# Marco Teórico para Demostrar Propiedades de Programas sobre Memoria Compartida Distribuida

**Jorge Baier, José M. González y Ricardo Guzmán**
Departamento de Ciencia de la Computación,
Pontificia Universidad Católica de Chile,
Casilla 306, Santiago 22, Chile.
FAX: (562) 6864444
Email: {**jabaier, jmgonzal, raguzman**}**@ing.puc.cl**

**Resumen**

Es de especial relevancia al programar bajo memoria compartida distribuida, ser capaz de saber si los programas que se construyen bajo estos modelos funcionan como se espera o no.

En este artículo se propone una forma de modelar en un lenguaje lógico (el Cálculo de Situaciones) dos modelos de consistencia: consistencia secuencial y consistencia PRAM. Luego se propone una manera de traducir programas a esta lógica, utilizando un lenguaje de programación construido sobre ella. Se muestra que los modelos efectivamente representan los tipos de consistencia representados. Finalmente, se da argumentos en favor de el uso de estos modelos para demostrar propiedades de programas de manera formal.

Este artículo pretende dar un método para representar programas y no una forma de demostrar propiedades.

## 1 Introducción

Al utilizar memoria compartida distribuida [9, 10] se supone que se cuenta con una colección de estaciones de trabajo que comparten un único espacio de direccionamiento. Dentro de este contexto, se ha creado distintos *modelos de consistencia* [1], los cuales son esencialmente contratos entre los programas y la memoria. Estos contratos establecen reglas que debe cumplir el software para que la memoria funcione de una cierta manera.

Es de especial relevancia al programar bajo memoria compartida distribuida, ser capaz de saber si los programas que se construyen bajo estos modelos funcionan como se espera o no.

En este artículo se propone una forma de modelar en un lenguaje lógico (el Cálculo de Situaciones) dos modelos de consistencia: consistencia secuencial y consistencia PRAM. Luego se propone una manera de traducir programas a esta lógica, utilizando un lenguaje de programación construido sobre ella (con esto no se pretende ejecutar los programas sobre lógica, sino ser capaz de hablar sobre ellos). Se mostrará que los modelos efectivamente representan los tipos de consistencia representados. Finalmente, se darán argumentos en favor del uso de estos modelos para demostrar propiedades de programas de manera formal.

El uso del Cálculo de Situaciones [12] se justifica ya que éste es un lenguaje de alta expresividad sobre el cual se ha realizado bastante investigación en los últimos años. Ha habido trabajos sobre demostración de correctitud de algoritmos de exclusión mutua usando esta lógica [4].

El artículo se estructura de la siguiente manera: en la sección 2 se da una introducción al lenguaje de programación GOLOG, en la sección 3 se describe el Cálculo de Situaciones, en la sección 4 se especifica un método de traducción de programas concurrentes a GOLOG, en la sección 5 se modela en lógica el modelo de consistencia secuencial, en la sección 6 de modela la consistencia PRAM. Finalmente, en la

sección 7 se dan conclusiones y se habla sobre posible trabajo futuro.

## 2 Introducción al Lenguaje GOLOG

GOLOG [8, 7] es un lenguaje interpretado basado en lógica originalmente diseñado para modelar mundos dinámicos en inteligencia artificial. La semántica del lenguaje está basada en el Cálculo de Situaciones, un lenguaje de lógica de segundo orden, el cual será descrito en las próximas secciones.

En la versión de GOLOG utilizada en este artículo, se distingue los siguientes elementos:

**Acciones Primitivas y Condiciones de Espera**

- Acciones primitivas, denotados por la letra griega $\alpha$ (posiblemente con subíndices). Estas acciones pueden ser un conjunto de operaciones atómicas ejecutadas en paralelo (por distintos procesadores, por ejemplo). Corresponden exactamente a las acciones concurrentes del Cálculo de Situaciones (introducidas más adelantes). Son instantáneas.

- $\phi$?: condiciones de prueba/espera. Aquí, $\phi$ es una fórmula de lógica de primer orden que no contiene términos de situación, aunque en la semántica subyacente se agrega el término de situación correspondiente a la situación en la que se encuentra el programa cuando la condición es verificada. En GOLOG, estas fórmulas cumplen la misma función que las expresiones booleanas en lenguajes imperativos tradicionales como C o Pascal. La principal diferencia es que su poder de expresividad es mayor, ya que son fórmulas de primer orden.

**Acciones Complejas** Pueden ser vistas como *programas*. Serán denotadas por las letras griegas $\sigma$ y $\delta$ y se definen por:

- $\alpha$, una acción primitiva es una acción compleja.

- Si $\sigma_1$ y $\sigma_2$ son acciones complejas, entonces las siguientes también son acciones complejas:
  - $(\sigma_1;\sigma_2)$, son *secuencias de acciones*. La ejecución de esta secuencia corresponde a la ejecución de $\sigma_1$ seguida de $\sigma_2$.
  - $(\sigma_1||\sigma_2)$, son *programas concurrentes*. Corresponden a la ejecución concurrente de las acciones complejas $\sigma_1$ y $\sigma_2$.
  - $\sigma^*$, es una *iteración no determinística*. La acción compleja $\sigma$ puede ser ejecutada un número arbitrario de veces.
  - **if** $\phi$ **then** $\sigma_1$ **else** $\sigma_2$, es una *sentencia condicional*. $\sigma_1$ es ejecutada si $\phi$ es verdadera, en otro caso, $\sigma_2$ es ejecutada.
  - **while** $\phi$ **do** $\sigma$: *iteraciones while*. $\sigma$ es ejecutado mientras $\phi$ es verdadero.

**Definición de Procedimientos** Un procedimiento en GOLOG tiene la siguiente forma:

$$\textbf{proc}\ \beta(\vec{x})\ \sigma\ \textbf{end},$$

donde $\beta$ es el nombre del procedimiento, $\vec{x}$ son los argumentos y $\sigma$ es el cuerpo del procedimiento.

**Definición 1** *Un programa* GOLOG *es un conjunto de cero o más definiciones de procedimientos seguidos de una acción compleja, la cual corresponde al programa principal. Un programa* GOLOG *se ve de la siguiente manera:*

$$\textbf{proc}\ P_1(\vec{v}_1)\ \sigma_1\ \textbf{end};\ldots\textbf{proc}\ P_n(\vec{v}_n)\sigma_n\ \textbf{end};\ \sigma$$

### 2.1 Semántica de GOLOG

La semántica de GOLOG está basada en el Cálculo de Situaciones.

Para definir la semántica del lenguaje, se utilizan los predicados *Final* y *Trans* tal como en [5], donde $Final(\sigma,s)$ significa que el programa $\sigma$ puede terminar en la situación $s$. $Trans(\sigma,s,\sigma',s')$ es verdadero cuando el programa $\sigma$ puede ejecutar un paso (transición) en la situación $s$, terminando en la situación $s'$ quedando $\sigma'$ aún por ejecutar.

En la axiomatización, se define el valor de verdad de *Trans* y *Final* para cada tipo de acción compleja del lenguaje. Tal como en [3], una transición puede corresponder a la ejecución de una acción en un programa que no es legal (no es posible). La condición de legalidad se impondrá más adelante.

Es particularmente relevante saber que la axiomatización para $(\sigma_1||\sigma_2)$ corresponde, o bien a una transición de $\sigma_1$, o a una transición de $\sigma_2$. Es decir, se logra secuencialidad de la ejecución de ambos programas. Sin embargo, es posible extender esta semántica para que, además, se consideren ejecuciones de ambos programas en paralelo. En las próximas secciones se verá esto en más detalle.

Es posible definir el predicado *Trans**, el cual es la clausura transitiva de *Trans* más la condición de legalidad de transiciones. Así, *s* es la situación resultante de la ejecución de un programa completo $\sigma$, si $Trans^*(\sigma, S_0, \{\}, s)$ es verdadero. Aquí, $\{\}$ es el programa vacío. $S_0$ es la situación inicial y será descrita en la próxima sección.

# 3 El Cálculo de Situaciones y Concurrencia

Esta sección describe el Cálculo de Situaciones Concurrente [13, 15], el cual extiende el Cálculo de Situaciones original con un tipo (*sort*) para acciones concurrentes.

## 3.1 Marco Formal

En la terminología utilizada tradicionalmente en el Cálculo de Situaciones, las acciones *concurrentes* son acciones primitivas (simples) ejecutadas en paralelo.

El cálculo de situaciones es un lenguaje de segundo orden con tipos $\mathcal{A}$, $\mathcal{C}$, $\mathcal{S}$ y $\mathcal{D}$ para acciones atómicas, acciones concurrentes, situaciones y objetos del dominio. Existe una situación inicial distinguida, denotada por la constante $S_0$. La función *do* toma una acción concurrente y una situación y entrega otra situación, que corresponde a la situación resultante de la ejecución de dicha acción en la primera situación. Los fluentes son predicados que toman un argumento de tipo situación y representan propiedades que son estáticas dentro de una situación, pero que pueden cambiar a entre situaciones. Además, se utiliza el predicado *Poss*, que toma como argumentos a una acción concurrente y una situación y es verdadero cuando la acción es ejecutable en esa situación.

Los axiomas fundacionales para el Cálculo de Situaciones Concurrente definen la estructura de las situaciones. La definición establece que existe una única situación inicial, llamada $S_0$ y que cualquier situación existente es obtenida a través de la función *do*. Así, si $c_1 \ldots c_n$ son acciones concurrentes, entonces $do(c_1, S_0)$, $do(c_1, do(c_1, S_0))$ y $do(c_1, do(c_3, do(c_4, S_0)))$[1] son situaciones. Además, los axiomas fundacionales establecen que todas las situaciones que existen son aquellas que resultan de la aplicación de una cantidad finita cualesquiera de acciones en la situación inicial $S_0$ (para establecer esto se necesita el axioma de segundo orden). Finalmente, los axiomas fundacionales establecen un orden parcial entre situaciones. Así, si $s_1$ y $s_2$ son situaciones, entonces $s_1 < s_2$ será verdadero si $s_1$ está en el camino que lleva hasta $s_2$ desde $S_0$.

Las acciones concurrentes (paralelas) son tratadas como conjuntos de acciones primitivas. Así, $\in$ es usada como relación entre acciones atómicas y acciones concurrentes. Escribimos $a \in c$ para decir que la acción atómica *a* es parte de la acción concurrente *c*.

En la siguiente sección se describe brevemente como se escriben teorías de acción en el Cálculo de Situaciones Concurrente. El enfoque de este artículo está basado en la solución al problema del marco dada por Reiter[14].

## 3.2 Axiomatización del Dominio

Una axiomatización de dominio es dividida en varios conjuntos de axiomas. Primero se tiene un conjunto de axiomas ($T_d$) que menciona sólo términos del tipo $\mathcal{D}$. Además, se utiliza un conjunto para nombres axiomas de nombres únicos $T_{una}$. El resto de los conjuntos de axiomas se describe a continuación.

**Axiomas de Precondición para Acciones** Los axiomas de precondición para acciones se agrupan en el conjunto $T_{prec}$. Dichos axiomas tienen la si-

[1] Normalmente se utiliza la abreviatura $do([c_1, c_2, c_3], S_0)$ para $do(c_3, do(c_2, do(c_1, S_0)))$.

[2] Se escribe $\underline{\mathbf{x}}$ para denotar una tupla de variables del dominio

guiente forma[2]:

$$Poss(\{a_l(\underline{\mathbf{x}})\}, s) \equiv \Phi_{a_l}(\underline{\mathbf{x}}, s),$$

donde $a_l(\underline{\mathbf{x}})$ es un término en $\mathcal{A}$, y $\Phi_{a_l}(\underline{\mathbf{x}}, s)$ es una fórmula simple en $s$[3]. Las precondiciones para acciones son formuladas como condiciones necesarias y suficientes para *Poss*.

**Ejemplo**:

$$Poss(drop(x), s) \equiv holding(x, s)$$

Es posible ejecutar la acción $drop(x)$ en la situación $s$ si y sólo si es verdadero que $holding(x, s)$

**Axiomas de Efecto** El conjunto $T_{eff}$ de *axiomas de efecto* especifica los efectos directos de acciones primitivas sobre el mundo, sin considerar los efectos de otras acciones que se ejecuten posiblemente en paralelo.

Un axioma de efecto positivo tiene la forma sintáctica (1), mientras que un axioma de efecto negativo tiene la forma (2).

$$Poss(c, s) \wedge A \in c \wedge G_f^+(\underline{\mathbf{x}}, A, s) \supset f(\underline{\mathbf{x}}, do(c, s)), \quad (1)$$

$$Poss(c, s) \wedge A \in c \wedge G_f^-(\underline{\mathbf{x}}, A, s) \supset \neg f(\underline{\mathbf{x}}, do(c, s)). \quad (2)$$

**Ejemplo** (cont.):

$$Poss(c, s) \wedge drop(x) \in c \supset broken(x, do(c, s))$$

Si $drop(x)$ se ejecuta en $s$, entonces $broken(x, do(c, s))$ será verdadero si $drop(x)$ pertenece a $c$.

Todos los axiomas de efecto pueden ser agrupados en un solo axioma de efecto positivo y otro negativo para cada fluente:

$$Poss(c, s) \wedge \gamma_f^+(\underline{\mathbf{x}}, c, s) \supset f(\underline{\mathbf{x}}, do(c, s)),$$
$$Poss(c, s) \wedge \gamma_f^-(\underline{\mathbf{x}}, c, s) \supset \neg f(\underline{\mathbf{x}}, do(c, s)).$$

Si no existen axiomas de efecto positivos (o negativos) para un fluente $f$, entonces la fórmula $\gamma_f^+(\underline{\mathbf{x}}, c, s)$ (o $\gamma_f^-(\underline{\mathbf{x}}, c, s)$ para el caso negativo) es falsa.

Los axiomas de efecto, si bien describen cuándo cambian los valores de verdad de los fluentes, no especifican cuándo no cambian. Para definir completamente el cambio de valor de verdad de los fluentes se pueden generar *axiomas de marco* [14], los cuales tienen la siguiente forma:

$$Poss(c, s) \supset [f(\underline{\mathbf{x}}, do(c, s)) \equiv (\gamma_f^+(\underline{\mathbf{x}}, c, s) \vee f(\underline{\mathbf{x}}, s) \wedge \neg\gamma_f^-(\underline{\mathbf{x}}, c, s))].$$

# 4 Programas Concurrentes y su Traducción a Golog

Se considera que puede existir un número arbitrario de procesadores que pueden ejecutar programas que alteran la memoria compartida. Cada programa es una secuencia de sentencias que pueden leer o escribir en la memoria. Para cada variable existe una dirección única asociada en la memoria compartida. Inicialmente se supone que el valor de todas las variables es 0.

**Ejemplo** Un programa ejecutado en memoria compartida:

```
a := 1;
while (b <> 1)
 print(a);
```

En general, es deseable demostrar propiedades sobre la memoria o sobre lo que los procesos saben (son capaces de leer) de ella. Así, las asignaciones corresponden a escrituras en la memoria, las sentencias del tipo $print(a)$ a una lectura de la variable $a$, y una comparación a una lectura seguida de una acción de comparación. Más formalmente:

- Si la sentencia `a := v` es parte de un programa ejecutado por un procesador $P$, la sentencia GOLOG equivalente (acción primitiva) es $write(P, A, V)$. Aquí `v` es un valor constante (numérico) y $A$ es un constante lógica que representa la posición de memoria de la variable `a`.

[3]Una fórmula es simple en un término de situación $s_t$, si no menciona ningún otro término de situación aparte de $s_t$, y no cuantifica sobre $s_t$.

- Si la sentencia `print(a)` es parte de un programa ejecutado por un procesador $P$, la sentencia GOLOG equivalente es $read(P,A,x)$. $x$ es una variable cualquiera. Durante la traducción de más de un comando de lectura de memoria no se deben repetir los nombres de las variables.

- Si la sentencia es `if a = v then <sent1> else <sent2>`, la traducción equivalente es

  **if** $writtenTo(A,V)$? **then** $\sigma_1$ **else** $\sigma_2$

  donde $\sigma_1$ y $\sigma_2$ corresponden a la traducción de las sentencias `<sent1>` y `<sent2>`, y `v` es un valor constante.

- Si la sentencia es `while a = v do <sent>`, se escribe

  **while** $writtenTo(A,V)$? **do** $\sigma$,

  donde $x$ es una variable arbitraria, no utilizada en el resto del programa y $\sigma$ es la traducción de `<sent>`. Aquí se permite cualquier tipo de expresiones booleanas. La traducción de ellas es directa, ya que sólo hay que cambiar los operadores `and`, `or` y `not` por sus correspondientes en lógica.

**Ejemplo** (cont.): La traducción del programa del ejemplo es:

$write(P1,A,1);$
**while** $\neg writtenTo(B,1)$? **do** $read(P1,A,y);$

Del método de traducción es inmediato que se utilizará sólo dos acciones primitivas GOLOG para describir programas concurrentes. Éstas son $read(p,x,y)$ que corresponde a la lectura que realiza el procesador $p$ del valor $y$ desde la ubicación de memoria $x$, y $write(p,x,y)$ que es la escritura del valor $y$ en la ubicación $x$ por el procesador $p$.

Dependiendo del modelo de consistencia utilizado, los efectos sobre la memoria de *read* y *write* serán distintos. En la siguientes secciones se verá cómo se realiza esto para dos modelos de consistencia.

# 5 Consistencia Secuencial

El modelo de consistencia secuencial fue introducido por Lamport [6] y satisface la siguiente condición:

> El resultado de cualquier ejecución es el mismo que si todos los procesadores ejecutaran en algún orden secuencial, manteniéndose el orden que las operaciones tenían dentro de los programas.

Esta definición se reduce a que la ejecución de programas concurrentes debe cumplir las siguientes condiciones:

1. Las instrucciones se deben ejecutar en el orden impuesto por el programa.

2. La ejecución de dos programas paralelos corresponde a una mezcla de la de ambos programas.

3. Hay coherencia en memoria, es decir, las lecturas a una ubicación de memoria deben retornar el valor más recientemente escrito en dicha ubicación.

La primera condición está garantizada por la definición de la semántica del lenguaje GOLOG, los programas siempre se ejecutan en orden.

Debido a que los programas concurrentes ejecutan secuencialmente, se escribe el siguiente axioma de definición de semántica de GOLOG para las acciones complejas concurrentes:

$$\begin{aligned} &Trans((\sigma_1||\sigma_2),s,\delta,s') \equiv \\ &\quad \exists\delta'.\delta = (\delta'||\sigma_2) \wedge Trans(\sigma_1,s,\delta',s') \vee \\ &\quad\quad \delta = (\sigma_1||\delta') \wedge Trans(\sigma_2,s,\delta',s'). \end{aligned}$$

Aquí se establece que una transición de un programa concurrente $(\sigma_1||\sigma_2)$ corresponde a ejecutar una transición de $\sigma_1$ o una transición de $\sigma_2$. Este axioma garantiza que si los programas sólo contienen acciones simples, siempre se ejecutarán acciones simples, es decir, no habrá paralelismo entre los programas.

La tercera condición se satisfará luego de agregar los siguientes axiomas:

**Axiomas de Precondición para** *read* **y** *write*

$$Poss(read(p,x,y),s) \equiv writtenTo(x,y,s). \quad (3)$$
$$Poss(write(p,x,y),s) \equiv True. \quad (4)$$

Se introduce el fluente $writtenTo(x,y,s)$, el cual es verdadero si el valor $y$ está escrito en la posición de memoria $x$ en la situación $s$. Los axiomas dicen que es posible leer un valor $y$ desde la ubicación de memoria $x$ si dicho valor está escrito en $x$ y que siempre es posible escribir en memoria.

**Axiomas de Efecto Directos** Si la acción $write(p,x,y)$ es ejecutada, $y$ estará escrito en la posición $x$, y si existía otro valor escrito, dejará de estarlo.

$$Poss(c,s) \wedge write(p,x,y) \in c \supset writtenTo(x,y,do(c,s)), \quad (5)$$

$$Poss(c,s) \wedge write(p,x,y) \in c \wedge writtenTo(x,z,s) \wedge y \neq z \supset \neg writtenTo(x,z,do(c,s)) \quad (6)$$

La acción *read* no tiene efectos sobre la memoria.

A partir de estos axiomas se genera el siguiente axioma de marco para *writtenTo*:

$$Poss(c,s) \supset [writtenTo(x,y,do(c,s)) \equiv write(p,x,y) \in c \vee writtenTo(x,y,s) \wedge \neg(write(p,x,z) \in c \wedge y \neq z)] \quad (7)$$

En la situación inicial, las variables tienen valor 0:

$$writtenTo(x,0,S_0).$$

**Teorema 1** *Sea $\Sigma$ una teoría que incluye la axiomatización dada por los axiomas fundacionales del Cálculo de Situaciones, axiomas de precondición ($T_{prec}$), de marco ($T_{ssa}$) y de descripción inicial $T_{S_0}$, más los axiomas de la semántica de* GOLOG *garantizan coherencia en memoria. Es decir,*

$$\Sigma \models Poss(read(p,x,y),s) \supset \exists c,s' do(c,s') < s \wedge write(p,x,y) \in c \wedge \neg\exists c',s'' do(c,s') < do(c',s'') < s \wedge write(p,x,z) \in c$$

DEMOSTRACIÓN: *Si $Poss(read(p,x,y),s)$ es verdadero, entonces, en todo modelo de $\Sigma$, por* (3)*, $writtenTo(x,y,s)$ también lo es. Mirando la estructura de* (7)*, la única posibilidad de que esto sea verdadero es que exista un situación en el pasado de $s$ en la que se ejecutó $write(p,x,y)$, y, más aún, no se ha escrito desde entonces otro valor en la posición $x$.* □

**Ejemplo 1** Suponga que los siguientes programas se ejecutan concurrentemente sobre una memoria distribuida bajo consistencia secuencial:

```
P1:                 P2:

a := 1;             b := 1;
print(b);           print(a);
while (b <> 1)      if a = 1
  print(b);           print(a)
                    else
                      print(c);
```

Sean $\sigma_1$ y $\sigma_2$ los programas GOLOG para los programas `P1` y `P2`. Entonces:

$\sigma_1 \stackrel{def}{=} write(P_1,A,1);$
$read(P_1,B,x);$
**while** $\neg writtenTo(B,1)?$ **do** $read(P_1,B,y)$

$\sigma_2 \stackrel{def}{=} write(P_2,B,1);$
$read(P_2,A,z);$
**if** $\neg writtenTo(A,1)?$ **then** $read(P_2,A,y)$
**else** $read(P_2,C,y)$

**Proposición 1** *Sea $\sigma = (\sigma_1||\sigma_2)$, entonces $S_1$ y $S_2$ son dos posibles ejecuciones de $\sigma_1$ y $\sigma_2$. Donde $S_1$ y $S_2$ se definen como:*

$$S_1 = do([write(P_1,A,1), write(P_2,B,1), read(P_2,A,1), read(P_2,A,1), read(P_1,B,1)], S_0)$$
$$S_2 = do([write(P_1,A,1), read(P_1,B,0), write(P_2,B,1), read(P_1,B,1), read(P_2,A,1), read(P_2,A,1)], S_0)$$

DEMOSTRACIÓN: *Sigue de la definición de semántica de* GOLOG *y los axiomas de marco, precondición y fundacionales del Cálculo de Situaciones.*

# 6 Consistencia PRAM

El modelo de consistencia PRAM se debe a Lipton y Sandberg [11]. La consistencia de la memoria está sujeta a la siguiente condición:

> Las escrituras hechas por un procesador único son recibidas por los otros procesos en el orden en que éstas fueron hechas, pero las escrituras de diferentes procesadores pueden ser vistas en un orden diferente por procesadores diferentes.

A diferencia del modelo anterior, en PRAM no se exige que el resultado de un programa corresponda a una ejecución secuencial, por lo tanto, se permite que exista paralelismo entre las operaciones atómicas de distintos procesadores.

De la definición se desprende que se debe cumplir las siguientes condiciones:

1. Es posible que existan ejecuciones paralelas de las distintas operaciones atómicas de los procesadores (*write*s y *read*s).

2. Debe haber consistencia en memoria en escrituras hechas por un mismo procesador. Es decir, si un procesador lee un valor que él ha escrito desde una posición de memoria, éste debe ser el último que ha escrito en tal posición.

3. Si un procesador $P_1$ lee un valor escrito por otro procesador $P_2$ en alguna posición de memoria, el próximo valor que $P_1$ lea desde esa misma posición escrito por $P_2$, debe haber sido escrito por $P_2$ en una escritura posterior o ser el mismo que ya había leído.

Para representar este tipo de consistencia en el Cálculo de Situaciones, se supone que cada procesador, en cualquier instante (no determinísticamente), "cuenta" a los demás procesos cuáles son los valores que ha escrito en su memoria. Cuando un proceso cuenta a los demás sus valores, los demás procesadores actualizan los valores de sus variables. De esta manera, se logra que los procesos vean en orden los cambios en variables de otros procesos.

## 6.1 Modelo en el Cálculo de Situaciones

En esta sección muestra cómo modelar este modelo de consistencia en el Cálculo de Situaciones.

**Acciones**

- $write(p,x,y)$: El procesador $p$ escribe el valor $y$ en la posición de memoria $x$.

- $read(p,x,y)$: El procesador $p$ lee el valor $y$ desde la posición de memoria $x$.

- $broadcast(p,x,y)$: El procesador $p$ comunica a los demás procesadores que el valor que conoce de la posición de memoria $x$ es $y$.

**Fluentes**

- $writtenTo(p,x,y,s)$: Una variación de $writtenTo(x,y,s)$ del modelo secuencial. Es verdadero cuando $y$ es el valor de memoria que $p$ conoce que está en la posición de memoria $x$.

**Axiomas de Precondición**

$$Poss(read(p,x,y),s) \equiv writtenTo(p,x,y,s). \tag{8}$$

$$Poss(write(p,x,y),s) \equiv True. \tag{9}$$

$$Poss(broadcast(p,x,y),s) \equiv writtenTo(p,x,y,s). \tag{10}$$

La acción $read(p,x,y)$ es posible si el $y$ es el valor que $p$ conoce que está en $x$. La acción *write* es siempre posible y $broadcast(p,x,y)$ es posible si $p$ considera que $y$ es el valor que hay en $x$.

Es necesario agregar el siguiente axioma para acciones concurrentes:

$$Poss(c,s) \equiv [\forall a(a \in c) \supset Poss(a,s)]. \tag{11}$$

Es decir, una acción concurrente es posible si todas las acciones que la componen son posibles.

**Axiomas de Efecto Directos**

$$Poss(c,s) \wedge write(p,x,y) \in c \supset writtenTo(p,x,y,do(c,s)), \quad (12)$$

$$Poss(c,s) \wedge (write(p,x,y) \in c \vee \exists p' broadcast(p',x,y) \in c) \wedge writtenTo(p,x,z,s) \wedge y \neq z \supset \neg writtenTo(p,x,z,do(c,s)) \quad (13)$$

$$Poss(c,s) \wedge \exists p' broadcast(p',x,y) \in c \supset writtenTo(p,x,y,do(c,s)) \quad (14)$$

Si la acción $write(p,x,y)$ es ejecutada, $y$ estará escrito en la posición $x$ (12) y si existía otro valor escrito, dejará de estarlo (13). Por otra parte, si un procesador envía un broadcast, todos los demás procesadores actualizan sus valores (14).

**Axiomas de Estado Sucesor** A partir de los axiomas de efecto, se genera el siguiente axioma de estado sucesor para *writtenTo*.

$$Poss(c,s) \supset [writtenTo(p,x,y,do(c,s)) \equiv write(p,x,y) \in c \vee \exists p' broadcast(p',x,y) \in c \vee writtenTo(x,y,s) \wedge \neg((write(p,x,z) \in c \vee \exists p' broadcast(p',x,y) \in c) \wedge y \neq z)] \quad (15)$$

**Caracterización de la Situación Inicial** Tal como en el modelo anterior, en la situación inicial, todas las variables tienen valor 0.

$$writtenTo(p,x,0,S_0). \quad (16)$$

Para establecer el hecho de que, ahora, las operaciones de los procesadores se pueden ejecutar en paralelo, se modifica el axioma de semántica de GOLOG para $(\sigma_1||\sigma_2)$ tal como en [3]:

$$\begin{aligned} &Trans((\sigma_1||\sigma_2),s,\delta,s') \equiv \\ &\quad \exists \delta_1,\delta_2,c_1,c_2(Trans(\sigma_1,s,\delta_1,do(c_1,s)) \wedge \\ &\qquad Trans(\sigma_2,s,\delta_2,do(c_2,s)) \wedge \\ &\qquad s' = do(c_1 \cup c_2,s) \wedge \delta = (\delta_1||\delta_2)) \vee \\ &\quad \exists \delta'.\delta = (\delta'||\sigma_2) \wedge Trans(\sigma_1,s,\delta',s') \vee \\ &\qquad \delta = (\sigma_1||\delta') \wedge Trans(\sigma_2,s,\delta',s'). \end{aligned} \quad (17)$$

## 6.2 Ejecutando Programas

En el modelo anterior, para modelar la ejecución de dos programas, bastaba con traducirlos a GOLOG y luego ejecutarlos concurrentemente. En este modelo, esto no es suficiente, debido a se tiene que considerar que cada proceso puede hacer un broadcast de las variables que ha modificado en cualquier momento. Para modelar este fenómeno, se dice que si $\sigma_1$ corresponde a la ejecución de un programa concurrente (como los de la sección 4) en un procesador $P_1$, entonces la ejecución de dicho programa queda modelado por $(\sigma_1||\sigma_1^b)$ donde $\sigma_1^b$ se define de la siguiente manera:

$$\sigma_1^b \stackrel{def}{=} (broadcast(P_1,x_1,y_1)|| broadcast(P_1,x_2,y_2)|| \ldots broadcast(P_1,x_n,y_n))^* \quad (18)$$

donde $x_1,\ldots x_n$ son todas las variables que aparecen modificándose en alguna parte del cuerpo de $\sigma_1$.

**Ejemplo 2** Considere los mismos programas del ejemplo 1 y sean $\sigma_1$ y $\sigma_2$ definidos de la misma manera. Entonces $\sigma_1' \stackrel{def}{=} (\sigma_1||\sigma_1^b)$ y $\sigma_2' \stackrel{def}{=} (\sigma_2||\sigma_2^b)$ con

$$\sigma_1^b \stackrel{def}{=} (broadcast(P_1,A,y))^*$$
$$\sigma_2^b \stackrel{def}{=} (broadcast(P_2,B,y))^*$$

**Proposición 2** *Sea* $\sigma' = (\sigma_1'||\sigma_2')$, *entonces* $S_1$ *y* $S_2$ *son ejecuciones de* $\sigma_1$ *y* $\sigma_2$. *Donde* $S_1$ *y* $S_2$ *se definen*

[4]Se han omitido los paréntesis de llave cuando la acción concurrente tiene una sola componente

*como*[4]*:*

$$S_1 = do([\{write(P_1,A,1), write(P_2,B,1)\},$$
$$read(P_2,A,0), read(P_2,C,0),$$
$$broadcast(P_2,B,1), read(P_1,B,1)], S_0)$$
$$S_2 = do([write(P_1,A,1), read(P_1,B,0),$$
$$write(P_2,B,1), read(P_1,B,1),$$
$$read(P_2,A,1), read(P_2,A,1)], S_0)$$

DEMOSTRACIÓN: *Sigue de la definición de semántica de* GOLOG *y los axiomas de marco, precondición y fundacionales del Cálculo de Situaciones.*

Al comienzo de esta sección se enumera tres principios que debe cumplir un modelo para PRAM. El primero se cumple a través del axioma (11). El segundo se cumple por la misma razón que en el modelo de consistencia secuencial. Ahora se demuestra que también el tercero se cumple.

**Teorema 2** *Sea* $\Sigma$ *una teoría que incluye la axiomatización dada por los axiomas fundacionales del Cálculo de Situaciones, axiomas de precondición* ($T_{prec}$)*, de marco* ($T_{ssa}$) *y de descripción inicial* $T_{S_0}$*, más los axiomas de la semántica de* GOLOG*. Entonces los procesadores ven en orden las escrituras de otro procesador.*

DEMOSTRACIÓN: *La única manera que un procesador vea la escritura realizada por otro es que en algún momento se ejecute la operación* $broadcast(p,x,y)$*, para algún valor de x e y. Por* (10) *se sabe que esto es posible siempre que* $writtenTo(p,x,y)$*. Además se sabe que, por la propiedad de coherencia de memoria, el valor y en x es el último que p conoce. Esto implica si en s, p hace* $broadcast(p,x,y)$*, en el futuro nunca hará* $broadcast(p,x,y')$ *donde* $y'$ *es un valor que estaba en x antes que y porque* $writtenTo(p,x,y')$ *no sería verdadero en tal situación (de hecho no es verdadero en s).* □

## 7 Conclusiones y Trabajo Futuro

En este artículo se ha presentado un método para traducir programas concurrentes a un lenguaje lógico. Se ha visto, además, cómo modelar dos modelos de consistencia de memoria dentro de este mismo lenguaje.

Lo anterior constituye el primer paso para ser capaz de demostrar propiedades de los programas (por ejemplo, que el programa hace lo que se pretende). Como parte de futuros trabajos, se pretende modelar más modelos de consistencia y dar un método sencillo para demostrar diversas propiedades de programas. Se pretende también, en el futuro, explorar la posibilidad de realizar las demostraciones a través de un demostrador mecánico de teoremas.

Si bien los programas mostrados en este artículo están regidos por modelos de consistencia en memoria, se cree que esta forma de modelar programas también puede ser utilizada para demostrar propiedades de algoritmos que corren bajo diferentes *protocolos* de memoria. Por ejemplo, se cree que es posible demostrar que un protocolo de exclusión mutua es correcto. La ventaja de nuestro enfoque sobre otros propuestos, sería la gran simplicidad que ofrece representar los programas a través de un lenguaje de programación como GOLOG.

Todas las operaciones ejecutadas por programas son atómicas e instantáneas. Aunque esto podría verse como una limitación importante para demostrar propiedades de programas en algunos dominios, es perfectamente posible modelar acciones con duración en el Cálculo de Situaciones [2]. Sin embargo, esto escapa del alcance de este artículo.

## Referencias


[1] ADVE, S., AND HILL, M. Weak Ordering: A New Definition. In *Proc. of the 17th Annual International Symposium on Computer Architecture* (1990), ACM, pp. 2–14.

[2] BAIER, J., AND PINTO, J. Non-instantaneous Actions and Concurrency in the Situation Calculus (Extended Abstract). In *10th European Summer School in Logic, Language and Information* (1998), G. de Giacomo and D.Ñardi, Eds.

[3] BAIER, J., AND PINTO, J. Integrating True Concurrency into the Robot Programming Language Golog. In *Proceedings of the 19th Chi-*

*lean Computer Science Conference* (Talca, Chile, 1999), SCCC.

[4] CAMPOS, A. E., NAVARRETE, C. C., AND PINTO, J. Logical Modelling of a Distributed Mutual Exclusion Algorithm in the Situation Calculus. Submitted to the Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems, Nov. 1999.

[5] GIACOMO, G. D., LESPÉRANCE, Y., AND LEVESQUE, H. Congolog, a concurrent programming language based on the situation calculus: foundations. Submitted for publication, February 1999.

[6] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers C*, 28 (July 1979), 690–691.

[7] LEVESQUE, H., PIRRI, F., AND REITER, R. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science 3*, 18 (1998). http://www.ep.liu.se/ea/cis/1998/018/.

[8] LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F., AND SCHERL, R. B. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming 31* (1997), 59–84.

[9] LI, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986.

[10] LI, K., AND HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems 7*, 28 (Nov. 1989), 321–359.

[11] LIPTON, R. J., AND SANDBERG, J. S. PRAM: A Scalable Shared Memory. Tech. Rep. CS-TR-180-88, Princeton University, September 1988.

[12] MCCARTHY, J., AND HAYES, P. J. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence* **4**, B. Meltzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, Scotland, 1969, pp. 463–502.

[13] PINTO, J. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, Feb. 1994. URL = ftp://ftp.-cs.toronto.edu/~cogrobo/jpThesis.ps.Z.

[14] REITER, R. *The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression.* Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Academic Press, San Diego, CA, 1991, pp. 359–380.

[15] REITER, R. Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference(KR'96)* (Cambridge, Massachussetts, U.S.A., Nov. 1996), Morgan Kaufmann.