Programación en Paralelo Utilizando un Modelo de Sistemas Multi-agente

Natalia L. Weinbach[1] nlw@cs.uns.edu.ar
Mariano Tucat[2] mt@cs.uns.edu.ar
Alejandro J. García[3] ajg@cs.uns.edu.ar

Laboratorio de Investigación en Inteligencia Artificial (LIDIA)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253 - (8000) Bahía Blanca, Argentina

Resumen

En este trabajo se describe como utilizar el modelo de SMA para el diseño de programas con paralelismo. Uno de los objetivos es identificar las características que debería poseer un lenguaje de programación con paralelismo que utilice un modelo de Sistemas Multi-agente. Además, se evalúa uno de los pocos lenguajes que existen con estas características.

Los SMA se han concentrado en proveer un modelo abstracto cuya intención es lograr que el programador se involucre más con el modelado del problema y la lógica del programa que con los detalles de la ejecución paralela en sí misma. Los desarrollos de paralelismo se han orientado más hacia la eficiencia, siendo la mayoría programas de cálculo numérico. En este trabajo mostramos cómo las áreas de SMA y de paralelismo pueden complementarse y ofrecer soluciones a problemas de cada una. El enfoque será hacia campos de procesamiento no numérico, donde las interacciones son menos predecibles o generalizables.

Palabras clave: Programación en paralelo. Sistemas Multi-agente. Lenguajes de Programación.

1. Introducción

En este trabajo mostramos cómo las áreas de SMA y de paralelismo pueden complementarse y ofrecer soluciones a problemas de cada una. En particular se describe cómo utilizar el modelo de SMA para el diseño de programas con paralelismo. Uno de los objetivos es identificar las características que debería poseer un lenguaje de programación con paralelismo que utilice un modelo de Sistemas Multi-agente. Además, se evalúa uno de los pocos lenguajes que existen con estas características.

Los Sistemas Multi-agente (SMA) son sistemas en los cuales varios agentes interactúan para conseguir algún objetivo o realizar alguna tarea. En estos sistemas el control es distribuido, los datos están descentralizados y la computación es inherentemente asincrónica. Además, cada agente posee información incompleta y capacidades limitadas. La implementación de SMA requiere de un soporte, i. e., una herramienta o un lenguaje que permita especificar la estructura, el comportamiento y el razonamiento de los agentes. Uno de los lenguajes más populares que se utilizan para programar agentes es Java, dada la cantidad de librerías en existencia. La

[1]Becaria de la Universidad Nacional del Sur

[2]Becario del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

[3]Parcialmente financiado por CONICET (PIP 5050), Universidad Nacional del Sur (24/ZN09), y Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 Nro 13096)

mayor desventaja que tiene este lenguaje es que fue concebido para programación orientada a objetos, y falla cuando se trata con la autonomía del agente. Actualmente se han desarrollado una variedad de plataformas de agentes, cuyo objetivo es agregar un nivel de abstracción en la programación. En ejecución, un SMA se puede plantear naturalmente como un ambiente donde los agentes se ejecutan en paralelo, pero este paralelismo debe ser programado explícitamente en la mayoría de los sistemas.

El paralelismo se ha convertido en una de las herramientas computacionales de mayor relevancia, ya que permite dividir el problema en partes, de modo tal que cada parte pueda resolverse en un procesador diferente, simultáneamente con las demás. De igual manera, los SMA proveen un modelo para resolver problemas que no pueden ser resueltos de manera satisfactoria con técnicas clásicas, ya sea por el tamaño o la naturaleza del problema.

A pesar de esta similitud existen grandes diferencias. Los SMA se han concentrado en proveer un modelo abstracto cuya intención es lograr que el programador se involucre más con el modelado del problema y la lógica del programa que con los detalles de la ejecución paralela en sí misma. Los desarrollos de paralelismo se han orientado más hacia la eficiencia, siendo la mayoría programas de cálculo numérico.

2. Programación en paralelo con un modelo de SMA

La programación en paralelo ofrece una herramienta computacional imprescindible para aprovechar el uso de múltiples procesadores y en la resolución de problemas que no pueden resolverse mediante técnicas clásicas. En el proceso de diseño de programas paralelos hay que tener en cuenta lo siguiente [9]:

1. *Descomposición*: involucra el proceso de dividir el problema y la solución en partes más pequeñas. Es decir, determinar qué parte del software realiza qué tarea.
2. *Comunicación*: se debe determinar cómo se lleva a cabo la comunicación entre los distintos procesos o computadoras, cómo sabe un componente de software cuando otro terminó o falló, cómo solicita un servicio a otro componente, qué componente debe iniciar la ejecución, etc.
3. *Sincronización*: se debe determinar el orden de ejecución de los componentes, si todos los componentes inician su ejecución simultáneamente, o alguno debe esperar mientras otros trabajan, etc.

De lo anterior es posible identificar los siguientes obstáculos que se encuentran comúnmente al programar en paralelo:

- dividir el trabajo de forma eficiente y efectiva entre dos o más componentes de software;
- diseñar e implementar una comunicación apropiada entre los componentes;
- implementar la coordinación en una ejecución concurrente o paralela de estos componentes; y
- manejar excepciones, errores y fallas parciales.

A continuación mostraremos sintéticamente las soluciones clásicas utilizadas para superar estos obstáculos, y luego se describe cómo pueden solucionarse de una forma mucho más natural si se utiliza un SMA.

a) Soluciones clásicas

En un lenguaje de programación de bajo nivel que sea multi-threaded, como por ejemplo los threads de Java o C+Pthread, los obstáculos anteriores los resuelve el programador intentando asegurar performance y correctitud. Por ejemplo, en la descomposición del trabajo, el programador va a crear nuevos threads cuando sea necesario; en la comunicación entre threads lo más común es utilizar algún área de memoria compartida. Para sincronizar el acceso a los datos compartidos, se implementa la exclusión mutua mediante *semáforos* y sus primitivas de `lock`/`unlock` (adquirir y liberar el acceso respectivamente). Si en cambio, la comunicación es entre procesos, lo más común es dejar abierto un puerto P determinado y un thread que se encargue de sensar por mensajes nuevos. La sincronización de los procesos puede hacerse también mediante el envío y recepción de mensajes. El manejo de excepciones y fallas depende del lenguaje, en Java por ejemplo, se tienen las primitivas de `throw`/`catch`/`try`.

Existen otras herramientas de bajo nivel que se basan en un sistema de pasaje de mensajes, como es el caso de la librería de funciones de PVM [8] o MPI [6]. La tarea de descomposición se resuelve mediante funciones que crean automáticamente uno o más procesos (`spawn`) o desde la línea de comandos cuando se invoca al programa (según el número de procesadores disponibles). La comunicación se lleva a cabo mediante el intercambio de mensajes con las operaciones de `send` y `receive`. La coordinación se logra sincronizando los procesos cuando llegan a cierto punto en su código, a través de *barreras*. El manejo de excepciones, errores y fallas es responsabilidad del programador, quien debe chequear la información de estado que devuelven las funciones en algún parámetro asociado.

También existen lenguajes de más alto nivel, donde el programador participa en algunas etapas del paralelismo, pero se esconden la mayoría de los detalles de la ejecución paralela de bajo nivel. Este es el caso de lenguajes como MultiPascal [11], OpenMP y HPF, entre otros. En este tipo de lenguajes existen rutinas que paralelizan los bucles, de manera que crean procesos que se encargarán de la ejecución paralela de las sentencias dentro del mismo. También se cuenta con la primitiva `fork` para crear procesos. La comunicación se realiza con una abstracción mayor, mediante *canales* que se definen entre los procesos. Los canales son variables que se comportan como una cola FIFO de valores cierto tipo y también pueden utilizarse en operaciones de sincronización. En lenguajes declarativos como BinProlog, la comunicación y sincronización pueden realizarse mediante un *blackboard* (espacio de tuplas) compartido de mensajes, donde los procesos leen y escriben (modelo Linda [4]).

b) Solución de SMA

En sistemas multi-agente estas complicaciones se resuelven naturalmente, ya que salvo el item que corresponde a la comunicación, los demás son manejados implícitamente por las capacidades de los agentes. Al utilizar agentes, el programador puede concentrarse en el correcto modelado de la tarea que debe realizar el agente, en vez de preocuparse por el control explícito del paralelismo en el programa. La descomposición ocurre simplemente por el hecho de asignar al agente una o más tareas. Debido a que los límites de los agentes son claros y bien definidos, la comunicación entre agentes es más fácil que entre módulos anónimos (procesos o threads). En cuanto a la sincronización, los problemas se ven reducidos debido a que la racionalidad propia del agente le indica cuándo debe y/o puede realizar una acción.

Cabe destacar que el empleo de agentes resulta, finalmente, en un programa en paralelo que es más fácil de *mantener y mejorar*. Corresponde a un importante cambio de paradigma, dado que simplifica la tarea del programador, a la hora de pensar la solución del problema.

3. Características deseables de un lenguaje basado en SMA

Un lenguaje de programación en paralelo basado en el modelo de Sistemas Multi-agente debería reunir como mínimo las siguientes características:

- Descomposición: el lenguaje debe proveer la funcionalidad necesaria para permitir la creación de agentes en distintas unidades de ejecución.
- Comunicación: se deben proveer primitivas de comunicación que resuelvan los protocolos de transporte, que permitan optar por operaciones sincrónicas (donde el agente se bloquea esperando por un mensaje) o asincrónicas, primitivas para la identificación de los agentes con independencia de su localización (Sistema de Manejo de Agentes), y de comunicación entre grupos de agentes (manejar adhesiones a grupos e identificación), entre otras operaciones posibles.
- Sincronización: en problemas de paralelismo usualmente se necesita sincronizar la ejecución de varias tareas en cierto punto de la computación. Para tener un mecanismo similar entre los agentes se podría utilizar un *Agente Barrera*, al que los agentes que necesitan sincronizarse le envían un mensaje para que este agente especial les avise cuándo pueden continuar (una vez que todos estén esperando en el mismo punto). Otro tipo de sincronización puede manejarse con operaciones de comunicación o protocolos de interacción.
- Performance: la sobrecarga que surge a partir de la creación de agentes y su comunicación debe ser la menor posible, a fin de no afectar la ganancia que se obtiene de una ejecución en paralelo.

Si el lenguaje de programación reúne las características antes mencionadas, automáticamente se obtiene la capacidad de representar el paralelismo a alto nivel, facilitando a su vez la tarea del programador, mediante un lenguaje con un estilo similar al de la programación secuencial. Hay otras características que podrían incluirse al disponer de un modelo de sistemas multi-agente, tales como:

+ Soportar movilidad de agentes.
+ Implementar algún lenguaje de comunicación entre agentes (ACL).
+ Utilizar tecnología "estandarizada" (e. g., FIPA, KQML, entre otros).
+ Otorgar flexibilidad en los lenguajes en que pueden programarse los agentes (multi-lenguaje).
+ Facilitar el desarrollo de distintos tipos de agentes (colaborativos, deliberativos, BDI, de propósito general, etc).
+ Brindar una interface gráfica al usuario.

La movilidad de agentes es importante tanto para la descomposición del problema como para la performance, ya que se podría cambiar el mapeo a los procesadores en tiempo de ejecución. El agente podría recibir una nueva tarea a resolver, para lo cual le conviene migrar de procesador y así disminuir el costo de las comunicaciones o estar más cerca de los recursos que necesita.

Un lenguaje de comunicación estándar permite que los agentes puedan comunicarse conociendo los tipos de mensajes que pueden enviar y recibir. Los protocolos de interacción son una

manera de definir normas para la coordinación de actividades entre agentes, e introducen un contexto a los mensajes transmitidos durante una conversación, facilitando así su interpretación.

Si se utiliza tecnología estandarizada y se tiene flexibilidad en cuanto al lenguaje de programación de los agentes, entonces se puede pensar en la interoperabilidad de los agentes desarrollados por distintos grupos de programadores, e incluso, empleando lenguajes de programación diferentes. Además, si el lenguaje cuenta con tipos de agentes predefinidos, estos actuarían como un constructor de alto nivel para el programador (ya podrían tener parte de la lógica implementada). Por último, si el lenguaje cuenta con una interface gráfica para el usuario es posible que sea utilizado por un público más general, sin demasiados conocimientos de programación.

4. Caso de Estudio

En esta sección presentaremos en primer lugar uno de los pocos lenguajes de programación que utilizan el modelo de agentes para programar con paralelismo. Luego, en la sección 4.2 realizaremos una evaluación de este lenguaje contrastándolo con las características elaboradas en la sección 3. El paradigma de programación de Orgel [13] plantea un ambiente de múltiples *agentes* conectados por canales de comunicación abstractos llamados *streams*. Los agentes corren en paralelo, intercambiando mensajes por el stream.

Orgel fue desarrollado alrededor del año 2000, por investigadores de la Universidad de Tecnología de Toyohashi, Japón. Los autores buscaron un modelo con estilo similar al que exhibe la programación secuencial y con un manejo de alto nivel de las comunicaciones, para que estuviera al alcance de una amplia gama de programadores. En este artículo se resaltan estas buenas decisiones de diseño y se plantean algunas limitaciones que exhibe el lenguaje, que en su mayoría se deben a que no se ha continuado con la implementación del mismo.

4.1. Resumen de Orgel

La sintaxis de Orgel está basada en C. Se agregan declaraciones de streams, agentes, mensajes y conexiones de red; sentencias para creación, transmisión y dereferencia de mensajes, terminación de agentes y servicios o funciones de agentes (para más detalles consulte el Apéndice A). Además, se reemplazan a las variables globales por variables propias de cada agente (lo cual favorece al encapsulamiento).

El comportamiento de streams y agentes está definido por los *tipos* de streams y de agentes. Cuando un programa en Orgel es ejecutado, se crea y se corre un agente `main`. Si este agente contiene definiciones de variables de tipo agente o stream, sus instancias son automáticamente creadas, los streams se conectan a los agentes y los agentes comienzan su ejecución en paralelo.

Un stream es un canal de mensajes abstracto, que tiene la dirección del flujo de mensajes, y uno o más agentes pueden estar conectados a cada extremo. La declaración es la siguiente:

```
stream StreamType [inherits StreamType1 [, ...]] {
   MessageType[(Type Arg: Mode [, ...])];
   ...
};
```

En la declaración se enumeran los tipos de mensajes que el stream *StreamType* está preparado para manejar. Un tipo de mensaje en Orgel toma la forma de una función: *MessageType* es usado como identificador del mensaje, y una lista de argumentos con tipos y modos de entrada/salida (`in/out`).

Un agente es una unidad de ejecución activa, que envía mensajes a otros agentes mientras realiza su computación. La forma de la declaración de un tipo de agente es similar a la declaración de una función de C; la sintaxis es:

```
agent AgentType([ StreamType StreamName: Mode [, ...]]) {
    member function prototype declarations
    member variable declarations
    connection declarations
    initial Initializer;
    final Finalizer;
    task TaskHandler;
    dispatch (StreamName) {
        MessageType: MessageHandler;
        ...
    };
};
```

Los argumentos de un tipo agente son sus streams de entrada/salida con tipos y modos. Las funciones de agente se declaran de la misma forma que una función en C++, siguiendo la convención *AgentType*::*FunctionName*. El entorno de las variables del agente se extiende dentro de la declaración de agente y de las funciones de servicio. Las declaraciones de conexión especifican cómo conectar a los agentes y streams definidos como variables de agente.

Los últimos cuatro elementos: `initial`, `final`, `task` y `dispatch`, definen manejadores de eventos. Los handlers son código C secuencial, con extensiones para manejar mensajes. Los procedimientos *Inicializer* y *Finalizer* son ejecutados en la creación y destrucción del agente, respectivamente. *TaskHandler* define la computación propia del agente; es ejecutado cuando el agente no está procesando mensajes. La declaración `dispatch` especifica el handler para cada tipo de mensaje procedente de un stream de entrada *StreamName*, enumerando los tipos *MessageType* de mensajes aceptables y el código del handler *MessageHandler*. Esta declaración funciona como un framework para el procesamiento asincrónico de mensajes.

4.2. Evaluación

Algunas de las características planteadas en la sección 3 están presentes en Orgel. Por ejemplo, el lenguaje estudiado exhibe una eficiencia y estilo similar al de la programación secuencial. Asimismo, adopta un modelo de programación más efectivo, donde el programador puede contribuir en el desarrollo de una implementación eficiente mediante la especificación abstracta del paralelismo y las comunicaciones. [13]

El tipo de los agentes soportados es amplio y flexible, pero queda como responsabilidad del programador la definición de modelos de agentes colaborativos, deliverativos o BDI, los cuales suelen estar incluidos en *plataformas de agentes* (ver [12]).

Una interface gráfica con el usuario podría acercar el lenguaje a más programadores. Sobre todo cuando se considera la declaración de agentes, streams y tipos de mensajes, y la conexión de los streams. El diseño es bastante abierto: al estar programado sobre C, es posible que el programador defina sus propias librerías y que las utilice en sucesivos desarrollos. La declaración de los agentes y su funcionalidad, al estar autocontenida en la misma declaración (como si fuese un objeto), es totalmente reutilizable.

Funcionalidad de streams y mensajes

La abstracción de los canales de comunicación que provee el stream es muy útil desde el punto de vista del programador. Sin embargo, aún para transferir tan sólo un entero, se debe construir un mensaje, declarar un tipo de mensaje y enviar el mensaje vía stream. Esto puede ser visto como un punto en contra, pero se alivia si se tiene en cuenta que el overhead que se presenta en este caso no es tanto mayor cuando se transfieren más datos. Además, se tiene la funcionalidad de manejar los mensajes como si de eventos se tratara, definiendo las funciones handler que se utilizan para el procesamiento de cada tipo de mensaje.

Orgel permite la definición de argumentos de salida en los mensajes. Esto implica que se enviará el mensaje con una variable sin instanciar, y que el receptor del mensaje al instanciar la variable con un objeto mensaje envía el objeto de vuelta al emisor. La semántica no es clara en esta parte, pues quedaría implícito el stream en la otra dirección.

El modelo de comunicación que plantea el lenguaje es estático, quedando definido a través de las declaraciones de `connect`. Es decir, toda comunicación que suceda entre los agentes, al realizarse a través de los canales, debe estar prevista en tiempo de compilación. Esta característica aparta a Orgel del modelo de SMA clásico, donde hay dinamismo en las comunicaciones.

Performance

Con respecto a la performance, en [13] se presenta una evaluación del desempeño de Orgel contrastado con la librería Pthreads estándar ante dos ejemplos de prueba. Los resultados muestran que programando en Orgel se pueden obtener buenos speed-ups, aunque dependiendo del ejemplo se pueden llegar a necesitar ciertas optimizaciones estáticas. Existe una sobrecarga del 5-11 % cuando se utiliza Orgel en lugar de C. La sobrecarga de conmutación de threads en Orgel es un 22 % mayor que con C y Pthreads.

Como reporta [13], se supone que Orgel debería tener menos sobrecarga que otras alternativas de programación con semánticas de ejecución paralela, pero no hay pruebas al respecto.

Implementación

La implementación de Orgel existente utiliza Pthreads y soporta ejecución concurrente en una máquina con un único procesador o ejecución en paralelo en máquinas multiprocesador (en máquinas SPARC con Solaris). El paquete consiste de un compilador Orgel y unas librerías de soporte para el manejo de los agentes y los streams de comunicación.

El compilador cumple principalmente la función de un pre-procesador. Es en realidad un traductor a código C de las sentencias de Orgel, luego compila automáticamente el código generado con el compilador de C, y finalmente es vinculado con las librerías de Orgel para generar el ejecutable.

Para suprimir el número de threads por eficiencia, y para permitir una creación demorada de agentes, cada instancia de agente se representa en un registro de agente. Correspondiendo a la creación lógica de agentes, se crean los registros de agentes con los parámetros adecuados. El ambiente de ejecución de Orgel planifica agentes utilizando estos registros, y crea nuevos threads en caso de que sea necesario. Este modelo es interesante porque posibilita que el planificador realice un balance de carga entre los procesadores.

La implementación actual se basa en un ambiente multithreaded, es por eso que se compara con Pthreads. No obstante, cuando se trata de sistemas multi-agente, una ejecución concurrente en un procesador no es del todo real. Si cada agente es una entidad independiente, sería bueno que más de uno de ellos esté activo al mismo tiempo, e incluso que se encuentren dispersos geográficamente.

Los autores de Orgel tenían planeado el desarrollo de una implementación para procesadores con memoria distribuida, pero aparentemente no se ha llevado a cabo. Esto limita mucho las capacidades de experimentación con el lenguaje, pues no siempre se cuenta con una máquina de múltiples procesadores y memoria compartida. Es más fácil la experimentación con un lenguaje como PVM o MPI, porque siempre se dispone de una red de computadoras donde correr los programas.

Para un lenguaje de agentes, sería interesante que el código de los mismos pueda ser distribuido en varias máquinas, y la configuración pueda ser controlada mediante una interface gráfica remota. Un paso mayor sería que la configuración pueda incluso ser cambiada en tiempo de ejecución, moviendo agentes de una máquina a otra cuando sea necesario, al estilo de Jade.

5. Conclusiones

En este trabajo hemos planteado un conjunto de características deseables que deberían estar presentes en un lenguaje de programación en paralelo orientado a agentes. Estas características constituyen la funcionalidad básica que debe proveer el lenguaje de programación para hacer posible la codificación de programas con paralelismo.

El proceso de diseño de programas paralelos incluye tres tareas básicas: descomposición, comunicación y sincronización de las partes. Mediante la utilización de un sistema multi-agente estas complicaciones se resuelven naturalmente, ya que salvo el item que corresponde a la comunicación, los demás son manejados implícitamente por las capacidades de los agentes. De este modo, el programador puede concentrarse en el correcto modelado de la tarea que debe realizar el agente, en vez de preocuparse por el control explícito del paralelismo en el programa.

Se presentó un resumen de los distintos lenguajes de programación de alto y bajo nivel que constituyen las soluciones tradicionales a los problemas de paralelismo. Asimismo, se evaluó uno de los pocos lenguajes de programación que utilizan el modelo de agentes para programar con paralelismo. Orgel, gracias a su estructura de diseño, se adapta a la autonomía e independencia de los agentes facilitando un desarrollo ágil, y el mantenimiento y reuso de los componentes de software. En Orgel la descomposición ocurre al inicio del programa, cuando el agente `main` crea a todos los demás agentes. La comunicación y sincronización se lleva a cabo mediante los streams. La sincronización entre agentes de un sistema multi-agente se considera de más alto nivel, por lo que se asocia con políticas de conversación o protocolos de interacción, que pueden quedar fuera del lenguaje y en un futuro ser incorporados mediante funciones de librería.

Apéndice A. Detalles de Orgel

En este apéndice incluiremos algunos detalles para completar la descripción de Orgel, con el objetivo de que este trabajo sea autocontenido. De todas formas, para mayor información puede consultar a los autores en [13]. Como última sección de este apéndice se incluye un ejemplo que ilustra el uso de las primitivas presentadas.

A.1. Declaración de conexión

La conexión entre agentes y streams puede ser especificada de la siguiente forma:

`connect` [*Agent0.S0 Dir0*] *Stream Dir1 Agent1.S1* ;

La declaración toma una variable de tipo stream, *Stream*, y los streams de entrada/salida *S0*, *S1* de las variables de agente *Agent0*, *Agent1*. La palabra reservada `self` puede ser utilizada en lugar de una variable de agente para conectar al agente que contiene la declaración de conexión. *Dir0* y *Dir1* son especificadores de dirección (`==>` o `<==`) que indican el flujo del mensaje.

Si la variable de agente o stream es un arreglo, se necesita un especificador de subíndice de la forma [*SubcriptExpression*]. Si la expresión entre corchetes es una constante, el argumento representa un elemento del arreglo. Si la expresión se omite, el argumento representa todos los elementos del arreglo. La expresión del subíndice puede contener un identificador llamado *pseudo-variable*, que funciona como una variable cuyo entorno es local a la declaración de la conexión, y representa todos los valores enteros, con la restricción de que no se excedan los límites del arreglo.

Si el stream tiene múltiples receptores, en el momento de enviar un mensaje se realiza un *multicast* a todos los receptores. Los mensajes desde múltiples emisores son ordenados en forma secuencial no determinísticamente. Dado que la declaración de conexión se define estáticamente, el compilador puede hacer un análisis preciso para optimizar la planificación y las comunicaciones.

A.2. Mensajes

Utilizando los tipos definidos en la declaración de tipos de stream, se pueden definir variables para los mensajes. Una variable de mensaje actúa como una variable lógica en los lenguajes de programación en lógica: puede estar ligada o sin ligar. En su estado inicial se encuentra sin ligadura (unbound), y recibe una ligadura cuando se le asigna un objeto de tipo mensaje u otra variable de mensaje. El estado de ligadura (bound) tiene dos casos: si a la variable se le asigna un objeto mensaje, el estado es *instanciado*; si la variable se asigna a otra variable que está sin instanciar, el estado es *sin instanciar*. En el último caso, una asignación posterior puede cambiar el estado de todas las variables relacionadas a instanciadas.

A.3. Crear y enviar mensajes

Para crear un objeto de tipo mensaje, se describen en forma funcional el tipo de mensaje y los argumentos actuales. El tipo de un argumento de mensaje debe ser cualquier tipo de dato de C (excepto el tipo puntero) o cualquier otro tipo de mensaje.

En el primer caso, el argumento es almacenado en el mismo objeto mensaje. Si el argumento es un arreglo, el argumento actual es un puntero indicando la cabeza del arreglo de datos del tamaño declarado.

En el caso de que un argumento sea de tipo mensaje, el modo puede ser `in` o `out`. Si el argumento es de entrada, el parámetro actual puede estar sin instanciar en la creación del mensaje, pero debe ser instanciado por el emisor en su momento. Si el argumento es de salida, el receptor es quien instancia el argumento, y por lo tanto el parámetro actual debe ser una variable de mensaje sin instanciar.

La sentencia *send* tiene la forma: *Stream* `<==` *Message* `;`

A.4. Recibir mensajes

Un agente conectado al extremo receptor de un stream, recibe mensajes y los procesa de acuerdo a la declaración `dispatch` de su tipo de agente. Cuando un mensaje de algún tipo permitido es recibido, el valor de los argumentos del mensaje se almacena en las variables correspondientes especificadas en el `dispatch`. De manera similar que en el caso de la creación de un mensaje, la variable que corresponde a un argumento de tipo arreglo se ve como un puntero, y se copia el área del tamaño del arreglo.

Si el mensaje tiene argumentos de salida, serán variables sin instanciar. Estas variables se ligan a las correspondientes variables del emisor, y cuando se instancian las variables con objetos de tipo mensaje, los objetos son enviados de vuelta al emisor.

Los mensajes que aparecen como argumentos de otros mensajes pueden obtenerse mediante una operación de *dereferencia*:

Variable `?==` *MessageType*`[(`*Arg1*`[, ...])];`

Si una variable de mensaje *Variable* no está instanciada, el agente que ejecuta esa expresión se suspende, y se restaura cuando otro agente instancia la variable, i. e., es bloqueante.

A.5. Ejemplo de aplicación

Para ilustrar el uso de las primitivas y declaraciones de tipos y variables de Orgel, se programará una solución para el conocido problema de *Productor-Consumidor*. En este caso tenemos un agente productor y `N` agentes consumidores. El agente productor genera `CANT` elementos de tipo entero y los envía dentro de alguno de los tres tipos de mensajes. Dependiendo del tipo de mensaje que contenga al entero, variará el procesamiento que reciba el mismo. Las declaraciones de constantes y tipos de streams con los mensajes que maneja, será:

```
#define N 10
#define CANT 15

//Stream utilizado por el Productor para enviar los elementos generados
stream SProducts {
    //tipos de elementos que maneja el stream
    Elem1(int num: in);
    Elem2(int num: in);
    Elem3(int num: in);
}
```

El agente `main` declara las variables de tipo stream, productor y consumidor, y realiza las conexiones de los streams. Cada stream del productor se conecta al stream de uno de los consumidores. Nótese la ausencia del subíndice del arreglo en la declaración de conexión. Esto vale porque, como se dijo anteriormente, si no se especifica un subíndice, se reemplaza por todos los valores enteros permitidos.

```
agent main(void) {
    AProducer Producer; //Agente Productor
    AConsumer Consumer[N]; //N Agentes Consumidores
    SProducts Products[N]; //Streams que conectan al Prod. con los Cons.
    connect Producer.Prod[] ==> Products[] ==> Consumer[].Prod;
}
```

A continuación se transcribe el código del agente productor y del agente consumidor. Los distintos tipos de mensajes se generan de acuerdo a un número aleatorio. El agente consumidor implementa las funciones de agente `ProcesarElemX(int)` de acuerdo a cada tipo de mensaje.

```
agent AProducer(SProducts Prod[N]: out) { //....Agente Productor....
//Genera CANT productos de distintos tipos para cada consumidor
    int i,j,Rnd;
    task {
        for (i=1;i++;i<CANT) {
            for (j=0;j++;j<N) {
                Rnd = random() MOD 3;
                if (Rnd==0) Prod[j] <== Elem1(i);
                if (Rnd==1) Prod[j] <== Elem2(i);
                if (Rnd==2) Prod[j] <== Elem3(i);
            }
        }
        terminate;
    }
}

agent AConsumer(SProducts Prod: in) {  //.....Agente Consumidor.....
//Consume productos de distintos tipos
    ProcesarElem1(int n){...}
    ProcesarElem2(int n){...}
    ProcesarElem3(int n){...}

    dispatch (Prod) {
        Elem1(num): {ProcesarElem1(num)}
        Elem2(num): {ProcesarElem2(num)}
        Elem3(num): {ProcesarElem3(num)}
    }
}
```

Referencias

[1] Gail A. Alverson, William G. Griswold, Calvin Lin, David Notkin, and Lawrence Snyder. Abstractions for Portable, Scalable Parallel Programming. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):71–86, 1998.

[2] H.E. Bal. A Comparative Study of Five Parallel Programming Languages. In *EurOpen Spring Conference on Open Distributed Systems*, pages 209–228, Tromso, May 1991.

[3] Heri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[4] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.

[5] Hesham El-Rewini and Ted G. Lewis. *Distributed and Parallel Computing*. Manning Publications Co., Hardbound, 1997.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm.

[7] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. http://www-unix.mcs.anl.gov/dbpp/.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[9] Cameron Hughes and Tracey Hughes. *Parallel and Distributed Programming Using C++*. Addison Wesley, Aug. 2003.

[10] Y. Labrou. Standardizing Agent Communication. In *Proceedings of the Advanced Course on Artificial Intelligence (ACAI'01)*. Springer-Verlag, 2001.

[11] Bruce P. Lester. *The Art of Parallel Programming*. Prentice-Hall, Inc., 1993.

[12] Tulio Marchetti and Alejandro J. García. Evaluación de Plataformas para el Desarrollo de Sistemas Multiagente. In *IX Congreso Argentino de Ciencias de la Computación (CACiC'03)*, pages 625–636, La Plata, Buenos Aires.

[13] Kazuhiko Ohno, Shigehiro Yamamoto, Takanori Okano, and Hiroshi Nakashima. Orgel: A Parallel Programming Language with Declarative Communication Streams. In *Proc. of the 3rd Int. Symposium on High Performance Computing (ISHPC'00)*, volume 1940 of *Lecture Notes in Computer Science*, pages 344–354. Springer, 2000.

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