Verificación formal del algoritmos de distribución de carga para procesamiento de datos independientes

Para obtener el grado de
Maestro en Ciencias
(Ciencias y Tecnologías de la Información)

P R E S E N T A

José Luis Quiroz Fabián

Asesores

Dr. Graciela Román Alonso
Dr. Manuel Aguilar Cornejo

Sinodales

Presidente: Dr. Ricardo Marcelín Jiménez
Secretario: Dr. Manuel Aguilar Cornejo
Vocal: Dr. José Oscar Olmedo Aguirre

21 de abril de 2007

UNIVERSIDAD AUTÓNOMA METROPOLITANA
MAESTRÍA EN CIENCIAS Y TECNOLOGÍAS
DE LA INFORMACIÓN

Casa abierta al tiempo

VERIFICACIÓN FORMAL DE ALGORITMOS DE DISTRIBUCIÓN DE CARGA PARA PROCESAMIENTO DE DATOS INDEPENDIENTES

PROYECTO PARA OBTENER
EL GRADO DE MAESTRO EN CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN

Postulante: **José Luis Quiroz Fabián**
Asesores: Dra. Graciela Román Alonso
Dr. Manuel Aguilar Cornejo

México - DF
Abril 2008

Resumen

Los algoritmos de distribución de datos (carga) de procesamiento independiente proveen un conjunto de beneficios a las aplicaciones paralelas tales como: la minimización de su tiempo de ejecución, la maximización de uso de los recursos, etc. Pero por su naturaleza paralela, la implementación de un algoritmo de distribución de datos es compleja lo que puede originar que no cumpla con las especificaciones para las que fue diseñado presentando problemas como: violación a la exclusión mutua, no terminación de la ejecución paralela, abrazos mortales, etc.

En esta tesis de maestría, como primer etapa, se propone, modela y verifica formalmente una estructura básica que integra un algoritmo de distribución cíclico en una aplicación SPMD (Simple Program Multiple Data) de procesamiento de datos independientes. Para este proceso de verificación, auxiliándonos de la lógica temporal, se propone un conjunto de propiedades que reflejan un buen funcionamiento del sistema independientemente del algoritmo de distribución usado. La herramienta de verificación utilizada fue Spin, la cual aplica la técnica de verificación de model checking (un método que permite verificar algoritmos paralelos con un espacio de estados finito) y nos permite obtener un diagnóstico del cumplimiento de las propiedades. El sistema se modeló mediante el lenguaje *promela* utilizado por Spin, realizando la verificación de todas las propiedades especificadas.

En la segunda etapa de este proyecto se propone un modelo *promela* para la verificación de la herramienta DLML (Data List Management Library) basándonos en la estructura propuesta en la primera etapa. DLML es una librería creada en el Laboratorio de Sistemas Distribuidos y Paralelos de la UAM-I para distribuir la carga (datos) generada por las aplicaciones, de manera transparente para el programador.

Para verificar DLML se valida el cumplimiento de las propiedades propuestas y de dos nuevas propiedades relacionadas a la implementación. Después de verificar el modelo de la versión original de DLML se proponen nuevas implementaciones de este distribuidor, donde se contemplan aspectos como la capacidad de los canales, la no dependencia en la granularidad de los datos y la disminución de la cantidad de mensajes (y con ello mejorar su rendimiento).

La versión que muestra mejor rendimiento, nombrada DLML-híbrido, es verificada utilizando las mismas propiedades que se verificaron sobre la versión original de DLML. Además se realiza su implementación en lenguaje C-MPI en la que se incorpora el uso de memoria compartida e hilos de ejecución, aprovechando el surgimiento de las nuevas arquitectura multicore/multiprocesador (donde se tiene más de un núcleo (procesador) por nodo).

El trabajo de la presente tesis concluye mostrando una comparación de rendimiento de la versión DLML-híbrido con la versión original de DLML, para la cual se utilizaron aplicaciones que manejan datos de granularidad diferente (fina y gruesa). En la comparación se efectúa al utilizar un cluster de 32 nodos de tipo multiprocesador (4 procesadores por nodo, teniendo un total de 128 procesadores). Para la aplicación de granularidad fina DLML-híbrido presenta mejores tiempos de ejecución mientras más procesadores se utilizan. La versión DLML original para este caso obtiene mejores tiempos de ejecución con un número menor de procesadores y pierde eficiencia cuando el número de nodos aumenta. Para la aplicación de granularidad gruesa la versión DLML- híbrido siempre obtiene mejores tiempos de ejecución que la versión original.

Agradecimientos

A mis padres, **María** y **Moisés**, les agradezco su apoyo, su guía y su confianza en la realización de mis estudios. Soy afortunado por contar con su amor y apoyo.

A mis hermanos, **Angélica**, **Claudia**, **Israel** y **Moisés**, por estar siempre a mi lado y en especial a mi hermano Moisés a quien tuve la fortuna de conocer 16 años de mi vida.

A mis asesores la **Dra. Graciela Román Alonso** y el **Dr. Manuel Aguilar Cornejo**, por su constante apoyo y asesoramiento en todos los aspectos de la investigación y elaboración de esta tesis así como por la confianza depositada en mi.

Al **Dr. Miguel Alfonso Castro García**, por el apoyo y consejos brindados durante mi estancia en la maestría así como por la facilitación de la infraestructura necesaria para la realización de esta tesis.

A la **Dra. Elizabeth Pérez Cortés**, quien es sin una de las personas que más ha contribuido en mi formación profesional

A la **UAM (Universidad Autónoma Metropolitana)**, por la beca que me otorgó durante mi estancia en la maestría.

En general a todos los **integrantes de la Maestría en Ciencias y Tecnologías de la Información**, amigos y profesores, sin su ayuda no estaría donde me encuentro ahora.

Índice general

Índice de figuras

Capítulo 1

Introducción

"A veces sentimos que lo que hacemos es tan sólo una gota en el mar; pero el mar sería menos si le faltara una gota."

–Madre Teresa de Calcuta 1910-1997 Religiosa católica yugoslava

En las siguientes secciones de este capítulo, damos una breve descripción del cómputo en paralelo, su necesidad y problemáticas. Comenzamos explicando las partes que se deben tomar en cuenta cuando se realiza cómputo en paralelo, tales como el modelo de comunicación y particionamiento, posteriormente describimos el por qué de la necesidad de la distribución dinámica de carga y las problemáticas que se pueden presentar cuando se desea construir algoritmos paralelos correctos. Explicamos la necesidad de la verificación formal para poder garantizar que un programa que integra la distribución dinámica de carga cumpla con sus especificaciones, reduciendo así, en la medida de lo posible los errores de implementación. Posteriormente presentamos los objetivos, la justificación y metodología seguida en esta tesis para su realización.

1.1. Cómputo en paralelo: una necesidad

La computación tiene su origen en el deseo del hombre por efectuar operaciones matemáticas cada vez mas rápidas y eficientes. Ésto ha llevado a la evolución de las computadoras para poder aumentar sus recursos minimizando su espacio y aumentando su rendimiento.

La computación paralela aparece a raíz de la necesidad de reducir tiempos de ejecución de aplicaciones de cómputo intensivo dando como consecuencia el desarrollo de computadoras con más de un procesador. La computación paralela la podemos definir como el uso simultáneo de múltiples recursos computacionales para resolver un problema en común [MM01].

Los recursos computacionales pueden estar disponibles mediante:

- Una máquina multiprocesador[1].
- Un conjunto de máquinas unidas por un enlace de comunicación (multicomputadoras).
- Una combinación de ambos.

[1]Computadora con más de un procesador compartiendo memoria.

El alto costo de las máquinas multiprocesador ha llevado a la creación de sistemas construidos mediante la interconexión de componentes de hardware comunes. Estos sistemas reciben el nombre de *clusters*.

Un algoritmo que consiste en un conjunto de tareas que pueden ser ejecutadas simultáneamente por distintas unidades de procesamiento cooperando para resolver un problema, se denomina algoritmo paralelo[MM01]. Los algoritmos paralelos difieren de los algoritmos secuenciales en la incorporación de primitivas y/o funciones para la creación, comunicación y sincronización entre procesos (tareas).

Principalmente existen dos modelos de comunicación para la construcción de algoritmos paralelos, el modelo de paso de mensajes y el modelo de memoria compartida.

En el modelo de comunicación por paso de mensajes se tienen básicamente dos operaciones: $send(mensaje)$ y $receive(mensaje)$. En este modelo es necesario el conocimiento de los siguientes conceptos:

- Canal. La cardinalidad (el número de entidades *n* y *m*, donde *n* son las entidades en un extremo del enlace y *m* en el otro involucradas en la comunicación), capacidad y dirección del canal.
- Mensaje. La identificación del proceso emisor/receptor, tamaño del mensaje a transmitir, tipo o etiqueta del mensaje.
- Tipo de evento. Síncrono o Asíncrono[CA00].

Una de las principales problemáticas al desarrollar algoritmos paralelos bajo el modelo de envío y recepción de mensajes es que cada recepción debe tener la invocación de su correspondiente envío. En otro caso puede ocurrir un interbloqueo si es una comunicación síncrona.

En el modelo de memoria compartida, para lograr la comunicación entre procesos se definen variables de uso común, las cuales reciben el nombre de *variables compartidas*. La sección del programa donde se manipulan las variables compartidas se denomina *sección critica*. Utilizar variables compartidas por diferentes procesos puede originar conflictos. Si un proceso escribe mientras otro escribe o lee simultáneamente en la misma variable compartida, pueden obtenerse resultados incorrectos, por lo que, estas variables se deben accesar en *exclusión mutua*. Con la exclusión mutua, sólo un proceso a la vez puede manipular con fines de escritura o lectura una variable compartida. Para implementar lo anterior se utilizan mecanismos de sincronización tales como semáforos, barreras, candados, etc[RR95].

La problemática principal al desarrollar algoritmos paralelos bajo el modelo de comunicación de memoria compartida es, al igual que en el modelo de paso de mensajes, el interbloqueo, si un proceso no libera una sección critica accesada de manera exclusiva.

Independientemente del modelo de comunicación usado, la implementación de un algoritmo paralelo puede presentar tres posibles métodos de particionamiento[2][Fos95]:

- El *particionamiento de código* consiste en dividir el código de un programa en varios programas o tareas que se ejecuten simultáneamente sobre diferentes procesadores. Cada tarea trabaja sobre un conjunto diferente de datos. Es posible que las tareas se creen antes o durante la ejecución . Si las tareas son diferentes se habla de una programación MPMD (Multiple Program Multiple Data).

[2]Forma de descomponer el problema en tareas más pequeñas para que se ejecuten en paralelo

- En el *particionamiento de datos* se busca distribuir dinámicamente los datos a procesar entre los procesadores y, cuando cada uno termina con el procesamiento que le corresponde envía sus resultados a un nodo central. Por lo general el particionamiento de datos se usa con aplicaciones que constan de un conjunto de tareas que ejecutan el mismo código SPMD (Simple Program Multiple Data).

- El tercer método es una combinación de los dos anteriores.

Típicamente, en las aplicaciones de cómputo paralelo se asigna una tarea por unidad de procesamiento, explotando lo más posible los recursos de un cluster. Cuando la cantidad de datos a procesar es una constante preestablecida y el cluster es utilizado de manera dedicada, es posible determinar un particionamiento y asignación de datos a cada tarea, de tal modo que se obtengan tiempos óptimos de ejecución[Kum02][Fos95].

Un problema surge cuando la cantidad y el volumen de datos a procesar cambia durante la ejecución de una aplicación. En este caso cada tarea genera dinámicamente datos a procesar, ocasionando que algunos procesadores puedan verse saturados mientras que otros estén descargados al terminar pronto su trabajo. El mismo problema surge cuando el cluster no es usado de manera dedicada, ya que de manera impredecible algunos procesadores pueden sobrecargarse al ejecutar simultáneamente aplicaciones de varios usuarios que ingresan al cluster.

En este caso surge la pregunta, ¿a qué tareas pueden transferirse los datos generados para evitar tener nodos sobrecargados? Para responder esta pregunta se han realizado algoritmos para distribuir de manera eficiente entre los procesadores el trabajo generado durante la ejecución de una aplicación. Estos algoritmos reciben el nombre de **algoritmos de distribución dinámica de carga** y forman parte de la familia de algoritmos paralelos. Los algoritmos de distribución de carga buscan metas tales como[RR96]:

- Aumentar el rendimiento.

- Minimizar el tiempo promedio de respuesta al tratar de deshacer cuellos de botella generados por un nodo sobrecargado funcionando como servidor.

- Minimizar el tiempo de ocio de los procesadores .

Por su diversidad, hay varias maneras de clasificar estos algoritmos, basándose en la arquitectura y las acciones que toman de acuerdo a diferentes eventos [LMR91] [JM93] [CK88].

1.2. Justificación

Como hemos mencionado, las aplicaciones han crecido en demanda de cómputo de manera considerable hoy en día, debido a esto investigadores de áreas como Física, Química, Biología, Redes, etc. han optado por realizar sus simulaciones de manera paralela.

La mayoría de las herramientas que se utilizan para hacer estas simulaciones (MPI [Han98] [BDV94] [SL03], TreadMarks[ACD+96], OpenMP [CDK+01] PVM[BDG+91], etc.) aplican estrategias de distribución muy sencillas y en algunos casos no las incorporan (RMI [RMI]). Debido a esto las herramientas que se utilizan no necesariamente ocupan todos los recursos de cómputo, de manera que, si bien los tiempos de respuesta son mejores que los algoritmos secuenciales, se

podrían mejorar aún mas incorporando una política de distribución cuando la carga es generada dinámicamente. Por otra parte, la integración de una política de distribución de carga en la aplicación puede aumentar la complejidad de la implementación debido a que ésta incluiría realmente la interacción de dos programas paralelos, el que accesa los datos para procesarlos y el que los accesa para distribuirlos. La implementación deberá garantizar que ambos programas cumplan con las especificaciones para los que fueron creados. El algoritmo final podría presentar problemas tales como:

- La violación a la exclusión mutua, es decir, que más de un procesador esté manipulando una variable compartida en un mismo instante.
- La no terminación de la ejecución paralela. Un algoritmo paralelo termina cuando se ejecutaron todos los últimos segmentos (tareas) de los procesos que lo conforman. Esto nos lleva a que si un proceso en un nodo no termina debido a un error de programación – por ejemplo que implemente cálculos infinitos– puede ocasionar que se bloqueen todos los demás y nunca se termine la ejecución paralela.
- Interbloqueos o abrazos mortales. Lo cual provocaría también la no terminación.
- Pérdida de mensajes (datos).
- Terminación anticipada, etc.

Los problemas anteriores se pueden deber a factores como una mala sincronización, inicialización incorrecta de variables, retardos en las comunicaciones, etc. Dado que los programas paralelos pueden tener un excesivo número de posibles trazas de ejecución, el programador en la mayoría de los casos incurre en errores sin darse cuenta.

Para evitar los problemas anteriores, existen varios métodos para tratar de garantizar el cumplimiento de un conjunto de propiedades sobre la implementación de un sistema computacional. Entre esos métodos tenemos las simulaciones, las pruebas, la verificación deductiva y el *model-checking*[PGS01][CGP99].

Las simulaciones y pruebas se basan en la ejecución de instancias del sistema. La simulación realiza esto a través de una abstracción del sistema, mientras que las pruebas lo realizan cuando el sistema ha sido terminado probando con un conjunto de entradas y observando las salidas que originan. Las simulaciones y pruebas no son exhaustivas por lo que su uso no garantiza que el programa cumple con los requerimientos deseados por lo cual utilizamos el *model-checking*.

La verificación deductiva busca demostrar el cumplimiento de propiedades en el sistema a través de un conjunto de axiomas y reglas de demostración. Esta técnica no es completamente automática (dadas sus restricciones teóricas) por lo que la persona encargada de la verificación del sistema (por lo regular un matemático o un lógico) debe disponer la suficiente capacidad de abstracción para demostrar una propiedad. Esto ha originado su escaso uso[PGS01].

El *model-checking* es un método cien por ciento automático, el cual, permite verificar algoritmos paralelos con un espacio de estados finito. Esta técnica emplea algoritmos muy eficientes de búsqueda y reducción de estados, los cuales, cuando se ejecutan con una suficiente cantidad de recursos de cómputo (procesador y memoria principalmente) reflejan muy buen rendimiento en la generación del diagnóstico.

Existen herramientas que sólo necesitan la especificación del programa a verificar en un lenguaje de modelado y las propiedades que hay que verificar en el programa para aplicar el método de *model-checking* y automáticamente proporcionar un diagnóstico [Hol04] [CGP02] [DDHY92] [BBD+02]. Una de las herramientas más utilizadas la cual se utiliza en este trabajo es *Spin* [Hol04].

Como se ha mencionado anteriormente, la integración de un algoritmo de distribución dinámica de datos en una aplicación específica de procesamiento de datos, es una buena opción para mejorar su rendimiento[Gar07]. Debido a que la implementación de dicha integración requiere la coexistencia del procedimiento de procesamiento y del procedimiento del distribuidor de datos, es importante garantizar un adecuado funcionamiento del sistema completo. Se requiere por ejemplo que los datos no se pierdan al ser distribuidos, que un dato sea procesado en un tiempo finito y que todos los procesadores cooperen (compartan carga) mientras haya datos que procesar posiblemente creados durante la ejecución.

Generalmente las propuestas que implementan algoritmos de distribución dinámica de carga, han sido validadas únicamente mediante métodos de simulación o pruebas [RLCB05], [CCGRA05] lo cual no implica que cumpla con todas sus especificaciones de funcionamiento. En esos trabajos no se estudia la estructura básica de interacción del distribuidor con la aplicación de procesamiento, sino más bien se dedican solamente a la descripción del algoritmo de distribución usado sin verificar formalmente el funcionamiento de su implementación.

A diferencia de otras propuestas, en este trabajo como **primera etapa** se propone una estructura básica de interacción de un algoritmo de distribución de datos con una aplicación de procesamiento independiente de datos. Posteriormente se identifican y se verifican formalmente ciertas propiedades que todo sistema de distribución debe cumplir independientemente del algoritmo de distribución usado.

Por simplicidad, en la primer etapa se eligió como algoritmo de distribución en la estructura básica un algoritmo cíclico. Éste algoritmo no considera la implementación de una política de información para recolectar estados de carga de todos los procesos del sistema. Con una topología cíclica, cada proceso se comunica únicamente con sus dos vecinos : sucesor y predecesor. Esta característica de limitar el número de posibilidades de interacción entre procesos favorece la verificación de sistemas más grandes.

Una herramienta de programación paralela que refina a la estructura básica es DLML. DLML (*Data List Management Library*) [Gar07][CCGRA05] es una de librería creada en el Laboratorio de Sistemas Distribuidos y Paralelos de la UAM-I para distribuir la carga (datos) generada por las aplicaciones. La distribución de carga se realiza de manera transparente para el usuario. La principal característica radica en el uso de un TDA (Tipo de dato Abstracto) lista el cual contiene los datos (carga) a distribuir de manera transparente. Las operaciones del TDA ocultan la distribución dinámica de carga la cual en general nos ayuda a mejorar el rendimiento global del sistema. Las operaciones de DLML se separan en 3 tipos: de señalización , de manipulación y de recopilación. En las operaciones de señalización se incluyen primitivas de inicialización y que conceden el control a un sólo proceso. Las de manipulación se encargan del manejo de los datos de la lista. Por último, las operaciones de recopilación sirven para la recolección de resultados parciales, por ejemplo, cuando se desea obtener un resultado final. Es importante señalar que DLML trabaja con aplicaciones que manejan datos de procesamiento independiente, es decir que pueden ser procesados por cualquier procesador.

Como **segunda etapa** aplicamos el mismo procedimiento de verificación realizado con la estructura básica propuesta para verificar la versión original de DLML. En esta etapa únicamente se

considera la verificación de la operación de manipulación que activa al algoritmo de distribución de datos de DLML. Bajo la misma perspectiva se verifica una nueva propuesta de implementación de DLML para ser ejecutada sobre un cluster de nodos multicore.

1.3. Objetivos

Los objetivos de esta tesis son:

- Proponer una estructura básica de procesamiento de datos independientes que integre técnicas de distribución dinámica de carga mediante el uso de una lista.
- Definir un modelo formal de las operaciones básicas de distribución de datos (de manipulación) en la estructura propuesta.
- Identificar las propiedades requeridas en el funcionamiento de la estructura básica y verificar que se cumplen aplicando el método exhaustivo de *model-checking*.
- Proponer una nueva versión de implementación de la herramienta DLML combinando los modelos de memoria compartida y envío / recepción de mensajes para cluster multicore.
- Aplicar el método *model-checking* para verificar la nueva implementación de DLML.

1.4. Metodología

La metodología para la realización de la tesis se enfocó en los siguientes pasos:

1. Estudio de los algoritmos de distribución de carga y estudio de la herramienta de distribución de carga DLML y su arquitectura.
2. Estudio de métodos formales para la verificación de algoritmos paralelos.
3. Propuesta de un algoritmo con estructura básica de interacción de distribuidor y procesamiento independiente de datos basado en el uso de una lista.
4. Identificación de la propiedades que debe cumplir el algoritmo propuesto y su verificación.
5. Modelado y verificación de la versión original de DLML.
6. Modelado y verificación de una nueva implementación del algoritmo de distribución para DLML sobre arquitecturas de cluster con nodos multicore
7. Desarrollo y evaluación de la implementación propuesta.

1.5. Organización de la tesis

En el capítulo 1, se presenta un perspectiva general del problema y los objetivos que se plantea en esta tesis. En el capítulo 2, se estudian los fundamentos teóricos de los algoritmos de distribución de carga. Una vez descritos se muestra la arquitectura de DLML y el algoritmo de distribución que implementa esta librería. En el capítulo 3, se presenta la teoría relacionada con la verificación formal de sistemas.

En el capítulo 4 proponemos un modelo de la estructura básica de una aplicación SPMD de procesamiento de datos la cual integra una política de distribución cíclica. Junto con este modelo proponemos y verificamos un conjunto de propiedades para garantizar su buen funcionamiento.

En el capítulo 5, se presentan el modelado y la verificación de DLML. Se presentan las propiedades que se verificaron sobre este distribuidor junto con los resultados obtenidos. Además, en este capítulo proponemos diferentes implementaciones de DLML mostrando las diferencias entre cada implementación. En el capítulo 6, realizamos una comparación de rendimiento entre la versión original de DLML y la mejor implementación propuesta de DLML.

El capítulo 7 concluye presentando los principales resultados de este trabajo, y planteando posibles líneas de continuación del mismo. En el apéndice A, se realiza una descripción de diferentes escenarios de ejecución para las nuevas propuestas de DLML mostradas en el capítulo 5.

Capítulo 2

La distribución de carga y la herramienta DLML

"Divide las dificultades que examinas en tantas partes como sea posible para su mejor solución."

–René Descartes 1596-1650 Filósofo y matemático francés.

En este capítulo presentamos un panorama de cómo se ha estudiado el problema de la distribución de carga entre un conjunto de procesadores. Mostramos las principales políticas de los algoritmos. También presentamos las características de la librería DLML[CCGRA05] para proveer el servicio de distribución de datos (carga) de manera transparente para el programador. Finalmente se presenta la arquitectura y el protocolo de distribución de DLML.

2.1. Algoritmos de distribución de carga

2.1.1. Introducción

La distribución de carga busca mejorar el desempeño de un sistema distribuido, usualmente en tiempos de respuesta o disponibilidad de recursos, asignando el trabajo entre un conjunto de procesadores cooperantes.

La distribución de carga puede tomar lugar estáticamente o dinámicamente:

1. **Distribución estática**

   La distribución estática asigna trabajo a los procesadores antes de que el programa comience a ejecutarse sin considerar los eventos que ocurren en tiempo de ejecución. La ventaja de los algoritmos de distribución estática de carga se debe a que si se cuenta con la información necesaria acerca del comportamiento de la aplicación y del sistema, es posible, determinar en tiempo de compilación una asignación óptima.

2. **Distribución dinámica**

   Por lo general, no es posible tener una estimación sobre el comportamiento de la aplicación (cantidad de cálculos, de datos generados, de comunicaciones, etc.) ni del sistema (cuando

permite multiprogramación). La distribución dinámica de carga, se define como la reasignación de carga entre los procesadores a tiempo de ejecución en función de la capacidad de los procesadores. Entre las ventajas de la distribución dinámica de carga está que no se necesita saber con anterioridad el tiempo de ejecución de las tareas de una aplicación, la carga de los procesadores se ajusta a los cambios imprevistos generado por el uso de recursos de parte de la aplicación. Entre las desventajas está el costo que involucra esta redistribución o ajuste.

En las siguientes secciones, nos enfocamos a describir las características de los algoritmos de distribución de carga, dando algunos ejemplos.

2.1.2. Arquitectura de los algoritmos de distribución dinámica de carga

La implementación de un algoritmo de distribución dinámica de carga está conformada básicamente por dos elementos[SKS92]:

1. Una política de información.
2. Una política de control.

Política de información

La política de información se encarga de cuantificar la carga del sistema y/o de los procesadores. La carga o estado de carga puede cuantificarse de diversas maneras, por ejemplo, promediando el número de procesos que hay en el/los nodo(s), considerando el porcentaje de CPU utilizado, contando el número de datos que tiene una aplicación para procesar, etc.

La recolección de la información sobre el estado de carga puede efectuarse de manera global o parcial. Es global cuando cada nodo posee una visión sobre el estado de carga de todos los demás nodos del sistema. Esta recolección puede ser demasiado cara debido a que se puede requerir un gran número de comunicaciones, limitando la escalabilidad. Es parcial cuando cada nodo recolecta información sobre el estado de carga de un subconjunto de nodos y, con base a esta información se toman decisiones. Este esquema es usado por diferentes algoritmos de distribución [LM92] [BS85].

Política de control

La política de control se encarga de decidir en qué momento se necesita hacer una transferencia de carga; hacia cuál nodo se hará la transferencia y cuales elementos de carga se transferirán. La política de control puede ser de tres tipos:

1. Control centralizado.
2. Control distribuido.
3. Control híbrido.

En los algoritmos de control centralizado la decisión cae sobre un nodo en particular, ésto puede llevar a problemas como cuellos de botella, no terminación del algoritmo debido a la falla del nodo que recolecta la información, etc. Estos algoritmos generalmente son muy simples de implementar.

Los algoritmos de control distribuido son más efectivos debido a que permiten que más de un procesador pueda tomar decisiones respecto a la transferencia de carga, pudiendo contar con alguna información sobre el estado de carga del sistema (global o parcial).

La política de control híbrida se da mediante la combinación (interacción) de una política de control centralizada y otra distribuida. Ya sea que utilicemos una política de control centralizada, distribuida o híbrida, está dividida a su vez en tres políticas[SKS92]:

1. Política de transferencia.
2. Política de localización.
3. Política de selección.

La política de transferencia nos indica cuándo debe hacerse la transferencia de carga de un nodo hacia otro. Para tomar esta decisión nos podemos basar en un mecanismo de doble frontera[NXG85], esto es, se definen tres estados
posibles en cada nodo:

1. DESCARGADO.
2. NORMAL.
3. SOBRECARGADO.

Las fronteras son dadas por un valor de referencia que se puede estar variando. Como se muestra en la Figura 2.1, la primer frontera se encuentra entre el estado DESCARGADO y el estado NORMAL, la segunda frontera está entre el estado NORMAL y el estado SOBRECARGADO.

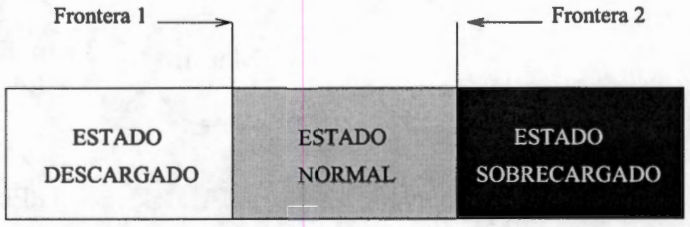


Figura 2.1: Esquema de doble frontera.

Cuando el estado de un nodo es DESCARGADO, entonces este nodo tiene la posibilidad de ejecutar la carga local que se genere y además puede aceptar la carga proveniente de otros nodos. Por otra parte, si el estado de un nodo es NORMAL, solo ejecuta la carga generada localmente y rechaza la carga remota. En el último caso selecciona un nodo descargado para trasferirle parte de su carga.

La política de localización determina a dónde (a qué nodo del sistema) se puede transferir carga, ésta puede ser:

1. No condicionada.
2. Condicionada.

Si la política es no condicionada, no se usa información sobre el estado de carga del sistema para seleccionar un nodo destino. Esta política es mejor conocida como política de localización ciega. Si la política es condicionada se usa la información recolectada por el elemento de información para seleccionar el nodo destino.

La política de selección se encarga básicamente de seleccionar cuáles y cuántas unidades de carga podrán ser transferidas.

2.2. Ejemplos de algoritmos de distribución de carga

Como vimos en la sección anterior, los algoritmos de distribución de carga pueden clasificarse de diferentes maneras. En esta sección mostraremos dos ejemplos de algoritmos de distribución de carga usados frecuentemente.

2.2.1. Algoritmo de control centralizado Maestro-Esclavos

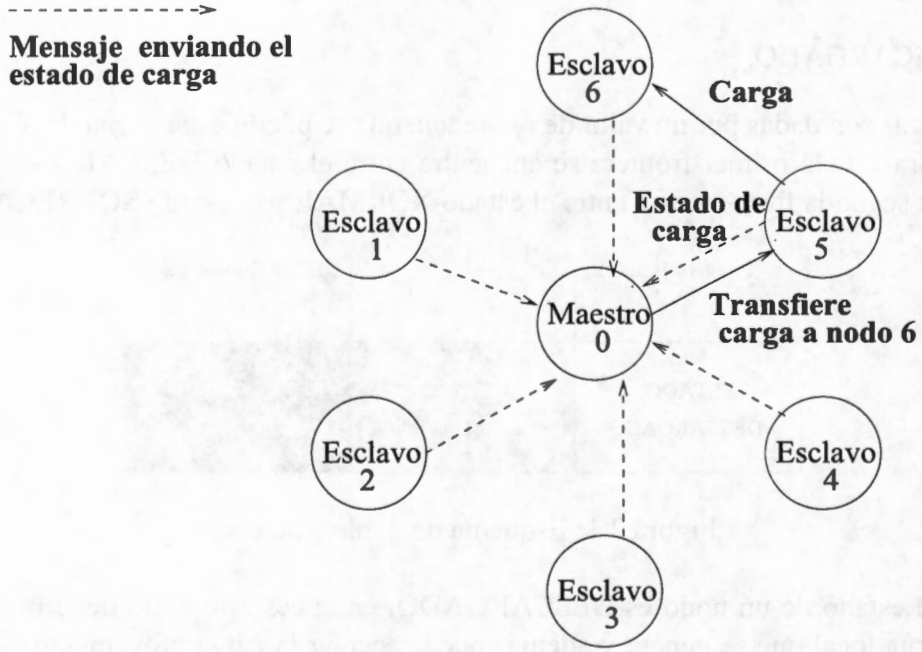


Figura 2.2: Modelo de asignación centralizado.

En este algoritmo existe un nodo maestro y un conjunto de nodos esclavos. Los nodos esclavos constantemente envían la información de su carga al nodo maestro. De esta manera el maestro tiene una información global del estado de carga del sistema. Cuando el nodo maestro detecta que hay un nodo sobrecargado X en el sistema y existe otro nodo descargado Y, indica a X con un mensaje que haga transferencia de carga hacia Y. En la Figura 2.2 se muestra un ejemplo donde el nodo 0 es el nodo maestro y los nodos 1 a 6 los esclavos. En esta figura X es el esclavo 5 y Y el 6.

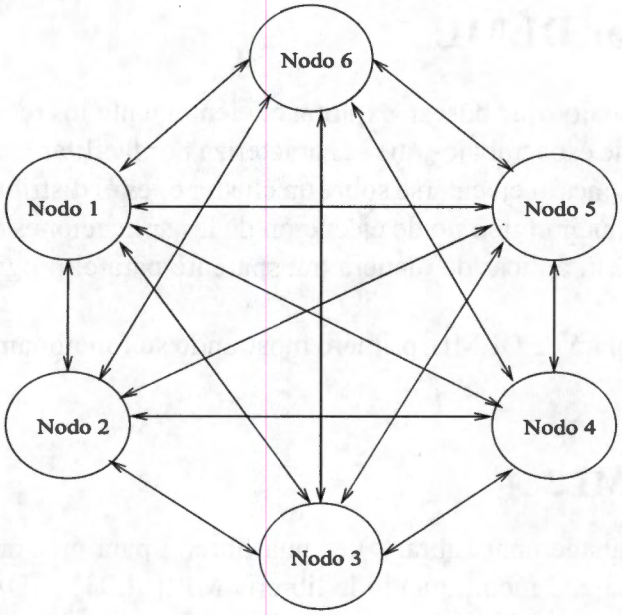


Figura 2.3: Algoritmo de control distribuido con información global.

2.2.2. Algoritmo de control distribuido con información global

Para este algoritmo de distribución no existen jerarquías entre los nodos, todos realizan las mismas actividades y cada uno conoce la información del estado carga de los nodos. Cuando un nodo se da cuenta que su carga ha cambiado le avisa a los demás por medio de una difusión. Cuando un nodo recibe una actualización de carga, éste modifica su lista de estados de carga.

Un problema que se puede presentar utilizando este algoritmo es la saturación de los canales de comunicación.

En la Figura 2.3 se presenta un esquema en donde se muestra las interacciones de cada nodo conectado con el resto formando una topología similar a un *grafo completo* [1]. Como se puede observar si el número de nodos crece, los enlaces aumentan drásticamente.

En este ejemplo el control es distribuido ya que cada nodo puede decidir sobre la transferencia de su propia carga hacia otro nodo descargado, basándose en su información global colectada.

Otra variante de este algoritmo es el algoritmo de subasta[SS84]. En este algoritmo, la información sobre el estado de carga se colecta cada vez que un nodo esta descargado. En este caso el nodo que ya terminó su trabajo busca más elementos a procesar con el nodo que esté más cargado, disminuyendo así, el efecto del desbalance provocado por la multiprogramación, la heterogeneidad y la generación dinámica de carga. El funcionamiento de este algoritmo será explicado en detalle en la siguiente sección.

Otros algoritmos no realizan una colecta de información lo cual ayuda a reducir el número de mensajes. Uno de estos algoritmos es el algoritmo de distribución cíclico el cual se revisa a detalle en el siguiente capítulo 4.

[1] En un grafo completo, existe una arista bidireccional entre cualesquiera dos nodos distintos

2.3. Distribuidor DLML

Ha habido muchos trabajos que buscan explotar eficientemente los recursos de cómputo[Col91] [GDN+98] [ST98]. Uno de esos trabajos que se caracteriza por facilitar la programación paralela de una diversidad de aplicaciones al ejecutarse sobre un cluster es es el distribuidor DLML[CCGRA05] [Gar07]. DLML busca reducir el tiempo de ejecución de las aplicaciones que utilizan datos de procesamiento independiente al aplicar de manera transparente para el programador una distribución dinámica de datos.

En esta sección se hablará de DLML, primero mostrando su funcionamiento y uso para finalizar con su arquitectura.

2.3.1. ¿Qué es DLML?

DLML (Data List Management Library) es una librería para programación paralela en clusters, está escrita en lenguaje C incorporando la librería MPI[SL03] [BDV94]. DLML es útil para desarrollar aplicaciones que pueden organizar sus datos en una lista, y el procesamiento de cada elemento de la lista no dependa de otros elementos de la lista. La programación es casi secuencial, y en tiempo de ejecución la distribución de carga toma lugar de manera transparente para el programador. Con esta librería, los datos de la lista son accesados usando funciones típicas de listas, tal como ***Get*** o ***Insert***, proporcionadas por este ambiente. Aunque el uso de estas funciones es el típico, internamente ellas operan sobre varias listas distribuidas, una sobre cada nodo. Cuando un procesador encuentra su lista vacía, aplica un algoritmo de subasta para identificar al nodo con mayor cantidad de datos que pueda trasferirle carga. El conjunto de primitivas de DLML (interfaz) se separan en 3 tipos [Gar07]: señalización, recopilación y manipulación las cuales mencionamos a continuación.

2.3.2. Interfaz de DLML

Primitivas de señalización

En las operaciones de señalización se incluyen primitivas de inicialización y que conceden el control a un sólo proceso. Estas son:

DLML_Init - Inicializa la lista

Sintaxis - *void DLML_Init(Lista *)*

Descripción - Esta función se encarga de inicializar una lista y hacerla vacía (longitud=0). Después de esto la lista puede ser manipulada.

DLML_Finalize - Finaliza DLML

Sintaxis - *void DLML_Finalize(void)*

Descripción - Asegura la terminación del procesamiento de la lista y devuelve el control la sistema operativo.

DLML_Only_One - Sólo un proceso

Sintaxis - *void DLML_Only_One { }*

Descripción - Esta macro señala que un solo procesador ejecutará las instrucciones contenidas en ésta. La macro se delimita por llaves.

Primitivas de recopilación

En este tipo se concentran las funciones que recuperan resultados parciales o previos. Recordamos que DLML trabaja sobre listas distribuidas en todos los procesadores, por lo que en muchas ocasiones los resultados parciales se encuentran esparcidos. Las funciones que ayudan a esta recopilación son:

DLML_Exchange - Intercambia resultados

Sintaxis - *void DLML_Exchange (void *fbuffer, int cantidad, void *dbuffer)*

Parámetros de entrada

fbuffer - dirección de inicio de la variable fuente

cantidad - número de datos a ser transferidos

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza un intercambio de datos, entre todos los procesadores. Para ello la función se auxilia de un arreglo donde se guardan todos los datos enviados. Para tener un orden, cada posición del arreglo contiene el dato enviado por dicho procesador. Al final de la función `DLML_Exchange` los procesadores tienen el mismo arreglo.

DLML_Reduce_Add - Reducción con suma

Sintaxis - *void DLML_Reduce_Add (void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envío de todos los datos parciales a un solo

procesador (procesador cero). Adicionalmente al envío, los resultados parciales son sumados y el resultado es dejado en la variable dbuffer del procesador cero.

DLML_Reduce_Avarage - Reducción con promedio

Sintaxis - *void DLML_Reduce_Average (void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Adicionalmente al envío, los resultados parciales son promediados.

DLML_Reduce_Max - Reducción con máximo

Sintaxis - *void DLML_Reduce_Max (void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envió de todos los datos parciales a un solo procesador (procesador cero). Adicionalmente, la función devuelve el valor máximo de todos los envíos.

Primitivas de manipulación

Las primitivas de manipulación se encargan del manejo de los datos de la lista. Estas primitivas son:

DLML_Lenght - Longitud de la lista

Sintaxis - *void DLML_Lenght (void)*

Descripción - Devuelve un entero, el cual indica la longitud actual de la lista local.

DLML_Get - Obtiene elemento

Sintaxis - *char DLML_Get (Lista *, node *)*

Descripción - El elemento es obtenido de la lista apuntada por *Lista* y es dejado en el apuntador de tipo *node*. Éste elemento es obtenido de la lista local, decrementando su tamaño en uno. DLML se encarga de insertar elementos en la lista local cuando queda vacía, suprimiéndolos de otras listas locales de forma transparente para el programador.

DLML_Query - Consulta elemento

Sintaxis - *void DLML_Query (Lista *, node *)*

Descripción - Es similar a `DLML_Get`. La única diferencia es que el elemento no es eliminado de la lista, sólo se obtiene una copia. **DLML_Insert** - Inserta elemento

DLML_Insert - Inserta elemento

Sintaxis - *void DLML_Insert (Lista *, node)*

Descripción - Inserta localmente un elemento del tipo *node* en la lista apuntada por *Lista*. Esta inserción se da normalmente al inicio del programa, pero puede haber aplicaciones donde la inserción se haga a tiempo de ejecución.

2.3.3. Uso de DLML

DLML puede trabajar con aplicaciones estáticas o dinámicas [Gar07], en las aplicaciones estáticas se conoce de antemano la cantidad de datos a procesar a diferencia de las dinámicas.

Mediante el ejemplo del problema de las N-Reinas [BD75] ilustramos cómo usar DLML en una aplicación. Este ejemplo se muestra en la Figura 2.4.

En la Figura 2.4 se observa que la estructura de programar con DLML es muy parecida a si estuviéramos programando con un simple TDA lista.

DLML sigue el modelo de ejecución SPMD, es decir, el código de la Figura 2.4 lo ejecutan todos los procesos instanciados (uno por procesador). En las líneas 3-6 un solo proceso crea la primera posible solución la cual es insertada como primer elemento en la lista mediante la función `DLML_Insert()`. Esta posible solución (y todas las demás) se componen de un tablero (arreglo) y una variable que indica la siguiente reina a colocar. El tablero se modela con un arreglo que guarda en cada posición (renglón) un número de columna. Esta combinación de columnas es lo que forma las posibles soluciones. En la línea 7, mediante la función `DLML_Get()` los procesos intentan obtener elementos (posibles soluciones) a explorar. Este elemento puede ser obtenido de la lista local o de una lista remota. La primera vez sólo un proceso obtiene el único elemento que existe. Los demás procesos al no poder obtener elementos de la lista, permanecen bloqueados. El proceso que obtuvo el elemento, hace el procesamiento y verifica si puede generar otras posibles soluciones parciales, si es así, el proceso las crea (línea 12) y las inserta en la lista (línea 13) mediante

```
1   function Non-attacking-queens(List *L){
2    queen = 1; chessboard[1 <= i <= N] = 0;nr_solutions = 0;
3   DLML_Only_One {
4     q = Create_Element(queen,chessboard);// la función crea una primer
solución
5     DLML_Insert(&L, q);//Se inserta un elemento en la lista
6   }
7   while( DLML_Get(&L, &B) ) {
8     for(column=1;column <= N; column++ ) {// se generan posibles soluciones
9       if (! Attacked(B.queen,column,B.chessboard) )//si la reina no es
atacada en la columna indicada
10        if (B.queen < N) { // si no es la última reina a ser colocada
11           B.chessboard[B.queen] = column;// se coloca en el tablero
12          q = Create_Element(B.queen+1,B.chessboard);// se crea un elemento
de la lista
13           DLML_Insert(&L,q); // y se inserta
14         }else
15           nr_solutions= nr_solutions +1;// se tiene otra solución completa
16     }// N-queens have been placed // el contador
17   }
18   final_result= DLML_Reduce_Add(nr_solutions );
19   return final_result;
20   }
```

Figura 2.4: Algoritmo para resolver el problema de las N Reinas con DLML.

`DLML_Insert()`. A partir de ahí los demás procesos pueden obtener elementos de la lista (posibles soluciones) e ir insertando o eliminando soluciones. Al final cada uno de los procesos tienen parte de las soluciones encontradas, por lo que para encontrar el total, éstas se suman globalmente (línea 18).

2.3.4. Arquitectura e implementación de DLML

La arquitectura de DLML consta de dos procesos por procesador: *Aplicacion* y *DLML* (Figura 2.5). Además en base al tipo de proceso se crean dos grupos. El primer grupo solamente incorpora procesos *Aplicacion* y el segundo grupo sólo procesos *DLML*. Los procesos del mismo grupo se pueden comunicar por medio de mensajes. Estos grupos sirven para permitir la ejecución de diferentes tipos de funciones (de las vistas en la sección anterior). Si un proceso pertenece al grupo de los procesos *Aplicacion* puede ejecutar las funciones de señalización, recopilación y manipulación. Si el proceso no pertenece a este grupo (es un proceso *DLML*) solamente participa en la implementación las funciones de *manipulación*. A continuación se presenta como DLML implementa cada una de las funciones vistas.

Implementación de las funciones de señalización

La implementación de `DLML_Init` consiste en la inicialización de la lista y en la creación de un identificador `id` para identificar si un proceso pertenece al grupo de procesos *DLML* ($N/2 \leq$ `id` $< N$) o al grupo de procesos *Aplicacion* ($0 \leq$ `id` $< N/2$) donde N es el número total de procesos (entre proceso *DLML* y procesos *Aplicacion*).

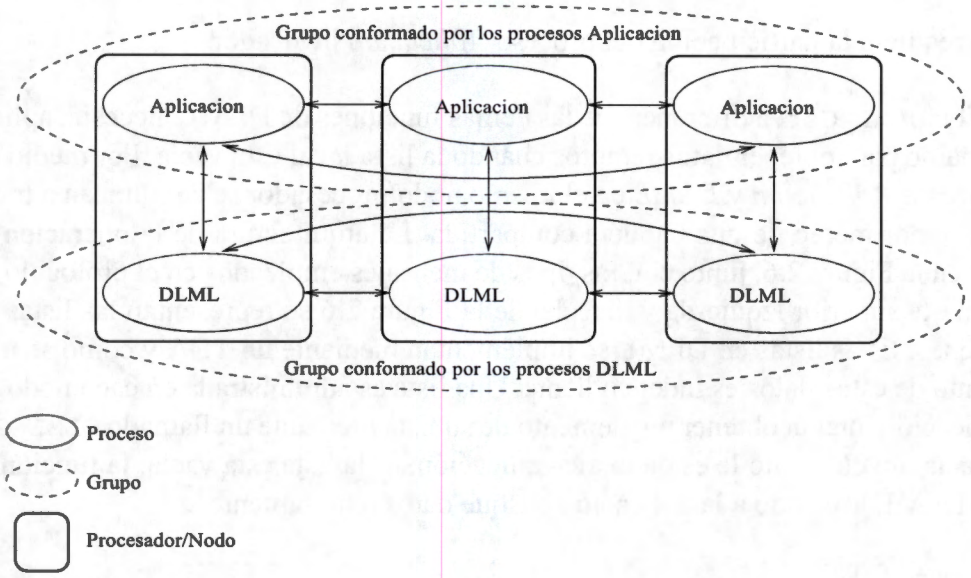


Figura 2.5: Arquitectura del distribuidor DLML.

La implementación de `DLML_finalize` consiste en liberar los recursos utilizados por la lista local. Debido a lo anterior después de ejecutar un `DLML_Finalize` ya no es posible ejecutar alguna instrucción de DLML.

El macro `DLML_Only_One` se implementa mediante una estructura condicional `if(id==0){ }`, donde id es una variable de tipo entero donde se almacena el identificador del proceso.

Implementación de las funciones de recolección

La implementación de la funciones de recolección de DLML se realiza utilizando dos funciones de difusión de MPI: `MPI_All_gather` y `MPI_Reduce`.

La función `MPI_All_gather` recolecta un conjunto de datos de un grupo de procesos y los distribuye a todos los procesos del grupo. Esta función se utiliza en la implementación de `DLML_Exchange` la cual presenta su misma semántica.

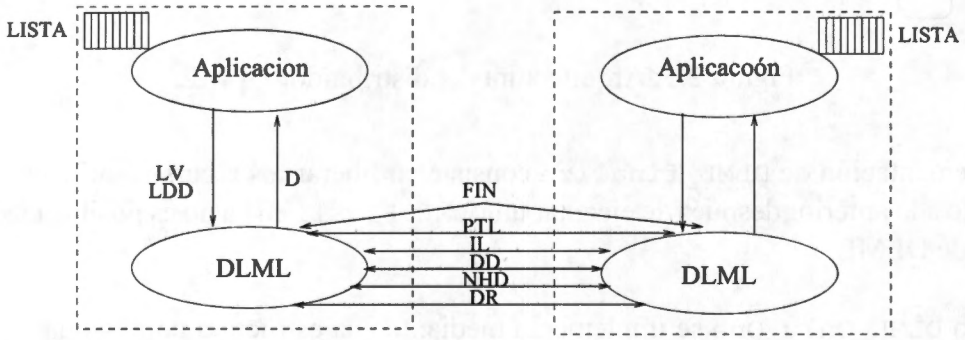
La función `MPI_Reduce` reduce un conjunto de datos de un grupo de procesos en un sólo dato, aplicando una operación de suma, promedio, máximo o mínimo. Este nuevo dato generado se almacena en un `buffer` de un proceso del grupo identificado por *root* (en nuestro caso root tiene id=0). Esta función se utiliza en las implementaciones de las funciones `DLML_Reduce_Add`, `DLML_Reduce_Average` y `DLML_Reduce_Max`.

Implementación de las funciones de manipulación

Dado que DLML internamente trabaja sobre listas locales la mayoría de las funciones de manipulación se implementan de manera similar que en el TDA lista en un programa secuencial. Tal es el caso de `DLML_Lenght`, `DLML_Insert` y `DLML_Query`. La implementación de estas operaciones

únicamente requiere la participación del proceso *Aplicacion* invocador.

La función `DLML_Get`, a diferencia de las demás funciones de DLML, necesita aplicar un algoritmo distribuido para obtener datos remotos cuando la lista local está vacía. Por medio de esta función los procesos *Aplicacion* y *DLML* en el mismo nodo/procesador se comunican a través de paso de mensajes y por medio de una bandera compartida. La arquitectura de la operación `DLML_Get` puede verse en la Figura 2.6, junto con los tipos de mensajes empleados en el protocolo de distribución. En la parte superior izquierda y derecha de la Figura 2.6 se representan las listas de datos de las aplicaciones. Estas listas en DLML se implementan mediante un TDA y como se mencionó, el procesamiento de estos datos es independiente. Una lista es administrada en cada nodo. Cuando un proceso aplicación intenta obtener un elemento de su lista mediante un llamado `DLML_Get` y su lista no esta vacía, un elemento le es dado a la aplicación. Si la lista está vacía, la función `DLML_Get` hace que el DLML asociado a la aplicación busque datos remotamente.



PTL = PETICION_TAM_LISTA
IL = INFORMACION_LISTA
DD = DAME_DATOS
NHD= NO_HAY_DATOS
DR = DATOS_REMOTOS
D = DATOS
LV = LISTA_VACIA
LDD= LISTA_DE_DATOS
FIN= FIN

Figura 2.6: Tipos de mensajes en el protocolo de subasta durante la primitiva `DLML_Get`.

Para entender el funcionamiento del protocolo utilizado por DLML en la función `DLML_Get` revisaremos los siguientes posibles casos que se pueden presentar durante una ejecución donde se use DLML.

1. **Primer caso**: Un proceso aplicación termina de procesar sus datos, por tanto, al tratar de obtener otro su proceso DLML se activa para buscar datos. En este caso hay dos posibilidades, cuando DLML puede obtener datos remotos y cuando ya no existen datos en el sistema.

   Durante la explicación de la primer posibilidad iremos utilizando el ejemplo de la Figura 2.7 donde intervienen 4 nodos. El proceso aplicación *Aplic_x* y su proceso asociado *DLML_x* se sitúan en el nodo X, los procesos *DLML_y*, *DLML_z* y *DLML_w* están situados en nodos diferentes.

   Cuando la aplicación *Aplic_x* termina de procesar sus datos, envía un mensaje de tipo *LV (LISTA_VACIA)* a su proceso asociado *DLML_x*.

El proceso *DLML_x*, al recibir un mensaje *LV*, inicia la subasta de su poder computacional y envía un mensaje *PTL (PETICION_TAM_LISTA)* a los procesos *DLML_y*, *DLML_z* y *DLML_w*.

Cuando *DLML_y*, *DLML_z* y *DLML_w* reciben el mensaje *PTL*, responden con un mensaje *IL (INFORMACION_LISTA)* con el tamaño de su lista local (el proceso *DLML_y* envía 2; el proceso *DLML_z* envía 6 y, el proceso *DLML_w* envía 4 al proceso *DLML_x*).

Cuando el proceso *DLML_x* recibe el mensaje *IL* de todos sus homólogos DLML, selecciona al proceso que le envió el tamaño más grande de lista y le envía un mensaje *DD (DAME_DATOS)* solicitando datos. En este ejemplo el proceso *DLML_x* selecciona al proceso *DLML_z* ya que éste le envió el tamaño más grande de la lista.

Cuando el proceso *DLML_z* recibe el mensaje *DD*, envía a *DLML_x* una parte de sus datos en un mensaje *DR (DATOS_REMOTOS)*. Cuando el proceso *DLML_x* recibe el mensaje *DR*, envía los datos recibidos a su aplicación con un mensaje de tipo *D (DATOS)*.

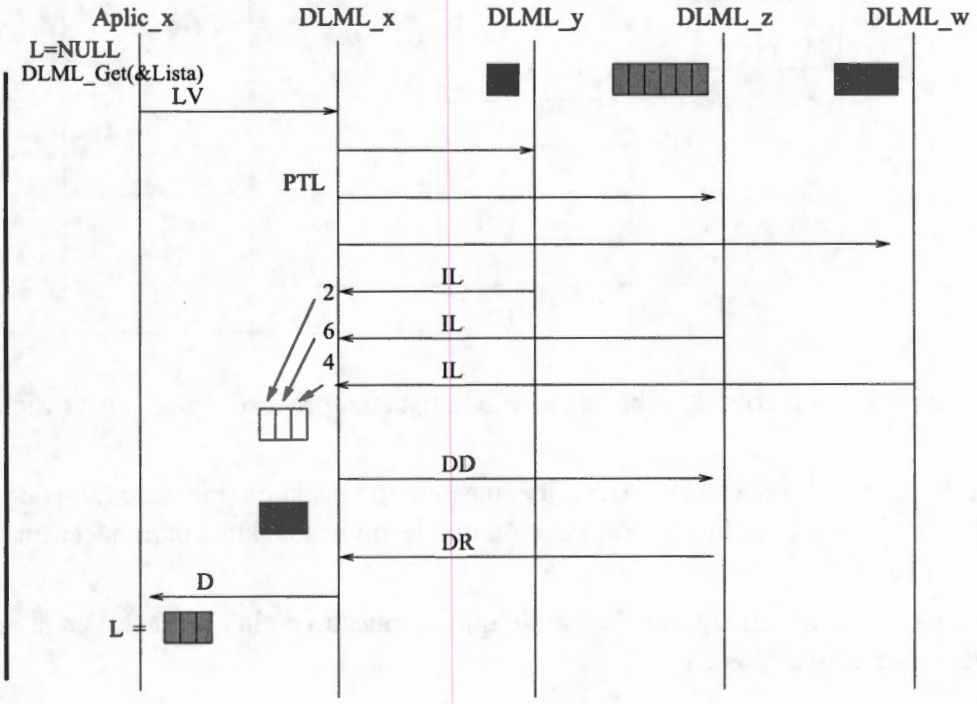


Figura 2.7: Protocolo de subasta cuando la lista del proceso *Aplic_x* esta vacía.

El comportamiento de la segunda posibilidad en donde se han terminado todos los datos del sistema se ilustra con la Figura 2.8. El funcionamiento es el mismo que en la primer posibilidad hasta que *DLML_x* recibe los mensajes *IL* de sus homólogos DLML. En este

punto *DLML_x* descubre que ya no hay datos en el sistema (tamaños de listas igual a 0) por lo que hace que *Aplic_x* termine, devolviendo 0 en la llamada a `DLML_Get` y la lista vacía. Por último *DLML_x* envía a los otros DLML's un mensaje FIN y queda en espera de la recepción de N-1 mensajes FIN para confirmar su terminación.

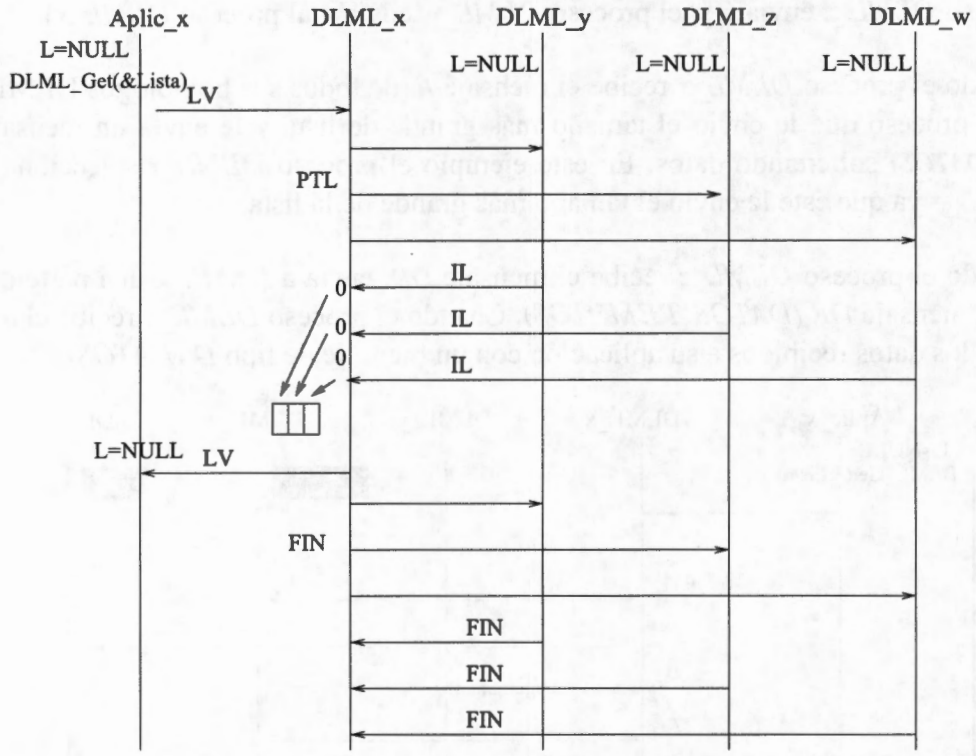


Figura 2.8: Protocolo de subasta cuando la lista del proceso *Aplic_x* esta vacía.

2. **Segundo caso**: Un proceso DLML recibe una petición de datos (mensaje *DD*) de parte de otro DLML, por lo que le indica a su aplicación que le envíe sus datos para hacer una repartición de los mismos.

   En este caso, iremos utilizando el ejemplo que se muestra en la Figura 2.9 en la que se tienen los mismos cuatro nodos en

   el sistema.

   Cuando el proceso *DLML_y* ha seleccionado a *DLML_x* para pedirle datos le envía un mensaje *DD*. Cuando *DLML_x* recibe la petición activa la bandera flag (la inicializa con el valor 1) para indicarle a *Aplic_x* que le envíe la lista local.

   Si la aplicación *Aplica_x* asociada al proceso *DLML_x* se encuentra procesando un dato, entonces *DLML_x* tiene que esperarse a que *Aplica_x* termine de procesar el dato para que ésta le envíe su lista . Si el tiempo de procesamiento es largo, es posible que *DLML_x* acumule varias peticiones de datos, de otros DLML. En el ejemplo de la Figura 2.9 vemos que *DLML_w* también envió otro mensaje *DAME_DATOS* a *DLML_x*.

   Cuando la aplicación *Aplica_x* termina de procesar el dato envía a *DLML_x* un mensaje de tipo *LDD* con su lista local. *DLML_x* revisará cuántas peticiones de datos tiene acumuladas,

con el fin de repartir los datos entre el número de peticiones mas uno (tomando en cuenta a su aplicación). En el caso de la Figura 2.9 *DLML_x* envía datos a *Aplica_x*, *DLML_y* y *DLML_w* en mensajes de tipo *DR*.

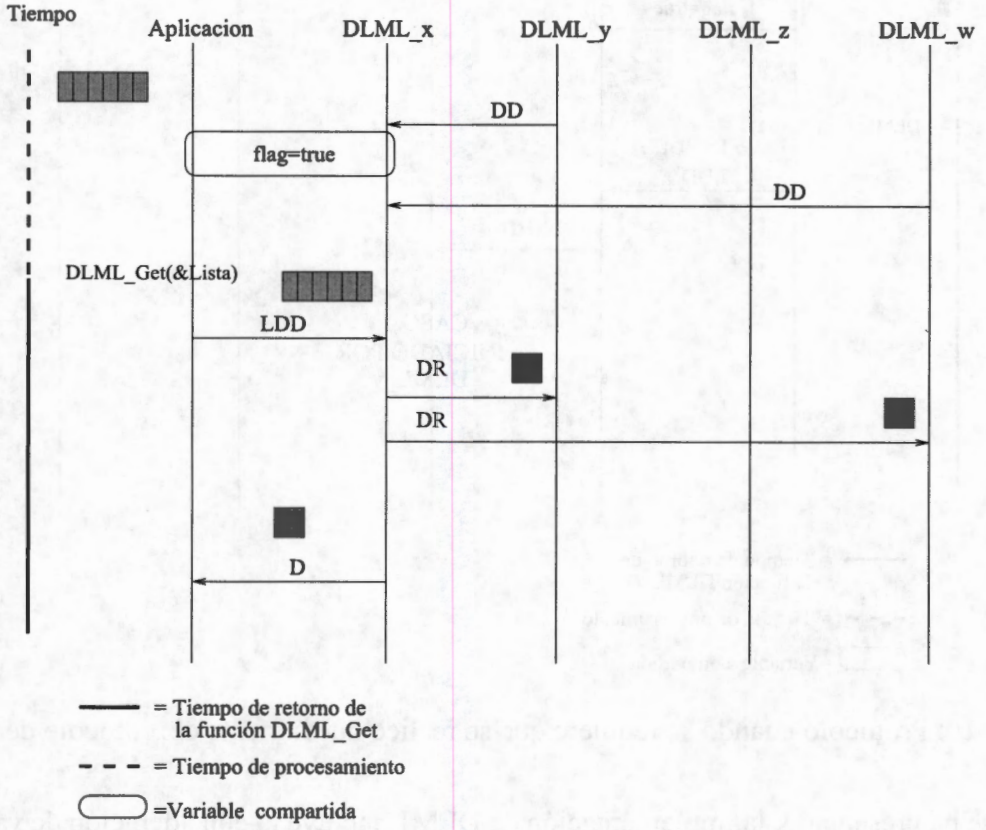


Figura 2.9: Protocolo de subasta cuando se realiza una distribución de datos entre un grupo de procesos.

Si el número de peticiones es más grande que el número de datos en la lista local de *DLML_x* entonces a los procesos DLML que no alcancen datos se les envía un mensaje de tipo *NO_HAY_DATOS*, ejecutándose un comportamiento similar al caso 3.

3. **Tercer caso**: Un proceso DLML recibe una petición de datos pero debido a que su proceso *Aplicacion* asociado terminó sus datos, se le responde con un mensaje al DLML solicitante que ya no se cuenta con datos. Así, éste último realizará el protocolo de subasta nuevamente.

   En ejemplo de la Figura 2.10 *DLML_y* envía un mensaje DAME_DATOS a *DLML_x*. Cuando *DLML_x* recibe el mensaje y su aplicación ha terminado de procesar todos sus datos (L=NULL), entonces Aplic_x envía su lista vacía a *DLML_x*.

   Cuando *DLML_x* recibe la lista vacía, revisa las peticiones de datos acumuladas y a los procesos que le enviaron estas peticiones (en este caso *DLML_y*) les envía un mensaje *NO_HAY_DATOS*. Con este mensaje el proceso *DLML_y* sabe que tiene que realizar nuevamente el protocolo de subasta mostrado en el caso 1.

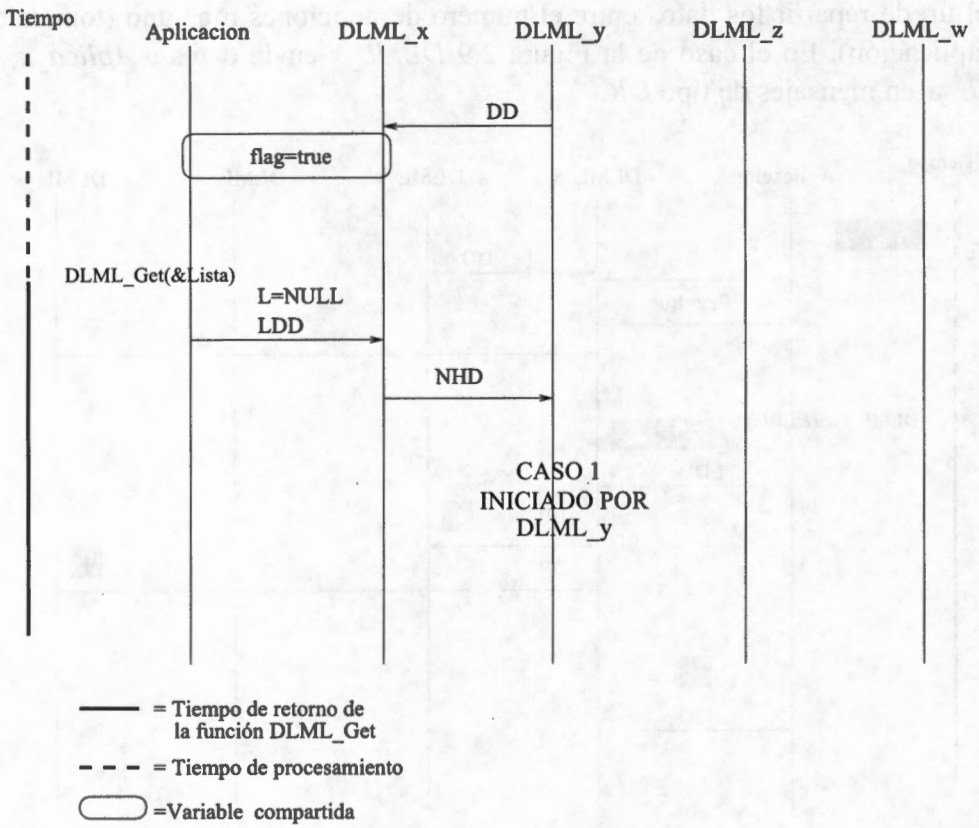


Figura 2.10: Protocolo cuando se requiere que se realice nuevamente el protocolo de subasta.

Como se ha presentado, la implementación de DLML incluye la consideración de varios casos, por lo que sería deseable garantizar su buen funcionamiento. El conjunto de propiedades indispensables que se espera cumpla la herramienta son enumeradas a continuación.

1. Todo proceso sólo recibirá los mensajes dirigidos hacia él.
2. Todo proceso ejecutará la acción correspondiente a cada mensaje.
3. No hay pérdida de mensajes.
4. No hay abrazos mortales en el sistema.
5. El algoritmo terminará cuando no haya datos a procesar en el sistema.
6. Si un proceso inicia el protocolo de terminación, eventualmente cada proceso en el sistema iniciará el protocolo de terminación y recibirá N mensajes FIN.
7. No hay terminación prematura del distribuidor de datos.

Capítulo 3

Model checking y teoría de autómatas

"Lo que sabemos es una gota de agua, lo que ignoramos es el océano"

–Isaac Newton 1643-1727 Físico, filósofo, alquimista y matemático inglés

El *model-checking* es una técnica automática para verificar sistemas concurrentes con estados finitos [PGS01]. Este método ha sido usado exitósamente en la práctica para verificar diseño de circuitos complejos y protocolos de comunicación. En este capítulo explicamos el enfoque del *model-checking* que se basa en la teoría de autómatas y la lógica temporal. Primero comenzamos con una breve introducción de las tareas que se tienen que hacer para aplicar esta técnica. Después estudiamos *promela*, el cual es lenguaje que utiliza *Spin* para modelar sistemas/programas. Terminamos explicando como funciona el *model-checking*.

3.1. Introducción

Aplicar el *model-checking* a un programa consiste en las siguientes tareas [CGP99].

Modelado:

La primer tarea es convertir nuestro programa (sistema) a un formalismo aceptado por una herramienta para *model-checking*también llamada *model-checker*. En muchos casos, esto es simplemente una tarea de compilación. En otros casos, debido a limitaciones sobre tiempo y memoria, el programa puede requerir el uso de abstracciones para eliminar detalles irrelevantes o sin importancia.

Especificación:

Antes de la verificación es necesario exponer las propiedades de comportamiento que el programa debe satisfacer. La especificación es dada normalmente en algún formalismo de lógica. Para sistemas de Software y Hardware es común usar lógica temporal, la cual puede describir cómo evoluciona el comportamiento del sistema en el tiempo.

Verificación:

La idea subyacente de la técnica del *model-checking* es construir un autómata A_p del programa y un autómata A_{prop} de la propiedad a verificar. Dado que un autómata describe un lenguaje, entonces debemos probar que $L(A_p) \subseteq L(A_{prop})$, esto es, que el lenguaje del autómata del programa es un subconjunto del lenguaje del autómata de la propiedad a verificar. Dicho de otro forma, tenemos que probar que $L(A_p) \cap \overline{L(A_{prop})}=\phi$, es decir, no hay un comportamiento en el programa que no satisfaga la propiedad, pero la solución de este problema es difícil de obtener por lo que tenemos que usar una variación a la teoría de autómatas conocida como ω-autómatas. La principal diferencia con la teoría de autómatas estándar es que las condiciones de aceptación para los ω-autómatas cubren ejecuciones finitas e infinitas.

3.2. Promela

Como mencionamos en este proyecto se utilizó el *model-checker Spin*. *Spin* acepta como entrada un sistema escrito en un lenguaje de modelado llamado *promela* (*Process Meta Language*). Este lenguaje permite la creación dinámica de procesos y la comunicación entre ellos mediante canales síncronos o asíncronos ó bien vía variables compartidas. En las siguientes sub-secciones daremos un descripción del lenguaje *promela*.

3.2.1. Tipos de objetos

Un programa escrito en *promela* está constituido básicamente por tres tipos de objetos:

- Procesos.
- datos.
- Canales de mensajes.

Un proceso en ejecución es una instancia de un proceso previamente declarado mediante la palabra reservada `proctypes` (tipo de proceso) y sirve como base para la construcción del modelo de un sistema. El modelo de un sistema/programa debe contener por lo menos un proceso. El cuerpo de un `proctype` consiste en cero o varias declaraciones de datos y uno o varios segmentos.

Los objetos de tipo proceso son siempre declarados como globales, los objetos de datos o los canales pueden ser declarados como globales o locales. Son globales cuando se declaran en la parte superior de nuestro programa y locales cuando son declarados dentro del cuerpo de un `proctype`. En las siguientes subsecciones entramos más a detalle sobre los tres tipos de objetos mencionados dando algunos ejemplos de estos.

Procesos

Como mencionamos anteriormente, los procesos son declarados por medio de `proctype`. Para instanciar un conjunto de procesos hay varias maneras. Podemos crear multiples instacias de un `proctype` utilizando la palabra reservada `active [n]`, con ellos especificamos la creación de

```
1: active [2]proctype procesosEjemplo
2: {
3:    printf("Mi ID es:%d ", _pid)
4: }
```

Figura 3.1: Ejemplo donde se utiliza `active` para instanciar 2 procesos `procesosEjemplo`.

```
proctype procesosEjemplo(bytex) {
  printf("x =%d, ID =%d", x,_pid)
}
init
{
  run procesosEjemplo(0);
  run procesosEjemplo(1);
}
```

Figura 3.2: Utilizando el proceso `init` y `run` para instanciar procesos.

n procesos. En el ejemplo de la Figura 3.1 se instancian dos procesos *procesosEjemplo* usando el método anterior.

Cada vez que se instancia un proceso se le asigna un identificador o numero de identificación de proceso. El identificador es único y lo podemos obtener mediante la variable local predefinida `_pid`.

Otra manera de crear instancias de procesos es utilizando el operador `run`. Cualquier proceso en ejecución puede crear más procesos utilizando esté operador. La desventaja de esta manera de crear procesos es que debemos generar un proceso de más (el proceso inicial que genera todos los demás). Un proceso más no dificulta la simulación de nuestro modelo, pero a la hora de la verificación la cantidad de estados que éste puede originar puede ser muy grande. Para ilustrar lo anterior reescribimos nuestro ejemplo anterior como se muestra en la Figura 3.2 :

En *Spin*, cuando un proceso ejecuta su última línea de código se dice que termina su ejecución pero no muere. Un proceso muere después de que todos los procesos que fueron creados después de él hayan muerto.

Cláusula restrictiva

La ejecución de los procesos puede ser regulada por reglas (condiciones) de sincronización dadas dentro de los segmentos de `proctype` pero es posible sincronizar a los procesos para que se estén ejecutando de manera sincronizada, esto por medio de la palabra reservada `provided`.

```
bool toggle = true; /* Variables globales */
short cnt;
active proctype procesoA() provided (toggle ==true)
{
  L: cnt++; /* significa: cnt = cnt+1 */
  printf("procesoA : cnt=%d ",cnt);
  toggle = false; /* Cambio de control a procesoB */
  goto L /* Repetir nuevamente */
}
active proctype procesoB() provided (toggle ==
false)
{
  L: cnt-; /* significa: cnt = cnt-1 */
  printf("procesoB : cnt=%d", cnt);
  toggle = true; /* Cambio de control a procesoA */
  goto L
}
```

Figura 3.3: Ejemplo donde dos procesos (procesoA y procesoB) se sincronizan usando `provided`.

```
procesoA: cnt=1
  procesoB: cnt=0
procesoA: cnt=1
  procesoB: cnt=0
procesoA: cnt=1
  procesoB: cnt=0
procesoA: cnt=1
```

Figura 3.4: Resultado al ejecutar el código de la Figura 3.3.

En la Figura 3.3 los procesos se ejecutan uno después del otro debido a que fueron sincronizados por `provided`, su ejecución se muestra en la Figura 3.4:

Objetos de datos

Las variables en *promela* pueden ser locales o globales. Las variables globales pueden ser vistas por todos los procesos, mientras que las locales sólo dentro del proceso donde se declara. *Promela* tiene diferentes tipos de datos, ellos se muestran en el cuadro 3.1. Además se pueden definir arreglos de los diferentes tipos.

Tipo	Rango
bit	0, 1
bool	false, true
byte	0...255
chan	1...255
mtype	1...255
pid	0...255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..,2^{n}-1$

Tabla 3.1: Tipos de datos en promela

```
mtype = { Autobus, Camion, Taxi };
mtype = { Vehiculo_con_motor,Vehiculo_sin_motor
};
init
{
  mtype n = Autobus; /* inicializan a Autobus */
  printf("El valor de n es");
  printm(n); }
}
```

Figura 3.5: Imprimiendo del nombre simbólico de la variable n.

Las variables de tipo `mtype` ayudan a mantener valores simbólicos, los cuales se deben introducir entre corchetes y no deben incluir ninguna de las palabras reservadas de *promela*. Si deseamos imprimir el nombre simbólico de una variable de tipo `mtype` debemos usar la función `printm`.

A veces es necesario crear nuestro propio tipo de dato, esto se puede hacer utilizando la palabra reservada `typedef` . En la Figura 3.6 se muestra la creación de un nuevo tipo de dato llamado `Nodo`. Una variable de tipo `Nodo` contiene un campo de tipo `byte`, uno del tipo arreglo de tres enteros y uno del tipo `bool`.

```
typedef Nodo{
byte a;
int b[3];
bool g;
};
```

Figura 3.6: Estructura haciendo uso de `typedef` .

Canales de mensajes

Los canales de mensajes son utilizados para intercambiar datos entre procesos, estos pueden ser declarados locales o globales. Si el canal es declarado en la parte superior del programa y fuera de los `proctype` entonces tendrá un alcance global, pero si es declarado dentro de un `proctype` tendrá un alcance local. Para declarar un canal usamos la palabra reservada `chan`.

El canal `CM` de la Figura 3.7 transporta tres tipos de datos, el primero es un entero seguido de un byte y otro entero. El canal tiene una capacidad de almacenamiento (lo cual define un canal asíncrono) de diez mensajes. También es posible definir arreglos de canales. En la Figura 3.8 se define un arreglo de cinco canales con capacidad de almacenamiento de 10 mensajes.

```
chan CM = [10] of { int, byte, int }
```

Figura 3.7: Canal donde se pueden almacenar 10 elementos (donde cada elemento esta conformado por dos enteros y un byte)

```
chan CM[5] = [10] of {int, byte, int }
```

Figura 3.8: Arreglo de 5 canales.

Para poder escribir un mensaje en una canal, se utiliza el símbolo ! después del nombre del canal y precedido del mensaje a enviar. Lo anterior se muestra en la Figura 3.9

```
int a=1;
byte
b=2;
int c=3;
CM!a,b,c
```

Figura 3.9: Envío de un mensaje por medio del canal `CM`.

La declaración de recepción de mensajes vía un canal de comunicación se realiza mediante el símbolo ?. La operación de recepción es ejecutable si y solo si hay un mensaje en el canal de comunicación. En la Figura 3.10 se recibe un mensaje del canal `CM` y se almacenan los datos recibidos en las variables e, f, y g.

```
int e;
byte f;
int g;
CM?e,f,g
```

Figura 3.10: Recibiendo un mensaje del canal `CM`.

Una alternativa equivalente para enviar y recibir mensajes es utilizar el primer campo del mensaje y encerrar los demás campos entre paréntesis como se muestra en la figura 3.11.

```
canal!expr1(expr2,expr3)
canal?var1(var2,var3)
```

Figura 3.11: Recepción de un mensaje haciendo una indicación.

Algunos o todos los campos de los mensajes pueden ser constantes. Esto pone la restricción de que para recibir el mensaje éste tiene que tener los mismos campos con valores constantes (ver Figura 3.12).

```
canal?const(expr2,expr3)
```

Figura 3.12: Recepción de un mensaje usando una constante.

También podemos utilizar la función `eval` para condicionar la recepción del mensaje, en este caso el mensaje debe traer el mismo valor de una determinada variable del receptor. En la Figura 3.13 se muestra un ejemplo donde el mensaje que se va a recibir debe traer en el primer campo el mismo valor de la variable `var1`.

```
canal?eval(var1),var2,var3
```

Figura 3.13: Recepción usando la función `eval`.

Es posible leer un mensaje de un canal sin eliminar el mensaje del canal, esto se realiza mediante la operación `poll`.

```
canal?<eval(var1),var2>
```

Figura 3.14: Recepción de un mensaje sin eliminarlo del canal.

En el ejemplo que se muestra en la Figura 3.14, la operación de recepción sólo se puede efectuar si hay un mensaje y el primer campo del mensaje en el canal tiene el mismo valor de `var1`. Cuando se efectúa la operación, el valor del segundo campo del mensaje en el canal es pasado a `var2` pero el mensaje no es removido del canal.

Si los canales de comunicación que se definen no tienen capacidad de almacenamiento, se dice que estos canales son síncronos. En el ejemplo que se muestra en la Figura 3.15, el proceso A se queda bloqueado si el proceso B no ejecuta la operación de recepción ya que el canal `name` no tiene capacidad de almacenamiento.

```
mtype = { msgtype };
chan name = [0] of { mtype, byte };
active proctype A(){
  name!msgtype(124);
  name!msgtype(121)}
  active proctype B() {
  byte state;
  name?msgtype(state)
}
```

Figura 3.15: Declaración de un canal síncrono entre un proceso A y uno B.

3.2.2. Control de flujo: Declaraciones compuestas

Promela tiene cinco operaciones compuestas

1. Secuencias atómicas.
2. Pasos determinísticos.
3. Selección.
4. Repeticiones.
5. Secuencias de escape.

Se pueden definir otras operaciones compuestas utilizando las macros de *promela* pero no entramos a detalle en ello. A continuación damos una breve explicación de las operaciones compuestas que se mostraron en la lista anterior.

Secuencias atómicas

Las secuencias atómicas nos permiten ejecutar una secuencia de pasos como si fuese una sola instrucción.

Usualmente se utiliza una secuencia atómica para inicializar simultáneamente un grupo de procesos.

```
1 init{
2 atomic {
3 run A(1,2);
4 run B(2,3)
5 } }
```

Figura 3.16: Ejemplo del uso de la instrucción `atomic`.

En la Figura 3.16 el proceso `init` crea de manera atómica dos procesos, por lo que estos procesos iniciaran simultáneamente su ejecución cuando `init` salga del segmento `atomic`.

Pasos determinísticos

Podemos declarar segmentos de códigos que sean completamente determinísticos, esto, utilizando la instrucción `d_step`. `d_step` es similar a `atomic` solo que debe ser completamente determinista y no incluir saltos desde o hacia fuera.

```
d_step{
tmp = b; /*Intercambio de valores entre a y b*/
b = a;
a = tmp
}
```

Figura 3.17: Ejemplo del uso de la instrucción `d_step`.

Selección

La selección es muy similar al `switch` y al `if` del lenguaje C, con ella podemos definir grupos de segmentos de los cuales en un determinado momento solo uno será ejecutado. Es ejecutado el segmento de código cuya guardia sea verdadera. El primer segmento de la selección se denomina guardia. Este es el primero en verificar si se puede ejecutar, de lo contrario se pasa al siguiente.

```
if
:: (a != b) ->opcion1
:: (a == b) ->opcion2
fi;
```

Figura 3.18: Uso de la instrucción `if`

Repetición

La repetición sirve para poder generar ciclos, esto por medio de la instrucción `do :: od`. Los segmentos después de los dos puntos son verificados. Es ejecutado el segmento cuya guardia sea verdadera, si hubiera dos segmentos que pudieran ser ejecutados se escoge uno de manera no determinística.

```
do
:: O1 ->opcion1
:: O2 ->opcion2
:: O3 ->opcion3
:: .
:: .
:: .
od
```

Figura 3.19: Uso de la instrucción `do`

Secuencias de escape

Las secuencias de escape sirven para poder establecer un orden en la ejecución de dos segmentos, para lo cual se utiliza la palabra reservada `unless`. Cuando se declara A `unless` B se indica que el segmento A puede ser ejecutada siempre y cuando el segmento B no se pueda ejecutar. Cuando B pueda ser ejecutado se se cambia el control de la ejecución de A a B.

En la Figura 3.20 se muestra un pequeño ejemplo hecho en *promela*. En este ejemplo se crean tres procesos: *P*, *C* e `init`. El proceso `init` crea a los procesos *P* y *C* utilizando la llamada `run` (líneas 20-21), note que esto se hace atómicamente lo que origina que tanto el proceso *P* como el *C* inicien su ejecución al mismo tiempo. Los procesos *P* y *C* se sincronizan utilizando la variable *x*. Cuando la variable *x* tiene un valor 0 el proceso *P* puede asignar un valor 1, 2 o 3 a *y*, la selección de que valor le asignará la hace no determinística.

```
1    byte x, byte y;
2    proctype P(){
3    do
4    :: x==0 →
5      do
6      :: true → y = 1;
7      :: true → y = 2;
8      :: true → y = 3;
9      od
10       x = 1;
11   od;
12   }
13   proctype C(){
14   do
15   :: x==1 →
16     x = 0;
17   od;
18   }
19   init{
20   atomic{
21     run P();
22     run C()
23   }
24   }
```

Figura 3.20: Simple programa en *promela* donde se crean dos procesos (P y C) por medio de la instrucción `run`.

El proceso *C* asigna el valor de 0 a *x* (línea 16) paraque el proceso *P* escriba un nuevo valor en *y*.

Definiciones inline

Por medio de `inline` *promela* provee un mecanismo para estructurar un programa usando módulos similares a un procedimiento en lenguaje C pero funcionando como macros, es decir,

```
1    inline ejemplo() {
2      printf("El valor de x es %d",x){
3    }
4    active proctype principal() }
5     int x=0;
6       do
7      ::x%2==0 ->
8        x=x+1;
9        ejemplo();
10      ::else ->
11       x=x+2;
12       ejemplo();
13     od;
14      }
```

Figura 3.21: Ejemplo de la definición `inline`.

cuando el verificador *Spin* encuentra una llamada a un inline la sustituye por código dentro de esta llamada.

La Figura 3.21 muestra un ejemplo del uso de `inline`. Observamos en este código que la variable `x` es usada en `inline ejemplo` (línea 2) dado que fue declarada en `proctype principal`.

Never-Claim

Un `never-claim` nos ayuda a especificar un comportamiento que nunca debe ocurrir en nuestro sistema. Para escribir un `never-claim` se debe hacer uso de la la palabra reservada `never` como se muestra en la Figura 3.22.

```
never{
}
```

Figura 3.22: Declaración de un `never-claim`.

Los `never-claim` son usados por *Spin* para verificar una propiedad en el modelo *promela*. En la Figura 3.23 se muestra un never-claim para verificar que se cumpla una propiedad cualquiera p en un modelo de un sistema. Si la propiedad p no se llegara a cumplir en el sistema donde se está verificando se sale del ciclo y *Spin* reportar un error.

```
 never{
do
:: !p -> break
:: else
od
}
```

Figura 3.23: Ejemplo de un `never-claim` para verificar unapropiedad invariante p.

3.3. Marco teórico

3.3.1. Traducción de programa-autómata

Antes de explicar cómo funciona el *model-checking*, comenzamos dando algunas definiciones sobre la teoría de autómatas y, posteriormente su relación con la lógica temporal.

Definición 3.3.1. *Un autómata finito es un tupla (S, s_0, L, T, F), en donde*

S *es un conjunto finito de estados,*

s_0 *es el estado inicial distinguido,* $s_0 \in S$,

L *es un conjunto finito de etiquetas,*

T *es un conjunto de transiciones* $T \subseteq (S \times L \times S)$, *y*

F *es un conjunto de estados finales,* $F \subseteq S$

Si el estado sucesor de cada transición está definido únicamente por el estado fuente y la etiqueta de transición, entonces decimos que el autómata es determinista. El determinismo en un autómata es definido formalmente como sigue.

Definición 3.3.2. *Un autómata finito (S, s_0, L, T, F) es determinista si, y solo si,*
$\forall s \in S$, *si* $a \in L \wedge ((s, a, s') \in T \wedge (s, a, s'') \in T) \rightarrow s' = s''$

Normalmente los autómatas que usamos no tienen esta propiedad, esto es, especifican comportamientos no determinísticos.

Definición 3.3.3. *Una ejecución de un autómata finito (S, s_0, L, T, F) es un conjunto posiblemente infinito ordenado de transiciones.*
$\{(s_0, a_0, s_1), (s_1, a_1, s_2), (s_2, a_2, s_3), ...\}$
tal que
$\forall (i \geq 0) \rightarrow (s_i, a_i, s_{i+1}) \in T \wedge s_i, s_{i+1} \in S \wedge a_i \in L$

Note que para autómatas no deterministas la secuencia de estados visitados no puede ser necesariamente derivada de la secuencia de etiquetas de transiciones, y viceversa.

Definición 3.3.4. *Una ejecución de aceptación de un autómata finito (S, s_0, L, T, F) es una ejecución finita en donde la transición final* (s_{n-1}, a_{n-1}, s_n) *tiene la propiedad que* $s_n \in F$. *Una ejecución es considerada aceptada si y solo si termina en un estado final del autómata.*

Consideremos el programa que se muestra en la Figura 3.24. En él se utiliza una variable de tipo `byte` llamada `x`. Si `x` tiene el valor cero (línea 4) se ejecuta `x = x + 1` (línea 5), de lo contrario, si el valor de `x` es distinto de cero (línea 6), se ejecuta `x = x -1` (línea 7). Finalmente, independientemente del valor de `x`, se ejecutará `x = x + 2` (línea 9). Para generar el autómata que representa a este programa debemos tomar en cuenta la secuencia de asignaciones y comparaciones del programa ya que éstas representarán las etiquetas de las transiciones de nuestro autómata.

De manera que al ejecutar una comparación o asignación se generan estados (estos estados no necesariamente son nuevos, por ejemplo, los ciclos pueden generar estados que ya fueron creados anteriormente).

```
1    byte x=5;
2    active proctype proceso(){
3      if
4      :: x==0 →
5        x = x + 1;
6      :: x!=0 →
7        x = x - 1 ;
8      fi;
9      x = x + 2;
10   }
```

Figura 3.24: Simple programa en *promela.*

El autómata que representa el programa de la Figura 3.24 se muestra en la Figura 3.25. Este autómata no toma en cuenta los valores de la variables ya que representa de manera general el comportamiento de nuestro programa. Este autómata recibe el nombre de *autómata no extendido del programa (ANEP).* A partir del *ANEP* podemos generar el *autómata extendido del programa (AEP)* el cual toma en cuenta los valores de las variables. Para nuestro programa, dado que el valor inicial de x es 5 la condición `x != 0` se cumple (ya que $5 \neq 0$), por lo que se ejecuta la asignación `x = x -1` esto asigna el valor de 4 a x. Por último se ejecuta `x = x + 2` por lo que el valor final de x es 6. El *AEP* para este caso se muestra en la Figura3.26.

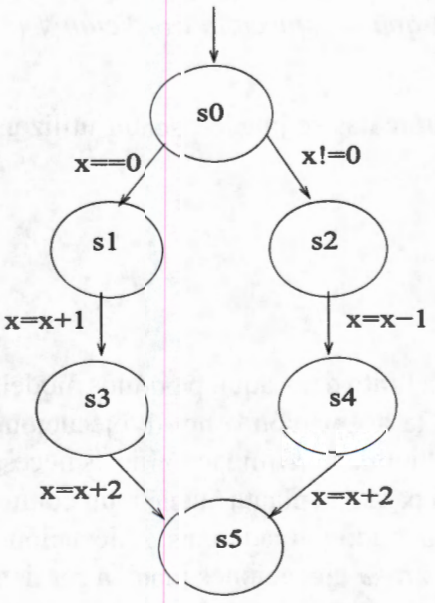


Figura 3.25: Autómata generado a partir del código de la Figura 3.24.

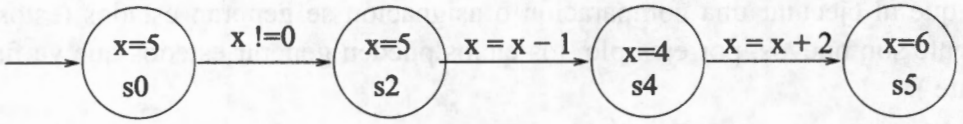


Figura 3.26: Autómata extendido generado a partir del autómata de la Figura 3.25.

Los sistemas por lo general tienen más de un proceso y como mencionamos cada proceso tiene su respectivo modelo representado por un autómata. Para representar al sistema completo debemos modelar todas las posibles interacciones de los procesos, para lo cual se utiliza el producto asíncrono entre los autómatas de los procesos. El producto asíncrono de autómatas se define de la siguiente manera:

Definición 3.3.5. ***Producto asíncrono***

El producto asíncrono de un conjunto de autómatas de estados finitos A_1, A_2,..., A_n es un autómata finito de estados $A=(S, s_0, L, T, F)$, en donde:

- *S es el producto cartesiano $S_1 \times S_2 \times ... \times S_n$*
  donde S_i es el conjunto de estados de A_i
- *s_0 es la tupla $(s_{0_1}, s_{0_2}, ..., s_{0_n})$ donde $s_{0_i} \in A_i$*
- *L es el conjunto $L_1 \cup L_2 \cup ... \cup L_n$ donde $L_i \in A_i$*
- *T es el conjunto de tuplas $((x_1, ..., x_n), I, (y_1, ..., y_n))$ tal que $\exists\, i,\ 1 \leq i \leq n, (x_i, I, y_i) \in T_i$, y $\forall\, j,\ 1 \leq j \leq n, j \neq i \rightarrow (x_i = y_j)$*
- *F es el subconjunto de A que satisfacen la condición $\forall\, (s_1, ..., s_n) \in F, \exists\, i, s_i \in F_i$ donde $F_i \in A_i$*

El producto asíncrono de autómatas se puede escribir utilizando el símbolo $\prod$ de la siguiente manera:

$$S = (\prod_{i=1}^{n} A_i)$$

Aceptación omega

Con la definición de autómata finito dada aquí, podemos modelar la terminación de la ejecución, pero no podemos decidir sobre la aceptación o no de ejecuciones infinitas. La necesidad de tal aceptación surge en los casos en donde la terminación no es necesariamente un resultado deseable, tales como el software de control para una planta nuclear, un conmutador telefónico, o un semáforo.

Una ejecución infinita es a menudo llamada una ω-ejecución (pronunciada omega ejecución). Las propiedades de aceptación para ω-ejecuciones pueden ser definidas de diferentes maneras. La que adoptaremos fue establecida por J.R. Büchi[1960].

Si σ es una ejecución infinita. El símbolo σ^{ω} representa el conjunto de estados que aparecen infinitamente frecuente dentro del conjunto de transiciones de σ, σ^{+} el conjunto de estados que aparecen sólo finitamente muchas veces.

La notación de aceptación de Büchi es definida como sigue.

Definición 3.3.6. *Una ω-ejecución de aceptación de un autómata finito (S,s0, L, T, F) es cualquier ejecución infinita σ tal que $\exists s_f$, $s_f \in F \wedge s_f \in \sigma^{\omega}$, esto es, una ejecución infinita es aceptada si y solo si algún estado en F es visitado infinitamente frecuente en la ejecución.*

Para ilustrar lo anterior considere el siguiente autómata A={S, s_0, L, T, F } en donde:

- S = {s_0, s_1}
- s_0 = {s_0}
- L = {a, b}
- T = {(s_0, a, s_0), (s_0, b, s_1), (s_1, a, s_0), (s_1, b, s_1)}
- F = {s_1}

La representación de este autómata se muestra en la Figura 3.27. ¿Cuál es el lenguaje que acepta este autómata?. Debido a que el único estado final del autómata de la Figura 3.27 es s_1, el lenguaje que acepta *A* son aquellas cadenas que tienen un número infinito de símbolos *b*.

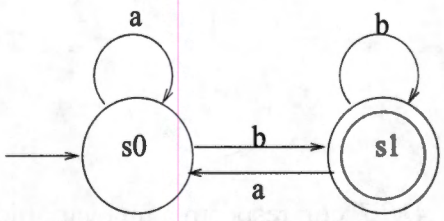


Figura 3.27: Autómata de Büchi que acepta cadenas que tienen infinitas b's.

Una $\omega-$ejecución de aceptación para el autómata de la Figura 3.27 puede ser:

σ = s_0, s_0, s_0, s_1, s_1, s_1, s_1, s_1, s_1, s_1...

Para este ejemplo σ^{+}=s_0, s_0, s_0 y σ^{ω} = s_1, s_1, s_1, s_1...

La regla de extensión *stutter*

La definición de aceptación aplica sólo para ejecuciones infinitas. Una manera de extender la noción de aceptación para ejecuciones finitas es usando la transición especial conocida como *stutter*. Para ello anexamos la etiqueta nula ϵ al conjunto *L* la cual representa una no-operación que es siempre ejecutable y no tiene ningún efecto. Esto es, anexamos las transiciones $(s_i, \epsilon, s_i)\, \forall s_i \in S$ a nuestro autómata. La extensión stutter de una ejecución finita puede ser definida como sigue.

Definición 3.3.7. *La extensión stutter de una ejecución finita σ con estado final s_n es la ω-ejecución σ, tal que (s_n, ε, s_n).*

El estado final persiste por siempre repitiendo la acción nula ϵ. Tal ejecución satisface las reglas Büchi aceptación dado que $s_n \in F$.

Es común referirse a este tipo de autómatas como autómatas de Büchi[Muk96].

Muchas propiedades interesantes de autómatas de Büchi han sido mostradas como decidibles. La mayoría de estas aplican para verificar si el lenguaje de un autómata es vacío, o si el lenguaje de la intersección de autómatas es vacío o no.

Hasta ahora hemos visto cómo generar el modelo que representa a nuestro sistema. Ahora veamos cómo formalizar la propiedades a verificar sobre el sistema.

Considere el autómata de la Figura 3.28 con el estado inicial s_0 y final s_1. La formalización de una propiedad que describa una ejecución correcta de este autómata requiere de la habilidad para interpretar y distinguir las buenas ejecuciones de las malas.

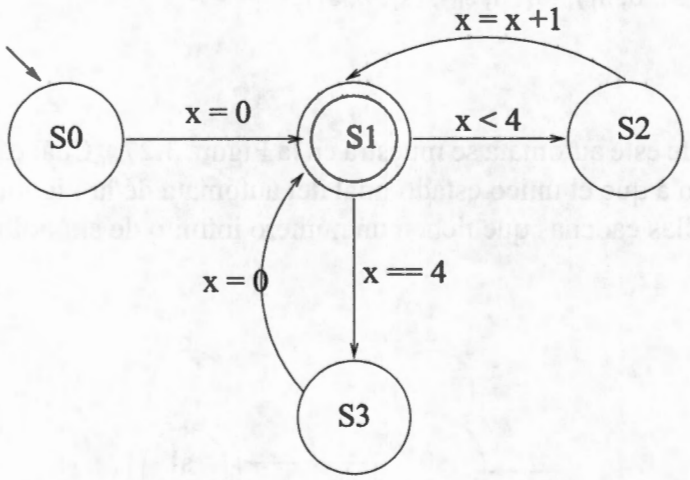


Figura 3.28: *ANEP* con respecto a una variable llamada x.

Dado que a x le asignamos el valor 0 y pasamos a s_1, el *AEP* nos da la secuencia de estados visitados durante la ejecución así como el valor de x en cada estado.

(s0, ?), (s1, 0), (s2, 0), (s1, 1), (s2, 1), (s1, 2), (s2, 2), (s1, 3), (s2, 3), (s1, 4), (s3, 4), (s1, 0), ...

Teniendo al *AEP* podemos formular propiedades sobre él. Las propiedades más interesantes se ocupan de los valores alcanzables y no alcanzables de x durante una ejecución. Considere, por ejemplo, las siguientes propiedades:

p: "El valor de x no es mayor a cinco"

q: "El valor de x es cuatro"

r: "El valor de x es par"

Podemos deducir un valor de verdad para p, q y r en cada estado del sistema extendido. Estos tipos de propiedades son formalmente llamados fórmulas de estado. La secuencia de los valores de verdad para p, q y r es:

p: verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, verdadero, ...

q: falso, falso, falso, falso, falso, falso, falso, falso, verdadero, verdadero, falso, ...

r: verdadero, verdadero, falso, falso, verdadero, verdadero, falso, falso, verdadero, verdadero, verdadero, ...

Como observamos podemos hacer enunciados sobre posibles e imposibles secuencias de valores booleanos para *p*, *q* y *r* durante toda la ejecución. Estos enunciados pueden ser evaluados sobre la(s) ejecución(es) del sistema. Algunos ejemplos de estos enunciados son:

1. *p* es invariante verdadero,
2. *p* eventualmente llegará a ser falso por siempre,
3. *q* implica *p*,
4. *p* implica eventualmente *q*.

Estos enunciados pueden ser examinados en las ejecuciones de nuestro sistema y así podemos verificar si se cumplen o no. Para esto hacemos uso de la lógica temporal, la cual nos permite razonar sobre relaciones causales y temporales de nuestro sistema y así, poder expresar por medio de esta el comportamiento que deseamos que tenga el sistema. Una fórmula de la lógica temporal puede ser transformada a un autómata de Büchi[Pnu77]. Al tener el sistema y propiedades expresados como autómatas, podemos distinguir dos lenguajes, el lenguaje de las ejecuciones del sistema como $L(A_p)$, y el lenguaje de la propiedad deseada $L(A_{prop})$.

3.3.2. Lógica temporal

La lógica temporal fue estudiada a finales de los 60 y principios de los 70, como una herramienta para argumentos filosóficos que envolvían el paso del tiempo. Un primer artículo proponiendo la utilización de este tipo de lógica para el análisis de sistemas distribuidos fue realizado por Amir Pnueli en 1977[Pnu77]. Tomó más de una década, pensar, la importancia fundamental de estas ideas para ser mayormente utilizada.

Como mencionamos, la lógica temporal sirve para formalizar las propiedades de la ejecución de un sistema, para ello se utiliza un pequeño número de operadores temporales. Una rama de la lógica temporal muy relevante para la verificación de sistemas con procesos asíncronos es conocida como Lógica Temporal Lineal (LTL). La semántica de LTL es definida sobre ejecuciones infinitas. Sin embargo, con la ayuda de la regla de extensión stutter, se puede aplicar igualmente a ejecuciones finitas.

Las propiedades más interesantes se ocupan de los valores alcanzables y no alcanzables de la variables del sistema durante una ejecución. Las propiedades que deseamos verificar se expresan por medio de proposiciones, tal como *p="El valor de* x *es par"* la cual se mencionó en el ejemplo de la Figura 3.28.

Una formula temporal bien formada es construida de formulas de estado y operadores lógicos y/o temporales usando las siguientes dos reglas básicas.

Definición 3.3.8. Fórmulas temporales bien formadas

1. Todas las fórmulas de estado, incluyendo falso y verdadero son fórmulas temporales bien formadas.
2. Si α y β son fórmulas de estado, entonces, $\neg\alpha$, $\alpha \vee \beta$, $\alpha \wedge \beta$, $\Box\alpha$, $\Diamond\alpha$, $\bigcirc\alpha$ y $\alpha \cup \beta$ son fórmulas bien formadas.

A continuación presentamos los principales operadores temporales que se usan en el *model-checking*.

Definición 3.3.9. Until(hasta): $\cup$

La fórmula $p \cup q$ indica que la propiedad q se cumple en una ejecución mientras la propiedad p no se cumpla. Esto se muestra gráficamente en la Figura 3.29. Los rectángulos de esta figura representan los estados visitados durante la ejecución. Cuando un rectángulo tiene la letra p o q, significa que en ese estado se cumplió la propiedad p o q respectivamente.

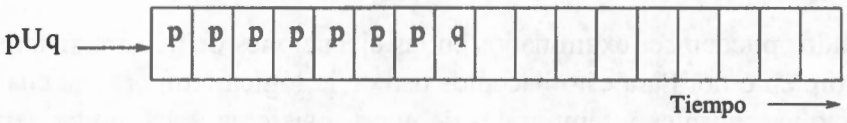


Figura 3.29: Representación en lógica temporal de la fórmula $p \cup q$.

Definición 3.3.10. Always(siempre): $\Box$

La fórmula $\Box p$ (siempre p o always p) indica que la propiedad p se mantiene verdadera durante toda la ejecución. Gráficamente esto se muestra en la Figura 3.30.

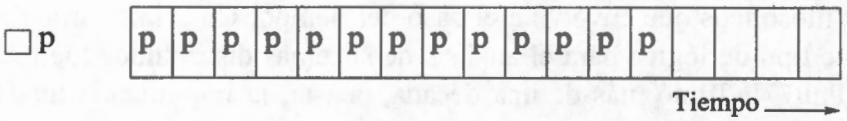


Figura 3.30: Representación en lógica temporal de de la fórmula $\Box p$.

Definición 3.3.11. Eventually(eventualmente): $\Diamond$

La fórmula $\Diamond p$ (eventually p o eventualmente p) captura la noción que la propiedad p es verdadera al menos una vez en un estado de ejecución. Esto se muestra gráficamente en la Figura 3.31.

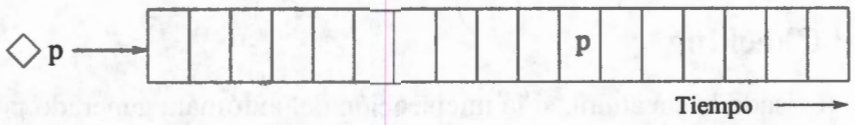


Figura 3.31: Representación en lógica temporal de la fórmula $\diamond p$.

3.3.3. De lógica a autómata

Se mostró a mediados de los ochenta que para cada fórmula de lógica temporal existe un autómata Büchi . Hay algoritmos eficientes que pueden convertir automáticamente cualquier fórmula de lógica temporal en el equivalente autómata de Büchi. Aquí mostraremos un ejemplo, una discusión más detallada se puede encontrar en [CGP99] [Hol04] [Wol02] [PGS01].

Si deseamos probar $\Box p$ entonces el autómata que identifica este comportamiento esta dado por el que se muestra en la figura 3.32

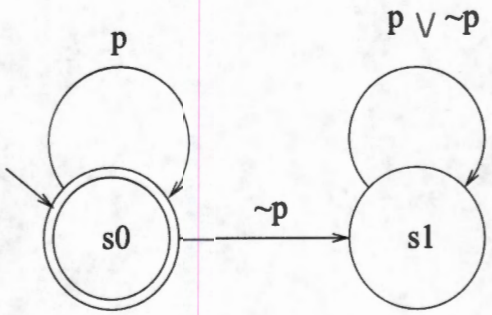


Figura 3.32: Autómata que representa la fórmula temporal $\Box p$, donde s_0 es el estado inicial y final.

Si evaluamos todas las ejecuciones de nuestro sistema y existe una ejecución en donde la proposición $\Box p$ es falsa entonces podemos concluir que el sistema no cumple con la propiedad. De manera general si alguna de las ejecuciones presentara un comportamiento que sea equivalente a una cadena aceptada por el autómata de la figura 3.33, entonces $\Box p$ no se cumple, ya que este autómata representa a $\sim \Box p$.

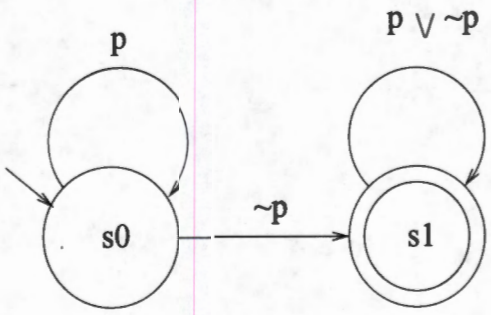


Figura 3.33: Autómata que representa la fórmula temporal $\sim \Box p = \diamond(\sim p)$, donde , donde s_0 es el estado inicial y s_1 el estado final.

3.3.4. Model Checking

Como hemos revisado hasta ahora, si la intersección del autómata generado por el programa y el autómata generado por la negación de la propiedad que se esta verificando es vacío, entonces la propiedad se cumple en el programa. Hasta ahora hemos revisado cómo generar el autómata del programa y el de la propiedad a verificar. Para ver cómo puede realizarse la intersección de autómatas el lector puede consultar [Hol04], [CGP99] y [PGS01]. En la siguiente sección se explica brevemente como realiza todo esto la herramienta *Spin*; se explica sus componentes y su funcionamiento.

Capítulo 4

Verificación de una aplicación SPMD de procesamiento de datos que integra un algoritmo de distribución cíclico

"El sentido común no es nada común."

–Francois Marie Arouet Voltaire 1694-1778 Escritor y filósofo francés

Como hemos visto en capítulos anteriores, los algoritmos de distribución de carga son complejos de estudiar debido a su naturaleza, lo que conlleva a que su implementación no necesariamente sea correcta. Como punto de partida para el proceso de verificación de este tipo de sistemas comenzamos con una propuesta de una estructura básica de interacción de un algoritmo de distribución de carga con una aplicación de procesamiento independiente de datos. El algoritmo de distribución utilizado en esta estructura es una política cíclica la cual se eligió no tanto por su eficiencia sino por ser una de las políticas más conocidas con una dificultad mediana de implementación. Este algoritmo no considera la implementación de una política de información para recolectar estados de carga de todos los procesos del sistema.

Para garantizar el buen funcionamiento del sistema asociado a la arquitectura básica, proponemos un conjunto de propiedades las cuales deben ser validadas independientemente del algoritmo de distribución usado. Dichas propiedades son verificadas sobre el modelo del sistema usando la herramienta *Spin*.

4.1. Estructura básica de integración Aplicación-Distribuidor

Como se mencionó en el capítulo 1, podemos identificar dos maneras de particionar un problema: dividiendo el código en pequeñas tareas, o ejecutando el mismo código sobre conjuntos distintos de datos a través de una programación SPMD. Debido a su simplicidad, este último modelo de programación ha llegado a ser uno de los modelos de programación más utilizados para desarrollar aplicaciones paralelas [BDV94] [Han98].

En un sistema de memoria distribuida la estructura básica de un programa SPMD puede ser como la que se muestra en el código de la Figura 4.1. Un proceso aplicación (identificado por su

identificador de proceso ó _pid) es ejecutado en cada procesador. La primer instrucción es para obtener (por medio de un proceso maestro o generados localmente) los datos locales a procesar (línea 2). Los datos son almacenados por cada procesador en una lista local L. Entonces, a través de un ciclo (líneas 3-6), elementos de L son obtenidos para ser procesados (líneas 4-5). Nuestro protocolo de distribución considera que el procesamiento de los elementos de la lista son independientes entre si. Después del procesamiento local de los datos, normalmente son necesarias algunas comunicaciones entre los procesadores a fin de tener un resultado global (línea 7).

```
1   Proceso_Aplicación(_pid) {
2     Generacion_De_Datos_Locales(L,_pid);
3      Mientras ( ¬(Vacia(L)){
4        item=Obten_Elemento(L);
5        Procesando(item,resultado_local);
6      }
7      Resultado_Global(resultado_local,_pid);
8      Escribe("Fin de ejecución");
9   }
```

Figura 4.1: Estructura general de un programa SPMD.

En este tipo de algoritmos paralelos la cantidad de datos a procesar puede ser distribuida equitativamente entre los procesadores si se maneja un número determinado de datos. Sin embargo, un buen desempeño no siempre puede ser obtenido si los datos se generan durante la ejecución o si presentan diferentes granularidades (tiempos de procesamiento), o si estamos trabajando con sistemas paralelos heterogéneos. En estos casos es posible que un procesador esté descargado mientras otros estén sobrecargados.

Un enfoque para mejorar el rendimiento es la integración de una política de distribución de carga dentro del algoritmo SPMD para distribuir el trabajo del sistema. En el código de la Figura 4.2 mostramos cómo modificar la estructura del programa SPMD del código de la Figura 4.1 a fin de proponer una estructura básica que integre tal política. La estructura básica es representada por el proceso Proceso_aplicacion_conDC.

```
1    Proceso_aplicacion_conDC(_pid) {
2      Terminación=0;
3      Generacion_De_Datos_Locales(L,_pid);
4      Mientras(Terminación < NUMERO_DE_PROCESOS) {
5        Politica_De_Distribucion(pid,&Terminación);
6        Si(¬(Vacia(L)) {
7          item=get(&L);
8          Procesando(item,resultado_local);
9        }
10      }
11     Resultado_global(resultado_local,_pid);
12     Escribe("Fin de ejecución");
13   }
```

Figura 4.2: Estructura básica de integración de una política de distribución en un programa SPMD.

En el código de la Figura 4.2, el procesamiento de datos y la política de distribución coexisten. Las instrucciones de las líneas 3 y 11 se comportan similar a la versión previa para generar datos locales iniciales y obtener un resultado global. A causa de la posible transferencia dinámica de datos entre los procesadores, ahora la salida del ciclo no debe depender más del tamaño de la lista local. Una lista local puede permanecer vacía temporalmente mientras la política de distribución está buscando datos remotos. La función `Politica_De_Distribucion()` (línea 5) implementa una política de distribución particular. El objetivo principal de esta función es transferir parte de los elementos de la lista local hacia otros procesadores descargados, o bien; insertar en la lista local elementos transferidos por procesadores cargados, cuando ésta ha quedado vacía. En este caso desarrollamos una política de distribución cíclica la cual se distingue por tener un funcionamiento sencillo basado en un control distribuido sin elemento de información. La función `Politica_De_Distribucion()` regresa como parámetro la variable `terminación` que indica cuantos procesos del sistema han terminado sus datos. La salida del ciclo depende del valor de la variable terminación. El valor de esta variable debe ser igual al número total de procesos para que salga del ciclo. Después de llamar a la función de distribución de datos, si la lista local de un proceso no está vacía, se procede a obtener un elemento `item` de L (línea7) y se procesa ejecutando la función `Procesando` (línea 8). A diferencia de la función `Procesando` del código de la Figura 4.1, en la estructura básica propuesta la invocación a `Procesando` permite una creación dinámica de datos pasando a `L` como argumento. Después de esta llamada L puede quedar con una cantidad de elementos igual o mayor a la que tenía.

4.2. Política de distribución cíclica

En el proceso aplicación presentado en el código de la Figura 4.2, el procesamiento es alternado con la llamada a la política de distribución. La llamada a la función `Politica_De_Distribucion` en la estructura básica propuesta invoca la ejecución de una política de distribución cíclica. La implementación de la política de distribución se realizó de tal modo que se activa cuando una lista local `L` en el sistema esta vacía. De esta manera, si la carga es la misma para todos los procesadores, podrían terminar su procesamiento de datos al mismo tiempo, no teniendo que haber transferido datos durante la ejecución.

El funcionamiento del algoritmo cíclico requiere que los procesos (procesadores) estén organizados en una topología de anillo lógico. En este caso cada proceso tiene un identificador `id`, $0 \leq$ `id` $\leq$ (`NUMERO_DE_PROCESOS` -1). Un proceso puede obtener el identificador de su sucesor o predecesor en el anillo llamando a las funciones `Sucesor(id)` o `Predecesor(id)`. En esta implementación cada proceso tiene su propia `Ficha`, que es usada para hacerla circular sobre el anillo cuando se quiera solicitar transferencia de datos. Una `Ficha` siempre se envía hacia el proceso sucesor y se recibe del predecesor siguiendo una comunicación asíncrona. Una `Ficha` tiene tres elementos de información: `Ficha.Propietario`, `Ficha.Lista` y `Ficha.Un_dato`. `Ficha.Propietario` indica el identificador del proceso propietario de la `Ficha`. El elemento `Ficha.Lista` representa la lista donde los datos pueden ser insertados para ser transferidos a `Ficha.Propietario`. El tercer elemento, `Ficha.Un_dato` es una bandera que es `true` cuando ningún proceso a puesto datos en `Ficha.Lista` debido a que contaban únicamente con un dato en su lista local. El código de la Figura 4.3 ilustra el comportamiento de la política de distribución cíclica.

Cuando un proceso encuentra su lista vacía envía a su sucesor una petición `DAME_DATOS` usan-

do para ello su Ficha asociada (líneas 2-7). Cuando un mensaje de petición de datos ha sido enviado, la bandera Peticion_enviada es true (inicialmente tiene el valor false) indicando que el proceso está esperando-datos (línea 5) y evitando así transferencias sucesivas de peticiones de datos.

Después el proceso permanece monitoreando la llegada de su Ficha que está viajando a través de un mensaje en el anillo de procesos. Una Ficha puede ser recibida por medio de un mensaje DAME_DATOS o bien TERMINAR enviado por el proceso predecesor (líneas 8-9).

```
1   Politica_De_Distribucion(pid,Terminación){
2     Si (Vacia(L) y (Peticion_enviada==false)){
3       Ficha.Propietario=id;
4       Ficha.Lista = NULL;
5       Peticion_enviada = true;
6       Envia(Sucesor(id), DAME_DATOS,Ficha)
7     }
8     Si (llego_ficha()){
9       Recepcion(Predecesor(id),tipo,Ficha);
10       Caso (tipo) {
11         DAME_DATOS:
12           Si (Ficha.Propietario)
13             Llega_Respueta( Ficha, id, Peticion_enviada,
               Datos_en_transferencia);
14           Otro
15             Responder(Ficha, id, Peticion_enviada,
              Datos_en_transferencia);
16         TERMINAR: {
17           Terminacion = Terminacion + 1;
18           Si (Ficha.Propietario ≠ id)
19             Envia(Sucesor(id), TERMINAR,Ficha)
20       } } }
21 }
```

Figura 4.3: Política de distribución cíclica.

Cuando un mensaje DAME_DATOS es recibido (líneas 11-15 del código de la Figura 4.3), dos situaciones pueden ocurrir, que la Ficha del mensaje pertenezca al proceso receptor o a algún otro proceso en el anillo. En el primer caso la Ficha representa una respuesta al mensaje DAME_DATOS enviado por el proceso receptor. En otro caso el proceso receptor necesita responder la petición de datos enviada por otro proceso. Ambos casos serán explicados en los códigos de la Figuras 4.4 y 4.5 respectivamente. Si un proceso recibe un mensaje TERMINAR (líneas 16-20), entonces significa que no hay más datos en el sistema. De manera que el proceso está listo para terminar de procesar y espera la recepción de otros NUMERO_DE_PROCESOS-1 mensajes TERMINAR, viniendo del resto de los procesos del sistema. La variable Terminacion se incrementa en uno por cada recepción de un mensaje TERMINAR (línea 17) y cuando alcanza el valor de NUMERO_DE_PROCESOS el proceso Aplicacion_con_DC termina, finalizando el programa. En este caso, si la Ficha recibida en el mensaje no pertenece al proceso receptor, entonces se retransmite al sucesor (líneas 18-19).

En el caso de que un proceso reciba un mensaje DAME_DATOS con una Ficha que no le pertenece, sigue el comportamiento que se muestra en el código de la Figura 4.4:

```
1    Responder(Ficha, id, Peticion_enviada, Datos_en_transferencia);
2     Si (¬ Vacia(Ficha.Lista)){
3       Si (Peticion_enviada == true)
4         Datos_en_transferencia=true;
5     } Otro
6       Si (Tamaño(L) == 1)
7         Ficha.Un_dato=true;
8       Otro
9         Si (Tamaño(L) ≥ 2)
10           Ficha.Lista=Mitad(L);
11     Envia(Sucesor(id),DAME_DATOS,Ficha)
12    }
```

Figura 4.4: Manejador del un mensaje DAME_DATOS que no pertenece al proceso receptor.

Si la Ficha recibida tiene su campo Ficha.Lista no vacío, significa que algún proceso en el sistema ha transferido algunos datos al proceso Ficha.Propietario (línea 2 del la código de la Figura 4.4). En este caso, si el proceso receptor está esperando datos coloca su variable Datos_en_transferencia a true (inicialmente tiene el valor false), para señalar que hay datos siendo transferidos sobre el anillo. Esta bandera ayuda a evitar una terminación temprana de la política de distribución (líneas 3-4). En otro caso, si la Ficha recibida tiene la Ficha.Lista vacía (línea 5), el proceso receptor examina su lista local. Si el proceso receptor tiene mas de un dato en su lista local (línea 9), saca de ella datos para insertarlos en Ficha.Lista (línea 10), pero si tiene un sólo dato no hace una transferencia pero lo indica cambiando el valor de Ficha.Un_dato a true (Línea 7). Finalmente, en los tres casos la Ficha es retransmitida al proceso sucesor (línea 11).

En el código de la Figura 4.5 mostramos el fragmento de código que atiende la respuesta de cuando la Ficha cuyo mensaje DAME_DATOS dio la vuelta completa al anillo y regresa nuevamente al proceso origen.

```
1    Llega_Respuesta( Ficha, id, Peticion_enviada,
     Datos_en_transferencia);
2     Si (¬ Vacia(Ficha.Lista)){
3       L=Ficha.Lista;
4       Peticion_enviada = false
5     }Otro
6       Si ((Datos_en_transferencia == true)∨ (Ficha.Un_dato==true)){
7         Peticion_enviada = false
8         Datos_en_transferencia=false;
9       } Otro
10         Envia(Sucesor(id), TERMINAR,Ficha)
11    }
```

Figura 4.5: Manejador de un mensaje DAME_DATOS con Ficha que pertenece al proceso receptor.

Si hay datos remotos insertados en `Ficha.Lista`, el receptor los inserta en su lista local `L` para ser procesados (líneas 2-3). Además, la bandera `Peticion_enviada` es puesta a `false` (línea 4) indicando que el proceso no está más esperando datos.

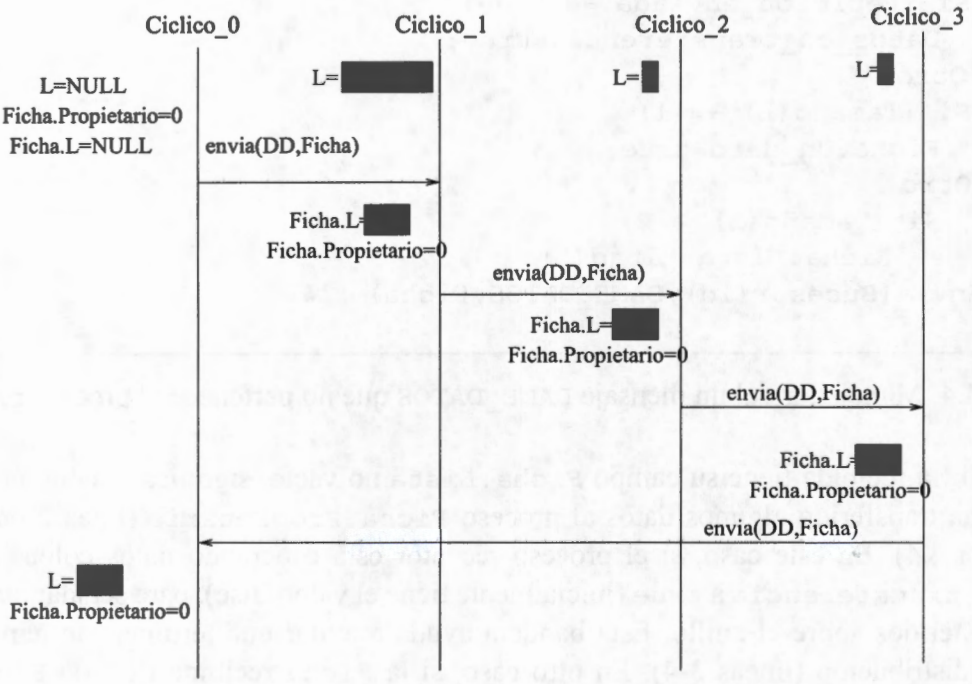


Figura 4.6: Algoritmo de distribución cíclico en ejecución.

La Figura 4.6 muestra un ejemplo donde intervienen 4 procesos (`Cíclico_0, Cíclico_1, Cíclico_2 y Cíclico_3`). En este ejemplo el proceso `Cíclico_0` detecta que no tiene datos, por lo que cambia el valor de su variable `Peticion_enviada` a `true` y procede a enviar un mensaje `DAME_DATOS` con su `Ficha` al proceso `Cíclico_1`. El proceso `Cíclico_1` al percatarse que la `Ficha` de `Cíclico_0` no tiene datos, le agrega la mitad de sus datos a la `Ficha`. Debido a esto los procesos `Cíclico_2` y `Cíclico_3` sólo retransmiten la Ficha a su respectivo sucesor. Mientras la Ficha de `Cíclico_0` está circulando por el anillo el proceso `Cíclico_0` no puede hacer una nueva petición de datos dado que `Peticion_enviada` es `true`. Cuando `Cíclico_0` recibe su `Ficha` obtiene los datos de ella y cambia el valor de `Peticion_enviada` a `false` para poder hacer una petición de datos cuando termine nuevamente de procesar sus datos.

Cuando un proceso recibe su `Ficha` pero `Ficha.Lista` está vacía, (no le transfirieron datos), examina el valor de su bandera `Datos_en_transferencia` y el valor de `Ficha.Un_dato` (líneas 6). Si `Datos_en_transferencia` o `Ficha.Un_dato` es `true` , significa que mientras el proceso estaba esperando datos, detectó algunos `datos` siendo transferidos por el anillo en dirección de otro proceso o que no le dieron datos ya que solamente se contaba con uno (el cual puede generar más datos después de su procesamiento). De manera que, el proceso pone el valor de `Peticion_enviada` y `Datos_en_transferencia` a `false` (líneas 7 y 8) permitiendo que el procedimiento de petición de datos sea lanzado nuevamente.

En esta sección se mostró un algoritmo de una aplicación SPMD que tiene integrada una política de distribución cíclica. Para asegurar que cualquier implementación del algoritmo cumpla con un conjunto de propiedades que garanticen su buen funcionamiento. En la siguiente sección proponemos y desarrollamos su respectivo modelo en *promela* haciendo ciertas abstracciones para luego verificarlo con *Spin*.

4.3. Modelo Promela de la estructura básica propuesta

Para verificar la estructura básica de integración de la política de distribución cíclica se modelaron en *promela* los algoritmos vistos en la sección anterior. En el proceso de modelado se tomaron en cuenta aspectos como la topología de anillo de los procesos, la inserción y obtención de datos de la lista, la generación dinámica de datos, el procesamiento de datos, etc.

Comenzamos modelando el algoritmo básico mostrado en el código de la Figura 4.2, en el cual se encuentra la política de distribución integrada en la aplicación SPMD. El modelo *promela* para este algoritmo se muestra en el código de la Figura 4.7. Este código es ejecutado por un conjunto de N procesos lo cual es representado por `active[N]` (línea 1). Cuando se inicia la ejecución de este código, a cada proceso generado se le asigna automáticamente un identificador llamado `_pid`. Este identificador es diferente para cada proceso y $0 \leq$ `_pid` $<$ N.

```
1   active[N] proctype Proceso_Aplicacion_conDC(){
2   bool Datos_en_transferencia=false;
3   bool Peticion_enviada=false;
4   byte Terminacion=0;
5   byte L=CERO;/*LA LISTA SE MODELA USANDO UNA VARIABLE BYTE LLAMADA L*/
6   Generacion_De_Datos_Locales(L);
7   do
8   ::Terminacion < PROCESOS ->
9     Politica_De_Distribucion(Lista,Peticion_enviada,Terminacion,
      Datos_en_transferencia);
10     if
11     ::L != CERO ->/*SI L ES ≠ CERO SE MODELA QUE HAY DATOS */
12       Procesando(L),
13     ::else ->
14       skip;
15     fi;
16  ::Terminacion==PROCESOS ->
17     break;
18  od;
19  }
```

Figura 4.7: Modelo *promela* de la aplicación SPMD para la integración de una política de distribución en un programa SPMD (Figura4.2).

Las líneas 2-4 indican inicializaciones indispensables para el funcionamiento del algoritmo de distribución cíclico. La lista es modelada por medio de una variable de tipo `byte` llamada `L` (línea

```
1    inline Generacion_De_Datos_Locales(L) {
2    do /*ASIGNAMOS UN VALOR NO DETERMINÍSTICAMENTE A L */
3    ::true ->
4      L=CERO;/*SE MODELA QUE L INICIA SIN DATOS*/
5      break;
6    ::true ->
7      L=UNO;/*SE MODELA QUE L INICIA CON UN DATO*/
8      break;
9    ::true ->
10     L=MAYOR_QUE_UNO; /*SE MODELA QUE L INICIA CON MÁS DE UN DATO. */
11     break;
12   od;
13   }
```

Figura 4.8: Modelo para la generación de datos locales.

5). L puede tener los valores: CERO para modelar la lista vacía, UNO para modelar que la lista tiene un dato y MAYOR_QUE_UNO para modelar que la lista tiene más de un dato. La generación de datos locales (Línea 6) se modela asignándole no determinísticamente el valor CERO, UNO o MAYOR_QUE_UNO a L. Esta generación de datos se muestra en el código de la Figura 4.8

Regresando al código de la Figura 4.7, el ciclo do-od (líneas 7 y 18) modelan el ciclo while donde se realiza el procesamiento y distribución de datos. La política de distribución es modelada por el procedimiento Politica_De_Distribucion (línea 9). Este procedimiento será explicado más adelante. El procesamiento de datos se modela ejecutando el procedimiento Procesando (línea 12). Este procedimiento se encuentra desglosado en el código de la Figura 4.9.

Como se observa en el código de la Figura 4.9 si L es:

- UNO antes de modelar el procesamiento (líneas 3-14), después del procesamiento L puede tomar los valores CERO (para modelar que se procesó el último dato y este no generó más datos), UNO (para modelar que se procesó un dato y éste generó uno nuevo) o MAYOR_QUE_UNO (para modelar que se proceso el dato y éste generó más de un dato).

- MAYOR_QUE_UNO (líneas 15-23) los posibles valores que puede tomar son UNO (para modelar que la lista sólo tenía 2 datos y después del procesamiento sólo se quedó con un dato) y MAYOR_QUE_UNO (para modelar que se procesó el dato y éste generó más datos). En este último caso no se contempla que L pueda tener el valor CERO dado que no es posible que se tenga más de un dato en una lista y después de procesar uno de ellos la lista quede vacía.

```
1    inline Procesando(L){
2   if
3    ::L == UNO ->/*SE MODELA QUE HAY UNA DATO EN LA LISTA L */
4      do/*SI SE MODELA QUE L ES UNO PUEDEN SUCEDER ESTOS CASOS. */
5      ::true ->
6        L=CERO;/*SE PROCESO EL DATO Y NO GENERÓ MÁS DATOS */
7        break;
8      ::true ->
9        L=UNO;/*SE PROCESO EL DATO Y GENERÓ UN DATO */
10         break;
11     ::true ->
12       L=MAYOR_QUE_UNO;/*SE PROCESO EL DATO Y GENERÓ MÁS DE UN DATO */
13       break;
14     od;
15   ::else ->
16     do
17     ::true ->
18       L=UNO;/*SI ERAN DOS DATOS Y SE PROCESO UNO QUE NO GENERÓ DATOS.*/
19       break;
20     ::true ->
21       L=MAYOR_QUE_UNO;/*SE GENERA MÁS DE UN DATO*/
22       break;
23     od;
24   fi; }
```

Figura 4.9: Modelo para el procesamiento de datos.

Para modelar el algoritmo de distribución cíclico debemos tomar en cuenta que a diferencia de librerías de paso de mensajes tales como MPI o PVM, donde para enviar un mensaje sólo basta con especificar quién es el proceso receptor o emisor, en el lenguaje de modelado *promela* para comunicar a dos o más procesos debemos establecer:

1. El número de canales de comunicación.

2. Los canales por donde un proceso envía/recibe mensajes

Para establecer el número de canales se debe tomar en cuenta la topología de comunicación que manejan los procesos. En nuestro caso la política de distribución cíclica utiliza una topología anillo por lo que se utilizan N canales (donde N es el número de procesos en el anillo). Para formar esta topología y establecer los canales de envío y recepción los procesos utilizan su variable `_pid`. Si un proceso tiene identificador `_pid=X` recibirá mensajes por medio del canal `canal[X]` y enviará mensajes utilizando el canal `canal[(X+1) % N ]`. Un ejemplo de lo anterior se muestra en la Figura 4.10.

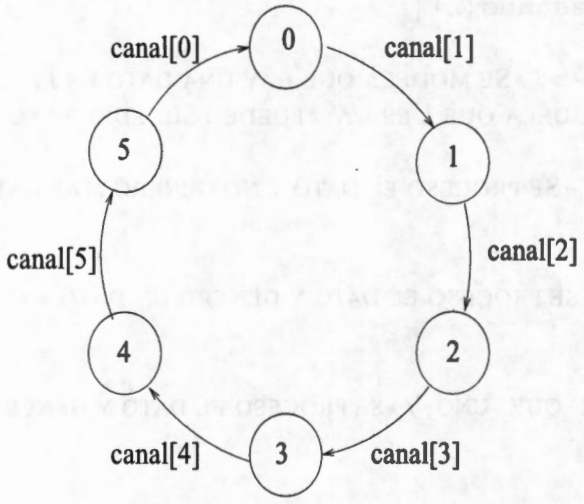


Figura 4.10: Topología de un anillo de procesos definido en *promela*.

La información que se envía por medio de estos canales es un mensaje que contiene dos campos:`Tipo` y `Ficha`. El campo `Tipo` es una variable `bit` que almacena el tipo de mensaje que se enviará. Si `tipo`= 0 el mensaje es de tipo `DAME_DATOS`, de lo contrario es de tipo `TERMINAR`. El campo `Ficha` se modela con una estructura de tipo que presenta 3 campos: `Propietario`, `Lista` y `Un_dato`. Estos campos siguen el mismo funcionamiento descrito en la sección anterior.

El procedimiento `Politica_De_Distribucion` invocado en código de la Figura 4.7 se muestra desglosado en el código de la Figura 4.11. Como se observa en este código, se declara una variable `bit` que almacena el tipo de mensaje (línea 2) y una variable `Fichas` para enviarla junto con el tipo de mensaje (línea 3).

Como se había mencionado, un proceso envía un mensaje de solicitud de tipo `DAME_DATOS` si su lista está vacía. En el código de la Figura 4.11 se modela esto verificando si `L` y `Peticion_enviada` son `CERO` y `false` respectivamente (línea 5). Posteriormente se envía un mensaje `DAME_DATOS` mediante el canal `canal[(_pid+1)%PROCESOS]` (línea 11).

Para modelar la función que verifica si ha llegado un mensaje se hace uso de la función `len` –la función `len` es propia de *promela* – (línea 14). Esta función regresa el número de mensajes en el canal. Si este valor es > 0 hay un mensaje en el canal por lo que se procede a recibirlo leyendo del canal `canal[_pid]` (línea 15). Después se procede a identificar el mensaje (línea 16-32). Si el mensaje es `DAME_DATOS` (línea 17) se ejecuta el procedimiento `Llega_Respuesta` o `Responder` (dependiendo el propietario de la `Ficha`). Pero si el mensaje es de tipo `TERMINAR` (línea 26) se incrementa la variable `Terminación` o se envía este mensaje al proceso sucesor (líneas 27-31)

```
1    inline Politica_De_Distribucion(Lista,Datos_en_transferencia,
Peticion_enviada,Terminacion){
2    bit Tipo;
3    Fichas Ficha;
4    if/*SI ENTRA AQUÍ ES PARA UNA PETICIÓN DE DATOS*/
5    ::Lista==CERO && Peticion_enviada==false ->
6     Peticion_enviada=true;
7     Tipo=DAME_DATOS;
8     Ficha.Un_dato=false;
9     Ficha.Propietario=_pid;
10     Ficha.Lista=CERO;/*SE ENVÍA UN MENSAJE AL PROCESO SUCESOR */
11     canal[(_pid+1)%PROCESOS]!Tipo,Ficha;
12    fi;
13   if/*SE REVISA SI HAY MENSAJES POR RECIBIR */
14   :: len(canal[_pid]) > 0 ->
15     canal[_pid]?Tipo,Ficha;
16     if/*SE REVISA EL TIPO DE MENSAJE */
17     ::Tipo==DAME_DATOS ->
20       if
21       ::Ficha.Propietario==_pid ->
22        Llega_Respuesta(Lista,Peticion_enviada,Datos_en_transferencia,
Ficha);
23       ::else ->
24         Responder(Lista,Datos_en_transferencia,Peticion_enviada,
          Ficha);
25        fi;
26     ::Tipo==TERMINAR ->
27       Terminacion++;
28       if
29       ::Ficha.Propietario!=_pid ->
30         canal[(_pid+1)%PROCESOS]!Tipo,Ficha;
31       fi;
32     fi;
33   fi;
34   }
```

Figura 4.11: Modelo para la política de distribución del código de la Figura 4.3.

El procedimiento `Responder` del código de la Figura 4.7 se muestra desglosado en el código de la Figura 4.12.

Como se observa en el código de la Figura 4.12 cuando un proceso atiende la llegada de una `Ficha` que no es la propia, revisa si `Ficha.Lista` es mayor que CERO (línea 4), lo cual modelaría que esta `Ficha` ya trae datos insertados por algún otro proceso. Si este fuera el caso y el proceso receptor había enviado un mensaje de tipo`DAME_DATOS` entonces pone el valor de su variable `Datos_en_Transferencia` a `true` (línea 7). De lo contrario, si `Ficha.Lista` es CERO (línea 9) el proceso revisa si su lista local `L` tiene el valor de UNO o MAYOR_QUE_UNO. Si `L` es UNO, como se observó en el código de la Figura 4.4, no se pueden transferir datos y se reenvía el mensaje al

```
1    inlineResponder(Lista,Datos_en_Transferencia,Peticion_enviada,
     Ficha){
2    bit Tipo;
3    if
4    :: Ficha.Lista > CERO ->
5      if
6      :: Peticion_enviada == true ->
7        Datos_en_Transferencia=true;
8      fi;
9    ::else ->
10     Tipo=DAME_DATOS;
11     if/*SE MODELA QUE LA LISTA TIENE UN DATO */
12     ::Lista==UNO ->
13       Ficha.Un_dato = true;
14       Ficha.Lista=CERO;
15     ::Lista==MAYOR_QUE_UNO ->
16       do/*CASOS POSIBLES SI SE MODELA QUE HAY MÁS DE UN DATO*/
17       ::true ->/*ENVÍO UN DATO Y ME QUEDO CON UN DATO*/
18         Lista=UNO;
19         Ficha.Un_dato=false;
20         Ficha.Lista=UNO;
21         break;
22       ::true ->/*ENVÍO MÁS DE UN DATO Y ME QUEDO CON MÁS DE UN DATO*/
23         Lista=MAYOR_QUE_UNO;
24         Ficha.Un_dato=false;
25         Ficha.Lista=MAYOR_QUE_UNO;
26         break;
27       ::true ->/*ENVÍO UN DATO Y ME QUEDO CON MÁS DE UN DATO (LA LISTA TIENE 3
DATOS)*/
28         Lista=MAYOR_QUE_UNO;
29         Ficha.Un_dato=false;
30         Ficha.Lista=UNO;
31         break;
32       od;
33     fi; fi;
34   canal[(_pid+1)%PROCESOS]!Tipo,Ficha;}
```

Figura 4.12: Modelo del procedimiento `Responder` del código de la Figura 4.4.

proceso sucesor solamente cambiando el valor de `Ficha.Un_dato` a `true` (líneas 13-14). Si `L` es `MAYOR_QUE_UNO` (líneas 16-32) el proceso receptor si puede transferir parte de sus datos locales al proceso solicitante, entonces asigna a `Ficha.Lista` el valor de `UNO` o `MAYOR_QUE_UNO` para modelar que da parte de sus datos al proceso `Ficha.Propietario`. Debido a esto el proceso cambia el valor de `L` (dado que al modelar que dio parte de sus datos a `Ficha.Propietario` también se debe modelar la disminución de sus datos en su lista `L`) a `UNO` o lo mantiene `MAYOR_QUE_UNO`. Lo anterior es admisible debido a que en una aplicación real si el proceso tuviera dos datos podría ceder uno, lo cual origina que le quede un dato, o bien si tuviera más de dos datos, al ceder la mitad de ellos se quedaría con dos o más datos.

```
1  inline Llega_Respuesta(Lista,Peticion_enviada,Datos_en_transferencia,
   Ficha){
2  bit mensaje;
3  if
4  ::(Ficha.Lista > CERO) ->
5    Lista=Ficha.Lista;/*SE CAMBIA EL VALOR DE L PARA MODELAR QUE ARRIBARON DA-
TOS.*/
6    Peticion_enviada=false
7  :: else -> skip;
8    if
9    ::((Datos_en_transferencia ==true)|| (Ficha.Un_dato==true))->
10     Peticion_enviada=false;
11     Datos_en_transferencia=false;
12     mensaje=DAME_DATOS;
13     Ficha.Un_dato=false;
14     Ficha.Propietario=_pid;
15     Ficha.Lista=CERO;
16     canal[(_pid+1)%PROCESOS]!mensaje,Ficha
17   ::else ->
18     Ficha.Un_dato=false;
19     Ficha.Propietario=_pid;
20     Ficha.Lista=CERO;
21     mensaje=TERMINAR;
22     canal[(_pid+1)%PROCESOS]!mensaje,Ficha
23   fi;
24 fi
25 }
```

Figura 4.13: Modelo *promela* para el procedimiento `Llega_Respuesta` del código de la Figura 4.11.

Por último en el código de la Figura 4.13 se presenta el modelo *promela* que representa el procedimiento `Llega_Respuesta` visto en el código de la Figura 4.11. En el código de la Figura 4.13 se observa que si al recibir la `Ficha` respuesta de una solicitud `DAME_DATOS` el valor de `Ficha.Lista` es mayor que `CERO`, el proceso receptor asigna el valor de `Ficha.Lista` a su variable `L` para modelar que recibió datos (Línea 5). De lo contrario si `Ficha.Lista` == `CERO` puede enviar un mensaje `DAME_DATOS` para una nueva petición de datos (líneas 10-16) en ca-

so de haber posibilidades de creación dinámica de datos (`Datos_en_transferencia=true` o `Ficha.Un_dato=true`). Cuando no hay posibilidad de creación dinámica de datos, se envía un mensaje `TERMINAR` (líneas 18-22).

4.4. Verificación

Para la verificación de nuestro modelo tomamos en cuenta que tenemos modelada una aplicación SPMD de procesamiento de datos y un algoritmo de distribución en un mismo proceso (debido a la integración). Aún así podemos dividir las propiedades a verificar en dos partes: las referentes a la política de distribución y las referentes a la integración de esta política dentro de la aplicación. Aunque la verificación se divide en dos partes, toda propiedad se verifica sobre todo el sistema.

En las siguientes secciones se proponen las propiedades principales que se deben cumplir en cada parte del sistema.

4.4.1. Verificación de la política de distribución

En esta subsección proponemos tres propiedades a verificar. Estas propiedades se enfocan en el estado de los procesos cuando éstos toman acciones debido a la llegada de una petición de datos o a la petición de terminación. A continuación se mencionan estas propiedades.

Propiedad A: No hay terminación prematura del distribuidor de datos

La idea de verificar esta propiedad es asegurarnos de que el protocolo de terminación no sea iniciado cuando haya uno (o más) datos procesándose en uno (o más nodos) y que éste (os) generen a su vez mas datos. Al asegurarnos de que esto no ocurre estaríamos evitando la terminación anticipada del distribuidor .

La fórmula es:

$$\mathbf{A}= \Box (\mathit{Terminar}_i \rightarrow ((\Box L_1 == \mathit{CERO} \wedge \Box L_2 == \mathit{CERO} \wedge ... \wedge \Box L_n == \mathit{CERO})))$$

en donde:

$\mathit{Terminar}_i$: Un proceso$_i$ envía un mensaje `TERMINAR`.
L_i== CERO: La Lista L_i del proceso$_i$ es `CERO`.

El proceso *i* solamente puede enviar su mensaje de terminación si las listas locales de los demás procesos están vacías. Esto es, no hay más datos en el sistema por lo cual el proceso de terminación puede comenzar.

La Figura 4.14 muestra el resultado de la verificación de esta propiedad con 4 procesos interactuando. Se observa en la parte superior izquierda de esta figura que el resultado de la verificación es valido.

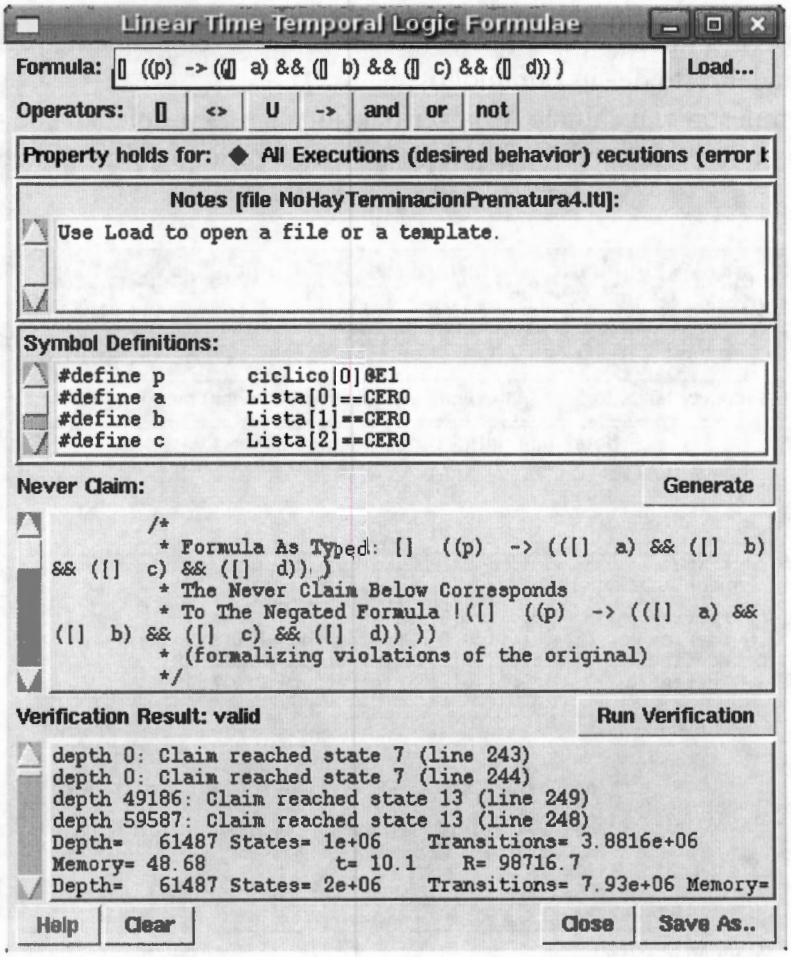


Figura 4.14: Resultado de la verificación de la propiedad A con 4 procesos.

Propiedad B: Si un proceso inicia el protocolo de terminación (enviando un mensaje TERMINAR), eventualmente cada proceso en el sistema iniciará el protocolo de terminación y recibirá N mensajes TERMINAR

La idea de verificar esta propiedad es garantizar que durante la fase de terminación del protocolo todos los procesos estén de acuerdo en terminar.

La fórmula es:

$$\mathbf{B} = \Box(Terminar_i \rightarrow (\diamond NumMsgTerminar_1 == \mathbb{N} \wedge \diamond NumMsgTerminar_2 == \mathbb{N} \wedge \ldots \wedge \diamond NumMsgTerminar_n == \mathbb{N}))$$

en donde:

$Terminar_i$: Un proceso$_i$ envía un mensaje TERMINAR.

$NumMsgTerminar_i == N$: La cantidad de mensajes recibidos de tipo TERMINAR en el proceso$_i$ es N.

Con esta propiedad se garantiza que si un proceso$_i$ inicia el protocolo de terminación, eventual-

mente todos los procesos iniciarán este protocolo por lo que eventualmente recibirán la confirmación de terminación del resto de sus homólogos.

La Figura 4.15 muestra el resultado de la verificación de esta propiedad con 4 procesos interactuando. Se observa en la parte superior izquierda de esta figura que el resultado de la verificación es valido.

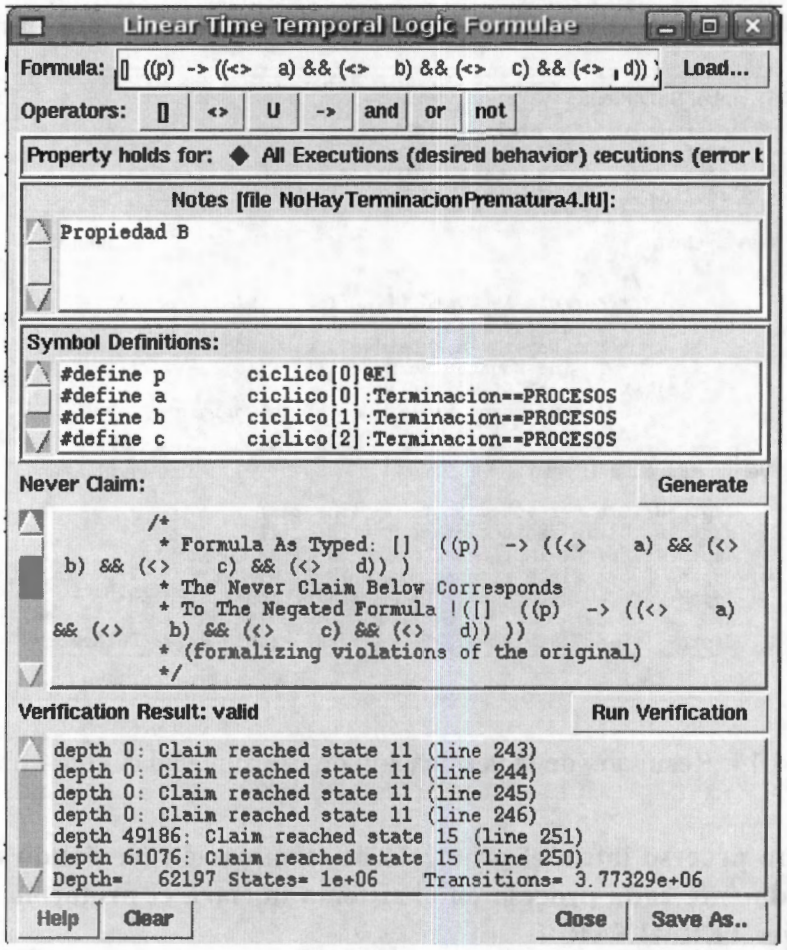


Figura 4.15: Resultado de la verificación de la propiedad B con 4 procesos.

Propiedad C: No hay pérdida de mensajes

El objetivo de esta propiedad es garantizar que todo mensaje es transmitido a su destino correcto.

La fórmula para representar esta propiedad es la siguiente:

$$\mathbf{C}=\Box((¡Terminar_i \rightarrow \Diamond ?Terminar_i) \wedge (¡DameDatos_i \rightarrow \Diamond ?DameDatos_i))$$

en donde:

$¡Termina_i$: Un proceso$_i$ envía su mensaje `TERMINAR`.
$?Terminar_i$: El proceso$_i$ recibe su mensaje `TERMINAR` que previamente envió el mismo.

¡DameDatos$_i$: Un proceso$_i$ envía su `Ficha` con un mensaje de tipo `DAME_DATOS`.

?DameDatos$_i$: El proceso proceso$_i$ recibe su `Ficha` después de viajar en el anillo con la respuesta a su mensaje `DAME_DATOS` que previamente envió.

Con la fórmula aseguramos que cualquier mensaje enviado eventualmente será recibido, asegurando así que ningún mensaje se pierde. En esta fórmula observamos que ¡Termina y ?Terminar se refieren al mismo proceso, esto se debe a que para que un proceso esté seguro que su mensaje fue recibido por todos los procesos debe él también recibir su mensaje(debido a la topología anillo). Esto sucede de igual manera para el mensaje `DAME_DATOS`.

La Figura 4.16 muestra el resultado de la verificación de esta propiedad con 4 procesos interactuando. Se observa en la parte superior izquierda de esta figura que el resultado de la verificación es valido.

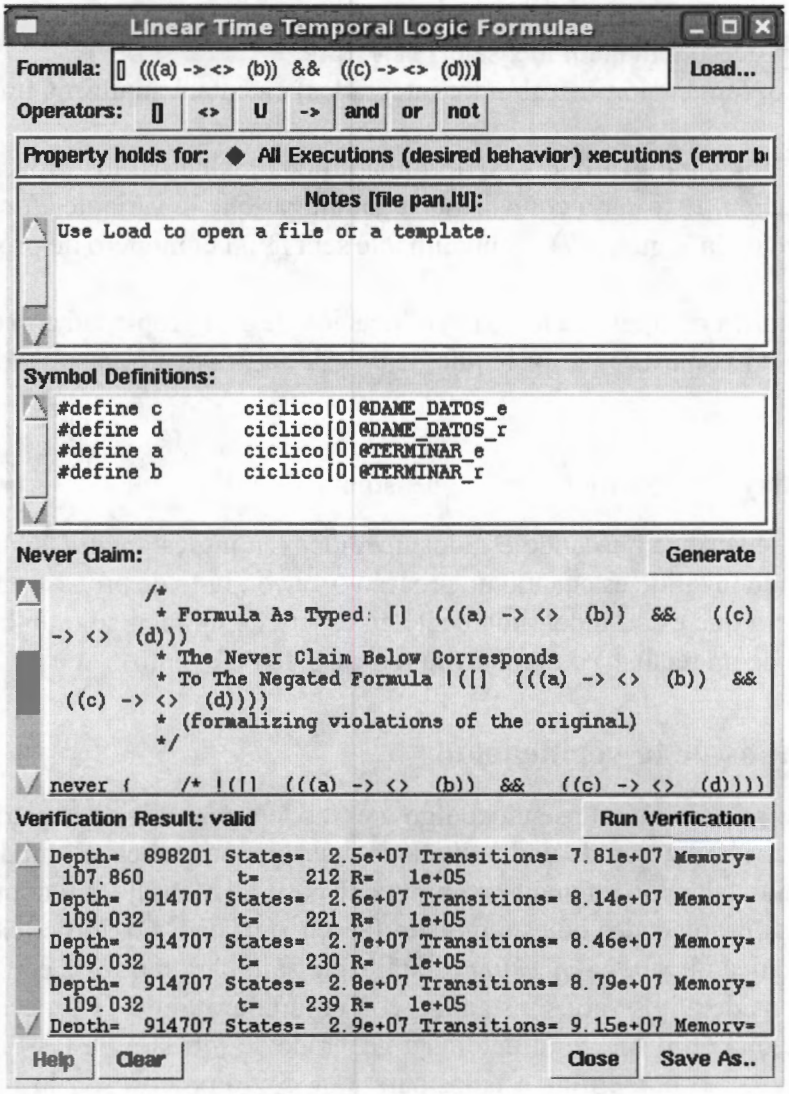


Figura 4.16: Resultado de la verificación de la propiedad C con 4 procesos.

4.4.2. Verificación de la integración de la política de distribución en la aplicación SPMD de procesamiento de datos

En seguida presentamos las propiedades que verificamos sobre el algoritmo de integración. Estas propiedades hacen referencia a todos el sistema (*Aplicacion* y *distribuidor*).

Propiedad D: El algoritmo terminará cuando no haya datos a procesar en el sistema

La idea es verificar si un proceso decidió terminar su ejecución, entonces en el futuro el resto de los procesos terminarán forzosamente.

La fórmula es:

$$\mathbf{D}=\Box(Terminar_i \rightarrow (\Diamond End_1 \wedge \Diamond End_2 \wedge ... \wedge \Diamond End_n))$$

en donde:

$Terminar_i$: Un proceso$_i$ envía un mensaje `TERMINAR`.
End_i: El proceso$_i$ (aplicación-distribuidor integrados) ejecutará su última línea de código.

Esta propiedad nos garantiza que si se terminan todos los datos, los procesos eventualmente terminarán su ejecución. Esto sería equivalente a garantizar que la variable *Terminacion* de cada proceso (ver Código de la Figura 4.7) eventualmente será igual el número de procesos que integran el sistema.

La Figura 4.17 muestra el resultado de la verificación de esta propiedad con 4 procesos interactuando. Se observa en la parte superior izquierda de esta figura que el resultado de la verificación es valido.

Propiedad E: No hay abrazos mortales en el sistema

Esta propiedad se verifica buscando estados inválidos (en inglés *invalid end-states*) en el sistema. En *Spin* un estado inválido es cuando un proceso o un conjunto de procesos no pueden terminar su ejecución debido a que están indefinidamente en espera de un recurso o un evento el cual no lo obtendrán (en caso de un recurso) o no ocurrirá (en caso de un evento).

4.4.3. Resultados de la verificación

En este capítulo se presentó el pseudocódigo y el modelo *promela* de la estructura de una aplicación SPMD con la integración de una política de distribución cíclica. El modelado se diseño de manera que su implementación en una herramienta de paso de mensajes tal como PVM o MPI sea sencilla. Además las propiedades de este modelo fueron verificadas usado el *model-checker Spin*. Debido a que la técnica de *model-checking* es aplicable sólo para modelos con estados finitos (de tamaño tratable) consideramos varias instancias con un número finito de procesos. Los resultados de la experimentación con respecto al autómata generado se muestra en las Figuras 4.18 y 4.19. La Figura 4.18 muestra el número de estados para 3, 4, 5 y 6 procesos, y la Figura 4.19 muestra el número de transiciones para el mismo número de procesos que en la Figura 4.18. Por medio de *Spin* garantizamos que las propiedades mencionadas anteriormente son válidas para cada instancia

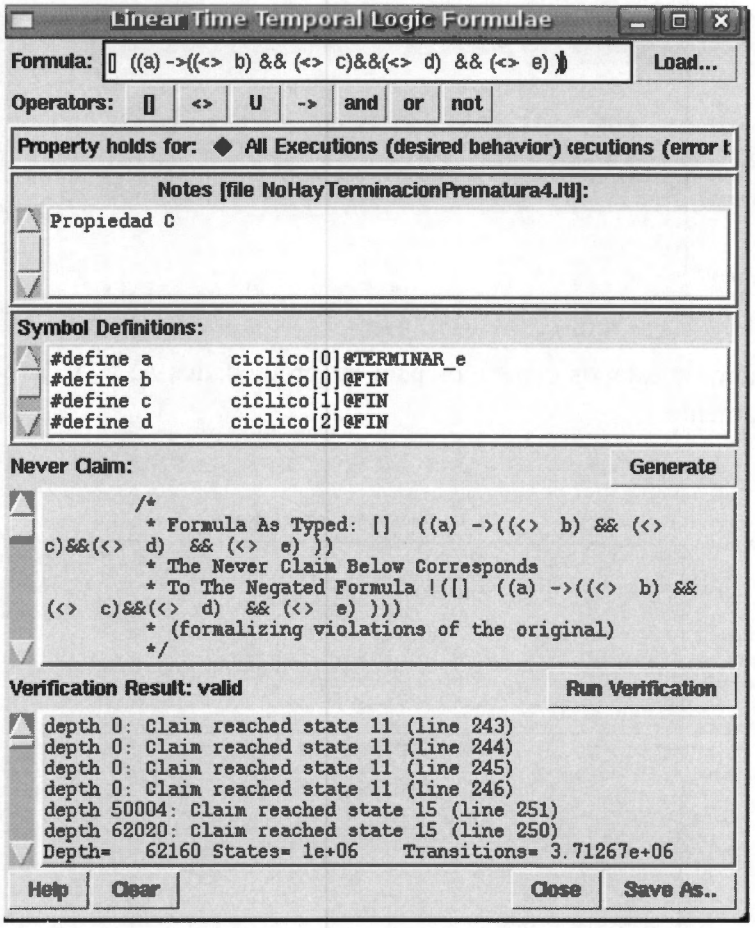


Figura 4.17: Resultado de la verificación de la propiedad D con 4 procesos.

considerada. Cabe mencionar que todos estos experimentos se realizaron sobre una PC con dos procesadores Xeon a 2.8 Ghz con 2 GB de memoria.

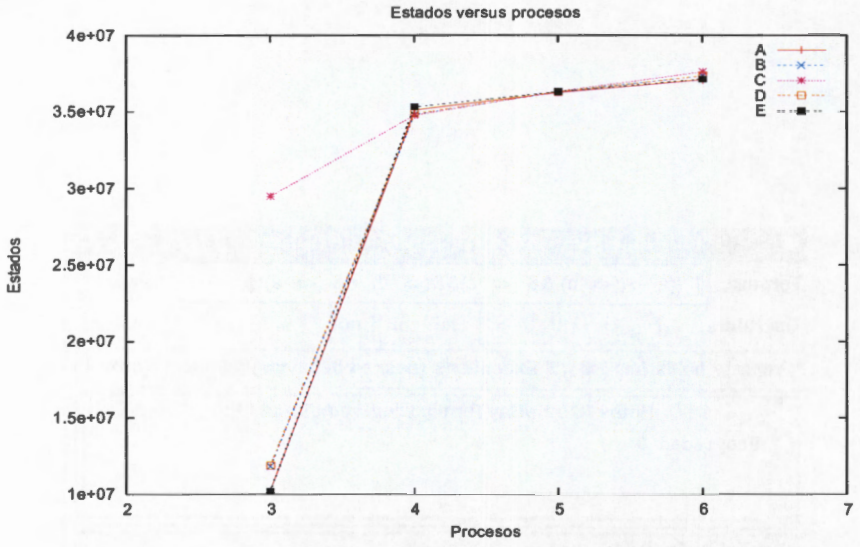


Figura 4.18: Número de estados generados para las propiedades A, B, C, D y E con 3, 4, 5, y 6 procesos respectivamente.

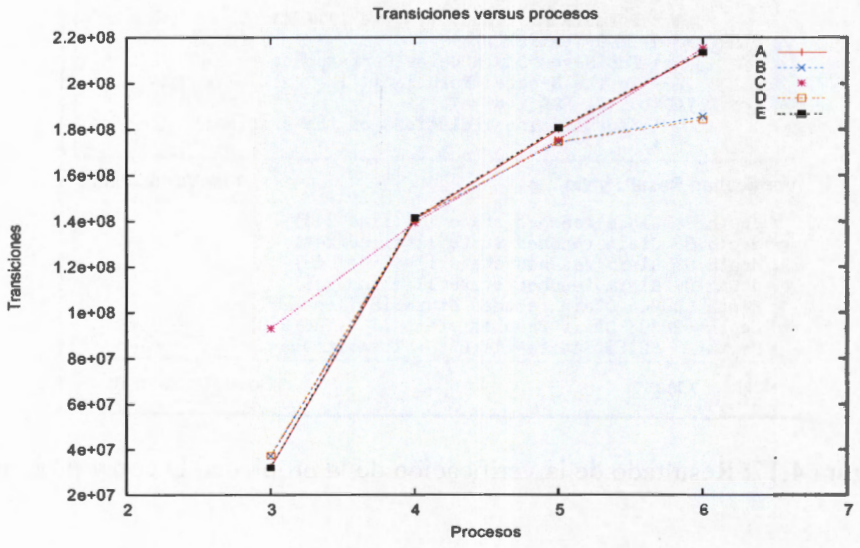


Figura 4.19: Número de transiciones generadas para las propiedades A, B, C, D y E con 3, 4, 5, y 6 procesos respectivamente.

4.4.4. Limitantes

En este algoritmo podemos encontrar diferentes limitantes con respecto a la distribución y a la verificación. Estas son:

- Con respecto a la verificación:
  - Complejidad en el código debido a la creación de un sólo proceso que integra el modelado de distribución y procesamiento de datos.
- Con respecto a la distribución:
  - Dependencia en la granularidad para permitir distribución de datos. La aplicación repercute en este aspecto al permitir la distribución únicamente al terminar el procesamiento de cada dato.
  - Sólo se transfiere la mitad de los datos del primer proceso que encuentra la Ficha vacía.
  - Debido a la topología cíclica del protocolo, en la cual entre más grande sea las Fichas tardan más en llegar.

En el siguiente capítulo se explica la verificación y el modelado de DLML (dado que en DLML se solucionan algunas de estas limitantes) y al igual que en este capítulo, se hace primeramente una descripción de cómo se generó el modelo para después mostrar las propiedades verificadas y los resultados obtenidos.

Capítulo 5

Verificación de DLML y nuevas propuestas

"Cada uno de nosotros es un modelo totalmente nuevo, parecido a otros modelos pero totalmente diferente."

–Autor desconocido

DLML puede resultar muy útil para distribuir sobre diferentes procesadores los datos de una aplicación, pero como todo programa que no fue sometido a un proceso de verificación, nada garantiza que cumpla con las especificaciones para lo que fue creado. En el capítulo anterior se propuso y se verificó una estructura básica en donde se integra un algoritmo de distribución en una aplicación de procesamiento de datos. Se identificaron cinco propiedades que un sistema de distribución de datos debe de cumplir para garantizar un buen funcionamiento independientemente del algoritmo de distribución usado.

En este capítulo proponemos un modelo *promela* para la implementación original de DLML y verificamos el conjunto de propiedades sugeridas en el capítulo anterior. La estructura de procesamiento de datos usando DLML refina la estructura básica del capítulo anterior considerando esencialmente las operaciones de inserción y eliminación de datos. La principal diferencia que repercute en la verificación de DLML es que se tiene una nueva implementación del sistema, definiendo un proceso para la aplicación y un proceso para el distribuidor, incorporando así características de transparencia y asincronía. Otra diferencia es que en DLML el algoritmo de distribución usado cambia, se aplica una política basada en subasta.

Por último proponemos diferentes implementaciones de DLML de las cuales verificamos la que presentó mejor rendimiento después de haberla probado sobre dos aplicaciones reales.

5.1. Introducción

Como se mencionó en el capítulo 2 las primitivas de manipulación de DLML se encargan del manejo de los datos de la lista. De estas primitivas la principal es `DLML_Get` mientras que las otras se implementan de manera local.

En el código de la Figura 5.1 se muestra la estructura básica de una aplicación que utiliza DLML, observamos que esta estructura es equivalente a la estructura básica que se propuso en el capítulo anterior (Figura 4.2). La operación `Generacion_De_Datos_Locales` conserva el mis-

mo propósito para la generación local de datos. Dado que DLML promueve la programación casi secuencial con distribución transparente de datos no existe una llamada explícita a una `Politica_De_Distribucion`. La función `DLML_Get` implementa de manera transparente para el programador, todo un protocolo de búsqueda de datos remotos manejando implícitamente la condición de terminación (regresa falso). Si `DLML_Get` regresa verdadero, un dato es obtenido de la lista (lista no vacía) y se procede a su procesamiento (línea 4). La función `Procesando` permite que se inserten datos en la lista si se requiere (considerando creación dinámica de datos). Finalmente la línea 7 se dedica a la recopilación de resultados parciales como en la estructura básica del capítulo anterior.

```
1    Proceso_aplicacion_conDLML(_pid) {
2      Generacion_De_Datos_Locales(L,_pid);
3      Mientras(DLML_Get(&L,&item)) {
4          Procesando(&L,item,resultado_local);
5        }
6      Resultado_global(resultado_local,_pid);
7      Escribe("Fin de ejecución");
8    }
```

Figura 5.1: Estructura básica de un programa que usa DLML.

En las siguientes secciones se presentará el modelo *promela* para la verificación de la estructura básica de DLML.

5.2. Modelado en Promela de la estructura de DLML

En DLML se separa la actividad de distribución del proceso *Aplicacion* manteniendo transparente para el usuario la programación paralela y la distribución de datos. Al separar la tarea de distribución y la de aplicación en dos procesos se origina que para modelar al sistema se tengan que crear dos tipos de procesos *promela*: uno para los procesos *Aplicacion* y otro para los procesos distribuidor *DLML*. Lo anterior se muestra en el código de la Figura 5.2. Como se observa en este código se usa `proctype` para definir los procesos *Aplicacion* (línea 1) y para definir los procesos *DLML* (línea 6). Cada proceso *Aplicacion* maneja una lista local de datos llamada `L` (línea 2). Además de estos dos tipos de procesos se utiliza un proceso principal llamado `init` (línea 10). En el proceso `init` se crean `N` instancias de procesos *DLML* (línea 15) y `N` instancias de procesos *Aplicacion* (línea 16) atómicamente (líneas 13-22) por lo que todos los procesos (exceptuando el `init`) inician su ejecución al mismo tiempo. Además los procesos *DLML* y los procesos *Aplicacion* al momento de ser creados se les pasa como parámetro una variable `byte` llamada `id`. Esta variable funciona como identificador para los procesos. Lo anterior origina que un proceso *DLML* y un proceso *Aplicacion* tengan el mismo identificador. El proceso distribuidor *DLML* y *Aplicacion* que tienen el mismo `id` forman una pareja *DLML-Aplicacion* la cual denotaremos como $DLML_{id}$ y $Aplicacion_{id}$.

```
1    proctype Aplicacion_conDLML(byte
id) {
2    byte L.
3    .
4    .
5    }
6    proctype DLML(byte id) {
7    .
8    .
9    }
10    init {
11    int id=0;
12    atomic  {
13    do
14    :: id < N ->
15      run DLML(id);
16      run Aplicacion_conDLML(id);
17      id++;
18    :: id == N ->
19     break;
20   od;
21    }
22    }
```

Figura 5.2: Procesos utilizados para modelar la herramienta DLML.

En las siguientes secciones desglosamos los modelos del proceso *Aplicacion* (línea 1) y el proceso *DLML* (línea 6) del código de la Figura 5.2.

5.2.1. Modelo del proceso Aplicacion

El modelo *promela* que representa la estructura básica de DLML se muestra en el código de la Figura 5.3. De manera similar que en el modelo del algoritmo de distribución cíclico, se utilizó una variable de tipo `byte` llamada `L` (línea 2) para modelar la lista local de datos. La variable `L` puede tener los valores de las constates `CERO`, `UNO` y `MAYOR_QUE_UNO` que se mencionaron en el capítulo anterior. Al inicio `L` se inicializa con alguna de las constantes mencionadas usando el procedimiento `Generacion_de_Datos(L)` (línea 4). Después de la inicialización de `L` se hace una llamada al procedimiento `DLML_Get` el cual recibe como parámetro la lista `L` y una variable llamada `salida`, esta última representa el valor que regresa `DLML_Get` (esto debido a que en promela los procedimientos `inline` no regresan valor). Cabe mencionar que en el modelado del procedimiento `DLML_Get` no se maneja el parámetro `item` dado que `L` simula una lista considerando únicamente la cantidad de datos (`CERO`, `UNO` o `MAYOR_QUE_UNO`).

En la línea 8 se modela el procesamiento de datos. El modelado de procesamiento de datos es parecido al mostrado en el capítulo anterior, en el cual el proceso $Aplicacion_{id}$ revisa el valor de su variable `L` y en base a éste determina de manera no determinística su siguiente valor. En este modelo si L es igual a `UNO` el procedimiento `Procesando` selecciona no determinísticamente

entre los valores CERO (si no se generan más datos), UNO (para el caso en que se genera un dato) o MAYOR_QUE_UNO (para el caso en que se genera más de un dato). Pero si L tiene el valor MAYOR_QUE_UNO se selecciona no determinísticamente entre los valores UNO (para el caso en que había dos datos) o MAYOR_QUE_UNO (para el caso en que se genera más de un dato).

```
1    proctype Aplicacion_conDLML(byte id) {
2     byte L;
3     bool salida=false;
4      Generacion_de_Datos_Locales(L);
5      DLML_Get(L, salida);
6      do
7      ::Salida==false ->
8        Procesando(L);
9        DLML_Get(L, salida);
10     ::salida==true ->
11       break;
12     od;
13     }
```

Figura 5.3: Modelo *promela* para el proceso *Aplicacion* de la librería DLML (estructura básica).

La implementación de DLML_Get, como se mencionó en el capítulo 2, requiere que el proceso $Aplicacion_{id}$ se comunique con su proceso $DLML_{id}$ por medio de memoria compartida y paso de mensajes. La memoria compartida sirve únicamente para que el proceso $Aplicacion_{id}$ reciba peticiones de envío de su lista de datos del proceso $DLML_{id}$.

El código de la Figura 5.5 contiene el procedimiento DLML_Get. Como se observa en este código, el proceso $Aplicacion_{id}$ revisa el valor de flag[id] (línea 6) para ver si $DLML_{id}$ requiere la lista local de datos. La variable flag es un arreglo global almacenado en memoria compartida de un nodo, su tamaño se determina por el número de procesadores por nodo (Figura 5.4).

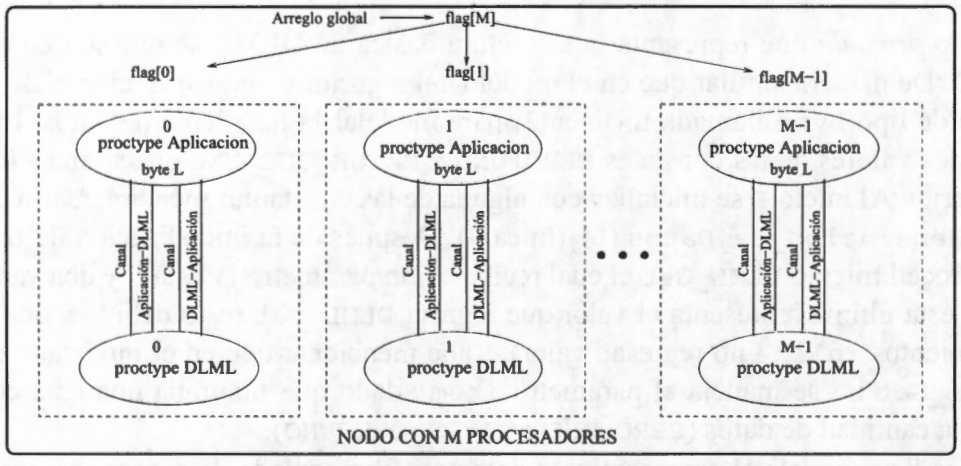


Figura 5.4: Arquitectura de comunicación DLML-Aplicacion.

Cabe mencionar que únicamente el proceso $Aplicacion_{id}$ y el proceso $DLML_{id}$ pueden hacer

uso de `flag[id]`, esto debido a que modelamos que estos dos procesos se encuentran en el mismo procesador por lo cual pueden accesar memoria compartida para comunicarse.

Si la variable `flag[id]` es igual a `true` el proceso $Aplicacion_{id}$ enviará un mensaje de tipo `LISTA_DE_DATOS` (línea 9) al proceso $DLML_{id}$ agregándole el valor de su variable `L` al campo `mensaje_envio.lista` (línea 13). Con lo anterior el proceso $Aplicacion_{id}$ modela que le envía sus datos a su proceso $DLML_{id}$ mediante un mensaje. Después de enviar la lista el proceso $Aplicacion_{id}$ se queda bloqueado en espera de un mensaje que le regrese su lista por medio de `canales_aplicacion[id]` (línea 16).

Si `flag[id]==false` pero `L==CERO` (línea 11) se envía al proceso *DLML id* un mensaje `LISTA_VACIA` por medio del canal `canales[id].canal[id]` (línea 10 y 11) solicitando datos remotos. Después $Aplicacion_{id}$ se queda bloqueado en espera de un mensaje de respuesta en el canal `canales_aplicacion[id]` (línea 16).

Cuando un proceso $Aplicacion_{id}$ se desbloquea (recibe su lista) procede a revisar si el campo `mensaje_recepcion.lista` es `CERO` (línea 18). Si éste es el caso, se cambia el valor de `salida` a `true` y de esta manera se indica que la Aplicacion id puede terminar. Si el campo `mensaje_recepcion.lista` es $\neq$ `CERO`, se asigna el valor `false` a `salida` (línea 21) y L se inicializa con el valor de `mensaje_recepcion.lista`(línea 22) modelando que recibió datos que posteriormente podrá procesar.

```
1   inline DLML_Get(byte L, bool salida){
2    mensaje mensaje_recepcion;
3    mensaje mensaje_envio;
4    mensaje_envio.destino=id;
5     if
6     :: (flag[id]==true) || (L ==CERO) ->
7       if
8       ::flag[id]==true ->
9        mensaje_envio.tipo_mensaje=LISTA_DE_DATOS;
10       ::else ->
11        mensaje_envio.tipo_mensaje=LISTA_VACIA;
12       fi;
13      mensaje_envio.lista=L;
14      L=CERO;
15      canales[id].canal[id]!mensaje_envio;
16      canales_aplicacion[id]?mensaje_recepcion;
17      if
18      ::mensaje_recepcion.lista==CERO ->
19        salida=true;
20      ::else ->
21        salida=false;
22        L=mensaje_recepcion.lista;
23      fi;
24    fi;
25  }
```

Figura 5.5: Modelo *promela* para el procedimiento `DLML_Get` de la librería DLML.

5.2.2. Modelo del proceso distribuidor DLML

Para modelar el proceso *DLML* consideramos el algoritmo de distribución de subasta visto en el Capítulo 2. Como se vió en el capítulo 2 este algoritmo de distribución inicia su ejecución cuando un proceso *Aplicacion* tiene su lista local vacía. Para ello se necesita que los procesos *DLML* definan una topología completa de comunicación, es decir, cada proceso *DLML* debe poder comunicarse con cualquiera de sus homólogos *DLML*. Un ejemplo de lo anterior se muestra en la Figura 5.6. En esta figura cada círculo representa un proceso *DLML*. Cada proceso *DLML* presenta un enlace a los demás *DLML* indicando que es posible la comunicación entre ellos.

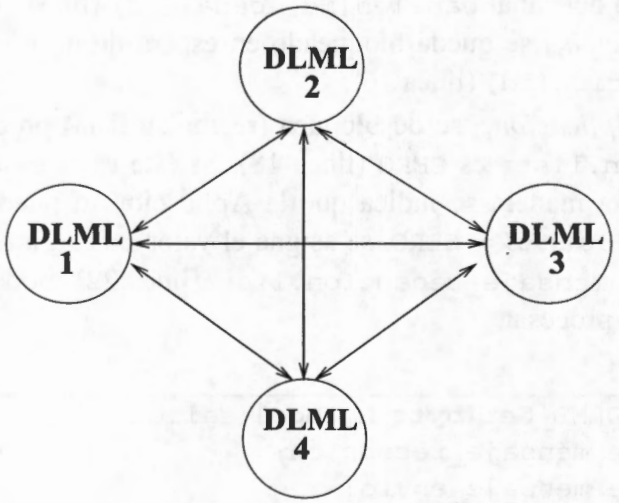


Figura 5.6: Topología completa necesaria para la comunicación de 4 procesos DLML.

Como se mencionó en el capítulo anterior, para generar el modelo *promela* de un conjunto de procesos que se comunican por mensajes debemos establecer el número de canales a utilizar e identificar los canales de envío y recepción de mensajes de cada proceso. En el caso de la librería DLML dado que se encuentra implementada en MPI, también se debe contemplar que la capacidad del `buffer` de envío de mensajes puede tener una capacidad de almacenamiento de uno o más mensajes.

Para modelar la topología de comunicación de los procesos *DLML* se utilizan `N` canales de comunicación (donde `N` es el número de procesos *DLML*). Estos canales se generan utilizando un arreglo de estructuras llamada `canales[N]` (esto dado que en *promela* no hay matrices). Al proceso $DLML_i$ donde `0 < i < N` se le asigna la estructura `Canales[i]`. Cada estructura `Canales[i]` mantiene un arreglo de `N` canales llamado `canal`. Por el canal `Canales[i].canal[i]` el proceso $DLML_i$ recibe mensajes de su proceso $Aplicacion_i$ y por `Canales[i].canal[x]` con `0< x` $\leq$`N` y `x` $\neq$ `i`, recibe mensajes del proceso $DLML_i$. Un ejemplo de lo anterior se muestra en la Figura 5.7.

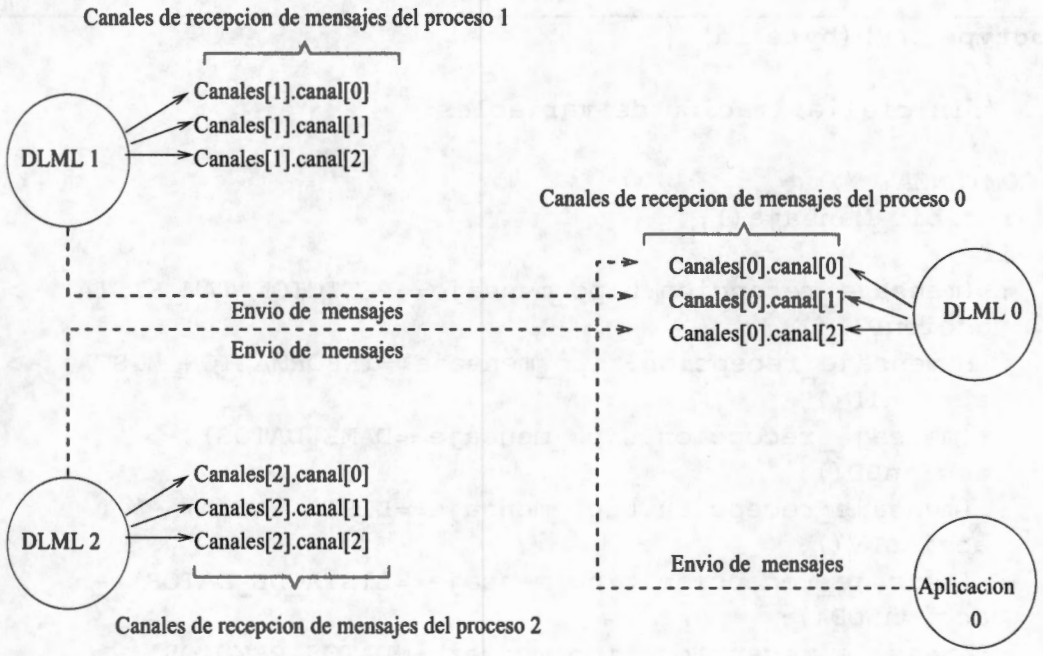


Figura 5.7: Modelado de los canales para 3 procesos distribuidores DLML.

En este ejemplo intervienen 3 procesos *DLML*. El proceso $DLML_i$ con $0 \leq i \leq 2$ utiliza `canales[i]`. Además se observa que si el proceso $DLML_1$ desea enviarle mensajes al proceso $DLML_0$ debe utilizar el canal `Canales[0].canal[1]`, si es el proceso $DLML_2$ quien desea enviarle un mensaje a *DLML 0* debe utilizar el canal `Canales[0].canal[2]` y finamente si es la $Aplicacion_0$ quien desea enviarle un mensaje a su distribuidor *DLML* debe utilizar el canal `Canales[0].canal[0]`. El canal `Canales_aplicación[0]` es el que usa $DLML_0$ para enviar un mensaje a $Aplicacion_0$.

La estructura que se envía por estos canales tiene 4 campos:

1. `lista`
2. `fuente`
3. `destino`
4. `tipo_mensaje`

El campo `lista` modela la lista local de datos que un proceso envía o recibe. El campo `fuente` contiene el `id` del proceso que generó del mensaje. El campo `destino` contiene el `id` del proceso a quien está dirigido el mensaje. Finalmente el campo `tipo_mensaje` contiene el tipo de mensaje (`PETICION_TAM_LISTA`, `INFORMACION_LISTA`, `DAME_DATOS`, `LISTA_DE_DATOS`, `etc`).

```
1 proctype DLML(byte id) {
2    . //
3    . //inicialialización de variables
4    . //
5    COMIENZA: //
6     recibir_Mensaje();
7     if
8     ::(mensaje_recepcion.tipo_mensaje==PETICION_TAM_LISTA) ->
9      accionPTL();
10     ::(mensaje_recepcion.tipo_mensaje==INFORMACION_LISTA) ->
11     accionIL();
12    ::(mensaje_recepcion.tipo_mensaje==DAME_DATOS) ->
13     accionDD();
14    ::(mensaje_recepcion.tipo_mensaje==LISTA_VACIA) ->
15     accionLV();
16    ::(mensaje_recepcion.tipo_mensaje==LISTA_DE_DATOS) ->
17     accionLDD();
18    ::(mensaje_recepcion.tipo_mensaje==DATOS_REMOTOS) ->
19     accionDR();
20    ::(mensaje_recepcion.tipo_mensaje==NO_HAY_DATOS) ->
21     accionNHD();
22    ::(mensaje_recepcion.tipo_mensaje==FIN) ->
23     contadorFin++;
24     if
25     :: contadorFin == (N) ->
26      mensaje_envio.lista=CERO;
25      canales_aplicacion[id]!mensaje_envio;
27      goto TERMINAR;
28     fi;
29    fi;
30    goto COMIENZA;
31   TERMINA:
32 }
```

Figura 5.8: Modelo *promela* del proceso DLML

La Figura 5.8 muestra la estructura de los procesos DLML. Como se muestra en esta Figura al inicio de la ejecución los procesos *DLML* están en una recepción no determinística de mensajes proveniente de algún otro proceso *DLML* o de su *Aplicacion* (línea 6). Esto es, el proceso $DLML_x$ con $0 \leq x < N$ puede de manera no determinística recibir un mensaje de alguno de los canales `canales[x].canal[i]` con `0` $\leq$ `i` $<$ `N`. El código de la Figura 5.9 muestra un ejemplo de la estructura de recepción no determinística donde `N` es igual a tres.

```
1 inline recibir_Mensaje(byte id) {
2    do
3    ::canales[id].canal[0]?mensaje_recepcion ->
4    break;
5    ::canales[id].canal[1]?mensaje_recepcion ->
6    break;
7   ::canales[id].canal[2]?mensaje_recepcion ->
8    break;
9     od;
10 }
```

Figura 5.9: Modelo *promela* del procedimiento de recepción del proceso DLML considerando 3 procesos en el sistema.

Cuando un proceso *DLML* recibe un mensaje lo guarda en su variable `mensaje_recepcion` e identifica el tipo del mensaje recibido revisando el campo `mensaje_recepcion.tipo_mensaje`. Para ésto hace uso de una estructura `if` (línea 17) la cual funciona como un `switch` del lenguaje C donde los `case` corresponden a los diferentes tipos de mensajes que puede recibir un proceso *DLML* (líneas 8, 10, 12, 14, 16, 18, 20 y 22). Después de identificar el tipo de mensaje ejecuta la acción correspondiente al mismo (líneas 9, 11, 13, 15, 17, 19, 21 y 23-28). Cuando se termina, se queda en espera nuevamente de otro mensaje saltando hacia la etiqueta `COMIENZA` (línea 30).

Cuando un proceso *DLML* recibe un mensaje `PETICION_TAM_LISTA` se ejecuta el código de la Figura 5.10 donde se inicializa un `mensaje_envio` con la última cantidad de datos que el proceso *Aplicacion* tenía. Esta valor se encuentra almacenado en una variable llamada `info_carga` la cual al igual que `L` puede tener los valores `CERO`, `UNO`, `MAYOR_QUE_UNO`. Despuésde inicializar `mensaje_envio` lo envía por el canal `canales[mensaje_recepcion.fuente].canal[id]` (línea 5).

```
1 inline accionPTL() {//Recepción de un mensaje PETICION_TAM_LISTA
2    mensaje_envio.tipo_mensaje=INFORMACION_LISTA;
3    mensaje_envio.destino=mensaje_recepcion.fuente;
4    mensaje_envio.lista=info_carga;
5    canales[mensaje_recepcion.fuente].canal[id]!mensaje_envio;
6 }
```

Figura 5.10: Acción correspondiente a la recepción de un mensaje `PETICION_TAM_LISTA`.

El código de la Figura 5.11 se ejecuta cuando un proceso *DLML* recibe un mensaje de tipo `INFORMACION_LISTA` como respuesta a una solicitud previa de `PETICION_TAM_LISTA` . Aquí se incrementa un contador llamado `contadorINFO` (línea 2) y se almacena el valor del campo `mensaje_recepcion.lista` en `longitud_listas[mensaje_recepcion.fuente]` (línea 3). Si el contador `contadorINFO` es igual a `(N-1)` significa que el proceso ha recibido un mensaje `INFORMACION_LISTA` de cada *DLML* del sistema (línea 4), por lo que haciendo uso de la función `busca_id_maxima_longitud()` (línea 8) revisa los valores que le enviaron y obtiene en `max_longitud` la maxima cantidad de datos recibida y en `id_max_longitud` el identificador del proceso correspondiente. En este procedimiento dado que estamos usando sólo 3 posibles va-

lores para el tamaño de la lista (CERO, UNO y MAYOR_QUE_UNO), si más de un proceso envió un valor similar al que tiene max_datos se selecciona no determinísticamente uno de ellos.

Si después de ejecutar el procedimiento busca_id_maxima_longitud() sucede que el valor de id_max_longitud es -1 significa que todos los procesos $DLML_i$ con $i \neq id$ respondieron al mensaje PETICION_TAM_LISTA con mensajes mensaje_recepcion.lista == CERO por lo que ejecuta la función envia_mensajes_FIN_a_DLMLs() (línea 11). En este procedimiento se les envía un mensaje FIN a los procesos *DLML* indicándoles que todas las listas L están en CERO. Después incrementa la variable contadorFin (línea 12) indicándose así mismo que ha iniciado el protocolo de terminación. Si fuera el caso que contadorFin se vuelve N significa que todos los procesos $DLML_i$ habían enviado mensajes FIN, de manera que, el proceso $DLML_{id}$ es el último y le envía a su proceso $Aplicacion_{id}$ un mensaje sin datos y posteriormente termina con su ejecución.

Si fuera el caso que busca_id_maxima_longitud $\neq$ -1, el proceso *DLML* le envía un mensaje DAME_DATOS al proceso *DLML* con id igual a busca_id_maxima_longitud (líneas 21-22)

```
1 inline accionIL() {
2    contadorINFO++;
3    longitud_listas[mensaje_recepcion.fuente]=mensaje_recepcion.lista;
4    if ::(contadorINFO==(N-1))->
5      contadorINFO=0;
7      id_max_longitud=-1;
8      busca_id_maxima_longitud(max_longitud, id_max_longitud);
9      if
10      :: id_max_longitud==-1 ->
11        envia_mensajes_FIN_a_DLMLs();
12        contadorFin++;
13        if
14        :: contadorFin == (N) ->
15          mensaje_envio.lista=CERO;
16          canales_aplicacion[id]!mensaje_envio;
17          goto TERMINAR;
18        fi;
19      ::else ->
21        mensaje_envio.destino= id_max_longitud;
21        mensaje_envio.tipo_mensaje=DAME_DATOS;
22        canales[id_max_longitud].canal[id]!mensaje_envio;
23     fi; fi; }
```

Figura 5.11: Acción correspondiente al mensaje INFORMACION_LISTA.

Cuando un proceso $DLML_{id}$ recibe a petición DAME_DATOS, ejecuta el código de la Figura 5.12. En este código se cambia el valor de shm[id] (línea 2) a true para solicitar a $Aplicacion_{id}$ su lista y se guarda el identificador del proceso que hace la petición dentro de un arreglo llamado peticiones (línea3). Luego se incrementa el contador llamado num_peticiones para poder guardar una proxima petición (línea 4). Después de incrementar el contador num_peticiones revisa si él o algún otro proceso DLML ha iniciado el protocolo de terminación (contadorFin!=0).

Si este es el caso envía a los procesos *DLML* que le han pedido datos un mensaje de NO_HAY_DATOS ejecutando la función envia_NHD() (línea 7).

```
1    inline accionDD(){
2      shm[id]=true;
3      peticiones[num_peticiones]=mensaje_recepcion.fuente;
4      num_peticiones++;
5      if
6      ::contadorFin!=0 ->
7        envia_NHD();
8      fi; }
```

Figura 5.12: Acción correspondiente al mensaje DAME_DATOS.

Cuando un proceso *DLML* recibe un mensaje LISTA_VACIA ejecuta el código de la Figura 5.13. En este código se actualiza el valor de shm[id] a false (línea 2) para después ejecutar el procedimiento envia_NHD() (línea 3), enviándoles así un mensaje de NO_HAY_DATOS a los procesos *DLML* que le enviaron un mensaje de DAME_DATOS. Después se actualiza el valor de info_carga (línea 4) y se ejecuta el procedimiento envia_PTL() (línea 5). Este procedimiento se desglosa en el código de la Figura 5.14 en el cual se observa que el proceso *DLML* envían un mensaje PETCION_TAM_LISTA a cada uno de sus homólogos *DLML* (líneas 5-17) .

```
1    inline accionLV(){
2      shm[id]=false;
3      envia_NHD();
4      info_carga=CERO;
5      envia_PTL();}
```

Figura 5.13: Acción correspondiente al mensaje LISTA_VACIA.

```
1    inline envia_PTL() {
2    contador=0;
3    mensaje_envio.tipo_mensaje=PETICION_TAM_LISTA;
4    mensaje_envio.lista=CERO;
5    do
6    ::contador < N ->
7      mensaje_envio.destino=contador;
8      if
9      ::contador!=(id)->
10       canales[contador].canal[id]!mensaje_envio;
11     fi;
12     contador++;
13   ::contador == N ->
14     break;
15   od; }
```

Figura 5.14: Envío de mensajes PTL originando el inicio del protocolo de subasta.

Cuando un proceso *DLML* recibe un mensaje LISTA_DE_DATOS ejecuta el código de la Figura 5.15. Como se observa en este código, el proceso *DLML* primero almacena el valor contenido en mensaje_recepcion.lista que le envió su proceso *Aplicacion* en una variable llamada LL (línea 2). Esta variable modela el almacenamiento de los datos para la redistribución. Después se actualiza la variable info_carga y shm[id] (líneas 3-4) para actualizar los últimos datos que llegaron y para indicar que se atenderán las peticiones respectivamente.

```
1    inline accionLDD(){
2      LL=mensaje_recepcion.lista;
3      info_carga=mensaje_recepcion.lista;
4      shm[id]=0;
5      if
6      ::info_carga==MAYOR_QUE_UNO ->
7        do
8        ::true ->
9          mas_peticiones_que_datos();
10         break;
11       ::true ->
12         menos_peticiones_que_datos();
13         break;
14       od;
15     ::else ->
16       LL=UNO;
17     fi;
18     envia_NHD();
19     info_carga=LL;
20     LL=CERO;
21     mensaje_envio.lista=info_carga;
22     mensaje_envio.destino=id;
23     mensaje_envio.tipo_mensaje=DATOS;
24     canales_aplicacion[id]!mensaje_envio;
25   }
```

Figura 5.15: Acción correspondiente al mensaje LISTA_DE_DATOS.

Después de actualizar sus variables revisa si info_carga es MAYOR_QUE_UNO. Si éste es el caso se modelan dos posibilidades:

1. Hay mas peticiones que datos: En este caso se ejecuta el procedimiento con el nombre mas_peticiones_que_datos() (línea 9), en el cual se actualiza LL con UNO y se le envía a un subconjunto de procesos que realizaron una petición de datos un mensaje con el campo mensaje_envio.lista igual a UNO.

2. Hay menos o igual peticiones que datos: En este caso el procedimiento que se ejecuta es menos_peticiones_que_datos() (línea 12), en el cual el valor de LL se actualiza no determinísticamente con el valor UNO o MAYOR_QUE_UNO y a los procesos *DLML* que le hicieron una petición se les envía un mensaje con el campo mensaje_envio.lista con UNO o MAYOR_QUE_UNO.

Si hay peticiones de procesos *DLML* que no alcanzaron datos se les envía a estos procesos un mensaje de `NO_HAY_DATOS` ejecutando la función `envia_NHD()` (línea 18). En esta función si no hay peticiones no se hace nada. Después se actualizan las variables `info_carga` y `LL`. Por último le envía un mensaje DATOS a su proceso *Aplicacion*.

Cuando un proceso *DLML* recibe un mensaje `DATOS_REMOTOS` ejecuta el código de la Figura 5.16. En este código se actualiza la variable `info_carga` (línea 2) con el valor que le llegó en mensaje_envio.lista y le envía este valor a su proceso Aplicación (líneas 5).

```
1   inline accionDR(){
2     info_carga=mensaje_recepcion.lista;
3     mensaje_envio.lista=info_carga;
4     mensaje_envio.tipo_mensaje=DATOS;
5     canales_aplicacion[id]!mensaje_envio;
6   }
```

Figura 5.16: Acción correspondiente al mensaje `DATOS_REMOTOS`.

Por último, si el mensaje que recibe un proceso DLML es de tipo `NO_HAY_DATOS` se ejecuta el código de la Figura 5.17. En este código el proceso DLML inicia nuevamente el protocolo de subasta para lo cual ejecuta el procedimiento `envia_PTL()` (ver Figura 5.14).

```
1   inline accionNHD(){
2     envia_PTL();
3   }
```

Figura 5.17: Acción correspondiente al mensaje `NO_HAY_DATOS`.

En esta sección se mostró como fue modelado el distribuidor DLML, en las siguiente sección se describen un conjunto de propiedades que se verificaron sobre este modelo.

5.3. Verificación

Después de haber estudiado y modelado en *promela* el distribuidor DLML se procedió a su verificación. La verificación de este distribuidor nos ayuda a garantizar para ciertas instancias si el distribuidor cumple con propiedades que garanticen su buen funcionamiento. Las propiedades se verificaron contemplando canales con capacidad de almacenamiento de uno y más de un mensaje.

5.3.1. Propiedades a verificar y resultados de la verificación

En el capítulo anterior se identificó un conjunto de cinco propiedades que debe cumplir cualquier algoritmo de distribución de carga. En dicho capítulo sección 5.3 se describieron de manera informal cada una de esas propiedades de de manera detallada cada una de las fórmula LTL correspondientes (por lo tanto omitiremos su descripción en este capítulo). Dichas propiedades fueron validadas sobre un algoritmo de distribución cíclica. Ahora debemos verificar que el distribuidor

DLML también las cumpla para con ello garantizar su buen funcionamiento. Adicionalmente a estas propiedades se proponen dos nuevas referentes a la interacción entre procesos. Las propiedades son:

1. **Propiedad A**: Todo proceso solo recibirá los mensaje dirigidos hacia él.
2. **Propiedad B**: Todo proceso ejecutará la acción correspondiente a cada mensaje.
3. **Propiedad C**: No hay perdida de mensajes.
4. **Propiedad D**: No hay abrazos mortales en el sistema.
5. **Propiedad E** El algoritmo terminará cuando no haya datos a procesar en el sistema.
6. **Propiedad F** Si un proceso inicia el protocolo de terminación, eventualmente cada proceso en el sistema iniciará el protocolo de terminación y recibirá N mensajes FIN.
7. **Propiedad G**: No hay terminación prematura del distribuidor de datos.

Al verificar cada una de las propiedades se genera su propio modelo(autómata). Las propiedades se verificaron considerando sistemas de 4 a 15 procesos. A continuación se procede a mostrar de qué manera fueron verificadas las propiedades anteriores, así como los resultados que arrojaron la verificación de cada propiedad.

Propiedad A: Todo proceso sólo recibirá los mensajes dirigidos hacia él.

La idea de garantizar esta fórmula es garantizar que todo proceso sólo recibirán los mensajes que fueron enviados para él.

```
.
.
.
Canales[id].canal[x]?mensaje_recepcion;
assert(pid==mensaje_recepcion.destino)
.
.
.
```

Figura 5.18: Verificación usando `assert`

Como se mencionó anteriormente, todo mensaje tiene un campo llamado *fuente* y uno llamado *destino*. Cuando un proceso recibe un mensaje utiliza una instrucción *assert* (la cual nos sirve para hacer afirmaciones) para asegurar que el mensaje que recibió el proceso estaba dirigido hacia él. Lo anterior se realiza como se muestra en la Figura 5.18, cada que un proceso realice un recepción afirmara que el mensaje es para él. Esta propiedad es requerida en *promela* ya que en este lenguaje debemos establecer los canales de un proceso a otro.

Propiedad B: Todo proceso ejecutará la acción correspondiente a cada mensaje.

Esta propiedad se verifica usando afirmaciones. Con esta propiedad se garantiza que las acciones que realiza un proceso son coherentes con respecto a los mensajes que llegan, esto es, no es posible que se ejecute una acción que no corresponda al mensaje que llegó. Al igual que la propiedades anterior, esta propiedad resulto ser valida para las instancias verificadas.

Propiedad C: No hay pérdida de mensajes.

La idea al verificar esta propiedad es garantizar que todo mensaje será entregado. Esta fórmula fue expresa previamente en el capítulo anterior. Por tanto sólo mostramos los resultados de la verificación para 4 procesos *DLML* en la Figura 5.19.

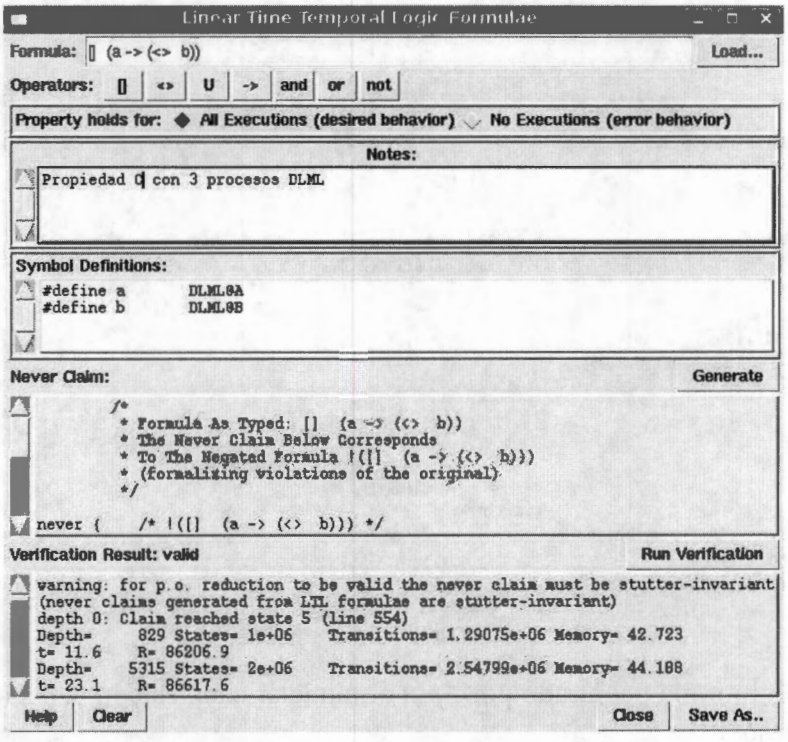


Figura 5.19: Resultado de la verificación de la propiedad C para DLML con 4 procesos

Cabe mencionar que dado que en DLML se cuenta con varios tipos de mensajes, se verifica que siempre que envíe un mensaje X eventualmente recibirá su contestación(es) del mensaje si es requerida.

Propiedad D: No hay abrazos mortales en el sistema

Esta propiedad se verifica buscando estados inválidos en el sistema. Como mencionamos en el capítulo anterior, en *Spin* un estado inválido es cuando un proceso o un conjunto de procesos no pueden terminar su ejecución dado que están en espera de un recurso o un evento de indefinidamente que cual no lo obtendrán (en caso de un recurso) ó no ocurrirá (en caso de un evento). Esta

propiedad cuando se verifica tomando en cuenta que los canales tienen una capacidad de almacenamiento de 1, *Spin* muestra que es posible un abrazo mortal (ver Figura 5.20). Este abrazo mortal que se presenta en una ejecución es muy raro que se presenta en un a ejecución con una aplicación real, ya que, *Spin* hace que un proceso aplicación procese todos los datos.

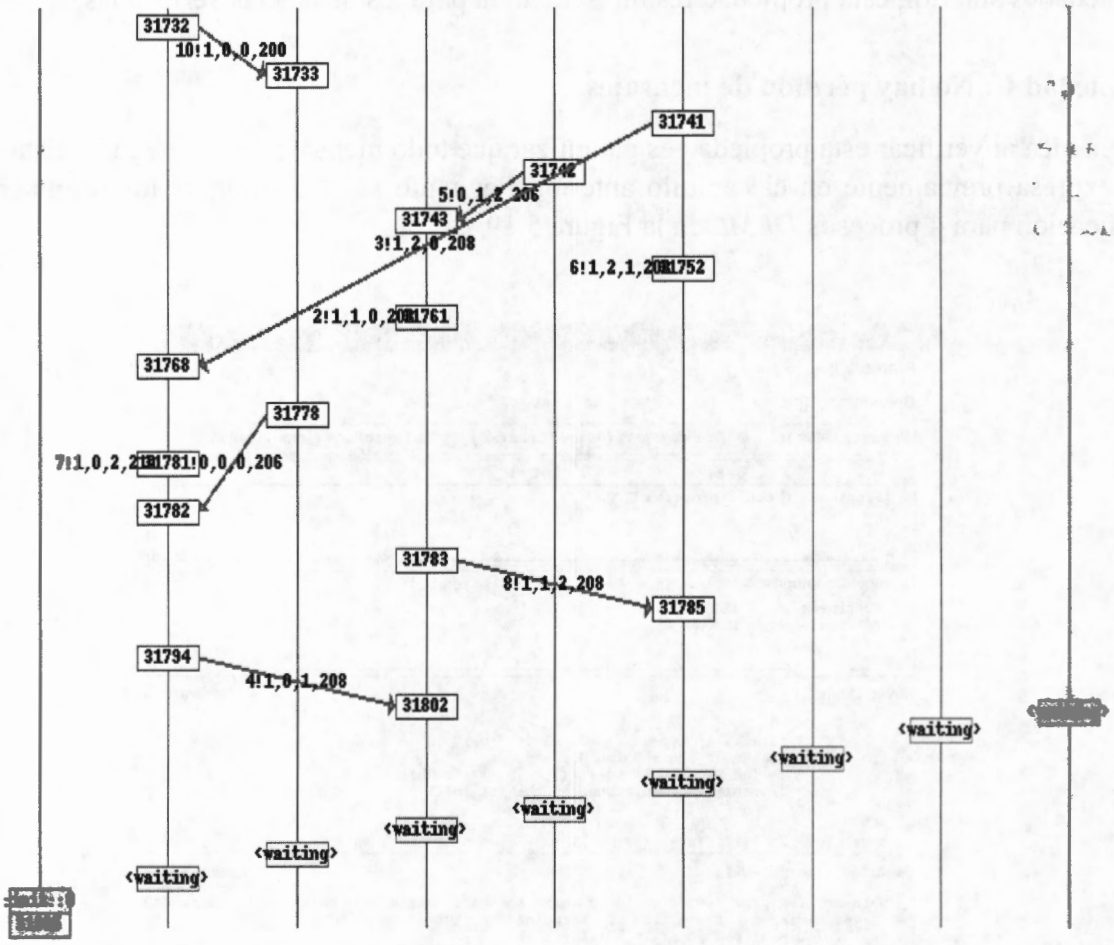


Figura 5.20: Abrazo mortal encontrado por *Spin* al modelar canales con capacidad de un mensaje (intervienen 4 procesos).

propiedad E: El algoritmo terminará cuando no haya datos a procesar en el sistema

La idea al garantizar esta propiedad es que si un proceso DLML inicio el protocolo de terminación, entonces eventualmente en el futuro todos los procesos terminaran. El resultado de la verificación de esta propiedad para 4 procesos DLML se muestra en la figura 5.21.

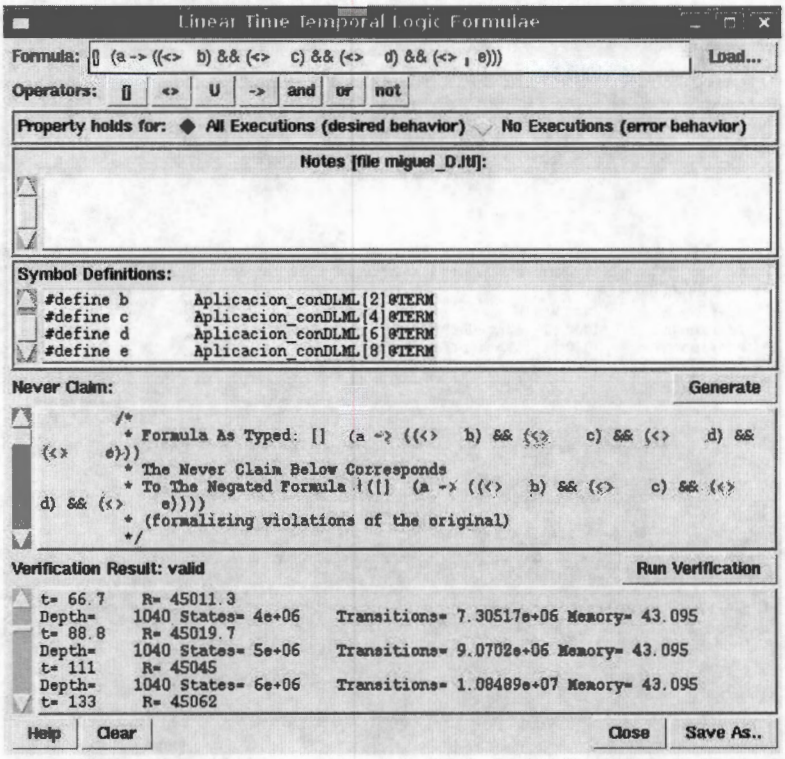


Figura 5.21: Resultado de la verificación de la propiedad D para DLML con 4 procesos.

Propiedad F: Si un proceso inicia el protocolo de terminación, eventualmente cada proceso en el sistema iniciará el protocolo de terminación y recibirá N mensajes FIN

La idea de verificar esta fórmula es garantizar cuando un proceso inicie el protocolo de terminación, todos los eventualmente se pondrán de acuerdo para terminar su ejecución. De igual que la propiedad anterior, esta fórmula fue expresa previamente en el capítulo anterior. Por tanto sólo mostramos los resultados de la verificación para 4 procesos *DLML* en la Figura 5.22

Propiedad G: No hay terminación prematura del distribuidor de datos.

Como se mencionó en el capítulo anterior, si un proceso no recibe datos por medio de una petición que previamente realizó esto no quiere decir necesariamente que en el sistema no haya datos. Esta propiedad nos ayuda a garantizar que el protocolo de terminación no sea iniciado cuando hay uno o más datos en el sistema. La Figura 5.23 muestra los resultados de la verificación para 4 procesos *DLML*.

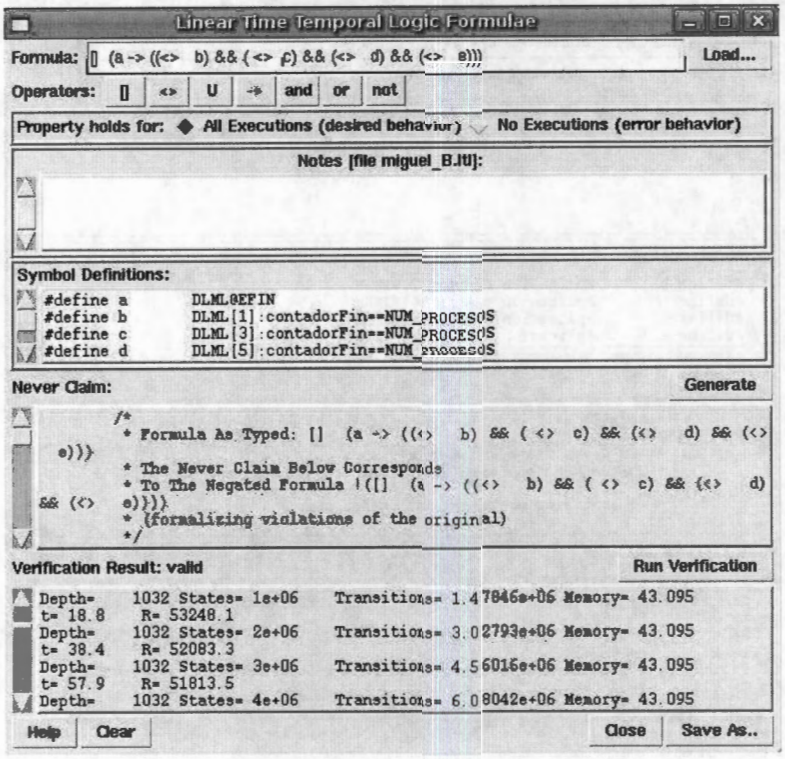


Figura 5.22: Resultado de la verificación de la propiedad B para DLML con 4 procesos.

Después de haber modelado y verificado la herramienta DLML, podemos darnos cuenta que el error que se presenta es debido a la configuración de la herramienta MPI, dado que para canales con capacidad de almacenamiento de más de un mensaje nunca se encontraron errores. En el siguiente capítulo damos algunas propuestas para solucionar lo anterior. Además proponemos una nueva propuesta de algoritmo de subasta para DLML, el la cual se reduce el número de procesos por nodo aprovechando que la mayoría de las computadores actuales están siendo desarrolladas con procesadores de más de un núcleo (multicore), y se elimina la dependencia de la granularidad de los datos para poder distribuirlos.

5.4. Propuestas de implementación de DLML

Después de realizar la verificación de DLML se propusieron y desarrollaron dos propuestas de implementación para ayudar a solucionar el " problema " encontrado y a disminuir el número de comunicaciones. Las propuestas hacen uso de la comunicación con memoria compartida para los procesadores de un nodo y con intercambio de mensajes entre nodos. En esta sección se explica cada una de las versiones, para más detalle en el apéndice A se hace uso de ejemplos que plantean posibles ejecuciones en cada implementación.

5.4.1. Propuesta 1: Manejo de memoria compartida

Esta primera propuesta muestra como se implementó el uso de memoria compartida para almacenar la lista de datos entre cada par *Aplicacion-Distribuidor*. En esta propuesta se plantearon dos

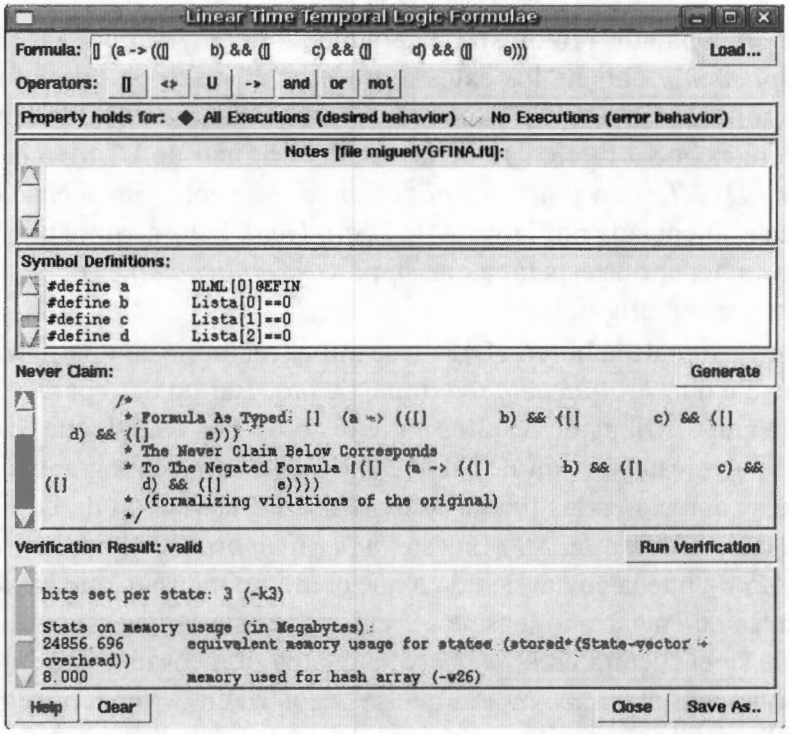


Figura 5.23: Resultado de la verificación de la propiedad A para DLML con 4 procesos.

versiones, una donde se utiliza memoria compartida entre procesos y otra entre hilos.

Uso de proceso DLML - proceso Aplicacion y memoria compartida entre ellos

En esta versión igual que en la versión original de DLML se cuenta con dos procesos por procesador: un proceso *Aplicacion* y un proceso *DLML*. Cada proceso tiene su identificador *id* con $0 \leq id \leq N$ (donde N es el número de procesos *DLML*) e *id* es distinto para cada par de procesos del mismo tipo (*DLML-DLML* o *Aplicación-Aplicación*). Esto origina que un proceso *DLML* y un proceso *Aplicacion* tendrán el mismo valor en su *id*. Los procesos *Aplicacion* y *DLML* que tienen el mismo valor en su identificador *id* se comunican por medio de cuatro variables compartidas, las cuales son:

- *Terminar*
- *Procesando*
- *Longitud_lista*
- *Lista_local*

La variable *Terminar* sirve para que un proceso *DLML* le comunique a su *Aplicacion* que ya no hay datos en el sistema por lo que puede proceder con la terminación de su ejecución. La variable *Procesando* sirve para que un proceso *Aplicacion* le comunique a su *DLML* si está procesando un dato(valor *true*) o, para que un proceso *DLML* le indique a su proceso *Aplicacion* si puede procesar

datos (debido a que obtuvo datos remotos). La variable *Longitud_lista* indica el tamaño de la lista donde el proceso *Aplicacion* obtiene los datos a procesar. El valor de esta última variable puede ser modificado por un proceso *DLML* cuando agrega u obtiene datos de la lista. Las variables anteriores surgen debido que a diferencia de la versión original de DLML, la lista de datos que comparte un proceso *DLML* y un proceso *Aplicacion* se encuentra almacenada en un espacio de memoria compartida referenciada por la variable *Lista_local*. Esto permite que un proceso *DLML* pueda obtener datos sin tener que esperar a que su proceso *Aplicacion* termine de procesar un dato (lo cual ocurría en la versión original).

Dado que en esta versión de la librería DLML se utiliza memoria compartida, un proceso *DLML* no tiene que esperarse a que su *Aplicación* termine de procesar un dato para que esta le envíe sus datos y así él realice una distribución de ellos entre los procesos *DLML* que le han hecho una petición. Lo anterior vuelve esta versión de DLML independiente de la granularidad de los datos. Además, aunque no se aprecia en las figuras anteriores en esta versión de DLML se mantiene un *buffer temporal* por cada proceso *DLML*. En este *buffer temporal* se almacenan aquellos mensajes que un proceso *DLML* no pueda enviar debido a que el último mensaje que envió no ha sido entregado, por lo que, antes de enviar un mensaje se revisa que este *buffer* este vacío, si no lo esta, se envía el mensaje que se encuentra en el *buffer* y se guarda el mensaje que se iba a enviar. Con lo anterior se busca solucionar el abrazo mortal que se encontró en la versión original de DLML debido al buffer de MPI cuando fue verificado suponiendo canales con capacidad de almacenamiento de un mensaje.

La principales desventajas de esta versión radica en:

- El uso de memoria compartida.

  Debido a que en su implementación se tiene que hacer una petición de memoria compartida, la cual en la gran mayoría de los sistemas Linux[1] no debe exceder el 50 % de la memoria del nodo por lo que la cantidad de datos que puede generar una *Aplicacion* tendría un cota que no necesariamente es la real del sistema.

- Cambios de contexto. Al usar un proceso DLML y Aplicación los procesos tiene que realizar constantemente cambios de contexto para ocupar el procesador.

Manejo de hilos

Esta versión es muy similar a la de procesos, sólo que en lugar de utilizar dos procesos por procesador se utiliza un proceso *DLML* con dos hilos: uno llamado hilo *Distribuidor* y otro hilo *Aplicacion*. La estructura de este proceso se muestra en la Figura 5.24. Como se observa en esta figura un proceso *DLML* utiliza cuatro variables globales: *Terminar, Procesando, Longitud_lista* y *Lista_local*. Estas variables tienen el mismo nombre de las variables compartidas que utilizaba el proceso *DLML* de la versión anterior. Esto se debe a que tienen una semántica similar a ellas, pero ahora sirven para comunicar al hilo *distribuidor* y al hilo *Aplicacion*. Estas variables igual que en la versión anterior se necesitan acceder en exclusión mutua.

[1]Lo anterior fue comprobado en varias varios de sistemas operativos tipo unix tales como SuSe 10; Fedora C1, ..., C6, Ubuntu, Debian, Centos 4.0 y 5.0, etc. entre otras versiones de Linux.

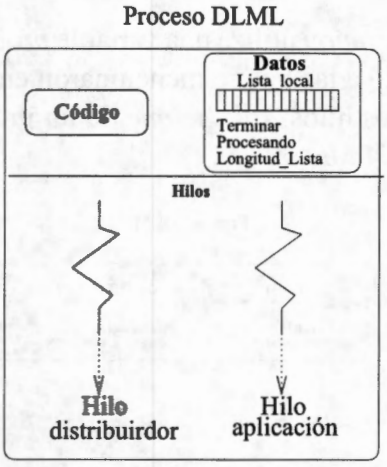


Figura 5.24: Estructura de un proceso DLML con un hilo *distribuidor* y uno *Aplicacion*.

En esta versión debido a que ahora la interacción *Aplicación-Distribuidor* se da mediante hilos, no es necesario obtener memoria compartida (para procesos) del sistema para comunicar al hilo *Distribuidor* y al hilo *Aplicacion*, sino simplemente con la memoria que el proceso *DLML* obtenga (dinámicamente) se pueden comunicar, lo cual resulta ventajoso por lo mencionado en la versión anterior.

La principal desventaja de esta versión radica en que un hilo Distribuidor para recibir peticiones de algún otro hilo Distribuidor o algún hilo Aplicación necesita el monitoreo constante de señales de sincronización lo que origina la utilización del procesador constantemente sin efectuar cálculos.

En esta versión se soluciona el problema de la memoria, pero la cantidad de mensajes en estas dos versiones sigue siendo la misma debido a que en ambas se considera un algoritmo de subasta global. Esto puede repercutir en la escalabilidad de DLML para aplicaciones donde los datos tuvieran una granularidad fina y se tuviera que efectuar constantemente el algoritmo de subasta.

5.4.2. Propuesta 2: Algoritmo híbrido (Uso de hilos y subastas globales y locales)

Como se vio en la primera propuesta, un proceso *DLML* se comunica con su proceso Aplicación por memoria compartida, con lo cual el proceso *DLML* no tiene que esperar hasta que su proceso *Aplicacion* termine de procesar un dato para tener acceso a la lista.

Como señalamos en las dos versiones anteriores el número de comunicaciones para buscar/obtener datos sigue siendo el mismo que en la versión original de DLML. Lo cual como se menciona en [Gar07] podría ocasionar que al ejecutar DLML sobre un cluster con una gran cantidad de procesadores surja un problema de escalabilidad debido a que se utiliza una política de información global. Para buscar solucionar lo anterior se generó esta tercer versión, la cual fue llamada DLML-híbrido y es una versión que combina ideas de las dos versiones anteriores y además aplica una subasta local y una global.

En esta versión se crea solamente un proceso *DLML* por nodo, este proceso contiene *M+1* hilos (donde *M* es el número de procesadores en el nodo), un hilo es llamado el hilo *Distribuidor* y los demás son hilos *Aplicacion*. La estructura del proceso *DLML* se observa en la Figura 5.25. Como se observa en esta figura un proceso DLML cuenta con *M* listas locales, una lista para cada hilo

Aplicacion. Además cada hilo *Aplicacion* utiliza una variable *procesando* y una *longitud_lista*, estas variables tiene el mismo significado a las que se mencionaron en la primer propuesta de DLML ya que ahora el hilo *Distribuidor* y los hilos *Aplicacione* de un proceso *DLML* se comunican por la memoria compartida del proceso *DLML*.

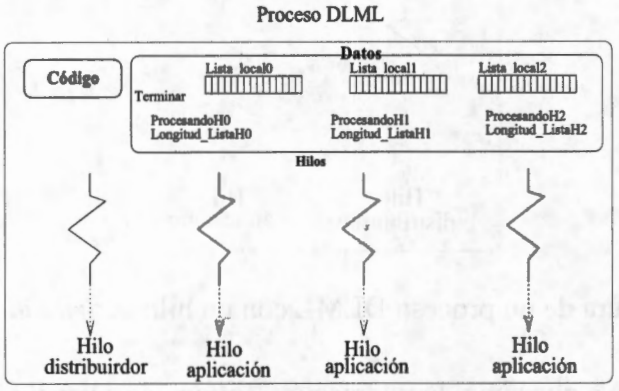


Figura 5.25: Estructura de un proceso *DLML* con *3* hilos *Aplicacion* y uno *Distribuidor*.

Cada hilo *Distribuidor* de un proceso *DLML* realiza el algoritmo de subasta con los otros hilos *Distribuidor* del sistema para obtener datos remotamente tal como se hacía en la primera y segunda versión. Esto origina que si estamos trabajando en un cluster con máquinas mono-procesador la cantidad de mensajes que se utilizan es el mismo en esta versión que en las dos versiones anteriores y en la versión original de DLML. La pregunta que cabría aquí es ¿qué pasa cuando la versión DLML-híbrido se ejecuta en máquinas multi-procesador o multi-core? En las dos versiones anteriores así como en la original de DLML la cantidad de mensajes sigue siendo la misma, pero en la versión de DLML-híbrido la cantidad de mensajes se reduce de manera considerable, por poner un ejemplo suponga un cluster con 32 nodos, cada nodo con 4 procesadores, esto origina un total de 128 procesadores. Si se ejecutara la versión DLML-híbrido en este cluster, para que un proceso sepa a quien le hará una petición de datos tendría que enviar únicamente 32 mensajes y recibir 32 mensajes de respuesta, lo que ocasiona un total de 64 mensajes. En las versiones anteriores se hubieran enviado 128 mensaje y recibido otros 128 mensajes para un total de 256 mensajes. Una de las razones por la que se reduce el número de los mensajes es que ahora el hilo *Distribuidor* y los hilos *Aplicación* de un proceso DLML realizan una subasta local.

En esta subasta local cuando un hilo *Aplicacion* termina los datos de su lista se lo notifica al hilo *Distribuidor* . Al recibir esta notificación el hilo *Distribuidor* obtiene la mitad de los datos del hilo *Aplicacion* que tenga más datos y se los envía al hilo que le hizo la petición. Cuando todas las locales de un nodo están vacías es el momento de iniciar la subasta global entre los hilos *Distribuidores* del sistema.

La versión DLML-híbrido fue verificada utilizando las mismas propiedades que se utilizaron en la versión DLML original, las cuales fueron:

1. No hay terminación prematura del distribuidor de datos.
2. Todo proceso sólo recibirá los mensaje dirigidos hacia él.
3. Todo proceso ejecutará la acción correspondiente a cada mensaje.

4. No hay pérdida de datos.

5. No hay abrazos mortales en el sistema.

Estas propiedades resultaron ser válidas contemplando canales con capacidad de almacenamiento de uno y más de un mensaje.

En este capítulo se han estudiado las diferentes propuestas que se plantearon para resolver los problemas encontrados en la librería DLML. Además en estas versiones se buscó disminuir el número de comunicaciones cuando DLML se ejecuta sobre un cluster multi-core o multiprocesador. En el siguiente capítulo presentamos el análisis de rendimiento de la versión de DLML-híbrido con respecto a la original, para lo cual se utilizaron dos aplicaciones reales.

Capítulo 6

Comparación de rendimiento de DLML híbrido

"Confía en el tiempo, que suele dar dulces salidas a muchas amargas dificultades."

–Miguel de Cervantes Saavedra, 1547-1616 Novelista, poeta y dramaturgo español.

En el capítulo anterior se mostraron diferentes versiones de la librería/herramienta DLML. Todas estas versiones solucionan el problema de abrazo mortal debido a que los *buffers* de mensajes de *MPI* pueden tener la capacidad de almacenar un mensaje. Cabe señalar que el mapeo entre el código *promela* y el código C-mpi fue realizado de manera informal. Si bien tuvimos suficiente cuidado para realizar una buena traducción, formalmente no podemos decir que ésta es matemáticamente correcta. Desafortunadamente la técnica de *model-checking* tiene esta desventaja, para solucionarlo se puede combinar con otras técnicas como métodos axiomáticos. Esta tesis no intenta ir hasta tales niveles de precisión por lo cual únicamente mencionaremos estos inconvenientes sobre estas implementaciones. De estas versiones la que presentó un mejor desempeño (en cuestión de tiempo) al ejecutar una aplicación con datos de granularidad fina y otra con granularidad gruesa fue la versión de DLML híbrida. En este capítulo describimos las características de cada una de estas aplicaciones, además mostramos una comparación del rendimiento de la versión original de DLML y la versión híbrida.

6.1. Aplicaciones

Las aplicaciones que fueron utilizadas para comparar el rendimiento de la versión híbrida de DLML con respecto a la versión original fueron: el algoritmo clásico para el problema de las N-Reinas [BD75] y, una para segmentación de imágenes usando Mean-Shift (MSH) [AAC+06]. A continuación se explica en qué consiste cada una de estas aplicaciones.

6.1.1. Problema de las N-Reinas

Consiste en encontrar todas las posibles colocaciones de N reinas en un tablero de ajedrez de tamaño $N \times N$ sin que se ataquen entre sí [BD75]. Para ello, se modelan las posibles soluciones en

un árbol de búsqueda. Las soluciones se encuentran explorando todo el árbol de búsqueda de las posibles soluciones eliminando aquellas que no puedan ser solución. En DLML cada elemento de la lista contiene una posible solución a explorar conformada por el número de reina a ser colocada y un vector de tamaño N con la posición de la reinas colocadas hasta ese momento. El tamaño de la lista crece y decrece conforme se van generando nuevas soluciones o eliminando, respectivamente. El costo para procesar cada elemento está en función de la profundidad del elemento en el árbol de búsqueda.

6.1.2. Segmentación de imágenes

Esta aplicación reconstruye imágenes de cerebro en 3D usando el método Mean-Shift (MSH) [AAC+06]. La reconstrucción se logra aplicando MSH sobre varias imágenes 2D (cortes) a nivel de pixel (costoso por el gran número de cortes y la alta resolución requerida). Para reducir tiempos los cortes se dividen entre los procesadores. No obstante que el número de cortes es fijo (número de imágenes), el costo de procesamiento de cada corte es diferente ya que el costo de MSH aumenta o disminuye de acuerdo a los cambios en la intensidad de los pixels.

6.2. Infraestructura

La infraestructura donde fueron ejecutadas estas aplicaciones fue un cluster de la Universidad Tecnológica de Munich de 32 nodos que permiten ejecuciones de 32 y 64 bits. Los nodos tienen cuatro procesadores Opteron 850 a 2.4 GHz con 8GB de Ram. Los nodos 25, 26, 27 tienen 16 GB de ram. Cada nodo tiene dos discos SCSI corriendo en RAID 1 (espejo). Los nodos están equipados con dos puertos Gigabit Ethernet y una tarjeta adaptador InfiniBand MT23108.

6.3. Resultados

En esta sección se realiza una comparación del rendimiento de la versión DLML híbrida con respecto a la versión original de DLML. Primero se presentan los resultados para la aplicación de las N-Reinas (en particular para N= 16, 17 y 18), después para la segmentación de imágenes.

6.3.1. Resultados respecto a la aplicación de las N-Reinas

La Figura 6.1 representa los tiempos de ejecución para la solución al problema de 16-Reinas (número de soluciones 14772512). El eje horizontal hace referencia al número de procesadores utilizados y el eje vertical al tiempo de ejecución obtenido. Como se observa en esta figura dado que la granularidad de los datos que utiliza esta aplicación es fina, entre menos procesadores tengamos es muy costoso realizar las operaciones *lock* y *unlock* sobre cada dato a procesar. Cuando el número de procesadores (nodos) aumenta, dado que el algoritmo híbrido realiza menos comunicaciones que la versión original de DLML, éste tarda menos en resolver el problema.

La figura 6.2 representa el problema de las 17-Reinas (número de soluciones 95815104). Los ejes tienen el mismo significado que en la figura anterior. En esta figura se observa un comportamiento similar al visto en la gráfica anterior ya que al inicio de la ejecución tarda menos en su

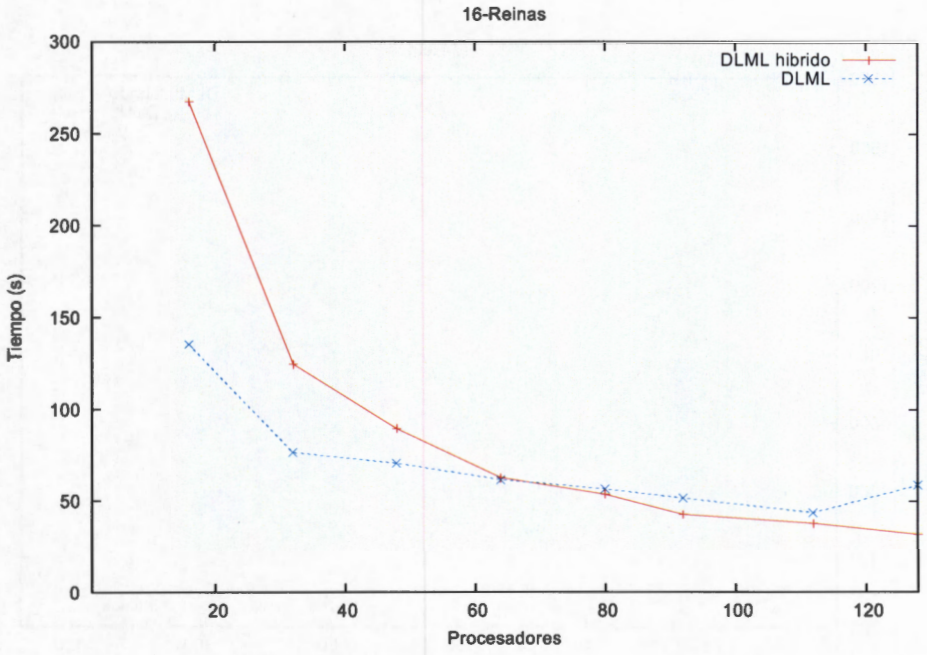


Figura 6.1: Tiempo en encontrar las soluciones del problema de 16-Reinas usando 16, 32, 48, 64, 80, 96, 112 y 128 procesadores.

ejecución la versión original de DLML, pero entre más procesadores se integren llega el momento que le cuestan mucho las comunicaciones, mientras tanto el DLML híbrido aún sigue siendo escalable.

La figura 6.3 representa el problema de las 18 Reinas (número de soluciones 666090624). En esta figura se observa que para los 128 procesadores utilizados la versión de DLML original siempre está abajo en tiempo de la versión de DLML híbrida como la cantidad de datos generados aumenta considerablemente se vuelve demasiado penalizante la sincronización para sacar cada dato de la lista. Esto no pasa en la versión original de DLML pues en tal propuesta no existe esa fuerte sincronización, la comunicación comienza hasta que una lista queda vacía. Se observa que al parecer la versión DLML híbrida con más procesadores podría seguir reduciendo sus tiempos, pero dado que solamente se cuenta con 128 no fue posible confirmar lo anterior.

6.3.2. Resultados respecto a la aplicación de segmentación de imágenes

Como se mencionó en la sección 6.1.2 en esta aplicación se tiene un conjunto de cortes los cuales se procesan en paralelo. En la figura 6.4 se muestra el tiempo de procesamiento para 165 cortes. El eje horizontal representa el número de corte y el eje vertical el tiempo de procesamiento del corte. Esta aplicación se ejecutó tanto con la versión original de DLML como con la versión DLML híbrido, cada corte con resolución de 217×181 pixeles. La figura 6.5 muestra el tiempo de procesar los 165 cortes para DLML original y DLML híbrido. Como se observa en esta figura, dado que la granularidad para cada corte es gruesa, DLML híbrido siempre está por abajo de DLML original. Lo anterior se debe a que en la versión de DLML original un proceso aplicación puede cederle datos a su proceso distribuidor hasta que termine de procesar el dato que tiene en ese

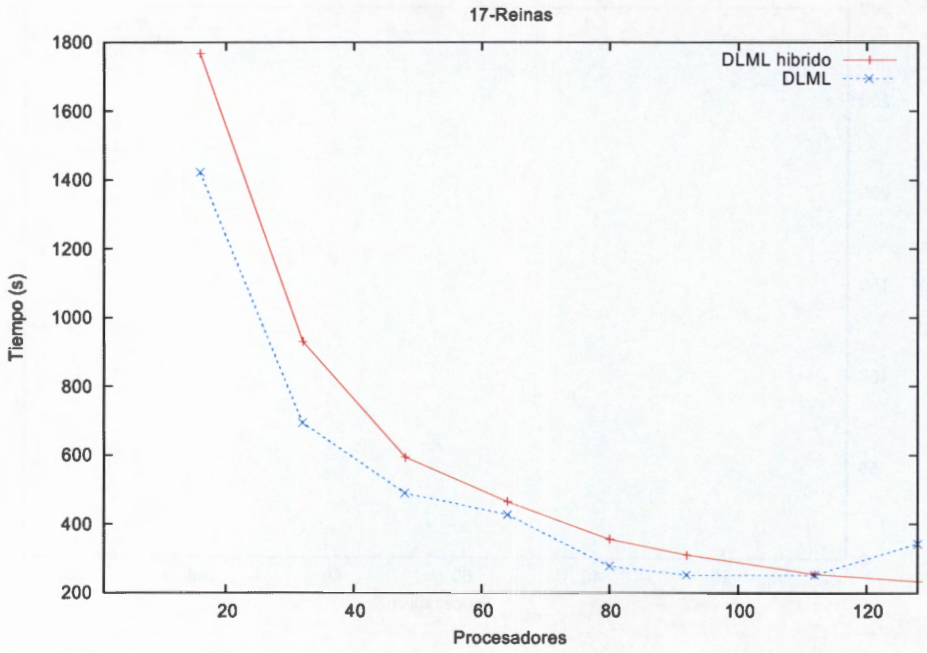


Figura 6.2: Tiempos en encontrar las soluciones del problema de 17-Reinas usando 16, 32, 48, 64, 80, 96, 112 y 128 procesadores.

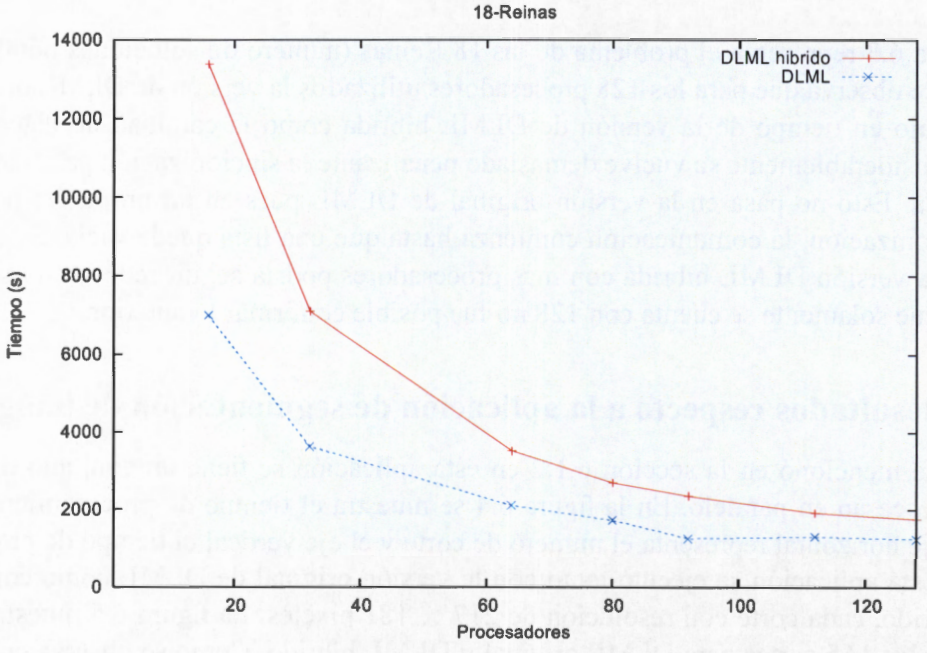


Figura 6.3: Tiempos en encontrar las soluciones del problema de 18-Reinas usando 16, 32, 48, 64, 80, 96, 112 y 128 procesadores.

momento. Como vimos en la versión de DLML híbrido un proceso DLML puede obtener datos de la lista sin tener que esperar a que su aplicación termine de procesar el dato que obtuvo en ese momento debido a que su lista de datos se encuentra compartida.

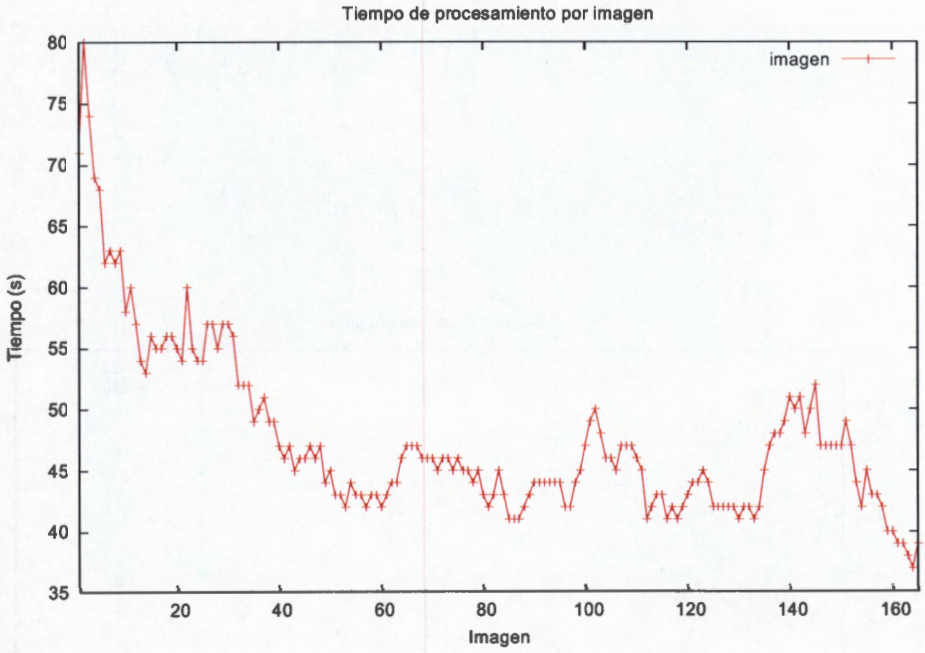


Figura 6.4: Tiempo de procesamiento de cada uno de los 165 corte (imágenes).

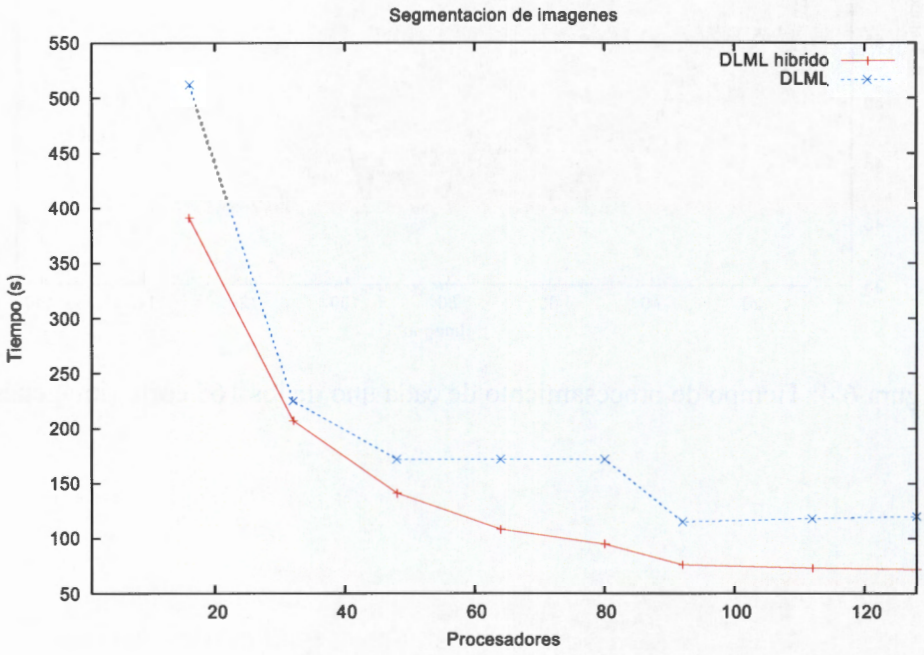


Figura 6.5: Tiempo de procesamiento de los 165 cortes (imágenes) en paralelo usando 16, 32, 48, 64, 80, 96, 112 y 128 procesadores.

Capítulo 7

Conclusiones y trabajo a futuro

"Duda siempre de ti mismo, hasta que los datos no dejen lugar a dudas."

–Louis Pasteur 1822-1895 Químico francés.

Este capítulo concluye esta tesis de maestría. Se resumen, a grandes rasgos, las principales ideas planteadas en la misma y se proponen posibles líneas futuras de investigación a partir de los resultados obtenidos en esta tesis.

7.1. Conclusiones

Como mencionamos, los algoritmos de distribución de datos (carga) proveen un conjunto de beneficios a nuestras aplicaciones paralelas tales como: la minimización de su tiempo de ejecución, la maximización de uso de nuestros recursos, etc. Pero debido a su naturaleza paralela, la implementación de un algoritmo de distribución de datos es compleja, lo que puede originar que no cumpla con las especificaciones para las que fue diseñado.

En este trabajo se propuso una estructura básica de integración de un algoritmo de distribución cíclico en una aplicación SPMD de procesamiento de datos independientes. A diferencia de otros trabajos la verificación de esta propuesta se realizó mediante el método formal *model-checking* utilizando la herramienta *Spin*. La estructura se modelo en el lenguaje *promela* requerido por *Spin* y se especificaron las propiedades necesarias para un buen funcionamiento. La especificación se realizó usando lógica temporal, afirmaciones (assert) y búsqueda de estados inválidos (invalid end-states). Estas propiedades consideraron los siguientes comportamientos:

1. No hay pérdida de mensajes.
2. No hay abrazos mortales en el sistema.
3. El algoritmo terminará cuando no haya datos a procesar en el sistema.
4. Si un proceso inicia el protocolo de terminación, eventualmente cada proceso en el sistema iniciará el protocolo de terminación y recibirá N mensajes FIN.
5. No hay terminación prematura del distribuidor de datos.

Después de haber modelado y verificado la estructura básica de distribución se estudió la arquitectura del distribuidor DLML (Data List Management Library). Debido a sus características, esta herramienta permitió refinar la estructura básica propuesta. Las principales características que presenta DLML son:

1. Un manejo transparente de datos para el usuario.
2. Utiliza un algoritmo de distribución de subasta con una política de control distribuida con política de información global.
3. Una metodología para la programación basada en listas.
4. Una dependencia en la granularidad de los datos para permitir la distribución de los mismos.

Se propuso el modelado de DLML en *promela* verificando las mismas propiedades que en la estructura básica y adicionando dos más asociadas con su implementación. Debido a que el distribuidor DLML se encuentra implementado en MPI la verificación de las propiedades propuestas se realizó con canales de comunicación de capacidad de uno y más de un mensaje. Las propiedades adicionales son las siguientes:

1. Todo proceso ejecutará la acción correspondiente a cada mensaje.
2. Todo proceso sólo recibirá los mensajes dirigidos para él.

El resultado de la verificación de la versión original de DLML arrojó un error de abrazo mortal al considerar canales con capacidad de un mensaje. Para solucionar lo anterior se propuso agregar un *buffer* temporal en el cual se almacenara un mensaje si el último que habían enviado aún no había sido entregado.

Después de haber verificado la versión original de DLML se propusieron varias implementaciones de este distribuidor donde se contemplan además de la capacidad de los canales, la no dependencia en la granularidad de los datos y en una de estas versiones la disminución de la cantidad de mensajes que se realizan en el protocolo de subasta (y con ello mejorar su rendimiento). A la nueva versión de DLML que considera la disminución de los mensajes se le llamó el DLML-híbrido (dado que incorpora subastas locales y globales). En el distribuidor DLML-híbrido se incorporó el uso de memoria compartida para comunicar a las parejas *Distribuidor-Aplicaciones*, pero además se aprovecha el surgimiento de las nuevas arquitectura *multicore* (donde se tiene más de un núcleo (procesador) por nodo) creando un sólo proceso *DLML* por cada nodo. En cada proceso DLML se crea un hilo para el distribuidor y un conjunto de hilos igual al número de procesadores del nodo, para las aplicaciones. Lo anterior reduce de manera significativa el número de mensajes que se utilizaba en la versión original de DLML, además dado que se incorpora memoria compartida evitamos la dependencia de la granularidad de los datos. La versión DLML-híbrido se modeló en *promela* y fue verificada utilizando las mismas propiedades que se verificaron sobre la versión original de DLML.

La versión DLML-híbrido fue comparada en rendimiento con la versión original de DLML, para la cual se utilizaron aplicaciones que manejan datos de granularidad diferente (fina y gruesa). Esta comparación se efectúo utilizando un cluster de 32 nodos de tipo multiprocesador (4 procesadores por nodo, teniendo un total de 128 procesadores) . Para la aplicación de granularidad fina

DLML-híbrido presentó mejores tiempos de ejecución mientras más procesadores se utilizan. La versión DLML original para este caso obtuvo mejores tiempos de ejecución con un número menor de procesadores y perdió eficiencia cuando el número de nodos aumentó. Para la aplicación de granularidad gruesa la versión DLML-híbrido siempre obtuvo mejores tiempos de ejecución que la versión original. Cabe mencionar que la nueva implementación de DLML-híbrido fue elaborada de manera que cada instrucción utilizada fuera posible modelarla en *promela*, más sin embargo nada garantiza que nuestro mapeo del modelo al código MPI sea matemáticamente correcto.

Una limitante de este trabajo es que los algoritmos de distribución de carga resultan ser muy complejos de verificar ya que se necesitan varias instancias de un mismo procesos para observar su interacción lo que origina el requerimiento del uso de máquinas con grandes recursos (procesador y memoria principalmente). Sin embargo, los beneficios que nos pueden proveer al poder ser verificados son enormes dado que pueden ser utilizados para construir con mayor confianza aplicaciones de computo de alto rendimiento.

7.2. Trabajo a futuro

Hemos modelado y verificado el funcionamiento de la distribución de carga en el procesamiento de datos independientes, en donde cada vez que un proceso está descargado solicita trabajo al más cargado del sistema. Este modelo puede servir de base para desarrollar propuestas de algoritmos de balance de carga, en donde el principal interés es que todos los procesos realicen una cantidad de trabajo equitativa durante la ejecución. Además, en base a los resultados en rendimiento obtenidos, sería muy interesante proponer un algoritmo de distribución de datos para DLML que se adapte dinámicamente a un funcionamiento que utilice memoria compartida (con sincronización por cada dato obtenido) y a un funcionamiento como el de DLML original (sincronización sólo cuando la lista está vacía). Este nuevo algoritmo debería ser verificado dado que su complejidad aumenta lo cual, puede ocasionar que presente errores en su especificación.

Apéndice A

Propuestas de DLML

A.1. Propuesta 1: Uso de proceso DLML - proceso Aplicación y memoria compartida entre ellos

Para mostrar como funciona esta nueva versión de DLML revisaremos los siguientes posibles casos que se pueden presentar durante su ejecución.

- **Primer caso** Un proceso *Aplicación* termina de procesar sus datos, por tanto, se lo comunica a su proceso *DLML* para que este último busque datos.

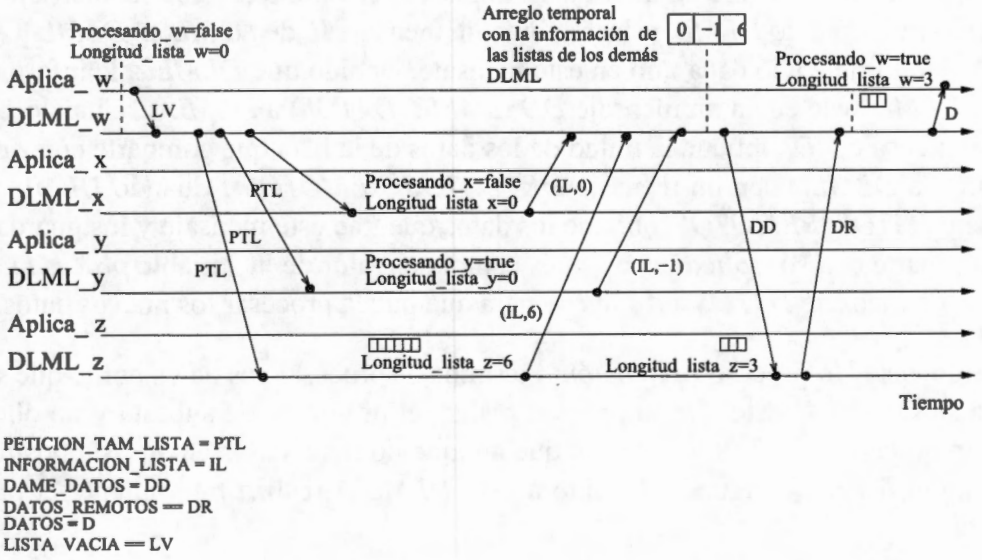


Figura A.1: Búsqueda y obtención de datos del proceso *DLML_w*

En la Figura A.1 se muestra un ejemplo donde interactúan 8 procesos: 4 procesos *DLML* (*DLML_w*, *DLML_x*, *DLML_y* y *DLML_z*) y 4 procesos *Aplicación* (*Aplica_w*, *Aplica_x*, *Aplica_y* y *Aplica_z*). En este ejemplo se observa que el proceso *Aplica_w* se encuentra procesando el último dato que sacó de su lista, por lo que las variables compartidas *Longitud_Lista_w* y *Procesando_w* tienen los valores 0 y *true* respectivamente. Este último dato

que esta procesando *Aplica_w* no genera más datos por lo que cuando se termina su procesamiento *Aplica_w* cambia el valor de *Procesando_w* a *false* y envía un mensaje *LV* (*LISTA_VACIA*) a *DLML_w* para que inicie la búsqueda de datos remotamente. Cuando *DLML_w* recibe el mensaje *LV* inicia el protocolo de subasta enviándoles un mensajes *PTL* (*PETICION_TAM_LISTA*) a *DLML_x, DLML_y* y *DLML_z*.

Cuando *DLML_x* recibe el mensaje *PTL* de *DLML_w*, realiza una lectura de la variable *Longitud_Lista_x*. Debido a que esta variable tiene el valor 0, procede a revisar el valor de *Procesando_x*. Dado que el valor de *Procesando_x* es *false*, esto significa que *Aplica_x* no tiene ni está procesando datos, por lo que DLML_x envía el mensaje *IL* (*INFORMACION_LISTA*) a *DLML_w*. Este mensaje lleva consigo un entero con valor 0 con el cual *DLML_x* le indica a *DLML_w* que no tiene datos.

Cuando *DLML_y* recibe el mensaje *PTL* de *DLML_w* revisa las variables *Longitud_Lista_w* y *Procesando_w* . Dado que están tiene los valores *0* y *true* respectivamente, entonces *DLML_y* enviará un mensaje *IL* con un entero con valor -1, con lo cual *DLML_w* sabrá que posiblemente *Aplica_y* puede generar datos ya que se encuentra procesando el último dato que obtuvo de su lista.

Debido a que la lista que *DLML_z* comparte con *Aplica_z* tiene datos, cuando *DLML_z* recibe el mensaje *PTL* de *DLML_w* le contesta enviándole un mensaje *IL* con el tamaño de esta lista.

Cada que *DLML_w* recibe un mensaje *IL* almacena el valor que trae este mensaje en un arreglo temporal. Cuando *DLML_w* ha recibido un mensaje *IL* de *DLML_x, DLML_y* y *DLML_z*, revisa el valor le envió cada uno en este mensaje. Debido que *DLML_z* le informó que tiene datos, *DLML_w* le envía un mensaje *DD* (*DAME_DATOS*) a *DLML_z*. Cuando *DLML_z* recibe el mensaje *DD*, obtiene la mitad de los datos de la lista que comparte con *Aplica_z* y se los envía a *DLML_w* en un mensaje *DR* (*DATOS_REMOTOS*). Cuando *DLML_w* recibe el mensaje *DATOS_REMOTOS*, obtiene los datos que trae este mensaje y los guarda en su lista que comparte con su *Aplica_w*, después cambia el valor de la variable *procesando_w* a *true* y envía un mensaje *DATOS* a *Aplica_w* para que pueda procesar los nuevos datos.

- **Segundo caso** Un proceso *Aplicación A* termina de procesar sus datos por lo que se lo comunica a su *DLML A*. Este último proceso realiza el protocolo de subasta y no obtiene datos, pero un proceso *DLML B* le informa que aunque no tiene datos en su lista posiblemente su *Aplicación B* generé algunos. Debido a esto *DLML A* realiza nuevamente el protocolo de subasta.

  En este caso se hace uso del ejemplo en la Figura A.2 en el cual el proceso *DLML A* es representado por *DLML_w* y *DLML B* es *DLML_y*. Como se observa en esta figura, este ejemplo se diferencia del de la figura A.1 en que cuando *DLML_z* recibe el mensaje *PTL* de *DLML_w*, *Aplica_z* ha terminado de procesar todos los datos de su lista, debido a esto *DLML_z* agrega un valor 0 en un mensaje *IL* y se lo envía a *DLML_w*. Cuando *DLML_w* revisa los valores que le enviaron en los mensajes *IL* observa que le enviaron valores 0's (de los procesos *DLML_x* y *DLML_z*) y un valor -1 (del proceso *DLML_y*). Por el -1 que le envió *DLML_y*, *DLML_w* sabe que posiblemente *Aplica_y* pueda estar generando datos

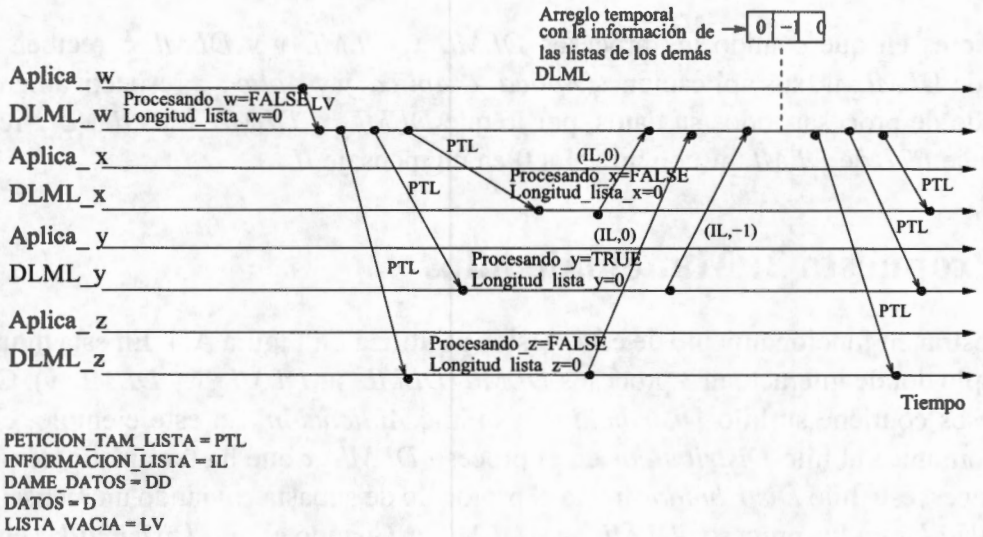


Figura A.2: Búsqueda de datos implicando una nueva realización del protocolo de subasta por el proceso *DLML_w*

(debido a la generación dinámica) por lo que realiza nuevamente el protocolo de subasta. Para este ejemplo sólo un proceso envió un mensaje *IL* con un valor -1, pero puede ocurrir que más de un proceso lo haga.

- **Tercer caso** Un proceso *Aplicación* A termina de procesar sus datos por lo que se lo comunica a su *DLML A*. Este último proceso realiza el protocolo de subasta y no obtiene datos, por lo que le comunica al proceso *Aplicación A* que termine con su ejecución e inicia un protocolo para que terminen su ejecución los procesos *DLML*.

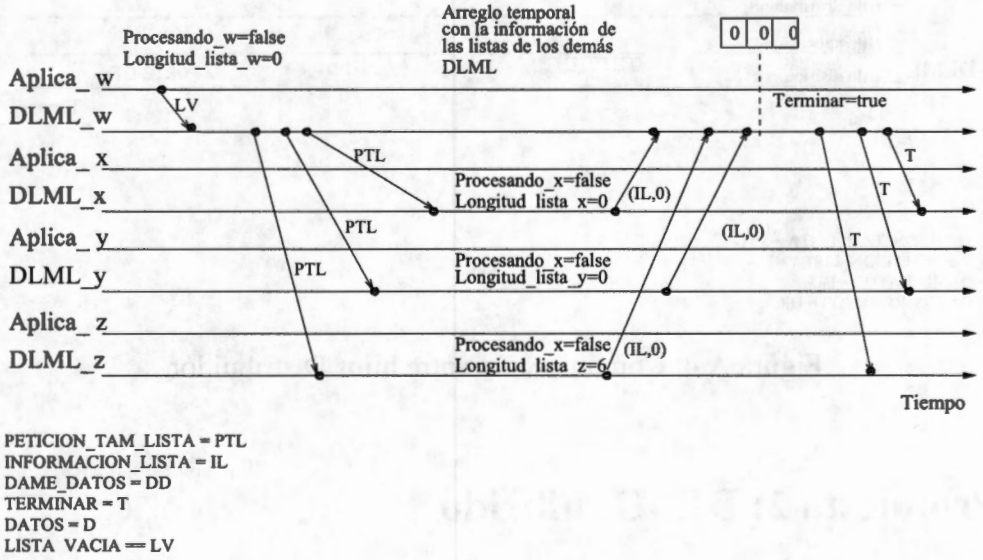


Figura A.3: Realización del protocolo de subasta por DLML_w y terminación de datos del sistema

Para este caso hacemos uso del ejemplo que se muestra en la Figura A.3, en este ejemplo *DLML_A* es *DLML_w* y *Aplicación A* es *Aplica_w* . Este ejemplo se diferencia de los dos

anteriores en que cuando los procesos *DLML_x*, *DLML_y* y *DLML_z* reciben el mensaje *PTL* de *DLML_w*, sus aplicaciones(*Aplica_x*, *Aplica_y* y *Aplica_z* respectivamente) han terminado de procesar todos su datos, por lo que *DLML_x*, *DLML_y* y *DLML_z* responden al mensaje *PTL* de *DLML_w* con un valor 0 en un mensaje *IL*.

A.2. Propuesta 2: Manejo de hilos

Para mostrar en funcionamiento de esta versión se utiliza La Figura A.4. En esta figura se muestra un ejemplo donde interactúan 3 procesos *DLML* (*DLML_w*, *DLML_x* y *DLML_y*). Cada uno de estos procesos contiene su hilo *Distribuidor* y su hilo *Aplicación*. En este ejemplo, el hilo *Aplicación* le comunica al hilo *Distribuidor* en el proceso *DLML_x* que ha terminado de procesar sus datos, entonces, este hilo *Distribuidor* inicia el protocolo de subasta enviando un mensaje *PTL* a los hilos *Distribuidor* en los procesos *DLML_w* y *DLML_y*. Cuando el hilo *Distribuidor* en el proceso *DLML_w* recibe el mensaje *PTL*, le envía al hilo *Distribuidor* en *DLML_x* un mensaje *IL* con el tamaño de su lista (para este ejemplo 3). En el caso del hilo *Distribuidor* en el proceso *DLML_y*, cuando recibe el mensaje *PTL* le envía el un mensaje *IL* con un valor 12 al hilos *Distribuidor* en le proceso *DLML_x*. Cuando el hilo *Distribuidor* en *DLML_x* recibe un mensaje *IL* de *DLML_w* y uno de *DLML_y*, le envía un mensaje *DD* al hilo *Distribuidor* en el proceso *DLML_y* ya que este fue quien le envió que tenia la mayor cantidad de datos. Cuando el hilo *Distribuidor* en el proceso *DLML_y* recibe el mensaje *DD*, le envía la mitad de los datos de su en un mensaje *DR*.

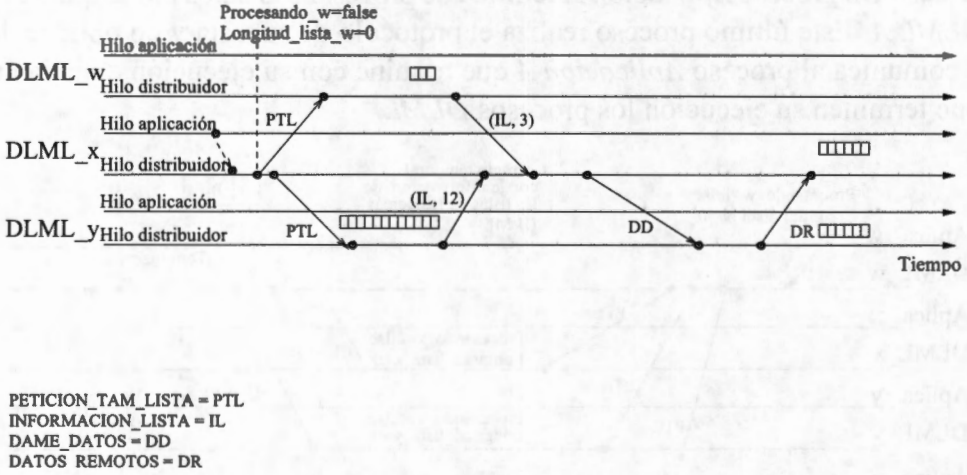


Figura A.4: Comunicación entre hilos Distribuidor.

A.3. Propuesta 2: DLML híbrido

Para mostrar como funciona esta versión, explicamos los siguientes 3 escenarios de ejecución de DLML:

- **Subasta Local** Un hilo *Aplicación A* le comunica al hilo *Distribuidor A* en el *proceso DLML A* que ha terminado sus datos, por lo que el *Distribuidor A* revisa cual es el hilos *Aplicación*

(dentro del proceso *DLML A*) con más datos en su lista para obtener la mitad de ellos y dárselos al hilo *Aplicación A*.

Para este escenario hacemos uso del ejemplo en la figura A.5 (en esta figura las líneas delgadas representa hilos de ejecución). En esta ejemplo el proceso *DLML A* es el proceso *DLML_0*, el hilo *Distribuidor A* es el hilo *Hilo_distrib_N0* y el hilo *Aplicación A* es el hilo *Hilo_aplic_0*.

En este ejemplo se observa que el hilo *Hilo_aplic_0* termina de procesar sus datos por lo que se lo notifica al hilo *Hilo_distrib_N0*. Cuando el *Hilo_distrib_N0* recibe esta notificación procede a leer la cantidad de datos que tienen los hilos *Hilo_aplic_1* (*R(Tlh1)*) y *Hilo_aplic_2* (*R(Tlh2)*). Debido a que el hilo *Hilo_aplic_1* tiene la mayor cantidad de datos, el hilo *Hilo_distrib_N0* ejecuta las operaciones *P(h1)* y *V(h1)* para obtener de manera exclusiva la mitad de los datos de la lista de *Hilo_aplic_1*. Los datos que obtuvo de la lista de *Hilo_aplic_1* se los agrega a la lista de *Hilo_aplic_0*, después le notifica a este último hilo que puede seguir con su ejecución.

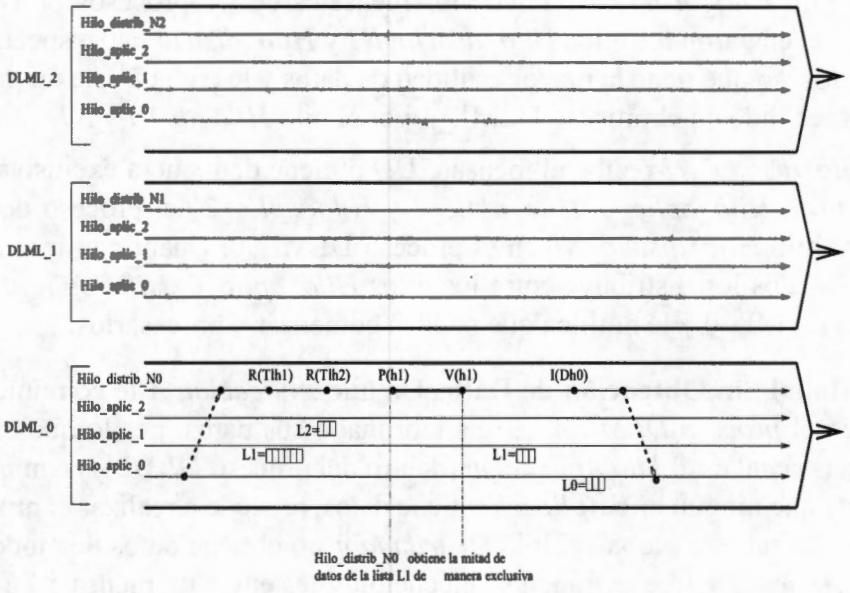


Figura A.5: Obtención de datos por subasta local en el proceso *DLML_0*.

- **Subasta Global** Un hilo *Aplicación A* le comunica al hilo *Distribuidor A* en el *proceso DLML A* que ha terminado sus datos, por lo que el *Distribuidor A* revisa cual es el hilo *Aplicación* (dentro del proceso *DLML A*) con más datos pero al darse cuenta que ningún hilo *Aplicación* tiene datos, procede a realizar el protocol de subasta global. Después que el hilo *Distribuidor A* recibe datos remotos, los reparte entre los hilos *Aplicación* del *proceso DLML A*.

  Para este escenario hacemos uso del ejemplo en la figura A.6. En esta ejemplo al igual que en el anterior, el proceso *DLML A* es el proceso *DLML_0*, el hilo *Distribuidor A* es el hilo *Hilo_distrib_N0* y el hilo *Aplicación A* es el hilo *Hilo_aplic_0*.

  Al igual que en el ejemplo anterior, en este ejemplo se observa que el hilo *Hilo_aplic_0* termina sus datos por lo que se lo notifica al hilo *Hilo_distrib_N0*. El hilo *Hilo_distrib_N0*

al recibir esta notificación realiza una 2 lecturas para saber el tamaño de las listas de *Hilo_aplic_1* y *Hilo_aplic_2* (ejecuta *R(Tlh1)* para saber el tamaño de la lista de *Hilo_aplic_1* y *R(Tlh2)* para el tamaño de *Hilo_aplic_2*). Debido a que las listas de *Hilo_aplic_1* y *Hilo_aplic_2* están vacías, esto significa que en el proceso *DLML_0* no tiene datos, de modo que el hilo *Hilo_distrib_N0* comienza el protocolo de subasta remoto, para lo cual les envía un mensaje *PTL* a los hilos distribuidores *Hilo_distrib_N1* y *Hilo_distrib_N2* de los procesos *DLML_1* y *DLML_2* respectivamente.

Cuando el hilo *Hilo_distrib_N1* recibe el mensaje *PTL*, revisa cantidad de datos que tiene los hilos *Hilo_aplic_0*, *Hilo_aplic_1* y *Hilo_aplic_2* en el proceso *DLML_1* ejecutando *R(Tlh0)*, *R(Tlh1)* y *R(Tlh2)*. Como en el proceso *DLML_1* L0=7, L1=3 y L2=4 el hilo *Hilo_distrib_N1* le envía a *Hilo_distrib_N0* un mensaje *IL* con entero de valor 14 indicando que esa es la cantidad de datos que tiene el proceso *DLML_1*. Para el caso del proceso DLML_2, como L0=1, L1=1 y L2=1 el hilo *Hilo_distrib_N2* le envía a *Hilo_distrib_N0* un mensaje *IL* con un valor 3.

Cuando el hilo *Hilo_distrib_N0* recibe los mensajes de los procesos procesos *DLML_1* y *DLML_2* (que enviaron los hilos *Hilo_distrib_N1* y *Hilo_distrib_N2* respectivamente) revisa cual es el proceso que tiene la mayor cantidad de datos y le envía a este un mensaje *DD*. Este mensaje es recibido en el proceso DLML_1 por el hilo *Hilo_distrib_N1*.

Cuando *Hilo_distrib_N1* recibe el mensaje *DD* obtiene de manera exclusiva la mitad de datos de los hilos *Hilo_aplic_0*, *Hilo_aplic_1* y *Hilo_aplic_2* del proceso donde el esta y se los envía al hilo *Hilo_distrib_N0* en el proceso DLML_0. Cuando el hilo *Hilo_distrib_N0* recibe estos datos los distribuye entre los hilos *Hilo_aplic_0*, *Hilo_aplic_1* y *Hilo_aplic_2* del proceso DLML_0 y le notifica que pueden comenzar a procesarlos.

- **Subasta Global Sin Obtención de Datos** Un hilo *Aplicación A* le comunica al hilo *Distribuidor A* en el *proceso DLML A* que ha terminado sus datos, por lo que el *Distribuidor A* primero revisa cual es el *hilo Aplicación* (dentro del proceso *DLML A*) con más datos pero al darse cuenta que ningún hilo *Aplicación* tiene datos, procede a realizar el protocol de subasta global. En esta subasta global, el hilo *Distribuidor* no obtiene datos de modo que le informa a los hilos *Aplicación* que terminen su ejecución y les envía un mensaje *TERMINAR* a cada proceso *DLML*.

  Para este escenario hacemos uso del ejemplo en la figura A.7. En esta ejemplo al igual que en el anterior, el proceso *DLML A* es el proceso *DLML_0*, el hilo *Distribuidor A* es el hilo *Hilo_distrib_N0* y el hilo *Aplicación A* es el hilo *Hilo_aplic_0*.

  El ejemplo de la Figura A.7 es muy similar al ejemplo anterior, sólo que ahora cuando el hilo *Hilo_distrib_N1* del proceso DLML_1 y el hilo *Hilo_distrib_N2* reciben el mensaje del hilo *Hilo_distrib_N0* del proceso DLML_0 al hacer revisar cuantos datos tienen observan que sus respectivos hilos aplicaciones han terminado de procesar sus datos, por lo que le envían un mensaje IF con un valor 0. Cuando el hilo *Hilo_distrib_N0* del proceso DLML_0 recibe los mensajes al percatarse que no hay datos en los demás procesos remotos envía un mensaje T a los hilos *Hilo_distrib_N1* y *Hilo_distrib_N2*, después le comunica a los hilos *Hilo_aplic_0*, *Hilo_aplic_1* y *Hilo_aplic_2* que terminen su ejecución.

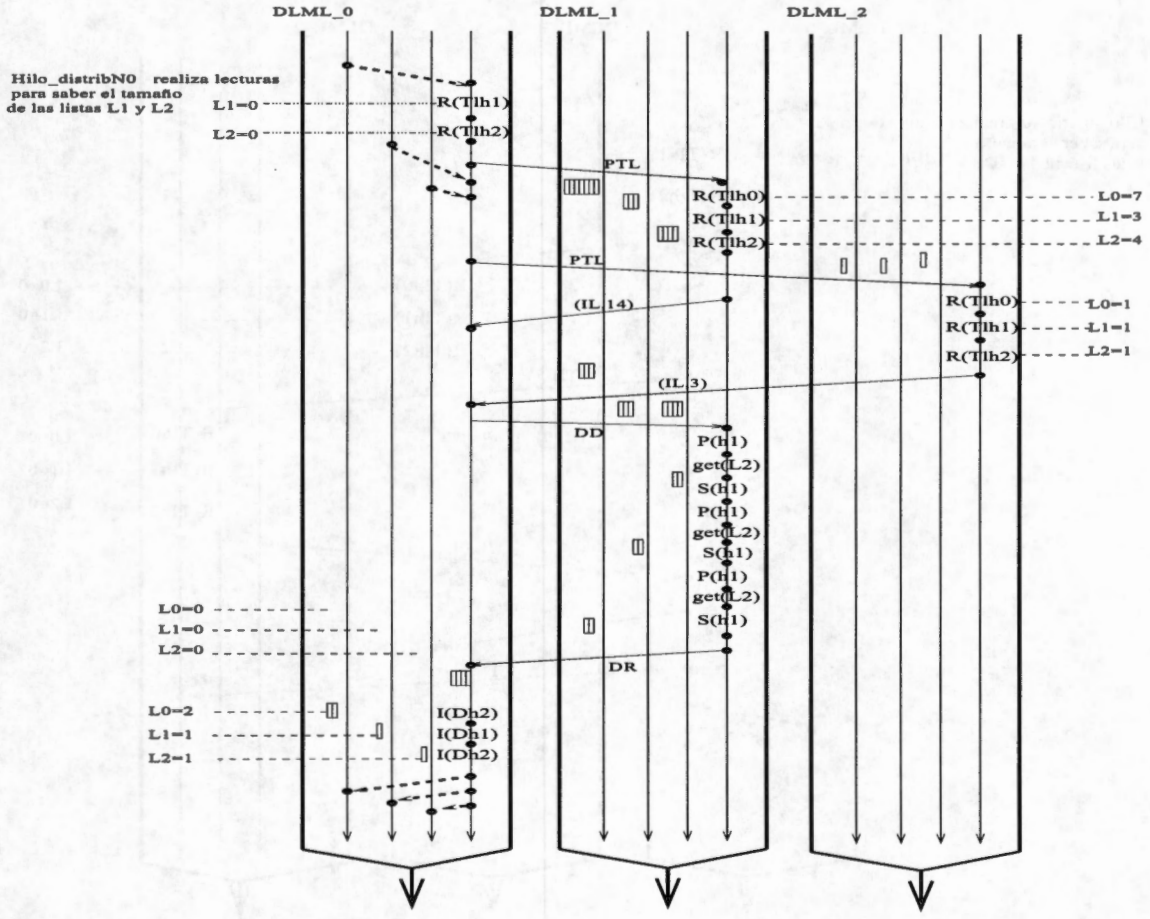


Figura A.6: Obtención de datos por subasta global originada en el proceso *DLML_0*.

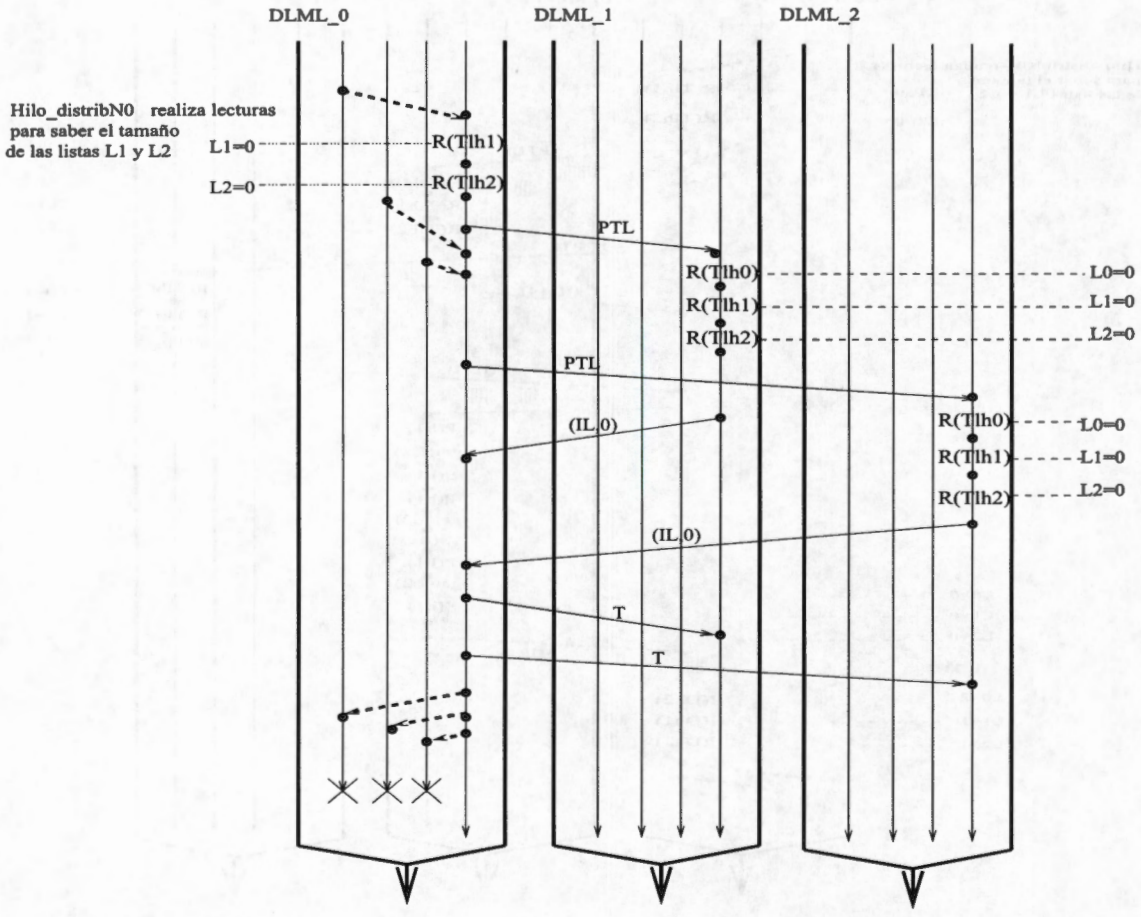


Figura A.7: Terminación de datos del sistema.

Apéndice B

Model Checker Spin

Spin (Simple Promela Interpreter) es una herramienta de software libre que puede ser usada para la simulación y verificación de sistemas distribuidos. Fue desarrollada en los laboratorios Bell por el grupo de investigación en ciencias de la computación.

Xspin es la interfaz gráfica de *Spin*, ésta puede ser muy útil para visualizar ejecuciones del sistema, lo cual, sirve para su depuración. La interfaz ejecuta el *model checker Spin* en segundo plano y sólo muestra los resultados dados por este último. *Xspin* está escrito en el lenguaje Tcl/Tk[1]. La ventana principal de *Xspin* se muestra en la Figura B.1. Como se observa en esta figura, *Xspin* cuenta con cinco menús:

1. ***File***: El menú *File* contiene las opciones para abrir, crear y guardar un nuevo archivo escrito en *promela*.

2. ***Edit***: El menú *Edit* contiene operaciones para copiar, cortar y pegar texto.

3. ***View***: El menú *View* nos permite modificar el tamaño de texto del área de edición del editor.

4. ***Run***: El menú *Run* contiene los parámetros para realizar una simulación o verificación.

5. ***Help***: El último menú es *Help*, contiene ayuda básica del funcionamiento del editor.

De los menús que mencionamos, el cuarto es el más importante (Figura B.2). Este menú contiene la opciones para la simulación y verificación del modelo del sistema, ya que en él se colocan los parámetros de *Spin* que se desean utilizar. Sus opciones son:

1. *Run Syntax Check*: Comprueba si la sintaxis de nuestro programa escrito en *promela* es correcta.

2. *Run Slicing Algorithm*: Indica segmentos y objetos de datos que son redundantes para la verificación en afirmaciones y *never claim*[2].

3. *Set Simulation Parameters..*: En esta ventana se establecen los parámetros para una simulación.

[1]Para más información consultar http://www.tcl.tk/

[2]Un *never claim* es una propiedad representada por un autómata escrito con sintaxis de *promela*.

Figura B.1: Ventana de principal de Xspin.

4. *(Re)Run Simulation...*: Realiza una simulación con los parámetros establecidos en *Set Simulation Parameters.*

5. *Set Verification Parameters...*: Se establecen los parámetros para la verificación de una propiedad

6. *(Re)Run Verification*: Realiza una verificación con los parámetros establecidos en *Set Verification Parameters.*

7. *LTL Property manager...*: Sirve para introducir una fórmula en lógica temporal y verificar ésta en nuestro sistema.

8. *View Spin Automaton for each Proctype...*: Realiza el *ANEP* para cada proceso de nuestro modelo.

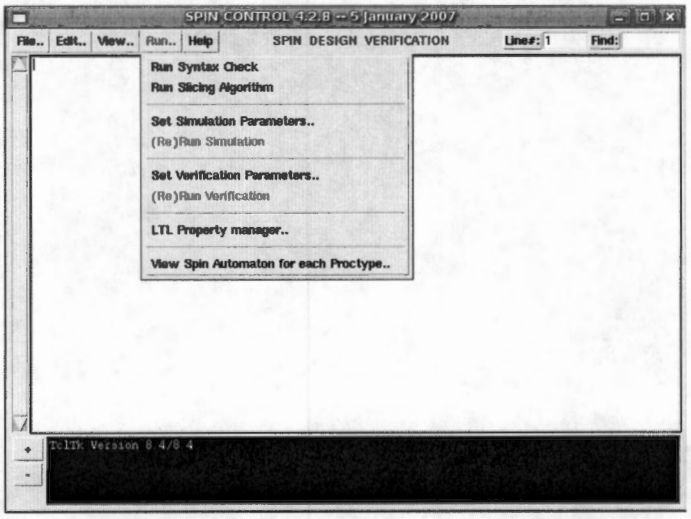


Figura B.2: Menú Run de de la ventana de Xspin.

Bibliografía

[AAC+06] ALONSO, G. R., ALANIZ, J. R. J., CHAVEZ, J. B., GARCIA, M. A. C., AND RODRIGUEZ, A. H. V. Segmentation of brain image volumes using the data list management library.

[ACD+96] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (1996), 18–28.

[BBD+02] BEHRMANN, G., BENGTSSON, J., DAVID, A., LARSEN, K. G., PETTERSSON, P., AND YI, W. Uppaal implementation secrets. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* (London, UK, 2002), Springer-Verlag, pp. 3–22.

[BD75] BRUEN, A., AND DIXON, R. The n-queens problem. *Discrete Mathematics 12* (1975), 393–395.

[BDG+91] BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., AND SUNDERAM, V. A user"s guide to pvm parallel virtual machine. Tech. rep., Knoxville, TN, USA, 1991.

[BDV94] BURNS, G., DAOUD, R., AND VAIGL, J. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium* (1994), pp. 379–386.

[BS85] BARAK, A., AND SHILOH, A. A distributed load-balancing policy for a multicomputer. *Softw. Pract. Exper. 15*, 9 (1985), 901–913.

[CA00] CORTÉS., E. P., AND ALONSO, G. R. ¿memoria compartida o mensajes? *Contactos UAM-I 38* (2000), 27–34.

[CCGRA05] CHÁVEZ, J. B., CASTRO-GARCÍA, M. A., AND ROMÁN-ALONSO, G. Simple, list-based parallel programming with transparent load balancing. In *PPAM* (2005), R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, Eds., vol. 3911 of *Lecture Notes in Computer Science*, Springer, pp. 920–927.

[CDK+01] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MCDONALD, J., AND MENON, R. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CGP99] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CGP02] CHANDRA, S., GODEFROID, P., AND PALM, C. Software model checking in practice: an industrial case study. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM, pp. 431–441.

[CK88] CASAVANT, T. L., AND KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng. 14*, 2 (1988), 141–154.

[Col91] COLE, M. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[DDHY92] DILL, D. L., DREXLER, A. J., HU, A. J., AND YANG, C. H. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors* (Washington, DC, USA, 1992), IEEE Computer Society, pp. 522–525.

[Fos95] FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gar07] GARCIA, M. C. *Programación con Listas de Datos para Cómputo Paralelo en Clusters*. Instituto Politecnico Nacional., México, DF, 2007.

[GDN+98] GOODEVE, D. M., DOBSON, S. A., NASH, J. M., DAVY, J. R., DEW, P. M., KARA, M., AND WADSWORTH, C. P. Toward a model for shared data abstraction with performance. *J. Parallel Distrib. Comput. 49*, 1 (1998), 156–167.

[Han98] HANSEN, P. B. An evaluation of the message-passing interface. *ACM Sigplan Notices 33*, 3 (March 1998), 65–72.

[Hol04] HOLZMANN, G. J. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.

[JM93] JACQMOT, C., AND MILGROM, E. A systematic approach to load distribution strategies for distributed systems. In *Proceedings of the IFIP WG10.3 International Conference on Decentralized and Distributed Systems* (Los Alamitos, California, 1993), North-Holland Publishing Co., pp. 291–303.

[Kum02] KUMAR, V. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LM92] LÜLING, R., AND MONIEN, B. Load balancing for distributed branch and bound algorithms. In *Proceedings of the 6th International Parallel Processing Symposium* (Washington, DC, USA, 1992), IEEE Computer Society, pp. 543–548.

[LMR91] LLING, R., MONIEN, B., AND RAMME, F. A study of dynamic load balancing algorithms. In *Proceedings of the 3rd IEEE SPDP* (Dallas, Texas, United States, 1991), IEEE Computer Society, pp. 686–689.

[MM01] MEZA MONTOYA, F. *Memoria Compartida Distribuida en Ambientes de Bajo costo*. Tesis, 2001.

[Muk96] MUKUND, M. Finite-state automata on infinite inputs. In *Tutorial talk, Sixth National Seminar on Theoretical Computer Science* (Banasthali, Vidyapith, Rajasthan, 1996).

[NXG85] NI, L. M., XU, C.-W., AND GENDREAU, T. B. A distributed drafting algorithm for load balancing. *IEEE Trans. Softw. Eng. 11*, 10 (1985), 1153–1161.

[PGS01] PELED, D. A., GRIES, D., AND SCHNEIDER, F. B., Eds. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Pnu77] PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium Foundations of Computer Science* (Washington, DC, USA, 1977), IEEE Computer Society, pp. 46–57.

[RLCB05] RICOLINDO L. CARI N., AND BANICESCU, I. A load balancing tool for distributed parallel loops. *Cluster Computing 8*, 4 (2005), 313–321.

[RMI] REMOTE, METHOD, AND INVOCATION. Remote method invocation home. In *http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp*.

[RR95] ROBBINS, K. A., AND ROBBINS, S. *Practical UNIX programming: a guide to concurrency, communication, and multithreading*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[RR96] RIEDL, R., AND RICHTER, L. Classification of load distribution algorithms. In *PDP '96: Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)* (Washington, DC, USA, 1996), IEEE Computer Society, p. 404.

[SKS92] SHIVARATRI, N. G., KRUEGER, P., AND SINGHAL, M. Load distributing for locally distributed systems. *Computer 25*, 12 (1992), 33–44.

[SL03] SQUYRES, J. M., AND LUMSDAINE, A. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting* (Venice, Italy, September / October 2003), no. 2840 in Lecture Notes in Computer Science, Springer-Verlag, pp. 379–387.

[SS84] STANKOVIC, J. A., AND SIDHU, I. S. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Unknown proceedings* (1984), IEEE, pp. 49–59.

[ST98] SKILLICORN, D. B., AND TALIA, D. Models and languages for parallel computation. *ACM Comput. Surv. 30*, 2 (1998), 123–169.

[Wol02] WOLPER, P. Constructing automata from temporal logic formulas: a tutorial. In *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science* (New York, NY, USA, 2002), Springer-Verlag New York, Inc., pp. 261–277.