Un Ambiente de Programación Visual Paralela Capaz de Generar Código Sequent y MPI

Ricardo Raúl Jacinto Montes
javier@gdl.cinvestav.mx
CINVESTAV del IPN,
Unidad Guadalajara
Prol. López Mateos Sur 590,
45090, Guadalajara, Jal.,
México

Javier González Sánchez
javier@gdl.cinvestav.mx
CINVESTAV del IPN,
Unidad Guadalajara
Prol. López Mateos Sur 590,
45090, Guadalajara, Jal.,
México

Ignacio Navarro Alvarez
inavarro@gdl.cinvestav.mx
CINVESTAV del IPN, Unidad
Guadalajara
Prol. López Mateos Sur 590,
45090, Guadalajara, Jal.,
México

Resumen

El reto principal para las ciencias de la computación sin lugar a duda radica en los lenguajes y herramientas de programación para las maquinas paralelas; esa es la orientación de este trabajo. El desarrollo de un lenguaje y un ambiente de programación paralela capaces de generar programas portables y que auxilie al usuario en la generación de sistemas eficientemente paralelos es la meta.

Introducción

La programación de computadoras paralelas es generalmente una tarea difícil, pero necesaria. Los lenguajes de programación visual representan un cambio en la notación más que un cambio en el nivel o el paradigma del lenguaje; la finalidad del enfoque visual es hacer el proceso de programación más fácil, rápido y confiable; un lenguaje visual, de forma simple, es un lenguaje en el cual un programa es construido utilizando elementos gráficos y textuales para crear expresiones multidimensionales. Este trabajo describe la estructura de un ambiente de programación visual para maquinas paralelas que utiliza el modelo de Redes de Petri y con ello es capaz de generar código ejecutable en arquitecturas de memoria compartida, utilizando las bibliotecas Sequent, y en sistemas distribuidos utilizando el estándar MPI.

Antecedentes sobre paralelismo.

Uno de los por qué del cómputo paralelo es el hecho de que nunca nada es suficiente, incluido el poder de un sistema de cómputo. El cómputo paralelo se enfoca a problemas que requieren gran poder de cómputo he incluye desde aplicaciones tradicionales de cálculo numérico hasta sistemas de tiempo real, pasando por ambientes virtuales, reconocimiento de imágenes, monitoreo en tiempo real de mecanismos, sistemas de base de datos, soporte en toma de decisiones y mucho otros. Por todo esto es necesario un uso efectivo del cómputo paralelo [Alm 89] y para ello existen dos aspectos primordiales a considerar:

Diseño de las computadoras paralelas. Las diferentes formas existentes de combinar procesadores, memoria y sistemas de control. Este es el aspecto del cómputo paralelo con mayores avances y madurez [Hambrusch 96] [Maggs 95]. En este trabajo consideramos 2 arqutecturas, una basada en el modelo MIMD con memoria compartida en grano grueso [SGI 95] y otra que representa al sistema distribuido clásico usando la arquitectura MPI [MPI 95].

Programación paralela. Si de entrada la programación de computadoras es una tarea difícil, más aun tratándose de sistemas paralelos. Los lenguajes tradicionales de alto nivel no son propiamente una solución cuando se trata de representar situaciones de paralelismo o concurrencia por ello se ha buscado una mejor forma de representar los elementos de un programa sustituyendo las cadenas de texto con elementos visuales.

Programación visual.

La finalidad del enfoque visual es hacer el proceso de programación más fácil, rápido y confiable a través de proveer al programador de un entendimiento de que es lo que hace el programa, que puede hacer, como trabaja, porque trabaja y los efectos de cada trabajo. Un lenguaje visual, de forma simple es un lenguaje en el cual un programa se construye utilizando elementos gráficos y textuales para crear expresiones. Los lenguajes visuales ofrecen grandes ventajas sobre los lenguajes textuales cuando se desea realizar programación en paralelo dado que la interconexión entre las diferentes partes del programa puede ser expresadas de una manera natural (multidimensional), sin embargo no se excluye el uso de textos en la definición de programas: nombrado de objetos, expresión de formulas matemáticas, comentarios en el pr

ograma, etc. Existen algunos problemas asociados con los referenciado es el del espacio en pantalla. En general este problema se resuelve fácilmente considerando: abstracción de la sintaxis de un programa visual, cosa que se mejora en gran medida con el uso de análisis incremental, esto es, realizar el análisis del programa desde que el programa comienza a ser construido en lugar de esperar a que el programa este totalmente terminado [Bla 91].

El lenguaje visual propuesto y su modelo subyacente.

La especificación del lenguaje propuesto considera los elementos más importantes del ambiente: sintaxis del lenguaje visual, reglas de edición, semántica del lenguaje visual y representación de la ejecución. Además se describe el modelo subyacente de desarrollo, un modelo basado en grafos. El lenguaje visual propuesto esta basado en la manipulación de un grafo subyacente, donde los programas se definen interconectando diferentes tipos de nodos y enlaces y llenando los valores de los atributos asociados con los nodos y los enlaces. En el nivel superior contamos con distintos tipos de nodos, incluso con nodos predefinidos para representar las distintas estructuras de control de un lenguaje de programación convencional. Así que desde un punto de vista simple los sistemas de software a desarrollar pueden ser vistos como una colección jerárquica de bloques individuales interconectados. Los bloques tienen un modelo asociado, esto es, su comportamiento se deriva de una descripción estructural. Cada bloque puede tener datos locales asociados (acorde a su definición) o contener un conjunto de bloques propio, lo cual permite la construcción y vista de sistemas de tamaño real. Pueden definirse datos globales o compartidos en el sistema a través de una primitiva creada para tal propósito.

La especificación formal del lenguaje visual consta de tres partes: primero una especificación gráfica de la forma de cada elemento y reglas de conexión; segundo de un conjunto de reglas de producción para especificar la estructura del lenguaje, dadas en la gramática Glide [Kleyn 95] como un sustituto de los metalenguajes textuales estilo BNF; y finalmente de una proyección de cada elemento gráfico a un modelo subyacente para describir el comportamiento de cada elemento y las acciones asociadas al programa, así como su representación al momento de simular su ejecución.

Aunque la programación gráfica de sistemas varia ampliamente en métodos y enfoques una gran cantidad, sino es que la mayoría, de los sistemas se basan en la manipulación de una estructura subyacente al lenguaje basada en grafos. El trabajo presentado en este documento no es la excepción. El uso de digrafos como modelos base para sistemas de cómputo se plantea en la literatura en gran medida [Skillicorn 96] [Jahanian 88] [Keller 82] [SC 91] y con características distintas: modelos de transición de estados, data flow, control flow, ModelChart, redes de Petri, etc. Para los propósitos de este trabajo se encontró en las lenguajes visuales, probablemente el problema más y jerarquización del problema. Un factor adicional, algunas veces considerado como problema, es el tiempo de análisis Redes de Petri un modelo propio para el cómputo paralelo. Por ello se empleará una derivación de este modelo denominado Redes de Petri Jerárquicas (RdPJ), un formalismo gráfico que permite la representación de sistemas concurrentes y permiten describir la secuencialidad de acciones, la selección no determinista, el asincronismo y el paralelismo, características propias de un sistema paralelo. Además de facilitar el análisis y descubrimiento de propiedades deseables en los sistemas paralelos (reinicialización, ausencia de bloqueos, vivacidad, seguridad, etc.). Es posible incrementar la potencia de expresión de las Redes de Petri asociando predicados a los elementos de las RdPJ, con lo que se cubre la mayoría de las necesidades de programación paralela. Las redes de Petri por tanto son útiles en ambos sentidos, para la definición y modelado matemático del sistema y como una herramienta de uso para describir visualmente el modelo de trabajo [Peterson 81].

Elementos gráficos del lenguaje.

Cada uno de los iconos denominados *actores* tiene asociadas expresiones o atributos propios que representan ya sea una operación atómica o un programa completo desarrollado con la herramienta, en C o en Fortran. Para crear un programa el usuario selecciona los "actores" deseados, incluyendo las estructuras de control, las coloca en el escenario y las interconecta para indicar el flujo de información de uno a otro formando así una red en el escenario. Dicho escenario puede luego ser almacenado o encapsulado como un actor único o generar a partir de él una aplicación paralela independiente para arquitecturas distribuidas o de memoria compartida. De manera anexa a la red de actores se proporciona una arquitectura de ejecución esto es una descripción de la arquitectura para la cual se generara el sistema final, esto es extremadamente útil para generar sistemas distribuidos ya que el sistema al conocer la estructura física de la red en la cual será ejecutada, así como el costo de comunicación entre cada uno de los elementos es capaz de generar el sistema final más adecuado; algo similar es necesario para maquinas de memoria compartida, donde lo más importante es conocer el número de procesadores que comparten la memoria y que están dispuestos a ejecutar nuestros procesos. El lenguaje visual propuesto extiende el paradigma básico de flujo para convertirlo en un ambiente de diseño más poderoso. Los datos y el control del flujo del programa se realiza a través de actores de control de flujo como bifurcación, ciclos condicionales, transiciones, etc. Las subrutinas o procedimientos están disponibles gracias al soporte de grafos jerárquicos. La creación de variables puede ser inherente al nodo o al escenario, esto es podemos tener variables tanto locales como globales. En resumen, podemos desarrollar de manera interactiva programas de manera visual con solo combinar "actores". Fig. 1. Cada elemento contiene ciertos componentes que proveen un poco de

información sobre ella; en su mayoría son botones que se emplean para realizar alguna operación sobre el elemento. Los distintos elementos gráficos que conforman el lenguaje visual comparten los siguientes componentes: conexiones de entrada y salida que le permiten al actor relacionarse con su entorno; panel de control, utilizado para especificar valores al conjunto de atributos particulares del actor; conexión de control**,** para la interconexión de elementos de control de flujo adicionales a los primitivos (transición) tal es el caso de un actor Semáforo.
La figura 1 muestra a los actores disponibles, mismos que podemos agrupar en cuatro categorías.

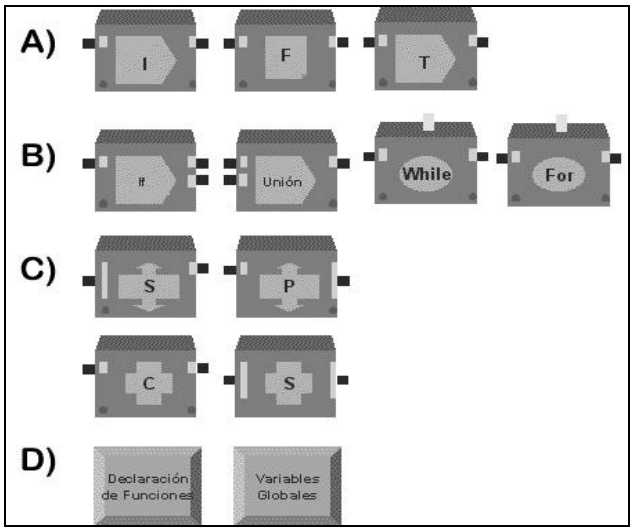

Figura1. Clasificación de Actores.

a) Actores primitivos. Como instrucciones, funciones y transiciones.
b) Actores modelo de control. Como bifurcaciones, uniones, ciclos, etc.
c) Actores modelo de Paralelismo. Paralelización, sincronización, candados y semáforos.
d) Espacios de Declaración. Los dos últimos elementos enumerados no corresponden propiamente a un actor sino a un espacio en el escenario, estos son los nodos para declaración de variables o código (funciones textuales).

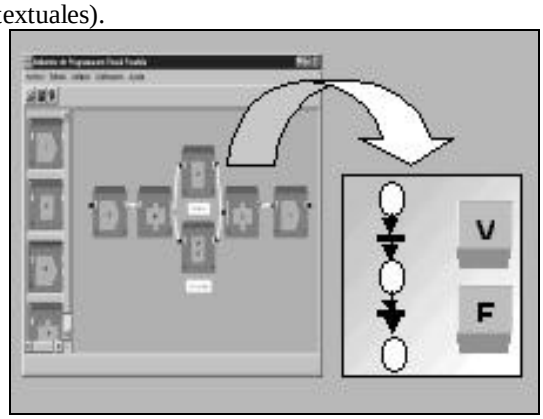

Figura 2. Modelo Subyacente.

El modelo subyacente de un escenario, con espacios y actores, se construye con la semántica operacional del lenguaje. Una vez que el programador definió un escenario, debemos generar una Red de Petri Jerárquica equivalente la cual definirá la semántica del sistema y a partir de ésta se realizara el análisis de dependencias, optimización del paralelismo y detección de errores estructurales. Figura 2.

Estructura del modelo subyacente asociado.

Para cada elemento se tienen definidas sus producciones estructurales, que no son otra cosa que el conjunto de reglas de producción que describen sus componentes y su modelo asociado equivalente como RdPJ; de manera que tomando el programa hecho con el conjunto de elementos definidos podamos convertir a cada uno de sus componentes en una pequeña RdPJ y por ende a todo el programa. El programa que se construye tiene pues una interpretación. Cada bloque constituye una subred delimitada y cada enlace entre bloques tiene asociada una transición en la RdPJ resultante. Nótese que la RdPJ como modelo subyacente del programa visual, nunca es manipulada directamente por el programador. Los modelos subyacentes se generan acorde a las siguientes definiciones individuales.

Actores primitivos. Los actores primitivos tienen una equivalencia directa a nodos de RdPJ, por ejemplo los actores instrucción y función que son actores con una serie de instrucciones asociadas o una llamada a una función definida en un lenguaje textual o bien un nodo conteniendo una subred de Petri. También Se definen como elementos semánticos nodos que de manera interna empleamos en la definición de otros elementos propios del lenguaje, nodos que no representan ninguna acción y se emplean primordialmente como conectores, por ejemplo los nodos inicio, nodos fin y nodos nulo. Estos actores pueden o no ser duplicables esto es una estructura como la que se muestra en la figura 3a puede ser representada como se muestra en 3b, donde el valor de la variable local replicación establece el número de replicas. Los nodos que no permiten duplicación se establecen con el fin de controlar la consistencia de la concurrencia, por ejemplo un nodo instrucción para E/S de datos deberá ser NO duplicable.
Existen actores, como candado y semáforo, que si bien es cierto se corresponden directamente con nodos lugar dentro de la red de Petri, tienen asociado un mayor poder semántico que el resto de los elementos ya que son estructuras propias de un lenguaje paralelo y por tanto sólo pueden ser utilizados dentro de una estructura de paralelización. El **candado** es un nodo instrucción donde el listado de instrucciones es bloqueado durante su ejecución paralela. Mientras que el **semáforo** representa una estructura semáforo, y sus operaciones básicas asociadas, el conjunto de ligas de entrada son operaciones V y el conjunto de ligas de salida son operaciones P. Este elemento puede ser inicializado con un número "N" de tokens, es el único elemento que puede

tener más de un token, sin embargo su existencia esta controlada por las reglas propias de la estructura.

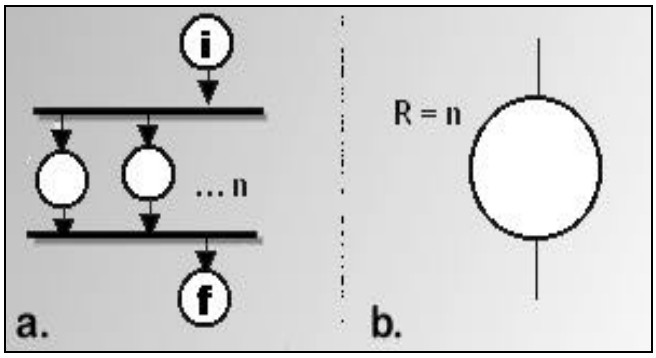

Figura 3. Duplicabilidad.

Actores modelo. A partir de los elementos primitivos, propios de una RdPJ generamos los siguientes modelos para los actores del lenguaje visual propuesto. Los actores modelo agrupan a los nodos de control de flujo y a las estructuras de programación paralela.

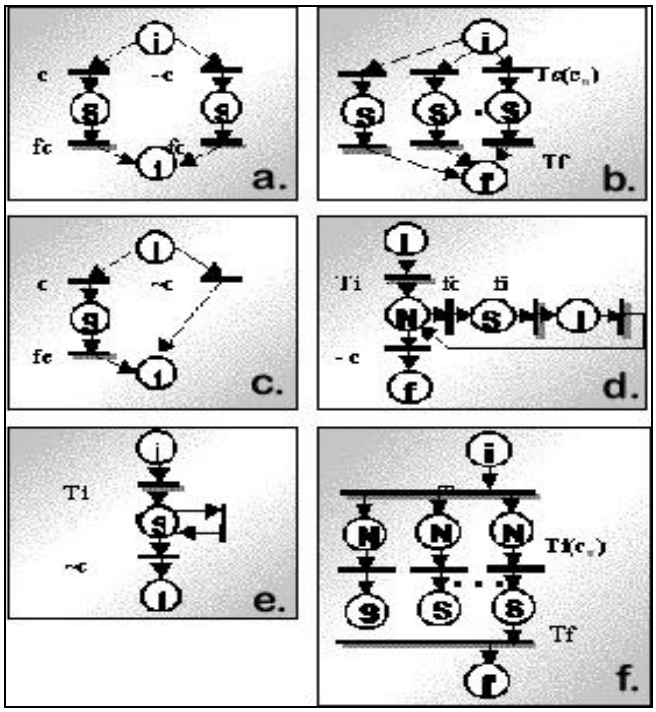

Figura 4. Modelos Subyacentes

Los **actores de control de flujo** incluyen por una parte los actores bifurcación y unión en su conjunto representan la instrucción de decisión más simple, equivalente a la sentencia If de los lenguajes de alto nivel convencionales. Una generalización de estos es la bifurcación múltiple que indica una decisión, donde el flujo de ejecución tiene diferentes caminos dependiendo del valor que tenga la variable asociada. De igual forma se tienen consideradas las estructuras clásicas de iteración cuantitativa y condicional.

Los **actores de paralelización** representan de manera explícita el paralelismo, se utiliza esta estructura, donde se indica que se ejecuten en paralelo todos los procesos que cumplan su condición asociada, estos procesos se encierran entre los actores paraleliza y sincroniza que equivalen a las transiciones superior e inferior de la figura 4f. El número de subredes definidas entre estos elementos no debe exceder el número de procesadores disponibles. El actor sincroniza o barrera permite reunificar la ejecución del programa de manera serial, esto es, terminar con los hilos o procesos ligeros que fueron creados por un actor paraleliza.

Revisión estructural.

A partir del programa dado por el usuario se realiza un análisis estructural de la red de Petri en busca de ciclos o estructuras donde se pueda aplicar una mejora estructural del sistema.

Para demostrar que todos los programas generados con el lenguaje visual propuesto son correctos, se prueba que las RdPJ subyacentes al lenguaje visual satisfacen ciertas propiedades. El análisis de estas propiedades se realiza utilizando el algoritmo para generar el grafo de alcanzabilidad, que a pesar de que puede tener complejidad NP, las RdPJ que se analizan son relativamente pequeñas por lo que no representan mayor problema. Existen muchas propiedades de las RdPJ, cada una de las cuales busca validar, definir o encontrar un comportamiento valido. Sin embargo, solo algunas propiedades se aplicaran en el análisis que se define, las propiedades se eligieron por la importancia y estrecha relación que tienen con las propiedades deseables en un sistema paralelo. El análisis del árbol de cobertura permite deducir las propiedades dinámicas de una red de Petri [Landweber 78]. Así pues sea T el árbol de cobertura de la red de Petri (N, Mo) las propiedades a considerar son alcanzabilidad, acotamiento, seguridad y vivacidad.

Alcanzabilidad. Existen lugares específicos que se deben alcanzar para asegurar que el programa (RdPJ) termina de correctamente.

Acotamiento. El número máximo de marcas que se pueden acumular en cada nodo será unitario, considerando el caso especial del actor semáforo cuyo acotamiento es dinámico y controlado.

Seguridad. Durante la ejecución de la RdPJ asociada al programa se debe asegurar que permanece como RdPJ binaria (segura). Aunque se puedan presentar flujos de ejecución paralelos, estos estan formados por flujos secuenciales, ya que no se permite el anidamiento de flujos paralelos, de tal forma que cada flujo secuencial es binario, y se garantiza que al terminar el flujo paralelo existirá un solo token.

Libre de bloqueos. No deberá haber nodos "muertos".

Vivacidad. Nos interesa el grado 1 de vivacidad. Una red de Petri será *1-viva*, si t puede ser disparada al menos una vez en alguna secuencia de disparo Una RdP 1-viva será condición suficiente de vivacidad en la búsqueda de

propiedades, por lo tanto, si una RdP es por lo menos 1-viva, esta libre de bloqueos y existe un marcado final valido. La aplicación de esta propiedad en el análisis busca asegurar que la RdP es Binaria.
La validez semántica de los actores definidos se analiza al verificar estas propiedades en sus modelos subyacentes, utilizando el método descrito.

Optimización del paralelismo.

Posterior al análisis de la RdPJ puede recurrirse a la reducción del modelo, con lo que se logra una mayor eficiencia en el uso de recursos. En principio, se deja de lado las consideraciones que conlleva la dependencia de datos y se ve exclusivamente la estructura del modelo como RdPJ, una vez descrita y analizada una posible opción de reducción se procede a enumerar las restricciones propias del uso de datos e inferencias. Las operaciones empleadas para transformar la RdPJ, descritas a continuación, preservan las propiedades descritas [Murata 89].
Fusión de lugares (L) y/o transiciones (T). Dados dos nodos L o T interconectados por un nodo T o L (con 1 entrada y 1 salida) se puede sustituir esos tres componentes por uno solo nodo L o T siempre y cuando: el nodo intermedio, sea un nodo nulo. Ninguno de los nodos L involucrados puede ser un nodo Candado o Semáforo.
Paralelización de caminos secuenciales. Encontrar recorridos secuenciales y expresarlos como paralelizaciones en muchos caso es factible, un algoritmo para tal efecto se describe en el trabajo de tesis que sustenta este articulo.

Generación de código fuente.

Dado que un programa compilado cualquiera siempre es más rápido y eficiente que cualquier código que se ejecuta bajo un ambiente controlado o maquina virtual, el siguiente paso luego de definir un lenguaje visual y transformar esa representación visual a RdPJ para su análisis es definir la estrategia para llevar esa modelo de sistema a un código paralelo compilable. El proceso a realizar será generar código C textual utilizando bibliotecas existentes para sistemas de memoria compartida o bien para sistemas distribuidos con el soporte MPI [LAM 95]. Se han predefinido zonas de generación de código, con el fin de controlar en que posiciones del programa estaremos escribiendo código, la necesidad de zonificar el programa es debido a las diferencias de paradigma entre un lenguaje visual multidimensional y la de un lenguaje textual unidimensional con un enfoque totalmente secuencial; aunque el código se vaya a ejecutar en un sistema multiprocesador. Existen ciertas estructuras visuales que afectan de manera inmediata diferentes zonas del programa. Figura 5.

- **Zona de Declaración Global (ZG).** Para declarar todas las variables globales y constantes globales que el programa necesita.
- **Zona de Inicialización (ZI).** Para agrupar todas las iniciaciones de variables globales que se indiquen ya sea por indicación explícita del usuario o por necesidades internas del sistema .
- **Zona de declaración de funciones (ZF).** Todo el código de las funciones definidas por el usuario.
- **Zona de declaración de funciones temporales(ZT).** Algunas veces será necesario definir funciones temporales por necesidades de implementación, así como para generar el código correspondiente de las funciones que representan las subredes RdPJ
- **Zona del Programa Principal (ZP).** Aquí se escribe todo el código que formara el cuerpo del programa principal.
- **Zona terminal de la Zona ZP (ZD).** Forma parte de la zona ZP, sin embargo todo el código escrito en esta zona será incluido al final del cuerpo del programa principal.
- **Zona Actual (ZA).** La escritura de código será controlada por un apuntador de generación que indica en que zona estamos escribiendo código, por lo que se le llama zona actual a la zona donde se encuentre dicho apuntador.

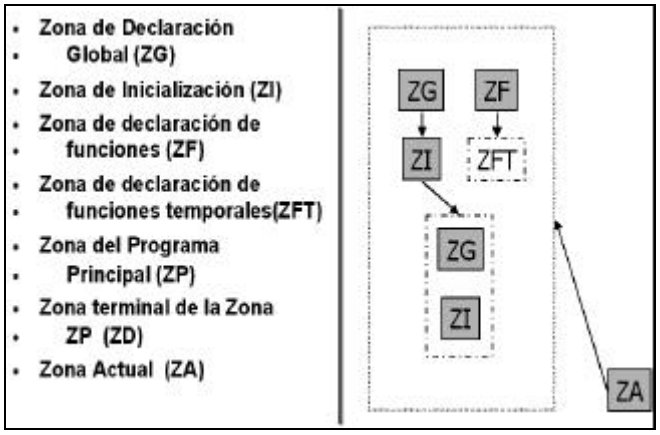

Figura 5. Generación de Código.

Los actores primitivos con equivalencia directa a nodos lugar o transición, así como los espacios de declaración generan código de manera directa en las zonas correspondientes. Los actores modelo abarcan a las estructuras de control, que representaran de manera directa o indirecta a un elemento de un lenguaje textual por lo que generan líneas de código sobre ZA directamente. Igualmente incluyen a las estructuras paralelas las cuales requieren mayor consideración durante la generación de código ya que modifican o colocan código en diversas secciones del programa. Esto es obvio si consideramos que al transformar estas estructuras gráficas en código de programación textual, estamos transformando un modelo multidimensional en un modelo totalmente secuencial, lo estamos aplanando. El uso de los elementos del lenguaje textual así como la declaración de variables auxiliares o propias del elemento es total responsabilidad del generador de código y resulta en su totalidad imperceptible

para el usuario. Utilizaremos como ejemplo una estructura paralelización para demostrar el proceso; para una información más detallada de las instrucciones utilizadas consulte[LAM 95] [SGI 95] y el algoritmos de aplanado se describe a detalle en el trabajo de tesis que sustenta el presente articulo. Entre otras cosas se crea una definición de función temporal en ZT para anexar el código requerido. En ZG se declaran variables necesarias para la ejecución en paralelo. en ZI se inicializa dichas variables, en ZP se crean todos los hilos de ejecución y en ZD se destruyen dichos hilos así como las variables utilizadas para la paralelización.

ZG	#define NODOS_PARALELIZADOS N; usptr_t *zona_glb; barrier_t *mi_barrera;
ZI	Zona_glb = usinit(" /tmp/zona_99061 "); Mi_barrera = new_barrier(zona_glb);
ZT	Function mi_funcion(.. ..) { /* Código de la función a ejecutar */ barrier(barrera, NODOS_PARALELIZADOS); return(x); }
ZP	Int id=0; If(condicion1){ sproc(mi_funcion,PR_SALL,id); id++; } .. .. if(condicionN-1){ sproc(mi_funcion,PR_SALL,id); id++; } if (schedctl(SCHEMODE,SGS_FREE,0)<0) perror("Error"); mi_funcion(.. ..) Barrier (barrera,NODOS_PARALELIZADOS);
ZD	Unlink(" /tmp/zona_99061 ");

Conclusiones y trabajo futuro.

La herramienta descrita promueve y facilita el uso del cómputo paralelo y aunque herramientas similares han sido desarrolladas para propósitos diversos se ha de mencionar que las metodológicas de cada herramienta difieren y el enfoque de cada una es muy particular. Se propone una herramienta que va más allá de un conjunto de bibliotecas y que no se detiene en ser un lenguaje complejo o semi-visual. El trabajo futuro incluye la comparación del lenguaje propuesto con tecnologías existentes, realizando un análisis de rendimiento en el código y facilidad en la representación del problema.

Referencias.

[Alm 89] Almasi George y Gottlieb Alan, *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, 1989.

[Bla 91] Blather M., Glinert E., Freaking C., *Visual tools and languages: Directions for the 90´s*, In workshop on Visual Languajes, pages 89-95. IEEE, 1991.

[Chow 88] Chow C, Lam S., *Prospec: An Interactive programming environment for designing and verifying communication protocols*, IEEE Transactions on Software Engineering, 14(3):327-338, March 1988.

[Hambrush 96] Susanne E. Hambrusch, *Overview of parallel models*, Department of computer Sciences, Purde University, International Conference on Parallel Processing, 1996.

[Jahanian 88] Jahanian A., Mok A., *Modechart: A specification languaje for real-time systems.* IEEE Transactions on Software Engineering, 1988.

[Keller 82] Keller,D. *Data Flow Program Graph*, In IEEE Computer Special Issue on Data Flow Languajes. 1982.

[Kleyn 95] Kleyn E., Florian M., **A** *High Level Language for Specifying Graph-Based Languages and their Programming Environments*, Faculty of the Graduate School of the University of Texas at Austin, August 1995.

[LAM 95] Ohio Supercomputer Center, The Ohio State University, "*MPI/LAM*", December 13, 1995.

[Landweber 78] L. H. Landweber, E. L. Robertson, *Propierties of Conflict Free and Persistent Petri Nets*, J. ACM, vol 25, No 3, pp 352-364, Julio 1978.

[Maggs 95] M. Maggs, L. R. Matheson, *Models of Parallel Computation*, Hawaii International Conference on Systems Sciences (HICSS), Vol. 2, enero 1995.

[MPI 95] *The Message Passing Interface estándar*, University of Tennessee, Knoxville, Tennessee, 1995.

[Murata 89] T. Murata, *Petri Nets: Propierties, Analysis and Applications*, IEEE vol. 77, No. 4, pp. 541-580, Abril 1989.

[Peterson 81] Peterson J**.,** *Petri net theory and the modeling of systems*, Prentice-Hall, Englewood Cliff, 1981.

[Petri 66] Petri, C.A. English translation, " *Communication with Automata*", New York:Griffiss Air Force Base. Tech. Rep. RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

[SC 91] Stardent Computer Inc., *AVS Reference Manual*, 1991.

[Scotts 90] Stotts, P. David., *Graphical Operational Semantics for Visual Parallel Programming*, in Visual Languages and Visual Programming, edited by Shi- Kuo Chang. New York: Plenum Press, 1990.

[SGI 95] Silicon Graphics Inc. Reference Manual, Paralell Programming, 1995.

[Skillcorn 96] David B. Skillicorn, Domenico Talia, *Models and Languajes for parallel computation*, Computing and Information Science Queen's University, Kingston Canada, octubre 1996.