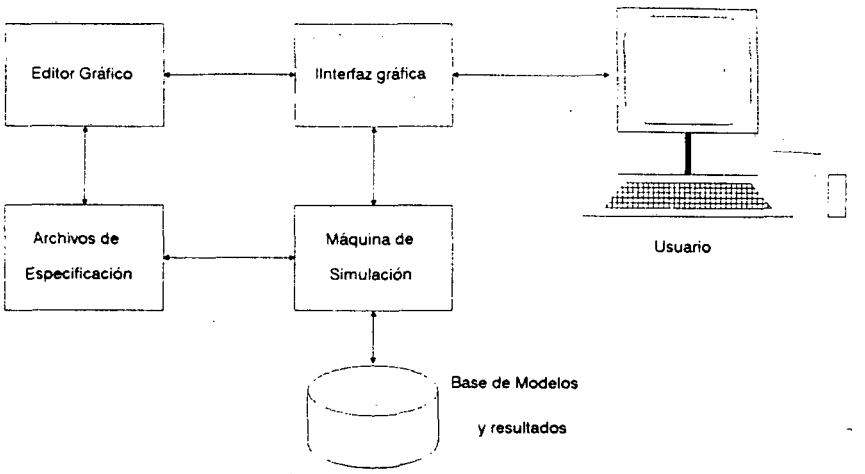


figura 5.1. Diagrama en bloques del sistema.

Cuando no se especifica el botón (izquierdo o derecho) del ratón, se sobreentiende que es el izquierdo. El botón central, al no ser estándar de todos los ratones, no se utiliza.

El término "video inverso" en referencia a un icono o cartel indica la utilización de blanco en lugar de negro y viceversa. Se emplea habitualmente para indicar el icono o cartel actualmente en operación.

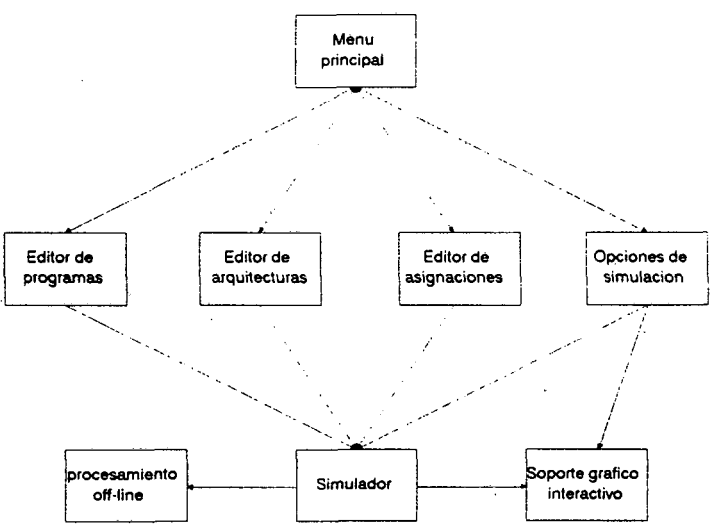


figura 5.2. Organización del software.

El ambiente interactivo utiliza diversos tipos de cursor:

Cursor de selección: Es el típico cursor gráfico en forma de flecha. Se utiliza en el modo normal del editor, para la selección de iconos, de menúes, de opciones dentro de los menúes y para la actualización de las coordenadas de la ventana activa en la zona de barras.

Cursor de dibujo: Este cursor, con forma de cruz, es utilizado por el editor para el dibujo de nodos y arcos del grafo.

Cursor de borrado: Es un cursor con forma de "goma de borrar", utilizado por el editor en el modo de borrado de nodos.

Cursor en "mano abierta": este cursor es el que se utiliza para las operaciones de movimiento de nodos y ventanas.

Cursor en "mano indicadora": utilizado en ls operaciones de selección múltiple de nodos en el editor, asignación manual de tareas e inicialización de la sesión de simulación.

Cursor de texto: utilizado en las operaciones de escritura de comentarios y etiquetas en el área de edición.

Cursor alfanumérico (underscore): utilizado en las operaciones de inserción y edición de caracteres alfanuméricos. Es el cursor estándar de escritura.

Las diferentes funciones del ambiente interactivo son activadas mediante menúes contenidos en "ventanas de selección". Todas estas ventanas se utilizan de la misma manera: se selecciona con el ratón la opción deseada, que se muestra en video inverso y letras mayúsculas (figura 3). La opción seleccionada se activa mediante un click. Para cancelar el menú, simplemente se hace un click fuera de la ventana.

5.3.Menú principal.

Este menú (figura 5.3), es la ventana de apertura del sistema. Mediante el mismo pueden accederse a las funciones principales del ambiente: edición de los grafos de programa y arquitectura, generación y modificación de la asignación de tareas a procesadores, simulación del modelo generado y finalización de la sesión interactiva.

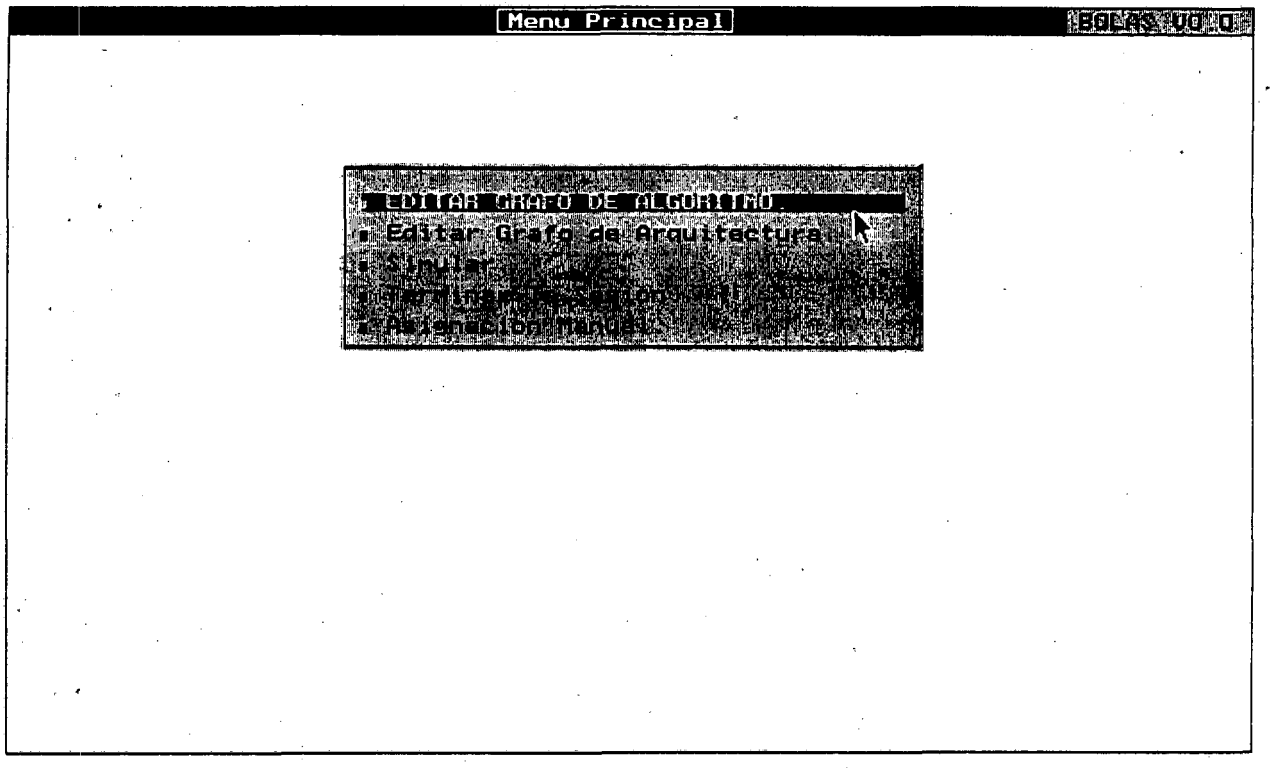


figura 5.3. Menú de apertura.

5.4.Editor gráfico.

El editor permite la creación, modificación y almacenamiento de grafos de programas y arquitecturas. El "buffer" de edición es compartido con el simulador, de forma tal que el usuario puede pasar de la edición a la simulación interactivamente, lo que reduce considerablemente el período de depuración del modelo.

La figura 5.4 ilustra una pantalla típica del editor de grafos de programas. Se distinguen 6 zonas:

-Zona de dibujo.
-Zona de menues.
-Zona de iconos.
-Zona de barras de control.
-Zona de información al operador.
-Indicador del nivel de anidamiento.

La zona de dibujo es una ventana sobre el grafo actualmente en edición. La misma puede moverse mediante los operadores de la zona de barras. La zona de menúes permite acceder a diversas funciones del editor. El menú de archivo permite realizar operaciones de disco tales como carga y almacenamiento de grafos o subgrafos, directorios, etc. El menú de edición soporta las funciones habituales del sistema operativo, operaciones sobre bloques y comandos generales.

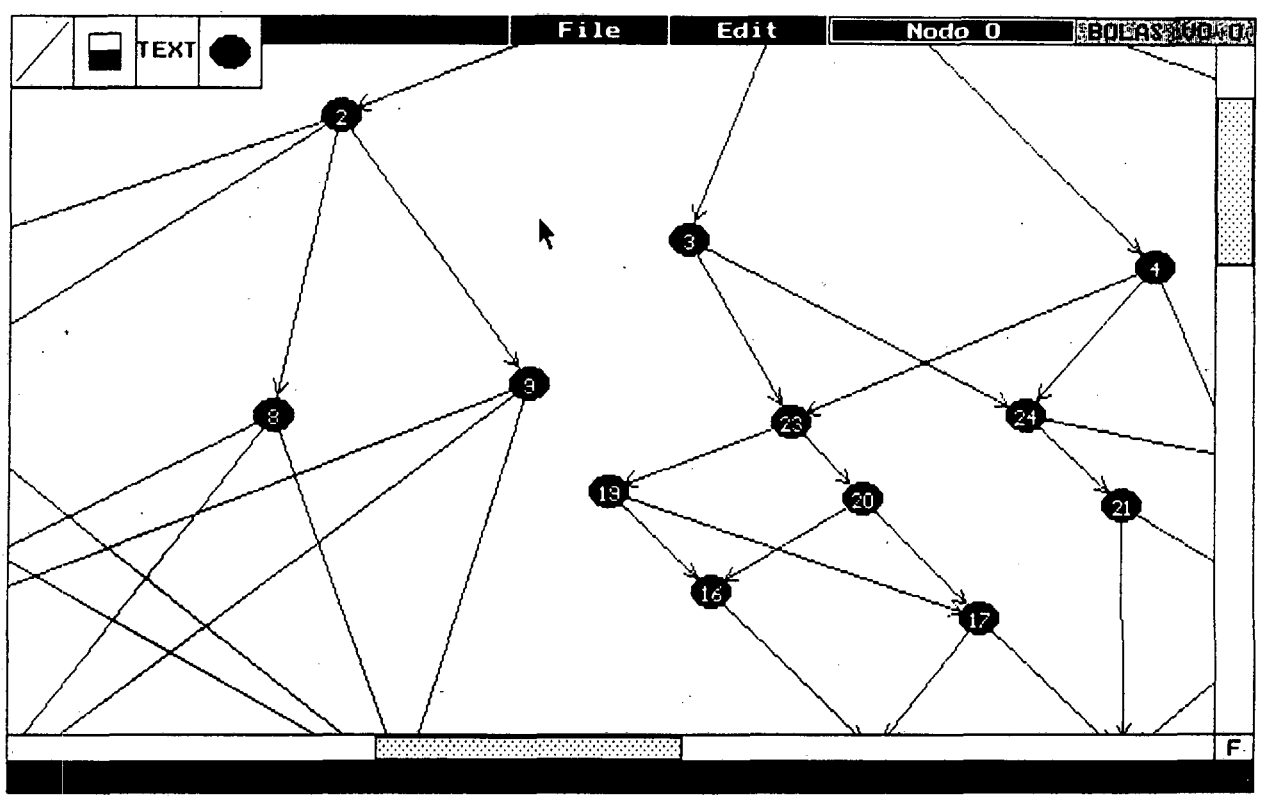


figura 5.4. Pantalla de edición.

La zona de iconos contiene los operadores elementales del editor. Los mismos permiten la creación y el borrado de nodos en la zona de edición, el trazado de arcos entre los nodos del grafo, y el agregado de texto en la pantalla.

La zona de información es utilizada para los mensajes de ayuda en cada uno de los modos de operación del editor, diversos mensajes de error, y la entrada de información alfanumérica (parámetros para los comandos del sistema operativo, por ejemplo).

El indicador de nivel de anidamiento señala, en todo momento, el nodo del grafo a quien pertenece la sesión de edición activa. El editor permite una especificación jerárquica del grafo en estudio, siendo este indicador una señal del nivel de la jerarquía en el que se está trabajando. Este tema será tratado detalladamente en el apartado "Especificación jerárquica y nodos MACRO".

5.4.1. Ventana de edición.

Los grafos son creados en la zona de edición. La misma es una "ventana" sobre el área total de edición, de forma tal que pueden editarse grafos mucho mayores que los que "caben" en la misma. Esta es una técnica profusamente utilizada en los editores gráficos de sistemas CAD.

Para mover la ventana sobre toda el área de edición se utilizan las barras de control. Estas barras indican la posición relativa de la ventana de edición en el área total. Para moverlas, el usuario simplemente sitúa el cursor sobre la barra, presiona el botón izquierdo del ratón y lleva la barra a la nueva posición. La pantalla se actualiza automáticamente a sus nuevas coordenadas.

De todas maneras, es muchas veces interesante poder tener una visión global de todo el área de edición, en escala reducida. Para ello el usuario debe hacer un "click" sobre el rectángulo indicado F (full view) en el ángulo inferior derecho de la pantalla. La misma se verá ahora como se indica en la figura 5.5. Obsérvese que las barras ocupan todo el espacio disponible. Un rectángulo en línea punteada indica la posición de la ventana de edición en visión normal. El usuario puede mover esta ventana a cualquier posición mediante el ratón. Pa-

ra volver al modo de visión normal, solo se requiere hacer un nuevo click sobre el ángulo inferior derecho de la pantalla, ahora identificado con la letra N (normal view, figura 5.5).

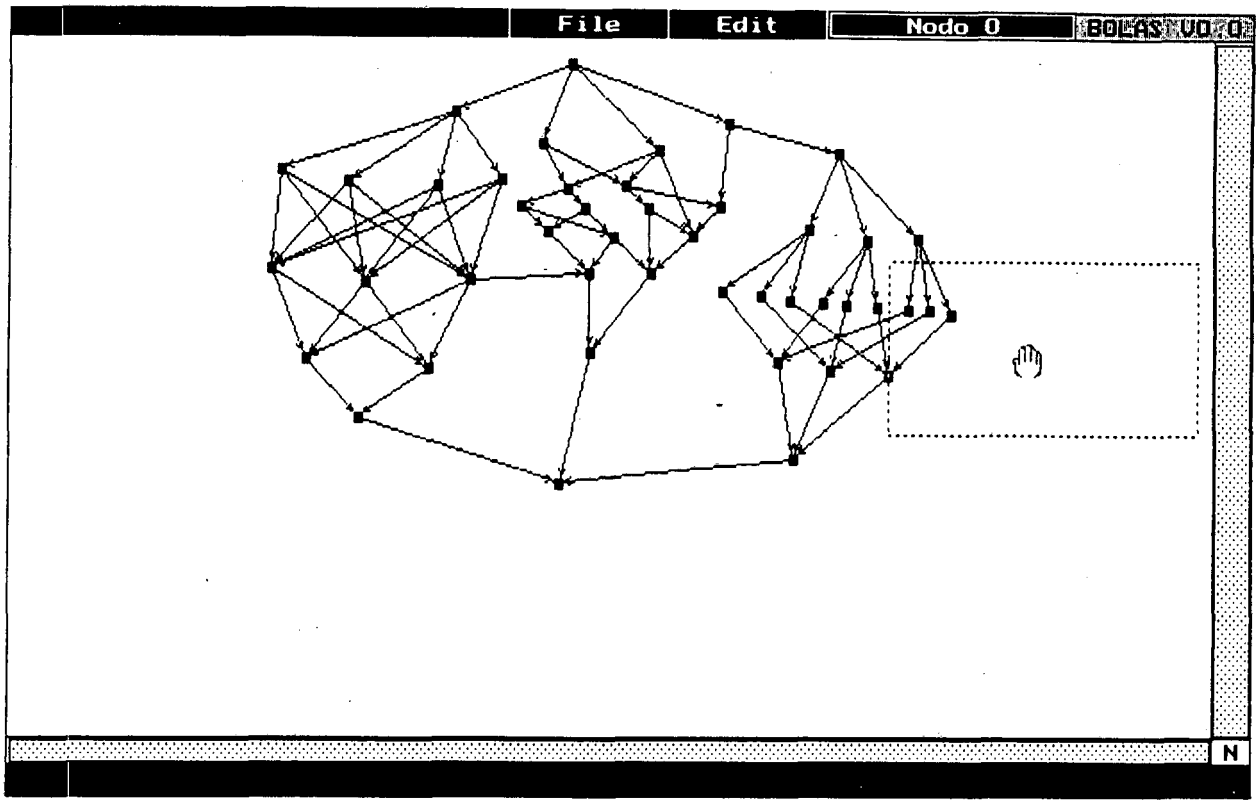


figura 5.5. Visión completa del area de edición

5.4.2.Creación de nodos.

Para crear nodos en la ventana de edición, el usuario simplemente efectúa un click sobre el icono de nodos. El icono se mostrará en video inverso, indicando el modo activo. Para fijar un nodo, basta con efectuar un click en las coordenadas deseadas de la ventana de edición. Los nodos son numerados automáticamente por el sistema de edición, para evitar inconsistencias en el grafo. Estando dentro de este modo de operación, el usuario puede mover la posición de la ventana mediante las barras de control, así como conmutar libremente entre visión normal y total. Para abandonar el modo de creación de nodos, simplemente se efectúa un click sobre el botón derecho del ratón.

5.4.3. Borrado de nodos.

Para eliminar nodos de la zona de edición, se selecciona el icono con forma de "goma de borrar" (figura 5.6). El icono se muestra en video inverso y el cursor adopta forma de goma.

Este modo sólo puede activarse si existen nodos editados. Si no es así, el click no tiene efecto alguno. Para borrar, simplemente se sitúa el cursor sobre el nodo en cuestión y se efectúa un click. El nodo se elimina, así como todos sus arcos asociados. Si el nodo es de la clase MACRO (sección 5.4.12), todos sus nodos internos también son eliminados. Al igual que en el modo anterior, puede moverse la ventana mediante las barras de control y conmutar entre visión total y normal libremente. El modo de borrado se termina mediante un click en el botón derecho.

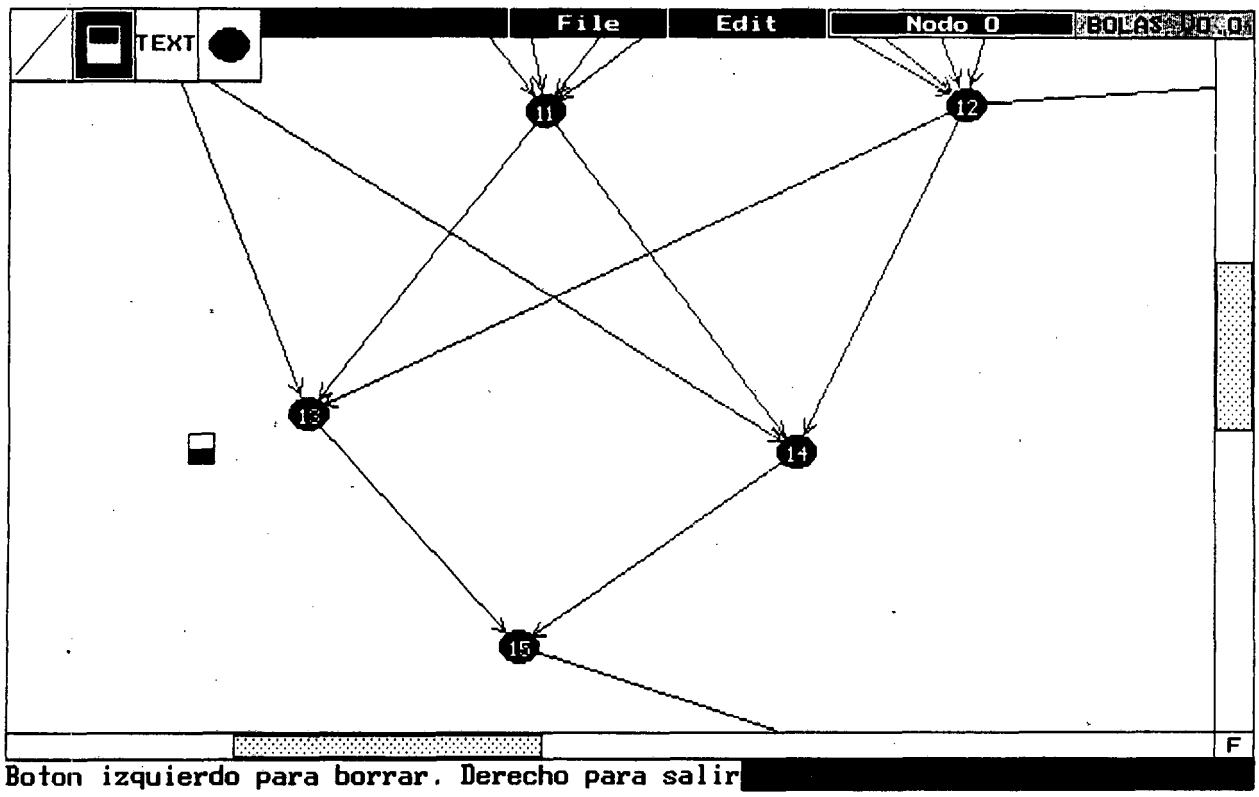


figura 5.6. Borrado de nodos en el área de edición.

5.4.4. Tendido de arcos entre nodos.

Si existen nodos en el buffer del editor, pueden tenderse arcos entre ellos haciendo un click sobre el ícono de líneas. Las mismas se indican especificando mediante el cursor sus nodos fuente-destino. Asimismo pueden indicarse mediante sucesivos clicks puntos intermedios de la trayectoria. De todas maneras, el sistema implementa un algoritmo de ruteo automático que sortea los posibles obstáculos encontrados en la trayectoria (figura 5.7). De esta forma el sistema pretende mantener la "legibilidad" del grafo en todo momento.

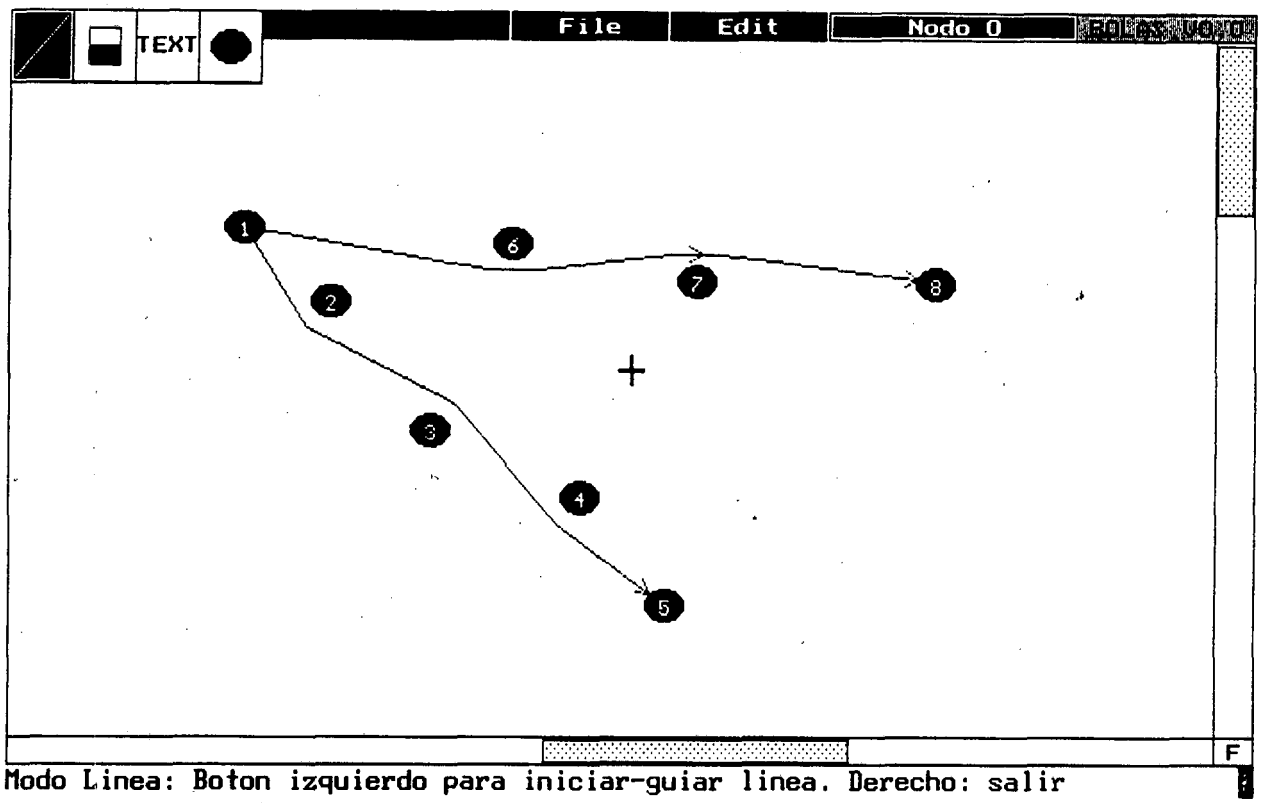


figura 5.7. Tendido de arcos entre nodos.

5.4.5. Inserción de texto.

Esta opción refuerza la documentación del grafo, al permitir que el usuario incluya comentarios y etiquetas alfanuméricas en la zona de edición. Para entrar en el modo texto, se debe realizar un click sobre el icono correspondiente (figura 5.8). El texto insertado puede ser editado en cualquier momento, incluso fuera del modo de inserción, simplemente selec-

cionándolo con el cursor. Cuando el grafo es almacenado en disco, el texto asociado se guarda en un archivo con el mismo nombre que el grafo, pero con extensión ".$TX". El modo texto, como todos los demás modos, se termina con un click en el botón derecho del ratón.

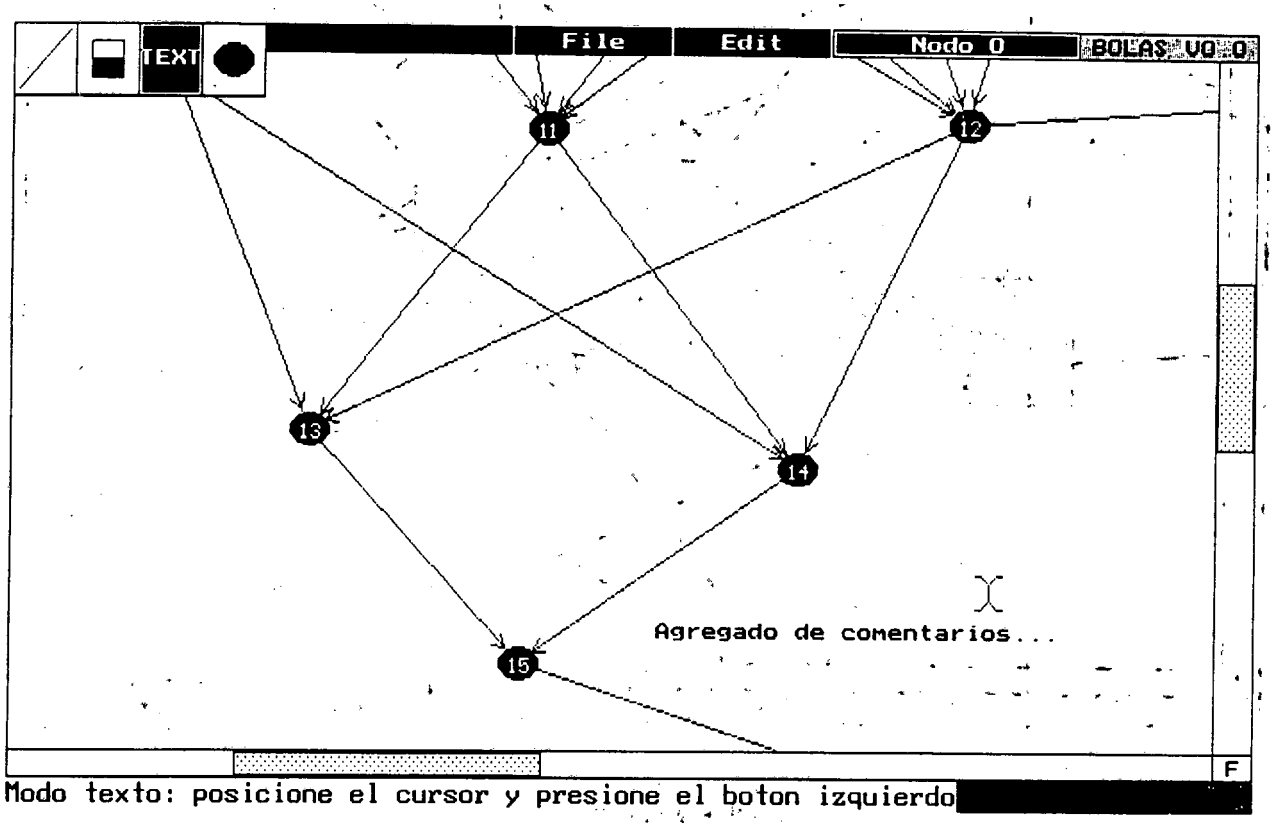


figura 5.8. Modo de inserción de texto

5.4.6. Programación del grafo.

Generada la topología del grafo, es necesario programar sus parámetros. Esta es una operación simple. Sólo es necesario efectuar un click sobre el nodo que se desea programar. Esto produce la apertura de una "ventana de programación" tal como al expuesta en la figura 5.9. A través de la misma es posible indicar los valores del volumen de cómputo, la clase del nodo y las politicas de entrada y salida. Una vez programado el nodo, se abrirá una ventana para cada una de los arcos de salida del mismo, con los parámetros de volumen de comunicaciones y probabilidad de token. También mediante esta ventana puede eliminarse el arco en cuestión, mediante un click sobre el rectágulo "DELETE" del borde superior derecho de la ventana.

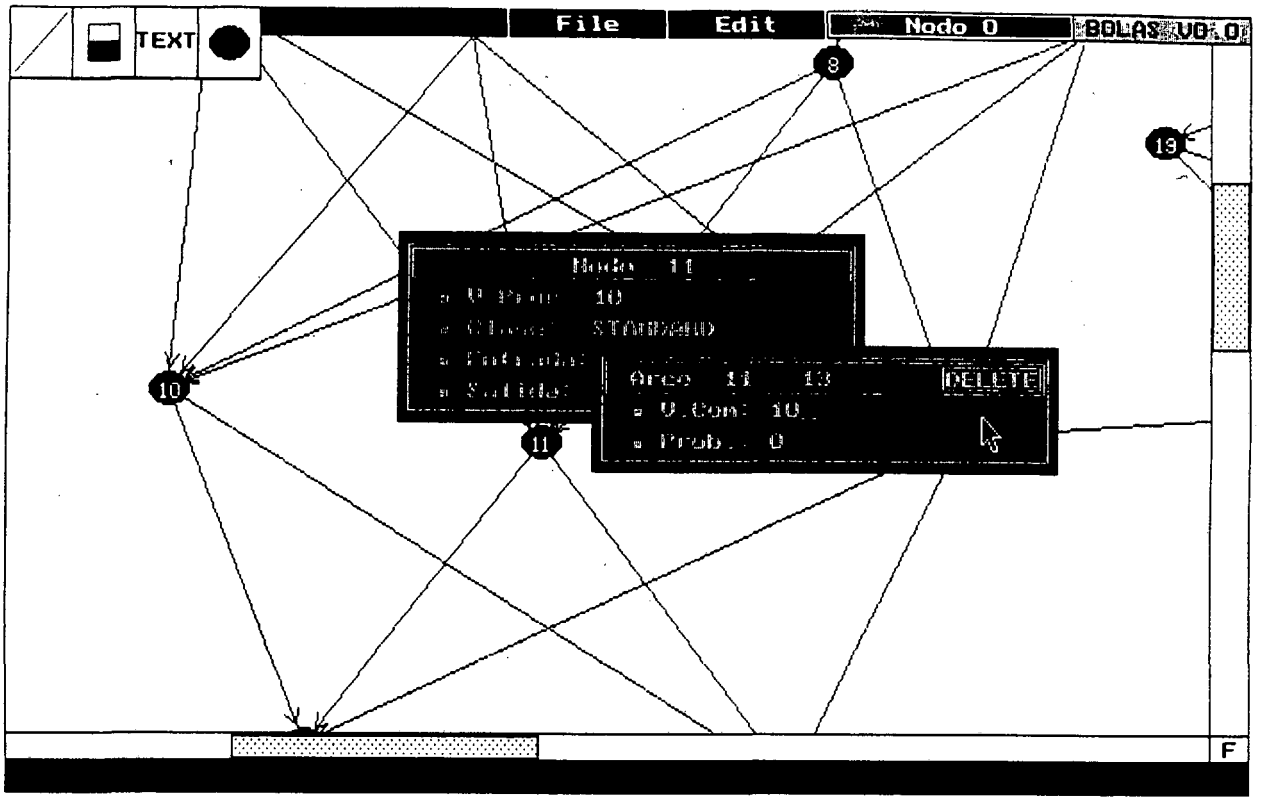


figura 5.9. Ventana de programación del grafo.

5.4.7.Operaciones de disco.

Para acceder a las funciones de disco del sistema, debe situarse el cursor sobre el rectángulo "FILE" de la zona de menúes. Esto abrirá una ventana de selección estándar, como la indicada en la figura 5.10. Las opciones de este menú permiten leer y almacenar grafos, así como visualizar directorios.

5.4.8.Opciones de edición.

Se accede a estas funciones a través del menú "EDIT" (figura 5.11). Las mismas permiten reinicializar el editor, redibujar el grafo, imprimirlo y acceder a funciones estándard del sistema operativo.

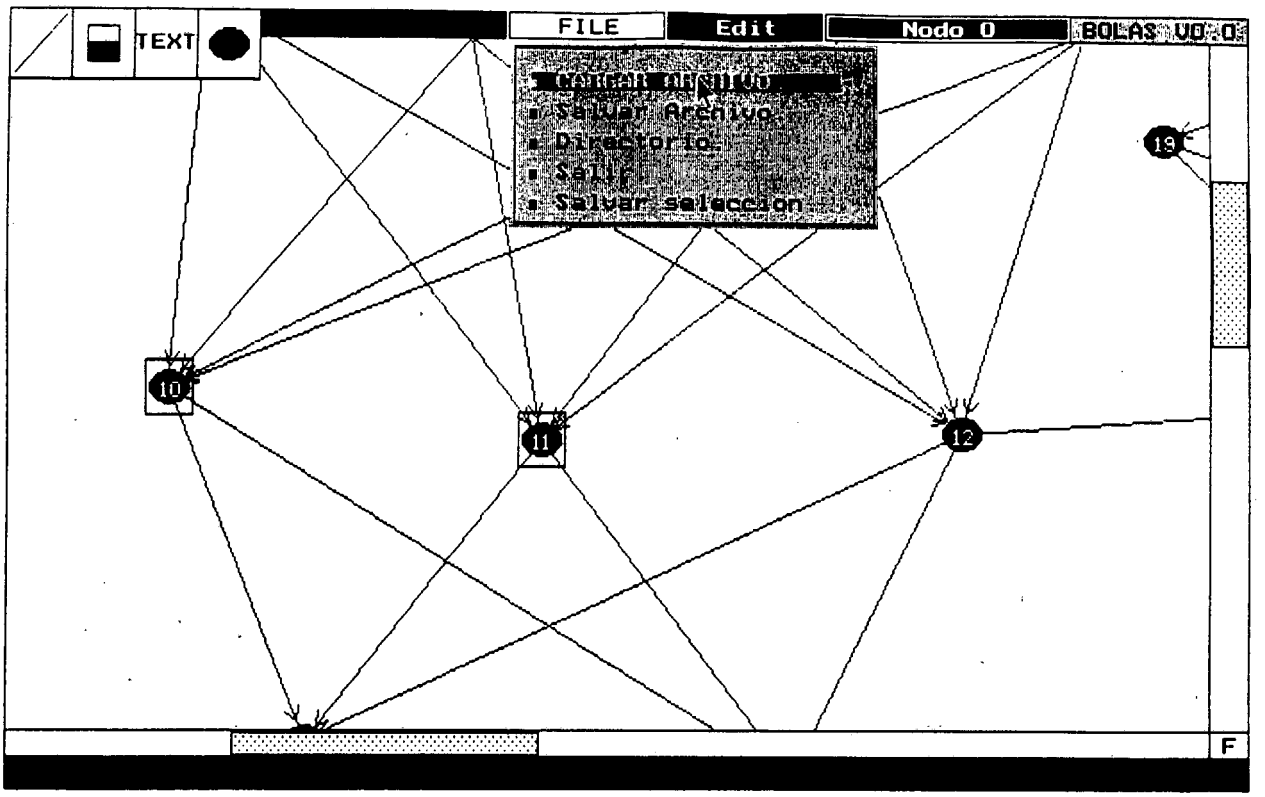


figura 5.10. Operaciones de disco.

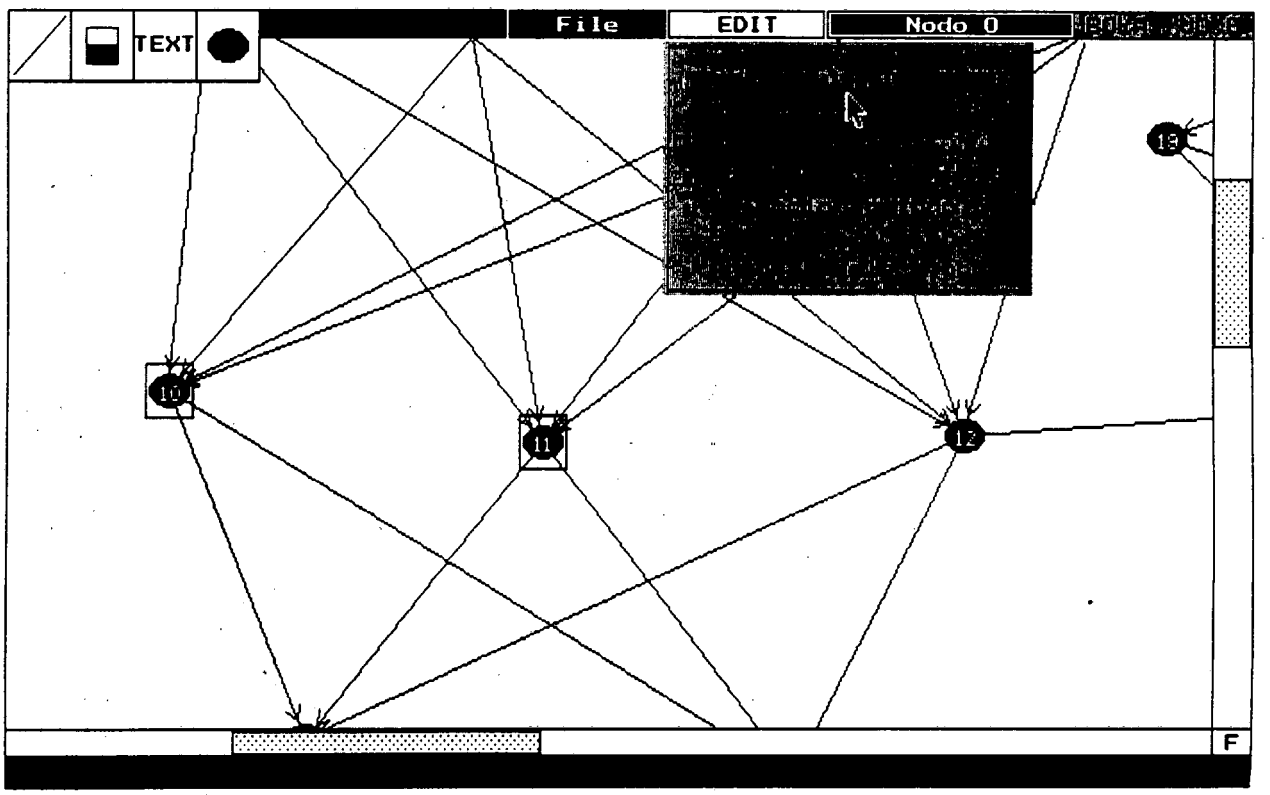


figura 5.11. Opciones de edición.

5.4.9.Operaciones sobre bloques.

Estas operaciones permiten seleccionar, mover, borrar y almacenar en disco un nodo o grupo de nodos del grafo total. Para seleccionar una zona del grafo, el usuario sitúa el cursor en un punto cualquiera de la pantalla y presiona el botón izquierdo del ratón. La forma del cursor pasa a ser una "mano indicadora". Ahora al mover el cursor, se dibujará un rectángulo en línea de puntos, una de cuyas diagonales tiene por extremos el punto inicial y el cursor (figura 5.12). El operador modifica este rectángulo hasta que abarque el o los nodos que se desea seleccionar. Hecho esto, libera el botón. Los nodos seleccionados se indican rodeándolos de un rectángulo en línea llena (figura 5.13).

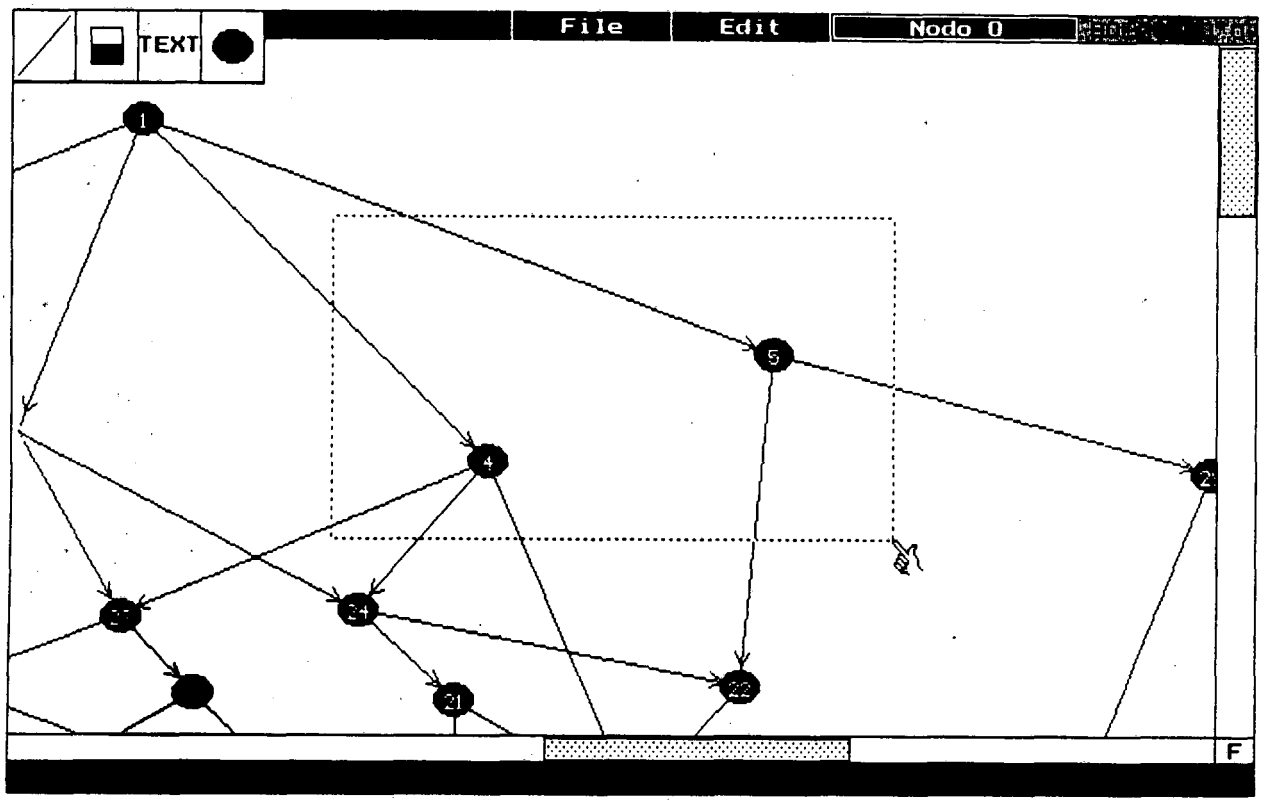


figura 5.12. Selección de nodos.

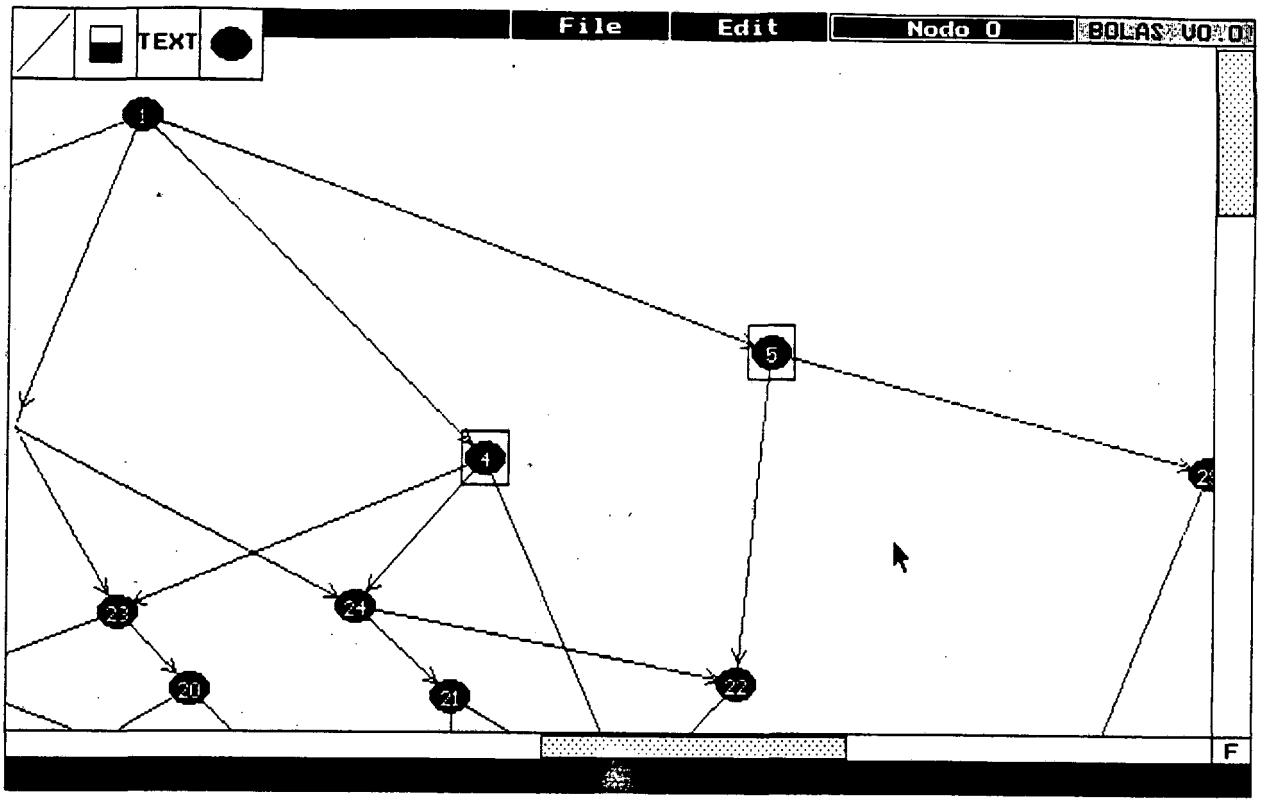


figura 5.13. Nodos seleccionados.

5.4.10.Movimiento de un nodo.

Para cambiar las coordenadas de un nodo seleccionado, se sitúa el cursor sobre el mismo y se presiona el botón izquierdo del ratón. La forma del cursor pasará a ser una mano abierta. Manteniendo presionado el botón, se lleva el cursor a la posición deseada. el nodo se moverá solidariamente. Al liberar el botón, el grafo se actualizará automáticamente.

5.4.11.Selección múltiple.

Para mover un grupo de nodos conservando su posición relativa, se selecciona el conjunto deseado (sección 5.4.9) y se activa la opción "Agrupar nodos" del menú de opciones de edición (figura 5.11). El conjunto será rodeado por un rectángulo en línea llena (figura 5.14). Ahora la operación de mover un nodo, vista mas arriba, hará que todos los demás nodos asociados se muevan en conjunto. Para deseleccionar los nodos, simplemente se efectúa un click sobre el botón derecho.

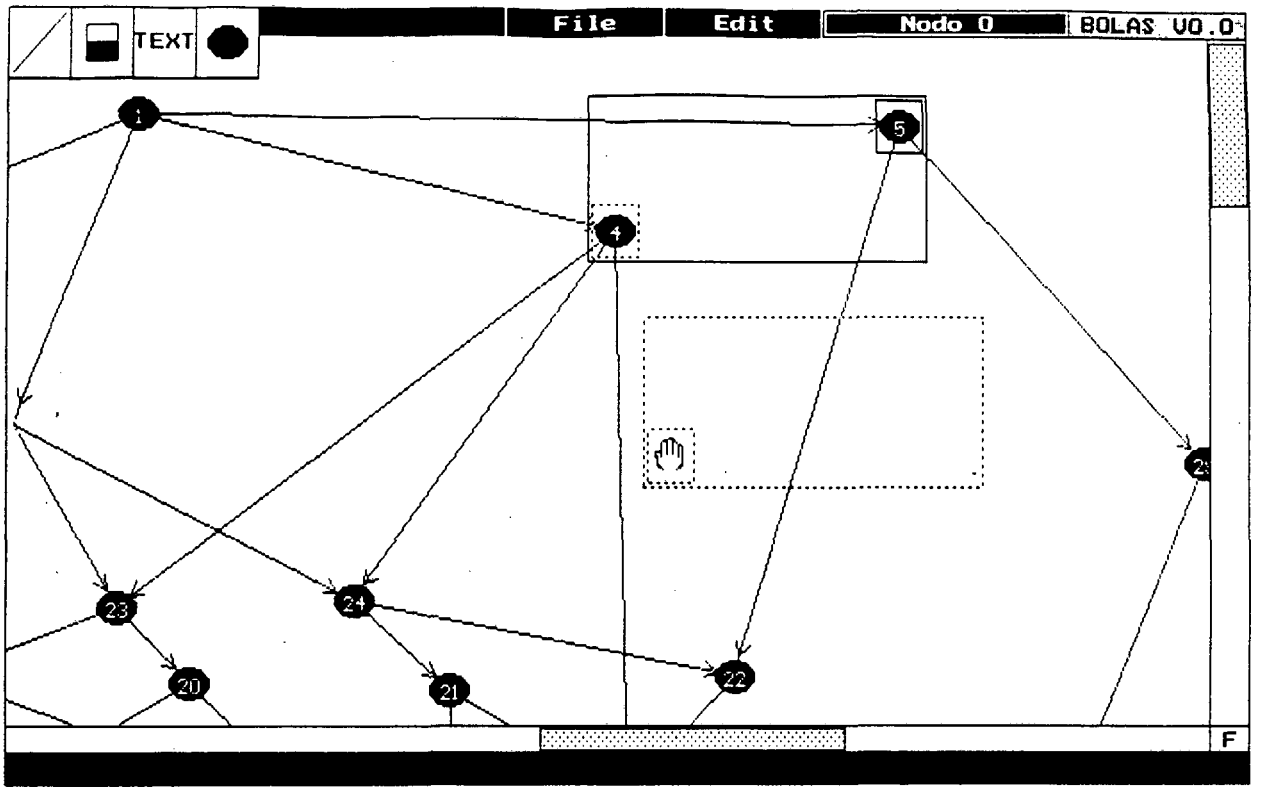


figura 5.14. Selección múltiple.

Mediante el procedimiento arriba citado, podrían seleccionarse todos los nodos del grafo. Otra forma de hacer esto es mediante el comando "seleccionar todo" del menú de opciones (figura 5.11). Mediante este menú también es posible borrar el conjunto seleccionado.

El grupo de nodos puede ser también almacenado en disco, mediante la opción "salvar selección" del menú "File".

5.4.12. Especificación jerárquica y nodos MACRO.

Como decíamos anteriormente, el editor soporta una especificación jerárquica del grafo del programa. Cada nodo del grafo puede, a su vez, estar formado por otro grafo (grafo "interno" del nodo MACRO, figura 5.15.). Esta forma de especificación permite un crecimiento ordenado del grafo, conservando su legibilidad. Asimismo permite administrar el aumento de nuestro conocimiento sobre el modelo. Por ejemplo, podemos comenzar a modelar los nodos como simples retardos, y a medida que vamos aumentando el nivel de detalle reemplazamos los retardos por subgrafos.

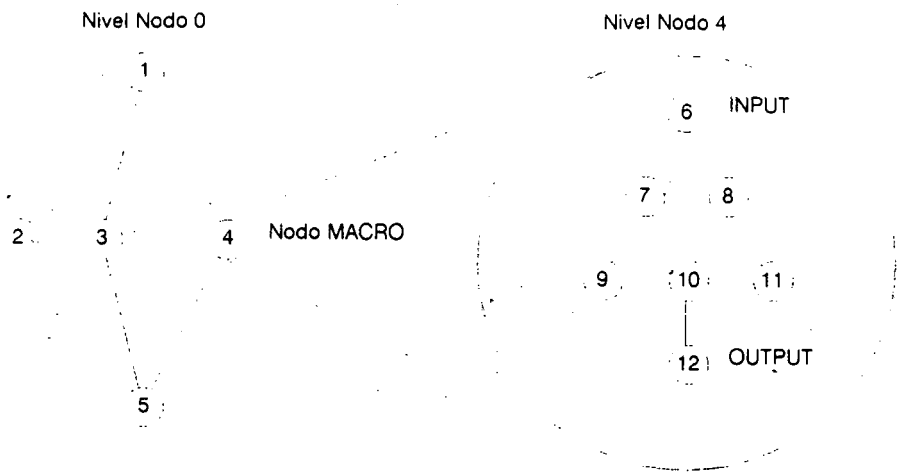


figura 5.15. Especificación jerárquica.

El nivel mas externo de la jerarquía (en el que nos encontramos cada vez que comienza una sesión de edición), se denomina "Nodo 0". El editor informa el nivel en el que se encuentra mediante el indicador de nivel de anidamiento (margen superior derecho de la pantalla, figura 5.4). Al programar un nodo como de tipo MACRO, se genera una nueva instancia del editor, en forma reentrante, con un juego "fresco" de todas sus variables internas, etc. La nueva sesión se denomina "nodo N", donde N es el número de identificación del nodo MACRO en el nivel inmediato superior. Por ejemplo, si en el nivel "nodo 0" programamos el nodo 4 como MACRO, entraremos a una nueva sesión de edición, rotulada como "nodo 4".

Todas las operaciones del editor son relativas al nivel de anidamiento. Por ejemplo, si nuestro nodo 4, a su vez, tiene nodos MACRO, y encontrándonos en el nivel "nodo 4" hacemos una operación de almacenamiento en disco, el grafo que se salvará será el correspondiente al nodo cuatro, y todos su niveles internos. El nivel cero no será almacenado. Lo mismo es válido para las operaciones de bloques, texto, etc. Obviamente, para salvar todo el grafo hay que hacer una operación de almacenamiento desde el "nodo 0".

Ahora bien, para preservar la filosofía de funcionamiento del WBG es necesario asegurar que el subgrafo se activará solo cuando se haya verificado la condición de disparo del nodo macro superior, y que todos los tokens de salida del mismo se generarán simultáneamente (es decir, no deben producirse tokens de salida "durante" la ejecución del nodo macro, si-

no solamente al final). Por tal motivo se han definido dos "clases" adicionales, denominadas INPUT y OUTPUT. Todo subgrafo interior a un nodo MACRO debe necesariamente comenzar con un nodo INPUT y finalizar con un nodo OUTPUT (figura 5.15). De esta forma aseguro que, macroscópicamente, los nodos MACRO se comportan como nodos estándar del WBG.

5.5.Editor de asignaciones.

Este editor permite crear o modificar la asignación estática de tareas a procesadores. La pantalla principal del mismo puede apreciarse en la figura 5.16. En la misma pueden apreciarse cuatro zonas:

-zona de grafo de programa.
-zona de grafo de arquitectura.
-zona de iconos.
-zona de información.

5.5.1.Asignación de tareas a procesadores.

En la zona de grafo de programa los nodos no asignados a procesadores se muestran en color azul. Para asignar una tarea el usuario efectúa un click sobre la misma. El cursor toma la forma del nodo seleccionado, que ahora se muestra en color verde. Luego simplemente lleva el cursor hasta situarlo sobre el procesador a quien la tarea será asignada, y efectúa un click. El cursor recupera su forma habitual (mano indicadora), y la tarea queda asignada. Si el nodo seleccionado es del tipo MACRO, el grafo actual del programa es reemplazado por el subgrafo interno del nodo, de modo tal que pueda efectuarse la asignación del mismo.

En cualquier momento el usuario puede consultar las tareas asignadas a un procesador, efectuando un click sobre el mismo. Esto produce que todas los nodos de programa asignados a ese procesador se muestren en color rojo.

Para anular la asignación de una tarea, se selecciona la misma en la forma habitual, pero en lugar de asignarla a un procesador se la lleva al icono con forma de papelera (figura 5.16). Para eliminar toda la asignación debe efectuarse un click sobre el icono "CLEAR". Los nodos serán liberados, y se dibujarán nuevamente en azul.

Esta forma interactiva de operación permite "sintonizar" rápidamente una asignación, favoreciendo un ciclo de "prueba y error", ya que el usuario puede pasar, en cualquier momento, del editor de asignaciones al simulador.

La edición de asignaciones se termina con un click en el botón derecho. El sistema mostrará entonces una tabla con el mapa de asignaciones generado.

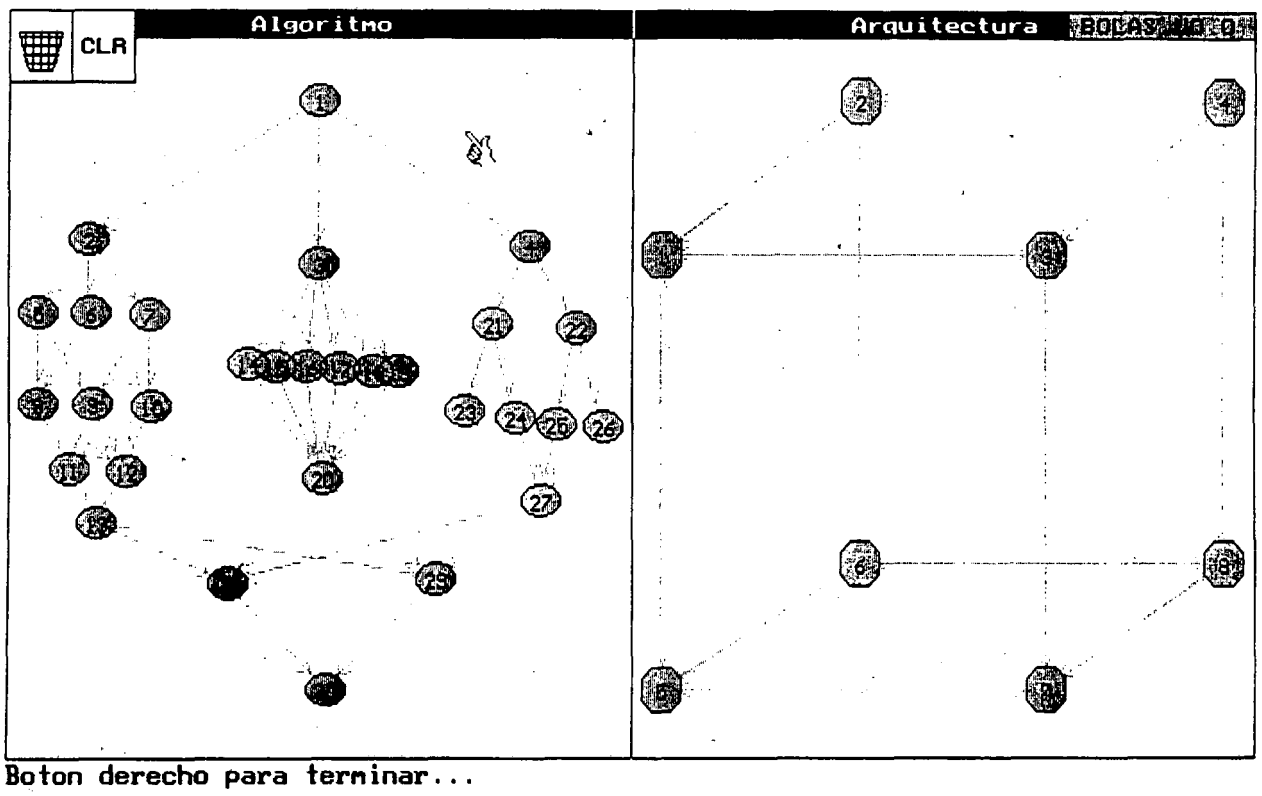


figura 5.16. Editor de asignaciones.

5.6.Simulador. Interfaz con el usuario.

En el capítulo anterior reseñamos las características de la máquina de simulación. En este apartado detallaremos la forma en que el simulador se comunica interactivamente con el usuario. Esta interfaz ha sido diseñada con la premisa de una operación simple e intuitiva.

5.6.1.Ingreso al simulador e inicialización de la sesión.

Una vez creados los grafos de entrada mediante el sistema de edición, se activa el simulador mediante la opción "simular" del menú principal. El sistema solicita entonces las opciones principales de simulación, como puede verse en la figura 5.17. La primer ventana de opciones permite decidir entre simulación del programa solamente, o del conjunto programa-arquitectura.

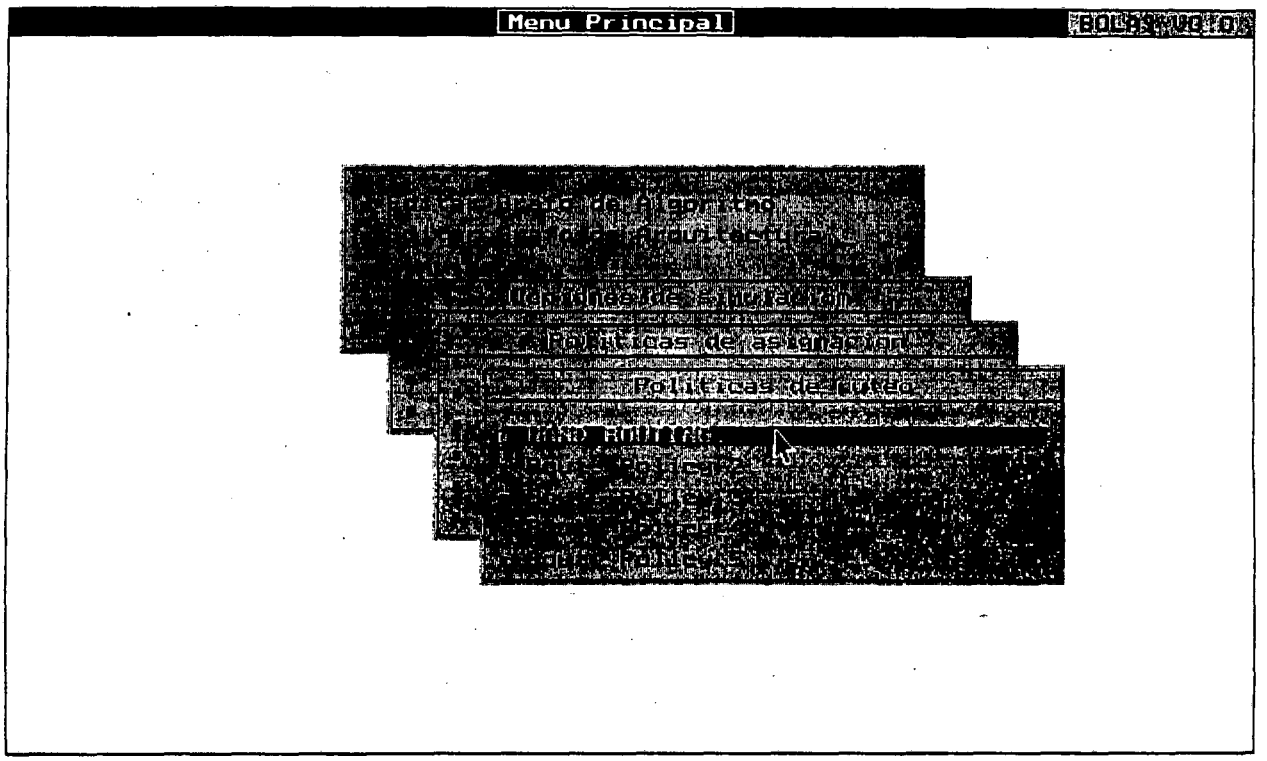


figura 5.17. Opciones de ingreso al simulador.

La segunda ventana es utilizada para decidir la política de asignación deseada (asignación manual, o diversos tipos de asignaciones estáticas y dinámicas). La tercer ventana permite elegir la política de ruteo de mensajes (ruteo manual o distintas versiones de ruteo automático).

5.6.2.Pantalla de simulación.

Según puede apreciarse en la figura 5.18, en la pantalla del simulador pueden reconocerse las siguientes zonas:

-Zona de Gráficos.
-Zona de menúes y controles.
-Zona de información.
-Zona de diálogo.

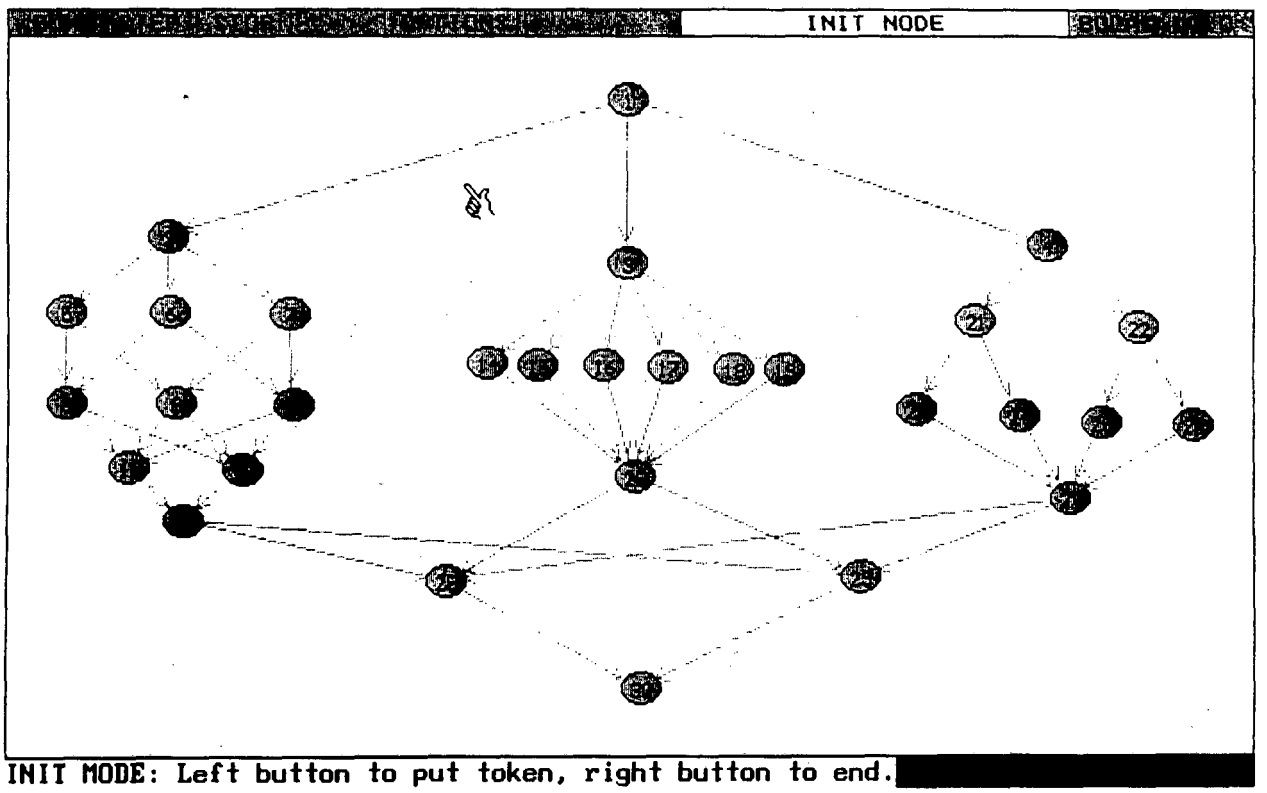


figura 5.18. Pantalla de simulación

En la zona de gráficos se ilustra el grafo del programa a nivel de nodo 0. Este grafo es animado en tiempo de simulación, mostrando en diferentes colores los nodos habilitados para su ejecución, los nodos actualmente en ejecución y su número de copias o instancias, y el paso de tokens por los arcos. El grafo está normalizado para que siempre quepa en las dimensiones de esta pantalla.

La zona de menúes y controles permite definir las opciones de la sesión de simulación. Las cinco "llaves" de control permiten iniciar y detener la simulación en modo "contínuo" o "paso a paso", y controlar la velocidad de simulación. El menú de opciones permite establecer el tiempo máximo de simulación, especificar un archivo de traza, y activar/desactivar la animación gráfica.

La zona de información expone, en todo momento, el "estado de la máquina de simulación. Los diferentes estados se analizarán mas adelante.

La zona de diálogo se utiliza para el intercambio de información entre el operador y el simulador (parámetros para los comandos, sincronización, etc).

5.6.3. Inicialización del grafo.

Una vez elegidas las opciones de apertura, la situación será la ilustrada en la figura 5.18. La zona de información presenta el mensaje "INIT NODE", indicando que el sistema espera la inicialización del grafo. El cursor tiene la forma de una mano indicadora. Para depositar un token en un nodo, se posiciona el cursor sobre el mismo y se efectúa un click. El cursor desaparece y en la zona de diálogo tendremos el siguiente mensaje:

Init alg. node number [NUM]. Token count?:

El usuario ingresa entonces el número de instancias paralelas del nodo que desea activar. El nodo cambiará de color (de azul a verde) para indicar que está habilitado, y el número de instancias activas se exhibirá en el borde inferior derecho del nodo (figura 5.19). El cursor se hace nuevamente visible, y puede continuarse el proceso de inicialización.

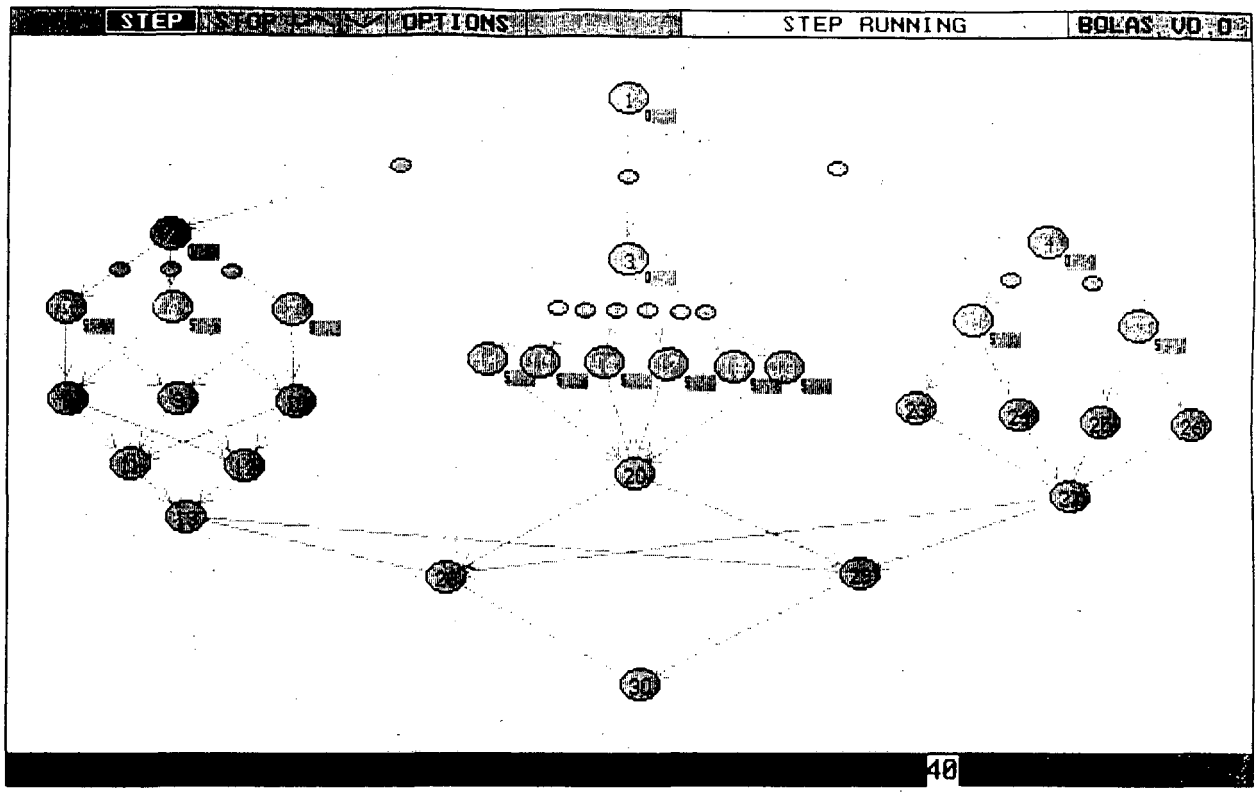


figura 5.19. Pantalla durante simulación.

Par finalizar la inicialización, se efectúa un click sobre el botón derecho. El cursor será ahora el de selección, y en la zona de información se exhibirá el mensaje WAITING, indicando que el sistema espera alguna orden desde la zona de menúes y controles.

5.6.4. Opciones de simulación.

El menú OPTIONS de la zona de menúes y controles (figura 5.20) permite elegir algunos parámetros adicionales de la sesión de simulación. Al seleccionar esta ventana, el mensaje SET OPTIONS aparecerá en la zona de información.

5.6.4.1. Tiempo de simulación máximo.

Mediante la opción "Set stop time", se programa el tiempo de finalización de la sesión. Normalmente este valor es máximo, es decir, la simulación continuará indefinidamente, o hasta que no existan mas eventos simulables. El tiempo máximo puede modificarse cuantas veces se desee, incluso durante tiempo de simulación.

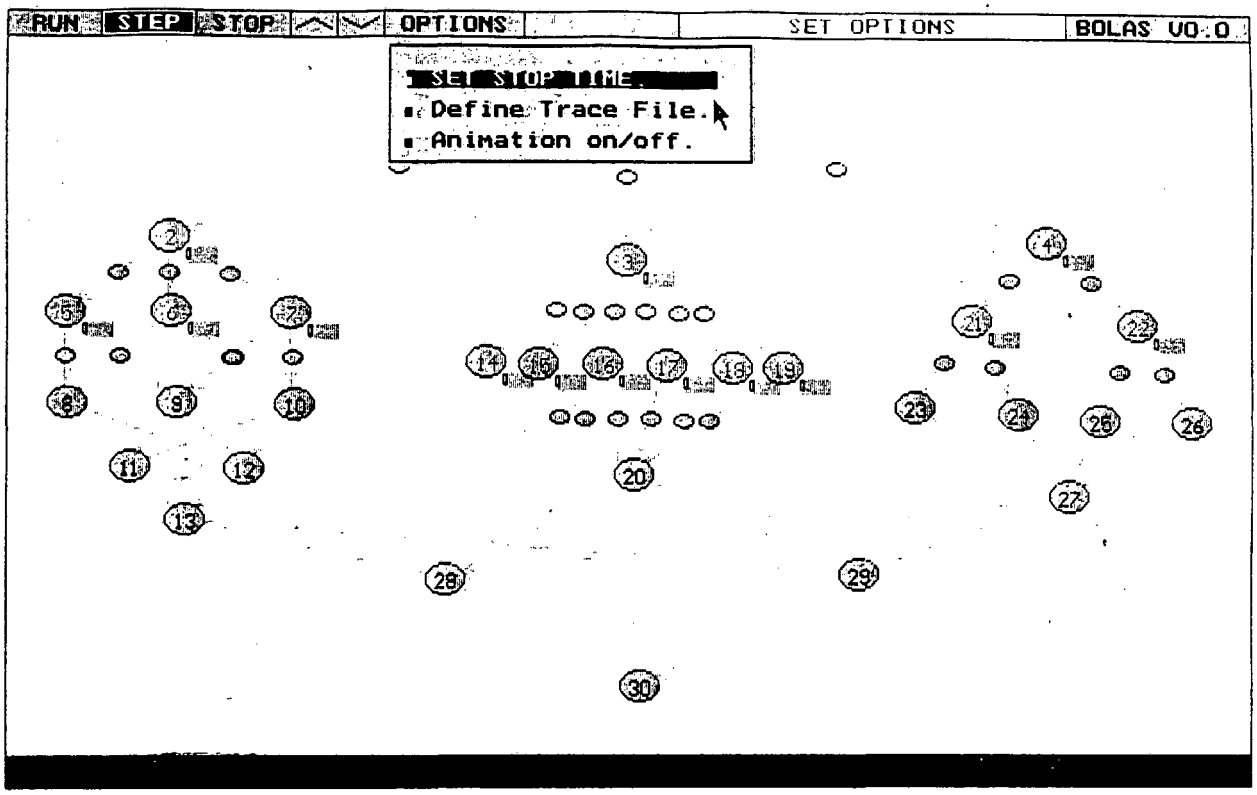


figura 5.20. Opciones de simulación.

5.6.4.2.Archivo de traza.

Esta opción (Define trace file) permite almacenar los eventos de la simulación en un archivo externo para su posterior procesamiento. Por defecto, no se realiza almacenamiento alguno. Cada elemento del archivo de traza es un evento, identificado por su tiempo asociado, una identificación y un conjunto de parámetros. Por ejemplo, el evento del tipo "fin de comunicación" tiene como parámetros los nodos fuente-destino del grafo de programa, y los nodos fuente- destino del último paso de ruteo.

5.6.4.3.Animación activada o desactivada.

Mediante esta opción (animation ON/OFF) se pueden habilitar o deshabilitar los gráficos animados de pantalla, con lo que la máquina de simulación se acelera aproximadamente diez veces. En simulaciones largas sobre modelos ya depurados, este comando permi-

te reducir notablemente el tiempo de ejecución de la sesión. Este modo de operación puede modificarse cuantas veces se desee, incluso durante la simulación.

5.6.5. Llaves de control.

Mediante las llaves de control el operador puede configurar interactivamente las características de la sesión de simulación en curso. Se accede a las mismas a través de la zona de menúes y comandos (figura 5.18).

5.6.5.1. Modo contínuo.

Para iniciar la simulación, el usuario efectúa un click sobre el cartel RUN de la línea de comandos. El grafo comenzará a ejecutarse en modo contínuo, el cursor desaparecerá y el mensaje en la zona de información será FREE RUNNING. La simulación puede detenerse temporalmente mediante un click. Esto produce la aparición del cursor de selección y el mensaje WAITING en la zona de información. Ahora puede efectuarse cualquier operación sobre la zona de menúes y controles. Para continuar simplemente se hace un click sobre el cartel RUN.

5.6.5.2. Control de la velocidad de animación.

Cuando se trabaja con grafos animados, es muchas veces útil poder controlar la velocidad a la que los mismos son actualizados. Esto se realiza mediante los dos controles de velocidad de la línea de menúes y controles. La flecha hacia arriba aumenta la velocidad y la otra la disminuye. Cuando se trabaja sin animación el simulador siempre trabaja a velocidad máxima.

Para aumentar la velocidad, se posiciona el cursor sobre la flecha ascendente y se presiona el botón izquierdo del ratón. En la zona de información aparecerá el mensaje SPEED UP. La velocidad aumentará hasta que deje de presionarse el botón, o hasta que se alcance la velocidad máxima. En ese caso el cartel será MAX. SPEED. Para disminuir la velocidad el procedimiento es similar. Ahora los carteles serán SPEED DOWN y MIN. SPEED, respectivamente.

5.6.5.3. Modo paso a paso.

Para ejecutar el grafo en modo paso a paso (un evento a la vez), se efectúa un click sobre el cartel STEP. El cursor desaparecerá y el mensaje en la zona de información será STEP RUNNING. El tiempo de simulación avanzará hasta el tiempo del próximo evento ejecutable y luego el simulador esperará hasta que el usuario presione cualquier tecla. De esta forma puede depurarse rápidamente la operación del grafo.

Al igual que en el caso anterior, la simulación puede detenerse mediante un click. El usuario puede conmutar entre operación contínua o paso a paso cuantas veces lo desee, y asimismo modificar cualquiera de las opciones disponibles.

5.6.6. Ruteo manual.

Si no se ha especificado una rutina de ruteo automático, o si la misma es incapaz de resolver algún conflicto, el sistema solicita la intervención del usuario. En este caso el sistema conmuta de la pantalla normal de simulación a una como la que se aprecia en la figura 5.21.

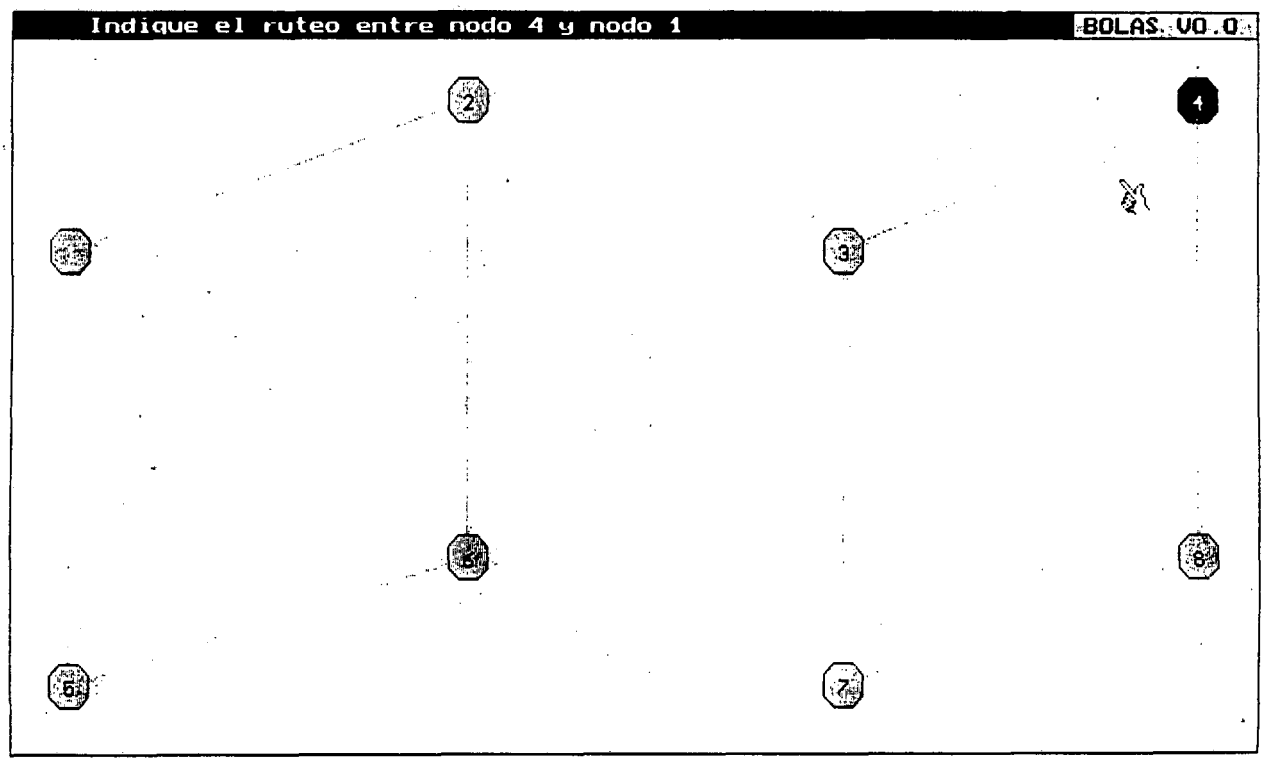


figura 5.21. Ruteo manual.

En ella se muestra el grafo de arquitectura, con los nodos fuente (4) y destino (1) del ruteo no resuelto en diferente color. Para indicar el ruteo, el usuario efectúa clicks sobre los sucesivos pasos intermedios. El sistema verifica cada paso, y si es correcto, "ilumina" el nodo en cuestión. Una vez se ha completado la trayectoria, se vuelve a la pantalla de simulación mediante un click en el botón derecho. Este procedimiento será repetido tantas veces como sea necesario para completar la ejecución del grafo de programa.

5.6.7.Finalización de la sesión

Existen tres causas que pueden detener al simulador. La primera es, obviamente, la ausencia de actividad en el modelo. La segunda es que se haya superado el tiempo de simulación máximo especificado en el menú de opciones. La tercera es que el usuario efectúe un click sobre el cartel STOP de la zona de menúes y controles. En los tres casos, el simulador informará al usuario el tiempo del ultimo evento ejecutado. Un click devuelve al usuario al menú principal.

5.7.Presentación de resultados "off line".

La presentación de resultados de una sesión de simulación se realiza a partir de la definición de un "instrumento". Un instrumento es un sistema interactivo de información gráfica, y contiene la información sobre las variables a medir a partir del archivo de traza de eventos, así como la forma en que deben ser presentadas. La figura 5.22. muestra la pantalla típica de un instrumento. En ella pueden apreciarse sus tres partes fundamentales: la zona de menúes, la de paneles y la de diálogo.

La zona de menúes permite acceder a las funciones provistas por el instrumento. Las funciones predefinidas permiten efectuar las siguientes mediciones:
-Tiempo de ejecución en función del número de procesadores (speedup).
-Latencia de mensajes en la arquitectura.
-Utilización de procesadores en función del tiempo.
-Congestión de los links de la arquitectura versus tiempo.
-Paralelismo del programa (número de nodos del programa activos versus tiempo).
-Traza de eventos.

Cada medición se ilustra en un "panel", que el usuario puede mover, cambiar de escala y eliminar mediante el ratón. Esta filosofía permite administrar en forma ordenada gran cantidad de información. El usuario puede también combinar en un sólo instrumento los resultados de varias sesiones de simulación, asi como determinar sus propias rutinas para el procesamiento del archivo de traza. La salida de estas rutinas puede ser luego incorporada a un panel.

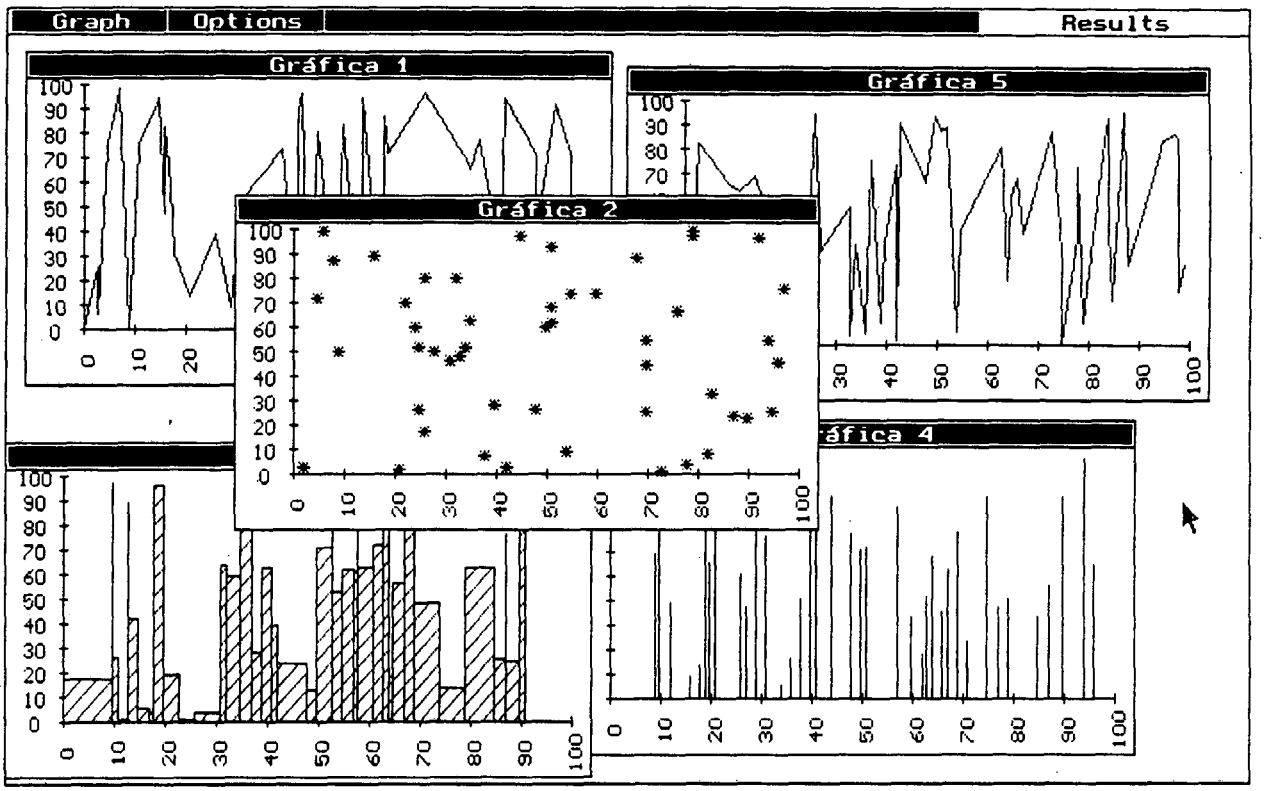


figura 5.22. Pantalla de instrumento

5.8.Resumen.

Esta capítulo detalla las características del ambiente interactivo de simulación. Se analizan las diferentes partes del sistema: edición gráfica, asignación de tareas y simulación. Cada subsistema se describe desde el punto de vista del usuario, a nivel de su interfaz con el sistema. Los detalles prácticos del sistema puede verse en el anexo A, que contiene los listados de la implementación en Pascal para un sistema MS-Dos.

Capítulo 6.

Una aplicación.

En este capítulo se describirá la utilización del ambiente integrado de simulación en una aplicación práctica. Analizaremos la implementación de un scheduler estático basado en el método "Simulated Annealing" de optimización combinatoria. El capítulo comienza con una descripción detallada del método y de los parámetros que describen su comportamiento. Luego se detallarán las decisiones adoptadas para su aplicación en asignación de tareas a multiprocesadores. Finalmente, tomando un grupo de grafos característicos, se realizará un estudio por simulación a fin de sintonizar los parámetros del algoritmo a una arquitectura específica, en este caso un hipercubo de tres dimensiones.

6.1.El método.

Simulated Annealing (SA) es una técnica de optimización combinatoria de propósito general. Es una extensión del método de Monte Carlo desarrollado por Metrópolis et al.[52] para determinar el estado de equilibrio de un conjunto de partículas a una dada temperatura. El método se basa en la analogía existente entre el proceso de recocido, en el que un material es calentado y enfriado muy lentamente, y la solución de problemas de optimización combinatoria.

Cuando un material está caliente, sus partículas pueden moverse aleatoriamente. En este estado de alta energía, existe igual probabilidad de que los átomos se muevan en cualquier dirección. Si la sustancia comienza a enfriarse muy lentamente, sus partículas tienden a

organizarse a medida que pierden movilidad, llegando a obtenerse una estructura cristalina perfectamente ordenada, que corresponde al estado de mínima energía termodinámica (solución) del sistema. Si el material se enfría rápidamente, queda en un estado amorfo o policristalino, de energía mayor (un mínimo local). La principal característica del procedimiento SA reside en su capacidad de explorar todo el espacio de soluciones de un problema permitiendo la adopción controlada de cambios que empeoran el resultado, de forma tal de minimizar la probabilidad de quedar "atrapado" en un mínimo local. Estas soluciones "peores" son aceptadas en función de un parámetro denominado habitualmente "temperatura", que hace este tipo de soluciones menos y menos probables a medida que nos acercamos al fin del proceso.

Al utilizar SA para problemas de optimización combinatoria, el parámetro a minimizar (la energía en el caso termodinámico) es una cierta función que calcula o estima el costo de la solución obtenida. El algoritmo puede apreciarse en la figura 6.1. Se comienza con una configuración inicial S_0, con un costo C_0. Se genera una nueva configuración S, mediante una perturbación de S_0, con un nuevo costo C. Si el cambio en la función de costo $\Delta C = C - C_0$ es menor que cero (el nuevo costo es menor que el anterior), la nueva solución es aceptada. Si no es así, el cambio se acepta de acuerdo a una probabilidad dada por la función de Boltzmann $e^{(-\Delta C/T)}$. La temperatura T del sistema es disminuída a medida que el algoritmo progresa, haciendo que la probabilidad de aceptar una solución que empeora el costo sea cada vez mas pequeña.

Por lo tanto, el algoritmo puede caracterizarse por los siguientes parámetros:

i) La perturbación que genera las nuevas configuraciones.(*New_Mapping* en la figura 6.1).
ii) El criterio con el cual se acepta una solución peor. En nuestro caso se ha dado explícitamente (se aceptan con una probabilidad $e^{-(\Delta C/T)}$).
iii) La función mediante la cual se actualiza la temperatura.(*Update* en la figura 6.1).
iv) La condición utilizada para evaluar si se ha alcanzado el equilibrio a una dada temperatura, también conocida como "condición de lazo interno". (*Equilibrium* en la figura 6.1).
v) La condición de finalización, lazo externo o "congelamiento" del proceso (*End_condition,* fin del algoritmo).

```
Begin
Generate initial configuration (T, S, C);
While (not end condition) do
Begin
  While (not equilibrium (T)) do
  Begin
    New mapping (S, Snew);
    Cnew : = Cost (Snew);
    DeltaC : = Cnew - C;
    if DeltaC < 0 then
    Begin
      S : = Snew;
      C : = Cnew;
    end
    Else begin
         R : = random(100) / 100;
         if R < exp (-DeltaC / T) then
         begin
           S : = Snew;
           C : = Cnew;
         end;
       end;
  end;
Tnew : = Update (T);
end;
end;
```

figura 6.1. El procedimiento Simulated Annealing.

En las secciones siguientes se detallarán las decisiones adoptadas para cada uno de estos puntos al aplicar esta técnica al problema de la asignación estática de tareas a procesadores.

Es intersante notar que, si la temperatura se fija en infinito (la probabilidad de aceptar una solución peor es 1), el algoritmo se convierte en una enumeración aleatoria de todas las soluciones posibles ("Random search"). Por el contrario, si T se fija a cero, el algoritmo simplemente busca el primer mínimo local ("local minima search") y queda "atrapado". La función de Boltzmann pretende ser una solución de compromiso entre estas dos alternativas.

Por otra parte, para una cierta temperatura T (es decir, dentro de una de las iteraciones del lazo interno), los cambios mas pequeños, esto es, los de menor ΔC, tienen mayor probabilidad de ser aceptados que los grandes. Esto nos asegura que las soluciones peores aceptadas se encuentren en un entorno del punto actual. Otra característica interesante es que si dos soluciones consecutivas tienen el mismo costo (ΔC nulo, probabilidad 1) el cambio es aceptado. De esta forma, para una temperatura dada, el algoritmo prueba varias alternativas de igual costo.

6.2.Simulated Annealing y asignación estática.

En nuestro caso, el problema podría resumirse como sigue: Tenemos un grafo para el programa paralelo, con N nodos, y un grafo de arquitectura con P procesadores. La idea es encontrar una asignación estática de las N tareas a los P procesadores tal que se minimice una figura de mérito C. Como detallamos en el capítulo 3, éste es un problema NP completo. Para reducir el espacio de búsqueda de una solución sub-óptima utilizaremos la técnica de Simulated Annealing. Para ello es necesario tomar una decisión con respecto a cada uno de los cinco parámetros especificados mas arriba.

Como configuración inicial S0 se adoptará una asignación aleatoria de tareas a procesadores. La función de costo C está formada por dos términos:

$$C := k_i * c_i + k_c * c_c \qquad (1)$$

donde c_i representa el costo debido al desbalance en la carga de los procesadores, calculado como

$$c_i := \Sigma \text{ abs (carga de cada procesador - carga media)} \qquad (2)$$

y c_c es el costo en el que se incurre al comunicar dos tareas asignadas a diferentes procesadores:

$$c_c := \Sigma \text{ costo link } * \text{ numero de pasos de ruteo} \qquad (3)$$

Obsérvese que los dos factores son antagónicos, puesto que mientras el primer término tiende a distribuir la carga homogéneamente entre todos los procesadores (mínimo desbalance), el segundo tiende a asignar todas las tareas al mismo procesador para evitar las comunicaciones (suponemos que el costo de comunicar dos tareas asignadas al mismo procesador es cero, capítulo 3).

Dada una configuración inicial, existen N*(P-1) posibles cambios de una tarea a cualquiera de los otros procesadores. La temperatura inicial T0 se calcula de forma tal que la probabilidad de aceptar la media de las soluciones que empeoran el costo inicial es 0.9. Esto

supone realizar efectivamente los N*(P-1) posibles cambios, de forma tal de calcular la media de los costos mayores al inicial. Así pues,

T_0 = -(media de los costos mayores al inicial - costo inicial) / ln(0.9) (4)

Las nuevas asignaciones son generadas cambiando una tarea seleccionada aleatoriamente de un procesador a otro cualquiera.

Siguiendo a [52], la temperatura será actualizada mediante la ecuación

$T_{new} := 0.95*T$

La condición del lazo interno es normalmente adoptada como un número suficientemente alto NA de soluciones aceptadas. Se ha demostrado experimentalmente en [52] que, partiendo de valores bajos de NA (0.01 N), las soluciones mejoran progresivamente con NA hasta llegar a un punto donde el proceso se estabiliza. Aumentarlo por encima de este valor empírico es ineficiente ya que, como puede verse en el algoritmo de la figura 6.1, el tiempo de cómputo es directamente proporcional a NA.

La condición de un cierto número NA de soluciones aceptadas es muy difícil de cumplir a bajas temperaturas (las partículas son menos libres para moverse, al ser menor la agitación térmica), por lo que normalmente se agrega una cota al número total NT de soluciones generadas. Es decir, la temperatura se actualiza si se han aceptado NA soluciones, o se han intentado NT nuevas configuraciones.

La condición de finalización del algoritmo es una temperatura baja tal que la probabilidad de aceptar una solución peor es suficientemente pequeña (en nuestro caso, 2^{-31}).

6.3.Sintonización de parámetros.

Habiendo tomado estas decisiones, sólo resta elegir el valor de las constantes k_l y k_c de la ecuación de costo. Es decir, con qué peso relativo serán medidos el desbalance en la carga y el costo de comunicaciones entre procesadores. Usaremos el ambiente de simulación para guiar esta etapa de diseño para el caso de un hipercubo de tres dimensiones tal como el

ilustrado en la figura 6.2. Supondremos iguales performances relativas de cómputo y comunicaciones en todos los procesadores y links, respectivamente.

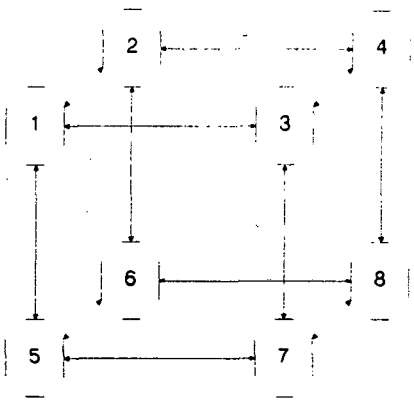


figura 6.2. Arquitectura utilizada.

En la figura 6.3. podemos apreciar los tres grafos seleccionados para nuestro estudio. El primero presenta una estructura altamente regular, el segundo tiene tres zonas claramente diferenciadas y el tercero es irregular. Para cada uno de los grafos se analizarán tres casos, donde la relación media entre volumen de comunicaciones de los arcos a volumen de cómputo de los nodos (Arcs-nodes cost ratio, ANCR) es 1, 2 y 5. Nuestro objetivo es analizar la influencia de esta relación con los mapeos obtenidos.

La figura 6.4 ilustra la relación tiempo de ejecución medio / número de procesadores para los tres grafos utilizados con ANCR = 1. En los tres casos se aprecia claramente el fenómeno de saturación, analizado en el capítulo 3. En este caso el mismo se produce por la fuerte penalización que sufren las comunicaciones en el hipercubo, y porque los grafos no presentan un paralelismo masivo durante toda su ejecución. De la simulación obtenemos que el número máximo posible de nodos paralelos para cada grafo es 11, 16 y 14, respectivamente, con valores medios de 3.7, 4.8 y 5 nodos.

Para cada valor de la relación k_c/k_i se realizaron diez mediciones, con NA = N y NT = 3N. En la tabla 6.1 puede observarse, para cada grafo y cada valor de la relación arco/nodo, los valores de costo medio, mejor y peor costo y varianza para $k_c/k_i = 1$.

La bondad de los mapeos se evalúa a través de su varianza. Según puede verse en la tabla 6.1, los resultados para $k_c/k_i = 1$ son altamente satisfactorios en los tres grafos con

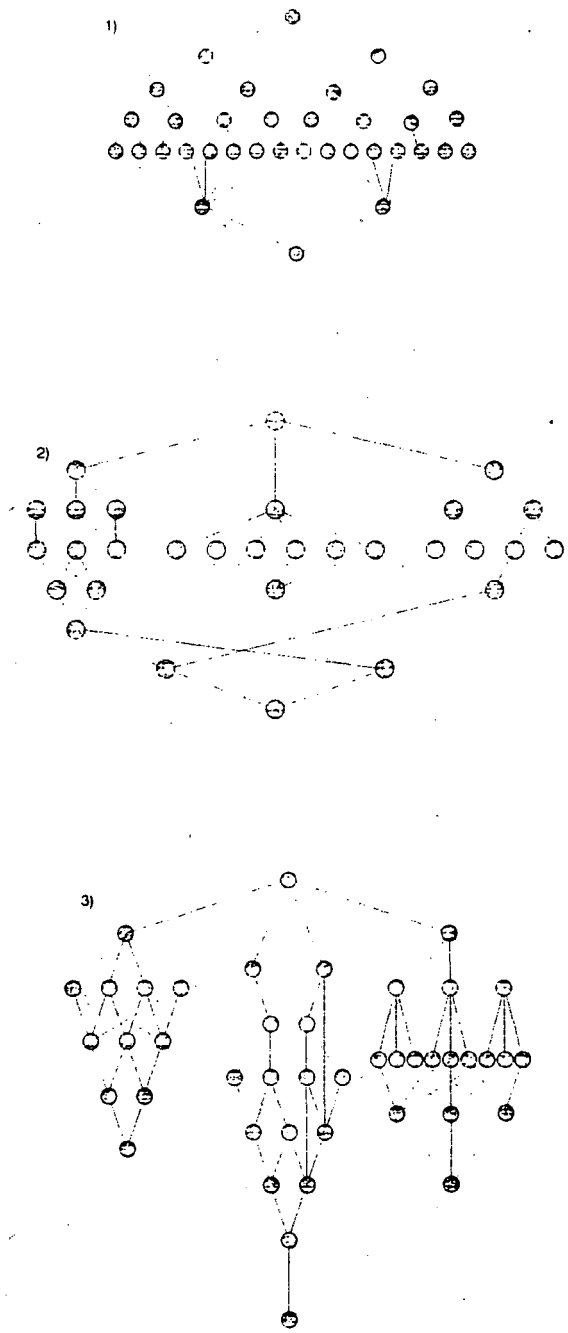


figura 6.3. Gráficos de prueba utilizados

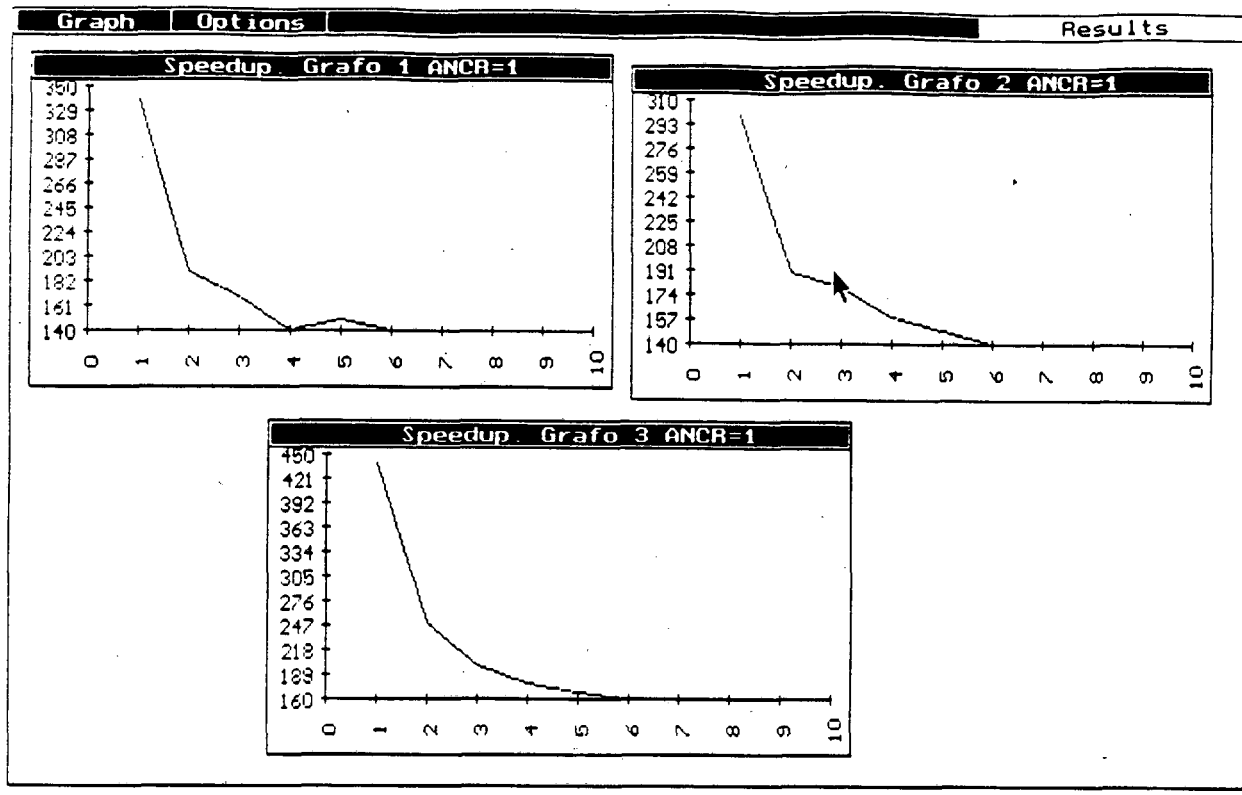


figura 6.4. Speedup para ANCR = 1

Simulated Annealing: Simulation Results for kl = kc = 1

Graph	ANCR	Worst Cost	Best Cost	Mean Cost	Variance
Graph 1	1	167	142	148	2.18
	2	272	232	244	4.30
	5	445	337	388	13.18
Graph 2	1	152	135	141	1.84
	2	280	245	257	3.52
	5	430	305	382	13.5
Graph 3	1	215	196	204	1.69
	2	355	331	346	2.45
	5	710	541	622	15.83

Tabla 6.1. Resultados para kc/ki = 1.

ANCR = 1. Una asignación típica para este caso puede verse en la figura 6.5. Como puede apreciarse, el algoritmo tiende a agrupar los nodos del grafo del algoritmo en conjuntos de aproximadamente la misma cantidad de nodos (balance de carga), directamente comunicados entre sí, y a asignar los grupos mas intercomunicados a procesadores adyacentes (minimizando las comunicaciones). En este caso, y para el hipercubo de tres dimensiones, para los 10 casos analizados, sólo 2 tenían algún camino con dos pasos de ruteo, y no existió ningún ruteo de tres pasos. Para este valor del ANCR se verifican los menores tiempos de ejecución.

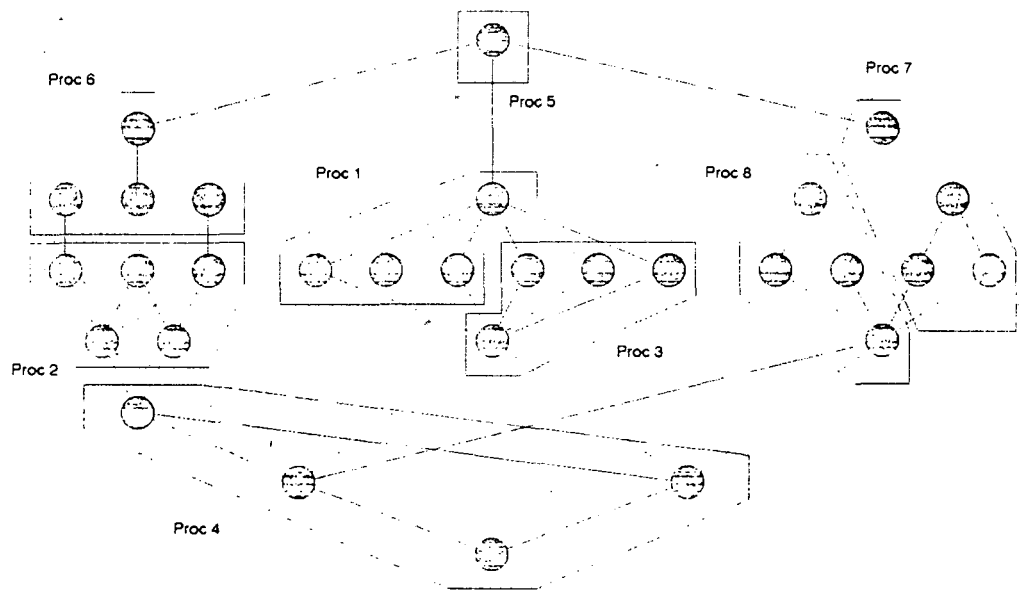


figura 6.5. Asignación típica para ANCR = 1, kc/ki = 1.

Sin embargo, para los casos ANCR = 2 y ANCR = 5 el resultado empeora sensiblemente. La razón es obvia: el mayor peso en las comunicaciones, debido al incremento del volumen de los arcos, hace que el algoritmo de asignación tienda a generar conjuntos grandes de tareas, con el consecuente desbalance en la carga. Siguiendo con el ejemplo anterior, la figura 6.6 ilustra un mapeo típico.

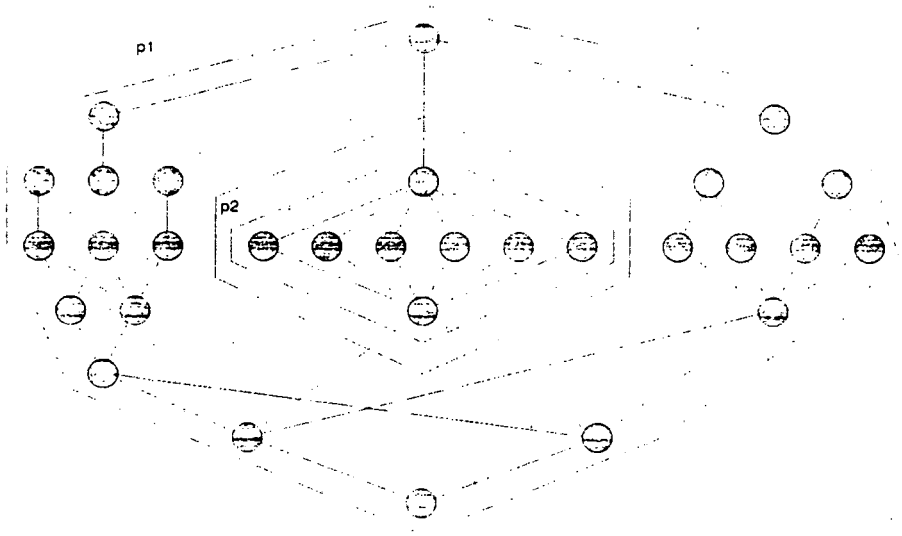


figura 6.6. Asignación típica para ANCR = 5, kc/ki = 1.

La conclusión es que el método es altamente dependiente de la relación ANCR. La pregunta es: Pueden modificarse los valores de los pesos de la ecuación de costo kc y ki de forma tal de compensar esta dependencia?

Podemos comprender un poco mejor el comportamiento del algortimo mediante la tabla 6.2. En ella se muestran, para 10 ejecuciones, los valores de costo total, costo de desbalance y de comunicación para el grafo 2 con ANCR = 5, para valores de k_c/k_i = 1 (parte superior) y k_c/k_i = 1/5. Obsérvese que los costos de comunicaciones son prácticamente iguales en ambos casos, pero los costos por desbalance son aproximadamente 4 veces menores en el caso k_c/k_i = 1/5. Aparentemente, para el caso k_c/k_i = 1 el algoritmo queda atrapado en un mínimo local, en el que se ha minimizado el costo de comunicaciones, pero el costo por imbalance es aún muy alto. El resultado mejora considerablemente en el caso k_c/k_i = 1/5, al aumentar el peso relativo del factor desbalance en la ecuación de costo.

La figura 6.7 muestra un "instrumento" en el que se aprecia, para el grafo 2, la varianza de los mapeos para valores de la relación k_c/k_i entre 1/5 y 1. El comportamiento observado, que se repite para los otros dos grafos, indica que una posible solución al problema consiste en mo-