Modelación Lógica de un Algoritmo de Exclusión Mutua Distribuido

Alvaro E. Campos, Claudia C. Navarrete, Javier Pinto
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 306, Santiago 22, Chile
{acampos, ccnavarr, jpinto}@ing.puc.cl

Resumen

Se presenta el desarrollo de un modelo lógico para un algoritmo de exclusión mutua distribuido. Este trabajo representa el primer paso hacia el objetivo principal de una investigación en la que se desea desarrollar un lenguaje lógico, basado en lógica de primer orden, que permita describir el comportamiento y analizar diversas propiedades de algoritmos distribuidos.

En el desarrollo de este modelo se utilizó el cálculo de situaciones como lenguaje lógico. Se muestra que la modelación sí es posible, pero con algunas limitaciones expresivas. Estas limitaciones motivan la necesidad de realizar extensiones al lenguaje del cálculo de situaciones para eliminarlas y además enriquecer el lenguaje.

1 Introducción

Los sistemas distribuidos presentan hoy en día un gran desafío respecto al desarrollo de aplicaciones que permitan aprovechar el poder computacional que éstos ofrecen [5]. Debido a la inherente complejidad de este tipo de sistemas, esta tarea no es sencilla. Por este motivo, en los últimos años se ha desarrollado una serie de algoritmos concebidos para solucionar diversos problemas presentes en la mayoría de los sistemas distribuidos.

Al desarrollar estos algoritmos se utilizan lenguajes informales para la especificación del problema que se desea resolver. Pero ¿cómo asegurar que los algoritmos satisfacen sus respectivas especificaciones?. Dado que las especificaciones son, en muchos casos, informales, las demostraciones de corrección son también informales.

Lo ideal sería contar con demostradores mecánicos, que permitieran determinar automáticamente si la implementación de un determinado algoritmo cumple con su especificación.

En esta investigación se desea desarrollar un lenguaje basado en lógica de primer orden para especificar las tareas a resolver por estos algoritmos, junto con mecanismos deductivos para el análisis de las propiedades de dichos algoritmos. Se pretende de esta manera proveer, a los desarrolladores de algoritmos, herramientas que aseguren el cumplimento de su propia especificación. Lo anterior permitiría una completa dedicación al desarrollo de aplicaciones innovadoras, sin tener que dedicar recursos a la construcción y análisis de algoritmos que ya estarían disponibles.

Este trabajo muestra un primer paso hacia este objetivo al modelar en forma lógica un algoritmo de exclusión mutua distribuido, junto con una demostración del cumplimiento de las especificaciones a que está sometido.

Como lenguaje lógico de modelación, se utiliza el cálculo de situaciones [1]. El cálculo de situaciones es un lenguaje de la lógica de primer orden extendido con un axioma de inducción de segundo orden que ha sido desarrollado para la representación de conocimiento sobre acción y cambio en mundos dinámicos. El lenguaje ha sido utilizado para modelar transacciones en bases de datos, y la semántica de lenguajes de programación. En la actualidad se investiga su aplicabilidad en la especificación de transacciones en bases de datos activas y temporales, en la especificación de agentes en mundos dinámicos, etc. Este trabajo busca extender el cálculo de situaciones para la modelación de agentes en un sistema distribuido. Este trabajo ha permitido identificar limitaciones expresivas del lenguaje y, como se discute más adelante, motiva la extensión del mismo para eliminar dichas limitaciones, que no son inherentes al cálculo de situaciones. Así, un algoritmo en un sistema distribuido es analizado como una secuencia de acciones, cada una de ellas realiza cambios en el estado de los objetos que participan en los algoritmos. Este trabajo se basa en extensiones al cálculo de situaciones propuestas en [2, 3], y en la hipótesis de que el cálculo de situaciones puede ser la base de un sistema de especificación y análisis de la dinámica de los sistemas distribuidos.

La estructura de este artículo es la que sigue: en la sección 2 se encuentra una descripción del algoritmo de exclusión mutua distribuido escogido. Luego, en la sección 3 se presenta el marco teórico sobre el cual se basa este trabajo. En la sección 4 se presenta el Modelo del Algoritmo y la demostración de su corrección. Por último en la sección 5 se presentan las conclusiones.

2 Descripción del Algoritmo

El algoritmo que se modela en este trabajo corresponde al propuesto por Ricart y Agrawala en el año 1981 [4]. Este algoritmo es distribuido, simétrico y trabaja en una red cuyos nodos no comparten memoria y se comunican sólo a través de mensajes.

Cuando un nodo quiere entrar a la sección crítica, por ejemplo el nodo A, envía un mensaje con una solicitud de ingreso, de ahora en adelante llamado request, a cada uno de los otros nodos de la red. El contenido de este mensaje incluye:

- número del nodo que lo envía
- número de secuencia

Cuando un nodo recibe el request enviado por A, por ejemplo el nodo B, debe enviar a A un mensaje con una autorización para el ingreso a la sección crítica, de ahora en adelante llamado reply. Dicho mensaje puede ser enviado de inmediato si es que B no ha iniciado por su cuenta el protocolo de ingreso a la sección crítica, o si A tiene prioridad sobre él, es decir, el número de secuencia de A es menor que el de B. Los empates se resuelven a través del número de nodo.

Si no ocurre lo anterior, B tiene prioridad sobre A, y por lo tanto debe entrar primero a la sección crítica. En este caso, B posterga el envío del reply a A hasta que éste salga de su sección crítica.

A continuación se presenta el algoritmo en seudo código para cada nodo.

VARIABLES GLOBALES

```
Constantes:
yo: Numero unico de secuencia  asignado a este nodo.
N: Numero de nodos de la red.

Enteros:
mi_num_sec: Numero de secuencia escogido por un request
        iniciado en este nodo.
mayor_num_sec: Mayor numero de secuencia visto en un
        mensaje de request enviado o recibido.
reply_faltante: Numero de replys que aun se esperan.

Boolean:
requesting_CS: Verdadero cuando este nodo esta pidiendo
        acceso a la region critica.
reply_postergado[1..N]=False : reply_postergado[j] es verdadero
        cuando este nodo ha postergado un reply al request de j.

Semaforos binarios:
vars_compartidas: acceso a variables compartidas de este nodo.
```

PROCESO QUE INVOCA LA EXCLUSION MUTUA PARA ESTE NODO

```
/* Pide permiso para entrar a la region critica: */
        P(vars_compartidas);
        /* Escoge numero de secuencia */
            requesting_CS=TRUE;
            mi_num_sec=mayor_num_sec+1;
        V(Vars_Compartidas);
        reply_faltante= N-1;
        for j=1 to N do if (j != yo) then
           send_Message(Request(mi_num_sec,yo),j);
           /* envia un request conteniendo el numero de secuencia
              y el numero de nodo a todos los otros nodos */

/* Espera por el reply de todos los otros  nodos */
         waitfor(reply_faltante=0);

/* La seccion critica puede ser ejecutada en este punto */

/* Sale de la seccion critica */
        requesting_CS=FALSE;
        for j=1 to n do
            if reply_postergado[j] then {
                 reply_postergado[j]=FALSE;
                 send_Message(Reply,j) }
                 /* envia reply al nodo j */
```

PROCESO QUE RECIBE EL MENSAJE DE REQUEST(k,j)

```
/* k es el numero de secuencia del request y j el numero
   de nodo del que envio el request */
boolean: postergar =true /* no se envia reply inmediatamente */

        mayor_num_sec=maximo(mayor_num_sec,k);
        P(vars_compartidas)
              postergar=requesting_CS and ((k>mi_num_sec) or
                        (k=mi_num_sec  and j>yo));
        V(vars_compartidas)
        /* postergar sera verdadero si este nodo tiene
           prioridad sobre el request del nodo j */
        if postergar then reply_postergado[j]=TRUE else
            send_Message(Reply,j)
```

PROCESO QUE RECIBE LOS MENSAJES DE REPLY

```
reply_faltante=reply_faltante-1;
```

3 Cálculo de Situaciones

El cálculo de situaciones [1] es un lenguaje de la lógica de primer orden extendido con inducción. Fue diseñado para permitir la representación de mundos que cambian como resultado de ejecutar acciones. Este lenguaje posee un axioma de segundo orden para caracterizar el espacio de situaciones y define los tipos $\mathcal{A}$ y $\mathcal{S}$ para agrupar las instancias posibles de acciones y situaciones, respectivamente.

En general se dice que el mundo se encuentra en una situación s, la cual sólo puede cambiar luego que un agente (persona, computador, proceso) realice alguna acción (mover un objeto, ejecutar una instrucción, etc). La situación resultante de realizar una acción a en la situación s se denota como $do(a;s)$, $a \in \mathcal{A}$, $s \in \mathcal{S}$. Se distingue una situación inicial, que se denota con el símbolo constante S_0, a partir de la cual se comienza a ejecutar acciones.

En este lenguaje las acciones se representan como funciones, las cuales generalmente reciben parámetros, por ejemplo la acción $actualizar(x; var; val)$ podría representar la acción en que el proceso x actualiza la variable var con el valor val.

Para caracterizar cada situación se utiliza relaciones, a las cuales se les llama fluentes. El valor de verdad de los fluentes puede cambiar de una situación a otra, como resultado de ejecutar una acción. Los fluentes se denotan con símbolos de predicados, los cuales deben tener un término de situación como uno de sus argumentos. Por ejemplo, si un proceso envía un mensaje en la situación s, entonces en la situación resultante de ejecutar esta acción existirá un mensaje en tránsito.

Generalmente, antes que una acción pueda ser ejecutada, ésta debe satisfacer ciertas condiciones. Estas son condiciones necesarias y suficientes, y se les llama precondiciones. Por ejemplo, para que un proceso reciba un mensaje sólo se necesita que exista un mensaje dirigido hacia él. Las precondiciones se denotan con un predicado que las caracteriza.

Para representar la evolución del mundo después de ejecutar una acción se utiliza los llamados axiomas de estado sucesor. Dichos axiomas describen el cambio de cada uno de los fluentes para todas la acciones posibles.

Comunmente se considera a las situaciones como fotografías del mundo en un momento determinado. En este trabajo se utiliza la noción de situación como historia, introducida por Pinto en [2]. Esta noción considera que cada situación posee una historia única que la genera, es decir, existe una única secuencia de acciones que permite llegar desde S_0 a ella.

3.1 Símbolos de funciones y predicados

$do : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{S}$: Si a denota una acción y s una situación, entonces $do(a;s)$ denota la situación resultante de ejecutar a en s.

$ocurred_s \subseteq \mathcal{A} \times \mathcal{S} \times \mathcal{S}$: Si a es una acción, y s_a una situación anterior o igual a s, $Ocurred_s(a;s_a;s)$ será verdadero si la acción a se ejecutó en la situación s_a a lo largo de la historia s.

$ocurredOnce \subseteq \mathcal{A} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S}$: Si una acción a ha ocurrido sólo una vez entre en la situación s_i y s_f, y la situación en la que ocurrió fue s_a, entonces $ocurredOnce(a;s_a;s_i;s_f)$ es verdadero.

$\prec \subseteq \mathcal{S} \times \mathcal{S}$: Si s_1 y s_2 son situaciones, entonces $s_1 \prec s_2$ es verdadero si alguna secuencia de acciones origina s_2 a partir de s_1. Se utiliza la abreviación: $s_1 \preceq s_2 \equiv s_1 \prec s_2 \vee s_1 = s_2$

$Poss \subseteq \mathcal{A} \times \mathcal{S}$: Si a es una acción y s una situación, entonces $Poss(a;s)$ es verdadero si es posible ejecutar la acción a en la situación s, es decir si se satisfacen las precondiciones en la situación s.

$< \subseteq \mathcal{S} \times \mathcal{S}$: Si s_1 y s_2 denotan situaciones, entonces $s_1 < s_2$ es verdadero si alguna secuencia posible de acciones lleva de s_1 a s_2. Se utiliza la abreviación: $s_1 \leq s_2 \equiv s_1 < s_2 \vee s_1 = s_2$

3.2 Ocurrencia de acciones

Como se dijo anteriormente las situaciones pueden ser vistas como historias. Al existir una única secuencia de acciones que permite ir desde la situación inicial S_0 hasta una situación s, esta última almacena en forma indirecta toda la información acerca de qué acciones se ejecutaron en las situaciones intermedias.

En muchos casos es deseable referirse en forma explícita a las acciones que se ejecutaron para producir una determinada historia. Así como también es común querer referirse a la evolución del valor de verdad de los fluentes a lo largo de la historia.

Para representar lo anterior se define una serie de predicados que versan sobre la ocurrencia de acciones en una historia. Además se presenta un lema que muestra la evolución del valor de verdad de los fluentes a lo largo de una historia.

La ejecución de un algoritmo distribuido posee una parte determinística y una no determinística. La parte determinística está dada por el orden único en que cada proceso ejecuta sus propias instrucciones. La parte no determinística proviene de la concurrencia en que cada proceso o procesador ejecuta su parte correspondiente del algoritmo.

En un determinado momento de la ejecución del algoritmo sólo es posible saber que instrucciones ha ejecutado cada proceso y el orden de éstas. Para capturar este concepto se ha definido una serie de predicados, los cuales se presentan a continuación.

Definición 1 Se dice que una acción a ocurrió en una situación s_a a lo largo de la historia s, si existe otra situación s' en la historia de s que sea el resultado de ejecutar dicha acción. De esta forma $occurred_s$ se define de la siguiente manera.

$$occurred_s(a; s_a; s) \equiv (\exists s')s' \leq s \wedge s' = do(a; s_a) \tag{1}$$

Definición 2 Se dice que una acción a ha ocurrido una única vez en la situación s_a, a partir de s_i, y a lo largo de una historia sf, si en dicho tramo existe una única situación que sea el resultado de ejecutar esta acción en s_a. Así se define el predicado occurredOnce de la siguiente forma.

$$occurredOnce(a; s_i; s_a; sf) \equiv s_i \leq s_a \wedge occurred_s(a; s_a; sf) \wedge [occurred_s(a; s'_a; sf) \supset s'_a < s_i] \tag{2}$$

Suponiendo que una acción a ocurre en una situación s_a anterior a s, y que además esta acción a vuelve verdadero el valor de verdad del fluente R, y que después de s_a no ha ocurrido ninguna acción que haga falso a R. Entonces, necesariamente R debe ser verdadero en s.

Lema 1 Sea s_i una situación anterior a sf. Si el valor de verdad de un fluente R en s_i es distinto al que tiene en sf, entonces este valor sólo puede haber sido cambiado por la ocurrencia de una acción a, en una situación posterior o igual a s_i y anterior a sf.

$$R(s_i) \wedge \neg R(sf) \wedge s_i < sf \supset (\exists s_a; a) occurred_s(a; s_a; sf) \wedge s_i < s_a \wedge R(s_a) \wedge \neg R(do(a; s_a)) \tag{3}$$

Para la demostración de este lema es necesario hacer uso del axioma de inducción, que no se incluye por falta de espacio. En síntesis, se demeustra que debe existir una situación s_a, entre s_i y sf tal que R cambia entre s_a y el sucesor inmediato de s_a en la historia de sf.

4 Modelo

4.1 Dominios, funciones y constantes

Además de los dominios $\mathcal{A}$ y $\mathcal{S}$ y la constante S_0 definidos en la sección anterior es necesario incluir un dominio que permita identificar a cada uno de los nodos del sistema distribuido que ejecuta el algoritmo de exclusión mutua.

$\mathcal{N}$: Conjunto de números naturales.

Nodos: Conjunto de nodos del sistema distribuido que ejecuta el algoritmo de exclusión mutua.

id : Nodos $\rightarrow \mathcal{N}$: Si x es un nodo del sistema, entonces id(x) devuelve un número natural único asignado a ese nodo.

4.2 Fluentes

Cada proceso necesita recordar una serie de variables asociadas a diversos eventos (instrucciones del programa) que afectan el estado de avance del algoritmo de exclusión mutua. Por ejemplo, cuando se actualiza una variable es necesario recordar el nuevo valor de éstas. Otros eventos que es necesario recordar son el envío y la recepción de un mensaje.

El modelo propuesto representa esta información por medio de fluentes. A continuación se presentan los fluentes involucrados en el desarrollo del algoritmo.

RequestingCS(x; s): Es verdadero si el nodo x quiere entrar a la sección crítica en la situación s, es decir, si ha iniciado el proceso de exclusión mutua en una situación anterior a s.

MayorNSec(x; n; s): Es verdadero si el mayor número de secuencia visto por el nodo x es n, en cualquier situación anterior a s.

NSec(x; n; s): Es verdadero si el número de secuencia escogido por el nodo x, para la petición de acceso a la región crítica es n, en la situación s. Más adelante, por simplicidad se hace uso de una función definida en términos de este fluente. La función tiene el mismo nombre del fluente y se define a a continuación:

$NSec(x; s) = n \equiv NSec(x; n; s)$

ReqEnTransito(origen; destino; cont; s): Es verdadero si el nodo origen ha enviado un request al nodo destino, con contenido cont, pero éste aún no sido recibido por el nodo destino en la situación s.

ReplyEnTransito(origen; destino; s): Es verdadero si el nodo origen ha enviado un reply al nodo destino, pero éste aún no ha sido recibido por el nodo destino en la situación s.

ReqPendiente(x; y; c; s): Es verdadero si en una situación anterior a s ha llegado un request de x a y con contenido c, y en la situación s el nodo x aún no ha enviado el reply.

4.3 Acciones

Como se dijo anteriormente, el avance del algoritmo de excludión mutua se produce por la ocurrencia de diversos eventos que transcurren a lo largo de la ejecución de este. Dichos eventos corresponden en realidad a un conjunto de instrucciones ejecutadas por los nodos del sistema. Por ejemplo, el evento iniciar protocolo de exclusión mutua representa el siguiente conjunto de instrucciones presentes al inicio del proceso que inicia la petición de acceso a la región crítica.

```
P(vars_compartidas);
/* Escoge numero de secuencia */
    requesting_CS=TRUE;
    mi_num_sec=mayor_num_sec+1;
V(Vars_Compartidas);
```

Estos conjuntos de instrucciones o eventos se modelan como acciones ejecutadas por cada uno de los nodos del sistema. A continuación se indica las acciones presentes en el transcurso del algoritmo. Es necesario destacar que las acciones aquí descritas aún no guardan ninguna relación con el orden en que éstas son ejecutadas por el algoritmo. El orden de ejecución de estas acciones será introducido más adelante como una restricción en las historias en las cuales un proceso puede estar en la región crítica.

iniciarProtocolo(x): El nodo x inicia el protocolo de exclusión mutua.

escogerNSec(x): El nodo x escoge un número de secuencia para iniciar el protocolo de exclusión mutua.

enviarReq(origen; destino; cont): el nodo origen, envía un request al nodo destino con contenido cont.

recibirReq(origen; destino; cont): el nodo destino recibe un request con contenido cont desde el nodo origen.

enviarReply(origen; destino): el nodo origen, envía un reply al nodo destino.

recibirReply(origen; destino): el nodo destino recibe un reply desde el nodo origen.

actualizarNSec(x): El nodo x actualiza el mayor número de secuencia que ha visto.

entrarSC(x): El nodo x comienza a ejecutar en la sección crítica.

salirSC(x): El nodo x termina de ejecutar en la sección crítica.

4.4 Precondiciones para las acciones

A continuación se muestra los axiomas de precondición para las acciones descritas arriba.

Las siguientes acciones no tienen precondiciones, ya que una implementación determinada podría ejecutarlas en cualquier momento. Como se dijo anteriormente, el orden de ejecución apropiado para que se provea exclusión mutua es introducido posteriormente como una restricción en las historias.

$Poss(entrarSC(x); s) \equiv Verdadero$

$Poss(escogerNSec(x); s) \equiv Verdadero$

$Poss(enviarReq(x; y; c); s) \equiv Verdadero$

$Poss(actualizarNSec(x); s) \equiv Verdadero$

$Poss(salirSC(x); s) \equiv Verdadero$

Ahora se muestran las instrucciones de control de flujo. Estas son modeladas como precondiciones para las acciones.

$Poss(iniciarProtocolo(x); s) \equiv \neg requestingCS(x; s)$

Para que el nodo x pueda iniciar el protocolo de exclusión mutua es necesario que no lo haya iniciado antes:

$$Poss(enviarReply(x; y); s) \equiv ReqPendiente(y; x; (nsecy; idy); s) \land$$
$$(\neg RequestingCS(x; s) \lor (nsecy \leq NSec(x; s) \land (nsecy \neq NSec(x; s) \lor idy \leq id(x)))$$

Un nodo x puede enviar el reply a y si es que había un request pendiente de y aún no atendido por x, y este último no quiere entrar a la sección crítica o y tiene prioridad frente a x.

Por último para ejecutar las acciones relacionadas con la recepción de mensajes es necesario que se haya enviado los mensajes en cuestión.

$Poss(recibirReq(x; y; c); s) \equiv ReqEnTransito(x; y; c; s)$

$Poss(recibirReply(x; y); s) \equiv ReplyEnTransito(x; y; s)$

4.5 Axiomas de estado sucesor

Los axiomas de estado sucesor permiten modelar el efecto de las acciones (instrucciones) sobre los fluentes descritos antes.

$$Poss(a; s) \supset RequestingCS(x; do(a; s)) \equiv$$
$$RequestingCS(x; s) \land a \neq salirSC(x) \lor a = iniciarProtocolo(x)$$

$$Poss(a; s) \supset NSec(x; n; do(a; s)) \equiv$$
$$NSec(x; n; s) \land a \neq escogerNSec(x) \lor MayorNSec(x; n; s) \land a = escogerNSec(x)$$

$$Poss(a; s) \supset MayorNSec(x; n; do(a; s)) \equiv$$
$$MayorNSec(x; n; s) \land (\nexists n')a \neq actualizarNSec(x; n') \lor a = actualizarNSec(x; n)$$

$$Poss(a; s) \supset ReqEnTransito(x; y; c; do(a; s)) \equiv$$
$$ReqEnTransito(x; y; c; s) \land (a \neq recibirReq(x; y; c) \lor a = enviarReq(x; y; c)$$

$$Poss(a; s) \supset ReplyEnTransito(x; y; do(a; s)) \equiv$$
$$ReplyEnTransito(x; y; s) \land (a \neq recibirReply(x; y; c) \lor a = enviarReply(x; y)$$

$$Poss(a; s) \supset ReqPendiente(x; y; c; do(a; s)) \equiv$$
$$ReqPendiente(x; y; c; s) \land (\forall c')a \neq recibirReq(y; x; c') \lor$$
$$a = recibirReq(x; y; c)$$

4.6 Modelación de exclusión mutua

Considerando la noción de situaciones como historias propuesta en [2] es posible decir que una determinada ejecución de instrucciones se representa por una situación s. Esta situación almacena indirectamente el orden en que las instrucciones fueron ejecutadas, pues existe una única secuencia de acciones que lleva desde la situación inicial S_0 a s.

De esta forma se define un predicado que impone una restricción sobre la historia s. Dicha restricción representa todas las historias posibles en las que un proceso puede estar en su sección crítica.

Esta definición refleja, de acuerdo al algoritmo, las acciones que se debieron ejecutar, y el orden en que éstas debieron ocurrir, para que en la situación s el nodo x se encuentre en la sección crítica.

Definición 3

$$
\begin{array}{l}
EnSeccionCritica(x;s) \equiv \\
(\exists\, s_1; s_2; s_8) s_1 < s_2 < s_8 \wedge \\
\quad occurredOnce(iniciarProtocolo(x); s_1; s_1; s) \wedge \\
\quad occurredOnce(escogerNSec(x); s_1; s_2; s) \wedge \\
\quad (\forall y)(x \neq y \wedge \\
\qquad (\exists\, s_3; s_4; s_5; s_6; s_7)(s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 \wedge \\
\qquad\quad occurredOnce(enviarReq(x;y;c); s_1; s_3; s) \wedge \\
\qquad\quad occurredOnce(recibirReq(y;x;c); s_1; s_4; s) \wedge \\
\qquad\qquad\qquad c = (NSec(x; s_3); nodo(x)) \wedge \\
\qquad\quad occurredOnce(actualizarNSec(y; NSec(x; s_3); s_1; s_5; s) \wedge \\
\qquad\quad occurredOnce(enviarreply(y;x); s_1; s_6; s) \wedge \\
\qquad\quad occurredOnce(recibirReply(x;y); s_1; s_7; s) \\
\qquad) \\
\quad) \\
\quad occurredOnce(entrarSC(x); s_1; s_8; s) \wedge \\
\quad (\nexists s_9)(s_1 < s_9 \wedge \\
\qquad occurred_s(salirSC(x); s_9; s))
\end{array}
$$

4.7 Correctitud del Modelo

El algoritmo es correcto si efectivamente provee exclusión mutua. El objetivo de la axiomatización anterior es establecer el siguiente teorema:

Teorema 4.1 A partir de la axiomatización anterior, que representa el protocolo para exclusión mutua, se infiere que si dos nodos se encuentran en la sección crítica en la misma situación s, entonces deben ser el mismo. Es decir:

$$\Sigma \models EnSeccionCritica(a;s) \wedge EnSeccionCritica(b;s) \supset id(a) = id(b)$$

Donde Σ representa a los axiomas base del cálculo de situaciones, junto con las definiciones y los axiomas descritos en los capítulos 3 y 4.

Demostración: La demostración formal se encuentra en el reporte técnico ???. Informalmente la demostración supone que dos nodos, a y b, se encuentran en la sección crítica en la situación s. Para que esto sea posible ambos nodos deben cumplir la Definición 3, propuesta en el capítulo anterior.

La definición 3 establece las acciones que deben haber ocurrido con anterioridad a s para que esto fuera posible.

Primero ambos nodos deben deben haber ejecutado la acción iniciarProtocolo, y por lo tanto, el fluente RequestingCS es verdadero para ambos nodos.

También deben haber ejecutado la acción escogerNSec, sea nseca el número escogido por a y nsecb el escogido por b.

Luego ambos nodos deben haber enviado mensajes de request al resto de los nodos, en particular a envió uno a b, y viceversa.

En particular el nodo a debe haber autorizado el ingreso del nodo b a la sección crítica y viceversa, es decir, ambos deben haber ejecutado la acción enviarReply. Si esto ocurrió, entonces las precondiciones de estas acciones fueron verdaderas en el momento en que se ejecutaron.

Además ambos nodo han actualizado sus números de secuencia con anterioridad a s.

Por último, se demuestra que para que estas precondiciones puedan haber sido verdaderas, entonces necesariamente el nodo a es igual al nodo b.

5 Trabajo Futuro y Conclusiones

El modelo presentado en este trabajo corresponde a una experimento de modelación de un algoritmo distribuido de exclusión mutua utilizando el lenguaje del cálculo de situaciones. Este trabajo muestra que esta modelación sí es posible. Así como también es posible demostrar en forma mecánica propiedades de los algoritmos. En este trabajo se demostró la la propiedad de exclusión mutua, sin embargo, también es posible modelar otras propiedades como deadlock free o entrada eventual.

Lo anterior demuestra que el cálculo de situaciones posee entonces la suficiente expresividad para representar los algoritmos de exclusión mutua. Sin embargo, fue posible constatar que en muchos aspectos la modelación era poco natural. Esto se puede apreciar claramente en la fórmula 3, la cual modela un determinado orden de ejecución de ciertas instrucciones dentro de un ambiente concurrente. Esta fórmula es demasiado compleja, haciendo que el lector guíe su atención hacia la complejidad, en desmedro de la comprensión del concepto.

De esta forma, se hace necesario extender el lenguaje del cálculo de situaciones para permitir expresar conceptos como el anterior en forma más natural. Esta extensión constituye el trabajo inmediato a desarrollar por los autores.

Una vez que se cuente con la extensión del lenguaje descrita arriba surge el desafío de automatizar la demostración de propiedades de los algoritmos. Así sería posible mecanizar las demostraciones de estas propiedades. Para que esto sea posible es necesario establecer una estrategia de demostración de dichas propiedades.

Por último es interesante explorar la formalización de otros algoritmos, con el fin de identificar otras extensiones al lenguaje del cálculo de situaciones que lo hagan cada vez más expresivo.

Referencias

[1] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, Machine Intelligence 4, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[2] Javier Pinto. Occurrences and Narratives as Constraints in the Branching Structure of the Situation Calculus. Submitted to the Journal of Logic and Computation
URL = ftp://lyrcc.ing.puc.cl/pub/jpinto/jlc.ps.gz.

[3] Raymond Reiter. The Frame Problem in the Situation Calculus: A Soundness and Completeness Result, with an Application to Database Updates. In Proceedings First International Conference on AI Planning Systems, College Park, Maryland, June 1992.

[4] Glenn Ricart and Ashok Agrawala. An optinal algorithm for mutual exclusion in computer networks. Comunications of the ACM, 1981.

[5] Andrew Tanenbaum. Distributed Operating Systems. Prentice Hall, 1995.

A Demostración del teorema 4.1

De la definición 3 se tiene:

$$
\begin{array}{l}
\mathrm{EnSeccionCritica}(a;s) \equiv \\
(\exists\, s_1;s_2;s_8) s_1 < s_2 < s_8 \wedge \\
\qquad \mathrm{occurredOnce}(\mathrm{iniciarProtocolo}(a);s_1;s_1;s) \wedge \\
\qquad \mathrm{occurredOnce}(\mathrm{escogerNSec}(a);s_1;s_2;s) \wedge \\
\qquad (\forall y)(a \neq y \wedge \\
\qquad\qquad (\exists\, s_3;s_4;s_5;s_6;s_7)(s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 \wedge \\
\qquad\qquad\quad \mathrm{occurredOnce}(\mathrm{enviarReq}(a;y;c);s_1;s_3;s) \wedge \\
\qquad\qquad\quad \mathrm{occurredOnce}(\mathrm{recibirReq}(y;a;c);s_1;s_4;s) \wedge \\
\qquad\qquad\qquad\qquad c = (\mathrm{NSec}(a;s_3);\mathrm{nodo}(a)) \wedge \\
\qquad\qquad\quad \mathrm{occurredOnce}(\mathrm{actualizarNSec}(y;\mathrm{NSec}(a;s_3);s_1;s_5;s) \wedge \\
\qquad\qquad\quad \mathrm{occurredOnce}(\mathrm{enviarreply}(y;a);s_1;s_6;s) \wedge \\
\qquad\qquad\quad \mathrm{occurredOnce}(\mathrm{recibirReply}(a;y);s_1;s_7;s) \\
\qquad\qquad) \\
\qquad) \\
\qquad \mathrm{occurredOnce}(\mathrm{entrarSC}(a);s_1;s_8;s) \wedge \\
\qquad (\nexists s_9)(s_1 < s_9 \wedge \\
\qquad\qquad \mathrm{occurred}_s(\mathrm{salirSC}(a);s_9;s))
\end{array}
\tag{4}
$$

Se extiende el lenguaje con las nuevas constantes S_1, S_2 y S_8 las cuales reemplazan a las variables s_1, s_2 y s_8 cuantificadas existencialmente en la formula 4. De esta forma se obtiene:

$$
\begin{array}{l}
S_1 < S_2 < S_8 \wedge \\
\mathrm{occurredOnce}(\mathrm{iniciarProtocolo}(a);S_1;S_1;s) \wedge \\
\mathrm{occurredOnce}(\mathrm{escogerNSec}(a);S_1;S_2;s) \wedge \\
(\forall y)(a \neq y \wedge \\
\qquad (\exists\, s_3;s_4;s_5;s_6;s_7)(s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 \wedge \\
\qquad\quad \mathrm{occurredOnce}(\mathrm{enviarReq}(a;y;c);S_1;s_3;s) \wedge \\
\qquad\quad \mathrm{occurredOnce}(\mathrm{recibirReq}(y;a;c);S_1;s_4;s) \wedge \\
\qquad\qquad\qquad c = (\mathrm{NSec}(a;s_3);\mathrm{nodo}(a)) \wedge \\
\qquad\quad \mathrm{occurredOnce}(\mathrm{actualizarNSec}(y;\mathrm{NSec}(a;s_3);S_1;s_5;s) \wedge \\
\qquad\quad \mathrm{occurredOnce}(\mathrm{enviarreply}(y;a);S_1;s_6;s) \wedge \\
\qquad\quad \mathrm{occurredOnce}(\mathrm{recibirReply}(a;y);S_1;s_7;s) \\
\qquad) \\
) \\
\mathrm{occurredOnce}(\mathrm{entrarSC}(a);S_1;S_8;s) \wedge \\
(\nexists s_9)(S_1 < s_9 \wedge \\
\qquad \mathrm{occurred}_s(\mathrm{salirSC}(a);s_9;s))
\end{array}
\tag{5}
$$

De la fórmula 5 se obtiene:

$$\mathrm{occurredOnce}(\mathrm{iniciarProtocolo}(a);S_1;S_1;s) \wedge (\nexists s9)\ s1 ¡ s9 ¡ s\ \mathrm{occurred}(\mathrm{salirSC}(a);s9;s) \tag{6}$$

De la fórmula 6 y del Lema 1 se obtiene:

$$(\forall s')S_1 \leq s' ¡ s \leq s\mathrm{RequestingCS}(x;s') \tag{7}$$

Lo que representa el hecho de que el valor de verdad del fluente RequestingCS no se ve alterado entre S_1 y s pues las únicas acciones que alteran el valor de verdad del fluente RequestingCS(a;s) son iniciarProtocolo(a), la cual lo hace verdadero, y salirSC(a), la cual lo hace falso.

De la fórmula 5 se obtiene:

$$\mathrm{occurredOnce}(\mathrm{escogerNSec}(a);S_1;S_1;s) \tag{8}$$

De la fórmula 8 y del Lema 1 se obtiene:

$$(\exists nseca)\mathrm{NSec}(a;nseca;s1) \wedge (\forall s')S_1 \leq s' \leq s\mathrm{NSec}(a;nseca;s') \tag{9}$$

El valor de verdad del fluente NSec es verdadero para todas las situaciones intermedias entre S_1 y s, pues esta acción es la única que afecta el valor de verdad de éste.

De la fórmula 5 se obtiene:

$$\begin{aligned}
&(\forall y)(\mathrm{id}(a) \neq \mathrm{id}(y) \wedge \\
&\quad (\exists\, s3;s4;s5;s6;s7)(S_2 ¡ s3 ¡ s4 ¡ s5 ¡ s6 ¡ s7 ¡ S_8 \wedge \\
&\qquad \mathrm{occurredOnce}(\mathrm{enviarReq}(a;y;c);S_1;s3;s) \wedge \\
&\qquad \mathrm{occurredOnce}(\mathrm{recibirReq}(y;a;c);S_1;s4;s) \wedge \\
&\qquad\qquad c = (\mathrm{NSec}(x;s3);\mathrm{nodo}(x)) \wedge \\
&\qquad \mathrm{occurredOnce}(\mathrm{actualizarNSec}(y;\mathrm{NSec}(x;s3));S_1;s5;s) \wedge \\
&\qquad \mathrm{occurredOnce}(\mathrm{enviarreply}(y;x);S_1;s6;s) \wedge \\
&\qquad \mathrm{occurredOnce}(\mathrm{recibirReply}(x;y);S_1;s7;s) \\
&\quad) \\
&)
\end{aligned} \tag{10}$$

La fórmula 10 representa el hecho de que a ha enviado un request y ha recibido un reply de todo los demás nodos. En particular, la fórmula 10 es verdadera reemplazando la variable y, cuantificada universalmente por el nodo b,

$$\begin{aligned}
&(\exists\, s3;s4;s5;s6;s7)(S_2 ¡ s3 ¡ s4 ¡ s5 ¡ s6 ¡ s7 ¡ S_8 \wedge \\
&\quad \mathrm{occurredOnce}(\mathrm{enviarReq}(a;b;c);S_1;s3;s) \wedge \\
&\quad \mathrm{occurredOnce}(\mathrm{recibirReq}(a;b;c);S_1;s4;s) \wedge \\
&\qquad c = (\mathrm{NSec}(a;s3);\mathrm{id}(a)) \wedge \\
&\quad \mathrm{occurredOnce}(\mathrm{actualizarNSec}(b;\mathrm{NSec}(a;s3));S_1;s5;s) \wedge \\
&\quad \mathrm{occurredOnce}(\mathrm{enviarreply}(b;a);S_1;s6;s) \wedge \\
&\quad \mathrm{occurredOnce}(\mathrm{recibirReply}(a;b);S_1;s7;s)))
\end{aligned} \tag{11}$$

Se extiende nuevamente el lenguaje con las constantes $S_3; S_4; S_5; S_6; S_7$ para reemplazar en la formula 11 las situaciones $s_3; s_4; s_5; s_6; s_7$, las cuales están cuantificadas existencialmente. Así se obtiene:

$$
\begin{aligned}
& S_2 < S_3 < S_4 < S_5 < S_6 < S_7 < S_8 \wedge \\
& \mathrm{occurredOnce}(\mathrm{enviarReq}(a; b; c); S_1; S_3; s) \wedge \\
& \mathrm{occurredOnce}(\mathrm{recibirReq}(a; b; c); S_1; S_4; s) \wedge \\
& \qquad c = (\mathrm{NSec}(a; S_3); \mathrm{id}(a)) \wedge \\
& \mathrm{occurredOnce}(\mathrm{actualizarNSec}(b; \mathrm{NSec}(a; S_3)); S_1; S_5; s) \wedge \\
& \mathrm{occurredOnce}(\mathrm{enviarreply}(b; a); S_1; S_6; s) \wedge \\
& \mathrm{occurredOnce}(\mathrm{recibirReply}(a; b); S_1; S_7; s)))
\end{aligned} \tag{12}
$$

De la fórmula 12 se obtiene:

$$
\begin{aligned}
& \mathrm{occurredOnce}(\mathrm{recibirReq}(b; a; c); S_1; S_4; s) \wedge c = (\mathrm{NSec}(a; s_3); \mathrm{id}(a)) \wedge \\
& \mathrm{occurredOnce}(\mathrm{recibirReq}(b; a; c); S_1; S_5; s)
\end{aligned} \tag{13}
$$

De la fórmula 13 y el Lema 1, se obtiene:

$$
(\forall s')S3 \leq s' \leq s \mathrm{ReqPendiente}(a; b; (\mathrm{NSec}(a; S_3); \mathrm{id}(a); s') \tag{14}
$$

De la fórmula 12 se tiene que ha ocurrido la acción enviarReply en S_6, entonces en la situación S_6 debe haber sido posible ejecutarla

$$
\mathrm{occurredOnce}(\mathrm{enviarReply}(b; a); S_6; s) \supset \mathrm{Poss}(\mathrm{enviarreply}(b; a); S_6) \tag{15}
$$

De la fórmula 15 y el axioma de precondición para enviarReply se obtiene:

$$
\begin{aligned}
& \mathrm{ReqPendiente}(b; a; (\mathrm{NSec}(a; S_3); \mathrm{id}(a); S_6) \wedge (\neg \mathrm{RequestingCS}(b; S_6) \vee \\
& (\mathrm{NSec}(a; s_2) \leq \mathrm{NSec}(b; s_5) \wedge (\mathrm{NSec}(a; S_3) \neq \mathrm{NSec}(b; S_6) \vee \mathrm{id}(a) \leq \mathrm{id}(b)))
\end{aligned} \tag{16}
$$

Realizando el mismo procedimiento anterior a partir de EnSeccionCritica(b;s) se obtienen resultados análogos a los de las formulas $4 \ldots 16$ para el nodo b, pero con las nuevas constantes $S'_1; S'_2; S'_3; S'_4; S'_5; S'_6; S'_7$ y S'_8 reemplazando a las constantes $S_1; S_2; S_3; S_4; S_5; S_6; S_7$ y S_8 respectivamente.

Así se obtiene una fórmula análoga a 16 para el nodo b:

$$
\begin{aligned}
& \mathrm{ReqPendiente}(a; b; (\mathrm{NSec}(b; S'_3); \mathrm{id}(b); S'_6) \wedge (\neg \mathrm{RequestingCS}(a; S'_6) \vee \\
& (\mathrm{NSec}(b; S'_2) \leq \mathrm{NSec}(a; S'_5) \wedge (\mathrm{NSec}(b; S'_3) \neq \mathrm{NSec}(a; S'_6) \vee \mathrm{id}(b) \leq \mathrm{id}(a)))
\end{aligned} \tag{17}
$$

Finalmente de las fórmulas 16 y 17, junto con las fórmulas 7 , 9, sus análogas para b y de la definición de la función NSec se obtiene:

$$
\mathrm{id}(a) = \mathrm{id}(b) \tag{18}
$$

La ecuación 18 representa lo que se quería demostrar.