Tesis de Licenciatura:

Hacia un enfoque híbrido del model checker Zeus

Alumno:

Fernando Pablo Taboada

Libreta Universitaria: 582/99

Correo Electrónico: fernando.taboada@gmail.com

Director:

Lic. Fernando Pablo Schapachnik

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Resumen

Los sistemas de tiempo real son aquellos en los que existen restricciones de tiempo cuyo cumplimiento es esencial para su corrección. En general, están formados por varios componentes que interactúan entre sí y suelen llevar a cabo tareas críticas. El desarrollo de este tipo de sistemas es cada vez más creciente y lógicamente se requiere que tengan altos niveles de calidad. En este sentido, en las últimas dos décadas han surgido diferentes formalismos para representarlos, así como para describir propiedades sobre su comportamiento. Uno de los más exitosos ha sido el de los autómatas temporizados [AD94], nacido como extensión a la teoría de autómatas, agregando la posibilidad de incorporar restricciones de tiempo.

Los model checkers temporizados son programas que dada una especificación sobre un sistema de tiempo real y propiedades sobre este último, determinan de manera automática si éstas se cumplen o no. Esta verificación suele ser muy costosa en términos computacionales, tanto en tiempo como en espacio, lo que limita una adopción más amplia. Son ejemplos de model checkers temporizados UPPAAL [BLL95] y KRONOS [DOT96].

Es por este motivo que gran parte del desarrollo e investigación en esta área está focalizado en obtener herramientas que permitan manejar problemas de mayor tamaño de manera más eficiente. En los últimos años ha crecido el interés en desarrollar model checkers distribuidos o paralelos de manera de aumentar la escalabilidad de los mismos.

En este trabajo partimos del model checker distribuido Zeus [SCH02] y proponemos un esquema híbrido (además de distribuido, paralelo) en donde se busca aprovechar la presencia de arquitecturas con más de un procesador con memoria compartida.

2004

Índice de contenido

Índice de Figuras

Índice de Tablas

Índice de Algorítmos

Capítulo 1

Introducción

Los sistemas de tiempo real son aquellos en los que existen restricciones de tiempo cuyo cumplimiento es esencial para su corrección. En general, están formados por varios componentes que interactúan entre sí y muchas veces su ejecución no termina. Además, suelen llevar a cabo tareas críticas, en las que una falla podría resultar en grandes pérdidas materiales o hasta poner en peligro vidas humanas. El desarrollo de este tipo de sistemas es cada vez más creciente y lógicamente se requiere que tengan altos niveles de calidad. En este sentido, en las últimas dos décadas han surgido diferentes formalismos para representarlos, así como para describir propiedades sobre su comportamiento. Uno de los más exitosos ha sido el de los autómatas temporizados [AD94], nacido como extensión a la teoría de autómatas, agregando la posibilidad de incorporar restricciones de tiempo.

Los model checkers temporizados son programas que dada una especificación de un sistema de tiempo real y propiedades sobre este último, determinan de manera automática si éstas se cumplen o no. Esta verificación suele ser muy costosa en términos computacionales, tanto en tiempo como en espacio, lo que limita una adopción más amplia. Son ejemplos de model checkers temporizados UPPAAL [BLL95] y KRONOS [DOT96].

Es por este motivo que gran parte del desarrollo e investigación en esta área está focalizado en obtener herramientas que permitan manejar problemas de mayor tamaño de manera más eficiente. En los últimos años ha crecido el interés en desarrollar model checkers distribuidos o paralelos de manera de aumentar la escalabilidad de los mismos.

En este trabajo partimos del model checker distribuido Zeus [SCH02] y proponemos un esquema híbrido (además de distribuido, paralelo) en donde se busca aprovechar la presencia de arquitecturas con más de un procesador con memoria compartida.

A continuación daremos algunas definiciones que servirán de base para el resto del trabajo y que formalizaremos en el capítulo siguiente. Tomaremos como base la introducción de [Yov97]. En el Capítulo 3, comentaremos el estado del arte, en el 4 la arquitectura conceptual de Zeus, en los dos siguientes nos concentraremos ya en el desarrollo del model checker paralelo y distribuido tanto a nivel conceptual como de implementación y en el capítulo 7 mostraremos los resultados obtenidos, finalizando con las conclusiones.

Autómata temporizado

Un autómata temporizado es una máquina de estados finita, dotada de un conjunto finito de relojes; dichos relojes son variables reales que representan el tiempo transcurrido entre la ocurrencia de eventos.

Los relojes de un autómata temporizado están sincronizados, esto implica que todos avanzan a la misma velocidad. En la evolución de los relojes pueden aparecer discontinuidades, provenientes de la ejecución de una transición en la máquina de estados, en las que a un subconjunto de relojes se les establece un nuevo valor, que generalmente es cero.

Las transiciones tienen asociadas una condición de activación, que es un predicado sobre los relojes del autómata; para que una transición en la componente discreta pueda ser tomada, los valores de los relojes deben satisfacer dicha condición de activación.

De la misma manera, las locaciones (o estados discretos) del autómata tienen asociado un predicado sobre los relojes que determina si es posible permanecer en dicha locación sin tomar ninguna transición; este predicado se denomina invariante de la locación.

Verificación algorítmica

Un conjunto de predicados sobre los relojes de un autómata temporizado impone un conjunto de restricciones sobre el

sistema de tiempo real que está siendo modelado. Analizar el comportamiento de dicho sistema consiste en la verificación de si una cierta propiedad que expresa un comportamiento se satisface o no. Estas propiedades se denominan requerimientos; luego si estos requerimientos son expresados de manera formal se puede representar el problema de la verificación de propiedades de la siguiente forma:

Dado G un autómata temporizado, R un conjunto de requerimientos y |= la relación de satisfacción: ¿vale G |= R?

La mayor parte de los investigadores del área canalizan sus esfuerzos en el desarrollo de métodos algorítmicos para resolver este problema; este enfoque es denominado usualmente model checking y consiste en la obtención de un algoritmo que responde si o no, dependiendo de si el autómata temporizado G satisface o no R.

El primer paso para poder obtener un algoritmo como el que fue mencionado en el párrafo anterior es dar el lenguaje que nos permitirá expresar de manera formal los requerimientos. El enfoque más utilizado para esto, y que ha demostrado muy buenos resultados, es el uso de lógicas para expresar las fórmulas considerando la relación de satisfacción como la característica de "ser modelo de". Este enfoque ha sido presentado en gran cantidad de trabajos, algunos de los cuales se citan a continuación: [ACD90][AH91][BLI96][HNS92].

La lógica que utiliza nuestra herramienta para expresar requerimientos se llama TCTL (*Timed Computational Tree Logic*) [ACD93], que es una extensión de la lógica CTL [EC81].

El problema de la alcanzabilidad

Este problema se expresa de la siguiente forma:

Dado un sistema y un requerimiento sobre el mismo; ¿cuál es el conjunto de estados *c* desde los que existe una ejecución que lleva al sistema, desde ellos a un estado *q* tal qué este satisface el requerimiento?.

Uno de los principales motivos para estudiar este tipo de propiedades, es que, a través de ellas, es posible modelar la no-alcanzabilidad de estados que expresan situaciones incorrectas o no deseadas. Estas propiedades son denominadas genéricamente propiedades "safety". Por otro lado, el análisis de otras propiedades se realiza usando técnicas basadas en las utilizadas para resolver el problema de la alcanzabilidad.

De la diversidad de fórmulas posibles nosotros nos concentraremos en las que expresan alcanzabilidad.

El método de cómputo para realizar el cálculo de alcanzabilidad en nuestra herramienta es *backwards*, lo que quiere decir que se parte del conjunto de estados finales y se van agregando iteración a iteración aquellos estados que alcanzan en un paso a los actuales. Esto se repite hasta que el estado inicial es alcanzado o bien no hay variaciones en los estados alcanzados.

En el próximo capítulo presentaremos formalmente estos conceptos.

Capítulo 2

Definiciones formales

En este capítulo formalizaremos algunas de las definiciones que introducimos en el capítulo anterior con menos detalle y que serán el sustento formal del resto del trabajo. Tomaremos como referencia las definiciones y propiedades de [SCH02]. Presentaremos la definición formal de autómata temporizado, la de análisis utilizando regiones temporales y los operadores básicos sobre las mismas. Luego mostraremos el algoritmo del cálculo de alcanzabilidad y la estructura de datos utilizada.

Autómata temporizado

Previo a la definición sintáctica y semántica formal de un automata temporizado daremos las definiciones de algunos conceptos que utilizaremos luego.

Relojes

Denotaremos $\mathbb{N}$ al conjunto de números naturales, $\mathbb{Z}$ al conjunto de números enteros, $\mathbb{R}$ al conjunto de números reales y $\mathbb{R}^+$ a los reales no negativos.

Definición: Relojes, valuaciones sobre relojes

Sea X un conjunto finito de variables que llamaremos relojes. Una valuación es una función que asigna un número real no negativo a cada reloj de X. Intuitivamente, una valuación de un reloj es un valor particular de éste.

El conjunto de valuaciones sobre X se denota como $\mathcal{V}_X$, y está compuesto por las funciones totales que van de X a $\mathbb{R}^+$: $[X \to \mathbb{R}^+]$.

Sea $v \in \mathcal{V}_X$ y $\delta \in \mathbb{R}^+$. Se denota $v + \delta$ la valuación definida de la siguiente forma: $(\forall x \in X)(v + \delta)(x) = v(x) + \delta$.

Definición: Asignaciones sobre relojes

Sea X^* el conjunto $X \cup \{0\}$. Una asignación es una función que asigna a cada reloj el valor de otro reloj ó 0.

El conjunto de asignaciones sobre los relojes de X se denota como Γ_X, y está compuesto por las funciones totales que va de X en X^*: $[X \to X^*]$.

Si $v \in \mathcal{V}_X$ es una valuación y $\gamma \in \Gamma_X$ es una asignación, denotaremos con $\mathcal{V}[\gamma]$ a la valuación definida como:

$$v[\gamma](x) = \begin{cases} v(\gamma(x)) & \text{si } \gamma(x) \in X \\ 0 & \text{si } \gamma(x) = 0 \end{cases}$$

Restricciones sobre relojes

Con el objetivo de poder predicar sobre los valores de los relojes, daremos una sintaxis y una semántica para expresar

restricciones sobre ellos. Nos permitirán compararlos entre ellos y con valores constantes.

Definición: Sintaxis de las restricciones sobre relojes

Dado un conjunto finito de relojes X, definiremos inductivamente la gramática del lenguaje Ψ_X que permite expresar restricciones sobre los relojes de X:

$$\begin{array}{l} \forall x, x' \in X, c \in \mathbb{N}, d \in \mathbb{Z}: \\ \psi \quad \rightarrow \quad c \prec x \quad | \quad x \prec c \quad | \quad x - x' \prec d \quad | \quad \psi \wedge \psi \quad | \quad \neg\psi \\ \prec \quad \rightarrow \quad < \quad | \quad = \end{array}$$

Sin pérdida de generalidad, a partir de este momento sólo nos referiremos a restricciones sobre relojes de longitud finita, que son las que nos interesarán.

Definición: Semántica de las restricciones sobre relojes

El lenguaje Ψ_X se interpreta sobre las valuaciones. La satisfacibilidad de una restricción $\psi \in \Psi_X$ por una valuación $v \in \mathcal{V}_X$, denotada como $v \models \psi$, se define inductivamente de la siguiente forma:

$$\begin{array}{l} \forall x, x' \in X, c \in \mathbb{N}, d \in Z: \\ v \models x < c \quad \Leftrightarrow \quad v(x) < c \\ v \models x = c \quad \Leftrightarrow \quad v(x) = c \\ v \models c < x \quad \Leftrightarrow \quad c < v(x) \\ v \models c = x \quad \Leftrightarrow \quad c = v(x) \\ v \models x - x' < d \quad \Leftrightarrow \quad v(x) - v(x') < d \\ v \models x - x' = d \quad \Leftrightarrow \quad v(x) - v(x') = d \\ v \models \psi \wedge \psi' \quad \Leftrightarrow \quad v \models \psi' \wedge v \models \psi'' \\ v \models \neg\psi' \quad \Leftrightarrow \quad v \not\models \psi' \end{array}$$

Definición: Conjunto característico

Llamaremos conjunto característico de ψ -y lo notaremos $[\![\psi]\!]$- al conjunto de valuaciones que satisfacen ψ. Formalmente:

$$[\![\psi]\!] = \{v : v \in \mathcal{V}_X \wedge v \models \psi\}$$

Sintaxis de los autómata temporizado

Podemos ahora definir los autómatas:

Definición: autómata temporizado

Un autómata temporizado G es una tupla $\langle S, X, E, A, I \rangle$ donde:

- $S = \{s_0, \ldots, s_m\}$ en un conjunto finito de estados discretos o locaciones.
- $X = \{x_0, \ldots, x_n\}$ en un conjunto finito de relojes.
- E es un conjunto finito de *etiquetas*.
- A es un conjunto finito de arcos de la forma $\langle s_i, e, cond, asig, s_j \rangle$ donde:
  - $s_i \in S$ es el origen,
  - $e \in E$ es la etiqueta,
  - $cond \in \Psi_X$ es la condición de activación,
  - $asig \in \Gamma_X$ es la asignación y
  - $s_j \in S$ es el destino.
- $I \in [S \to \Psi_X]$ (si $s \in S$ decimos que $I(s)$ es el invariante del estado discreto s).

Semántica de los autómata temporizado

Veamos como deben interpretarse los autómatas temporizados:

Definición: Sistema de transiciones etiquetado

Dado un autómata temporizado $G = \langle S, X, E, A, I \rangle$ y X' un conjunto de relojes (de especificación, su utilidad se verá más adelante) tal que $X \cap X' = \emptyset$, el significado de G es un sistema de transiciones etiquetado infinito $(Q_{X'}, \rightarrow)$ donde $Q_{X'}$ es el conjunto de estados y $\rightarrow$ es la relación de transición que se define formalmente como sigue:

- $Q_{X'} = \{(s, v) : s \in S \wedge v \in \mathcal{V}_{X \cup X'} \wedge v \models I(s)\}$ (Es decir, $Q_{X'}$ está compuesto por tuplas de estados discretos y valuaciones de relojes que satisfacen el invariante del estado discreto.)
- La relación de transición $\rightarrow \subseteq Q_{X'} \times (E \cup \mathbb{R}^+) \times Q_{X'}$ está definida por las siguientes reglas:
  - Transiciones discretas:

$$\frac{\langle s_i, e, cond, asig, s_j \rangle \in A \quad v \models cond \quad v[asig] \models I(s_j)}{(s_i, v) \rightarrow^e (s_j, v[asig])}$$

    El estado $(s_j, v[asig])$ se denomina sucesor discreto de (s_i, v) y este último predecesor discreto del primero. Intuitivamente vemos que a partir de un estado el sistema puede evolucionar a través de un arco que lo conduce a otro estado donde las valuaciones de los relojes son las mismas (i.e., no ha transcurrido tiempo) excepto por aquellos relojes que se ha vuelto a cero o se les ha asignado otro valor $(asig)$. Para que este tipo de transiciones pueda ser tomada las valuaciones del estado original deben satisfacer la condición de activación del arco y las resultantes de la asignación del arco, el invariante del estado destino.
  - Transiciones temporales:

$$\frac{\delta \in \mathbb{R}^+ \quad (\forall \delta')(\delta' \in \mathbb{R}^+ \wedge \delta' \leq \delta \implies v + \delta' \models I(s))}{(s, v) \rightarrow^\delta (s, v + \delta)}$$

    El estado $(s, v + \delta)$ se denomina sucesor temporal de (s, v) y este último predecesor temporal del primero. Intuitivamente, en ellas sólo transcurre tiempo. No hay asignaciones de valores a relojes, ni se altera el estado discreto. El único requisito para poder tomarla es que el invariante del estado discreto se debe seguir cumpliendo.

Para simplificar la notación, y cuando el contexto lo permita usaremos Q para referirnos a $Q_{X'}$.

Ejecuciones

Abordaremos el concepto de ejecución que nos permitirá hablar de una sucesión de eventos. Es el primer paso para describir comportamientos deseados y no deseados. En ésta y las siguientes secciones seguiremos además definiciones tomadas de [Yov97].

Definición: Ejecución

Sea $G = \langle S, X, E, A, I \rangle$ un autómata temporizado, $(Q, \rightarrow)$ el sistema de transición etiquetado asociado a dicho autómata. Una ejecución r de G es una secuencia infinita de aplicaciones de la relación $\rightarrow$ de la forma

$$(\forall i)(i \in \mathbb{N} \implies q_i \in Q \wedge l_i \in E \cup \mathbb{R}^+)$$
$$r = q_0 \rightarrow^{l_0} q_1 \rightarrow^{l_1} q_2 \rightarrow^{l_2} \ldots \rightarrow^{l_{i-1}} q_i \rightarrow^{l_i} \ldots$$

Luego, si $q \in Q$ se denotará $R_G(q)$ el conjunto de ejecuciones de G tal que $q_0 = q$ y $R_G = \bigcup_{q \in Q} R_G(q)$ al conjunto de ejecuciones de G.

A partir de esta noción de ejecución definiremos una posición p de una ejecución r como un par $(i, \delta) \in \mathbb{N} \times \mathbb{R}^+$ donde $\delta = 0$ si $l_i \in E$ y $0 < \delta \leq l_i$ en otro caso. Llamaremos Pos_r al conjunto de posiciones de r. Para todo $i \geq 0$ el conjunto de posiciones de la forma (i, δ) caracteriza el conjunto de estados de Q por los que la corrida r pasa mientras el tiempo fluye del estado q_i al q_{i+1}. Adicionalmente, si $q_i = (s_i, v_i)$ notaremos como $\Xi(i, \delta)$ al estado $(s_i, v_i + \delta)$.

Si $r \in R_G$, definiremos a $\tau_r : \mathbb{N} \rightarrow \mathbb{R}^+$ como el tiempo transcurrido en la ejecución r hasta alcanzar un estado dado de la corrida partiendo de q_0:

$$\begin{array}{lcl} \tau_r(0) & = & 0 \\ \tau_r(i+1) & = & \begin{cases} \tau_r(i) & \text{si } l_i \in E \\ \tau_r(i) + l_i & \text{si } l_i \in \mathbb{R}^+ \end{cases} \end{array}$$

A su vez definiremos $\tau_r(i, \delta) = \tau(i) + \delta$.

Definición: Paso de duración delta

Sea $G = \langle S, X, E, A, I \rangle$ un autómata temporizado, $(Q, \rightarrow)$ el sistema de transición etiquetado asociado a dicho autómata, $\delta \in \mathbb{R}^+$ y $q, q' \in Q$. El paso de duración δ entre los estados q y q' se define como:

$$q \triangleright^{\delta} q' \quad \Leftrightarrow \quad q \rightarrow^{\delta} q' \vee q \rightarrow^{\delta} q'' \rightarrow^{e} q' \quad \text{donde } q'' \in Q \text{ y } e \in E$$

Dados $q, q' \in Q$ y $\delta \in \mathbb{R}^+$ tales que $q \triangleright^{\delta} q'$ llamaremos estado intermedio del paso $q \triangleright^{\delta} q'$ a todo $\hat{q}$ que satisfaga $(\forall \delta')(\delta' \in \mathbb{R}^+ \wedge \delta' \leq \delta \implies q \rightarrow^{\delta'} \hat{q})$.

Como consecuencia de la definición anterior obtenemos una nueva forma de representar una ejecución $r \in R_G$:

$$(\forall i)(i \in \mathbb{N} \implies q_i \in Q \wedge \delta_i \in \mathbb{R}^+)$$
$$r = q_0 \triangleright^{\delta_0} q_1 \triangleright^{\delta_1} q_2 \triangleright^{\delta_2} \ldots \triangleright^{\delta_{i-1}} q_i \triangleright^{\delta_i} \ldots$$

Definición: Ejecucion divergente

Diremos que una ejecución $r \in R_G$ es divergente ssi $\lim_{i \rightarrow \infty} \tau_r(i) = \infty$.

Por extensión, si $q \in Q$ llamaremos $R_G^{\infty}(q)$ al conjunto de ejecuciones divergentes que comienzan en q y $R_G^{\infty} = \bigcup_{q \in Q} R_G^{\infty}(q)$ al conjunto de ejecuciones divergentes de G.

Intuitivamente, una ejecución es divergente si el tiempo no se "estanca" dentro de ella.

La importancia de ésta definición podrá ser apreciada cuando veamos la definición de autómata bien temporizado o non-zero.

Los elementos que tenemos hasta ahora nos permiten definir un orden total entre los componentes de una ejecución:

Definición: Orden total de las posiciones de una ejecución

Sea $r \in R_G$. Se define como $\ll_r \subseteq Pos_r \times Pos_r$ a la siguiente relación de orden total:

$$(i, \delta) \ll_r (j, \delta') \quad \Leftrightarrow \quad (i < j \vee (i = j \wedge \delta \leq \delta'))$$

Definición: Autómata bien temporizado o non-zero

Veamos ahora a qué llamamos autómata bien temporizado:

Sea $G = \langle S, X, E, A, I \rangle$ un autómata temporizado y $(Q, \rightarrow)$ el sistema de transición etiquetado asociado a dicho autómata. Diremos que G es non-zeno o bien temporizado ssi $(\forall q \in Q)(R_G^{\infty}(q) \neq \emptyset)$.

Es decir, un autómata está bien temporizado si en ninguna de sus ejecuciones el tiempo necesariamente "se traba".

Hasta aquí contamos con una idea bastante clara de qué es un autómata, por otro lado sabemos que contamos con la lógica TCTL [ACD93] para expresar fórmulas sobre un autómata, así que resta describir un método de cálculo que nos permita decidir si la fórmula se satisface o no.

Análisis utilizando regiones temporales

El primer obstáculo que hay que solucionar antes de realizar el cálculo es la infinitud del sistema de transiciones (Q, →), producto de la densidad de las valuaciones de los relojes.

Discretizando las valuaciones de los relojes

Presentaremos primero una relación que nos permitirá particionar las valuaciones de los relojes en una cantidad finita de clases de equivalencia.

Definición: Relación de equivalencia entre valuaciones

Sea $\widehat{\Psi} \subseteq \Psi_X$ un conjunto no vacío y finito de restricciones sobre relojes del conjunto X. Sea $C \in \mathbb{N}$ la constante más chica que es mayor o igual a $|c|$ para toda constante $c \in \mathbb{Z}$ que aparece en alguna restricción del conjunto $\widehat{\Psi}$.

Se define $\simeq_{\widehat{\Psi}} \subseteq \mathcal{V}_X \times \mathcal{V}_X$ como la relación reflexiva y simétrica más grande tal que, dados $v, v' \in \mathcal{V}_X$ se satisface $v \simeq_{\widehat{\Psi}} v'$ si y sólo si para todo $x, x' \in X$ valen

- Si $v(x) > C$ entonces $v'(x) > C$.
- Si $v(x) \leq C$ entonces:
  - $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$
  - $v(x) - \lfloor v(x) \rfloor = 0 \implies v'(x) - \lfloor v(x)' \rfloor = 0$
- Para toda restricción $r \in \widehat{\Psi}$, si $v \models r$, entonces $v' \models r$.

Llamaremos a $\simeq_{\widehat{\Psi}}$ relación de equivalencia de regiones temporales sobre el conjunto de restricciones sobre relojes $\widehat{\Psi}$. Escribiremos $[v]$ para referirnos a la clase de equivalencia (o la región) de v.

Propiedad

$\simeq_{\widehat{\Psi}}$ es una relación de equivalencia con una cantidad finita de clases. Si tomamos $n = \#X$ esta cantidad es $O(n!2^n C^n)$.

Esta propiedad nos da un marco finito para empezar a trabajar.

Propiedad

Si $v, v' \in \mathcal{V}_X \wedge [v] = [v']$, $(\forall \psi \in \widehat{\Psi})(v \models \psi \Leftrightarrow v' \models \psi)$

Esta propiedad nos dice que las valuaciones de relojes dentro de una misma clase de equivalencia son indistinguibles para el conjunto $\widehat{\Psi}$ de restricciones sobre relojes.

Propiedad

Dado un autómata temporizado G, sea $\widehat{\Psi}_G$ el conjunto de restricciones sobre relojes que aparecen en G. Dadas $v, v' \in \mathcal{V}_X$ tal que $v \simeq_{\widehat{\Psi}_G} v'$ se cumple que:

- Para toda $asig \in \Gamma_X$, $v[asig] \simeq_{\widehat{\Psi}_G} v'[asig]$.
- Para todo $\delta \in \mathbb{R}^+$, $(\exists \delta' \in \mathbb{R}^+)(v + \delta \simeq_{\widehat{\Psi}_G} v' + \delta')$.

Esta propiedad nos dice que si tomamos como origen un estado del autómata que se encuentra en una región determinada y nos movemos por una arista discreta o temporal terminaremos en la misma región en la que estaríamos si hubiésemos partido de cualquier otro estado dentro de la misma clase de equivalencia.

Discretizando el sistema de transiciones

Extenderemos ahora la relación de equivalencia $\simeq_{\widehat{\Psi}_G}$ a los sistemas de transiciones para poder obtener una cantidad finita de regiones en éstos.

Definición: Relación de equivalencia entre estados

Dado G un autómata temporizado, $(Q, \rightarrow)$ el sistema de transiciones asociado a G y $\widehat{\Psi}_G$ el conjunto de restricciones sobre los relojes que aparecen en G se define $\simeq_{\Psi_G} \subseteq Q \times Q$ tal que para todo $q = (s, v), q' = (s', v') \in Q$ se satisface $q \simeq_{\Psi_G} q' \Leftrightarrow (s = s' \wedge v \simeq_{\Psi_G} v')$.

Escribiremos $[q]$ para referirnos a la clase de equivalencia o región de q.

Veamos qué propiedades cumple $\simeq_{\Psi_G}$.

Propiedad

$\simeq_{\Psi_G}$ es una relación de equivalencia con una cantidad finita de clases. Si k es la cantidad de clases de equivalencia en las que se particiona $\mathcal{V}_X$ y n es la cantidad de estados discretos del autómata, $\simeq_{\Psi_G}$ genera $O(kn)$ clases.

Propiedad

Dado $G =< S, X, E, A, I >$ un autómata temporizado, $\widehat{\Psi}_G$ el conjunto de restricciones sobre los relojes que aparecen en G y $q_1, q_2 \in Q$ tal que $q_1 \simeq_{\Psi_G} q_2$ se satisfacen las siguientes dos propiedades:

- Para toda $e \in E$ tal que existe $q_1' \in Q$ y $q_1 \rightarrow^e q_1'$, existe $q_2' \in Q$ tal que:
  - $q_2 \rightarrow^e q_2'$
  - $q_1' \simeq_{\Psi_G} q_2'$
- Para todo $\delta \in \mathbb{R}^+$ tal que existe $q_1' \in Q$ y $q_1 \rightarrow^\delta q_1'$, existe $q_2' \in Q$ y $\delta' \in \mathbb{R}^+$ tal que:
  - $q_2 \rightarrow^{\delta'} q_2'$
  - $q_1' \simeq_{\Psi_G} q_2'$

Predicados sobre los sistemas de transiciones

Hasta ahora contamos con la lógica TCTL para expresar propiedades sobre los autómatas. Como parte del método de cálculo, necesitaremos expresar propiedades sobre el sistema de transiciones. Para ello definiremos el conjunto $Pred$ de predicados sobre los estados del sistema de transiciones.

Definición: Sintaxis de los predicados sobre Q

Dado un autómata temporizado $G =< S, X, E, A, I >$ y $(Q, \rightarrow)$ su sistema de transiciones asociado se define el conjunto $Pred(Q)$ de predicados sobre Q de manera inductiva sobre la siguiente gramática:

$$\phi \rightarrow @ = s \mid \psi \mid \phi \wedge \phi \mid \neg\phi$$

donde $s \in S$ y $\psi \in \Psi_{X \cup X'}$

Definición: Semántica de los predicados sobre Q

Dado un autómata temporizado $G =< S, X, E, A, I >$ y $(Q, \rightarrow)$ su sistema de transiciones asociado. La satisfacibilidad de un predicado $\phi \in Pred(Q)$ por un estado $(s_0, v_0) \in Q$, denotado como $(s_0, v_0) \models \phi$, se define inductivamente de la siguiente forma:

$$\begin{array}{lll} (s_0, v_0) \models @ = s & \Leftrightarrow & s_0 = s \\ (s_0, v_0) \models \psi & \Leftrightarrow & v_0 \models \psi \\ (s_0, v_0) \models \phi' \wedge \phi'' & \Leftrightarrow & (s_0, v_0) \models \phi' \wedge (s_0, v_0) \models \phi'' \\ (s_0, v_0) \models \neg\phi' & \Leftrightarrow & (s_0, v_0) \not\models \phi' \end{array}$$

donde $s \in S$ y $\psi \in \Psi_{X \cup X'}$.

Llamaremos conjunto característico -y lo denotaremos $[\![\phi]\!]$- al conjunto de estados que satisfacen ϕ:

$$[\![\phi]\!] = \{(s, v) : s \in S \wedge v \in \mathcal{V}_{X \cup X'} \wedge (s, v) \models \phi\}$$

Veamos algunas propiedades de los predicados de $Pred(Q)$.

Definición: Especialización de un estado

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$ un estado discreto de dicho autómata, $(Q, \rightarrow)$ su sistema de transiciones y $\phi \in Pred(Q)$. Definiremos:

$$\phi_s \equiv \phi|_s^{@}$$

Propiedad

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$ un estado discreto de dicho autómata, $(Q, \rightarrow)$ su sistema de transiciones y $\phi \in Pred(Q)$. Existe $\psi \in \Psi_{X \cup X'}$ tal que $\psi \equiv \phi_s$

Esta propiedad nos dice que para todo predicado de $Pred(Q)$ existe una restricción sobre relojes que lo caracteriza.

Definición: Forma normal disyuntiva

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones y $\phi \in Pred(Q)$. Definiremos:

$$\phi_{fnd} = \bigvee_{s \in S} @ = s \wedge \phi_s$$

Propiedad

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado y $\phi \in Pred(Q)$.

$$[\![\phi]\!] = [\![\phi_{fnd}]\!]$$

Propiedad

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$ un estado discreto de dicho autómata, $(Q, \rightarrow)$ su sistema de transiciones y $\phi, \phi' \in Pred(Q)$.

$$[\![\phi]\!] = [\![\phi']\!] \Leftrightarrow (\forall s)(s \in S \implies [\![\phi_s]\!] = [\![\phi'_s]\!])$$

Los operadores $pred_t$ $pred_e$ y $pred_\triangleright$

A continuación definiremos una familia de operadores que utilizaremos directamente en el cálculo de alcanzabilidad: los predecesores temporales y los predecesores discretos. Nuestras primeras definiciones serán de operadores sobre valuaciones de relojes y luego trasladaremos estas definiciones a los sistemas de transiciones etiquetados.

Definición: El operador $pred_t$

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$, $\psi \in \mathcal{P}(\mathcal{V}_{X \cup X'})$ y $v \in \mathcal{V}_{X \cup X'}$. Se define $pred_t^s : \mathcal{P}(\mathcal{V}_{X \cup X'}) \rightarrow \mathcal{P}(\mathcal{V}_{X \cup X'})$ de la siguiente forma:

$$v \in pred_t^s(\psi) \Leftrightarrow (\exists \delta)(\delta \in \mathbb{R}^+ \implies (s, v) \rightarrow^\delta (s, v + \delta) \wedge v + \delta \in \psi)$$

Definición: El operador $pred_t$ condicionado

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$, $\psi, \psi' \in \mathcal{P}(\mathcal{V}_{X \cup X'})$ y $v \in \mathcal{V}_{X \cup X'}$. Se define $pred_t^s : \mathcal{P}(\mathcal{V}_{X \cup X'}) \times \mathcal{P}(\mathcal{V}_{X \cup X'}) \rightarrow \mathcal{P}(\mathcal{V}_{X \cup X'})$ de la siguiente forma:

$$\begin{aligned} v \in pred_t^s(\psi', \psi) \quad \Leftrightarrow \quad & (\exists \delta)(\delta \in \mathbb{R}^+ \implies (s, v) \rightarrow^\delta (s, v + \delta) \wedge v + \delta \in \psi \wedge \\ & (\forall \delta')(\delta' \in \mathbb{R}^+ \wedge \delta' \leq \delta \implies v + \delta' \in \psi' \vee \psi)) \end{aligned}$$

Definición: El operador $pred_e$

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $s \in S$, $\psi \in \mathcal{P}(\mathcal{V}_{X \cup X'})$ y $v \in \mathcal{V}_{X \cup X'}$. Se define $pred_e^s : \mathcal{P}(\mathcal{V}_{X \cup X'}) \rightarrow \mathcal{P}(\mathcal{V}_{X \cup X'})$ de la siguiente forma:

$$\begin{aligned} v \in pred_e^s(\psi) \quad \Leftrightarrow \quad & (\exists < s, e, cond, asig, s' >)(\langle s, e, cond, asig, s' \rangle \in A \implies \\ & (s, v) \rightarrow^e (s', v[asig]) \wedge v[asig] \in \psi) \end{aligned}$$

Propiedad

Los operadores $pred_t$ y $pred_e$ son monótonos.

Haremos ahora definiciones análogas a las anteriores pero sobre elementos de $\mathcal{P}(Q)$.

Definición: El operador $pred_t$ sobre Q

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $pred_t : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:

$$(s, v) \in pred_t(\phi) \Leftrightarrow v \in pred_t^s(\phi_s)$$

Definición: El operador $pred_t$ condicionado sobre Q

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi, \phi' \in \mathcal{P}(Q)$. Se define $pred_t : \mathcal{P}(Q) \times \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:

$$(s, v) \in pred_t(\phi', \phi) \Leftrightarrow v \in pred_t^s({\phi'}_s, \phi_s)$$

Definición: El operador $pred_\epsilon$ sobre Q

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $pred_\epsilon : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:

$$(s, v) \in pred_\epsilon(\phi) \Leftrightarrow v \in \bigcup_{\langle s, \epsilon, cond, asig, s' \rangle \in A} pred_\epsilon^s(\phi_{s'})$$

Definición: El operador $pred_\triangleright$ sobre Q

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $pred_\triangleright : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:

$$(s, v) \in pred_\triangleright(\phi) \Leftrightarrow (s, v) \in pred_t(\phi \cup pred_\epsilon(\phi))$$

Definición: El operador $pred_\triangleright$ condicionado sobre Q

Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $pred_\triangleright : \mathcal{P}(Q) \times \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:

$$(s, v) \in pred_\triangleright(\phi', \phi) \Leftrightarrow (s, v) \in pred_t(\phi', \phi \cup pred_\epsilon(\phi))$$

Reticulados

Definición: Orden parcial

Sea P un conjunto y $\sqsubseteq$ una relación binaria definida sobre los elementos de dicho conjunto. $(P, \sqsubseteq)$ es un orden parcial ssi $\sqsubseteq$ es reflexiva, antisimétrica y transitiva en P.

Definición: Ínfimo y supremo de un orden parcial

Dado un orden parcial $(P, \sqsubseteq)$ y un subconjunto $X \subseteq P$ se dice que $p \in P$ es una cota inferior (resp. cota superior) de X ssi $(\forall q \in X)(p \sqsubseteq q)$ (resp. $(\forall q \in X)(q \sqsubseteq p)$).

Se dice que $p \in P$ es ínfimo (resp. supremo) de X-y se lo escribe $\sqcap X$ (resp. $\sqcup X$)- ssi

- p es cota inferior de X (resp. cota superior).
- Para toda cota inferior (resp. cota superior) q de X se tiene que $q \sqsubseteq p$ (resp. $p \sqsubseteq q$).

Definición: Reticulado completo

Dado P un conjunto y $\sqsubseteq$ una relación binaria sobre los elementos de P. $(P, \sqsubseteq)$ es un **reticulado completo** si satisface que:

- $(P, \sqsubseteq)$ es un orden parcial.
- Todo $X \subseteq P$ tiene ínfimo en $(P, \sqsubseteq)$.
- Todo $X \subseteq P$ tiene supremo en $(P, \sqsubseteq)$.

Propiedad

Sea $(Q, \rightarrow)$ el sistema de transiciones asociado a un autómata temporizado y $\cdot \sqsubseteq \cdot : \mathcal{P}(Q) \times \mathcal{P}(Q)$ una relación binaria tal que $(\forall r, s \in \mathcal{P}(Q))(r \sqsubseteq s \Leftrightarrow r \subseteq s)$. $(\mathcal{P}(Q), \sqsubseteq)$ es un reticulado completo.

Definición: Orden parcial completo

Sea $(P, \sqsubseteq)$ un orden parcial. Una cadena ω de un orden parcial, es una secuencia creciente de elementos de P que satisface $q_0 \sqsubseteq q_1 \sqsubseteq \ldots \sqsubseteq q_n \sqsubseteq \ldots$.

$(P, \sqsubseteq)$ se denomina orden parcial completo si para toda cadena ω posee supremo, esto es, toda cadena $\langle q_i : i \in \omega \rangle$ tiene supremo. Escribiremos $\sqcup_{i \in \omega} q_i$ para referirnos a ese supremo.

$(P, \sqsubseteq)$ se denomina orden parcial completo con bottom si es un orden parcial completo que posee un elemento distinguido ($\bot_P$) -que llamaremos "bottom"- que satisface $(\forall q)(q \in P \implies \bot_P \sqsubseteq q)$.

$(P, \sqsubseteq)$ se denomina orden parcial completo con top si es un orden parcial completo que posee un elemento distinguido ($\top_P$) -que llamaremos "top"- que satisface $(\forall q)(q \in P \implies q \sqsubseteq \top_P)$.

Propiedad

Sea $(Q, \rightarrow)$ el sistema de transiciones asociado a un autómata temporizado.

- $(\mathcal{P}(Q), \sqsubseteq)$ es un orden parcial completo con bottom y $\bot_{\mathcal{P}(Q)} = \emptyset$.
- $(\mathcal{P}(Q), \sqsubseteq)$ es un orden parcial completo con top y $\top_{\mathcal{P}(Q)} = Q$.

Definición: Monotonía y continuidad de operadores sobre reticulados

Sea $(P, \sqsubseteq)$ un orden parcial completo y $f : P \rightarrow P$ un operador.

- f es monótono ssi $(\forall p, p' \in P)(p \sqsubseteq p' \implies f(p) \sqsubseteq f(p'))$.
- f es continuo ssi es monótono y para toda cadena $\langle q_i : i \in \omega \rangle$ de elementos de P vale: $\sqcup_{i \in \omega} f(q_i) = f(\sqcup_{i \in \omega} q_i)$.

Definición: Teorema de Knaster-Tarski [Kna28][Tar55]

Si $(P, \sqsubseteq)$ un reticulado completo y $f : P \rightarrow P$ un operador monótono entonces f tiene punto fijo. Además, se definen

- $m_P(f) = \sqcap\{x \in P : f(x) \sqsubseteq x\}$
- $M_P(f) = \sqcup\{x \in P : x \sqsubseteq f(x)\}$

como el menor y el mayor punto fijo de f sobre P, respectivamente.

Definición: Teorema del punto fijo

Dado $(P, \sqsubseteq)$ un orden parcial completo con $\bot_P$ y $f : P \rightarrow P$ un operador continuo. El menor punto fijo de f sobre P, $m_P(f)$ se puede calcular mediante la siguiente fórmula:

$$m_P(f) = \sqcup_{i \in \omega} f^i(\bot_P)$$

Alcanzabilidad

El método de cómputo para realizar el cálculo de alcanzabilidad que utiliza nuestra herramienta está basado en la búsqueda del menor punto fijo de la propiedad que se quiere verificar. Como el modelo de los autómatas temporizados (o mejor dicho los sistemas de transiciones) al ser discretizados por regiones temporales forman un reticulado completo, podemos realizar el cálculo del punto fijo en una cantidad finita de pasos. Al obtener el punto fijo obtenemos un conjunto de estados del sistema de transiciones, si en él encontramos al estado inicial (s_0, v_0) sabemos que el autómata satisface la fórmula y por ende el sistema modelado cumple el requisito que se quizo verificar.

Como dijimos en el capítulo anterior, el cálculo de alzanzabilidad en nuestra herramienta y Zeus se hace de manera *backwards;* desde el punto de vista del sistema de transiciones, partimos de una región[1] [q'] y, mediante sucesivas aplicaciones del operador $\text{pred}_\triangleright$ visitamos todos sus predecesores, los predecesores de estos y así hasta que todas las regiones desde las que es posible alcanzar [q'] fueron visitadas. Entonces el cálculo consiste en el armado de la secuencia de conjuntos de regiones $B_0 \subseteq B_1 \subseteq B_2 \ldots$ de la siguiente manera:

B_0 = [q']

1 Nos referimos a la clase de equivalencia y no a la representación de datos en regiones.

$B_{i+1} = B_i \cup pred_{\triangleright}(B_i)$

donde $pred_{\triangleright}(B_i) = \{r \mid \exists r_i \in Bi \,.\, r \rightarrow r_i\}$

Representación de datos

Matrices DBM

Como ya mencionamos, la existencia de relojes que pueden tomar valores reales genera un espacio infinito de estados; este espacio infinito, para el problema de la alcanzabilidad, puede ser discretizado representando conjuntos convexos de valuaciones de relojes como inecuaciones que contemplen un reloj o la diferencia entre dos de ellos.

Para representar estas inecuaciones, nuestra herramienta, continua utilizando la misma representación simbólica que Zeus y que la mayor parte de las herramientas del área, llamada DBM (*Difference Bound Matrix)* [DIL90]. Esta representación es dada en forma de una matriz de restricciones de la forma $x_i - x_j \sim \overline{b_{ij}}$ (donde $\sim \varepsilon \{<, \leq\}$) siendo x_i y x_j relojes y b_{ij} la restricción de diferencia que existe entre ambos. Cada DBM es llamada zona y permite representar conjuntos convexos de valuaciones de relojes. Un conjunto convexo es aquel que dados dos puntos cualesquiera del conjunto, el segmento lineal cerrado que los une está totalmente contenido en el conjunto. Las zonas son convexas por definición.

Diremos que una zona representa el conjunto de valuaciones de relojes que satisfacen las restricciones representadas en la matriz. Un conjunto de zonas es llamada región, y está dada por las DBMs de cada una de sus zonas. Las regiones permiten representar conjuntos no convexos de valuaciones de relojes en forma de unión de conjuntos convexos.

Entonces, en un instante dado cada locación del autómata tendrá asociada una región que represente el conjunto de valuaciones de relojes posibles.

Canonicidad de DBMs

Una DBM es canónica si ninguna de sus *constraints* puede ser cambiada por un menor valor sin alterar la solución que representa la matriz. Una de las formas de calcular la matriz canónica es ajustando los *constraints* por el algoritmo de caminos mínimos de Floyd-Warshall [FLO62]. Se debe representar un grafo en el que los nodos son los relojes y los arcos dirigidos contienen como peso los constraints de la diferencia entre los relojes. Entonces el algoritmo de caminos mínimos determinará entre cada par de nodos/relojes cuál es el menor valor de *constraint* posible. Nosotros trabajeremos siempre con regiones cuyas zonas son canónicas. La importancia de esto reside en que tengamos una única representación posible para una zona determinada. Notar que no existe la noción de canonicidad para regiones.

Algoritmo de alcanzabilidad

A continuación, en el Algoritmo 1 [Sch02], podemos ver el pseudocódigo del algoritmo de alcanzabilidad que efectua el cálculo de punto fijo. En cada iteración se busca incrementar la región de cada estado discreto del autómata. Para esto, para cada sucesor discreto de cada estado se amplia la región teniendo en cuenta los predecesores discretos y la región actual. Luego a la región resultante se le aplica el cálculo que tiene en cuenta los predecesores temporales. Es importante que notemos como iteración a iteración el operador $pred_e$ utiliza la diferencia de regiones para no recalcular la región completa.

```
alcanzabilidad(regiones): regiones {
  para cada estado discreto s del autómata hacer
      R_s <- obtener region correspondiente a s
      PredE <- VACIO
      para cada sucesor discreto suc de s hacer
            R_suc <- obtener diferencia region almacenada para SUC
            PredE <- PredE U  pred_e(suc, R_suc)
      PredT <- pred_t(PredE)
      DifR <- PredT - R_s
      guardar en resultado como region correspondiente a s PredT
      guardar en resultado como diferencia de region correspondiente a s DifR
  devolver resultado
```

Algoritmo 1: Algoritmo de alcanzabilidad

Algoritmo del cálculo de la diferencia de regiones

En el Algoritmo 2, podemos ver el pseudocódigo del algoritmo de diferencia de regiones. La región resultado de hacer la diferencia entre r_1 y r_2 está dada por la intersección de r_1 con el complemento de cada una de las zonas de r_2 .

Para calcular la intersección de r_1 con z_2', el complemento de una zona z_2 de la region r_2, se debe hacer la unión de los sucesivos resultados de intersecar cada zona z_1 de r_1 con z_2' o lo que es lo mismo la diferencia entre cada z_1 y z_2.

Esta diferencia entre zonas no siempre resulta en una zona, sino que en una región.

```
diferencia_region(region1, region2): region
  si region1 no vacia hacer
     si region2 no vacia hacer
        nueva_region <- region1
        para cada zona z de region2
            nueva_region <- nueva_region ∩ compl(z)
        resultado <- nueva_region
     sino
        resultado <- region1
  sino
      resultado <- region vacía
  devolver resultado
```

Algoritmo 2: Cálculo de la diferencia de regiones: region1 - region2

Capítulo 3

Estado del arte

En las últimas dos décadas ha habido numerosos avances en el campo del model checking distribuido y paralelo. La mayoría de estos avances han abordado el problema de la explosión de estados, y una de las técnicas ha sido la paralelización de los model checkers.

En general en todos los casos se busca mejorar la escalabilidad distribuyendo el trabajo entre nodos y/o procesadores y disminuyendo la comunicación entre ellos. Sin embargo, los casos temporizado y no temporizado difieren significativamente en las estrategias de distribución y paralelismo, ya que involucran diferentes estructuras de datos. La estructura de datos más utilizada en los temporizados ha sido la *Difference Bound Matrix* (DBM), mientras en el caso de los no temporizados, el *Binary Decision Diagram* (BDD) [Ake78].

Los BDD son una estructura de datos para representar funciones booleanas y consisten en un grafo dirigido con raíz y acíclico con nodos de decision y dos nodos terminales, uno correspondiente a un resultado verdadero, y otro a falso. Cada nodo de decisión corresponde a un a variable booleana y tiene dos hijos: uno correspondiente a la asignación a la variable de 0 y otro a 1. En el model checking con BDD se crea el grafo que representa el conjunto de estados que satisfacen la fórmula y se realizan sobre él operaciones de disyunción, conjunción, negación, existencialidad y universalidad.

Un aspecto importante es el que tiene que ver con la disponibilidad de equipos que ofrece la industria. Hasta hace poco tiempo los equipos disponibles que ofrecían paralelismo eran únicamente clusters con memoria distribuida y se creía que los avances iban a a ir en esta dirección. En los últimos años notamos una modificación en esta tendencia, vemos que la industria está poniendo énfasis en el desarrollo de equipos multiprocesador con memoria compartida y en los llamados *distributed-shared memory* (DSM), equipos multiprocesador con memoria distribuida, pero en la que desde el punto de vista del programador se utiliza un único espacio de memoria, abstrayéndose de la distribución de la misma.

Por este motivo, es que actualmente hay pocos trabajos en el campo del model checking paralelo en comparación con el distribuido y hay aún menos casos, si buscamos enfoques híbridos al problema como el que estamos dando en este trabajo. De hecho, hasta la realización de esta tesis no encontramos ningún caso de desarrollo híbrido como el que estamos planteando.

Uno de los primeros trabajos en model checking paralelo fue el de Stern y Dill en el año 1997, que plantearon el problema de la alcanzabilidad no temporal, con una versión paralela del verificador Murφ [SD97]. Ellos particionan los estados visitados entre los procesadores con una función de *hash*. El algoritmo propuesto alcanza una performance cercana a la lineal y funciona en arquitecturas con memoria distribuida y RET[2].

Uno de los trabajos más recientes fue el de Cornelia Inggs [CB06][CB02], quien propone un model cheker paralelo y temporizado para la lógica CTL* en arquitecturas de memoria compartida. En su trabajo proponen un algoritmo de exploración de estados que está implementado como la composición paralela de n procesos exploradores de estados donde cada uno tiene una cola compartida y otra privada en la cual se almacenan los estados no visitados. Estas colas permiten balancear la carga ya que cuando un proceso no tiene trabajo puede "robar" de la cola de otro proceso. Los estados visitados son guardados en una tabla de *hash* global. Este algoritmo de alcanzabilidad paralelo luego lo aplican al desarrollo de un model checker paralelo basado en teoría de juegos y *Hesitant Alternating Automata (HAA)* [BER95] para representar las fórmulas de CTL*. En el trabajo reportan buenos resultados e identifican como problemas en este tipo de arquitectura el *overhead* de sincronización y *false sharing*, que ocurre cuando dos procesos intentan escribir diferentes palabras en la misma línea de cache. Esto último lo han solucionado utilizando *mutex*[3].

Otro caso de model checker paralelo y temporizado es el trabajo de Martin Lange [LL05] en la lógica de punto fijo con chop, una lógica modal capaz de definir propiedades no regulares, como por ejemplo: en todos los caminos el número de a's excede el número de b's o algo sucede en todos los caminos al mismo tiempo. En el trabajo presentan un model checker simbólico basado en BDDs. Dado que en la lógica de punto fijo con chop la semántica de una fórmula es una función, aprovechan esto para hacer en paralelo las dos partes que componen una fórmula booleana. Implementan una versión en Glascow Parallel Haskell (GPH) accediendo a una libreria de BDD en C. Reportan que no obtuvieron buenos resultados, por ejemplo un incremento de 1,2 en 4 procesadores. Esto lo atribuyen a que el algoritmo genera un

2 Red de estaciones de trabajo
3 Hablaremos sobre los mutex en el capítulo de implementación

paralelismo muy irrregular dividiendo en una gran cantidad de threads.

En cuanto a model checkers paralelos y no temporizados, Benedikt Bollig [BLW01] propone distintas variantes de un verificador paralelo para el *alternation-free mu-calculus*, un fragmento del *u-calculus* [Koz83]. Este fragmento permite expresar tanto propiedades de *safety* como de *liveness* e incluye a la lógica CTL. El model checker está planteado en términos de un juego de 2 jugadores, y se presenta un algoritmo de coloreo en paralelo para el grafo del juego que corresponde al problema a verificar. Lo han implementado en C++ y en Haskell, obteniendo buenos resultados en el sentido de resolver problemas que con la versión secuencial no.

Otro caso paralelo y no temporizado es el trabajo de Sahoo [SJI05] en el que se propone un algoritmo paralelo basado en BDD, en el que al clásico enfoque de particionar el espacio de estados entre los procesadores agrega dos heurísticas llamadas *early comunication* y *partial comunication.* La primera, trata de comunicar los estados entre procesadores luego de un punto fijo local y la segunda de envíar a algún procesador libre una parte del trabajo de otro que esté ocupado. Más allá de esto, identifican posibles inconvenientes en implementaciones multithread, como por ejemplo las fallas de *cache*, que se dan cuando se cambia la asignación de un thread de un procesador a otro, entonces el procesador tiene que *cachear* toda la información del thread nuevamente.

Si hablamos de model checkers distribuidos, tenemos una versión de UPPAL [BHV00][Beh05] que hace análisis de alcanzabilidad sobre BDD de manera *forward.* Muestran resultados sobre arquitecturas multiprocesador con memoria distribuida y RET[4], obteniendo en los primeros mejoras supralineales y lineales en los segundos.

Finalmente tenemos una versión distribuida de SPIN [LS99][BBS01], un model checker no temporizado para alcanzabilidad y LTL. La nueva versión les permitió hacer terminar problemas que antes no finalizaban, pero para los problemas chicos encuentraron que los tiempos aumentaban.

Si bien hemos encontrado enfoques paralelos, alguno de ellos exitosos, nos llama la atención que en varios de ellos se han encontrado con numerosos inconvenientes en las implementaciones con memoria compartida de manera que queda evidenciado que no es trivial conseguir buenos resultados en este tipo de arquitecturas.

4 Red de estaciones de trabajo

Capítulo 4

Motivación

En este capítulo veremos la arquitectura conceptual de Zeus y las limitaciones que sirvieron de motivación para iniciar este trabajo.

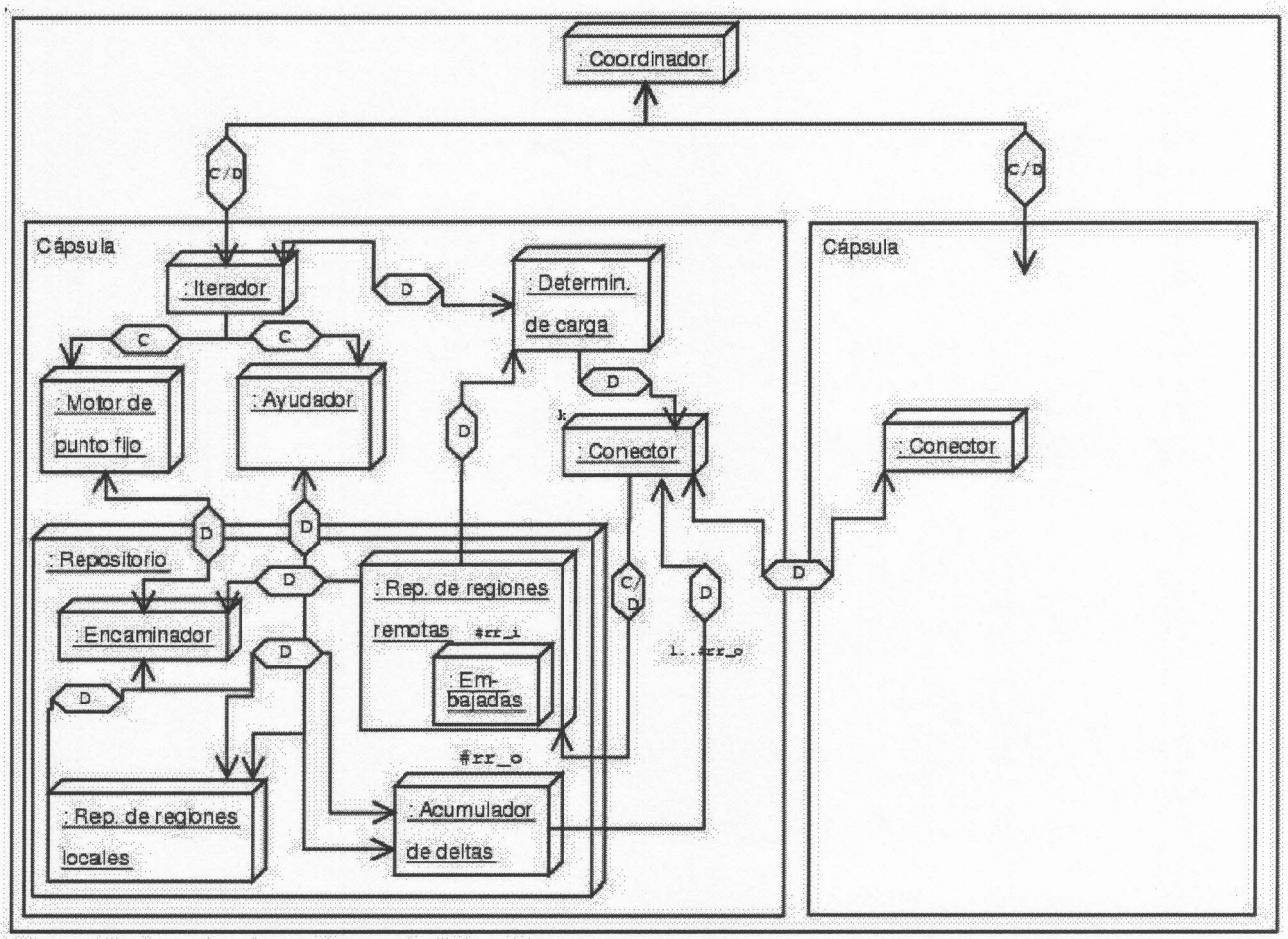

Figura 1: Arquitectura Conceptual de Zeus

La arquitectura de Zeus fue concebida teniendo como principio el diseño para el cambio, de manera que fuera simple probar diferentes aspectos como sincronización, intercambio de regiones y balance de carga.

Cada procesador en el cálculo distribuido es llamado *cápsula*. También llamamos cápsula a los procesos que corren en cada procesador junto a sus estructuras de datos y componentes.

La arquitectura de Zeus puede ser más fácilmente entendida si nos centramos en el componente del *motor de punto fijo*, el que, como su nombre lo indica, realiza el cálculo propiamente dicho. Este lee y escribe regiones del *repositorio*, las que pueden pertenecer a la *cápsula local* o a otras *cápsulas*. El *motor de punto fijo* ignora esa distribución, de manera que trata indistintamente a unas regiones de otras, excepto porque escribe resultados sólo para las regiones locales. El *encaminador* es el componente que conoce qué estados discretos son locales y cuáles remotos y en base a esa información distribuye los pedidos al *repositorio local* o *remoto*. Cuando al *encaminador* le llega un pedido de escritura, lo graba en el *repositorio local* en un *acumulador de deltas*. Hay tantos de estos como regiones remotas; y hacen de colas en las que se guardan regiones que son de interés para otras cápsulas. La razón por la que existen los *acumuladores* tiene que ver con un aspecto técnico: conviene guardar el cálculo de la diferencia en lugar de recalcularlo cuando se necesita.

Dentro de cada *cápsula* existe una *embajada* por cada *cápsula* vecina. Así como los *acumuladores* facilitaban la fase de envío de información entre *cápsulas*, las *embajadas* facilitan la recepción de información de otras.

Una *cápsula* A tiene un *conector A-B* si y sólo sí existe un eje entre una locación en *A* y otra en *B*. Los *ayudadores* realizan tareas de ayuda a la propia *cápsula* o a otra, tales como compresión de datos.

El *coordinador* inicia el proceso, particiona el grafo, distribuye la carga entre las cápsulas y determina cuando el punto fijo global es alcanzado.

A continuación en el Algoritmo 3 podemos ver el algoritmo de alcanzabilidad en Zeus, el cúal se ejecuta dentro del motor de punto fijo. En la figura Figura 2 mostramos la interfaz que ofrece el repositorio al motor de punto fijo.

Para más referencia acerca del desarrollo y arquitectura de Zeus referirse a [SCH02].

```
motor_punto_fijo.iterar(): nat {
  cantidad_cambios = 0
  para cada s en repositorio.estados_discretos_locales() hacer
      R_s <- repositorio.obtener_region(s)
      PredE <- VACIO
      hay_nueva_region <- FALSE
      para cada sucesor discreto suc de s hacer
            hay_nueva_region_suc <- repositorio.cambio_region?(suc)
            si hay_nueva_region_suc hacer
                  R_suc <- repositorio.obtener_dif_region(suc)
                  PredE <- PredE U  pred_e( R_suc, inv(suc), reset_clocks(suc))
            hay_nueva_region <- hay_nueva_region v hay_nueva_region_suc
      si hay_nueva_region hacer
            PredT <- pred_t(PredE)
            DifR_s <- PredT - R_s
            si DifR_s no VACIO hacer
                  cantidad_cambios++
                  repositorio.almacenar_dif_region(s,DifR_s)
                  repositorio.almacenar_region(s, R_s U PredT)
  repositorio.fin_iteracion()
  devolver cantidad_cambios
```

Algoritmo 3: Algoritmo de alcanzabilidad en Zeus

repositorio.estados_discretos_locales(): conjunto(estado_discreto)

Devuelve el conjunto de los estados discretos asignados a la cápsula.

repositorio.obtener_region(s: estado_discreto): region_temporal

Devuelve la región temporal correspondiente al estado discreto s.

repositorio.cambio_region?(s: estado_discreto): bool

Indica si desde la última llamada a repositorio.obtener_dif_region(s) se produjo algún cambio en la región temporal de s.

repositorio.obtener_dif_region(s: estado_discreto): region_temporal

Devuelve la región temporal correspondiente a la diferencia entre la última disponible y la devuelta por la invocación anterior de la función.

repositorio.almacenar_dif_region(s: estado_discreto, delta: region_temporal)

Almacena delta como la región temporal correspondiente a la diferencia entre la iteración actual y la anterior para el estado discreto s.

repositorio.almacenar_region(s: estado_discreto, r: region_temporal)

Almacena r como la región temporal correspondiente al estado discreto s.

repositorio.fin_iteracion()

Notifica al repositorio de regiones que ha finalizado una iteración. Esta información es particularmente útil para manejar la liberación de recursos ya utilizados, especialmente en las embajadas.

Figura 2: Interfaz que brinda el repositorio al motor de punto fijo en Zeus

Limitación

Zeus es un model checker distribuido que aprovecha de arquitecturas con más de un nodo de procesamiento y memoria distribuida. La limitación que tiene es que no aprovecha la presencia de arquitecturas con más de un procesador y memoria compartida. Como dijimos en el Capítulo 2, cada vez hay disponibles más equipos multiprocesador y parece haber un interés en la industria en el desarrollo de este tipo de equipos. Por otro lado, la mayoría de los sistimas operativos actuales soportan multithreading.

Otra de las limitaciones tiene que ver con la distribución del trabajo: como se menciona en [SCH02], cada locación va a un único procesador y puede pasar que algunas locaciones tomen más trabajo que otra y se genere un cuello de botella en las mismas. Existe una versión distribuida de Zeus con redistribución de la carga que soluciona este problema [BOS04].

Finalmente, al dividir por locaciones no se aprovecha el caso en que halla más nodos de procesamiento que locaciones. Para un análisis más exhaustivo de las limitaciones de Zeus referirse a [BOS06].

Esto motivó el desarrollo de este trabajo buscando un enfoque híbrido, además de distribuido, paralelo, pudiendo aprovechar cualquiera de las dos arquitecturas así como también entornos en donde se dan simultáneamente las dos.

Capítulo 5

Hacia una arquitectura híbrida

Tomando como punto de partida la versión sincrónica de Zeus presentada en [SCH02], analizamos de qué manera podríamos aprovechar la presencia de más de un procesador en las cápsulas de la verificación distribuida. La función más costosa dentro del motor de punto fijo es el cálculo de la diferencia de regiones[5], que además tiene el problema de que iteración a iteración el número de zonas dentro de cada región tiende a crecer a causa del mismo cálculo de la diferencia. Esto implica que en las sucesivas iteraciones el número de zonas a procesar por región va a ser mayor y por ende el costo también. Por esto, es que decidimos paralelizar esta función.

En este capítulo veremos de qué forma transformamos la arquitectura distribuida a una híbrida, que además de aprovechar la división en nodos dentro de un cluster explota la presencia de más procesadores en los nodos del clúster que los posean.

Presentaremos el algoritmo de diferencia de regiones en Zeus, comentaremos las diferentes estrategias que contemplamos antes de elegir una y finalmente explicaremos el algoritmo paralelo del cálculo de la diferencia, la arquitectura y diseño de objetos del mismo y lo mostraremos correcto.

Algoritmo de la diferencia de regiones en Zeus

En el cálculo de la diferencia se parte de dos regiones R_1 y R_2. Como mencionamos en el Capítulo 1 las regiones son representadas como las uniones de DBM que representan cada zona. Recoredemos que la DBM de una zona contiene en sus celdas las restricciones de relojes que representa. La $celda_{i,j}$ tendrá el valor c_{ij} si y sólo sí se esta representando la diferencia de relojes $x_i - x_j < c_{ij}$.

La diferencia entre cada zona Z_1 de R_1 y cada zona Z_2 de R_2 implica tomar las matrices DBM de cada zona y para cada celda i, j hacer la diferencia de las restricciones que indican, o lo que es lo mismo, la intersección de la restricción de $Z_{1i,j}$ con el complemento de la restricción de $Z_{2i,j}$.

En el Algoritmo 4, el Algoritmo 5 y el Algoritmo 6, podemos ver el algoritmo del cálculo. En la función *diferencia_region()* si las regiones no son vacías se iteran las zonas Z_{2i} de R_2 y se llama a la función *interseccion_region_compl_zona()* para cada Z_{2i} y R_1, sobre la cuál se van haciendo las modificaciones.

La función *interseccion_region_compl_zona()* calcula la intersección entre cada zona Z_{1j} de la región parámetro R_1 y el complemento de la zona parámetro Z_{2i}. Por eso, se itera R_1 y se hace la interseccion de cada Z_{1j} contra el complemento de Z_{2i} llamando a la función *interseccion_zona_compl_zona()*.

Las regiones que se obtienen iteración a iteración van a formar el resultado final. Para eso la función *agregar_zonas()* arma una región resultado que contiene las zonas de las dos regiones. De aquí en más llamaremos a esto *append* de los resultados parciales.

5 En el Capítulo 7 veremos que el tiempo que insume el cálculo de la diferencia de regiones representa entre un 80% y un 95% del total según el caso.

```
diferencia_region(region1, region2): region
  si region1 no vacia hacer
     si region2 no vacia hacer
        nueva_region <- region1;
        para i desde 0 hasta tamaño(region2)-1 hacer
            zona = region2[i]
            nueva_region <- interseccion_region_compl_zona(nueva_region, zona);
        resultado <- nueva_region
     sino
        resultado <- region1
  sino
      resultado <- region_vacia()
  devolver resultado
```

Algoritmo 4: Cálculo de la diferencia de regiones

```
interseccion_region_compl_zona(region, zona_compl): region {
  si zona no vacia hacer
     si region no vacia hacer
        zona <- region[0]
        resultado <- interseccion_zona_compl_zona(zona, zona_compl)
        para i desde 1 hasta tamaño(region)-1 hacer
            zona <- region[i]
            aux_region <- interseccion_zona_compl_zona(zona, zona_compl))
            resultado <- agregar_zonas(resultado, aux_region)
     sino
        resultado <- region_vacia()
  sino
      resultado <- region
  devolver resultado
```

Algoritmo 5: Cálculo de intersección entre una región y el complemento de una zona

```
interseccion_zona_compl_zona(zona1, zona2): region {
  si zona1 vacia
     resultado <- zona_vacia()
  sino
      resultado <- armar_region(zona1)
      si zona2 no vacia
            para i desde 0 a nro_relojes
                  para j desde 0 a nro_relojes
                        si i <> j
                              si zona2[i,j] > 0
                                    compl <- compl(zona2[i,j])
                                    si zona1[j,i] >= compl
                                          resultado[j,i] <- compl
                                    sino
                                          devolver resultado
      devolver resultado
}
```

Algoritmo 6: Cálculo de intersección entre una zona y el complemento de otra

Posibles estrategias de paralelización

Una vez que decidimos paralelizar el cálculo de la diferencia de regiones se evaluaron distintas formas de llevarla a cabo, cada una paralelizando con distinta granularidad.

La primera opción fue dividir una de las regiones y hacer en paralelo cada zona de ésta contra toda la otra región (ver Figura 3). La segunda opción, con una granularidad más fina, fue una vez en la iteración de cada zona de una región, hacer en paralelo el procesamiento de la diferencia de esta zona contra cada una de las zonas de la otra región (ver Figura 4). En el primer caso a cada procesador se asignaba una o más zonas de una región y la otra región completa, mientras que en este caso se asigna una zona de una de las regiones y una o mas zonas de la otra región a cada procesador. Por último, como las zonas se representan con matrices se pensó en paralelizar el cálculo de la diferencia entre dos matrices de zonas a nivel celda. Esta opción representa el nivel más fino de granularidad ya que se paraleliza cada diferencia de relojes entre las dos zonas.

Si tomamos la primera opción la función a paralelizar es *diferencia_region()*, con la segunda opción *interseccion_region_compl_zona()* mientras que si nos decidimos por la tercera, *interseccion_zona_compl_zona()*.

Luego de evaluar ventajas y desventajas de las tres opciones, se optó por la segunda, con un grado de granularidad intermedio. La primera opción tenía la desventaja de que si el cálculo para algunas regiones era más costoso que para otras se iba a generar un cuello de botella en el procesamiento de las mismas. La última, al tener un nivel tan fino de granularidad presenta el inconveniente de estar agregando el overhead de dividir el problema y juntar los resultados una mayor cantidad de veces. La segunda presenta una buena división del problema al no asignar una región entera a un procesador.

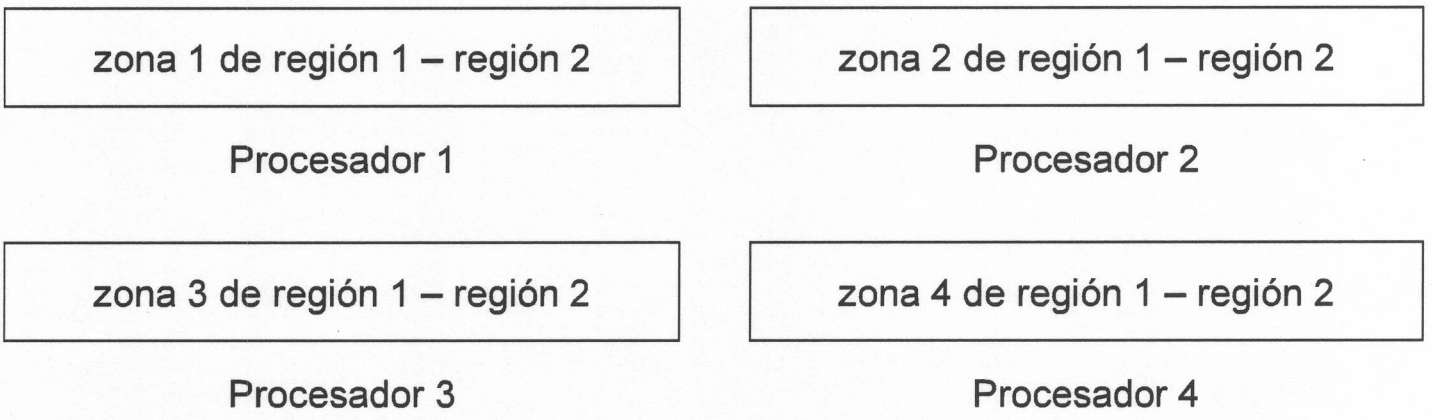

Figura 3: Ejemplo de Estrategia 1

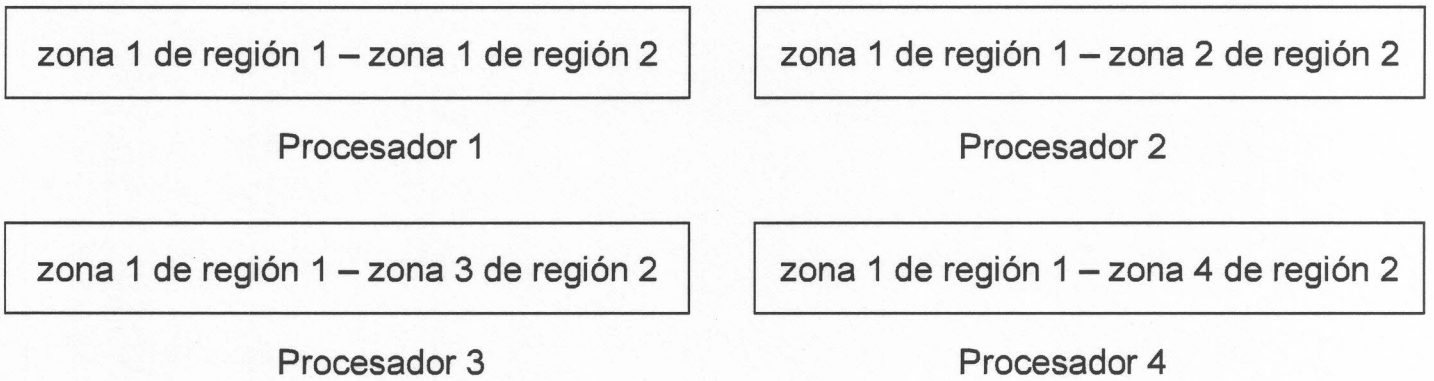

Figura 4: Ejemplo de Estrategia 2

Algoritmo paralelo de diferencia de regiones

Como ya mencionamos, al elegir la segunda opción, hay que paralelizar la función *interseccion_region_compl_zona()* de modo que ya estamos dentro de la iteración de cada zona dentro de una de las regiones. Así resulta que el trabajo que se va asignar a cada procesador es el cálculo de la diferencia de la zona en cuestión contra una "porción" de las zonas de la otra región.

Entonces comenzaremos describiendo la estrategia para paralelizar *interseccion_region_compl_zona()*, que como veremos se hace en dos fases bien diferenciadas, y finalmente el rol que cumplen el procesador principal y el resto.

De aquí en más hablaremos de threads en lugar de procesos o procesadores.[6]

Estrategia

La estrategia que utilizamos para paralelizar *interseccion_region_compl_zona()* se basa en los siguientes pilares:

1. El thread principal[7] particiona la región parámetro: divide las zonas de la región entre la cantidad de procesadores disponibles en cada cápsula. Luego invoca -en diferentes threads- un algoritmo paralelo asignando a cada uno una porción de región.

2. Cada thread ejecuta *interseccion_zona_compl_zona(R, Z)* para cada una de las zonas que le corresponde (según la partición realizada en el paso anterior) contra la zona parámetro. Además realiza el *append* de las regiones resultado de las llamadas a *interseccion_zona_compl_zona(Z1, Z2)* para armar el resultado final.

3. Cada thread agrega a la región resultado, las zonas de las regiones resultado de otros threads en la medida en que estos finalizan el cálculo mencionado en 2. Cuando un thread termina dicho calculo chequea si algún otro finalizó también y su resultado aún no fue utilizado por otro thread:

    3.a. Si no es así, el thread finaliza su tarea y queda la región resultado disponible para que otro haga la unión con su propio resultado.

    3b. Si existe disponible el resultado de otro thread, entonces procede a unir el resultado parcial propio y el del otro (*append* de los resultados). Cuando termina, nuevamente intenta repetir la operación con algún otro resultado parcial de otro thread (vuelve al punto 3).

Calculo de la diferencia en 2 fases

El algoritmo paralelo consta de 2 fases bien definidas que corresponden a los pasos 2 y 3 de la estrategia respectivamente. Entonces queda una primera fase en la que se calcula la intersección entre una zona y el complemento de la otra (*interseccion_zona_compl_zona()*) para todas las zonas que correspondan, haciendo también el append de los resultado parciales.
En la segunda fase, se busca no perder ciclos de reloj en aquellos threads que terminan antes y así ganar en paralelismo. Cuando alguno termina la fase 1 intentará hacer el append de su resultado con el resultado de algún otro que halla finalizado, evitando así hacer un append centralizado al final de todos los resultados parciales.
En el siguiente apartado, en el Algoritmo 8, el Algoritmo 9 y el Algoritmo 10 se puede ver el pseudocódigo en donde se hace efectivo las 2 fases de las que hablamos.

Rol del thread principal y del resto

---

6 En el siguiente capítulo veremos los detalles de la implementación concurrente con threads.

7 Cuando hablamos de thread principal nos referimos sólo a un concepto. Todos los threads son iguales, sólo que el llamado principal ejecuta una rutina ligeramente diferente.

Además de hacer la división de la región, el thread principal se diferencia levemente del resto en la fase 2. La diferencia radica en que el principal nunca deja su resultado parcial para que otro haga el append, sino que siempre espera el resultado de otro para hacer el append con su resultado. O sea que, el thread principal nunca finaliza[8] y, cuando todos los demás finalizan (ya habrán cedido su resultado parcial al principal o a otro), como el principal es el único que queda, va a contener el resultado final.

Arquitectura y diseño de objetos del módulo paralelo

Para incorporar el enfoque híbrido agregamos algunos componentes al motor de punto fijo de Zeus. El resto de la arquitectura se preservó intacta.

Componentes

La arquitectura consta de los siguientes componentes:

runner: conoce el número de procesadores disponibles en la cápsula. Se encarga de lanzar en algún procesador que este libre cada procedimiento que se le solicite con sus parámetros[9]. Comentaremos más sobre las responsabilidades del *runner* en el capítulo de implementación.

coordinador_particion: se encarga de dividir la región y mantener la información de la división. Por otro lado, guarda el resultado parcial de aquel thread finalizado para luego ofrecerlo a aquel thread que lo requiera, sirviendo de esta manera a la comunicación entre los threads.

partición: es el objeto que maneja el *coordinador_partición* y mantiene la información de una *partición*, tales como la región, cuáles zonas de ésta corresponde tomar, la otra zona y el resultado parcial de la diferencia correspondiente a la partición.

Interfaz del runner

```
crear_runner(nro_procesadores);
destruir_runner();
run(procedimiento, parámetros);
obtener_nro_procesadores(): int;
```

Figura 5: Interfaz del runner

Los dos primeros procedimientos de la Figura 5 son utilizados por el *coordinador* y cada una de las *cápsulas* de Zeus al comienzo y al final de la ejecución del verificador y construyen y destruyen, respectivamente, el runner. En la implementación veremos cuál es el sentido de que el *runner* se cree una sola vez para toda la verificación.

run() es utilizado por el motor de punto fijo para indicarle al runner que ejecute un determinado procedimiento (*interseccion_region_compl_zona2()*) sobre un determinado parámetro (una *partición*). El *runner* mantendrá abstracto como hace para ejecutar cada procedimiento que se le pida en un procesador libre.

El último es utilizado por el *motor de punto fijo* para saber la cantidad de procesadores disponibles en la cápsula y así poder inicializar al coordinador de partición, que necesita ese dato para realizar sus tareas, tales como dividir la región.

8 Conceptualmente hablando. Ver siguiente capítulo.
9 Veremos en el siguiente capítulo como esto implica crear un nuevo thread.

Interfaz del coordinador_particion

La interfaz que brinda el *coordinador de partición* y que podemos ver en la Figura 6 es la siguiente:

```
dividir_region(zone, region);
dame_particiones(): Listado<particion>;
dame_particion_principal(): particion ;
enviar_recibir_novedades(particion): particion;
recibir_novedades(particion): particion;
```

Figura 6: Interfaz que brinda el coordinador de partición

dividir_region() simplemente divide la región de manera equitativa en cuanto a cantidad de zonas, según el número de procesadores disponibles.

dame_particiones() es utilizado por el *motor de punto fijo* para obtener un listado de objetos *partición*, cada uno de los cuales contiene los datos que deberá procesar un thread. Cada *partición* servirá de parámetro para invocar al *runner*, que ejecutará en un thread, la rutina indicada sobre los datos que indique la *partición*.

dame_particion_principal() es utilizado por el *motor de punto fijo* para obtener la *partición* principal. En este caso en el cálculo del punto fijo se invoca el procedimiento de manera natural con la *partición*, o sea sin recurrir al *runner*, de manera que se ejecuta en el thread principal. Cabe destacar que el procedimiento que se invoca es distinto del que se le pide al *runner* que corra en el caso del resto de las particiones. Esto es por lo que explicamos anteriormente: los comportamientos del thread principal y del resto difieren en la segunda fase.

recibir_novedades() es utilizado por el *motor de punto fijo* en el thread principal para enterarse si hay algún resultado parcial disponible para hacer el *append*. Si hay resultado disponible en el *coordinador_particion*, éste lo entrega al *motor de punto fijo* para que proceda a hacer el *append*; sino, queda en espera bloqueante de algún resultado. El resultado del que hablamos es precisamente una partición para la que ya se hizo el cálculo y por ende está disponible su resultado parcial.

enviar_recibir_novedades() es utilizado por el resto de los threads para enterarse si hay algún resultado parcial para hacer el *append*, o bien, en caso que no haya, para informar su resultado parcial al coordinador de partición y finalizar su ejecución.

El *coordinador_partición* al dar el resultado a un thread *P1* (tanto sea el principal como otro), se queda sin resultado parcial disponible para entregar hasta tanto otro thread *P2* en fase 2 intente enviar y recibir novedades y se encuentre con que no hay resultado, entonces el resultado que quedará disponible en el coordinador de particion será el de *P2* y éste finalizará.

Algoritmo paralelo de interseccion_region_compl_zona()

Como mencionamos antes, la función que resultó paralelizada es *interseccion_region_compl_zona()*. En el Algoritmo 7 podemos ver cómo se utiliza el *coordinador de partición* para dividir la región y el *runner* para invocar a *interseccion_region_compl_zona2()* para cada partición. Por otro lado, vemos cómo se llama a *interseccion_region_compl_zona2_main()* para la partición principal sin utilizar el *runner*.

Podemos ver en el Algoritmo 8, que *interseccion_region_compl_zona()* e *interseccion_region_compl_zona_main()* tienen en común el proceso que ejecuta la fase1 y difieren en la fase2. *interseccion_region_compl_zona_main()* se ejecuta en el thread principal mientras que *interseccion_region_compl_zona()* se ejecuta en el *runner*. En el Algoritmo 9 podemos ver el pseudocódigo de la fase 1.

En lo que difiere el thread principal del resto en la fase 2, es justamente en la forma de pasar información con otros threads, uno utilizando enviar_recibir_novedades() (ver el Algoritmo 10) y otro recibir_novedades() (ver el Algoritmo 11) . El principal, si no hay resultados de otros, espera y no ofrece el suyo, en cambio el resto hace *appends* de otros

resultados mientras existan y cuando no, termina su ejecución y deja el resultado disponible para otro.

```
interseccion_region_compl_zona(region, zona): region {
  si zona no vacia hacer
     si region no vacia hacer

        nro_procesadores <- runner.obtener_nro_procesadores();
        nro_particiones <- nro_procesadores < tamaño(region) ? nro_procesadores
: tamaño(region)


        coordinador_particion.inicializar(nro_particiones)
        coordinador_particion.dividir_region(zone, region)

        particiones <- controlador_particion.dame_particiones()
        para cada particion en particiones
            //lanzo en paralelo
            runner.run(interseccion_region_compl_zona2(particion))

        principal <-  coordinador_particion.dame_particion_principal()
        interseccion_region_compl_zona2_main(principal)

        resultado <- principal.dame_resultado()
        coordinador_particion.destruir()

     sino
        resultado <- region_vacia()
  sino
      resultado <- region
  devolver resultado
}
```

Algoritmo 7: Pseudocódigo del algoritmo paralelo de interseccion_region_compl_zona()

```
interseccion_region_compl_zona2(particion) {
      interseccion_region_compl_zona2_fase1(particion)
      interseccion_region_compl_zona2_fase2(particion)
}

intersection_region_not_zone2_main(particion) {
      interseccion_region_compl_zona2_fase1(particion)
      interseccion_region_compl_zona2_fase2_main(particion)
}
```

Algoritmo 8: Pseudocódigo de los algoritmos utilizados por interseccion_region_compl_zona() para cualquier procesador y el principal

```
interseccion_region_compl_zona2_fase1(particion){
      region <-  particion.dame_region()
      zona_compl <- particion.dame_zona_compl()
      from <- particion.dame_desde()
      to <- particion.dame_hasta()

      zona <- region[from]
      resultado <- interseccion_zona_compl_zona(zona, zona_compl)

      para i desde from+1 hasta to hacer
            zona <- region[i]
            aux_region <- interseccion_zona_compl_zona(zona, zona_compl)
            resultado <- agregar_zonas(resultado, aux_region)

      particion.guardar_resultado(resultado)
}
```

Algoritmo 9: Pseudocódigo del algoritmo de la fase 1 del cálculo de la diferencia

```
interseccion_region_compl_zona2_fase2(particion){
      otra_particion <-
      coordinador_particion.enviar_recibir_novedades(particion)
      mientras no_nula(otra_particion) hacer
            resultado_local <-  particion.dame_resultado()
            resultado_externo <-  otra_particion.dame_resultado()
            resultado_local <- agregar_zonas(resultado_local, resultado_externo)
            particion.guardar_resultado(resultado_local)
            otra_particion <-
      coordinador_particion.enviar_recibir_novedades(particion)

}
```

Algoritmo 10: Pseudocódigo del algoritmo de la fase 2 del cálculo de la diferencia para los procesadores distintos del principal

```
interseccion_region_compl_zona2_fase2_main(particion){
      otra_particion <- coordinador_particion.recibir_novedades (particion)
      mientras no_nula(otra_particion) hacer
            resultado_local <- particion.dame_resultado()
            resultado_externo <- otra_particion.dame_resultado()

            resultado_local <- agregar_zonas(resultado_local, resultado_externo)
            particion.guardar_resultado(resultado_local)

            otra_particion <- coordinador_particion.recibir_novedades(particion)
}
```

Algoritmo 11: Pseudocódigo del algoritmo de la fase 2 del cálculo de la diferencia para el procesador principal

Algoritmos del coordinador de particion

En el Algoritmo 12 podemos ver el algoritmo que divide la región según el número de procesadores disponibles y vemos que hace una división equitativa. En *enviar_recibir_novedades()* y *recibir_novedades()* (ver el Algoritmo 13 y el Algoritmo 14) vemos cómo el coordinador de partición mantiene la partición finalizada por el último thread que finalizó y se la ofrece al próximo que finalice su fase 1 para que proceda con la fase 2.

Vemos también que en *recibir_novedades()*, utilizado por el principal, cuando no hay una partición finalizada se aguarda la señal de que haya una y hasta tanto eso suceda no se prosigue con la ejecución. En este mismo sentido vemos que en *enviar_recibir_novedades()* cuando no hay una partición finalizada y se está cediendo al *controlador de partición la partición* se envía una señal (veremos cómo se hace esto en el siguiente capítulo) por si el thread principal está esperando novedades.

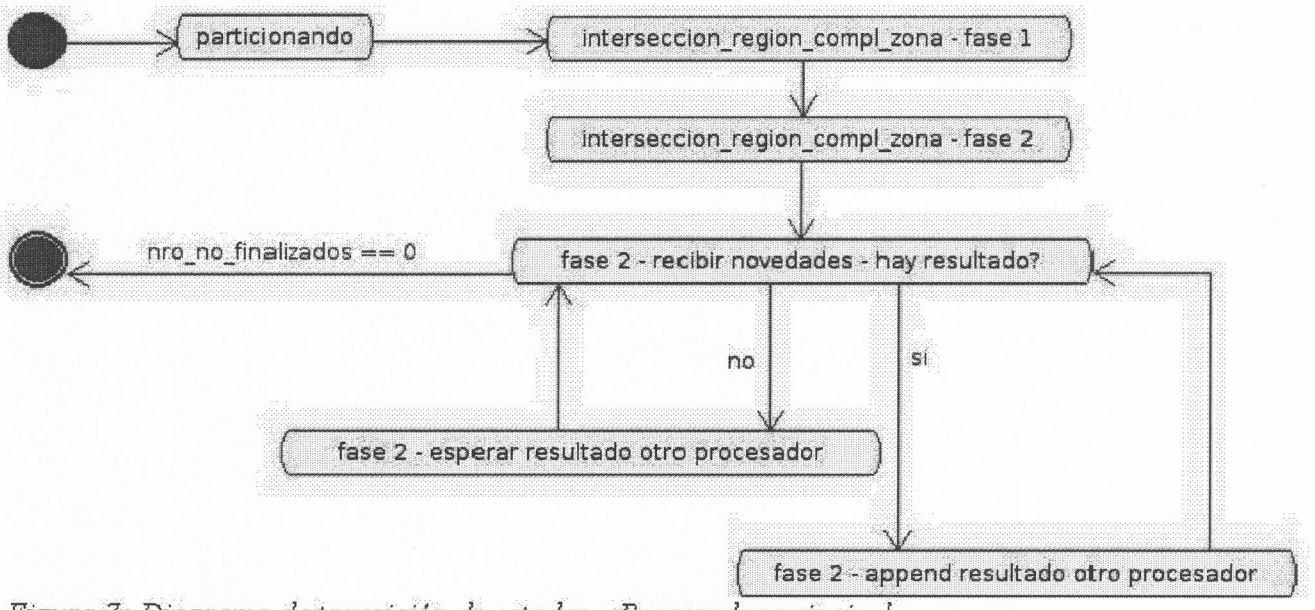

Figura 7: Diagrama de transición de estados - Procesador principal

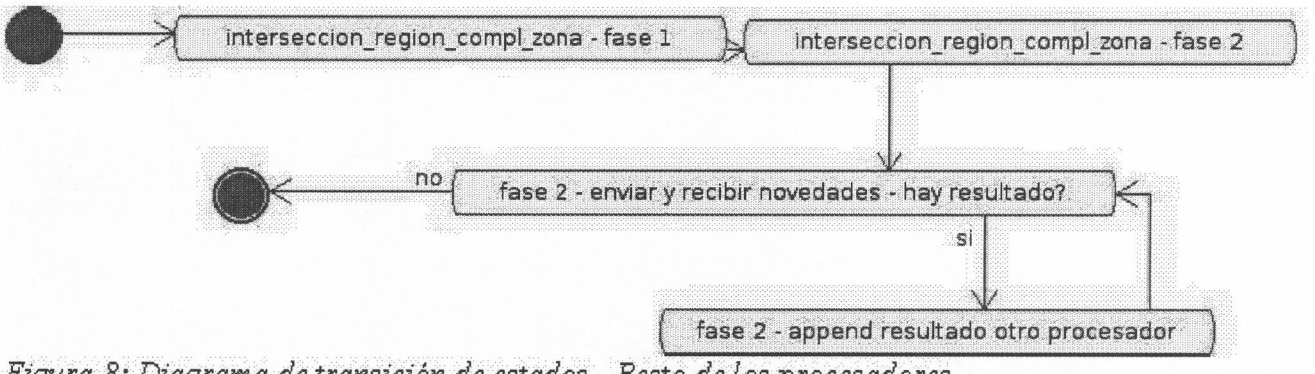

Figura 8: Diagrama de transición de estados - Resto de los procesadores

```
dividir_region(zona, region){
      listado_particion = crear_particiones(numero_procesadores)

      //cuenta la finalizacion de los procesadores, excepto el ppal
      no_finalizados <- numero_procesadores-1
      particion_finalizada <- VACIA

      from <- 0

      //tamaño en q divido la region
      sz <- tamaño(region)
      size <- sz / numero_ procesadores
      resto <- sz - (size * numero_procesadores)

      para i desde 0 hasta resto-1 hacer

            particion <- crear_particion()
            particion.setear_desde(from)
            particion.setear_zona_compl(zona)
            particion.setear_region(region)
            to <- from + size + 1;
            particion.setear_hasta(to)
            from <- to

      para i desde resto hasta numero_procesadores-1 hacer

            particion <- crear_particion()
            particion.setear_desde(from)
            particion.setear_zona_compl(zona)
            particion.setear_region(region)
            to <- from + size;
            particion.setear_hasta(to)
            from <- to

}
```

Algoritmo 12: Pseudocódigo de la división en particiones

```
enviar_recibir_novedades(particion): particion {
      proteger_acceso_a_resultado()

      //si algún otro thread termino devuelve su resultado parcial
      //sino el coordinador se queda el resultado parcial para luego ofrecerlo a
      //otro thread que lo solicite
      si particion_finalizada no VACIA
            resultado <- particion_finalizada
            particion_finalizada <- VACIA
      sino
            particion_finalizada <- particion
            no_finalizados <- no_finalizados-1
            enviar_señal_a_principal()

      liberar_acceso_a_resultado()
      devolver resultado
}
```

Algoritmo 13: Pseudocódigo de enviar_recibir_novedades()

```
recibir_novedades(particion): particion {
      proteger_acceso_a_resultado()

      mientras no_finalizados > 0
            y particion_finalizada VACIA hacer

            esperar_señal_de_otros_procesadores()

      resultado <- particion_finalizada
      no_finalizados <- no_finalizados-1
      particion_finalizada <- VACIA

      liberar_acceso_a_resultado()
      devolver resultado
}
```

Algoritmo 14: Pseudocódigo de recibir_novedades()

Análisis de corrección

Para demostrar que luego de los cambios realizados el cálculo sigue siendo correcto debemos corroborar dos cosas: por un lado que el nuevo orden en que se realiza el cálculo no altera el resultado final y por otro que no se llega a un deadlock al hacer el cálculo en paralelo.

En cuanto al primer tema basta ver que la unión de regiones es asociativa. El hecho de que los cálculos de las intersecciones entre dos zonas se realicen todos en un mismo thread o en diferentes no afecta. Entonces lo único que cambió es el orden en que se van uniendo los resultados parciales, pero como la unión de regiones es asociativa el algoritmo continua siendo correcto.

En cuanto a la ausencia de deadlock debemos asegurarnos que no se den en simultáneo las cuatro condiciones de Coffman [CES71], que son las siguientes:

Dados los procesos *Po, P1,.. Pn* y los recursos *Ro, R1,..., Rm*:

- Condición de exclusión mutua: Existencia al menos de un recurso compartido por los procesos, al cual sólo puede acceder uno simultáneamente.
- Condición de posesión y espera: Al menos un proceso *Pi* ha adquirido un recurso *Ri*, y lo mantiene mientras espera al menos un recurso *Rj* que ya ha sido asignado a otro proceso.
- Condición de no expropiación: Los recursos no pueden ser apropiados por los procesos, es decir, los recursos sólo podrán ser liberados voluntariamente por sus propietarios.
- Condición de espera circular: Dado el conjunto de procesos *P0...Pn*, *P0* está esperando un recurso adquirido por *P1*, que está esperando un recurso adquirido por *P2*, que...,que está esperando un recurso adquirido por *Pn*, que está esperando un recurso adquirido por *P0*. Esta condición implica la condición de posesión y espera.

En general, estas condiciones son necesarias, pero no suficientes. En este caso como de cada recurso compartido hay uno sólo, además son suficientes.

El recurso en nuestro caso son las variables *particion_finalizada* y *no_finalizados*, que almacenan respectivamente, la partición finalizada del último thread que finalizó y la cantidad de threads que aún continuan ejecutando el cálculo. Podemos ver en el pseudocódigo del Algoritmo 13 y del Algoritmo 14 como se protege el acceso concurrente con los llamados a *proteger_acceso_a_resultado()* y *liberar_acceso_a_resultado()*[10]. Si bien son dos variables las trataremos como un único recurso ya que siempre se protege y libera el acceso a ambas en simultáneo.

Un detalle importante es la forma en que se comunican el thread principal con el resto. Podemos ver en el Algoritmo 13, ejecutado por un thread diferente al principal, como el coordinador de partición cuando guarda una partición temporal, envía una señal al thread principal por si este está ocioso esperando una señal. En el Algoritmo 14, vemos como el principal aguarda la señal de otro thread cuando no hay una partición con la cual continuar la ejecución. En el siguiente capítulo veremos los detalles implementativos del envío y recepción de señales, pero vale la pena aclarar en este momento que cuando se espera una señal, implícitamente se libera la protección de acceso a resultado, de la misma manera que si hubiesemos invocado a *liberar_acceso_a_resultado()*. A su vez cuando se recibe una señal, implícitamente, se vuelve a proteger el acceso a resultado de la misma manera que invocando a *proteger_acceso_a_resultado()*.

Si analizamos nuestro caso, podemos ver que la condición de exclusión mutua ocurre porque sólo puede acceder al recurso un proceso a la vez. La condición de no expropiación, también, porque sólo pueden ser liberados por el proceso que los adquirió. La condición de posesión y espera, si bien existe un único recurso, puede darse en el caso de que un thread se quede esperando la señal de otro. El único que queda esperando una señal es el principal, y lo hace mientras no halla recibido la señal de todos los otros threads. Entonces, para el resto de los threads no puede darse la condición de posesión y espera y tampoco la de espera circular, ya que esta última implica la primera. Por lo tanto, el resto de los threads están libres de deadlock y por ende van a poder finalizar su tarea con éxito y enviar la señal al principal evitando así la espera indefinida de éste. Entonces hemos demostrado la ausencia de deadlock.

10 Veremos como se implementa en el siguiente capítulo.

Capítulo 6

Implementación

En este capítulo describiremos los detalles de implementación más importantes del desarrollo descripto en el capítulo anterior.

Como mencionamos antes, para este desarrollo partimos del model checker distribuido Zeus, que está implementado en ANSI C, al igual que su predecesor. El sistema operativo para el cual está desarrollado es Unix. Por este motivo, este trabajo también se hizo sobre el mismo lenguaje y sistema operativo.

Para realizar el procesamiento en paralelo decidimos utilizar la API de POSIX THREADS de C [MAR99], que es una interfaz de programación estándar definida por IEEE y soportada por la mayoría de sistemas operativos y hardware.

Decidimos utilizar threads en lugar de procesos debido a que:

- Toma menos tiempo la creación porque usa el mismo espacio de direcciones que el proceso padre.
- Toma menos tiempo cambiar entre la ejecución de threads dentro de un mismo proceso, en el caso de que no halla tantos procesadores como threads corriendo o si un thread está ocioso esperando el resultado de otro.
- Tienen menos overhead de comunicación porque al compartir memoria, el resultado de un thread está inmediatamente disponible para otro.
- Tienen primitivas de sincronización.

Utilizando POSIX threads queda bajo la responsabilidad del sistema operativo asignar los threads a diferentes procesadores, con la ventaja de evitar hacer este tipo de trabajo desde la aplicación, pero lógicamente perdiendo algo en flexibilidad.

Los cambios realizados a Zeus quedaron dentro del motor de punto fijo por lo que básicamente el programa sigue siendo el mismo. Continua teniendo dos ejecutables, uno para el coordinador y otro para las cápsulas. Toma como input los mismos tres archivos que Zeus: el de configuración, el del autómata y el de la fórmula. Al archivo de configuración se agregó el número de procesadores que posee cada cápsula.

Arquitectura de implementación multithread

En cuanto a la arquitectura de implementación paralela se siguió uno de los patrones de diseño para threads más comunes que es el "Boss/Worker" y que consiste en que el boss o thread principal asigna trabajo a hacerse en paralelo a los workers (otros threads).

Si bien este patrón se adapta bien a nuestras necesidades, presenta el gran inconveniente de un alto overhead de creación y destrucción de threads, más precisamente de su contexto de ejecución. Por esto, una de las decisiones claves de la implementación fue agregar un pool de workers de manera de crear y destruir los threads una sola vez por cápsula.

Otro patrón de diseño utilizado fue el de pipeline de threads para la segunda fase del cálculo de la diferencia, pero su uso ya quedó evidenciado en la arquitectura del desarrollo híbrido, mientras que el pool quedó como un detalle de implementación.

Pool de workers

El pool de workers está encapsulado por el componente runner, dejando oculta su presencia al motor de punto fijo. El runner, cumpliendo el rol de boss, internamente mantiene una colección de workers sabiendo cuáles están libres.

Cada worker internamente, al ser creado, crea un thread que inicia la corrida de una rutina especial que explicaremos más adelante y la llamaremos *run_thread()*.

Un detalle importante es que nunca se le pide al runner que haga más tareas que la cantidad de procesadores disponibles en la cápsula. Así se evita que el runner esté sobrecargado de trabajo y no tenga workers disponible a quien asignar el trabajo.

Una suposición importante es que estamos hablando siempre de casos en que ninguna otra aplicación y/o usuario esá usando la máquina donde se está ejecutando el model checker al mismo tiempo. Se trata de una suposición razonable para este tipo de software.

Los principios de diseño del pool fueron los siguientes:

- Todos los workers son creados una sola vez al comienzo por el runner
- La relación entre el runner y los worker siguen el patrón de diseño producer/consumer: los workers aguardan señal del runner que les indique que tienen que "hacer un trabajo". El runner cuando tiene una tarea que ejecutar, busca un worker disponible y le manda una señal para que se ejecute, dejándole los parámetros necesarios.
- La cantidad de workers en el pool, y por ende de threads, es igual a la cantidad de procesadores disponibles en cada cápsula (menos uno, por el principal).

Para la implementación del patrón boss/worker con pool, utilizamos variables condicionales y mutex que forman parte de la API de POSIX THREADS. Los mutex sirven para proteger el acceso concurrente a variables compartidas por más de un thread. Las variables condicionales son utilizadas para bloquear un thread hasta que determinada condición sea verdadera. El acceso a las variables condicionales es protegido por un mutex. El uso típico de las variables condiciones es el siguiente:

- Cuando una condición es falsa el thread se bloquea en una variable condicional y libera el mutex aguardando que la condición cambie.
- Cuando otro thread cambia la condición, envía una señal a la variable condicional de manera que los threads esperando en ella "despierten".

En ese mismo sentido fueron utilizadas para implementar la rutina run_thread que inicia cada worker al ser creado. En esta rutina, los workers esperan en variable condicional que el runner los avise para "hacer algo"; al finalizar vuelven a esperar en variable condicional hasta recibir una señal de apagado en la que finalizan su ejecución.

Pipeline de threads

Como vimos en el capítulo anterior, cuando un thread finaliza su fase 1 del cálculo de la diferencia, procede con la fase 2 utilizando resultados de otro thread. En este sentido decimos que utilizamos un pipeline de threads en el que el resultado de uno es utilizado por otro y así sucesivamente.

La responsabilidad de la ejecución en pipeline quedó dentro del *coordinador de partición*, en los métodos *enviar_recibir_novedades()* y *recibir_novedades()*, utilizados por los métodos que se ejecutan en paralelo dentro del *motor de punto fijo*.

Utilizamos mutex para proteger el acceso a la partición resultado y al contador que conoce el número de threads no finalizados. En el pseudocódigo vemos el llamado a *proteger_acceso_a_resultado()* y a *liberar_acceso_a_resultado()* tanto en *enviar_recibir_novedades()* como en *recibir_novedades()*.

También fue necesario utilizar variables condicionales para implementar la segunda fase del cálculo de la diferencia de regiones paralela. Recordemos que el thread principal nunca deja su resultado y finaliza su ejecución; en cambio, cuando no hay resultado disponible, aguarda que haya resultado de otro thread. Esto lo hace aguardando en una variable condicional. El resto de los threads al dejar el resultado envían una señal al principal, justamente haciendo un *signal* a esa misma variable condicional, lo que despierta al thread principal para que continúe con la ejecución.

Algunos pseudocódigos importantes

A continuación en el Algoritmo 15, vemos el procedimiento worker_run_thread(), la rutina que corre cada uno de los threads. Podemos observar en detalle como un worker espera en una variable condicional[11] hasta que haya una señal de que debe "hacer un trabajo"; en este caso el trabajo que debe hacer es precisamente ejecutar intersection_region_not_zone(). Cuando el coordinador o una cápsula finalicen su ejecución destruirán el runner, el cual destruirá cada uno de sus workers; para hacer esto último cambiará el estado de la variable is_continue de cada worker. Lógicamente, el acceso a esta variable como el acceso a la variable condicional están protegidos por un mutex. No sólo cambiará el estado, sino que enviará una señal a cada worker, pero esta vez para que el worker se entere que debe finalizar su ejecución. Vemos que para esto invoca a pthread_exit() de la API de threads, lo que destruye el thread.

```
worker_run_thread(parametro){
      WORKER worker = (WORKER) parametro;
      worker.lockear_mutex();
      mientras sea verdadero worker.is_continue() hacer
            workers.esperar_en_variable_condicional()
            si es verdadero worker.is_continue() hacer
                  //va a ser interseccion_region_compl_zona2
                  //y el parámetro una particion
                  invocar worker.obtener_procedimiento()
                  con parametro worker.obtener_parametro()

      worker.deslockear_mutex()
      pthread_exit(NULL)
      devolver 0
}
```

Algoritmo 15: Pseudocódigo del método worker_run_thread()

En el Algoritmo 16 veremos el procedimiento *do_job()* de los *workers* que es llamado por el *runner* sobre el *worker* que encuentra libre cuando necesita ejecutar una tarea. El runner invoca a este método que envía una señal despertando el thread del worker que comenzará a ejecutar el procedimiento indicado ya que tiene la variable is_continue encendida.

```
do_job(procedimiento, parametros){
      this.setear_parametros(parametros)
      this.setear_procedimiento(procedimiento)

      this.lockear_mutex ()
      this.enviar_signal()
      this.deslockear_mutex ()
}
```

Algoritmo 16: Pseudocódigo del método do_job() de cada worker

11 Las variables condicionales van asociadas siempre a un mutex. Cuando se espera en una variable condicional se libera el mutex hasta que llegue la señal, y recién en ese momento se lo vuelve a tomar.

Capítulo 7

Resultados

Las pruebas las realizamos en un equipo con 4 procesadores Intel(R) Xeon(TM) de 2.66GHz cada uno y 2GB de memoria RAM que corre el kernel Linux 2.6.

Para las pruebas utilizamos los siguientes casos de estudio: *RCS6, Railroad Crossing System* con 6 trenes, que se encuentra descripto en [ACD92], *Mine Pump* [Bra00], *Remote Sensing (Correlation properties)* [ABO04] y *Struct Sliced* [Str], que fue procesado previamente por ObsSlice -un reductor de estados conservativo- [BGO04].

En la Tabla 1 podemos ver los tamaños de los autómatas temporizados de cada uno de los ejemplos y el resultado que producen, o sea si tienen o no estado de error alcanzable.

Caso	Relojes	Estados discretos	Transiciones	Resultado
RCS6	9	5288	36072	Falso
Mine Pump	6	4452	21932	Falso
Remote Sensing	13	25975	156944	Verdadero
Struct Sliced	7	495	1665	Verdadero

Tabla 1: Tamaño de los casos de estudio

Para cada caso de estudio hicimos pruebas con la versión original (distribuida) y la nueva versión híbrida. Con la original corrimos pruebas con 1, 2, 3 y 4 cápsulas. A la corrida con 1 cápsula la llamaremos original. Con la versión híbrida corrimos pruebas utilizando 1, 2, 3 y 4 threads. Por otro lado, con esta misma versión hicimos pruebas híbridas que combinaran más de una cápsula y en cada cápsula más de un thread. Para todos los casos ejecutamos 3 corridas teniendo siempre como precondición que el equipo esté libre.

En el resto del capítulo veremos para cada caso de estudio una tabla con los resultados obtenidos y un gráfico donde se podrán comparar los desempeños de las distintas versiones y configuraciones.

En todas las tablas de resultados mostramos una primera columna con la configuración de cápsulas y threads (t significa thread y c cápsula), una columna con el promedio de las 3 corridas en segundos, otra con el porcentaje de mejora y finalmente una con la ganancia obtenida.

El porcentaje de mejora lo calculamos como:

$$M_x = 100 - \frac{(tt_x * 100)}{t_1}$$

donde tt_x es el tiempo total con x procesadores, t_1 es el tiempo total con 1 procesador, y por lo tanto M_x representa el porcentaje de mejora obtenido con x procesadores sobre 1 procesador.

Por otro lado, la ganancia G_x esta dada por:

$$G_x = \frac{t_1}{tt_x}$$

Un valor de Gx superior a 1 indica una ganancia, mientras que inferior una pérdida. Llamaremos mejoras lineales a las que utilizando x threads o x cápsulas se dieron ganancias de valores aproximados a x.

En los gráficos, mostraremos en una serie las ganancias obtenidas ejecutando con la versión original y dividiendo en 1, 2, 3 y 4 cápsulas; en otra, la ganancia de la versión híbrida con configuraciones de 1, 2, 3 y 4 threads, y una tercera con

la ganancia de la versión híbrida dividiendo en 2 cápsulas con 1 sólo thread por cápsula y 2 cápsulas con 2 threads cada una. Así podemos observar las diferencias entre las ejecuciones distribuidas, paralelas e híbridas.

Un detalle importante es que la versión distribuida, en este caso, se ejecutó en una arquitectura paralela con memoria compartida, lo cuál resulta en que se crearán distintos threads para cada cápsula y que queda en el sistema operativo la responsabilidad de asignarlos a diferentes procesadores al igual que en la nueva versión. Una ventaja es que el tamaño de la memoria es compartido por todas las cápsulas, de modo que si una requiere más espacio que el resto no habría problema. Por límites de tiempo, para todos los casos se probó una sola distribución posible al dividir en cápsulas.

Por otro lado, para cada caso de estudio mostraremos una tabla adicional que exhibe en que medida influye el cálculo de la diferencia de regiones en el total del cálculo. Estos resultados los obtuvimos probando en un equipo con un procesador Intel(R) Pentium(R) 4 de 3.00GHz con HyperThreading y 2GB de memoria RAM. La primera columna de la tabla muestra la configuración, la segunda el tiempo total en segundos, la tercera el tiempo que insume la diferencia de regiones, la cuarta qué porcentaje representa la diferencia de regiones del total y la última la mejora obtenida al utilizar 2 threads. Los resultados expuestos en Tabla 3, Tabla 5, Tabla 7 y Tabla 9 muestran que la diferencia de regiones representa entre el 80% y el 95% del total del cálculo, lo que justifica que es la función más costosa del motor de punto fijo y que por lo tanto valió la pena paralelizarla.

Además de los casos de estudio que aquí mencionamos se hicieron otras pruebas que no documentamos o bien por no contar con una versión estable en el momento de la misma o bien porque no produjeron resultados destacables. Por ejemplo, algunas de ellas daban tiempos despreciables en un procesador y al dividir tanto en cápsulas o threads los resultados empeoraban.

RCS6

Como se puede ver en este ejemplo se obtuvieron mejoras del 22% dividiendo en 4 threads, mientras que con 4 cápsulas no se obtuvieron ventajas. Esto puede ser atribuible a que algunas locaciones concentran una mayor cantidad de carga de trabajo, con lo cual no resulta provechoso una distribución de las locaciones en cápsulas. Confirmando esto, la configuración en la que combinamos cápsulas y threads dió un resultado intermedio. Por otro lado, comparando la versión original con la versión híbrida con un thread podemos ver que el overhead agregado en la nueva versión no es demasiado.

Una hipótesis posible para justificar la mejora de sólo el 22% con respecto al 95% que representa el cálculo de la diferencia de regiones (ver Tabla 3), puede ser una distribución no equitativa del trabajo entre los threads causada porque no todas las celdas de la DBM son utilizadas en el cálculo. Entonces, si bien todas las zonas tienen el mismo tamaño, no todas las llamadas a *interseccion_zona_compl_zona()* toman lo mismo. Zeus utiliza un bitmap para cada zona que indica que celdas son suficientes para representar la matriz. En conclusión, la fase 1 (ver Algoritmo 9) de cada thread no necesariamente tiene la misma carga de trabajo.

También probamos para este mismo ejemplo un caso con estado de error alcanzable, pero como daba tiempos despreciables en 1 procesador, al agregar threads o cápsulas la performance se decrementaba.

Procesadores	Promedio	% mejora	Ganancia
original	986,41	0,0000	1,0000
1t	988,16	-0,1777	0,9982
2t	834,85	15,3643	1,1815
3t	817,44	17,1294	1,2067
4t	761,1	22,8412	1,2960
2c	1001,85	-1,5651	0,9846
3c	1009,21	-2,3110	0,9774
4c	1015,59	-2,9579	0,9713
2c1t	1000,19	-1,3971	0,9862
2c2t	846,43	14,1908	1,1654
2c4t	782,05	20,7174	1,2613
4c2t	857,29	13,0900	1,1506
4c4t	792,3	19,6786	1,2450

Tabla 2: Resultados obtenidos con el caso de estudio RCS6

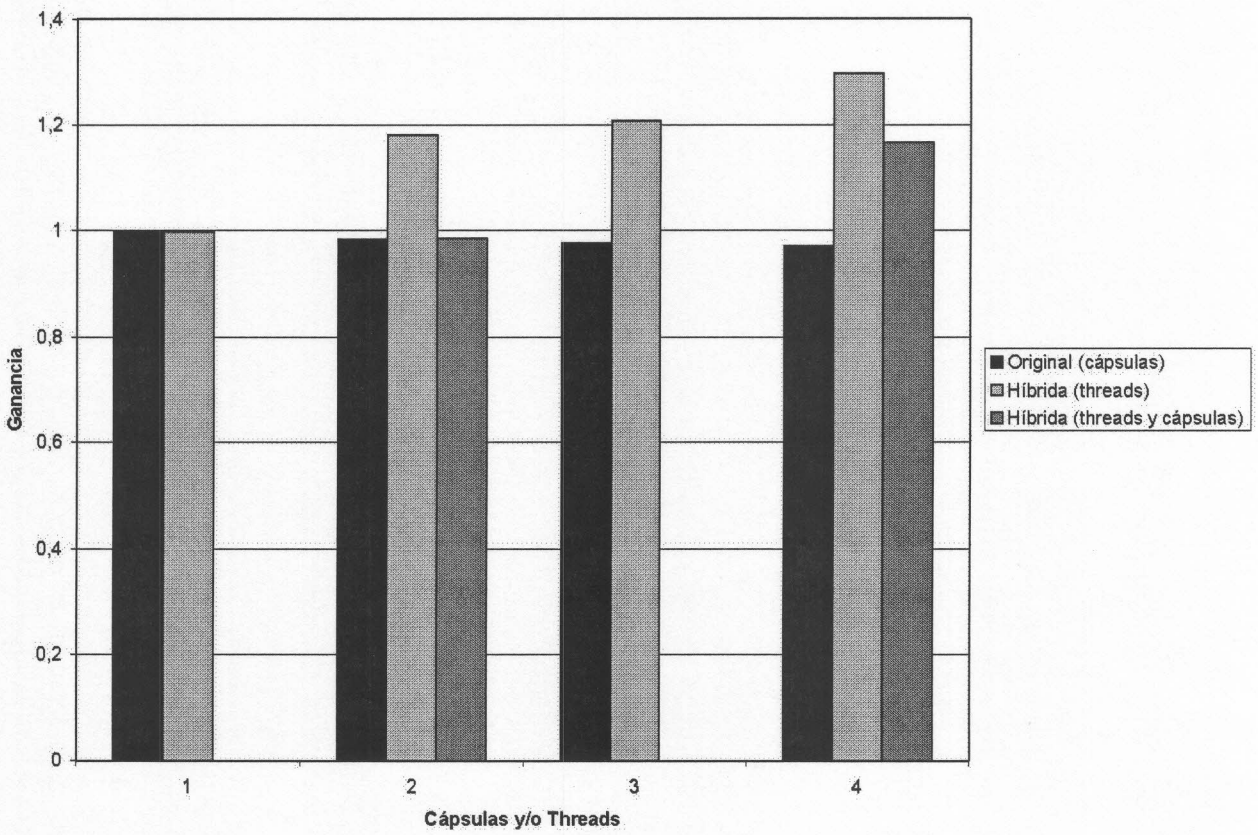

Figura 9: Resultados obtenidos con el caso de estudio RCS6

Procesadores	Total	Dif region	% dif reg/total	% mejora
1t	1111,2	1059,22	95,32	0
2t	940,13	888,4	94,5	15,4

Tabla 3: Tiempo del cálculo de la diferencia sobre el total en RCS6

Mine Pump

Este caso de estudio demuestra el mismo comportamiento que el anterior en todas las configuraciones, pero obteniendo una mejora para el caso de 4 threads del 25%. La posible causa de haber obtenido sólo esta mejora podría ser la misma que la explicada para el caso de estudio anterior.

Procesadores	Promedio	% mejora	Ganancia
original	1481,51	0,0000	1,0000
1t	1481,89	-0,0261	0,9997
2t	1193,31	19,4533	1,2415
3t	1198,91	19,0749	1,2357
4t	1108,1	25,2048	1,3370
2c	1518,55	-2,5002	0,9756
3c	1392,3	6,0214	1,0641
4c	1411,94	4,6960	1,0493
2c1t	1516,01	-2,3289	0,9772
2c2t	1252,92	15,4296	1,1824
2c4t	1154,37	22,0814	1,2834

Tabla 4: Resultados obtenidos con el caso de estudio Mine Pump

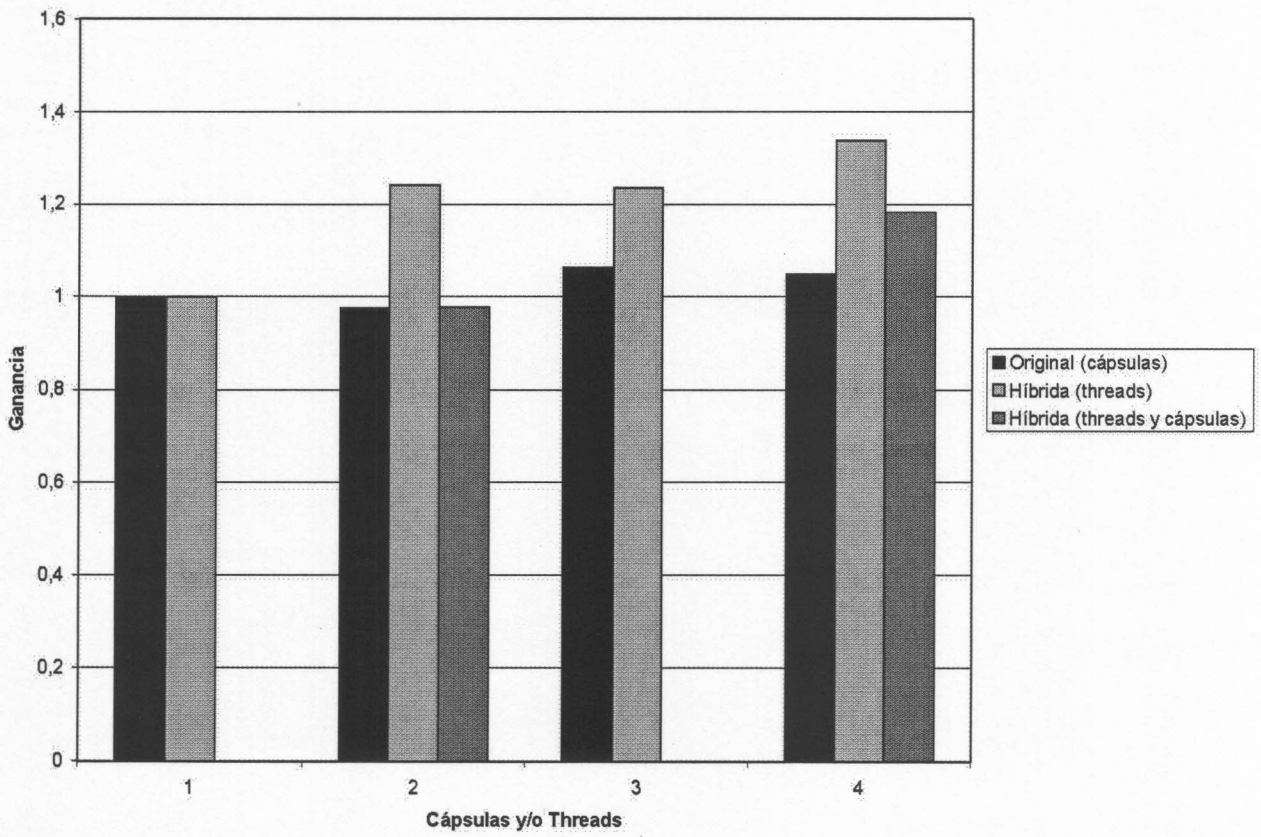

Figura 10: Resultados obtenidos con el caso de estudio Mine Pump

1t	1724,8	1645,52	95,4	0
2t	1494,85	1415,59	94,7	13,33

Tabla 5: Tiempo del cálculo de la diferencia sobre el total en Mine Pump

Remote Sensing

En este caso de estudio obtuvimos resultados diferentes a los dos primeros; la versión distribuida dió un porcentaje de mejora importante (61%) y notablemente superior a la versión híbrida dividiendo en 4 threads (26%). Esto puede ser atribuido a que este caso da como resultado "Verdadero", es decir que tiene estado de error alcanzable, con lo cuál al dividir en cápsulas y dado que en cada iteración se chequea si se alcanzó el estado inicial -independientemente de si se logró alcanzar un punto fijo-, se logró llegar al resultado en menos tiempo. Pudo haber influido también una distribución favorable de las locaciones. Resulta más interesante aún como la versión híbrida con 2 cápsulas y 2 threads fue la que mejores resultados arrojó (72% mejora), obteniéndose ganancias cercanas a las lineales.

También en este caso, la mejora de sólo el 26% dividiendo en 4 *threads* podría ser atribuida a una distribución no equitativa del trabajo entre los *threads* al igual que en los casos anteriores.

original	782,86	0,0000	1,0000
1t	781,42	0,1844	1,0018
2t	517,89	33,8463	1,5116
3t	581,75	25,6895	1,3457
2c	329,25	57,9430	2,3777
3c	290,61	62,8788	2,6939
2c1t	327,31	58,1907	2,3918
2c4t	210,36	73,1289	3,7215

Tabla 6: Resultados obtenidos con el caso de estudio Remote Sensing

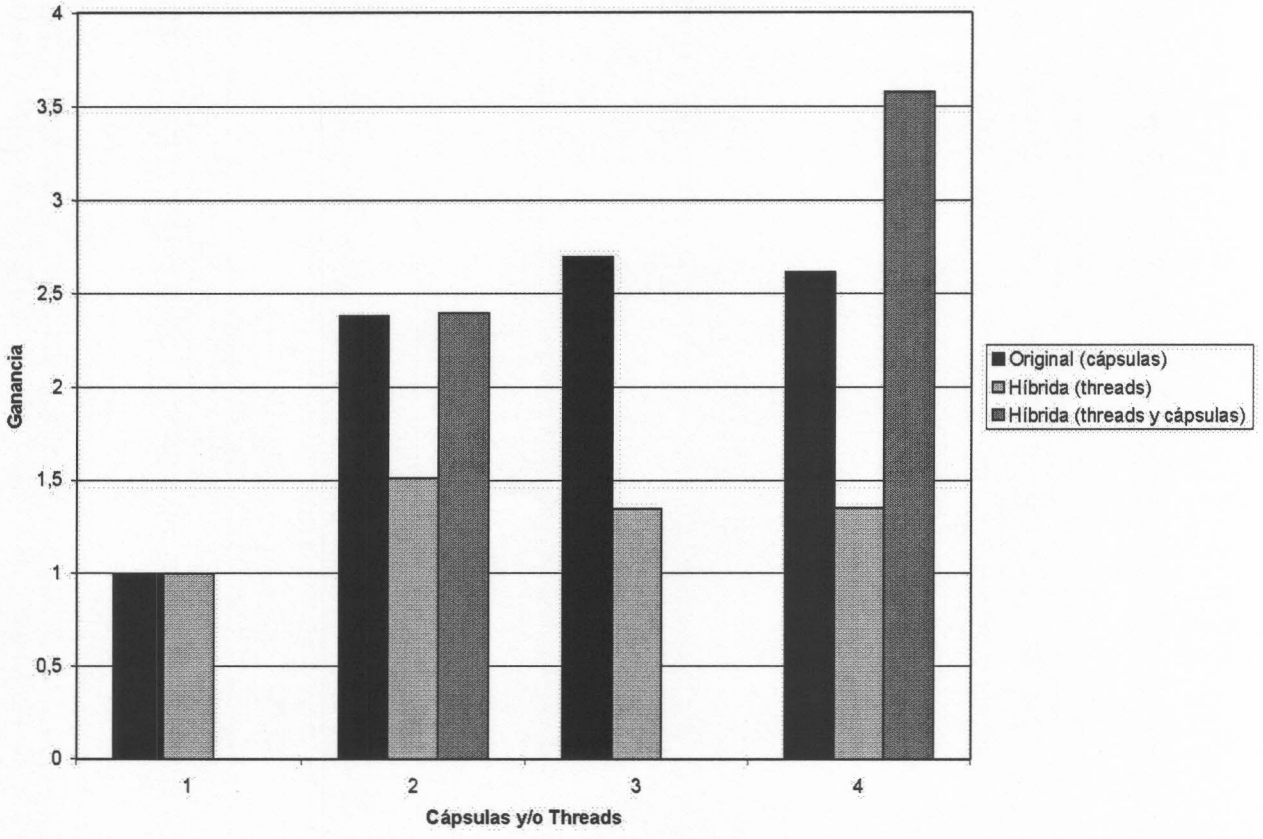

Figura 11: Resultados obtenidos con el caso de estudio Remote Sensing

Procesadores	Total	Dif region	% dif reg/total	% mejora
1t	784,53	671,16	85,55	0
2t	585,04	471,35	80,57	25,43

Tabla 7: Tiempo del cálculo de la diferencia sobre el total en Remote Sensing

Struct Sliced

En este caso, se puede ver que los resultados obtenidos son similares al de los 2 primeros casos, pero con una mejora del 29% con respecto a la versión original. Esta vez, si bien el estado de error es alcanzable, el hecho de dividir en cápsulas no ayudó a conseguir una mejora importante. Esto podría atribuirse a un fenómeno de mala distribución de carga como el reportado en [BOS06].

La hipótesis que justifique el porque el nivel de mejora obtenido diviendo en threads sería la misma que en el resto de los casos.

Procesadores	Promedio	% mejora	Ganancia
original	707,9	0,0000	1,0000
1t	707,72	0,0256	1,0003
2t	546,15	22,8497	1,2962
3t	546,12	22,8544	1,2962
4t	498,79	29,5394	1,4192
2c	737,4	-4,1662	0,9600
3c	734,4	-3,7425	0,9639
4c	745,44	-5,3018	0,9497
2c1t	731,13	-3,2815	0,9682
2c2t	572,09	19,1849	1,2374
2c4t	531,81	24,8761	1,3311
4c2t	584,69	17,4058	1,2107
4c4t	543,49	23,2258	1,3025

Tabla 8: Resultados obtenidos con el caso de estudio Struct Sliced

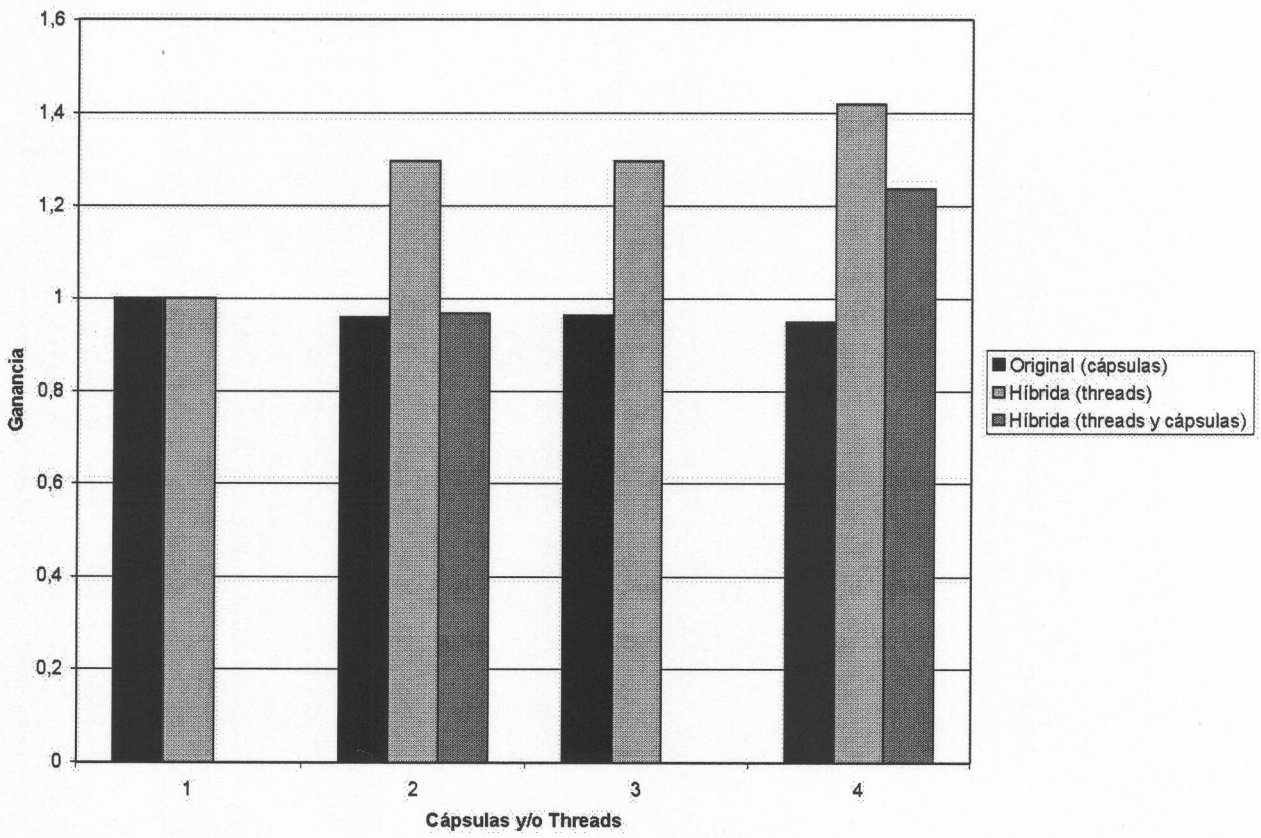

Figura 12: Resultados obtenidos con el caso de estudio Struct Sliced

Procesadores	Total	Dif region	% dif reg/total	% mejora
1t	926,73	852,48	91,99	0
2t	773,41	699,57	90,45	16,54

Tabla 9: Tiempo del cálculo de la diferencia sobre el total en Struct Sliced

Capítulo 8

Conclusión y trabajo futuro

Zeus es un model checker distribuido para modelos representados con autómatas temporizados sobre los que verifica propiedades utilizando la técnica de alcanzabilidad antes mencionada. Usa la técnica del cálculo de punto fijo de manera *backwards*, de forma que partiendo de los estados finales intenta alcanzar el estado inicial. La estructura de datos principal es la DBM, que permite representar valuaciones de relojes.

En este trabajo tomamos la versión sincrónica de Zeus y el hecho de analizar y conocer sus limitaciones nos motivó a implementar una nueva versión, además de distribuida paralela, que saque provecho de arquitecturas con más de un procesador con memoria compartida. Luego de que decidimos que partes eran factibles de paralelizar, elegimos la función que efectúa la resta de regiones, una de las operaciones del cálculo de punto fijo, que resulta ser la más cara en cuanto a tiempo de procesamiento que insume.

Una vez implementada la nueva versión hicimos pruebas con la versión distribuida y con la híbrida tomando diferentes configuraciones como dividir sólo en cápsulas, dividir sólo en threads o combinaciones de ambas opciones. En todos los casos con la versión híbrida conseguimos mejoras de alrededor del 25%. Con la versión distribuida se dió una mejora del 61% en uno de los casos y en el resto no obtuvimos buenos resultados. Lo que nos resultó más interesante, es que en este mismo ejemplo, se obtuvo una mejora del 72% con la versión híbrida y una configuración con 2 cápsulas en la que cada una ejecutaba 2 threads. Así comprobamos como se sumaron las ventajas de ambas versiones, resultando en una buena combinación que explota la nueva funcionalidad multithreading del cálculo de la diferencia de regiones al alcanzar los límites de mejora de la versión original.

Como vemos, si bien con la nueva versión híbrida en todos los casos se consiguieron mejoras, en general estas estuvieron lejos de ser lineales, salvo en el caso de la mejora del 72% que resultó en un *speed up* de casi 4 en 4 procesadores. Nos queda analizar por qué sólo se dió este nivel de mejora. En la versión distribuida estamos limitados en la performance alcanzable porque algunas cápsulas concentran la mayor carga de trabajo[12]. En cuanto a la versión híbrida, hay que tener en cuenta que al dividir las zonas del cálculo de la diferencia de regiones en threads no estamos partiendo de la totalidad del problema sino que estamos paralelizando sólo esta operación. Aun así el calculo de la diferencia representa entre un 80 y un 95% del total. Creemos que la causa de los resultados obtenidos puede ser la hipótesis planteada en los casos de estudio del capítulo anterior: una distribución no equitativa de la carga entre los threads dado que no se utilizan todas las celdas de la DBM en el cálculo y por lo tanto *interseccion_zona_compl_zona()* no siempre toma lo mismo. Otra causa posible al límite en la mejora puede ser que el número de zonas que componen una región tiende a aumentar iteracion a iteración por la misma naturaleza del cálculo de punto fijo con DBM. Así no contamos con la totalidad de la carga de trabajo a dividir desde el comienzo, sino que siempre queda para las últimas iteraciones una mayor carga.

Dado que la experimentación sólo pudo realizarse en un equipo con 4 procesadores, queda como trabajo futuro analizar los resultados en un cluster con más nodos. Debe notarse, sin embargo, que el hecho de que la versión original fuese ejecutada en un multiprocesador en lugar de en 4 equipos separados no la perjudica. Por el contrario, la beneficia, ya que no se utiliza la red para intercomunicar a las cápsulas. De esta manera, creemos que si se hubiese corrido en un entorno realmente distribuido los resultados de la nueva versión se hubieran visto, comparativamente, un poco mejor.

En un trabajo futuro sería interseante probar casos de estudio más grandes, probar con diferentes alternativas de distribuciones de locaciones a cápsulas y verificar si al contar con más procesadores y dividir en más cantidad de threads se sigue manteniendo el mismo nivel de mejora.

Nos ha quedado fuera de este trabajo probar el resto de las alternativas que pensamos para paralelizar el calculo de la diferencia de regiones. Si bien la opción elegida es la que creemos mejor, no estaría demás probar las otras, sobre todo la tercera, que era paralelizar el cálculo de la DBM a nivel celda.

Otro de los problemas a tratar en un trabajo futuro es el tema de la explosión de zonas, una de las causas más importantes que dificultan la escalabilidad de los model checkers que usan DBM como estructura de datos. Una de las opciones es compactando las regiones luego de cada operación de resta. Lamentablemente esta operación es muy costosa también, pero quizá paralelizándola se puedan obtener resultados positivos.

Hasta la finalización de esta tesis no hemos encontrado otro model checker temporizado en el que se haya efectuado un

12 Ver [BOS06] para un análisis a fondo del problema y de los límites de la versión distribuida

enfoque híbrido, distribuido y paralelo con memoria compartida.

Bibliografía

[ABO04] A. Alfonso, Braberman V., Kicillof N., and Olivero A. Visual Timed Event Scenarios. In: Proc. of the 26th ACM/IEEE International Conference on Software Engineering. 2004.
[ACD90] R. Alur, C. Courcoubetis, D. Dill. Model Checking for Real-Timesystems. En Proc. of 5th Symp. on Logic in Computer Science, IEEE Computer Society Press, pages: 414-425. 1990.
[ACD92] R. Alur, Courcoubetis C., Dill D, Halbwachs N., and Wong-Toi H.. AnImplementation of three algorithms for timing verification based on automataemptiness. In Proceedings of the 13th IEEE Real-time Systems Symposium.Phoenix, Arizona, pages: 157-166. 1992.
[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real-time. Information and Computation, 104(1): 2–34. 1993.
[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. Theoretical Computer Science, 126(2):183–235. 1994.
[AH91] R. Alur and T. A. Henzinger. Logics and Models of Real-Time: a survey. Proc. of REX Workshop on Real-time: Theory in Practice. LectureNotes in Computer Science 600, Springer-Verlag, . 1991.
[Ake78] Sheldon B. Akers. Binary Decision Diagrams. IEEE Transactions on Computers, 509–516. 1978.
[BBS01] J. Barnat, L. Brim, and J. Strbrn. Distributed LTL model-checking in SPIN. In SPIN, pages: 200-216. 2001.
[Beh05] G. Behrmann. Distributed reachability analysis in timed automata. International Journal on Software Tools for Technology Transfer (STTT), 7(1): 19-31. February 2005.
[BER95] O. Bernholtz. Model Checking for Branching Time Temporal Logics. PhD thesis, The Technion, Haifa, Israel. Junio 1995.
[BGO04] Braberman V. and Garbervetsky D. and Olivero A.. ObsSlice: A timed automata slicer based on Observers. InProc. of the 16th Intl. Conf. CAV~'04, LNCS. Springer Verlag, 2004.
[BHV00] G. Behrmann, T. Hune and F. W. Vaandrager. Distributing timed model checking - how the search order matters. Computer Aided Verication, volume 1855 of LNCS, pages 216-231. 2000.
[BLI96] A. Bouajjani, Y. Lakhnech and S. Yovine. Model Checking for Extended Time Temporal Logics. En Proc. of 4th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, . 1996.
[BLL95] J. Bengtsson, K. Guldstrand Larsen, F. Larsson, PaulPettersson, and Wang Yi. UPPAAL - a tool suite for automaticverification of real-time systems. In Hybrid Systems, pages:232–243. 1995.
[BLW01] B. Bollig, M. Leucke and M. Weber. Parallel Model Checking for the Alternation Free μ-Calculus. InTools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, 2031: 543-558. Springer Berlin / Heidelberg, 2001.
[BOS04] V. Braberman, A. Olivero, F. Schapachnik. On-the-fly Workload Prediction and Redistribution in the Distributed Timed Model Checker Zeus. PDMC: 3rd International Workshop on Parallel and Distributed Methods in verification, ENTCS(128): 13-18. Septiembre 2004.
[BOS06] V. Braberman, A. Olivero, F. Schapachnik. Dealing with Practical Limitations of Distributed Timed Model Checking for Timed Automata. Formal Methods in System Design, 29(2): 197-214. 2006.
[Bra00] V. Braberman. Modeling and Checking Real-Time Systems Designs. Phd. thesis, Departamento de Computaci´on, Facultad de Ciencias Exactas yNaturales, Universidad de Buenos Aires. 2000.
[CB02] C. P. Inggs, H. Barringer. Effective State Exploration for Model Checking on a Shared

Memory Architecture. InProceedings Workshop on Parallel and Distributed Model Checking (PDMC 2002), Electronic Notes in Theoretical Computer Science , 68(4). Elsevier Science, 2002.
[CB06] C. P. Inggs, H. Barringer. CTL* model checking on a shared-memory architecture. Formal Methods in System Design, 29(2): 135-155. 2006.
[CES71] Coffman, E.G., M.J. Elphick, and A. Shoshani. System Deadlocks. ACM Computing Surveys, 3(2): 67-78. 1971.
[DIL90] D. L. Dill. Timing assumptions and verification of finitestate concurrent systems. In International Workshop of Automatic Verification Methods for Finite State Systems. , LNCS(407): 197 212. 1990.
[DOT96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. InProceedings of Hybrid Systems III, volume 1066 ofLNCS, pages:208–219. Springer-Verlag, 1996.
[EC81] E. A. Emerson and E. Clarke. Design and synthesis of synchronization skeletons usingbranching-time temporal logics. InWorkshop of Logics of Programs, volume 131 of LNCS, . Springer-Verlag, 1981.
[FLO62] R. Floyd. Acm algortithm 97: Shortest path. Communications of ACM, 5(6): 345. 1962.
[HNS92] T. A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine. Symbolic Model Checking for Real-Time Systems. En Proc. of 7th Symp. on Logicin Computer Science. IEEE Computer Society Press, pages: 394-406. 1992.
[Kna28] B. Knaster. Un theoreme sur les fonctions des ensembles. Ann. Soc. Polon. Math, 6: 133-134. 1928.
[Koz83] D. Kozen. Results on the propositional mu-calculus. Theoretical Computer Science, 27: 333–354. 1983.
[LL05] M. Lange, H. W. Loidl. Parallel and Symbolic Model Checking for Fixpoint Logic with Chop. InProc. of the 3rd Int. Workshop on Parallel and Distributed Methods in Verification, PDMC'04, London, UK. Elect. Notes in Theor. Comp. Science, 128(3): 125-138. Elsevier Science, 2005.
[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. InProc. of the 5th International SPIN Workshop, volume 1680 of LNCS. Springer-Verlag, 1999.
[MAR99] A. D. Marshall. Programming in C. UNIX System Calls and Subroutines using C. , . 1994-2005.
[Sch02] F. Schapachnik. Verificación distribuida y paralela de sistema de tiempo real. Tesis de licenciatura, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Junio 2002.
[SCH02] F. Schapachnik. Verificación distribuida y paralela de sistema de tiempo real. Tesis de Licenciatura, Departamento deComputación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Junio 2002.
[SCH02] F. Schapachnik. Verificación distribuida y paralela de sistema de tiempo real. Tesis de Licenciatura, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Junio 2002.
[SD97] U. Stern and D. Dill. Parallelizing the Murφ verifier. Computer Aided Verification, 9th International Conference, Springer-Verlag, 1254: 256-267. 1997.
[SJI05] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill. A New Reachability Algorithm for Symmetric Multi-processor Architecture. 4th International Workshop on Parallel and Distributed Methods in verification PDMC 2005. 2005.
[Str] http://lafhis.dc.uba.ar/obsslice/examples.html#Struct
[Tar55] A. Tarski. A lattice theoretical xpoint theorem and its applications. Pacic Journal of Math, 5: 285-309. 195.
[Yov97] S. Yovine. Model-checking timed automata. InG. Rozenberg and F. Vaandrager,editors, Embedded Systems, volume 1494 of LNCS. Springer-Verlag, .