Una Extensión de la Programación en Lógica que incluye Eventos y Comunicación

Natalia L. Weinbach
nlw@cs.uns.edu.ar

Alejandro J. García
ajg@cs.uns.edu.ar

Laboratorio de Investigación y Desarrollo en Inteligencia Artificial (LIDIA)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253, (8000) Bahía Blanca, Argentina
Tel: (0291) 459-5135 / Fax: (0291) 459-5136

Palabras clave: Sistemas Multi-agente. Inteligencia Artificial Distribuida. Programación en Lógica. Programación Manejada por Eventos.

1. Introducción

Este trabajo presenta una línea de investigación para el desarrollo de Sistemas Multi-agente en un entorno distribuido. La investigación está centrada en el estudio, diseño e implementación de una extensión de la programación en lógica que incluye manejo de eventos y primitivas de comunicación entre procesos.

El manejo de eventos es necesario para poder desarrollar agentes que reaccionen ante ciertos sucesos, producidos por cambios en su entorno o provocados por otros agentes. Por ejemplo, un agente robot que choca contra un obstáculo debería poder reaccionar directamente ante este evento y ejecutar algún algoritmo de evasión. Por otro lado, si varios agentes deben interactuar para resolver un problema, entonces el lenguaje debe proveer un mecanismo para la comunicación y coordinación entre los agentes. Uniendo las facilidades de comunicación y el manejo de eventos, se permitirá programar la respuesta a eventos externos. Esto es, eventos que son despachados remotamente por otros agentes.

2. Manejo de eventos

En una aplicación de software, un evento designa la ocurrencia de algo significativo para un proceso o programa, como por ejemplo la finalización de una operación de I/O. Se ha observado que en ciertas aplicaciones, resulta útil contar con procesos que se ejecutan automáticamente cuando suceden ciertos eventos [1, 2]. Entre estas aplicaciones se encuentran los sistemas reactivos (e. g., un controlador de termostato) y agentes de software en general que trabajan de manera cooperativa o competitiva (e. g., fútbol de robots [5, 6]). Es importante destacar que, si un agente tuviera que responder a más de un evento debería estar preparado para manejarlos a todos, y sería inaceptable que se ignoren algunos de ellos. Además, los tiempos de arribo de los eventos generalmente no son predecibles y pueden existir múltiples fuentes.

Con la motivación de agregar predicados para el manejo de eventos a programas escritos en lenguajes de programación en lógica, se han investigado las capacidades de distintas implementaciones de Prolog y los mecanismos que han sido desarrollados para el manejo de eventos en

Financiado parcialmente por SeCyT Universidad Nacional del Sur (subsidio: 24/ZN09) y por la Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 Nro 13096)

otros ambientes. Ningún Prolog que se haya estudiado provee la funcionalidad necesaria para que un usuario programador pueda definir sus propios eventos al intentar resolver un problema.

Por lo tanto, en este proyecto se propone desarrollar una extensión de una implementación de Prolog para poder manejar eventos. Para su implementación, se encontró un interesante mecanismo de comunicación entre objetos denominado "*signals* & *slots*", en una librería que está siendo muy utilizada en distintos ambientes de programación, conocida como Qt [8, 10]. Qt es un conjunto de herramientas multiplataforma escrito en C++ (completamente orientado a objetos) y está disponible para una gran cantidad de plataformas, por lo que los programas que utilizan esta librería pueden ser portables. Para utilizar Qt, se debe escoger una implementación de Prolog que tenga una interface entre lenguajes con C, implementada en ambos sentidos, i. e., para que funciones de C puedan ser llamadas desde Prolog y para que C pueda invocar predicados de Prolog. Otras características que se buscaron en el Prolog elegido fueron que su distribución sea gratuita y que contara con alguna forma de comunicación entre procesos. El prolog elegido fue SWI-Prolog [7, 13, 14] ya cumple con todas estas características.

En Qt, el manejo de eventos está compuesto por dos elementos fundamentales: *señales* y *slots* [9]. Una señal es emitida cuando un evento particular ocurre. Un slot, en cambio, es el handler del evento, i. e., una función que se invoca en respuesta a una señal. Se pueden conectar tantas señales como se quiera a un único slot; y una señal puede ser conectada a tantos slots como se desee. En nuestra propuesta, el comportamiento de un programa quedará definido entonces por los eventos que son testeados y por las funciones que se ejecutan en respuesta a esos eventos. En la parte de código que corresponde a C++ existirán objetos que podrán comunicarse utilizando señales y slots. A su vez, el programa contendrá otra parte expresada en lenguaje Prolog, donde habrá predicados que se ejecutarán ante un evento. Un esquema del mecanismo propuesto se ilustra en la Figura 1. El código en C es el encargado de emitir una señal, la cual ejecuta un predicado de Prolog para proceder con la ejecución de la aplicación. Por convención, se invoca a un predicado Prolog que posee el mismo nombre que la señal.

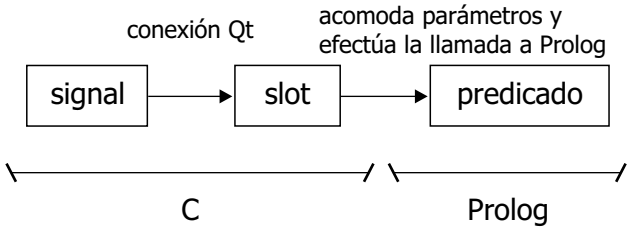

Figura 1: Diagrama de la interacción

En nuestra propuesta, también puede ocurrir que el código en Prolog al intentar resolver una meta dada llame a código en C para completar alguna acción (e. g., emitir una señal cuando sea apropiado). Por ejemplo, supongamos que tenemos una clase Termostato en C++, con dos señales. Una señal se utiliza cuando ocurre un cambio en la temperatura actual del ambiente, y la otra señal se emite cuando cambió la temperatura deseada (a la que queremos llegar). Se debería contar con una definición de clase que, por lo menos, provea la funcionalidad que se muestra en la Figura 2. Las líneas punteadas en esta figura ilustran las conexiones entre señales y slots, correspondientes a la librería Qt. La interacción entre Prolog y C está representada a través de las líneas sólidas.

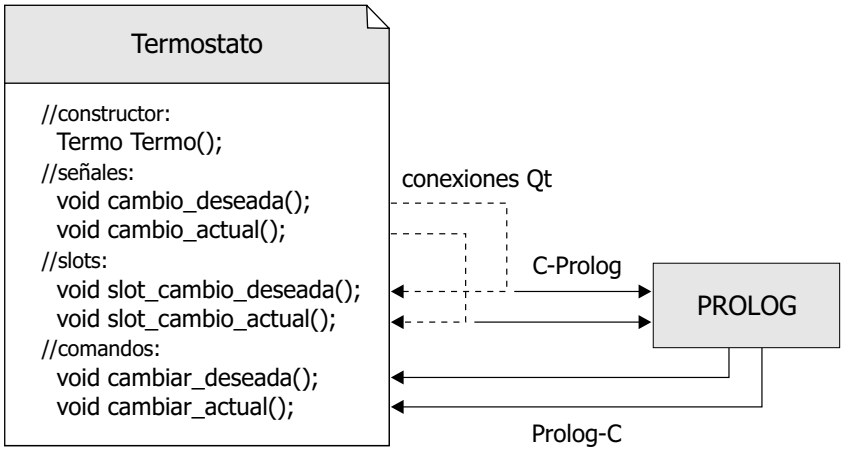

Figura 2: Clase Termostato.

3. Comunicación entre agentes

En un sistema multi-agente es importante que los agentes puedan comunicarse y coordinar sus tareas. Para esto, el lenguaje de implementación del sistema debe proveer primitivas de comunicación y coordinación.

Los sockets proveen un mecanismo de comunicación bidireccional, punto a punto, entre dos procesos. Existen primitivas de comunicación a través de sockets en algunas versiones de Prolog, sin embargo, ninguna de ellas permite una lectura asincrónica de los datos que llegan al mismo. En el módulo *Network* de Qt se encuentran definidas dos clases que facilitan y dan portabilidad a la programación en redes. Lo más interesante es que provee funciones para realizar una lectura asincrónica de los datos recibidos: emite una señal cuando llegan datos al socket.

Para desarrollar un mecanismo completo de comunicación, además de la interface de sockets, hay que acordar un lenguaje común de más alto nivel, con el que se expresarán los mensajes, y un protocolo que defina cómo se relacionan los mensajes de una misma conversación. En este trabajo, el lenguaje que se empleará en el intercambio de mensajes es FIPA ACL [3].

FIPA ACL es un lenguaje de comunicación entre agentes inspirado en los actos del habla, donde un mensaje se representa como una lista de paréntesis balanceados. El acto comunicativo presente en el mensaje es el que define la intención principal del mismo. Para ordenar el intercambio de mensajes han sido desarrollados distintos protocolos de interacción. Los que se destacan para la mayor parte de las tareas son: FIPA Request (un proceso solicita a otro la ejecución de cierta acción) y FIPA Query (para realizar alguna consulta).

La Figura 3 muestra un esquema de los distintos niveles considerados en este trabajo para la comunicación y el manejo de eventos. FIPA ACL es el lenguaje de especificación de los mensajes que puede utilizarse tanto en programas escritos en Prolog como en C++. Prolog utiliza llamadas a C++ para invocar funciones de más bajo nivel o utilizar los mecanismos definidos en Qt. La librería Qt se ve como una extensión de C++. En el nivel más bajo se encuentra el sistema operativo; la comunicación con éste la realiza C, mediante system calls.

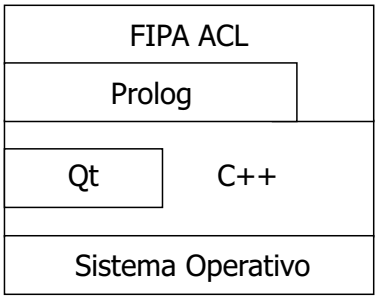

Figura 3: Niveles de abstracción.

4. Eventos remotos

Contar con capacidad para generar eventos desde otro procesador, abre las puertas a una gran cantidad de aplicaciones en Sistemas Multi-agente. De esta manera, el resultado de la ejecución de un agente no sólo dependerá de su código, sino de la coordinación y comunicación con otros agentes, que pueden brindarle información, generar eventos, y requerirle la ejecución de ciertas funciones, entre otras cosas. Uniendo las facilidades desarrolladas para la comunicación y para el manejo de eventos, se permitirá programar la respuesta a eventos externos en Prolog.

Cuando un agente genera un evento de interés para otro proceso, se lo hace conocer por medio de un mensaje. En la implementación de eventos remotos, los procesos tienen abierto un puerto P y esperan asincrónicamente por el arribo de un mensaje. Para agilizar el manejo de las primitivas de comunicación de Qt y de los sockets, se ha definido un módulo de IPC que puede incluirse en cualquier programa y que contiene las clases y funciones básicas que se deben utilizar [11]. Este módulo contiene la definición del socket para el cliente y el servidor, y emite señales que deben ser manejadas al querer leer los datos que se van recibiendo.

El manejo y análisis de los mensajes recibidos también puede hacerse en Prolog, donde la tarea de parsing es más sencilla. El código en C actúa como una capa inferior en el modelo de comunicación y se encarga del transporte de los mensajes. Se desarrollaron dos predicados básicos para enviar y recibir mensajes: `msg_enviar/4` y `msg_recibir/4` (ver detalles de implementación en [11]). En ambos predicados se especifican como argumentos el acto comunicativo, la lista de parámetros, el host y port del destinatario. Lo interesante es que en el código Prolog se puede poner especial atención a los mensajes más relevantes para la tarea que se está realizando, unificando el término del acto comunicativo.

5. Conclusiones y trabajo futuro

En el presente trabajo se ha presentado un mecanismo para que programas escritos en Prolog puedan reaccionar ante la ocurrencia de eventos. Estos eventos no sólo se generan en el mismo programa, sino que pueden ser enviados desde otro procesador, por otro proceso, como un mensaje. A su vez, se ha definido un mecanismo completo de comunicación y se presentaron nuevos predicados de alto nivel para enviar y recibir mensajes; esto implica que no hay necesidad de manejar sockets directamente, ni poseer conocimientos de protocolos de red.

Estas dos nuevas posibilidades que se tienen ahora en un programa escrito en Prolog, eventos y comunicación, abren paso a nuevas aplicaciones y facilitan la tarea del programador al implementar agentes que interactúan.

En trabajos futuros podrían implementarse predicados de alto nivel para el manejo de los protocolos de FIPA y agregar algunos más avanzados para realizar negociaciones, como pueden ser los de red de contrato (*Contract-Net*) y subastas (*Auction*). Podrían desarrollarse casos de estudio que involucren una gran cantidad de agentes que se ejecutan en red, y que se comunican para alcanzar un objetivo común. Incluso, se podría trabajar con agentes que realicen planificaciones al recibir ciertos requerimientos de ejecución de tareas. En este último caso, se podría analizar la disponibilidad de los recursos y decidir, en forma inteligente, si se puede cometer la tarea, o si es preferible delegarla a algún otro agente que esté dispuesto a realizarla.

Referencias

[1] J. Edward Carryer. *Event Driven Programming*, Feb. 2002. http://design.stanford.edu/spdl/ee118/pdf_files/EventDrivenProg.pdf.

[2] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven Programming for Robust Software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Sept. 2002.

[3] Foundation for Intelligent Physical Agents. FIPA Interaction Protocol Library Specification, 2002. http://www.fipa.org.

[4] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley and Sons, 3rd edition, 1998.

[5] Fernando A. Mandolesi. *Plataforma para el desarrollo de agentes Robocup*. Tesis de Licenciatura en Ciencias de la Computación. Universidad Nacional del Sur, Diciembre 2002.

[6] RoboCup. http://www.robocup.org.

[7] SWI-Prolog. http://www.swi−prolog.org.

[8] Trolltech. http://www.trolltech.com.

[9] Trolltech. *Qt 3.1 Whitepaper*, 2003. ftp://ftp.trolltech.com/qt/pdf/3.1/qt-white-paper-a4-web.pdf.

[10] Trolltech. *Qt Reference Documentation*, 2003. http://doc.trolltech.com/3.3/ index.html.

[11] Natalia L. Weinbach. *Una Extensión de la Programación en Lógica que incluye Eventos y Comunicación. Estudio, Diseño e Implementación*. Proyecto Final de Carrera. Universidad Nacional del Sur, Marzo 2004.

[12] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[13] Jan Wielemaker. *A C++ interface to SWI-Prolog*. Amsterdam, 1990-2002.

[14] Jan Wielemaker. *SWI-Prolog 5.2 Reference Manual*. Amsterdam, Nov. 2003.