"Las grandes obras son hechas no con la fuerza, sino con la perseverancia."
Samuel Johnson

Capítulo 3

Implementación de la plataforma de distribución GALATEA con FIPAOS

En este capítulo realizamos la implementación de la plataforma distribuible del simulador multi-agente GALATEA. Comenzamos en la sección 3.1 describiendo las diferentes clases que la componen calsificándolas de acuerdo a las funciones que realizan cada una de ellas dentro de la plataforma de distribución también explicamos las características y métodos más importantes de cada una de las clases. En la sección 3.2 explicamos la estructura de archivos final del paquete `galatea.hla.fipaos` y `galatea.hla.fipaos.remote`. También se muestra sus clases ejecutables, la documentación de la plataforma de distribución, los códigos fuentes y los ejemplos.

3.1. Descripción de las clases de la plataforma de distribución Multi-Agente GALATEA

La implementación de la plataforma de distribución multi-agente GALATEA fue realizada utilizando el manejador de agentes FIPAOS. La plataforma está compuesta por un conjunto de clases Java. En la figura E.3 se representan las clases que contienen a los agentes Gestionadores de la plataforma (en color azul, símbolo estrella), la nueva clase `GInterface` (en color vinotinto, símbolo sombrilla), la clase que contiene los agentes GALATEA que se alojan en la plataforma de distribución (color verde, símbolo estatuillas) y las clases que le dan soporte a FIPAOS (color violeta, símbolo rectangulo). El producto final de la implementación son los paquetes `galatea.hla.fipaos` y `galatea.hla.fipaos.remote`. Cada uno de ellos realiza funciones específicas dentro de la plataforma de distribución.

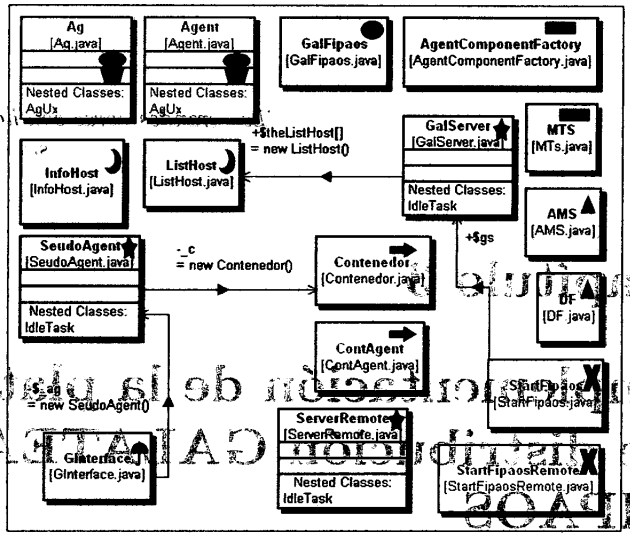

Figura 3.1: En el diagrama de clases Java se implementa la plataforma de distribución multi-agente GALATEA con FIPAOS. Las clases de color rojo (símbolo estatuilla) representan los agentes GALATEA. Las clases de color azul (símbolo estrella) representan a los agentes Gestionadores. Las clases de color purpura (símbolo rectangulo) son utilizadas para la creación de agentes FIPAOS, las clases de color morado (símbolo media luna) son utilizadas para manipular las direcciones de los agentes; las clases de color verde oscuro (símbolo flecha) son utilizadas para manejar información dentro de mensajes ACL, las clases de color naranja (símbolo triangulo) forma parte de los componentes de FIPAOS, las clases de color marrón (símbolo equis) inicializan la plataforma; y por último, tenemos la clase de color negro (símbolo óvalo) que configura los parámetros de la plataforma.

3.1. Descripción de las clases de la plataforma de distribución Multi-Agente GALATEA

A continuación describiremos los métodos y las características más importantes de cada una de las clases clasificándolas por medio de las funciones que realizan según la figura 3.2.

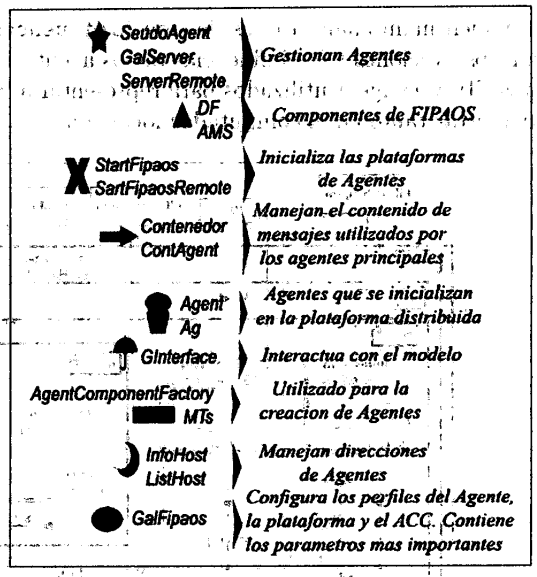

Figura 3.2: Conjunto de clases de la plataforma de distribución multi-agente GALATEA con FIPAOS clasificadas de acuerdo a las principales funciones de cada una de ellas.

3.1.1. Agentes Gestionadores

Empezamos describiendo los agentes gestionadores `SeudoAgent`, `GalServer` y `ServerRemote`. Como se aprecia en la figura 3.3 estos agentes extienden de `FIPAOSAgent`. `SeudoAgent` y `GalServer` forman parte del paquete `galatea.hla.fipaos` y `ServerRemote` pertenece al paquete `galatea.hla.fipaos.remote`. Internamente poseen la clase `IdleTask` que extiende de la clase `fipaos.agent.task.Task` la cual permite programar las diferentes tareas que realizan cada uno de ellos. El Agente `SeudoAgent` utiliza un objeto de tipo `Contenedor`, como contenido de un mensaje ACL para solicitar información al Agente `GalServer`. `GalServer` utiliza un objeto de tipo `ListHost` para almacenar los computadores de la arquitectura de distribución y los agentes del sistema que pueden estar alojados en cada uno de ellos, también utiliza un objeto de tipo `URL` perteneciente al paquete `fipaos.util.URL` para obtener la dirección de los computadores donde se encuentran los `ServerRemote`. Por su parte `ServerRemote` posee objetos de tipo `Map` y `List`, para almacenar los agentes que cada uno de ellos hayan inicializado en cada uno de los computadores donde ellos se encuentran alojados. Los agentes gestionadores `SeudoAgent` y `ServerRemote`, manipulan internamente objeto de tipo `galatea.hla.Agent` cuando el `ServerRemote` inicializa un

agente, también manipulan objetos de tipo `LInfluences`, `LOutput` y `Element` para manipular observaciones y las influencias de los agentes. En la figura 3.3 observamos dos objetos FIPAOSAgent utilizados para representar al `ServerRemote` como objetos que pueden estar tanto en el computador local como en algún computador remoto.

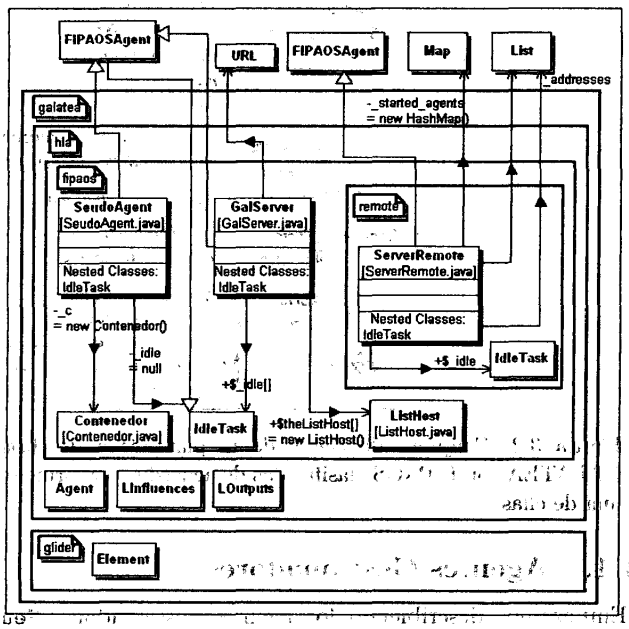

Figura 3.3: El diagrama de clases muestra los Agentes Gestionadores `SeudoAgent`, `GalServer` y `ServerRemote`. Estos extiende de la clase FIPAOSAgent. Internamente poseen la clase IdleTask, que le permite realizar las tareas dentro de la plataforma de distribución. Utilizan los objetos ListHost, Map y List, para llevar control interno de los agentes que se almacenan, y manipulan objetos de tipo Agent, LInfluences, LOutput y Element, pertenecientes al paquete de galatea.hla.* y galatea.glider.*, para gestionar las diferentes tarea del simulador, dentro de la plataforma de distribución.

SeudoAgent

Agente heredado de `FIPAOSAgent`. Por medio de este agente `GInterface` se comunica con los agentes del sistema que son heredados de la clase `galatea.hla.fipaos.Ag` y `galatea.hla.fipaos.Agent`.

Constantes más importantes

`AGENT_TYPE`: permite registrar el agente dentro del Directorio Facilitador.

`NAME`: es el nombre del Agente. A todos los agentes que interactúan con la `GInterface` por medio de `SeudoAgent` se le asigna el valor de esta constante a la etiqueta `owner`, de esta forma `SeudoAgent` pasa ser dueño del agente creado.

Instancias más importantes

`_idle`: instancia de la clase `IdleTask`. `IdleTask` es una clase anidada de `SeudoAgent`, heredada de la clase `fipaos.agent.task.Task`. Por medio de está clase se gestiona las diferentes tareas de `SeudoAgent`.

`c_`: instancia de la clase `Contenedor`. Por medio de este objeto se le envía a los diferentes servidores remotos los parámetros necesarios para que un agente sea inicializado.

Métodos más importantes

`sendAgent`: Es un conjunto de métodos (ver código 1) que permite enviarle a los diferentes `ServerRemote` los parámetros necesarios para inicializar los agentes. Entre los parámetros más importantes se encuentran:

- `class_name`, nombre de la clase.
- `agent_name`, nombre del agente.
- `agent_owner`, dueño del agente.
- `n`, número de metas permanentes del agente.
- `m`, cantidad de sensores a implementar en el agente.

`sendMessage`: permite enviarle un mensaje a otro agente. Utiliza el método *sendMessage* de su clase interna `IdleTask` para tal fin. Sus parámetros más importantes son (ver código 2):

- `content`, almacena el contenido del mensaje que se desea enviar a otro agente.
- `performative`, utilizado para definir el tipo de mensaje que se va utilizar.
- `name_agent_receiver`, contiene el nombre del agente que recibe el mensaje.
- `host_agent_receiver`, contiene el nombre del computador asociado al agente que recibe el mensaje.

Code 1 Métodos que permiten enviar agentes

```
public void sendAgent(
   String class_name,
   String agent_name
)
public void sendAgent(
   String class_name,
   String agent_name,
   String agent_owner
)
public void sendAgent(
   String class_name,
   String agent_name,
   int n
)
public void sendAgent(
   String class_name,
   String agent_name,
   String agent_owner,
   int n
)
public void sendAgent(
   String class_name,
   String agent_name,
   int n,
   int m
)
public void sendAgent(
   String class_name,
   String agent_name,
   String agent_owner,
   int n,
   int m
)
```

Code 2 Método para enviar mensajes

```
public  void sendMessage(
   String content,
   String performative,
   String name_agent_receiver,
   String host_agent_receiver,
   String name_agent_sender
)
public  void sendMessage(
   Object content,
   String performative,
   String name_agent_receiver,
   String host_agent_receiver,
   String name_agent_sender
)
```

- `name_agent_sender`, identifica el nombre del agente que envía el mensaje.

`shutdown`: invoca el método "`shutdown`" de un agente particular para ser apagado.

```
public  void shutdown(String name_agent)
```

- `name_agent`, nombre del agente que será apagado

`invoke`: le permite a `GInterface` invocar cualquier tipo de método de un agente, los parámetros más importantes se muestra en el código 3.

Code 3 Invoca un método dinámicamente

```
public Object invoke(
    String name_agent,
    String method,
    Object[] args
)

public Object invoke(
    String name_agent,
    String method
)
```

- `name_agent`: nombre del agente al que se le invoca el método.
- `method`: nombre del método hacer invocado.
- `args`: arreglo de objetos que almacena los diferentes parámetros del método ha invocar.

La clase interna IdleTask de SeudoAgent

Esta clase maneja las tareas internas del agente `SeudoAgent` y controla las conversaciones que puede llevar con los agentes GALATEA del sistema, con el agente `GalServer` y `ServerRemote`. Por medio de las diferentes performativas establecidas en FIPAOS a través de los métodos `handleX` que sobre-escribe de su clase madre `Task`, por ejemplo:

```
public void handleX( Conversation conv );
```

Le permite al agente manejar las conversaciones donde se utiliza la performativa de tipo `X`. Dentro de los métodos más importante están:

`handleSubscribe(Conversation conv)`: a través de este método se recibe información de la inicialización de un agente en el computador local o en los computadores remotos proveniente de los diferentes `ServerRemote`.

`handleInformIf(Conversation conv)`: permite gestionar las diferentes conversaciones del agente donde se utiliza la performativa `INFORMIF`, handleInformIf recibe mensajes de los métodos propios de los agentes GALATEA que se heredan de `galatea.hla.fipaos.Ag`.

`handleInform(Conversation conv)`: gestiona las diferentes conversaciones del agente donde se utiliza la performativa `INFORM`, handleInform recibe mensajes de los métodos propios de los agentes GALATEA de tipo `Agent.java`.

`handleRequest(Conversation conv)`: recibe mensajes de los agentes cuando sus métodos son invocados de manera dinámica.

`handleInformRef(Conversation conv)`: recibe las listas de influencias que son enviadas por los `ServerRemote`.

`handleCfp(Conversation conv)`: recibe información por parte del servidor principal de los diferentes computadores que tiene disponible el sistema, y cuantos agentes hay inicializados en cada uno de ellos.

`toProcess`: procesa la información recibida de los agentes de tipo `Ag` y `Agent`. Recibe un objeto de tipo `ContAgent` enviado por los agentes con el contenido que `GInterface` le retorna al simulador.

GalServer

Permite llevar control de los diferentes `ServerRemote`, que serán los encargados de poder inicializar los agentes tanto en el computador local, como en los computadores remotos. Lleva el registro de la cantidad de agentes que se han inicializado en cada computador. También almacena el nombre del agente que se ha inicializado y el computador asociado donde se inicializó.

Instancias más importantes

`host`: contiene todos los computadores que están disponibles donde se pueden inicializar los diferentes agentes del sistema.

`auxHost`: vector de arreglo que contiene la cantidad de `ServerRemote` que hay en la plataforma, para inicializar agentes GALATEA de manera distribuida.

`_idle`: permite al agente poder manipular sus tareas internas.

`ag_name_addresses`: almacena el nombre de cada agente y su computador asociado donde se encuentra corriendo.

La clase interna IdleTask de GalServer

Maneja las diferentes tareas del agente

Instancias más importantes

info_host: almacena el número de computadores que hay en la plataforma de distribución y la cantidad de agentes que están alojados en cada uno de ellos.

Métodos más importante

handleConfirm(Conversation conv); recibe información cuando es iniciado un **ServerRemote** en el computador local o en algún computador remoto.

handleCfp(Conversation conv): maneja las diferentes conversaciones del agente donde se utiliza la performativa **CFP**. Recibe solicitud de información por parte de **SeudoAgent** de los diferentes computadores que tiene disponible del sistema, y cuantos agentes están inicializados en cada uno de ellos. También le permite actualizar la lista de agentes cuando **SeudoAgent** inicializa un nuevo agente.

ServerRemote

ServerRemote pertenece al paquete **galatea.hla.fipaos.remote**. Se encarga de enviar al servidor principal la dirección del computador donde se está corriendo el **ServerRemote** y está atento a recibir cualquier agente de **SeudoAgent** para ser inicializado.

Métodos más importantes

startAgent: este método permite inicializar un agente. Sus parámetros más importantes son (ver código 4):

Code 4 Método que inicializa un agente

```
public void startAgent(
    final Contenedor _c,
    boolean async,
    final String owner_host
);
```

- _c: contiene todas los parámetros necesarios para inicializar el agente.
- async, define si el agente será sincronizado o no.

- `owner_host`, indica el nombre del computador donde se encuentra inicializado el dueño del agente.

`sendMessage`: permite a todos los agentes que se encuentran corriendo dentro del computador donde se inicializó el agente `ServerRemote` enviar mensajes a otros agentes. Este método utiliza el objeto `_idle.sendMessage()`, que es una instancia de su clase interna `IdleTask`. Sus parámetros más importantes son (ver código 5):

Code 5 Método que puede ser utilizado para que otros agentes puedan enviar mensaje

```
public static void sendMessage(
    Object content,
    String performative,
    AgentID sender,
    AgentID receiver
)
public static void sendMessage(
    String content,
    String performative,
    AgentID sender,
    AgentID receiver
)
```

- `content`: almacena el contenido del mensaje hacer enviado.
- `performative`: utilizado para definir de que manera va ha recibir el mensaje su agente homólogo.
- `sender`: identificador del agente que envía el mensaje.
- `receiver`: identificador del agente que recibe el mensaje.

La clase interna IdleTask de ServerRemote

Esta clase permite programar las diferentes tareas de `ServerRemote`

Métodos más importantes

`sendAddresses`: envía la dirección donde se está corriendo el `ServerRemote` al servidor principal, de esta forma el servidor tiene ya el nombre del computador que puede estar disponible para inicializar un nuevo agente. La dirección es enviada a los siete segundos de su activación por medio de la programación automática de tareas.

`handleSubscribe(Conversation conv)`: por medio de este método el `ServerRemote` recibe de `SeudoAgent` todos los parámetros necesarios para inicializar el agente.

handleInformRef(Conversation conv): recibe notificación por parte de SeudoAgent para recojer todas las influencias de los agentes que estan en el computador local donde se inicializó ServerRemote, para que sean enviadas a SeudoAgent y posteriormente ser procesadas.

gatherInfluences: es invocado dentro de handleInformRef para recoger las influencias de todos los agentes que inicializó ServerRemote.

3.1.2. Componentes de FIPAOS

DF

Extiende de la clase fipaos.platform.DirectoryFacilitador, se encargada de proporcionar el servicio de páginas amarillas a otros agentes.

AMS

Extiende de la clase fipaos.platform.AgentManagementSystem es la encargada de controlar las actividades de los agentes dentro de la plataforma, contiene la lista de todos los agentes y debe asegurar el buen funcionamiento de todos los componentes que están dentro y fuera de la plataforma.

3.1.3. Clases que inicializan la plataforma de agentes

StartFipaos:

Esta clase permite activar los servicios necesarios para el funcionamiento de FIPAOS, tales como el Protocolo de Transporte de Mensajes internos y externos. Inicializa el ACC, AMS, DF, GalServer, y el ServerRemote del computador local.

StartFipaosRemote:

Esta clase pertenece al paquete galatea.hla.fipaos.remote, y se encarga de activar los servicios necesarios para el funcionamiento de FIPAOS en los computadores remotos, tales como el Protocolo de Transporte de Mensajes internos y externo. Inicializa también el ACC, AMS, DF y por último el agente gestionador ServerRemote.

3.1.4. Clases que manejan el contenido de mensajes de los agentes gestionadores

Contenedor

Pertenece al paquete `galatea.hla.fipaos`, implementa la interface `java.io.Serializable` y es utilizada por `SeudoAgent` para enviarle al `ServerRemote` todo lo necesario para inicializar un agente. Para tal fin envía un mensaje ACL con: nombre del agente, nombre de la clase, dueño del agente; así como también los demás parámetros de iniciación. Los métodos más importantes que poseen son los diferentes métodos `set` y `get` que describen las ontología necesarias para inicializar los agentes.

ContAgent

Esta clase pertenece al paquete `galatea.hla.fipaos.remote` le permite a `SeudoAgent` envíar a los agentes de tipo `galatea.hla.fipaos.Agent` y `galatea.hla.fipaos.Ag` las diferentes peticiones solicitadas por `GInterface`. También recibe el contenido enviado por cada uno de ellos.

`ContAgent` implementa la interface `java.io.Serializable` para poder asignarlo `SeudoAgent` y los agentes GALATEA dentro del contenido de un mensaje ACL, por medio de la sentencia `setContentObject`.

Está clase implementa un conjunto de métodos `set` y `get` que describen las diferentes ontologías de las consultas que se le realizan a los agentes.

3.1.5. Los Agentes GALATEA

Las clases `Ag` y `Agent` implementan a los agentes GALATEA extendiendo de los paquetes `galatea.glorias.Ag` y `galatea.hla.Agent` respectivamente, (figura 3.4). Ambos agentes GALATEA de la plataforma de distribución pertenecen al paquete `galatea.hla.fipaos`.

Internamente poseen un agente de tipo FIPAOSAgent para poderse alojar dentro de la plataforma de distribución, comunicarse con los agentes gestionadores y realizar las tareas solicitadas por el Modelo de GALATEA de manera remota.

Agent

Pertenece al paquete `galatea.hla.fipaos`. Es una versión de la clase `galatea.hla.Agent` a la que se le agregó una clase interna `AgUx`, de tipo `FIPAOSAgent`, como un nuevo componente para que `Agent` pueda alojarse dentro de la plataforma de distribución y de esta forma realizar actos comunicativos con los agentes gestionadores, recibir información del modelo de GALATEA, procesarla y dar respuesta

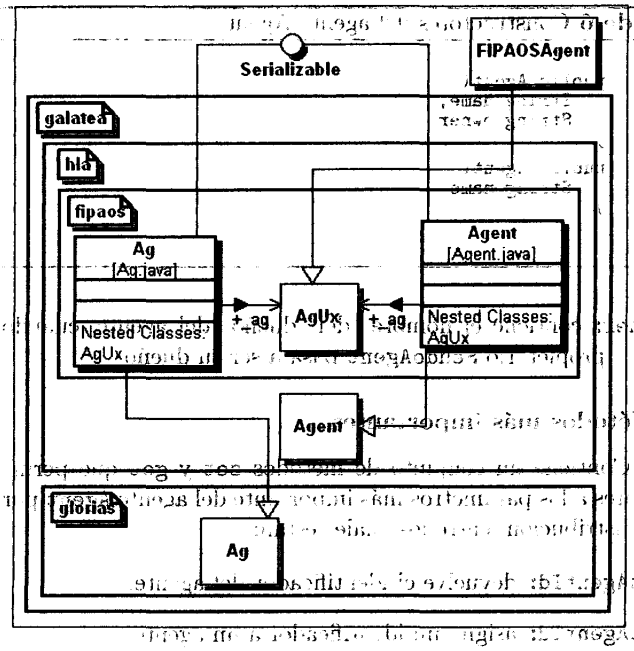

Figura 3.4: Diagrama de clases que muestra los agentes **Ag** y **Agent** de la plataforma de distribución. Ambos pertenecen al paquete **galatea.hla.fipaos**, el primero extiende de **galatea.gloria.Ag** y el segundo extiende **galatea.hla.Agent**. Poseen internamente un agente shell de FIPAOS (**AgUx**), que les permite interactuar con el modelo de manera distribuida.

de manera remota.

Instancias más importantes

_ag: crea una instancia del agente **AgUx**, le permite a **Agent** alojarse dentro de la plataforma de distribución GALATEA con FIPAOS.

Constructores

Permiten crear una instancia del agente **Agent**, los parámetros más importantes son (ver código 6):

name: contiene el nombre del agente que lo identifica, dentro del manejador de agente. Este nombre debe ser único. También es utilizado para inicializar la variable **agentType**.

Code 6 Constructores del agente Agent

```
public Agent(
   String name,
   String owner
)
public Agent(
   String name
)
```

owner: contiene el nombre del dueño del agente, cuando el agente no posee un propietario `SeudoAgent` pasa a ser su dueño.

Métodos más importantes

Contiene un conjunto de métodos `set y get` que permiten consultar y asignar valores a los parámetros más importante del agente `Agent` por medio de la plataforma de distribución, entre los cuales están:

`getAgentId`: devuelve el identificador del agente.

`setAgentId`: asigna un identificador a un agente

`getAgentType`: devuelve el tipo de agente, como se encuentra registrado dentro del DF.

`getPopulation`: devuelve la cantidad de instancias realizadas a `Agent` dentro del sistema.

`setClock`: actualiza el reloj interno del agente.

`getClock`: devuelve el valor de tiempo que posee el agente en un determinado instante.

`inputsAdd`: asigna una entrada de tipo `String` al agente por medio de `GInterface`.

`inputsAdd`: asigna una entrada de tipo `Object` al agente por medio de `GInterface`.

`getInputs`: devuelve la lista de entrada del agente.

`setInputs`: asigna una lista de entrada al agente.

`setOutputs`: asigna una lista de salida al agente.

`setOutputs`: retorna la lista de salida del agente.

`initOutputs`: inicializa la lista de salida del agente.

`resetInputs`: inicializa la lista de observaciones.

`invoke`: permite invocar de manera dinámica algún método del agente.

La clase interna AgUx de Agent

Extiende de `fipaos.agent.FIPAOSAgent` y le permite realizar a `Agent` actos comunicativos dentro de la plataforma de distribución. `AgUx` posee internamente la clase `IdleTask` que le permite manipular sus tareas internas. Los métodos más importantes que posee esta clase son:

`handleInform(Conversation conv)`: a través de este método se gestionan las diferentes solicitudes realizadas por `GInterface` para consultar los principales métodos y atributos del agente, y de esta forma `Agent` o sus clases derivadas puedan interactuar con el mundo físico.

`handleRequest(Conversation conv)`: a través de este método se gestionan las diferentes solicitudes realizadas por `GInterface` para invocar métodos de manera dinámica y así `Agent` o sus clases derivadas puedan interactuar con el mundo físico.

`sendMessage`: envía un mensaje a otro agente con una performativa específica, el identificador del agente que envía el mensaje, y el identificador del agente que recibe el mensaje (ver código 5).

`toProcess`: procesa las diferentes peticiones realizadas por `GInterface`. Recibe un objeto de tipo `ContAgent` con las posibles peticiones que pueden ser solicitadas por el modelo GALATEA, por medio de `GInterface` y retorna un objeto de tipo `ContAgent`, con la solicitud realizada.

Ag

Clase que pertenece al paquete `galatea.hla.fipaos`, y extiende de la clase `Ag` que pertenece al paquete `galatea.glorias.*`, es utilizada para implementar la comunicación dentro de la plataforma de distribución multi-agente GALATEA con FIPAOS, posee internamente un agente de tipo FIPAOS, que le permite poderse alojar dentro de la plataforma de distribución, comunicarse con los agentes gestionadores, de tal forma que el modelo por medio de la `GInterface` puedan actualizarle su ambiente, `Ag` pueda razonar y dar respuesta de manera remota, al modelo de simulación.

Instancias más importantes

_ag: crea una instancia de AgUx esta clase extiende de FIPAOSAgent, le permite a Ag poder interactuar con el modelo de GALATEA de manera remota dentro de la plataforma de distribución.

Constructores

Crean una instancia del agente Ag, sus parámetros más importantes son (ver código 7):

Code 7 Constructores del agente Ag

```
public Ag(
   int n,
   String name,
   String owner
)
public Ag(
   int n,
   String name
)
public Ag(
   int n,
   int m,
   String name,
   String owner
)
public Ag(
   int n,
   int m,
   String name
)
```

- n: indica la cantidad de metas permanentes del agente.
- m: indica la cantidad de sensores a implementar en el agente.
- name: contiene el nombre del agente que permite identificarlo dentro del manejador de agente. El nombre debe ser un nombre único. También es utilizado para inicializar la variable agentType.
- owner: contiene el nombre del dueño del agente, cuando el agente no posee un propietario su dueño es SeudoAgent.

Métodos más importantes

La clase **Ag** implementa los mismos métodos que puede realizar el agente **Agent**, y también los métodos heredados de su clase superior.

La clase interna AgUx de Ag

Es adicionado como un nuevo componente de **Ag**, esta clase le permite poderse comunicar con el Modelo de GALATEA, posee internamente la clase `IdleTask` que le permite manipular internamente sus tareas, los métodos más importantes que posee esta clase son:

`handleInform(Conversation conv)`: permite gestionar las diferentes conversaciones del agente donde se utiliza la performativa `INFORM`. Gestiona las diferentes solicitudes realizadas por `GInterface`, para consultar o actualizar los atributos que `galatea.glorias.Ag` hereda de su clase superior `galatea.hla.Agent`.

`handleInformIf(Conversation conv)`: gestiona las conversaciones del agente donde se utiliza la performativa `INFORMIF`, y atiende las diferentes solicitudes realizadas por `GInterface` para consultar los métodos propios que **Ag** hereda de su clase superior `galatea.gloria.Ag`.

`handleRequest(Conversation conv)`: atiende todas las solicitudes realizadas por `GInterface`, para invocar métodos de manera dinámica.

sendMessage: envía un mensaje a otro agente con una performativa específica. (ver código 5).

3.1.6. Clase que interactúan con el Modelo de GALATEA

GInterface

Está clase extiende de `galatea.hla.GInterface` figura 3.5. Crea una instancia de **SeudoAgent**, y por medio de él, realiza todas las gestiones que se pueden hacer con agentes dentro del modelo y la plataforma distribuida del simulador multi-agente de GALATEA, tales como la incorporación al proceso de simulación a los agentes, activarlos o destruirlos. Permite que los agentes puedan recibir las observaciones y recoge la lista de influencia que los agentes GALATEA puedan generar durante su proceso de razonamiento y que influyen dentro de la dinámica del modelo. Los agentes sólo se pueden relacionar con el mundo físico mediante la `GInterface`.

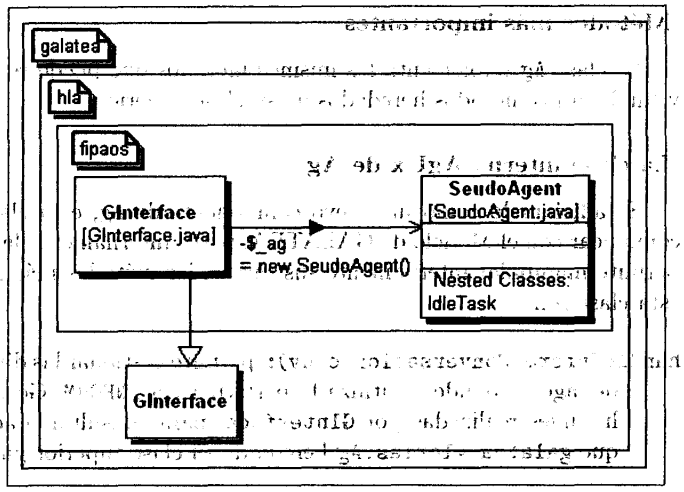

Figura 3.5: La nueva `GInterface` extiende de `galatea.hla.GInterface`. Crea una instancia de `SeudoAgent` para que dentro del modelo se puedan realizar todas las gestiones a los agentes de manera remota.

Instancia más importante:

`_ag`: extiende de `galatea.hla.fipaos.SeudoAgent`, le permite a `GInterface` comunicarse con todos los agentes de tipo `galatea.hla.fipaos.Agent`, `galatea.hla.fipaos.Ag` o cualquier otro agente FIPAOS que esté funcionando dentro de la plataforma de distribución.

Métodos más importantes que posee

`init`: invoca al método `init` de `galatea.hla.GInterface.init` para retornar una referencia a la interfaz del usuario (ver código 8), su parámetro más importante es:

- `interfaz`: posee el nombre de la clase que inicializa la interfaz del usuario.

Code 8 Método que permite realizar una referencia a la interfaz del Usuario

```
public static void init(Object interfaz)
```

initAgent: inicializa un agente en el computador local o en el computador remoto, sus parámetros más importante son (ver código 9):

Code 9 Métodos que permiten inicializar agentes

```
public static void initAgent(
    String class_name,
    String agent_name
)
public static void initAgent(
    String class_name,
    String agent_name,
    String owner_name
)
public static void initAgent(
    String class_name,
    String agent_name,
    int n
    )
public static void initAgent(
    String class_name,
    String agent_name,
    String owner_name,
    int n
)
public static void initAgent(
    String class_name,
    String agent_name,
    String owner_name,
    int n,
    int m
)
public static void initAgent(
    String class_name,
    String agent_name,
    int n,
    int m
)
```

- **class_name**: almacena el nombre de la clase del agente que será inicializado.
- **agent_name**: almacena el nombre del agente que es inicializado, además pasa a formar parte del identificador del agentes junto al las direcciones que posee.
- **owner_name**: almacena el nombre del dueño del agente inicializado. Cuando esta etiqueta no es utilizada, **SeudoAgent** pasa hacer el dueño del agente por omisión.
- **n**: número de metas permanentes del agente y se utiliza cuando se inicializan agentes que extienden de **Ag**.

- m: indica la cantidad de sensores a implementar en el agente, se utiliza cuando se inicializan agentes que extienden de **Ag**.

process_test: procesa la lista de influencia de todos los agentes.

shutdown: apaga un agente de la plataforma de distribución, y elimina el registro del agente en el AMS y DF, luego apaga el agente dentro de la Máquina Virtual de Java, su parámetro principal es el nombre del agente.

Code 10 Métodos disponibles para invocar métodos de manera dinámica

```
public static Object invoke(
   String name_agent,
   String method,
   Object[] args
)
public static Object invoke(
   String name_agent,
   String method
)
public static void invoke1(
   String name_agent,
   String method,
   Object[] args
)
public static void invoke1(
   String name_agent,
   String method
)
public static int invoke2(
   String name_agent,
   String method,
   Object[] args
)
public static int invoke2(
   String name_agent,
   String method
)
public static String invoke3(
   String name_agent,
   String method,
   Object[] args
)
public static String invoke3(
   String name_agent,
   String method
```

invoke: permite invocar métodos remotos de los agentes de manera dinámica, existen cuatro alternativas la primera **invoke**, utilizado cuando el método a invocar devuelve un objeto. El método **invoke1** es utilizado cuando el método que se llama

no devuelve ningún tipo de parámetro. `invoke2` utilizado cuando los métodos llamados de manera dinámica devuelve un valor de tipo `int`, y por útimo tenemos los métodos `invoke2` utilizados cuando el método devuelve un valor de tipo `String`, en el código 10 observamos los parámetros más importantes:

- `name_agent`: nombre del agente.
- `method`: nombre del método a invocar.
- `args`: contiene todos los parámetros que son necesarios para invocar el método.

`setAgentID`: actualiza el identificador GALATEA de un agente particular (ver código 11):

Code 11 Asigna un nuevo identificador a un agente GALATEA

```
public static void setAgentID(String agent_name, int aid)
```

- `agent_name`: nombre del agente
- `aid`: nuevo identificador interno dentro de GALATEA.

`getAgentID`: devuelve el identificador GALATEA de un agente particular.

`getAgentType`: devuelve la identificación de un agente dentro del DF.

`getPopulation`: devuelve la cantidad de instancias que se han realizado de `Agent`, dentro del sistema.

`updateClockAgent`: actualiza el reloj interno de un agente, (ver código 12):

- `agent_name`: nombre del agente
- `t`: valor del tiempo ha actualizar.

Code 12 Actualización del reloj interno de un agente GALATEA

```
updateClockAgent(String agent_name, double t)
```

`getClock`: muestra el valor del reloj interno de un agente.

`inputsAdd`: el agente recibe una observación del ambiente (ver código 13):

Code 13 El agente recibe observaciones del ambiente

```
public static void inputsAdd(
   String name_agent,
   String obs
)
public static void inputsAdd(
   String name_agent,
   Object obs
)
```

- **name_agent**: nombre del agente que recibe la observación.
- **obs**: observación recibida.

getInputs: devuelve la lista de entrada de un agente.

setInputs: asigna una lista de entrada a un agente particular.

setOutputs: asigna una lista de salida a un agente.

getOutputs: devuelve la lista de salida de un agente.

resumeReasoning: reinicia el proceso de razonamiento de un agente.

input: es una interfaz estándar que le permite al agente parar el proceso de razonamiento para adicionar un nueva observación a la lista de entrada **inputs**, y reiniciar nuevamente el proceso de razonamiento. (ver código 14).

Code 14 Adiciona un nueva observación de un agente por medio de la **GInterface**

```
public static void input(String name_agent, String o)
```

stopReasoning: para el proceso de razonamiento de un agente en particular.

isReasoningSuspended: retorna el estado de **suspendReasoning** el cual indica si el agente continua o no razonando.

activateSensor: activa el sensor que corresponde a una determinada posición del arreglo **Sensor** (arreglo que contiene la totalidad de sensores del agente).

addPermanentGoal: permite especificar la estructura de cada una de las metas del agente (ver código 15).

- **name_agent:** nombre del agente.

Code 15 Uso de `addPermanentGoal` dentro de la `GInterface`

```
public static void addPermanentGoal(
   String name_agent,
   String name_goal,
   int n,
   String[] obs
)
```

- `name_goal`: indica el nombre de la meta que será ejecutada.
- `n`: indica la posición de la meta en el conjunto total de metas del agente.
- `obs`: lista de las observaciones requeridas para ejecutar la meta permanente.

`addObservation`: agrega una observación a la lista de observaciones del agente (ver código 16).

Code 16 Uso de `addObservation` dentro de la `GInterface`

```
public static void addObservation(
   String name_agent,
   String obs
)
```

- `name_agent`: nombre del agente.
- `obs`: Observación que debe agregarse a la lista de observaciones del agente.

`addObservationSensor`: es usado en aquellos agentes que implementan sensores. El método asigna a un determinado sensor aquellas observaciones que son requeridas para su activación (ver código 17)

Code 17 Uso de `addObservationSensor` dentro de la `GInterface`

```
public static void addObservationSensor(
   String name_agent,
   int n,
   String[] obs
)
```

- `name_agent`: nombre del agente.
- `n`: número del sensor.
- `obs`: contiene las observaciones requeridas para activar el sensor `n`

`cycle`: activa el ciclo de razonamiento de un agente, necesita pasarle cómo parámetro el nombre del agente.

`gatherInfluences_test`: recoge todas las influencias de los diferentes agentes, tanto del computador local, como los agentes alojados en los computadores remotos, de la plataforma distribuida del Simulador.

Implementación de los métodos más importantes de las listas `inputs` y `outputs` de un agente

`inputsGet`: retorna el objeto que se encuentra en la posición `arg0` de la lista de entrada del agente.

`inputsIsEnpty`: verifica si la lista de entrada `inputs` de un agente se encuentra vacía o no.

`inputsClear`: limpia la lista de entrada `inputs` de un agente particular.

`inputsRemove`: remueve un elemento de la lista `inputs` de un agente.

`inputsEquals`: compara si un elemento de la lista es igual al argumento pasado.

`outputsHead`: muestra el primer `Elemento`, de la lista de salida `outputs`, de un determinado agente.

`outputsNum`: muestra el número de elementos de la lista de salida `outputs` de un agente.

`outputsEmpty`: verifica si la lista de salida `outputs` de un agente particular está vacía o no.

`outputsGetDat`: muestra el elemento que se encuentra en una posición dentro de la lista de salida `outputs` de un agente.

`outputsAddL`: adiciona una lista a la lista de salida de tipo `galatea.glider.List`.

`utputsAdd0`: adiciona un `Object`, a la lista de salida `outputs`, de un agente.

`outputsAdd`: adiciona un `String`, a la lista de salida `outputs`, de un agente.

Estos métodos, así como todos los métodos implementados dentro de la `GInterface`, es necesario el nombre del agente.

3.1.7. Clases utilizadas para la creación de agentes

AgentComponentFactory

Esta clase implementa la interfaz `FIPAOSAgentComponentFactory` encargada de crear los componentes tales como: el manejo de las conversaciones, el `MTS` (Servicio de Transporte de Mensajes) y el manejo de tarea cuando es creado un agente.

MTs

Esta clase provee los mecanismo necesarios para que los agentes puedan interactuar dentro de los diferentes niveles de transporte de la plataforma.

3.1.8. Clases que manejan direcciones de agentes

InfoHost

Implementa la interface `java.io.Serializable`. La utiliza `SeudoAgent` para enviarle a `GalServer` un mensaje ACL solicitando la cantidad de computadores disponibles y el número de agentes que están corriendo en el sistema. Posee un conjunto de métodos **set** y **get** para tal fin.

ListHost

Permite almacenar los diferentes nombres de cada uno de los agentes con el nombre del computador donde fueron inicializados. `ListHost` utiliza la clase `java.util.TreeMap` para realizar el almacenamiento, búsqueda y eliminación de los agentes.

3.1.9. La clase de configuración

GalFipaos

Esta clase contiene los principales parámetros de configuración de la plataforma, así como también la creación de los diferentes agentes FIPAOS, dentro de esta clase se configuran el PlatformProfile, AgentProfile y el ACCProfile, en la sección A.3, realizamos una descripción más detallada de esta clase.

3.2. Estructura de Archivos

Los archivos que contiene la plataforma de distribución están contenidos dentro del directorio `galfipaos-0.0.1` y la distribución de sus archivos sigue de la siguiente manera (figura 3.6):

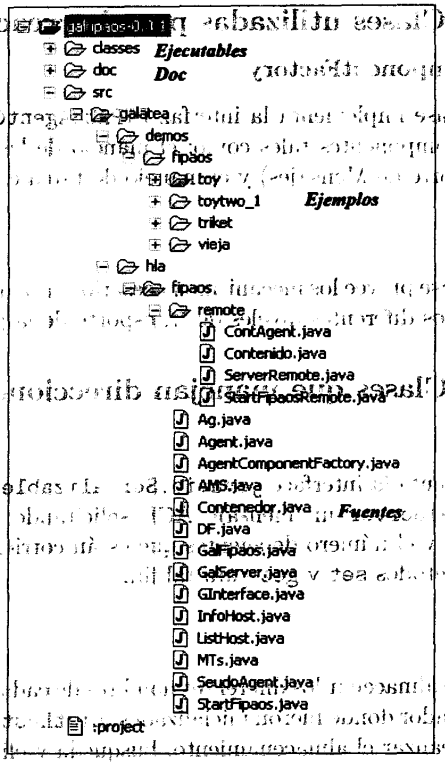

Figura 3.6: Estructura de Archivos de gFipaOS, en el se muestra, el directorio donde se encuentran las clases ejecutables, la documentación, ejemplos y los fuentes de la plataforma

- El subdirectorio `classes`, contiene las clases `class` de `gFipaos`. Debe ser adicionada a la variable de ambiente `CLASSPATH` por medio del jar o a través del directorio de clases.
- La documentación de las clases y los ejemplos realizados están en el subdirectorio `doc`. En él también se encuentra el archivo README.txt que explica los pasos de instalación y configuración de GALATEA y FIPAOS, y además contiene una guía rápida de iniciación de la plataforma.
- Los archivos que contienen el código java, comúnmente denominados archivos fuentes, están contenidos en los subdirectorios `src/galatea/hla/fipaos` y

src/galatea/hla/fipaos/remote. Dentro de **src/galatea** también se encuentra el subdirectorio **demos** donde están los diferentes ejemplos que explican el funcionamiento de la plataforma de una manera muy sencilla. Aquí se implementa los ejemplos de agentes contenidos como demos en GALATEA de manera distribuida, así como también otros ejemplos que puedan ser de interés y disfrute de los usuarios.

"Dar ejemplo no es la principal manera de influir sobre los demás; es la única manera"
Albert Einsten

Capítulo 4

Aprendamos por medio de ejemplos

En este capítulo nos concentraremos en explicar por medio de una serie de ejemplos muy sencillos que instrucciones son necesarias para que nuestros modelos funcionen de manera distribuida con FIPA-OS.

Empezaremos estudiando el primer ejemplo mostrado en el libro "Simulación multi-agente con GALATEA" sección 2.2 [33], ya que orienta al lector aprender acerca del lenguaje y del simulador GALATEA con agentes. Tomaremos algunas explicaciones de este ejemplo para fines didácticos. Invitamos al lector a leer y aprender un poco más de GALATEA con esta guía de estudio.

4.1. El ejemplo Toy

El primer ejemplo trata sobre el modelo de una reserva forestal sección 2.2.1 de [33], es un "modelo juguete" que ha sido construido como parte del proyecto "Biocomplexity: Integrating Models of Natural and Human Dynamics in forest Lanscapes Across Scales and Cultures", NFS GRANT CNH BCS-0216722, y donde GALATEA está siendo usada para construir modelos complejos de la dinámica de la reserva forestal de Caparo, ubicada en el Estado Barinas, al Sur-Occidente de Venezuela. Este modelo es presentado por sus autores como un juego de simulación, en el que el simulista debe tomar el lugar del ambiente. Los agentes están implementados

como una extensión de la clase `Ag` que pertenece al paquete `galatea.gloriosa.Ag`. Para nuestro caso los agentes serán implementados por la clase `Ag` que pertenecen al paquete `galatea.hla.fipaos`.

El resultado es un modelo en el que el simulista es invitado a llevar un registro manual de la evolución de una Reserva Forestal, mientras sobre ese ambiente actúa un conjunto de agentes artificiales que representan los actores humanos de la Reserva.

En la sección 2.2.2 de [33], se habla sobre algunos conceptos que le permiten al lector entender un poco del modelado multi-agente. Las cuatro clases que contiene el modelo, (figura 4.1) son:

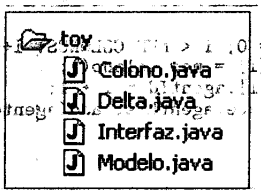

Figura 4.1: Clases del ejemplo Toy

1. `Colono`, define a los agentes de tipo colono que ocupan una Reserva. Varias instancias de esta clase hacen de este un modelo multi-agente.

2. `Delta`, implementa la función de transición del ambiente. En este caso, es un solo mecanismo de consulta e interacción con el simulista, quien estará "simulando", por su cuenta, la evolución del ambiente natural.

3. `Modelo`, contiene el programa principal y las variables globales del sistema.

4. `Interfaz`, es uno de los elementos más importantes del simulador multi-agente: contiene el conjunto de servicios que median entre los agentes y el ambiente cuando aquellos actúan y observan sobre éste.

Ahora veremos que cambios se deben realizar en estas cuatro clases para que el modelo funcione con FIPA-OS

Comencemos por la clase Delta, sección 2.2.3 de [33]:

En el apéndice A sección B.1 encontramos la clase original Delta (sección B.1.1 códigos 39, 40 y 41) y la nueva clase Delta (sección B.1.2 códigos 42, 43 y 44). Empecemos estudiando los códigos 39 y 42. En primer lugar importaremos la clase `GInterface` utilizando

```
import galatea.hla.fipaos.GInterface;
```

en vez de importarlo de la forma

```
import galatea.hla.*;
```

ya que la clase `GInterface` que pertenece al paquete `galatea.hla.fipaos.GInterface` posee todas las directivas necesarias para que funcione dentro de FIPAOS y se pueda inicializar de manera remota.

Code 18 Crea una instancia de un agente

```
for (int i = 0; i < NUM_COLONOS; i++) {
     agente[i] =new Colono();
     agente[i].agentId = i + 1;
     GInterface.agentList.add(agente[i]);
}
```

El fragmento del código 18 muestra la forma como crear los agentes de esta simulación de manera tradicional, asignando un identificador a cada uno de ellos y realizando el registro de cada uno en una base de datos central. Para FIPA-OS creamos un agente de la siguiente forma (código 19) donde se inicializa el agente

Code 19 Crea una instancia de un agente en la plataforma de distribución

```
for(int i = 0; i < NUM_COLONOS; i++){
    System.out.println("Inicializa el agente colono"+(i+1));
    GInterface.initAgent(
        "galatea.demos.fipaos.toy.Colono",
        "colono"+(i+1)
    );
}
```

con dos parámetros, el nombre completo del programa que incluye el paquete al que pertenece el agente `"galatea.demos.fipaos.toy.Colono"` y el nombre del agente `"colono+(i+1)"`. Este último no debe poseer espacios en blancos. Si se está trabajando en un único computador, automáticamente es inicializado el agente en el computador local; si hay varios computadores disponibles dentro de la arquitectura de distribución, se despliega por pantalla los computadores disponibles y el usuario, escoge en que computador desea inicializar el agente.

El nombre del agente colono1, colono2 ó colono3, pasan a formar parte de la identificación del agente (AID), junto con los roles y las direcciones de transporte que éste posee, tales como:

```
(agent-identifier
 :name colono@galatea
 :addresses (sequence
  fipaos-rmi://piache.gogigan.net.ve:4000/colono )
)
```

Por omisión es asignado el identificador de GALATEA que es de tipo `int`, dependiendo de la cantidad de agentes inicializados en el sistema, si se desease asignarle un identificador particular, se puede utilizar el método de `GInterface setAgentID`, como en el ejemplo siguiente:

```
GInterface.setAgentID("colono1", 1)
```

Internamente el `ServerRemote` del computador local o el computador remoto, inicializa el agente, lo asigna dentro de una base de datos que le permite estar registrada para el simulador y también se registra dentro del sistema manejador de agente.

El código 20 es un fragmento perteneciente al código 40 en el se actualiza el reloj

Code 20 Interacción de un agente en GALATEA

```
for (int l = 0; l < NUM_COLONOS; l++) {

  // Actualiza el reloj de cada agente
  agente[l].clock = Glider.getTime();

  // les transmite lo que deben ver
  actualizarsensores(agente[l]);

  // activa el razonado de cada agente
  agente[l].cycle();
}
```

del agente, se transmite lo que debe ver a través de sus sensores en ese instante de tiempo y se le pide que piense y decida que hacer. Para adaptarlo a la nueva plataforma este fragmento fue modificado de la forma como se muestra en el fragmento 21 del código 43

En este caso se actualiza el reloj del agente por medio del método `updateClockAgent` en el que se le pasa el nombre del agente y el tiempo a actualizar. El método `actualizarsensores(Agente[l])` que es un método propio de la clase `Delta.java` fue cambiado por el método `actualizarsensores(String name_agent)` ya que los agentes intervienen en el modelo es por medio de `GInterface`, entonces internamente dentro del método `actualizarsensores`, se actualizan los sensores del agente de la siguiente forma

```
GInterface.inputsAdd(name_agent,"No establecido");
```

Code 21 Interacción de un agente en GALATEA de manera distribuida

```
for (int l = 0; l < NUM_COLONOS; l++) {
   GInterface.updateClockAgent("colono"+(l+1),Glider.getTime());
   actualizarsensores("colono"+(l+1));
   GInterface.cycle("colono"+(l+1));
}
```

como se muestra en el código 44 en vez de realizarlo

```
agente.inputs.add("No establecido");
```

como se muestra en el código 41.

Para activar el motor de razonamiento del agente por medio de la sentencia `agente[1].cycle()` es activado por la `GInterface` de la siguiente forma:

```
GInterface.cycle("colono"+(l+1))
```

y también se puede llamar por medio de los métodos de invocación dinámica que ofrece `GInterface` de la siguiente forma:

```
GInteface.invoke1("colono"+(l+1), "cycle");
```

El fragmento de código que permite recoger y procesar las influencias de los agentes sigue igual:

```
GInterface.gatherInfluences_test();
GInterface.process_test();
```

El modelo de los agentes Colono, sección 2.2.4 [33]:

Dentro del modelo del agente a construir debemos realizar dos cambios importantes:

1. Debemos importar la clase `Ag` del paquete `galatea.hla.fipaos` y no del paquete `galatea.glorias.Ag`
2. Su constructor código 22 cambia de la forma código 23 donde `name`, es el nombre del agente, y forma parte de su AID. Lo recomendable es que los agentes de una misma plataforma de agentes no tengan nombres iguales, porque si poseen los mismo roles y la misma dirección habrían agentes con el mismo AID, el resto del código del agente permanece igual.

Code 22 Constructor original del agente Colono

```
public Colono() {
    super(6,"colono");
    Agent.population++;
    this.setAgentId(Agent.population);
    init();
}
```

Code 23 El nuevo constructor de Colono

```
public Colono( String name) {
    super(6,name);
    Agent.population++;
    this.setAgentId(Agent.population);
    init();
}
```

La Interfaz y el programa principal sección 2.2.5 [33]:

La Interfaz se debe importar la clase `Agent` del paquete `galatea.hla.fipaos.Agent` en vez de `galatea.hla.Agent`.

El programa principal cambia únicamente al importar `GInterface` del paquete `galatea.hla.GInterface` a `galatea.hla.fipaos.GInterface`.

4.2. El ejemplo Toytwo1

El segundo ejemplo trata nuevamente sobre el modelo de la reserva forestal, y forma parte de los demos con agentes presentados dentro de GALATEA, pero a diferencia del ejemplo anterior donde el simulista tomaba el lugar del ambiente, ahora el ambiente es representado por medio de un autómata celular, donde se consideran once estados de uso de la tierra. El modelo es representado nuevamente por cuatro clases importantes figura 4.2.

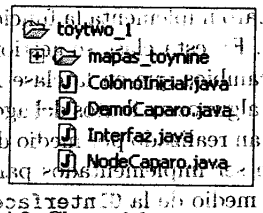

Figura 4.2: Clases del ejemplo ToyTwo1

La clase `ColonoInicial` define a los agentes de tipo colono que ocupan la reserva forestal. Varias instancia de esta clase hacen de éste un modelo multi-agente. Para que `ColonoInicial` funcione dentro de FIPAOS se deben realizar los mismos cambios que en la clase Colono del ejemplo Toy, se debe importar la clase `Ag` del paquete `galatea.hla.fipaos`, y se debe realizar los arreglos en el constructor como se muestra en el código 23. Dentro de la clase `NodeCaparo` se consultan parámetros del agente `ColonoInicial` y se deben realizar los métodos `set` y `get`, que le permitan poder realizar las consultas por medio de `GInterface` de manera dinámica como se muestra en el código 24.

Code 24 Implementación de métodos set y get para un agente

```
public int getTr(){
    return tr;
 }
 public void setTr(int _tr){
    tr = _tr;
 }
 public int getNumCeldas(){
    return num_celdas;
 }
 public void setNumCeldas(int n){
  num_celdas = n;
 }
 public int getMaxNumCeldas(){
    return this.MAX_NUM_CELDAS;
 }
 public int getPropiedad(int arg0, int arg1){
    return this.propiedad[arg0][arg1];
 }
 public int getX(){
    return this.x;
 }
 public int getY(){
    return this.y;
 }
```

La clase NodeCaparo implementa la función de transición del ambiente, por medio del autómata celular. En esta clase se gestiona todo lo referente al agente, se deben realizar los mismos cambios que en la clase Delta del ejemplo anterior. `NodeCaparo` realiza la consulta de algunos atributos del agente `ColonoInicial`, ahora las consultas a dichos atributos serán realizado por medio de la `GInterface`, utilizando los métodos `set` y `get` que deben ser implementados para todos los atributos del agente que se deseen consultar por medio de la `GInterface`.

Por ejemplo para obtener el número de celdas del agente, en el ejemplo original se realiza de la siguiente forma:

```
agente.num_celdas
```

Ahora dentro de la nueva implementación se utiliza:

```
GInterface.invoke2(name_agent,"getNumCeldas");
```

Para obtener el valor de determinado atributo donde se utilicen parámetros como el siguiente:

```
int valor;
for(int i = 0; i < agente.num_celdas; i++){
  valor = agente.propiedad[i][0];
  .....
}
```

Ahora creamos un vector de Objetos, con los argumentos necesarios para invocar al método por medio de la `GInterface` de la siguiente forma (código 25):

Code 25 Ejemplo de consultas de atributos del agente, utilizando `GInterface`

```
int valor;
n = GInterface.invoke2(name_agent,"getNumCeldas");
for(int i=0; i < agente.num_celdas; i++){
  Object[] _o = { new Integer(i), new Integer(0)};
  valor = GInterface.invoke2(name_agent,"getPropiedad", _o);
  .....
}
```

El programa principal y la Interfaz

La clase DemoCaparo contiene el programa principal y las variables globales del sistema, el único cambio que se realiza es el uso de la clase `GInterface` del paquete `galatea.hla.fipaos.GInterface`.

Mientras que la clase Interface nuevamente exportara la clase `Agent` del paquete `galatea.hla.fipao.Agent` como en el ejemplo anterior.

4.3. El ejemplo Tic Tac Toe o Vieja

Nuestro tercer ejemplo trata sobre el juego Tic Tac Toe o la vieja, como tradicionalmente se le conoce. Utilizaremos como modelo de referencia el juego realizado en [18], donde a través del diseño de una plantilla de simulación cliente-servidor para

la implementación de juegos en red, se realizó una primera aproximación a la plataforma de distribución de GALATEA utilizando el paradigma cliente-servidor por medio de sockets.

El juego consiste en ganarle al contrincante o a los contrincantes, que bien puede ser la computadora (un agente pasivo) o un jugador externo (un humano, un agente activo), por medio de estrategias que lo lleven a realizar jugadas ganadoras en cualquiera de las nueve posibilidades, que existen en un tablero en forma de matriz 3×3, realizando una línea vertical, horizontal o diagonal.

Su dinámica es:

- El jugador debe escoger el tipo de ficha con la que desea jugar (X o O).
- Comienza un jugador colocando en cualquiera de las casillas vacías del tablero su ficha.
- El jugador espera su turno, el cual llega en el momento en que el contrincante asignó una ficha en cualquiera de las casillas vacias.
- El juego termina cuando uno de los dos jugadores logra colocar tres de sus fichas en las posiciones establecidas para ganar, es decir que forme una línea diagonal, vertical u horizontal, en caso contrario, terminará el juego cuando todas las posiciones del tablero estén ocupadas y ninguno de los dos jugadores haya ganado.

El juego está diseñado para que puedan participar dos agentes, el primero un agente pasivo, que en este caso es la computadora y el segundo un jugador activo, es decir un agente humano. El juego esta compuesto por un programa GALATEA que internamente posee dos nodos autónomos, el primer nodo llamado inteligencia es el cerebro del juego, es diseñado por medio de una clase que contiene las reglas del juego y programa las jugadas que realiza la computadora y el segundo nodo llamado evento, permite actualizar las diferentes jugadas que realiza el agente humano. Además el juego tiene la clase Servidor que es la clase principal del modelo, posee las variables globales del modelo y construye la red GALATEA.

Existe también la clase Vieja, que es la interfaz por medio del cual el jugador externo o agente humano puede realizar las diferentes jugadas.

El nuevo diseño realizado (figura 4.3 y 4.4) siguiendo con la plantilla modelo juego para GALATEA en [18], el nodo Inteligencia pasa a ser un agente que extiende de la clase `Agent` del paquete `galatea.hla.fipaos.Agent` (código 50), con las reglas del juego y que puede funcionar de manera independiente. Su constructor recibe como parámetro el nombre del agente que permite inicializar la clase superior de `Agent` (segundo constructor mostrado en el código 6).

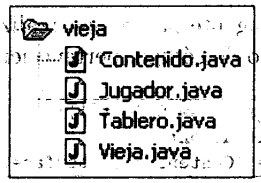

Figura 4.3: Estructura de clases del juego la vieja

```
public Jugador(String name) {
    super(name);
    .....
}
```

El programa principal (código 47) contiene un nodo autónomo y controla las diferentes jugadas del agente pasivo y el agente humano realizadas por medio de la clase tablero.

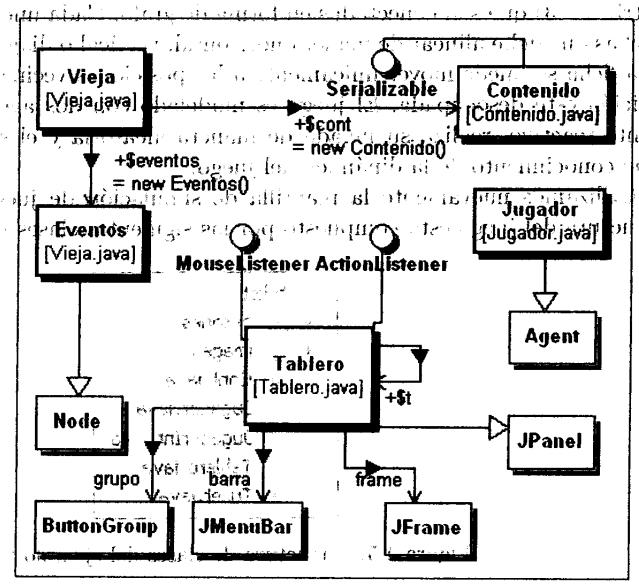

Figura 4.4: Diagrama de clases del ejemplo la vieja

La clase evento (código 48) es la encargada de interactuar con el agente y al igual que en el ejemplo anterior `GInterface` inicializa el agente.

Las jugadas del agente pasivo son realizadas por medio de la `GInterface`, como se ve en el fragmento de código, perteneciente al método **ganar** de la clase evento

```
Object[] args = new Object[1];
args[0] = Vieja.cont;
Vieja.cont = (Contenido)GInterface.invoke("jugador","toPlay",args);
```

que permite crear un vector de tipo `Object`, con el objeto que describe todas las ontologías de las jugadas realizadas, y que son enviadas al agente por medio de la `GInterface` que invoca el método `toPaly` del agente, para que observe las jugadas realizadas, se le pide que piense y decida cual es su próxima jugada dentro del tablero.

4.4. El ejemplo Triket

Nuestro último ejemplo trata sobre el juego triket o como muchos lo conocen tres en raya. Se juega entre dos personas en un tablero que posee nueve posiciones (una matriz 3×3) que son conectadas en forma de grafo. Cada uno de los jugadores posee 3 fichas que debe alinear de manera horizontal, vertical o diagonal, para poder ganar, cada ficha se puede mover únicamente a las posiciones vecinas, siempre y cuando la posición esté desocupada. El juego es modelado con dos agentes, el primero es un agente reactivo, realiza su jugada de manera aleatoria y el segundo un agente que tiene conocimiento de la dinámica del juego.

Utilizamos nuevamente la plantilla de simulación de juegos en red de [18]. La estructura del juego está compuesto por las siguientes clases (figura 4.5):

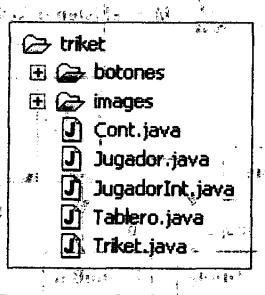

Figura 4.5: Estructura de clases del ejemplo triket

Las clases `JugadorInt` y `Jugador` extienden de la clase `Agent` del paquete `galatea.hla.fipaos`.

El programa principal está en la clase Triket (código 51) contiene las variables globales del sistema, inicializan los dos agentes y almacena la dirección de los host donde fueron inicializados los agentes.

```
for(int i = 0; i<2; i++)
  GInterface.initAgent(
     "galatea.demos.fipaos.triket.JugadorInt",
     "jugador"+(i+1),
  );
h1 = GalServer.ag_name_addresses.getHost("jugador1");
h2 = GalServer.ag_name_addresses.getHost("jugador2");
```

La clase `ActualizaTablero` que extiende la clase node (código 53), le envía por medio de un objeto el estado actual del tablero, para que cada uno de los agentes pueda procesar las diferentes jugadas realizadas y realizar la próxima jugada dentro del tablero.

```
Object[] args = new Object[1];
.....
  Triket.cont.setNombre("jugador1");
  Triket.cont.setTypeF(Triket.n);
  Triket.cont.setOrigenes(Triket.fichas1);
  Triket.cont.setTablero(Triket.tablero.p);
  args[0] =Triket.cont;
  Triket.cont = (Cont) GInterface.invoke("jugador1","toPlay",args);
```

"Nuestra recompensa se encuentra en el esfuerzo
y no en el resultado.
Un esfuerzo total
es una victoria completa"
Gandhi

Conclusiones

En este documento hemos presentado el diseño de la plataforma distribuible para el simulador multi-agente GALATEA utilizando el estándar de comunicación FIPA, y su implementación por medio del sistema manejador de agentes FIPAOS.

En el diseño de la plataforma de distribución multi-agente creó un módulo entre los agentes y `GInterface` para que el modelo de un sistema particular, con entidades que son representadas por medio de agentes, pueda ejecutarse en procesadores independientes. Para la interacción se utilizan los actos comunicativos básicos ofrecidos por FIPA `inform`, `request`, `informif`, `accept`, `suscribe`, entre otros; para crear lenguaje de comunicación propio dentro de la plataforma de distribución.

La plataforma de distribución multi-agente de GALATEA se diseñó por medio de una estructura centralizada, ya que dentro del simulador, `GInterface` es la única responsable de gestionar todas las tareas que realizan los agentes del modelo de simulación. Nosotros nos valimos de esto y realizamos dentro de la plataforma de distribución multi-agentes un diseño centralizado, que por medio de agentes gestionadores, permite administrar y controlar las diferentes actividades que pueden ser solicitadas por parte del modelo a los agentes durante la simulación de la dinámica de un determinado sistema. Los agentes gestionadores, como su nombre lo indica, se encargaran de gestionar dichas actividades dentro de la plataforma de distribución para que los agentes del modelo puedan interactuar de manera remota con el resto de los componentes de la simulación.

FIPAOS se encarga por su parte de dar el soporte para alojar a los agentes en las diferentes arquitectura de computadores que forman parte de la plataforma de distribución del sistema multi-agente de GALATEA. También le ofrece los estándares de comunicación FIPA para que los agentes gestionadores y los agentes del modelo puedan definir su propio protocolo de comunicación. De esta forma `GInterface` y los agentes interactúan entre sí dentro de la plataforma de distribución de agentes.

Las solicitudes realizadas por `GInterface` a los agentes en el modelo de simulación son clasificadas por los agentes gestionadores en diferentes performativas para que sean interpretadas por los agentes del modelo dentro de la plataforma de distribución. Los agentes obtienen la información de los modelos de simulación, razonan y finalmente producen un conjunto de influencias que son enviadas nuevamente a los agentes gestionadores por medio de mensajes ACL (*Agent Communication Language*). Los agentes gestionadores procesan esta información y la envían a `GInterface` para que sean entregadas al modelo de simulación.

La plataforma de distribución multi-agente puede ser activada por los diferentes eventos dentro del modelo que involucran a los agentes, tales como iniciación o consulta de cada uno de ellos. También se activa cuando se agrega un nuevo computador a la arquitectura de computadores de la plataforma de distribución multi-agente.

Los agentes gestionadores son `SeudoAgent`, `GalServer` y los `ServerRemote`. `SeudoAgent` es el encargado de interactuar con la clase `GInterface` dentro de la plataforma de distribución y este a su vez coordina con los diferentes agentes gestionadores de la plataforma de distribución por medio de divisiones/de tareas para atender las solicitudes de consulta o activación de agentes hechas por `GInterface`. `GalServer` se encarga de llevar el registro de los diferentes computadores que forma parte de la arquitectura distribuida, así como también lleva el registro de los agentes GALATEA que se han inicializados en cada uno de ellos. `ServerRemote` se encarga de inicializar los diferentes agentes GALATEA dentro de la plataforma de distribución y recoger las influencias que ellos generan para que `GInterface` las procese ante el modelo de simulación.

`GInterface` dentro de la plataforma de distribución de agentes posee toda la semántica y sintaxis para que los modelos de simulación dentro de GALATEA interactúen con los agentes de manera remota.

El modelo conoce de los agentes y, por que no decirlo de la plataforma de distribución, por medio de la sintaxis que ofrece `GInterface`. Es a través de `GInterface` que el modelo invoca de manera remota, cada uno de los agentes. Dentro de la sección de ejemplos mostramos algunos de los casos donde el modelo interactúa con los agentes dentro de la plataforma de distribución.

Se aprovechó la arquitectura basada en componentes que posee FIPAOS para adicionar las directrices que le permiten a los agentes GALATEA alojarse en las diferentes plataformas de agentes que están en las arquitecturas computacionales de la plataforma de distribución de agentes y a su vez adquirir actos comunicativos para que puedan atender las solicitudes del modelo de manera remota.

También describimos en este documento la configuración necesaria de FIPAOS para que el diseño de la plataforma multi-agente GALATEA pueda funcionar sobre esta. Principalmente se destaca los perfiles más importantes que configuran a la plataforma, tales como los perfiles que adquieren los agentes para ser creados, los

perfiles de cada una de las plataformas de agentes que se pueden crear y el perfil del canal de comunicación entre los agentes que permite configurar los prótocolos de comunicación para que dos o más plataformas de agentes puedan interactuar entre sí.

Creemos que uno de los trabajos futuros más relevantes es crear agentes GALATEA que puedan utilizar y aprovechar los estándares de FIPA. De esta forma funcionaría no únicamente dentro de FIPAOS, sino que además pueda funcionar dentro de los diferentes sistemas manejadores de agentes que soportan el protocolo FIPA, ganando con esto, el aprovechamiento de los diferentes servicios que pueden ofrecer cada uno de ellos, teniendo una mayor interoperabilidad entre comunidades de agentes que pueden estar funcionando en múltiples plataforma de agentes. Esto podría ser aprovechado por simulaciones con agentes de mayor grado de complejidad y además, podríamos aprovechar mejor los recurso computacionales. Para esto nos podemos valer de la arquitectura basada en componentes que ofrece FIPAOS que consideramos muy bien definida para adicionar los componentes que puedan ser necesarios para que funcione con otros manejadores de agentes que están en el mercado y que soportan el protocolo FIPA.

La fuerza de uno es alentadora,
la fuerza de varios es invencible"
Anónimo

Referencias

[1] Uzcátegui, M. Y.*Diseño de la plataforma de simulación de GALATEA.*, Tesis de Maestría, Maestría en Computación, Universidad de Los Andes. Mérida. Venezuela. (2002)

[2] *GALATEA, Plataforma de Simulación de Sistemas Multiagentes.*, http://cesimo.ing.ula.ve/

[3] *Foundation for Intelligent Physical Agents*, http://www.fipa.org

[4] *FIPA-OS*, http://fipa-os.sourceforge.net/index.htm

[5] *ZEUS*, http://www.labs.bt.com/projects/agents/zeus/index.htm

[6] *JADE*, http://sharon.cselt.it/projects/jade/

[7] *JACK*, http://www.agent-software.co.uk

[8] Zeigler, B. P. (1976). *Theory of modelling and simulation* Interscience. Jhon Wiley & Sons, New York.

[9] Zeigler, B. P. (1990). *Object-oriented simulation with hierarchical, Modular models (Intelligent agents and endomorphic systems).* Academic Press, Inc (Harcourt Brace Jovanovich, Publishers), Boston-Sydney.

[10] Zeigler, B. P., Praehofer, H., y Kim, T. G. (2000). *Theory of Modelling and Simulation.* Academic Press, second edicion.

[11] Davila, J. A. (1997). *Agents in Logic Programming.* Tesis PhD, Imperial College of Science, Technology and Medicine., London, UK.

[12] Dávila, J. A. (1999). *"Openlog: A logic programming language based on abduction,"* in PPDP99, ser. Lecture Notes in Computer Science. 1702. Paris, France: Springer, 1999. [Online]. Available: http://citeseer.nj.nec.com/64163.html

[13] Dávila, J. A. (2003). *"Actilog: An agent activation language,"* ser. LNCS, New Orleans, USA, 2003.

[14] Domingo, C. (1998). *GLIDER, a network oriented simulation language for continuous and discrete event simulation.* En International Conference on Mathematical Models, Madras, India.

[15] Domingo, C. y Hernández, M. (1985). *Ideas básicas del lenguaje GLIDER.* Reporte, Instituto de Estadística Aplicada en Computación. Universidad de Los Andes. Mérida. Venezuela.

[16] Domingo, C., Tonella, G., y Sananes, M. (1996). *GLIDER Reference Manual.* CESIMO-IEAC. Universidad de Los Andes, Mérida, Venezuela, 1 edición. CESIMO IT-9608.

[17] DoD DMSO, (1995). *"High level architecture (HLA),"* Departament of Defense. Defense Modeling and Simulation Office, Tech. Rep., 1995. [Online]. Available: http://www.dmso.mil

[18] Dávila, M. A. (2003). *Una plataforma cliente-servidor para simulación con modelos juegos.* Tesis de Pregrado. Escuela de Ingeniería de Sistemas, Facultad de Ingeniería. Universidad de Los Andes.

[19] Dávila, J. A. y Tucci K. A. (2002). *Towards a logic-based, multi-agent simulation theory.* AMSE Special Issue 2000. Association for the advancement of modelling & Simulation techniques in Enterprises, pp. 37-51, 2002, lion, France.

[20] J. A. Dávila and M. Uzcátegui, (2002) *"Galatea: A multi-agent simulation platform"*, AMSE Special Issue 2000. Association for the advancement of Modelling & Simulation techniques in Enterprises, pp. 5267, 2002, lion, France.

[21] J. A. Dávila and M. Uzcátegui, (2004). *"Gloria: An agents executable specification,"*, Collegium Logicum. Kurt G'odel Society, vol. VIII, pp. 3544, 2004, vien, Austria.

[22] Mangina, E. (June 2002). *Review of Software Products for Multi-Agent Systems* Applied Intelligence (UK) Ltd for AgentLink (www.AgentLink.org)

[23] Luck, M., McBurney, P., y Preist C. *Agent Tecnology: Enabling Next Generation Computing* A Roadmap for Agent Based Computing. AgentLink (www.AgentLink.org)

[24] Nortel Networks Corporation (1999 - 2000) *FIPA-OS Developers Guide.* 8200 Dixie Road, Suite 100, Brampton, Ontario, Canada L6R 5P6. All rights reserved. Open Source Copyright Notice and License: FIPA-OS

[25] Amor, M., Fuentes, L., y Pinto, M. *Interoperabilidad entre Plataformas de Agentes FIPA: Una aproximación Basada en Componentes* Dept. Lenguajes y C. C., Universidad de Málaga. Málaga, Spain.

[26] *Foundation for Intelligent Physical Agents, FIPA Agent Management Specification. 2000* http://www.fipa.org/specs/fipa00023/

[27] *Foundation for Intelligent Physical Agents, FIPA Agent Communication Language. 2000* http://www.fipa.org/repository/aclspecs.html

[28] Stuart J. R. y y Norvig P. (1.996). *Inteligencia Artificial: un enfoque moderno.* Prentice Hall Hispanoamericana, S.A. MÉXICO.

[29] Zavala, R.L., Carreño, A. A., Lematre, C. *CATNAgentToolkit: Una plataforma para el diseño y ejecución de sistemas multiagentes.* Laboratorio Nacional de Informática Avanzada Rébsamen 80 A.P. 696 C.P. 91090 Xalapa, Veracruz, México.

[30] Quintero, A., Rueda R., Sandra, Ucrós C., María, *AGENTES Y SISTEMAS MULTIAGENTE : INTEGRACIÓN DE CONCEPTOS BÁSICOS.* Grupo de Investigación HIDRA Departamento de Ingeniería de Sistemas y Computación Universidad de los Andes, Colombia.

[31] Lesser, Victor. (1.991). *A Retrospective View of FA/C Distributed Problem Solving. IEEE Transactions on Systems, Man and Cybernetics.* Vol. 21, No. 6.

[32] *Simulación en Economía*,http://www.economiaindustrial/simulacion.htm. visitada (05/02/01).

[33] Dávila, Jacinto A. Tucci, Kay A. y Uzcátegui, M. Y. (2.004) *SIMULACIÓN MULTI-AGENTE CON GALATEA*, (Versión 14 de Mayo) Universidad de Los Andes. Mérida. Venezuela.

[34] Sánchez, M., García, J. M., Gómez, A. F. y Martínez H. (1.999) *Generación de simuladores eficientes para procesos complejos basados en Arquitecturas Distribuidas.*, X Jornadas de paralelismo, La Manga Del Mar Menor Murcia, Septiembre 1999.

[35] A. Ames, D. Nadeau and J. Moreland. (1.997) *The VRML 2.0 sourcebook. John Wiley & Sons.* New York.

[36] IEEE Standard Board (1.995) *Distributed Interactive Simulation/Aplication protocol.* IEEE Standard 1278.1: IEEE Inc.

[37] Locke, J. (1.995) *An Introduction to the Internet Networking Environment and SIMNET/DIS.* Naval Postgraduated School. Monterey, CA, USA, 1995.

[38] Rational (1.991) *Rational Rose* http://www.rational.com

[39] Cantv (2.004) *Un año para aprender.* Agenda 2.005. Editado por la Gerencia General de Mercadeo de Cantv. Caracas, Diciembre de 2.004

[40] *Los MDT en los estudios de medio físico.* http://www.etsimo.uniovi.es/~feli/pdf/ITGE_150a.pdf

[41] Emili Miedes (2.003) *Java La Plataforma y el Lenguaje,* emiedes@iti.upv.es; http://www.javahispano.com

[42] Magdiel Ablan (2.000) *Guía de estudios del curso de Pregrado en Simulación.* Centro de simulación y Modelos, Universidad de los Andes. Mérida Venezuela.

[43] O. Biham, A. A. Middleton, and D. Levine, (1992) "*Self-organization and dynamical transition in traffic-flow models,*" Phys. Rev. A, no. 46, 1992.

[44] K. Nagel and J. H. Herrmann, (1993) "*Deterministic models for traffic jams,*" Physica A, no. 199, 1993.

[45] D. E. Wolf, M. Schreckenberg, and A. e. Bachem, (1996) "*Traffic and Granular Flow. World Scientific,*" 1996.

[46] D. Helbing, I. Farkas, and T. Vicsek, (2000) "*Simulating dynamical features of escape panic,*" Nature, no. 407, pp. 487490, 2000.

[47] K. Laffaille, (2005) "*Gspaces, meta-modelo para simular desalojos de espacios urbanos y arquitectónicos basado en GALATEA,*" Masters thesis, Maestría en Modelado y Simulación de Sistemas, Universidad de Los Andes. Mérida. Venezuela, 2005, tutor: Tucci, Kay.

[48] K. Laffaille, (2005) "*The first clima contest,*" http://clima.deis.unibo.it/contest.html, 2005.