Comunicación entre agentes inteligentes mediante cálculo de eventos usando Erlang

José Yedid Aguilar López, Felipe Trujillo-Romero, Manuel Hernández Gutiérrez

Universidad Tecnológica de la Mixteca,
Oaxaca, México

jyedid.al@gmail.com, ftrujillo@mixteco.utm.mx, hg.manuel@gmail.com

Resumen. En este trabajo se presenta una nueva representación para la comunicación entre robots durante la realización de una tarea en específico. Estos robots también son llamados agentes inteligentes. Dichos agentes se deben de comunicar para llevar a cabo la tarea asignada de manera eficiente. Esta comunicación se planea de inicio en forma directa mediante el cálculo de eventos y se traduce al lenguaje Erlang para que los mensajes puedan ser enviados y recibidos por los agentes. Así mismo esta arquitectura define de manera implícita las acciones a ejecutar por los mismos. Para ello se define la de un enfoque denominado Modelación de Procesos Erlang. Este enfoque es validado en una tarea de búsqueda realizada por tres robots, los cuales deberán encontrar varios patrones en la región de búsqueda y transmitir los mensajes a sus compañeros. Los mensajes serán gestionados por un servidor central que fungirá como pizarra en la cual los agentes podrán escribir y/o leer dichos mensajes para determinar las acciones a realizar.

Palabras clave: Cálculo de eventos, Erlang, robótica distribuida, agentes inteligentes, modelado de procesos.

1. Introducción

Los agentes son procesos de software o robots que se mueven alrededor de un sistema para realizar tareas específicas ya que están especialmente programados para ello. El nombre de "agente" se deriva del hecho de que trabajan en nombre de un objetivo más amplio. Los agentes recopilan y procesan información, así como también pueden tener la capacidad de intercambiar esa información con otros agentes. A menudo, los agentes cooperan tal como una colonia de hormigas, pero también pueden tener competencia amistosa como en una economía de libre mercado. Entre los desafíos que se pueden presentar en los sistemas de agentes distribuidos son los mecanismos de coordinación entre los agentes, el control de la movilidad de los agentes, así como su diseño de software y las interfaces. Cabe señalar que la investigación y trabajo en agentes es interdisciplinario abarcando: inteligencia artificial, cómputo móvil, cómputo distribuido e ingeniería de software.

Dentro de un sistema distribuido nos interesa que los agentes desarrollen una tarea en particular pero que cooperen entre ellos para llevarla a cabo de una manera más rápida y eficiente. Ahora bien para entender todo esto es necesario partir de la definición de un sistema distribuido. Para ello usemos la definición dada por Lamport [7]: "Un sistema distribuido consiste de una colección de distintos procesos los cuales están espacialmente separados y se comunican unos con otros a través del intercambio de mensajes."

Como se puede observar en la definición anterior el paso de mensajes es fundamental para que los agentes puedan tener una buena comunicación entre ellos y por lo tanto llevar a cabo la tarea encomendada. Por lo tanto lo que se desea proponer es una alternativa de comunicación de los agentes. Este nuevo enfoque hará uso del paradigma paso de mensajes definiendo de inicio los posibles eventos mediante una notación de Modelación de Procesos Erlang, basada en el formalismo lógico cálculo de eventos y que tiene correspondencia unidireccional con la programación concurrente de Erlang. Esto significa que la implementación de los agentes serán tratados como procesos de este lenguaje. El primer paso sería hacer el modelado de la comunicación entre agentes a través de dicha notación. El segundo sería conseguir la programación de los agentes pasando el modelado a código Erlang.

Si esto se traslada o se lleva al campo experimental empleando robots que puedan cooperar para reconocer patrones en un entorno cerrado o para recuperar el mapa de dicho entorno puede ser útil. Lo anterior debido a que esta implementación en Erlang sería un módulo que poseería dicho robot para compartir la información de su recorrido lo que le permitiría indicar su estatus actual, auxiliar a otros a terminar su actividad o solicitar apoyo.

En este mecanismo de coordinación se asume que existirá un servidor central que se encargará de gestionar los mensajes el cual será montado sobre una plataforma fija. Mientras que los agentes o robots serán los clientes de dicho servidor.

Ahora bien, para comenzar a modelar el paso de mensajes se empleará la notación de Modelación de Procesos. Esta notación es una especificación de sistemas concurrentes. Con esta notación se contemplarán los procesos y tiempos de envíos y recepción de mensajes entre ellos. Dejando la estructura del mensaje y su contenido a resolver en la fase de implementación, aunque también es posible especificarla.

2. Antecedentes

En esta sección vamos a revisar algunos conceptos necesarios para el planteamiento y resolución del problema propuesto.

2.1. Calculo de eventos

El cálculo de eventos es un formalismo lógico basado en la narrativa para el razonamiento acerca de las acciones y sus efectos que puede ser útil para representar la ocurrencia de eventos del mundo real. Fue propuesto originalmente desde la perspectiva de programación lógica por Kowalski y Sergot [1] con la premisa de que

la lógica formal se puede utilizar para representar muchos tipos de conocimiento para muchos propósitos. Por ejemplo puede ser utilizado para la especificación formal de programas, bases de datos y el lenguaje natural en general. Esto debido a que para muchas de estas aplicaciones de la lógica es necesaria una representación del tiempo y de un tipo de lógica temporal no clásica [2].

El cálculo de eventos tiene acciones y propiedades que varían en el tiempo llamados eventos y fluentes respectivamente. Además, como se comentó, está basado en la narrativa la cual es una especificación de un conjunto de ocurrencias de eventos reales. Ello nos permite abordar ciertos fenómenos de forma más natural en el cálculo de eventos incluyendo: (1) eventos concurrentes, (2) tiempo continuo, (3) el cambio continuo, (4) eventos con duración, (5) los efectos no deterministas, (6) eventos parcialmente ordenados y (7) eventos desencadenados.

Desde la versión original del cálculo de eventos este ha evolucionado considerablemente y varios autores han propuesto sus versiones [3] [4] [5]. Sin embargo se puede denotar dos clases de cálculo de eventos principales: (1) el clásico y (2) el simplificado.

El cálculo de eventos clásico propuesto por Kowalski y Sergot en 1986 [1] trabaja con eventos que ocurren en un tiempo determinado, fluentes y períodos de tiempo. A partir de estos se definen los predicados que utiliza este cálculo de eventos.

Desarrollado por Kowalski [6] el cálculo de eventos simplificado a diferencia del cálculo de eventos clásico agrega el concepto de puntos de tiempo específicos en que un evento ocurre y quita el de período de tiempo en el que un evento pudo haber ocurrido.

Ambos enfoques del cálculo de eventos tienen sus predicados los cuales les permiten describir la ocurrencia de los eventos. Sin embargo y dado que el que se va a utilizar en este trabajo es el cálculo simplificado se presentarán los predicados de este. Por lo tanto, sea e = evento, f = fluente y t, t_1, t_2 = puntos de tiempo, se tienen los siguientes predicados:

Initially (f)	El fluente *f* es verdadero en el punto de tiempo 0.
HoldsAt (f, t)	El fluente *f* es verdadero en el tiempo *t*.
Happens (e, t)	El evento *e* ocurre en el tiempo *t*.
Initiates (e, f, t)	Si el evento *e* ocurre en el tiempo *t*, entonces el fluente *f* es verdadero después del tiempo *t*.
Terminates (e, f, t)	Si el evento *e* ocurre en el tiempo *t*, entonces el fluente *f* es falso después del tiempo *t*.
StoppedI n (t_1, f, t_2)	El fluente *f* es detenido entre los tiempos t_1 y t_2

2.2. Sistemas distribuidos

El término de sistemas distribuidos ha sido definido por varios autores [7, 8, 9, 10, 11]. Sin embargo, la definición en esencia no ha cambiado a lo largo de los años. En dichas definiciones, todos coinciden en que se trata de un conjunto de componentes que pudieran ser procesos o equipos de cómputo independientes en su funcionamiento

interno y que se comunican a través del envío de mensajes para tratar parte de la solución de un problema.

En los últimos años los sistemas distribuidos han tomado gran importancia debido a su presencia en la vida diaria, esto debido principalmente a la necesidad de [12, 10]: (a) Tener un entorno de cómputo distribuido geográficamente, (b) Acelerar el cálculo de cómputo, (d) Compartir recursos: hardware y software y (e) Tener la capacidad de recuperarse ante fallos del sistema.

Las características principales de los sistemas distribuidos son [11, 10]: (1) No poseer un reloj físico común, (2) No compartir la misma memoria, (3) Separación geográfica, (4) Concurrencia y (5) Autonomía y heterogeneidad.

Los sistemas distribuidos presentan a lo largo de su desarrollo retos de alta complejidad en las etapas de especificación, diseño, implementación y verificación, lo que ha dado lugar a que se elaboren propuestas de herramientas y formalismos en el caso de la especificación de sistemas distribuidos y con ello ayudar a realizar más eficientemente el resto del ciclo de desarrollo. Una de las ventajas de la especificación es que a partir de ella es posible desarrollar técnicas y herramientas que hagan que la etapa de implementación se realice de forma automática al menos parcialmente.

De esta manera el presente trabajo, coadyuvando al desarrollo de sistemas distribuidos con paso de mensajes, se centra en proponer una especificación para estos sistemas a través del cálculo de eventos, además de presentar una correspondencia entre está especificación y la sintaxis del lenguaje de programación Erlang, el cual aborda la complejidad del desarrollo a gran escala de sistemas concurrentes y distribuidos. Esta correspondencia que se menciona ayudará a convertir una notación del cálculo de eventos a código Erlang con las limitantes que más tarde se indicarán.

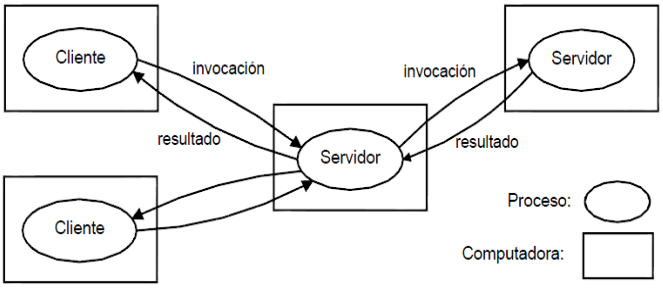

Fig. 1. Los procesos con roles Cliente o Servidor.

2.3. Arquitectura cliente/servidor

Si bien es cierto que existen varias arquitecturas empleadas en los sistemas distribuidos una de las más utilizadas es la arquitectura Cliente/Servidor [13, 14, 11] y es la que se va a discutir.

En la arquitectura Cliente/Servidor los procesos toman el rol de cliente o servidor. En particular, los procesos clientes interactúan con procesos servidores con el fin de

acceder a recursos que estos administran los cuales pueden estar hospedados en distintos equipos de cómputo, ver Fig. 1.

Los servidores pueden ser a su vez clientes de otros servidores, como lo muestra el gráfico. Por ejemplo, un servidor web es a menudo un cliente de un servidor de archivos local que gestiona los archivos en los que las páginas web se almacenan. Los servidores web así como la mayoría de los otros servicios de Internet son clientes del servicio DNS, el cual traduce los nombres de dominio de Internet en direcciones de red. Otro ejemplo relacionado con la web se refiere a los motores de búsqueda, los cuales permiten a los usuarios ver los resúmenes de la información disponible en las páginas web en los sitios de todo Internet.

La interacción entre cliente y servidor se da cuando el cliente hace una solicitud al servidor y entra en un tiempo de espera mientras este último trabaja para proporcionar la respuesta.

2.4. Programación en Erlang

El lenguaje de programación Erlang fue concebido para hacer frente a problemas relacionados con el desarrollo de sistemas distribuidos de tiempo real altamente concurrentes, buscando una solución para [15]: a) poder desarrollar este tipo de software de forma rápida y eficiente, b) disponer de sistemas tolerantes a fallos de software y hardware y c) poder actualizar el software sobre la marcha, sin detener la ejecución del sistema.

Las características que posee el lenguaje de programación Erlang y por las cuáles se creó son [16] [15]: (1) Construcciones de alto nivel, (2) Crear sistemas distribuidos, (3) Tolerante a fallos, (4) Escalabilidad, (5) Código en caliente, (6) Orientado a la Concurrencia y (7) el Paso de mensajes en lugar de memoria compartida.

Como Erlang fue diseñado pensando para tener múltiples tareas en ejecución simultáneamente resulta ser un soporte natural para la concurrencia. Este sistema concurrente utiliza el concepto de proceso para obtener una separación clara entre las tareas, lo que permite crear arquitecturas tolerantes a fallos y utilizar plenamente el hardware multinúcleo disponible hoy en día [16]. Por lo que el término proceso en el lenguaje de programación Erlang es un concepto fundamental debido a que es la unidad de concurrencia.

Un proceso representa una actividad en marcha. Esto es, la ejecución de una pieza de código de un programa y que es concurrente a otro proceso ejecutando también su propio código. La única manera en que los procesos interactúan es a través del paso de mensajes en donde el dato enviado va de un proceso a otro [15].

Los procesos en Erlang están separados unos de otros y asegurados para no perturbarse entre sí. Esto significa que los procesos son como un tipo de burbuja que proporciona aislamiento con otros procesos [16].

Cada proceso en Erlang tiene su propia memoria de trabajo y un buzón para recibir mensajes, mientras que los hilos en muchos otros lenguajes de programación y sistemas operativos son actividades concurrentes que comparten el mismo espacio de memoria.

Cuando un proceso necesita intercambiar información con otro entonces le envía un mensaje. Ese mensaje es una copia de sólo lectura de los datos que el emisor posee. La comunicación de procesos en Erlang siempre trabaja como si el receptor recibiera una copia del mensaje, incluso si el emisor se encuentra en la misma computadora. Esta distribución transparente de procesos permite a los programadores Erlang ver a la red como un conjunto de recursos. Una de las cosas que también Erlang hace, es que su máquina virtual balancea automáticamente la carga de trabajo sobre los procesadores disponibles. Por lo que los programas Erlang automáticamente se adaptan a distinto hardware.

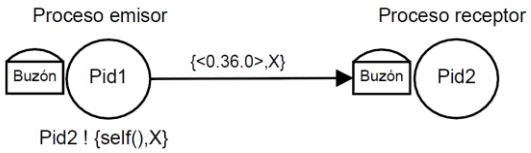

Fig. 2. Paso de mensajes entre procesos Erlang.

En Erlang las primitivas de paso de mensajes son asíncronas, a menudo el emisor no necesita saber si el mensaje llegó. Este método de comunicación asíncrono implica que el emisor no tiene que ser suspendido mientras se está entregando el mensaje, en particular si se envía el mensaje a través de un enlace de comunicación lento. El diagrama de la Fig. 2 muestra un ejemplo del envío de un mensaje de un proceso a otro.

3. Notación ProME

A la notación para la Modelación de Procesos Erlang se le denominó: ProME. La cual como ya se mencionó está basada en el formalismo lógico del cálculo de eventos. La notación ProME se centra en las acciones principales que se pueden hacer con los procesos de Erlang: crear procesos, enviar y recibir mensajes desde procesos.

En este trabajo se argumenta que la comunicación entre agentes puede realizarse vía paso de mensajes y modelar esto con la notación ProME, para más adelante conseguir las directrices de la implementación en Erlang dada la correspondencia de la notación con este último.

Antes, se mostrará un ejemplo de la modelación con esta notación describiendo la arquitectura *Cliente/Servidor* así como de su correspondiente representación en Erlang. Los puntos a considerar para realizar esto son:

1. Hay un único servidor y se llama *Server*,
2. Existe un cliente que se llama *Client* pero pueden haber varias instancias de este, y
3. La comunicación es síncrona entre el servidor y el cliente.

Sin embargo, también es necesario presentar los nombres de los eventos que utilizará la notación ProME del cálculo de eventos. Recordar que en la sección anterior se dijo que algunas de las acciones principales de la programación

concurrente en Erlang eran las de enviar y recibir mensajes. Por lo que estos eventos son:

1. *actuate* —involucrará la creación de una función Erlang.
2. *receive* —involucrará un patrón de la cláusula receive de Erlang.
   (a) *end* —es un caso especial del evento *receive*, para establecer cuándo un proceso debe terminar. Involucrará la definición de un patrón en la cláusula *receive* que haga que termine el proceso Erlang en cuestión.
3. *send* —involucrará el envío de un mensaje a un proceso con el operador de envío *!*.

Además se requiere que cada uno de estos eventos tenga una asignación lo cual significa que los eventos tendrán que estar relacionados con algunos valores. Por ejemplo, si se desea enviar un mensaje *Msg* a un proceso *Process1* entonces el evento *send* tendrá que conocer estos valores. La asignación quedaría como:

send(Process1,Msg)

Como se mencionó en secciones anteriores en una arquitectura *Cliente/Servidor* se puede ver a cada elemento como un proceso. Así que ahora vamos a describir el servidor como un proceso desde ambas perspectivas. Primero desde el punto de vista del cálculo de eventos y después realizando su correspondencia en lenguaje Erlang. Las funcionalidades serán las de lanzar el servidor, recibir peticiones, responder al cliente y como terminar la ejecución del servidor en caso de que se requiera.

3.1. Iniciar servidor

Para el evento *actuate*, aparte de relacionarse con algún valor se le tendrá que indicar qué es lo que tiene que accionar, valga la redundancia. En el caso del ejemplo *Server*, el nombre del servidor será el sufijo del nombre del evento *actuate* y quedará como:

actuateServer(Args)

Donde *Args* serían los argumentos que necesite el servidor expresados en los tipos de datos que emplea Erlang o en lenguaje natural. Con esta expresión lo que se tiene es que *actuateServer* es el evento que puede iniciar al proceso *Server* el cual necesita los argumentos *Args*. Empero hasta el momento todavía el servidor no inicia, sólo se está indicando qué iniciará y cómo se va a iniciar. Para poder iniciar el servidor se expresará con el predicado *Initiates* del cálculo de eventos, por lo que se tendría:

Initiates(actuateServer, liveServer, 0)

La sentencia anterior indica que se dará vida al servidor generando un fluente llamado *liveServer* con el evento *actuateServer* y es entonces cuando ya se pone en marcha.

Estas dos expresiones juntas representarían en Erlang las siguientes instrucciones:

```
1 -module(server).
2 -export([actuateServer/0, funtionServer/1]).
3
4 actuateServer()-> register(server,spawn(server,functionServer,[Args])).
5
6 functionServer(Args)-> functionServer(Args).
```

Este mapeo a código se lleva a cabo de la siguiente forma:

1. La línea 1 declara el uso de un módulo con nombre *server*. Este nombre se obtuvo del sufijo del evento *actuate*, que se formó como *actuateServer*.
2. La línea 2 declara las funciones *actuateServer* y *functionServer* que serán implementadas dentro del módulo. El nombre de las funciones se forma empleando también el sufijo *Server*. Los argumentos de la función *actuateServer* que lanza inicialmente al servidor, por default es cero, sin argumentos. En cambio los de la función *functionServer* será la cantidad de argumentos que se asignen en la expresión *actuateServer* del cálculo de eventos, en este caso se considerará *Args* como único argumento.
3. La línea 4 declara una función en la que se registra un proceso. El proceso que se registra es *serve*r, sufijo del evento *actuate*, y este proceso ejecutará la función *functionServer* del módulo *server*.
4. La línea 6 declara la función *functionServer* tomando como argumentos lo que se le haya asignado al evento *actuateServer*. El predicado *Initiates* hace que la función se invoque a sí misma, esto no es un problema para Erlang ya que soporta que los procesos puedan estar ejecutándose siempre.

3.2. Recepción en el servidor

Con el módulo *server* ya creado se definirán en este todas las funciones que necesite el servidor.com en el caso de: *actuateServer* y *functionSever*. Esta última será ejecutada por el proceso *server* y se le anexará las funcionalidades básicas del servidor como lo son las solicitudes que puede atender y las posibles respuestas para cada una de estas. Para el caso de las solicitudes que llegan, significa que se definirá una cláusula *receive* dentro de la función y se especificarán los patrones que tendrá. Con el evento *receive* de la notación ProME:

receive(From, Pattern1)

Para que este evento suceda se utiliza el predicado *Happens*:

Happens(receive, 1)

El segundo parámetro, valor 1, indica que debe realizarse el evento receive después del evento que sucede en el tiempo 0. La representación en código Erlang de esto consiste de un patrón tipo tupla con dos elementos. El primero de ellos es una variable no ligada que se encuentra a la espera de un Identificador de Proceso (PId) de algún proceso y el segundo es el patrón mismo que puede ser de cualquier tipo de dato Erlang. Dado que es la primera aparición del evento *receive* entonces se crea la constructiva *receive* de Erlang:

```
1 functionServer(Args)->
2     receive
3       {From,Pattern1}-> functionServer(Args)
4     end.
```

Para agregar otro patrón, la notación en el cálculo de eventos quedaría de la siguiente forma:

receive(From, Pattern2)
Happens(receive, 1)

Con las expresiones anteriores se genera una cláusula más en *receive* y se coloca como el segundo patrón lo cual hace que el código en Erlang se aumente una línea más.

3.3 Responder desde el servidor

Ahora bien, cada petición que llegue al servidor puede ser respondida por este. Para que el primer evento *receive* tenga su correspondiente evento *send*, después de las expresiones de recepción debería ir enseguida las siguientes:

send({ok,Response})
Happens(send, 2)

Lo cual haría que se modificara el cuerpo de la primera cláusula receive para agregar una operación de envío hacia el proceso del que se recibió la petición. La instrucción Erlang correspondiente a las expresiones anteriores se muestra en la línea 3 del siguiente código:

```
1 functionServer(Args)->
2     receive
3       {From,Pattern1}-> From ! {ok, Response},
4                         functionServer(Args);
5       {From,Pattern2}-> functionServer(Args)
6     end.
```

3.4. Terminar el servidor

Es recomendable que el servidor termine de buena manera haciendo antes una serie de acciones que se requieran sin que el proceso sea forzado a parar. Para eso se utiliza el evento *end* junto con el predicado *Terminates*. Esto queda expresado en el cálculo de eventos como:

end(Finish)
Terminates(end, liveServer, 1)

donde *Finish* será el patrón que tenga la sentencia *receive*, entonces cuando ocurre el evento *end* con el predicado *Terminates* hace que el fluente *liveServer* ahora sea falso. En código Erlang se agregara un patrón más en la función *functionServer*:

```
functionServer(Args)->
    receive
      {From,Pattern1}-> From ! {ok, Response},
                        functionServer(Args);
      {From,Pattern2}-> From ! {error, Reason},
                        functionServer(Args);
      {Finish}-> %to do some before to end the function
                 ''Process is finished.''
    end.
```

Estas instrucciones que se agregan a la función deberían de ser modificadas después para personalizar el fin del proceso. Incluso es posible continuar con eventos *send* para avisar a otros procesos de que el servidor terminará o enviar alguna información antes de esto.

Por razones de espacio se va a omitir el modelado del *Cliente*. Sin embargo se modela de forma análoga al *Servidor*.

4. Aplicación: búsqueda de patrones

Una vez mostrada la notación ProME pasaremos a validar dicha notación mediante la aplicación de la misma a una tarea de búsqueda. Esta tarea será realizada usando tres robots lo cual significa que se distribuirá el espacio de búsqueda para que esta se realice de una forma rápida y eficiente (Fig. 3).

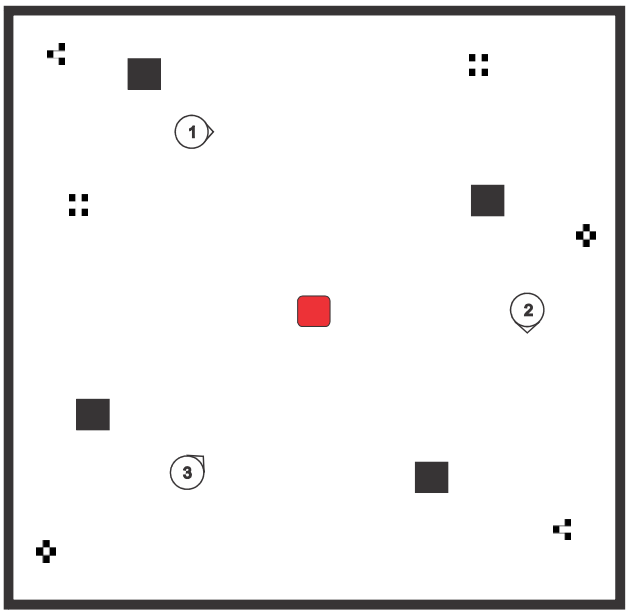

Fig. 3. Escenario de trabajo: los agentes son los círculos orientados, el cuadrado redondeado es el servidor y los cuadros negros son obstáculos. Además están los patrones a buscar.

4.1. Descripción del problema

En el escenario de trabajo los robots deberán buscar una serie de marcas en el piso e informar cada vez que han encontrado una de ellas en el instante que dos agentes encuentren la misma marca, en dos lugares diferentes, deberán de comunicarse para eliminarla. Es decir si la vuelven a encontrar no emitirán mensaje alguno. Además un mismo agente no podrá eliminar la marca aunque la encuentre en diferentes sitios del escenario. Cada marca es un patrón binario formado por una matriz de *3x3* (ver Fig. 4), tiene el tamaño necesario para que los sensores del robot puedan detectarlo al estar encima de la marca.

Dado que es una arquitectura Cliente/Servidor los agentes serán los clientes mientras que una computadora central fungirá como servidor. Este servidor será utilizado como pizarrón en donde se guardaran los diferentes mensajes emitidos por los agentes a los cuales pueden acceder los demás robots si poseen el patrón correcto.

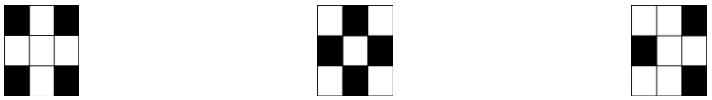

Fig. 4. Patrones utilizados.

4.2. Modelación con la notación ProME

Para el servidor *BBoard* las actividades que se modelarán son: permitir la sesión de un agente, enviar mensaje a otro usuario, enviar lista de agentes activos, terminar la sesión, enviar lista de patrones encontrados y determinar el fin de la actividad. La narrativa es la siguiente:

actuateBBoard()
Initiates(actuateBBoard, liveBBoard, 0)
receive(FromRobot, {login, RobotCredential})
Happens(receive, 1)
send(login accept)
Happens(send, 2)
send(login not accept)
Happens(send, 2)
receive(FromRobot,patterns)
Happens(receive, 1)
send({patterns, PatternsList})
Happens(send, 2)
receive(FromRobot, {invitation to, RobotID,Msg})
Happens(receive, 1)
send(RobotID, {invitation from, FromRobot,Msg})
Happens(send, 2)
receive(FromRobot,who)

Happens(receive, 1)
send({who, RobotList})
Happens(send, 2)
receive(FromRobot, logout)
Happens(receive, 1)

Los procesos del cliente se pueden dividir en dos partes: (1) Inicio de sesión y (2) actividades a realizar. Enseguida se muestra la primera narrativa para el cliente Robot donde se declara el inicio de sesión:

actuateRobotClient(RobotCredential)
Happens(actuateRobotClient, liveRobotClient, 1)
send(BBoard, {self(), login, RobotCredential})
Happens(send, 2)
receive(login_accept)
Happens(receive, 3)
end(login_not_accept)
Terminates(end, 3)

En caso de que el agente haya iniciado sesión exitosamente entonces recibirá como respuesta del servidor el átomo *login_accept* y ahora el agente ya puede realizar sus actividades, esto se modela en la siguiente narrativa:

actuateRobotClient()
Happens(actuateRobotClient, liveRobotClient, 4)
receive(Pattern_request)
Happens(receive, 5)
send(BBoard, {self(), patterns})
Happens(send, 6)
receive({invitation_to, RobotID,Msg})
Happens(receive, 5)
send(BBoard, {self(), invitation_to, RobotID,Msg})
Happens(send, 6)
receive({who, RobotList})
Happens(receive, 5)
end(logout_ request)
Terminates(end, 5)
send(BBoard, {self(), logout})
Happens(send, 6)

4.3 Implementation en Erlang

Se presenta aquí el código correspondiente a las tres narrativas de arriba descritas: dos del cliente *Robot* y una del servidor *BBoard*. Las dos narrativas del *Robot* tienen el mismo nombre de proceso, significa que se trata de la implementación de una

misma función pero que difiere en el tipo y en el número de argumentos. Comencemos por mostrar el código en Erlang de la narrativa correspondiente al inicialización del cliente, el cual queda como:

```
functionRobotClient(RobotCredential)->
  bboard ! {self(),login,RobotCredential},
  receive
    login_accept-> functionRobotClient();
    login_not_accept->
  end.
```

El código Erlang correspondiente al modelado de la actividad del cliente es:

```
functionRobotClient()->
  receive
    Pattern_request-> bboard ! {self(),patterns};
    {invitation_to,RobotID,Msg}-> bboard, {self(),invitation_to,RobotID,Msg};
    {who,RobotList}-> ;
    logout_request-> bboard ! {self(),logout};
  end.
```

Finalmente el código correspondiente al servidor *BBoard* es el siguiente:

```
-module(bboard).
-export([actuateBBoard/0, functionBBoard/0]).

actuateBBoard()-> register(bboard,spawn(bboard,functionBBoard,[])).

functionBBoard()->
  receive
    {FromRobot,login,RobotCredential}-> FromRobot ! login_accept,
                                        Fromrobot ! login_not_accept,
                                        functionBBoard();
    {FromRobot,patterns}-> FromRobot ! {patterns,PatternsList},
                           functionBBoard();
    {FromRobot,invitation_to,RobotID,Msg}-> RobotID ! {invitation_from,FromRobot,Msg},
                                            functionBBoard();
    {FromRobot,who}-> FromRobot ! {who,RobotList},
                      functionBBoard();
    {FromRobot,logout}->
  end.
```

Cabe mencionar que este ejemplo solo es una prueba de concepto y sirve para validar el modelado propuesto en una aplicación de robótica distribuida. En esta solo se modela el paso de mensajes que existirá entre los agentes al momento de llevar a cabo una tarea de esta índole. Por lo cual lo que se presenta es la evolución que se tiene en la comunicación realizada entre el servidor y los agentes durante la tarea asignada.

En esta tarea faltaría la codificación correspondiente a las acciones que realizaran cada uno de los agentes, las cuales solo se especifican. Sin embargo el código

generado se puede compilar en Erlang y es un ejemplo de que la metodología propuesta se puede aplicar a este tipo de sistemas distribuidos.

5. Conclusiones y perspectivas

En este trabajo se presentó un nuevo enfoque para realizar la comunicación de agentes inteligentes mediante el paso de mensajes. Este enfoque está basado en el cálculo de eventos el cual una vez definido se traduce al lenguaje Erlang para ser utilizado. Esto se realiza a partir de la notación ProME. El código generado en Erlang es solamente la comunicación entre el servidor y los clientes dejando la programación de las acciones al programador.

Para validar este enfoque se propuso una tarea de búsqueda de patrones usando tres agentes que se comunicaban para indicar cual patrón habían encontrado con la finalidad de descartarlos cuando dos robots encontraran el mismo patrón en dos sitios diferentes. De esta actividad se presentó la evolución de los eventos en el cálculo de eventos así como el código generado en Erlang.

Si bien no se tienen resultados prácticos

Entre los trabajos que se tienen considerados para continuar este proyecto se pueden mencionar principalmente: (1) Ampliación de la gramática de la notación ProME para que se puedan modelar diferentes actividades de propósito general, (2) Crear un software de traducción que permite la conversión a partir del cálculo de eventos hacia el lenguaje Erlang.(3) Validar el enfoque mediante la resolución de otro tipo de problemas y (4) Implementar la totalidad del problema de reconocimiento de patrones en simulación y en robots reales.

Referencias

1. R. Kowalski and M. Sergot: A logic-based calculus of events. New Gen. Comput., 4(1):67–95 (1986)
2. J. Mccarthy and P. J. Hayes: Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence, pp. 463–502, Edinburgh University Press (1969)
3. R. Miller and M. Shanahan: Some alternative formulations of the event calculus. Computer Science; Computational Logic; Logic programming and Beyond, pp. 452–490, Springer Verlag (2002)
4. E. Mueller: Event calculus. In: F. van Harmelen, V, Lifschitz, B, Porter, editors, Handbook of Knowledge Representation, pp. 671–708. Elsevier (2008)
5. V. Alexiev: The event calculus as a linear logic program (1995)
6. R. Kowalski: Database Updates in the Event Calculus. Journal of Logic Programming, pp. 121–146 (1992)
7. L. Lamport: Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565 (1978)

8. M. Singhal and N. G. Shivaratri: Advanced Concepts in Operating Systems. McGraw-Hill, Inc. (1994)
9. A.S. Tanenbaum and M. van Steen: Distributed Systems: Principles and Paradigms. Pearson Prentice Hall, 2 edition (2007)
10. A.D. Kshemkalyani and M. Singhal: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2008)
11. G.F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair: Distributed Systems: Concepts and Design. Addison-Wesley, 5 edition (2011)
12. S. Ghosh: Distributed Systems: An Algorithmic Approach. Chapman & Hall/CRC Computer & Information Science Series. Taylor & Francis (2007)
13. T. Özsu and P. Valduriez: Principles of Distributed Database Systems. Springer (2011)
14. I. Sommerville: Software Engineering. Pearson Education, 9 edition (2011)
15. F. Cesarini and S. Thompson: Erlang Programming. O'Reilly Media (2009)
16. M. Logan, E. Merritt, and R. Carlsson: Erlang and OTP in Action. Manning (2011)