Versión
14 de mayo de 2004

SIMULACIÓN MULTI-AGENTE CON GALATEA

Jacinto A. Dávila Q.
Kay A. Tucci K.
Mayerlin Y. Uzcátegui S.

UNIVERSIDAD DE LOS ANDES
MÉRIDA, VENEZUELA
Mérida, 2004

...
—M.Y.U.S

...
—K.A.T.K

Por las razones
para explorar por
nuevos caminos
—J.A.D.Q.

Índice General

Índice General vi

Índice de Figuras vii

Índice de Ejemplos x

Licensing xii

Licencia de uso xiii

Resumen xiv

1 De la mano con Java 1
- 1.1 Java desde Cero . . . 2
  - 1.1.1 Cómo instalar Java . . . 2
  - 1.1.2 Cómo instalar Galatea . . . 2
  - 1.1.3 Hola mundo empaquetado . . . 3
  - 1.1.4 Inducción rápida a la orientación por objetos . . . 4
  - 1.1.5 Qué es un objeto (de software) . . . 5
  - 1.1.6 Qué es una clase . . . 5
  - 1.1.7 Qué es una subclase . . . 5
  - 1.1.8 Qué es una superclase . . . 6
  - 1.1.9 Qué es poliformismo . . . 7
  - 1.1.10 Qué es un agente . . . 8
  - 1.1.11 Qué relación hay entre un objeto y un agente . . . 9
  - 1.1.12 Por qué objetos . . . 9
- 1.2 Java en 24 horas . . . 10
  - 1.2.1 Java de bolsillo . . . 10
  - 1.2.2 Los paquetes Java . . . 12
  - 1.2.3 Un applet . . . 14

1.2.4 Anatomía de un applet . . . 14
1.2.5 Parametrizando un applet . . . 15
1.2.6 Objetos `URL` . . . 16
1.2.7 Gráficos: colores . . . 16
1.2.8 Ejecutando el applet . . . 17
1.2.9 Capturando una imagen . . . 17
1.2.10 Eventos . . . 18
1.2.11 paint(): Pintando el applet . . . 20
1.2.12 El AppletMonteCarlo completo . . . 20

2 Directo al grano **22**
2.1 El primer modelo computacional . . . 22
2.1.1 Nodos: componentes del sistema a simular . . . 23
2.1.2 El esqueleto de un modelo Galatea en Java . . . 25
2.1.3 El método principal del simulador . . . 27
2.1.4 Cómo simular con Galatea, primera aproximación . . . 29
2.2 El primer modelo computacional multi-agente . . . 29
2.2.1 Un problema de sistemas multi-agentes . . . 31
2.2.2 Conceptos básicos de modelado multi-agente . . . 32
2.2.3 El modelo del ambiente . . . 33
2.2.4 El modelo de cada agente . . . 39
2.2.5 La interfaz Galatea: primera aproximación . . . 42
2.2.6 El programa principal del simulador multi-agente . . . 44
2.2.7 Cómo simular con Galatea. Un juego multi-agente . . . 44
2.3 Asuntos pendientes . . . 45

3 Teoría de simulación de sistemas multi-agentes **46**
3.1 Modelado y Simulación . . . 46
3.1.1 Formalismos para especificación de sistemas . . . 49
3.1.2 Formalismo DEVS . . . 53
3.1.3 Enfoques para Simulación . . . 59
3.1.4 Marco de Referencia HLA . . . 62
3.1.5 Simulación Orientada a Agentes . . . 66
3.2 Una teoría de simulación de sistemas multi-agente . . . 71
3.2.1 Razones para una nueva teoría . . . 72
3.2.2 Una teoría de influencias y reacciones . . . 72
3.2.3 Una jerarquía de arquitecturas de agente . . . 74
3.2.4 Un agente reactivo y racional . . . 77
3.2.5 El sistema multi-agente con ese agente racional . . . 80

4 Tributo al ancestro: GLIDER en GALATEA **82**
4.1 Plataforma de simulación GLIDER 83
4.1.1 Lenguaje de simulación GLIDER 83
4.1.2 El compilador GLIDER 86
4.1.3 Entorno de Desarrollo GLIDER 90
4.2 Aspectos de Diseño 90
4.2.1 Listas 92
4.2.2 Componentes básicos del simulador 97
4.2.3 Otros Componentes 102
4.2.4 Comportamiento del simulador 103
4.3 Aspectos de implementación 105
4.3.1 Pruebas del prototipo 109
4.4 Modeloteca 110
4.4.1 Sistema simple de tres taquillas 110
4.4.2 Sistema de ferrocarril 114
4.4.3 Sistema de supermercado 117
4.5 Simulación y Resultados 121
4.5.1 Estadísticas sobre los nodos 123
4.5.2 Traza de la simulación 124

5 Simulación de sistemas multi-agentes con ejemplos **140**
5.1 Interfaz 143
5.2 Simulador 144
5.2.1 Algoritmo general para el simulador 144
5.2.2 Comportamiento del simulador 146
5.3 Agentes del tipo reactivo-racional 150
5.3.1 Detalles del agente 150
5.3.2 Detalles del Sistema Multi-Agentes 157
5.3.3 Implementación del agente 160
5.4 Familia de Lenguajes 162
5.4.1 Estructura de un programa GALATEA 163
5.4.2 Semántica operacional 164
5.5 Modeloteca 168
5.5.1 Sistema simple de tres taquillas con agentes 168
5.5.2 Sistema de supermercado con agentes 171
5.5.3 Sistema elevador con agentes 173
5.6 Detalles de Implementación 177
5.6.1 Requerimientos generales 177
5.6.2 Ambiente de desarrollo 178
5.6.3 Estructura de archivos 179

6 El Compilador Galatea **182**

7 Un ambiente de desarrollo para simulación con GALATEA **183**

Referencias **184**

A Applet Montecarlo **190**

B Clases del Simulador GALATEA **191**
B.1 Clase Glider 192
B.2 Clase Element 193
B.3 Clase List 194
B.4 Clase Event 195
B.5 Clase LEvents 197
B.6 Clase Node 198
B.7 Clase HNode 199
B.8 Clase LNodes 200
B.9 Clase Field 200
B.10 Clase LFields 201
B.11 Clase Message 201
B.12 Clase HMess 202
B.13 Clase LMess 203
B.14 Clase LLMess 204
B.15 Clase GRnd 205
B.16 Clase GStr 206

Glosario **212**

Índice de Materias **215**

Índice de Figuras

1 De la mano con Java **1**

2 Directo al grano **22**
2.1 Salida del modelo en un terminal 30
2.2 Símbolos de estado en un ambiente natural 34

3 Teoría de simulación de sistemas multi-agentes **46**
3.1 Elementos básicos del modelado y simulación de sistemas 47
3.2 Concepto básico de sistemas 49
3.3 Clasificación de los modelos 50
3.4 Formalismo de Ecuaciones Diferenciales 51
3.5 Formalismo de Ecuaciones en Diferencias 51
3.6 Formalismo de Eventos Discretos 52
3.7 Jerarquía de Especificación de Sistemas 54
3.8 Protocolo DEVS básico 56
3.9 Protocolo DEVS acoplado 57
3.10 Infraestructura para modelado y simulación 58
3.11 Arquitectura para modelado y simulación 58
3.12 Esquema funcional de una federación HLA 63
3.13 Jerarquía de Genesereth y Nilsson 75
3.14 Jerarquía de Ferber y Müller extendida por Dávila y Tucci 76
3.15 Evolución del estado global del sistema 77

4 Tributo al ancestro: GLIDER en GALATEA **82**
4.1 Estructura de un programa GLIDER 85
4.2 Componentes básicos del simulador 91
4.3 Relación entre los componentes del simulador 92
4.4 Diagrama básico de lista 93

4.5 Métodos básicos de lista 94
4.6 Métodos para consultar en lista 128
4.7 Métodos para avanzar en lista 129
4.8 Métodos para agregar en lista 129
4.9 Métodos para eliminar en lista 130
4.10 Métodos para modificar en lista 130
4.11 Objetos asociados al componente evento 131
4.12 Objetos asociados al componente nodo 131
4.13 Objetos asociados al componente mensaje 131
4.14 Diagrama de clases del simulador 132
4.15 Diagrama de secuencia para el simulador 133
4.16 Diagrama de colaboración para el simulador 134
4.17 Jerarquía de clases del simulador 135
4.18 Estructura de archivos del simulador 136
4.19 Modelo del sistema simple de tres taquillas 137
4.20 Modelo del sistema de ferrocarril 137
4.21 Modelo del sistema de Supermercado 138
4.22 Traza para el sistema simple de tres taquillas 139

5 Simulación de sistemas multi-agentes con ejemplos 140
5.1 Arquitectura de la plataforma de simulación GALATEA 141
5.2 Diagrama de secuencia para GALATEA 146
5.3 Diagrama de colaboración para GALATEA 148
5.4 Componentes del agente reactivo-racional 150
5.5 GLORIA: Especificación Lógica de un agente 155
5.6 Ejemplo de sistema multi-agentes 158
5.7 Diagrama de clases de la implementación del agente 162
5.8 Estructura de un programa GALATEA 163
5.9 Modelo del sistema simple de tres taquillas con agentes 169
5.10 Modelo del sistema de supermercado con agentes 173
5.11 Modelo del sistema elevador con agentes 174
5.12 Ambiente de desarrollo para GALATEA 179
5.13 Estructura de archivos para GALATEA 180

6 El Compilador Galatea 182

7 Un ambiente de desarrollo para simulación con GALATEA 183

A Applet Montecarlo **190**

B Clases del Simulador GALATEA **191**
B.1 Diagrama general de clases del simulador 191
B.2 Diagrama de clases para el simulador 192
B.3 Diagrama de clases para listas 207
B.4 Diagrama de clases para eventos 208
B.5 Diagrama de clases para nodos 209
B.6 Diagrama de clases para mensajes 210
B.7 Diagrama de clases para métodos generales 211

Índice de Ejemplos

1 Hola Mundo empaquetado 3
2 Ejemplo de Applet 14
3 Applet Montecarlo 19
4 Clase Delta 24
5 Clase Analisis 26
6 Clase Modelo 27
7 Clase Delta 37
8 Clase Delta continuación 38
9 Clase Colono 40
10 Clase Colono continuación 41
11 Clase Interfaz 43

Sistema Simple de tres taquillas **110**
Modelo del sistema simple de tres taquillas 111
12 Código GLIDER: Sistema simple de tres taquillas 111
13 Código Java: Sistema simple de tres taquillas 113

Sistema de ferrocarril **114**
Modelo del sistema de ferrocarril 114
14 Código GLIDER: Sistema de ferrocarril 115
15 Código Java: Sistema de trenes 116

Sistema de supermercado **117**
Modelo del sistema de Supermercado 118
16 Código GLIDER: Sistema de supermercado 120
17 Código Java: Sistema de supermercado. 122

Implementación de agente **160**
18 Implementación del agente 161

Sistema simple de tres taquillas con agentes **168**
Modelo del sistema simple de tres taquillas con agentes 168
19 Código GALATEA: Sistema simple de tres taquillas con agentes . . . 170

Sistema de supermercado con agentes **171**
Modelo del sistema de supermercado con agentes 171
20 Código GALATEA: Sistema de supermercado con agentes 172

Sistema elevador con agentes **173**
Modelo del sistema elevador con agentes 174
21 Código GALATEA: Sistema de elevador con agentes 176

Licensing

This book contains both tutorial and reference information on all features of the simulation sofware Galatea.

Licencia de uso

Este libro contiene un tutorial y un cúmulo de información referencial sobre el software de simulación Galatea.

Resumen

Palabras clave:
Modelado, Simulación, GLIDER, Agentes Inteligentes, Sistemas Multi-agentes.

Capítulo 1

De la mano con Java

Este libro comienza con una revisión, estilo tutorial, de los conceptos básicos subyacentes al desarrollo de software orientado a los objetos (OO) en el lenguaje de programación Java.

No se trata, simplemente, de pagarle servicio al lenguaje Java que estaremos usando en casi todo este documento para modelar sistemas que simularemos en la plataforma Galatea. Lo que queremos hacer va un poco más allá. Los primeros esfuerzos en simulación con el computador estuvieron íntimamente ligados con esfuerzos en computación dirigidos a desarrollar lenguajes de programación más expresivos. La programación orientada a objetos surgió de esos esfuerzos y como respuesta a la urgencia de los simulistas por contar con herramientas lingüístico-computacionales para describir los sistemas que pretendían modelar y simular. Java es la culminación de una evolución que comenzó con el primer lenguaje de simulación, SIMULA [1, 2].

Nuestra intención con Galatea es repatriar la expresividad de la orientación por objetos, junto con la enorme cantidad de mejoras de Java, para su uso, ya no en simulación tradicional, sino en simulación orientada a los AGENTES.

Este tutorial Java ha sido dispuesto en dos partes: 1) Un recorrido por lo más elemental del lenguaje, incluyendo la instalación más económica que hemos podido imaginar, de manera que el simulista pueda comenzar a trabajar con los pocos recursos que pueda reunir; y 2) Un ejercicio guiado de desarrollo que termina con una aplicación Java, con el simulador incluido y con una interfaz gráfica. Las interfaces gráficas (a pesar de que ya contamos con un IDE [3]) es uno de los aspectos más débiles de Galatea. Así que queremos enfatizar, desde el principio, que toda ayuda será bien recibida.

1.1 Java desde Cero

Sin suponer mayor conocimiento previo, procedemos con los detalles de instalación.

1.1.1 Cómo instalar Java

El software básico de Java[1] está disponible gratuitamente en:

`http://java.sun.com/j2se/1.4.2/download.html` (Abril 2004)

Se debe obtener, desde luego, el software apropiado al sistema operativo de la máquina anfitrión. Si se trata de Windows, un `.exe` de autoinstalación es la opción más sencilla.

Al instalar el J2SE, el nombre de la versión gratuita de la plataforma, un par de variables de ambiente serán creadas o actualizadas. En `PATH` se agregará el directorio donde fue montado el software y en `CLASSPATH` el directorio donde están los ejecutables `.class` originales del sistema Java. Ambas variables deben se actualizadas para Galatea como se indica a continuación.

1.1.2 Cómo instalar Galatea

Galatea cuenta con una colección de clases de propósito general (no exclusivo para simulación), algunas de las cuáles aprovecharemos en este tutorial. Por ellos explicamos de inmediato como instalarla. Todo el software de Galatea viene empaquetado en el archivo `galatea.jar`. Para instalarlo, haga lo siguiente:

Configure el ambiente de ejecución así:
Unix:

```
setenv \$GALATEA_HOME\$ \${HOME}/galatea
setenv CLASSPATH \${CLASSPATH}:\${GALATEA_HOME}:.

\$GALATEA_HOME\$=\${HOME}/galatea
CLASSPATH=\${CLASSPATH}:\${GALATEA_HOME}:.
```

Windows:

```
set CLASSPATH = \$CLASSPATH:\Galatea;\Galatea\galatea.jar
```

donde `galatea`, en Unix y `Galatea`, en Windows, son nombres para el directorio donde el usuario alojará sus programas. No tiene que llamarse así, desde luego.

Coloque el archivo `galatea.jar` en este directorio (note que aparece listado en el `CLASSPATH`).

[1] En minúscula cuando nos refiramos al programa. En Mayúscula al lenguaje o a toda la plataforma.

1.1.3 Hola mundo empaquetado

Para verificar la instalación, puede usar el siguiente código, ubique el archivo `Ghola.java` en un directorio que designaremos como directorio de trabajo en lo sucesivo:

Código 1 Hola Mundo empaquetado

```
package demos.Ghola;
import galatea.*;
public class Ghola {
     /* @param argumentos, los argumentos de linea de comandos
     */
    public static void main(String args[]) {
        List unalista = new List();
        unalista.add("Simulista");
        System.out.println("Hola Mundo!, esta lista "+unalista);
        System.out.println("contiene la palabra: "+unalista.getDat());
    }
}
```

Este es un código simple de una clase, `GHola`, con un solo método, `main()`, que contiene una lista de instrucciones que la máquina virtual ejecutará cuando se le invoque. El programa debe antes ser "compilado" con la instrucción siguiente (suponiendo que estamos trabajando en ventana con el shell del sistema operativo, desde el directorio de trabajo):

```
javac Ghola.java
```

Si todo va bien, el sistema compilará exitosamente creando un archivo `Ghola.class` en el subdirectorio `demos` del subdirectorio *Ghola* del directorio de trabajo (verifiquen!).

Para correr[2] el programa use la instrucción:

```
java demos.Ghola.Ghola
```

Noten ese nombre compuesto por palabras y puntos. Ese es el nombre completo de su programa que incluye el paquete al que pertenece [3]. Si el directorio de Galatea está incluido en los caminos de ejecución (la variable `PATH` que mencionamos antes), podrá invocar su programa de la misma manera, no importa el directorio desde el que lo haga.

La salida del programa al correr será algo así:

```
Hola, esta lista List[1;1;/] -> Simulista
contiene la palabra: Simulista
```

Ud. ha corrido su primer programa en Java y en Galatea.

[2] correr es nuestra forma de decir: ejecutar o activar el programa.

[3] Esa notación corresponde también al directorio donde está el ejecutable. Revise `./demos/Ghola/Ghola` en Unix y `.\demos\Ghola\Ghola` en Windows

1.1.4 Inducción rápida a la orientación por objetos

Considere los siguientes dos problemas:

Problema 1.0 El problema de manejar la nomina de **empleado**s de una institución. Todos los empleados son personas con *nombre*s e *identificación*, quienes tienen asignado un *salario*. Algunos empleados son **gerente**s quienes, además de los ATRIBUTOS anteriores, tienen la responsabilidad de dirigir un departamento y reciben un *bono* específico por ello.

El problema es mantener un registro de empleados (los últimos, digamos, 12 pagos a cada uno) y ser capaces de ***actualizar*** salarios y bonos y de ***generar*** cheques de pago y reportes globales.

Para resolver el problema con la orientación por objetos, decimos, por ejemplo, que cada tipo de empleado es una clase. Cada empleado será representado por un objeto de esa clase. La descripción de la clase incluye también, los métodos para realizar las operaciones que transforman el estado (los *atributos*) de cada objeto, de acuerdo a las circunstancias de uso. Observen que necesitaremos otros "objetos" asociados a los empleados, tales como cheques y reportes.

Problema 2.0 Queremos controlar la **dedicación** de ciertas empleados a ciertas tareas. Todo **empleado** tiene una *identificación* asociada. Además, cada empleado puede ser responsable de un numero no determinado de *tareas*, cada una de las cuales consume parte de su *horario* semanal y de su *dedicación* anual al trabajo. Hay algunas *restricciones* sobre la dedicación anual al trabajo y sobre el numero de horas asignadas a los diversos tipos de tareas.

En este caso, el objeto principal es el **reporte** de carga laboral de cada **empleado**. El empleado puede ser modelado como un objeto sencillo, uno de cuyos atributos es la *carga* en cuestión. Necesitamos también, algunos objetos que controlen la comunicación entre el usuario del sistema y los objetos anteriores y que permitan elaborar el informe de carga progresivamente.

Lo que acabamos de hacer con esos dos ejemplos es un primer ejercicio de diseño orientado a los objetos, OO. En estos problemas, hemos identificado los tipos de objetos involucrados (a los que identificamos con **negritas** en el texto) y los atributos con (*cursivas*) y sus métodos con (***cursivas negritas***). También hemos asomado la posibilidad de incluir otro elemento importante en OO, las *restricciones*.

En la secciones siguiente trataremos de aclarar cada uno de esos términos[4] y el cómo se reflejan en un código Java.

[4]Salvo el de restricciones que queda lejos del alcance de un tutorial elemental

1.1.5 Qué es un objeto (de software)

En OO, un objeto es una "cosa virtual". En algunos casos (como en los problemas anteriores) es la representación de un objeto físico o "social". Los objetos de software son MODELOS de objetos reales, en ese contexto.

Desde el punto de vista de las estructuras de datos que manipula el computador, un objeto es "una cápsula que envía y recibe mensajes". Es una estructura de datos junto con las operaciones que la consultan y transforman. Estos son los métodos, los cuáles corresponden a las rutinas o procedimientos en los lenguajes de programación imperativa.

En lo sucesivo y siempre en OO, un objeto es simplemente una *instancia de una clase*. La clase es la especificación del objeto. Una especie de descripción general de cómo son todos los objetos de **este tipo**.

1.1.6 Qué es una clase

Ya lo dijimos la final de la sección anterior. Una clase es una especificación para un tipo de objetos. Por ejemplo, para nuestro primer problema podríamos usar:

```
class Empleado {
    String nombre;
    String ID;
    float salario; }
```

`Empleado` es una clase de objetos, cada uno de los cuáles tiene un `nombre` (cadena de caracteres, `String`), una identificación `ID` (otro `String`) y un `salario` (un número real, `float`).

1.1.7 Qué es una subclase

Una de las ventajas más interesantes de la OO es la posibilidad de **reusar** el software ajeno extendiéndolo y adaptándolo a nuestro problema inmediato. Por ejemplo, si alguien más hubiese creado la clase `Empleado` como en la sección anterior, para crear la clase `Gerente` (que entendemos como un tipo especial de empleado) podríamos usar:

```
class Gerente extends Empleado {
    String departamento;
    float  bono;

    Gerente(String n, String i, float f) {
        Empleado(n,i,f);
        departamento = "Gerencia";
        bono = 1.20f; // salario + 20\% = bono
    }
}
```

1.1.8 Qué es una superclase

La posibilidad de crear subclases implica la existencia de una jerarquía de tipos de objetos. Hacia abajo en la jerarquía, los objetos son más especializados. Hacia arriba son más generales. Así la superclase (clase inmediata superior) de Empleado podría ser la siguiente si consideramos que, en general, un Empleado es una Persona:

```
class Persona {
   String nombre;
   String id;

   public void Persona(String n, String id) {
      this.nombre = n;
      this.id = id;
   }
   public void identifiquese() {
      System.out.println("Mi nombre es " + this.id);
   }
}
```

Con lo cual, tendríamos que modificar la declaración de la clase Empleado, así:

```
class Empleado extends Persona {
    float salario ;

    public void Empleado(String n,String id,float primerSueldo) {
        Persona(n, id);
        this.salario = primerSueldo ;
    }

    public void identifiquese() {
        super.identifiquese() ;
        if (this.salario < 1000000) {
        System.out.println("... y me pagan como a un profesor") ;
        }
    }
}
```

En estas dos últimas especificaciones de las clase `Persona` y `Empleado` ha aparecido el otro elemento fundamental de un objeto de software: los métodos. Un método es una rutina (como los procedimientos en Pascal o las funciones en C) que implementan el algoritmo sobre las estructuras de datos y atributos del objeto (y de otros objetos asociados).

En OO, los métodos se heredan hacia abajo en la jerarquía de clases. Por eso, los métodos de nuestra clase `Persona`, son también métodos de la clase `Empleado`, con la posibilidad adicional, en Java, de poder invocar desde una sub-clase, los métodos de su superclase, como se observa en nuestro ejemplo (`super.identifiquese()`[5]).

[5]Algunas versiones de la máquina virtual permiten usar super para invocar al constructor de la superclase, siempre que aparezca como la primer instrucción del constructor de la clase hijo. Esto, sin embargo, no siempre funciona.

Un ultimo detalle muy importante es la presentación de un tipo especial de método: **el constructor**, que es invocado antes de que el objeto exista, justamente al momento de crearlo (con algo como `new Constructor(Parametros);`) y cuyo objetivo suele ser el de iniciar (*inicializar* decimos en computación siguiendo el anglicismo) las estructuras de datos del objeto. Noten el uso del constructor `Empleado(String n, String id, Float primerSueldo)` en el último ejemplo. Los métodos constructores no se heredan. Las previsiones para crear los objetos deben ser hechas en cada clase. Si no se declara un constructor, el compilador Java introduce uno por omisión que no tiene parámetros.

1.1.9 Qué es poliformismo

Con lo poco que hemos explicado, es posible introducir otro de los conceptos revolucionarios de la OO: el polimorfismo. En corto, esto significa que un objeto puede tener varias formas (cambiar sus atributos) o comportarse de varias maneras (cambiar sus métodos).

Por ejemplo, suponga que Ud declara la variable `fulano` de tipo `Persona` y crea una instancia de `Persona` en `fulano`:

```
Persona fulano = new Persona();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese() ;
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Persona`.

Sin embargo, si Ud declara:

```
Empleado fulano = new Empleado();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese() ;
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Empleado`, pues en este clase se **sobreescribió el método**. Es decir, se inscribió un método con el mismo nombre y lista de argumentos que el de la superclase que, queda entonces, oculto.

Noten que este comportamiento diverso ocurre, a pesar que un `Empleado` es también una `Persona`. Esta es una de las dimensiones del polimorfismo.

El polimorfismo, no obstante, tiene un carácter mucho más dinámico. Por ejemplo, esta instrucción es perfectamente válida en Java, con los códigos anteriores:

```
Empleado fulano = new Gerente("Fulano de Tal","F", 1000000f) ;
```

La variable `fulano` de tipo `Empleado`, se refiere realmente a un `Gerente` (que es una subclase de `Empleado`).

En este caso, fulano es un Empleado. Si Ud trata de utilizar algún atributo específico de la subclase, no podrá compilar. Trate con:

```
Empleado fulano = new Gerente("Fulano de Tal","F", 1000000f) ;
System.out.println("mi bono es " + fulano.bono);
```

Aquí `bono`, a pesar de ser un atributo de la clase `Gerente`, es desconocido si el objeto es de tipo `Empleado`. El compilador lo denuncia.

Al revés, en principio la instanciación también es posible:

```
Gerente fulano = (Gerente) new Empleado("Fulano de Tal", "F", 1000000f) ;
```

El compilador lo admite como válido al compilar. Sin embargo, la máquina virtual[6] lanzará una excepción (el error es `ClassCastException`) en tiempo de ejecución. Todo esto es parte del sofisticado sistema de seguridad de tipos de Java. No hay espacio en el tutorial para explicar todas sus implicaciones.

1.1.10 Qué es un agente

En la jerga de la Inteligencia Artificial, un agente es un dispositivo (hardware y software) que percibe su entorno, razona sobre una representación de ese entorno y de su conducta y actúa para cambiar ese mismo entorno (o para impedir que cambie). El término, desde luego, está asociado a la etimología tradicional (del latin *agere*: hacer) y se le usa para indicar a aquel (o aquello) que **hace algo** (posiblemente en beneficio de alguien más). La adopción del término en la literatura ingenieril de la Inteligencia Artificial tiene que ver con la aparición de una corriente o escuela de trabajo que enfatiza que los dispositivos inteligentes deben demostrar que lo son "haciendo algo". En el extremo opuesto (los que no hacen nada) se pretendió dejar a los primeros razonadores simbólicos que concentraban todo el esfuerzo en emular el "pensar" de los humanos, con un mínimo compromiso por la acción.

Como ilustraremos más adelante, en Galatea nos reencontramos con esa noción del razonador simbólico que procesa una representación de cierto dominio de conocimiento (asociado a un sistema) y usa el producto de ese procesamiento para guiar la acción de un agente. El objetivo de proyecto Galatea, en cuanto a agentes se refiere, es proveer a modelistas y simulistas de un lenguaje con suficiente expresividad para describir a los humanos que participan de un sistema que se quiere simular. Por esta razón, nuestros agentes son inteligentes.

Un agente inteligente tiene su propio "proyecto" de vida, con metas, creencias y capacidades que le permiten prestar, con efectividad, servicios para los que se requiere

[6] Algunas máquinas virtuales) no reportan error de ejecución.

inteligencia. A lo largo del documento, mostraremos cómo abordar la representación de los agentes y trataremos de justificar el que nos atrevamos a llamarles inteligentes.

1.1.11 Qué relación hay entre un objeto y un agente

Algunos autores reduce la caracterización de agente a una entidad que percibe y actúa [4]. Usando la matemática como lenguaje de especificación, dicen ellos que un agente es una función con dominio en una "historia perceptual" y rango en las acciones:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Esta caracterización matemática es ciertamente muy útil conceptualmente (con algunos complementos, como mostraremos en los capítulos centrales de este libro), pero lo es poco para el desarrollador de software.

Es quizás más provechoso decir que un agente es un objeto, en el sentido que hemos venido discutiendo, con una sofisticada estructura interna (estructuras de datos, atributos y métodos) y con una relación intensa de intercambio con su entorno. En Galatea, esta visión de agente se refleja en la implementación de la clase `galatea.hla.Agent` y en el paquete `galatea.gloria`, como explicaremos más adelante.

1.1.12 Por qué objetos

La programación orientada a los objetos se ha impuesto tan arrolladoramente que pocos harán ahora la pregunta que intitula esta sección. Sin embargo, queremos hacer énfasis en lo apropiado de la OO, y ahora la orientación a los agentes, OA, como herramientas para abordar la complejidad. Los profesores Oscar García y Ricardo Gutiérrez, de Wright University explican que una medida del esfuerzo de desarrollo en programación imperativa tal como:

$$\text{Esfuerzo de programación} = \textit{constante} * (\text{lineas de código en total}) * 1.5 \qquad (1.1)$$

Se convierte en la OO[5]:

$$\text{E. de pr. en OO} = \textit{constante} * \Sigma \textit{objetos}(\text{l. de c. de cada objeto}) * 1.5 \qquad (1.2)$$

Debemos admitir, sin embargo, el esfuerzo considerable que debe realizar un programador para aprender los lenguajes OO, buena parte del cual debe emplearlo en

aprender lo que otros programadores han hecho. De su habilidad para re-utilizar, dependerá su éxito casi definitivamente.

En particular, un lenguaje como Java que conserva parte de la sintaxis de nivel intermedio del lenguaje C puede resultar muy desafiante. Por esta razón, este texto incluye el siguiente recorrido Tutorial sobre Java.

1.2 Java en 24 horas

1.2.1 Java de bolsillo

El material incluido en esta sección está basado en un curso dictado por Andrew Coupe, Sun Microsystems Ltd, en 1994. Ha sido actualizado con los detalles relevantes para Java 2, la nueva generación de Java y hemos cambiado su ejemplo para adecuarlo a un problema más cercano a los simulistas.

Coupe resumió las características de este lenguaje OO, así:

> "Java es un lenguaje pequeño, simple, seguro, orientado a objetos, interpretado o optimizado dinámicamente, que genera byte-codes, neutral a las arquitecturas, con recolección de basura, multihebrado, con un riguroso mecanismo para controlar tipos de datos, diseñado para escribir programas dinámicos y distribuidos".

Más allá del ejercicio promocional, tradicional en un entorno industrial tan competido como este, esa última definición debe servir para ilustrar la cantidad de tecnologías que se encuentran en Java. Esa combinación junto con una política (más) solidaria de mercadeo y distribución, han convertido a Java, a nuestro juicio, en una plataforma de desarrollo muy efectiva y muy robusta.

Para el usuario final, el programador de aplicaciones, quizás la virtud que más le puede interesar de Java es la facilidad para desarrollos de software distribuidos: "Escríbelo aquí, córrelo dondequiera" es uno de los panfletos publicitarios de Java. Significa que uno puede producir el software acabado en una máquina con arquitectura de hardware completamente diferente de aquella en la que se ejecutará el software. Una máquina virtual se encarga de unificar las "interfases" entre programas Java y las máquinas reales.

Estas fueron la metas del diseño de Java. Las listamos aquí con una invitación al lector para que verifique su cumplimiento:

* Java tiene que ser simple, OO y familiar, para lograr que los programadores produzcan en poco tiempo. Java es muy parecido a C++.

* Java debe ser seguro y robusto. Debe reducir la ocurrencia de errores en el software. No se permite `goto`. `labels`, `break` y `continue`, si se permiten. Pero sin las armas blancas de doble filo: No a la aritmética de punteros (apuntadores).

* Java debe ser portátil, independiente de la plataforma de hardware.

* Java tiene que ser un lenguaje útil (usable).

* Java implanta un mecanismo para recolección de basura. No a las filtraciones de memoria.

* Java solo permite herencia simple entre clase (una clase solo es subclase de una más). Es más fácil de entender y, aún, permite simular la herencia múltiple con `interface`.

Java trata de eliminar las fuentes más comunes de error del programador C. No permite la aritmética de apuntadores. Esto no significa que no hay apuntadores, como algunas veces se ha dicho. En OO, una variable con la referencia a un objeto se puede considerar un apuntador al objeto. En ese sentido, todas las variables Java (de objetos no primitivos, como veremos) son apuntadores. Lo que no se permite es la manipulación algebraica de esos apuntadores que si se permite en C o C++.

El programador no tiene que preocuparse por gestionar la memoria. Esto significa que no hay errores de liberación de memoria. El trabajo de reservar memoria para los objetos se reduce a la invocación `new` que crea las instancias, pero sin que el programador tenga que calcular espacio de memoria (como se puede hacer con algunas invocaciones `malloc` en C). Para liberar la memoria, tampoco se requiere la intervención del programador. La plataforma incorpora un "recolector de basura" (garbage collector) que se encarga de recuperar a memoria de aquellos objetos que han dejado de ser utilizados (ninguna variable apunta hacia ellos). El mecanismo puede ser ineficiente sin la intervención del usuario, pero el mismo recolector puede ser invocado a voluntad por el programador (Ver objeto `gc` en el API Java).

Java es seguro por tipo. Esto quiere decir que se establecen tipos de datos muy rígidos para controlar que la memoria no pueda ser empleada para propósitos no anticipados por el programador. El mapa de memoria se define al ejecutar.

Hay un chequeo estricto de tipos en tiempo de compilación que incluye evaluación de cotas, una forma de verificación antivirus inteligente, cuando las clases se cargan a través de la red. El cargador de clases les asigna a los applets de diferentes máquinas, diferentes nombres de espacios.

Los applets, estas aplicaciones Java que se descargan desde la red, operan en lo que se conoce como una "caja de arena" (como las que usamos para las mascotas) que impide que el código desconocido pueda afectar la operación de la máquina que lo recibe. Esto es protección contra caballos de troya. Los applets son revisados al

ejecutarlos. El cargador evalúa los nombres locales primero. No hay acceso por la red. Los applets sólo puede acceder a su anfitrión. Los applets más allá de un firewall, sólo pueden acceder al espacio exterior.

Java pretende ser portátil (neutral al sistema operativo). Nada en Java es dependiente de la implementación en la máquina donde se le ejecute. El formato de todos los tipos está predefinido. La definición de cada uno incluye un extenso conjunto de bibliotecas de clases, como veremos en un momento.

Los fuente Java son compilados a un formato de bytecode independiente del sistema. Esto es lo que se mueve en la red. Los bytecodes son convertidos localmente en código de máquina. El esfuerzo de portabilidad es asunto de quienes implementan la máquina virtual en cada plataforma.

Java tiene tipos de datos llamados primitivos. Son enteros complemento dos con signo. `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits); Punto flotantes IEEE754, sencillo (`float`) y doble (`double`); y `Char` 16 bit Unicode.

Java es interpretado o compilado justo a tiempo para la ejecución. Esto significa que código que entiende la máquina real es producido justo antes de su ejecución y cada vez que se le ejecuta. La implicación más importante de esto es que el código Java tarda más (que un código compilado a lenguaje máquina) en comenzar a ejecutarse. Esto ha sido la fuente de numerosas críticas a la plataforma por su supuesta ineficiencia. Un análisis detallado de este problema debe incorporar, además de esas consideraciones, el hecho de que los tiempos de descarga a través de la red siguen siendo un orden de magnitud superiores a la compilación just-in-time y, además, que la plataforma Java permite enlazar bibliotecas (clases y paquetes) en tiempo de ejecución, incluso a través de la red.

La otra gran idea incorporada a Java desde su diseño es la plataforma para multiprogramación. Los programadores Java decimos que Java es multihebrado. El programador puede programar la ejecución simultánea[7] de "hilos de ejecución", hebras, en un mismo programa. La sincronización (basada en monitores) para exclusión mutua, un mecanismo para garantizar la integridad de las estructuras de datos en los programas con hebras, está interconstruida en el lenguaje (Ver `Threads` en el API Java).

1.2.2 Los paquetes Java

Un paquete es un módulo funcional de software y la via de crear colecciones o bibliotecas de objetos en Java. Se le referencia con la instrucción `import` para incorporar sus clases a un programa nuevo. Lo que llamamos la Application Programming Interface,

[7]Por lo menos pseudosimultánea. El paralelismo real depende, desde luego, de contar con más de un procesador real y una máquina virtual que paralelice.

API, de Java es un conjunto de paquetes con la documentación apropiada para que sean empleados en desarrollos. La especificación Java incluye un conjunto de bibliotecas de clases que acompañan cualquier distribución de la plataforma: `java.applet`, `java.awt`, `java.io`, `java.lang`, `java.net`, `java.util`.

El paquete `java.applet` proporciona una clase para programas Java imbuidos en páginas web: Manipulación de applets. Métodos para manipular audioclips. Acceso al entorno del applet.

El paquete `java.awt` es la caja de herramientas para armar GUIs, recientemente mejorada con la introducción de Swing. Tanto `Awt` como `Swing` implementan los componentes GUI usuales: `java.awt.Graphics` proporciona algunas primitivas de dibujo. Se apoya en el sistema de ventajas del sistema operativo local (`java.awt.peer`). `java.awt.image` proporciona manipuladores de imágenes.

El paquete `java.io` es la biblioteca de entrada y salida: Input/output `streams`. Descriptores de archivos. `Tokenizers`. Interfaz a las convenciones locales del sistema de archivos.

El paquete `java.lang` es la biblioteca del lenguaje Java: Clases de los tipos (`Class`, `Object`, `Float`, `String`, etc). La Clase `Thread` provee una interfaz independiente del sistema operativo para las funciones del sistema de multiprogramación.

El paquete `java.net` contiene las clases que representan a objetos de red: Direcciones internet. URLs y manipuladores MIME Objetos `socket` para clientes y servidores.

El paquete `java.util` Un paquete para los "utilitarios": `Date`, `Vector`, `StringTokenizer`, `Hashtable`, `Stack`.

Java es una plataforma completa de desarrollo de software muy económica (la mayor parte es gratis). Muchos proveedores de tecnología teleinformática son propietarios de licencias de desarrollo Java (IBM por ejemplo) aunque Sun Microsystems sigue siendo la casa matriz.

Los Java Developers Kits son los paquetes de distribución de la plataforma y ahora se denominan Software Developer Kits. Se distribuyen gratuitamente para todos los sistemas operativos comerciales. Incluye herramientas de documentación y BDKs. Hay muchos IDEs (Visual Age, por ejemplo). Browsers Explorer y Netscape lo incorporaron desde el principio como uno de sus lenguajes script (pero, cuidado que Java NO ES Javascript).

Es muy difícil producir un resumen que satisfaga todas las necesidades. Por fortuna, la información sobre Java es muy abundante en Internet. Nuestra intención aquí es proveer información mínima y concentrar la atención del lector en algunos detalles útiles para simulación en las próximas secciones.

1.2.3 Un applet

Un applet[8] es una forma inmediata de agregar dinámismo a los documentos Web. Es una especie de extensión indolora del browser, para las necesidades de cada página Web. El código se descarga automáticamente de la red y se ejecuta cuando la página web se expone al usuario. NO se requiere soporte en los servidores Internet, salvo para alojar el programa y la página Web.

Sin más preámbulo, esto es un applet:

Código 2 Ejemplo de Applet

```
import java.awt.*; import java.applet.*;
public class EjemploApplet extends Applet {
    Dimension app_size;
    public void init() {
        app_size = this.size();
    }
    double f(double x) {
        return (Math.cos(x/5) + Math.sin(x/7) + 2) * app_size.height / 4;
    }
    public void paint(Graphics g) {
        for (int x = 0; x < app_size.width ; x++) {
            g.drawLine(x, (int) f(x), x+1, (int) f(x+1));
        }
    }
}
```

Este último es un programa muy sencillo que dibuja una gráfica de la función f al momento de cargar el applet (que es cuando el sistema invoca el método `init()`. Noten que los applets no usan la función `main()` como punto de arranque los programas. Esta es reservada para las aplicaciones normales).

Para ejecutar el applet puede hacerse (luego de generar el `.class`):

```
appletviewer EjemploApplet
```

o invocarlo desde una página web como se mostrará más adelante.

1.2.4 Anatomía de un applet

Una applet es una aplicación Java construida sobre la colección de clases en el paquete `java.applet` o `java.swing`.

Así se declara la clase que lo contiene:

[8]El nombre es una variación de la palabra inglesa para aplicación "enchufable". Estas variaciones se han vuelto comunes para la comunidad Java con la aparición de los *Servlets*, los *Midlets* y los *Aglets*, todos componentes `enchufables`.

```
import java.applet.*;

public class AppletMonteCarlo extends Applet {

}
```

Donde `AppletMonteCarlo` es el nombre que hemos escogido para nuestro ejemplo (y que el programador puede cambiar a voluntad, desde luego).

A diferencia de las aplicaciones "normales", que solo requieren codificar un método `main()` como punto de arranque, los applets deben configurar, por lo menos 2, puntos de arranque. Con `init()` se indica al sistema el código que debe ejecutarse cada vez que se descargue la página web que contiene el applet. Con `start()` se indica el código que habrá de ejecutarse cada vez que el applet se exponga a la vista del usuario del navegador (cada vez que la página web se exhiba o se "maximice" la ventana del navegador).

```
Import java.applet.*;

public class AppletMonteCarlo extends Applet {
    public void init() {
    }
    public void start() {
    }
}
```

1.2.5 Parametrizando un applet

Los applets, imbuidos en páginas Web como están, tienen acceso a información el esas páginas. Por ejemplo, nuestro applet podría obtener un nombre que necesita, de una de las etiquetas de la página que lo aloja, con este código:

```
import java.applet.*;

public class AppletMonteCarlo extends Applet {
    String param;

    public void init() {
        param = getParameter("imagen");
    }
}
```

y con este texto incluido en la página, justo en el punto donde se inserta el applet. En los ejemplos que acompañan este texto, se muestra la página web (`AppletMonteCarlo.html`) que invoca a este applet, sin ese detalle del parámetro que se sugiere como ejercicio al lector.

```
\<param name=imagen value="imagen.gif"\>
```

1.2.6 Objetos `URL`

El applet puede, además, usar las clase de apoyo para construir direcciones Web bien formadas y usarlas en la recuperación de información:

```
import java.applet.*;
import java.net.*;

public class AppletMonteCarlo extends Applet {
    String param;

    public void init() {
        param = getParameter("imagen");
        try {
            imageURL = new URL(getDocumentBase(), param);
        } catch (MalformedURLException e) {
            System.out.println("URL mal escrito");
            return;
        }
    }
}
```

1.2.7 Gráficos: colores

El applet es una "aplicación gráfica". Es decir, toda la salida se realiza a través de un panel que es una ventana para dibujar. No es debe sorprender que exista una relación muy estrecha entre applets y herramientas de dibujo.

El siguiente código crea un arreglo para almacenar una paleta de 3 colores que se usará más adelante.

```
import java.applet.*;

int paints[] ;

/* prepara la caja de colores */
paints = new int[3];
paints[0]=Color.red.getRGB();
paints[1]= Color.green.getRGB();
paints[3]= Color.blue.getRGB();
```

Lo más interesante de esta relación entre applets y dibujos no es sólo lo que uno puede programar, sino lo que ya ha sido programado. Este código, por ejemplo, contiene una serie de objetos que nos permiten cargar una imagen a través de la red. La imagen puede estar en alguno de los formatos profesionales de internet (.gif, jpeg, entre otros). El sistema dispone de un objeto para almacenar la imagen (`Image`) y, además de todo un mecanismo para garantizar que la imagen es cargada integramente (`MediaTracker`).

```
Image picture;
MediaTracker tracker;
tracker = new MediaTracker(this);
picture = this.getImage(imageURL) ;
tracker.addImage(picture,0);
```

Java provee también de objetos para que el programador puede manipular gráficos (`Graphics`) que se dibujan en su panel.

```
Graphics gc;
gc = getGraphics;
```

1.2.8 Ejecutando el applet

El código a continuación, es una implementación del método `start()` que muestra como cargamos la imagen (con el `tracker`) y luego la dibujamos en el panel del applet (`gc.drawImage`), suponiendo, por supuesto que los objetos han sido declarados e inicializados en otros lugares del código.

```
import java.applet.*;

public class AppletMonteCarlo extends Applet {
    String param;
    ...
    public void start() {
        try {
            tracker.waitForID(0);
        } catch (InterrumptedException e) {
            System.out.print("No pude!");
        }

        image_width = picture.getWidth(this);
        image_heigth = picture.getHeight(this);

        gc.drawImage(picture, 0, 0, this);
    }
}
```

1.2.9 Capturando una imagen

Con muchas herramientas de visualización científica se hace un esfuerzo, quizás exagerado, por aislar al usuario de los detalles de manipulación de su data. En muchos casos, por el contrario, el usuario-programador necesita todas las facilidades para manipular esa data para sus propósitos particulares.

El código que se muestra a continuación es un ejemplo de cómo cargar la imagen que hemos obtenido a través de la red (en el objeto `picture`) en un arreglo de pixels. Un pixel corresponde a un punto en la pantalla del computador. Para el software, el pixel es un número que designa el color e intensidad que se "pinta" en cierta posición de la pantalla.

```
PixelGrabber pg;
int pixels[] ;

pixels = new int[image_width*image_height] ;
pg = new PixelGrabber(picture, 0, 0, image_width, image_height,
pixels, 0, image_width);

try {
    pg.grabPixels();
} catch (InterruptedException e) {
    System.err.println("No pude transformar la imagen");
}
```

Lo que hemos hecho ese código es capturar (agarrar, `Grabber`. Ver `PixelGrabber`) los píxeles de nuestra imagen en un arreglo desde donde los podemos manipular a voluntad (como se muestra en el ejemplo completo).

1.2.10 Eventos

Las aplicaciones de software moderna rara vez interactuan con el usuario a través de un comandos escritos con el teclado. Las formas más populares de interacción tienen que ver con el uso del ratón, de menús, de botones y de otra opciones gráficas.

Un problema con esos diversos modos de interacción es que requieren ser atendidos casi simultáneamente y en forma coherente con la aplicación y con sus interfaz al usuario.

Para resolver ese problema de raíz, la plataforma Java incorporó, desde el principio, un sistema de manejo de eventos en su interfaz gráfica. Los eventos son, para variar, representados como objetos (de la clase `Events`) y dan cuenta de cualquier cosa que pase en cualquier elemento de interfaz (o en cualquier otro, de hecho). Un evento es un objeto **producido** por aquel objeto sobre el que ocurre algo (por ejemplo un botón que es presionado, *click*). El objeto que produce el evento deben tener asociado un manejador del evento. Este es otro objeto que **escucha** los eventos (un `Listener`), los **reconoce** (de acuerdo a características predefinidas por el programador) y los **maneja**, ejecutando ciertos códigos que han sido definidos por el programador.

En un dramático cambio de dirección, Java 2 transformó el mecanismo de manejo de eventos y es por ello que algunos viejos programas Java que manipulaban eventos, tienen dificultades para funcionar.

Lamentablemente, los detalles del manejo de eventos son difíciles de resumir. Es mucho más productivo ver el código de algunos ejemplos. Este, por ejemplo, es el que usamos en el applet que hemos venido construyendo. Observen como cambia la declaración de la clase para "implementar" los objetos que escuchan eventos en cada dispositivo:

En este caso, los interesantes son los métodos `keyRelease()` y `mouseClicked()`. Los demás no hacen nada (salvo imprimir un mensaje en salida standard), pero **deben**

Código 3 Applet Montecarlo

```
public class AppletMonteCarlo extends Applet implements
KeyListener, MouseListener {
    ...
    public void keyPressed(KeyEvent e) {
        System.out.println("keyPressed");
    }
    public void keyReleased(KeyEvent e) {
        System.out.println("keyReleased");
        /* pasa al nuevo color */
        paint_col++;
        paint_col = paint_col % 3;
        System.out.println("Nuevo color: " + paint_col);
    }
    public void keyTyped(KeyEvent e) {
        System.out.println("keyTyped");
    }
    public void mouseClicked(MouseEvent e) {
        System.out.println("mouseClicked");
        /* Captura la posicion del ratn */
        int x = e.getX();
        int y = e.getY();
        /* Un click dispara la reconstruccin de la imagen */
        if (x < image_width & y < image_height) {
                /* llamar a la rutina de llenado: floodFill */
            floodFill(x,y, pixels[y*image_width+x]);
        }
        /* .. y luego la redibuja */
        repaint(); // desde el browser llama a paint()
    }
    public void mouseEntered(MouseEvent e) {
        System.out.println("mouseEntered");
    }
    public void mouseExited(MouseEvent e) {
        System.out.println("mouseExited");
    }
    public void mousePressed(MouseEvent e) {
        System.out.println("mousePressed");
    }
    public void mouseReleased(MouseEvent e) {
        System.out.println("mouseReleased");
    }
    ....
}
```

ser implementados (Ver `interface` en el API Java).

Sugerimos al lector que, sobre el código completo de este ejemplo (en el anexo A ubique a los productores de eventos y a los escuchas).

1.2.11 paint(): Pintando el applet

Un detalle importante para cerrar la discusión sobre visualización (sobretodo la animada) es quien pinta el dibujo del applet. Lo pinta el usuario cada vez que quiere, pero también lo pinta el propio navegador (browser) se exhibe es la página, en respuesta a su propia situación (abierta, cerrada, bajo otra ventanas, etc.).

Para proveer una interfaz uniforme al sistema y al programador, se ha dispuesto del método `paint()`. El programador escribe el código de paint, para definir como desea que "se pinte" sus applet. Pero el método es invocado por el navegador. Si el usuario quieren forzar una llamada a `paint()`, usa `repaint()`, como se muestra en el ejemplo.

Este es el código que usamos para `paint()`. Noten el uso de otro método `update()`.

```
public void paint(Graphics gc) {
    update(g);

public void update(Graphics g) {
    Image newpic;

    newpic = createImage(new MemoryImageSource(
             image_width, image_height, pixels, 0, image_width);
    g.drawImage(newpic, 0, 0, this) ;

}
```

1.2.12 El AppletMonteCarlo completo

.

Como dijimos al principio de esta parte, hemos querido ofrecer un ejemplo completo de una aplicación Java funcional, destacando el manejo gráfico en el que Galatea no es tan fuerte (por falta de tiempo de desarrollo). Eso es el `AppletMonteCarlo`. Pero su nombre también sugiere una herramienta conceptual sumamente popular en simulación.

El método de MonteCarlo es una aplicación de la generación de números aleatorios. En nombre rememora el principado Europeo, célebre por sus casinos y sus juegos de azar (entre otras cosas).

No vamos a diluirnos en los detalles, que se obscurecen fácilmente, de la estocástica. Lo que tenemos en el ejemplo es una aplicación simple de la rutina de

MonteCarlo de generación de números aleatorios para estimar areas encerradas por un perímetro que dibuja una función.

Lo interesante del ejemplo es que la función no tiene que ser alimentada la ejemplo con su fórmula matemática o con una tabla de puntos. Una imagen (`.gif` o `.jpeg`) es el forma que el programa espera para conocer la función sobre la que calculará el área, al mismo tiempo que cambia los colores de puntos para ilustrar el funcionamiento del algoritmo. No se ofrece como un método efectivo para el cálculo de áreas, sino como un ejemplo sencillo de la clase de manipulaciones que se pueden realizar con Java.

Noten, por favor, que en este ejemplo, el simulador de números aleatorios empleado no es el originario de Java, sino la clase `GRnd` de **Galatea**[9].

Ver ahora el apéndice A.

[9]Es muy importante notar como inicializamos la "semilla" del generador Galatea en el método `init()` del `AppletMonteCarlo`.

Para hacer algo, comienza por el principio, sigue directo hacia el final y, cuando llegues allí termina.

Capítulo 2

Directo al grano

Este segundo capítulo tiene como propósito dirigir a un simulista en la creación de un modelo de simulación Galatea. Nos proponemos hacer eso en dos fases. En la primera, mostramos paso a paso cómo codificar en Java un modelo básico para Galatea. En la segunda fase, repetimos el ejercicio, pero esta vez sobre un modelo de simulación multi-agente de tiempo discreto.

La intención trascendente del capítulo es motivar ejercicios de "hágalo Ud mismo", luego de darle al simulista las herramientas lingüísticas básicas. Explicar la infraestructura de simulación que las soporta nos tomará varios capítulos en el resto del libro. El lector no debería preocuparse, entretanto, por entender los conceptos subyacentes, más allá de la semántica operacional de los lenguajes de simulación.

En otro lugar explicamos que Galatea es una *familia de lenguajes de simulación*. En este capítulo, por simplicidad, sólo usamos Java.

2.1 El primer modelo computacional

Galatea heredó la semántica de Glider[6]. Entre varias otras cosas, esto significa que Galatea es, como Glider, *orientado a la red*. Esto, a su vez, refleja una postura particular de parte del modelista que codifica el modelo computacional en Galatea. En breve, significa que el modelista identificará componentes del sistema modelado y los ordenará en una red, cuyos enlaces corresponden a las vías por las que esos componentes se comunican. La comunicación se realiza, normalmente (aunque no exclusivamente como veremos en los primeros ejemplos) por medio de pase de mensajes.

La simulación se convierte en la reproducción del desenvolmiento de esos componentes y de la interacción entre ellos. Esta es, pura y simple, la idea original de la orientación por objetos que comentabamos al principio del capítulo anterior.

En Glider, y en Galatea, esos componentes se denominan **nodos**. Los hay de 7 tipos: `Gate`, `Line`, `Input`, `Decision`, `Exit`, `Resource`, `Continuous` y `Autonomous`[1]

Estos 7 tipos son estereotipos de componentes de sistemas que el lenguaje ofrece "ya pensandos" al modelista. El modelista, por ejemplo, no tiene que pensar en cómo caracterizar un componente de tipo recurso. Sólo debe preocuparse de los detalles específicos de los recursos en el sistema a modelar, para luego incluir los correspondientes nodos tipo `Resource` en su modelo computacional. Esta es la manifestación de otra idea central de la orientación por objetos: la reutilización de código.

Una de las líneas de desarrollo de Galatea apunta a la creación de un programa traductor para la plataforma, como explicaremos en un capítulo posterior. Ese interpretador nos permitirá tomar las especificaciones de nodos en un formato de texto muy similar al que usa Glider (lo llamaremos **nivel Glider**) y vertérlas en código Java, primero y, por esta vía, en programas ejecutables.

Aún con ese traductor, sin embargo, siempre es posible escribir los programas Galatea, respetando la semántica Glider, directamente en Java (lo llamaremos **nivel Java**). Hacerlo así tiene una ventaja adicional: los conceptos de la orientación por objetos translucen por doquier.

Por ejemplo, para definir un tipo de nodo, el modelista que crea una clase Java que es una subclase de la clase `Node` del paquete `galatea.glider`. Con algunos ajuste de atributos y métodos, esa clase Java se convierte en un `Node` de alguno de los tipos Glider. Para *armar* el simulador a nivel Java, el modelista usa las clases así definas para generar los objetos que representan a los nodos del sistema.

De esta forma, lo que llamamos un modelo computacional Galatea, a nivel Java, es un conjunto de subclases de Node y, por lo menos una clase en donde se definen las variables globales del sistema y se coloca el programa principal del simulador.

Todo esto se ilustra en las siguientes subsecciones, comenzando por las subclases `Node`.

2.1.1 Nodos: componentes del sistema a simular

Tomemos la caracterización más primitiva de un sistema dinámico: Una máquina abstracta definida por un conjunto de variables de estado y una función de transición, normalmente etiquetada con la letra griega δ, que actualiza ese estado a medida que pasa el tiempo. Una caracterización tal, es *monolítica*: no hay sino un solo componente y sus régimen de cambio es el que dicta δ.

La especificación de ese único componente del sistema sería, en el Nivel Java, algo como:

[1]Los primeros 5 dan cuenta del nombre del lenguaje GLIDER. Note el lector que Galatea también es un acrónimo, sólo que seleccionado para rendir tributo al género femenino como nos enseñaron varios maestros de la lengua española.

Código 4 Clase Delta

```
package demos.Grano;
import galatea.*; import galatea.glider.*;
public class Delta extends Node {
    /** Creates new Delta */
    public Delta() {
        super("Delta", 'A');
        Glider.nodesl.add(this);
    }
    /** una funcion cualquiera para describir un fenomeno */
    double f(double x) {
        return (Math.cos(x/5) + Math.sin(x/7) + 2) * 50 / 4;
    }
    /** funcion de activacion del nodo */
    public boolean fact(){
        Modelo.variableDep = f((double)Modelo.variableInd);
        Modelo.variableInd++;
        it(1);
        return true; }
}
```

Como el lector reconocerá (seguramente luego de leer el capítulo 1), ese código contiene la especificación de una clase Java, `Delta`, del paquete `demos.Grano`, que acompaña la distribución Galatea (como código libre y abierto).

Noten que esta clase "importa" (usa) servicios de los paquetes `galatea` y `galatea.glider`. El primero de estos paquetes, como dijimos en el capítulo 1, contiene servicios genéricos de toda la plataforma (como la clase `List`). Ese paquete no se usa en este ejemplo tan simple. Pero seguramente será importante en ejemplos más complejos.

El segundo paquete almacena toda la colección de objetos, clases necesarios para que la plataforma Galatea ofrezca los mismos servicios que el tradicional compilador Glider, incluyendo el núcleo del simulador (en la clase `galatea.glider.Glider`).

Los tres métodos que se incluyen en la clase `Delta` son paradigmáticos (es decir, un buen ejemplo de lo que suelen contener esas clases).

El primero es el constructor. Noten la invocación al constructor de la superclase `super("Delta", 'A')`. Los parámetros corresponden a un nombre para la "familia" de nodos y, el segundo, a un caracter que identifica el tipo de nodo en la semántica Glider[2]. Este es, por tanto, un nodo autónomo lo que, por el momento, significa que es un nodo que no envia, ni recibe mensajes de otros nodos.

[2]Decimos "familia" cuando estrictamente deberíamos decir tipo. Esto para evitar una confusión con el uso de la palabra tipo que sigue. Lo que está ocurriendo es que tenemos tipos de nodos en Glider, tipos de objetos, clases en Java y, por tanto, tipos de tipos de nodos: las clases Java que definen un tipo de nodo Glider para un modelo particular. Aquí se abre la posibilidad de compartir tipos de tipos de nodos entre varios modelos computacionales. El principio que ha inspirado la **modeloteca** y que se explicará más adelante

La segunda instrucción del constructor agrega el nuevo nodo a la lista de todos los nodos del sistema.

El método con el nombre `f` es simplemente una implementación auxiliar que podría corresponder a la especificación matemática de la función de transición del sistema en cuestión.

Finalmente en esta clase `Delta`, el método `fact()` contiene el código que será ejecutado cada vez que el nodo se active. Es decir, en el caso de los nodos autónomos, cada vez que se quiera reflejar un cambio en el componente que ese nodo representa.

Este código de fact merece atención cuidadosa. La instrucción:

```
Modelo.variableDep = f((double)Modelo.variableInd);
```

invoca a la nuestra función f para asignar lo que produzca a una variable de un objeto que no hemos definido aún. Se trata de un objeto anónimo asociado a la clase `Modelo` que es aquella clase principal de nuestro modelo computacional. Las variables globales de nuestro modelos son declaradas como atributos de esta clase que, desde luego, puede llamarse como el modelista prefiera. En la siguiente sección ampliaremos los detalles sobre `Modelo`. Noten que el argumento de entrada de `f`, es otra variable atributo de la misma clase.

La siguiente instrucción incrementa el valor de aquella variable de entrada, en 1 (Ver el manual de Java para entender esta sintaxis extraña. x++ es equivalente a x = x + 1).

Para efectos de la simulación, sin embargo, la instrucción más importante es:

```
it(1);
```

Se trata de una invocación al método `it` de la clase `Node` (y por tanto de su subclase `Delta`) cuyo efecto es el de reprogramar la ejecución de este código (el de la `fact`) para que ocurra nuevamente dentro de `1` unidad de tiempo. En simulación decimos que el tiempo entre llegadas (*interarrival time*, **it**) de estos *eventos de activación* del nodo `Delta` es de 1. Veremos en capítulos posteriores que estos eventos y su manejo son la clave del funcionamiento del simulador.

2.1.2 El esqueleto de un modelo Galatea en Java

Como el lector sabrá deducir, necesitamos por el momento, por lo menos dos clases para nuestro modelo computacional: `Delta`, con la función de transición del único nodo y `Modelo`, con las variables y el programa principal. Vamos a agregar la clase `Analisis`, que explicamos a continuación, con lo cual nuestro modelo queda compuesto por tres clases:

`Delta` Como ya hemos explicado, representa a todo el sistema e implementa su única función de transición.

`Analisis` Es un nodo auxiliar para realizar análisis sobre el sistema simulado mientras se le simula. Se explica a continuación.

`Modelo` Es la clase principal del modelo. Contiene la variables globales y el método `main()` desde el que se inicia la simulación. También se muestra a continuación.

Esta es la clase `Analisis`:

Código 5 Clase Analisis

```
package demos.Grano;
import galatea.*; import galatea.glider.*;
public class Analisis extends Node {
    /** Creates new Analisis */
    public Analisis() {
        super("Analisis",'A');
        Glider.nodesl.add(this);
    }
    public void paint() {
        for (int x = 0; x < (int) Modelo.variableDep ; x++) {
            System.out.print('.');
        };
        System.out.println('x');
    }
    /** funcion de activacion del nodo */
    public boolean fact(){
        paint();
        it(1);
        return true; }
}
```

Como habrán notado, su estructura es muy parecida a la de la clase `Demo`. No hay atributos de clase (aunque podrían incluirse (No hay ninguna restricción al respecto) y se cuentan 3 métodos. Aparecen el constructor y `fact()`, como en `Demo`. La diferencia está en `paint()`. Pero este no es más que un mecanismo para visualizar, dibujando una gráfica en la pantalla del computador (con puntos y x's), el valor de una variable.

`Analisis` es también un nodo autónomo que quizás no corresponda a ningún componente del sistema simulador, pero es parte de los dispositivos para hacer seguimiento a la simulación. Por eso se le incluye en este ejemplo. El lector debe notar, como detalle crucial, que la ejecución de los códigos `fact()` de `Demo` y `Analisis` se programan concurrentemente. Concurrencia es otro concepto clave en simulación, como veremos luego.

2.1.3 El método principal del simulador

Estamos listos ahora para conocer el código de la clase `Modelo`:

Código 6 Clase Modelo

```
package demos.Grano;
import galatea.*; import galatea.glider.*;
public class Modelo {
    /** Variables globales a todo el sistema se declaran aqui */
    public static double variableDep = 0d ; // esta es una variable dependiente.
    public static int variableInd = 0 ; // variable independiente.
    /** No hace falta, pues usaremos una sola instancia "estatica"*/
    public Modelo() {
    }
    /**
     * Este es el programa principal o locus de control de la simulacion
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        /** La red de nodos se inicializa aqui */
        Delta delta = new Delta();
        Analisis analisis = new Analisis();
        Glider.setTitle("Un modelo computacional muy simple");
        Glider.setTsim(50);
        Glider.setOutf(5);
        // Traza de la simulacion en archivo
        Glider.trace("Modelo.trc");
        // programa el primer evento
        Glider.act(delta,0);
        Glider.act(analisis,1);
        // Procesamiento de la red
        Glider.process();
        // Estadisticas de los nodos en archivo
        Glider.stat("Modelo.sta");
    }
}
```

Quizás el detalle más importante a destacar en esa clase es el uso del modificador Java `static`, en la declaración (la firma decimos en la comunidad OO) de los atributos y métodos de la clase. El efecto inmediato de la palabra static es, en el caso de los atributos, es que los convierte en **variables de clase**. Esto significa que "existen" (Se reserva espacio para ellos y se les puede direccionar, pues su valor permanece, es estático) existan o no instancias de esa clase. No hay que crear un objeto para usarlos.

Esto es lo que nos permite en el simulador usarlos como variables de globales de la simulación. Lo único que tiene que hacer el modelista es recordar el nombre de su clase principal y podrá acceder a sus variables. Esto hemos hecho con `Modelo.variableDep` y `Modelo.variableInd`.

El lector habrá notado que el `static` también aparece al frente del `main()`. Es obligatorio usarlo en el caso del `main()`, pero se puede usar con cualquier método para convertirlo en un método de clase. Como con las variables de clase, un método de clase puede ser invocado sin crear un objeto de esa clase. Simplemente se le invoca usando el nombre de la clase. En Galatea usamos mucho ese recurso. De hecho, todas las instrucciones en el código de `main()` que comienzan con `Glider.` son invocaciones a métodos de clase (declarados con `static` en la clase `galatea.glider.Glider`). Veamos cada uno de esos:

`Glider.setTitle(Ün modelo computacional muy simple");` Le asigna un título al modelo que será incluido en los reportes de salida.

`Glider.setTsim(50);` Fija el tiempo de simulación, en este caso en 50.

`Glider.setOutf(5)` Fija el número de decimales en las salidas numéricas.

`Glider.trace("Modelo.trc");` Designa el archivo `Modelo.trc` para que se coloque en él la traza que se genera mientras el simulador corre. Es una fuente muy importante de información a la que dedicaremos más espacio posteriormente. El nombre del archivo es, desde luego, elección del modelista. El archivo será colocado en el directorio desde donde se ejecute el simulador[3]

`Glider.act(delta,0);` Programa la primera activación del nodo `delta` (el objeto que representa el nodo `delta` del sistema simulado. Aclaramos esto un poco más adelante) para que ocurra en el tiempo `0` de simulación. Las activaciones posteriores, como vimos, son programadas desde el propio nodo `delta`, con la invocación `it(1)`.

`Glider.act(analisis,1);` Programa la primera activación del nodo `analisis`. Luego del tiempo `1`, y puesto que `analisis` también invoca a `it(1)`, ambos nodos se activarán en cada instante de simulación. Esta es la primera aproximación a la concurrencia en simulación.

`Glider.process();` Comienza la simulación.

`Glider.stat("Modelo.sta");` Designa al archivo

Solamente nos resta comentar este fragmento de código:

```
/** La red de nodos se inicializa aqui */
Delta delta = new Delta();
Analisis analisis = new Analisis();
```

[3]Cuidado con esto. Si usa un ambiente de desarrollo como el Sun One Studio, este colocará esa salida en uno de sus directorios.

En estas líneas se crean los objetos que representan, en la simulación, los componentes del sistema simulado. Ya explicamos que el único componente "real" es `delta`, pero `analisis` también debe ser creado aquí aunque solo se trate de un objeto de visualización.

Note el lector que podríamos usar clases con elementos `static` para crear esos componentes. Pero quizás lo más trascendental de poder modelar al Nivel Java es precisamente la posibilidad de crear y destruir componentes del sistema simulado durante la simulación, usando el código del mismo modelo. Esto no es posible en el viejo Glider. Será siempre posible en el Nivel Glider de Galatea, pero sólo porque mantendremos el acceso directo al entorno OO de bajo nivel (Java seguirá siendo el lenguaje matriz).

Esa posibilidad de crear y destruir nodos es importante porque puede ser usada para reflejar el **cambio estructural** que suele ocurrir en los sistemas reales. Este concepto se ha convertido en todo un proyecto de investigación para la comunidad de simulación, pues resulta difícil de acomodar en las plataformas tradicionales.

2.1.4 Cómo simular con Galatea, primera aproximación

Hemos concluido la presentación del código de nuestro primero modelo de simulación Galatea. Para simular debemos 1) compilar los códigos Java a `.class` (con el `javac`, como se explica en el capítulo 1) y ejecutar con la máquina virtual (por ejemplo con `java`, pero también podría ser con `appletviewer` o con un Navegador y una página web, si convertimos el modelo en un applet).

La figura 2.1 muestra la salida que produce nuestro modelo en un terminal de texto (un *shell* Unix o Windows).

No es un gráfico muy sofisticado, pero muestra el cambio de estado del sistema a lo largo del tiempo, con un mínimo de esfuerzo. Mucha más información útil se encontrará en los archivos de traza (`Modelos.trc`) y de estadísticas (`Modelo.sta`). Hablamos sobre ese tipo de salida más adelante en el libro.

Dejamos hasta aquí, sin haber conocido demasiado, la simulación tradicional. En la siguiente parte del capítulo conoceremos un modelo de simulación multi-agente.

2.2 El primer modelo computacional multi-agente

En esta parte del tutorial usamos elementos fundamentales de lo que se conoce como la tecnología de agentes inteligentes. En beneficio de una primera lección sucinta, no entraremos en detalles conceptuales. Sin embargo, un poco de contexto es esencial.

La ingeniería de sistemas basados en agentes (*Agent-Based Modelling*, ABM, como se le suele llamar) es un desarrollo reciente en las ciencias computacionales que promete una revolución similar a la causada por la orientación por objetos. Los agentes

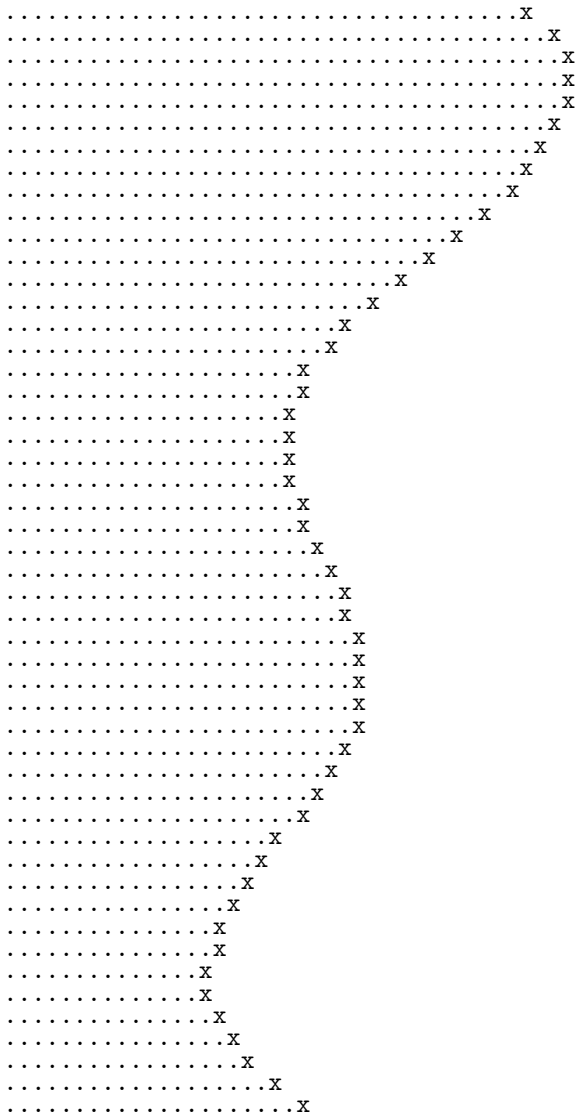

Figura 2.1: Salida del modelo en un terminal

a los que se refiere son productos de la Inteligencia Artificial, IA, [4], una disciplina tecnológica que ha sufrido un cambio de enfoque en los últimos años: los investigadores de la antigua (pero buena) Inteligencia Artificial comenzaron a reconocer que sus esfuerzos por modelar un dispositivo inteligente carecían de un compromiso sistemático (como ocurre siempre en ingeniería) con la posibilidad de que ese dispositivo "hiciera algo". Ese "hace algo inteligentemente" es la característica esencial de los agentes (inteligentes) tras la que se lanza el proyecto de IA aproximadamente desde la década de los noventa del siglo pasado.

El poder modelar un dispositivo que puede hacer algo (inteligente) y el enfoque modular que supone concebir un sistema como constituido por agentes (algo similar a concebirlo constituido por objetos) hacen de esta tecnología una herramienta muy efectiva para atacar problemas complejos [7], como la gestión del conocimiento y, en particular, el modelado y simulación de sistemas [8].

La noción de agente tiene, no obstante, profundas raíces en otras disciplinas. Filósofos, psicólogos y economistas ha propuesto modelos y teorías para explicar qué es un agente, pretendiendo explicar al mismo tiempo a todos los seres humanos. Muchas de esas explicaciones refieren como es que funcionamos en una suerte de ciclo en el que observamos nuestro entorno, razonamos al respecto y actuamos, procurando atender a lo que hemos percibido, pero también a nuestras propias creencias e intenciones. Ese ciclo en cada individuo, lo convierte en causante de cambios en el entorno o ambiente compartido normalmente con otros agentes, pero que puede tener también su propia disciplina de cambio. Algunos de los cambios del ambiente son el efecto sinergético de la acción combinada de dos o más agentes.

Esas explicaciones generales han inspirado descripciones lógicas de lo que es un agente y una sociedad de agentes. En el resto del libro revisamos algunas de ellos, las mismas que usamos como especificaciones para el diseño de Galatea. No hay, sin embargo, ningún compromiso de exclusividad con ninguna de ella y es muy posible que Galatea siga creciendo con la inclusión de nuevos elementos de la tecnología de agentes.

2.2.1 Un problema de sistemas multi-agentes

Biocomplejidad es el término seleccionado por la *National Science Foundation* de los EEUU para referencirse a "el fenómeno resultante de la interacción entre los componentes biológicos, físicos y sociales de los diversos sistemas ambientales de la tierra". Modelar sistemas con esas características es una tarea difícil que puede simplificarse con el marco conceptual y las plataformas de sistemas multi-agentes.

El ejemplo que mostramos en esta sección es una versión simplificada de 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. Galatea está siendo usada para construir modelos cada vez más complejos de la dinámica apreciable en la Reserva Forestal de Caparo[4], ubicada en los llanos de Barinas, al Sur-Occidente de Venezuela.

Hemos simplificado el más elemental de esos modelos juguetes[5] y lo hemos convertido en un **juego de simulación**, en el que el **simulista** debe tomar el lugar del ambiente. Los agentes están implementados en una versión especial del paquete `galatea.gloria` y que incluimos en el paquete `galatea.gloriosa`[6].

El resultado es un modelo en el que el simulista es invitado a llevar un registro manual de la evolución de una Reserva Forestal (inventando sus propias reglas, con algunas sugerencias claro), mientras sobre ese ambiente actúa un conjunto de agentes artificiales que representan a los actores humanos de la Reserva.

Nuestra intención con el juego es familiarizar a los lectores con el desarrollo de un modelo multi-agente, sin que tengan que sufrir sino una pequeña indicación de la complejidad intrínseca de estos sistemas.

2.2.2 Conceptos básicos de modelado multi-agente

Siguiendo la práctica habitual en el modelado de sistemas, para crear el *modelo socio-económico-biológico* de la Reserva Forestal de CAPARO, construimos las descripciones a partir de un conjunto de conceptos básicos. Esos conceptos son: estado del sistema, dinámica, proceso y restricciones contextuales.

El estado del sistema es la lista de variables o atributos que describen al sistema en un instante de tiempo dado. Es exactamente equivalente a lo que en simulación se suelen llamar variables de estado. Pero también puede ser visto como una lista de atributos del sistema que cambian a lo largo del tiempo y que no necesariamente corresponde a magnitudes medibles en números reales. Los juicios de valor (cierto o falso) también se pueden incluir entre los atributos. En virtud de su naturaleza cambiante, nos referiremos a esos atributos como **fluentes** o **propiedades** del sistema que cambian al transcurrir el tiempo. Ya sabemos que, en Galatea, corresponderán a atributos de algún objeto.

Una dinámica es una disciplina, normalmente modelada como una función matemática del tiempo, describiendo como cambia una o varias de esas variables de estados (atributos o fluentes). Noten, sin embargo, que esas funciones del tiempo no necesariamente tienen que ser sujetas a otros tratamientos matemáticos. Para describir agentes, por ejemplo, puede ser preciso recurrir a funciones discretas (no derivables) parciales sobre el tiempo.

[4]Detalles en `http://cesimo.ing.ula.ve/INVESTIGACION/PROYECTOS/BIOCOMPLEXITY/`

[5]Programado originalmente por Niandry Moreno y Raquel Quintero.

[6]adaptado al caso por Niandry Moreno, Raquel Quintero y Jacinto Dávila.

Un proceso es la conflagración de una o más de esas dinámicas en un arreglo complejo donde se afectan unas a otras. Es decir, el progreso de una dinámica al pasar el tiempo está restringido por el progreso de otra dinámica, en cuanto a que sus trayectorias (en el sentido de [9] están limitadas a ciertos conjuntos particulares (de trayectorias posibles).

Una restricción contextual es una limitación sobre los valores de las variables de estado o sobre las dinámicas que pueden estar involucradas en los procesos. Es decir, es una especificación de cuáles valores o dinámicas NO pueden ser parte del sistema modelado.

Le pedimos al lector que relea estas definiciones. Las usaremos a continuación en la medida en que conocemos el modelo juguete el cual está compuesto por:

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

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

La clase `Modelo` que, como antes, contiene el programa principal y las variables globales del simulador.

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

Con este modelo computacional queremos caracterizar el proceso en la Reserva Forestal que está siendo ocupada por Colonos con distintos planes de ocupación. La pregunta principal del proyecto es si existe una combinación de planes de ocupación que pudiese considerarse como intervención sustentable sobre la reserva, dado que no compromete la supervivencia de sus biodiversidad.

En nuestro ejemplo, estaremos muy lejos de poder abordar esa pregunta, pero confiamos que el lector encuentre el juego interesante e iluminador en esa dirección.

2.2.3 El modelo del ambiente

Llamamos `Delta` a la clase que implementa el modelo del ambiente. En este modelo juego, `Delta` es solo una colección de rutinas para que el simulador interactúe con el simulista y sea este quien registre el estado del sistema.

El propósito de esto es meramente instruccional. El simulista tendrá la oportunidad de identificar los detalles operativos de la simulación y no tendrá problemas de automatizar ese registro en otra ocasión.

Sin embargo, para no dejar al simulista desválido frente a la complejidad de un ambiente natural, incluimos las siguientes recomendaciones y explicaciones:

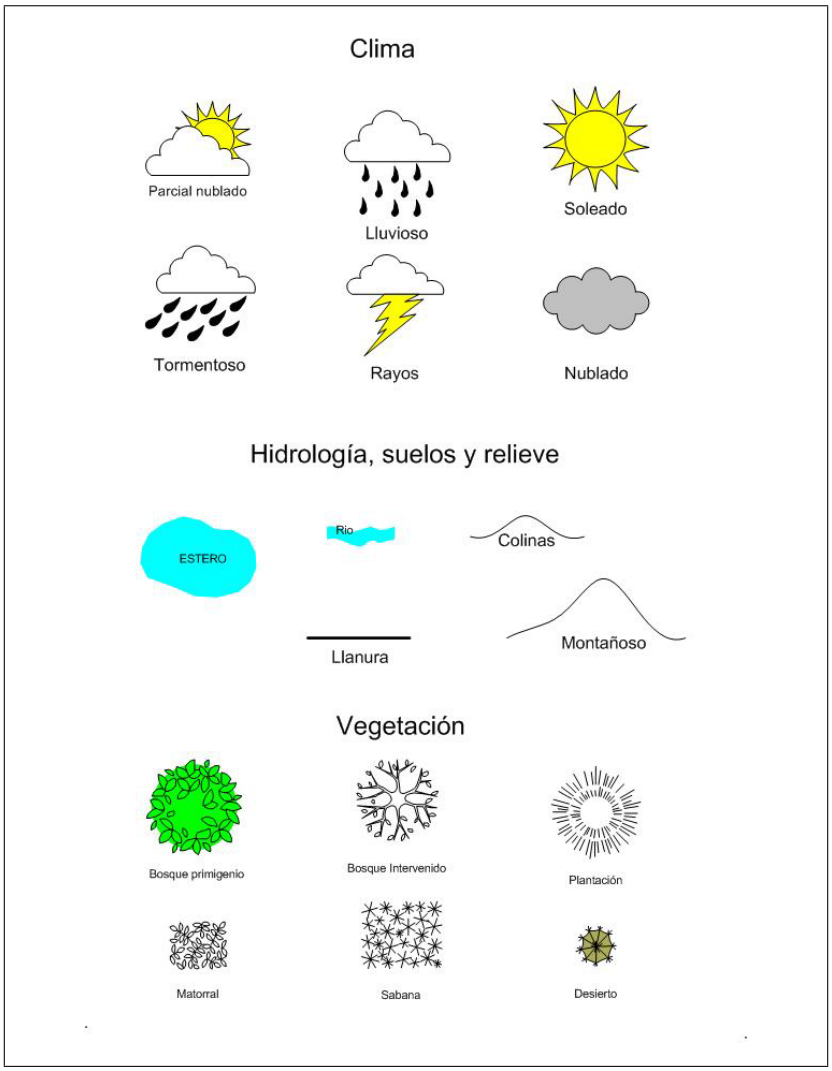

Figura 2.2: Símbolos de estado en un ambiente natural

Una reserva forestal es un ambiente natural sumamente complejo. Cualquier descripción dejará fuera elementos importantes. Con eso en mente, nos atrevemos a decir que son 3 los procesos macro en la reserva

forestal: 1) clima, 2) hidrología y relieve, con la dinámica de suelos y 3) vegetación que siempre será un resumen de la extraordinaria colección de dinámicas de crecimiento, reproducción y muerte de la capa vegetal.

Noten que estamos dejando de lado, expresamente, el proceso de la población animal, salvo, claro, por aquello que modelaremos acerca de los humanos, usando agentes. Sin embargo, el efecto del proceso animal debe ser tomando en cuenta en el proceso vegetación, particularmente el consumo de vegetación por parte del ganado que puede ser devastador para la reserva.

Para ayudar a nuestro simulista a llevar un registro de esos procesos y sus dinámicas, le sugerimos llevar anotaciones en papel del estado global de la reserva a lo largo del tiempo. Quizás sobre un mapa de la reserva, con la escala adecuada (al tiempo disponible para jugar), se pueda llevar ese registro mejor usando un código de símbolos como el que mostramos en la figura 2.2[7]

Los símbolos de clima son suficientemente claros. Los de hidrología y relieve son un poco más difíciles de interpretar, pero como cambian con poca frecuencia no debería haber mucho problema. La dinámica de cambio más compleja será la de la vegetación.

Lo que mostramos en la lámina es una idea de los tipos de cobertura vegetal. El más apreciado en una reserva de biodiversidad forestal es el que llamamos bosque primigenio, no tocado por la acción humana. Una vez que ha sido intervenido se convierte en bosque intervenido (y nunca más será primigenio). Pero los espacios naturales también pueden ser usados para plantaciones de especies explotables por su madera (como las coníferas que se ilustran con la figura). Los matorrales son, normalmente, bosques que han sido intervenidos y luego de ser abandonados (por los humanos y sus animales), han prosperado por su cuenta. El otro uso del espacio que es crucial, sobretodo para la ganadería, es la sabana. Finalmente, cuando el espacio es devastado sin posibilidad de recuperación se crea un desierto irrecuperable. Nuestro simulista querrá aprovechar estas indicaciones para modelar sus propias reglas de **cambio de uso de la tierra**.

El único detalle adicional a tener presente es la escala de tiempo en el que ocurren los cambios (por lo menos para la observación superficial). En el caso del clima, todos sabemos, es de minutos, horas, días y meses. En el caso de vegetación es de meses y años. En el caso de hidrología y

[7]El simulista bien podría fotocopiar esa imagen y recortar los símbolos para marcar el mapa, como solían hacer los cartógrafos (ahora lo hacen las máquinas, se admite, pero no es igual de divertido).

suelos, va desde años (con las crecidas e inundaciones estacionales) hasta milenios.

Ahora podemos discutir el código de `Delta`:

Comentaremos sobre tres fragmentos de ese código en detalle:

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

Esta pieza de código crea los agentes de esta simulación. Suele colocarse en el `main()` de `Modelo`, pero puede aparecer, como en este caso, en otro lugar, siempre que se invoque al comienzo de la simulación (en este caso, al crear el ambiente).

Note la invocación a `GInterface.agentList.add(agente[i])` con la que se registra a cada agente en una base de datos central para el simulador, gestionada por la clase GInterface del paquete galatea.hla. Así es como el simulador conoce a todos los agentes de la simulación.

El siguiente fragmento es puro código Java. Es la manera de leer desde el teclado y poder identificar las respuestas del usuario. El método `si`, incluido en el código simplemente lee el teclado y si el texto leído comienzo con el caracter `'S'`, devuelve **true**. Si la respuesta no es afirmativa, el simulador termina su ejecución con la invocación a `System.exit(0)`.

```
InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(ir);
if (!si(in)) {
    System.out.println("Ambiente: Fin del juego!");
    System.exit(0);
}
```

El último fragmento es sumamente importante para el simulador multi-agente:

```
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();
}
```

Allí se le dice a cada agente que hora es, que cosas está observando a esta hora y se le pide que piense y decida que hacer (`cycle()`).

Los detalles del cómo funciona todo esto aparecen en el resto del libro. Por lo pronto, le pedimos al lector que recuerde que este es un modelo de tiempo discreto. Cada vez que el ambiente cambia, el mismo ambiente indica los cambios a los agentes y espera su respuesta.

Implementaciones en las que el ambiente y los ambientes corren "simultáneamente" también son posibles con Galatea. Pero requieren más esfuerzo de implementación, como mostraremos más adelante.

Código 7 Clase Delta

```
package demos.Bioc.toy;
import galatea.*;
import galatea.glider.*;
import galatea.hla.*;
import galatea.gloriosa.*;
import java.io.*;
public class Delta extends Node {
  /**this variable indicates the quantity of settler agent
   * at the forest reserve. */
  public int NUM_COLONOS=3;
  /**This is an array of references to settler agents.
   */
  public Colono[] agente= new Colono[NUM_COLONOS];
  /** Creates new Delta */
  public Delta() {
    super("Delta", 'A');
    Glider.nodesl.add(this);
    // .... Se informa al usuario el numero de agentes
    //Se crean las instancias de cada uno de los agentes colono
    //Esto deberia ir junto con la declaracion de la red en el programa
    //principal del modelo
    for (int i=0;i<NUM_COLONOS;i++){
      agente[i]=new Colono();
      agente[i].agentId= i+1;
      GInterface.agentList.add(agente[i]);
    }
  }
  /** funcion de activacion del nodo */
  public boolean fact(){
    it(1);
    try{
      System.out.println("--------------- Es el tiempo "+Glider.getTime());
      System.out.println("Ambiente: Responda a cada una de estas preguntas:");
      System.out.println(" 1.- Donde esta cada uno de los agentes?");
      System.out.println(" 2.- Como es el clima en cada lugar?");
      System.out.println(" 3.- Como se estan comportando los torrentes
            y el nivel freatico");
      System.out.println(" 4.- Que cambios tienen lugar en la vegetacion");
      System.out.println(" 5.- Que puede observar cada uno de los agentes");
      System.out.println(" Continuar?(S/N)");
      InputStreamReader ir = new InputStreamReader(System.in);
      BufferedReader in = new BufferedReader(ir);
      if (!si(in)) {
          System.out.println("Ambiente: Fin del juego!");
          System.exit(0);
      }
    } catch(Exception e) { };
    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();
    }
    return  true ;
  }
```

Código 8 Clase Delta continuación

```
  public void actualizarsensores(Colono agente){
    try {
      InputStreamReader ir = new InputStreamReader(System.in);
      BufferedReader in = new BufferedReader(ir);
      System.out.println("Ambiente: Hablemos del agente: "+agente);
      // La entrevista
      System.out.println("Ambiente: Se establecio ya? (S/N)");
      if (!si(in)) {
          agente.inputs.add("No establecido");
      }
      // La entrevista 2
      System.out.println("Ambiente: Ve una celda desocupada? (S/N)");
      if (si(in)) {
          agente.inputs.add("Celda desocupada");
      }
      // La entrevista 3
      System.out.println("Ambiente: Ve una celda apta para
            establecerse? (S/N)");
      if (si(in)) {
          agente.inputs.add("Celda apta para establecerse");
      }
      // La entrevista 4
      System.out.println("Ambiente: Ve una celda apta para sembrar? (S/N)");
      if (si(in)) {
          agente.inputs.add("Celda apta para sembrar");
      }
      // La entrevista 5
      System.out.println("Ambiente: Se agotaron los recursos
            en su parcela? (S/N)");
      if (si(in)) {
          agente.inputs.add("Se agotaron los recursos");
      }
    } catch (Exception e) { };
    System.out.println("Ambiente: Este agente observara "+agente.inputs);
  }
  public boolean si(BufferedReader in) {
    int c, i;
    int resp [] = new int[10];
    i = 0;
    try {
        while ((c=in.read()) != -1 && c!=10) { resp[i]=c ;}
    } catch (Exception e) {};
    return (char) resp[0]=='S' ;
  }
}
```

2.2.4 El modelo de cada agente

En Galatea, un agente es un objeto con una sofisticada estructura interna. Considere este código[8]:

De nuevo concentraremos la atención, esta vez en 2 piezas del código:

El método `init()` es una muestra de lo que tenemos que hacer cuando queremos fijar las metas del agente desde el principio. Un agente tiene metas, es decir, las reglas que determinan lo que querrá hacer.

```
public void init(){
    outputs = new LOutputs();
    //finding a place(0)
    addPermanentGoal("buscarsitio",0,new String[]{"No establecido"});
    //settling down(1)
    addPermanentGoal("establecerse",1,new String[]{"Celda desocupada",
        "Celda apta para establecerse"});
    //cleaning the land and seeding agriculture of subsistence(2)
    addPermanentGoal("limpiarsembrar",2,new String[]{"Celda desocupada",
        "Celda apta para sembrar"});
    //deforesting and selling wood to illegal traders(3)
    addPermanentGoal("talarvender",3,new String[]{"Celda desocupada",
        "Celda con madera comercial"});
    //moving(4)
    addPermanentGoal("mudarse",4,new String[]{"Se agotaron los recursos"});
    //expanding settler's farms(5)
    addPermanentGoal("expandirse",5,new String[]{"Celda desocupada",
        "Celda apta para expandir"});
}
```

En este método, luego de crear la lista que guardará las salidas que este agente envía a su ambiente, se le asignan (al agente) sus metas permanentes (es decir, sus metas durante la simulación).

El formato de invocación de `addPermanentGoal` es, básicamente, una regla de **si** condiciones **entonces** acción. Por ejemplo, para decirle al agente que **si** *no se ha establecido*, **entonces** debe *buscar un sitio* para hacerlo, escribimos:

```
addPermanentGoal("buscarsitio",0,
    new String[]{"No establecido"});
```

La cadena "No establecido" será una entrada desde el ambiente para el agente (como se puede verificar en la sección anterior). "buscarsitio" es el nombre de uno de sus métodos, definido en el código que sigue y que prepara las **salidas** del agente para que las **ejecute** el ambiente como **las influencias de ese agente**.

[8]Por claridad, hemos retirado todos los comentarios en extenso. Estarán en el software que acompaña el libro.

Código 9 Clase Colono

```
package demos.Bioc.toy;
import galatea.glider.*; import galatea.gloriosa.*;
importgalatea.hla.*;
public class Colono extends Ag{
 // posicion X, Y del agente.
 int x=-1;
 int y=-1;
 // Los ahorros del agente.
 int recursos_economicos;
 // Los terrenos que poseee
 int num_celdas=0;//numero de celdas que posee
 int MAX_NUM_CELDAS=9;
 int[][] propiedad=new int[MAX_NUM_CELDAS][2];
 // tiempo de residencia
 int tr; //tiempo en la region invadida.
 //----fin de atributos
 public Colono(){
   super(6,"colono");
   this.population++;
   this.agentId=population;
   init();
 }
public void init(){
    outputs = new LOutputs();
    //finding a place(0)
    addPermanentGoal("buscarsitio",0,new String[]{"No establecido"});
    //settling down(1)
    addPermanentGoal("establecerse",1,new String[]{"Celda desocupada",
        "Celda apta para establecerse"});
    //cleaning the land and seeding agriculture of subsistence(2)
    addPermanentGoal("limpiarsembrar",2,new String[]{"Celda desocupada",
        "Celda apta para sembrar"});
    //deforesting and selling wood to illegal traders(3)
    addPermanentGoal("talarvender",3,new String[]{"Celda desocupada",
        "Celda con madera comercial"});
    //moving(4)
    addPermanentGoal("mudarse",4,new String[]{"Se agotaron los recursos"});
    //expanding settler's farms(5)
    addPermanentGoal("expandirse",5,new String[]{"Celda desocupada",
        "Celda apta para expandir"});
}
```

Código 10 Clase Colono continuación

```
public void buscarsitio(){
  Object[] args=new Object[1];
  args[0]=this;
   //Cargar los otros atributos necesarios en el arreglo args.
   //args.add(x);
   //args.add(y);
  Output o= new Output("buscarsitio",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
  buscarsitio()");
} public void establecerse(){
  Object[] args=new Object [] { this };
  Output o= new Output("establecerse",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
  establecerse()");
} public void limpiarsembrar(){
  Object[] args=new Object [] { this };
  Output o= new Output("limpiarsembrar",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
limpiarsembrar()"); } public void talarvender(){
  Object[] args=new Object [] { this };
  Output o= new Output("talarvender",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
talarvender()"); } public void mudarse(){
  Object[] args=new Object[1];
  args[0]=this;
  //Cargar los otros atributos necesarios en el arreglo args.
  //args.add(x);
  //args.add(y);
  Output o= new Output("mudarse",args); outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
mudarse()"); } public void expandirse(){
  Object[] args=new Object [] { this };
  Output o= new Output("expandirse",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
expandirse()"); } }
```

```
public void buscarsitio(){
  Object[] args=new Object[1];
  args[0]=this;
   //Cargar los otros atributos necesarios en el arreglo args.
   //args.add(x);
   //args.add(y);
  Output o= new Output("buscarsitio",args);
  outputs.add(o);
  System.out.println("Agent: " +this.agentId+" tratara de
  buscarsitio()"); }
```

Hay razones precisas para esta forma del objeto `Output` que tendrán un poco más de sentido en la próxima sección.

2.2.5 La interfaz Galatea: primera aproximación

La clase Interfaz es sumamente difícil de explicar en este punto, sin el soporte conceptual de las influencias. Baste decir que se trata de los métodos que ejecuta el ambiente como respuesta a las salidas de los agentes. Los agentes le dicen al ambiente que quieren hacer, con sus salidas, y el ambiente responde ejecutando estos métodos.

Es por ello que, en el código que sigue, el lector podrá ver métodos con los mismos nombres que usamos para crear los objetos `Output` de cada agente. La diferencia, decimos en Galatea, es que aquellos métodos de la sección anterior son ejecutados por el propio agente. Los que se muestran a continuación son cargados (en tiempo de simulación) y ejecutados por el simulador desde el objeto `GInterface` del paquete `galatea.hla`.

Los métodos en este ejemplos son sólo muestras. No hacen salvo indicar al simulista que la acción se ha realizado (con lo que el simulista deberá anotar algún cambio en su registro del ambiente). Pero, desde luego, el modelista puede colocar aquí, cualquier código Java que implemente los cambios automáticos apropiados para el sistema que pretende simular.

Noten, por favor, la lista de argumentos de estos métodos. Siempre aparecen un `double` y un `Agent` como primeros argumentos. Esta es una convención Galatea y la usamos para transferir el tiempo actual (en el `double`) y una referencia al objeto agente que ejecuta la acción (en `Agent`). Los siguiente argumentos serán aquellos que el modelista desee agregar desde sus `Output`s. Noten como, a modo ilustrativo, en este caso enviamos el objeto Colono como tercer argumento de algunos de los métodos.

Esta explicación, desde luego, tendría que ser suplementada con ejericicios de prueba. Pueden usar el código que se anexa y consultar con sus autores.

Código 11 Clase Interfaz

```
package demos.Bioc.toy;
import galatea.*; import galatea.glider.*; import galatea.hla.*;
import galatea.gloriosa.*; import java.io.*;
public class Interfaz {
   //-----------------------------------------------
   /**The settler agent needs to find a place inside the forest
    * reserve to settle down. */
   public void buscarsitio(double t, Agent a, Colono agente) {
       System.out.println("Interface: El agent "+a.agentId+"
            busco un sitio y lo encontro.");
   }
   //-----------------------------------------------
   /**This procedure updates the cellular automata layers due to a
    * Colono invasion to a forest reserve cell. */
   public void establecerse(double t, Agent a, Colono agente){
       System.out.println("Interface: El agent "+a.agentId+"
            se ha establecido.");
   }
   //-----------------------------------------------
   /**The settler agent needs to expand its farms due to a diminution
    * of the land fertility. */
   public void expandirse(double t, Agent a, Colono agente) {
       System.out.println("Interface: El agent "+a.agentId+"
            expandio su parcela.");
   }
   //-----------------------------------------------
   /**In this procedure the Colono agent seeds the forest reserve cell.
    */
   public void limpiarsembrar(double t, Agent a, Colono agente) {
       System.out.println("Interface: El agent "+a.agentId+"
            limpio el terreno y sembro.");
   }
   //-----------------------------------------------
   /** The settler agent must move to another place inside the
    * forest reserve, an abandon the inicial occupied place. */
   public void mudarse(double t, Agent a, Colono agente) {
       System.out.println("Interface: El agent "+a.agentId+"
            ha abandonado su terreno.");
   }
   //-----------------------------------------------
   /**The settler agent has settle down in a forest that contains
    * species with a high commercial, then the settler must clean
    * the place and sell the wood to illegal traders. */
   public void talarvender(double t, Agent a, Colono agente) {
       System.out.println("Interface: El agent "+a.agentId+"
            talo y vendio la madera.");
   }
   //-----------------------------------------------
   public void haceAlgo(double m, Agent a) {
       System.out.println("Interface: El agent "+a.agentId+" hizo algo.");
   }
}
```

2.2.6 El programa principal del simulador multi-agente

El código de la clase principal Modelo tiene pocos cambios respecto al ejemplo anterior, pero hay uno muy importante:

```
package demos.Bioc.toy;

import galatea.*;
import galatea.glider.*;
import galatea.hla.*;
import galatea.gloriosa.*;

 /**
 *@author Niandry Moreno.
 *@version Juguete #1.
 */
public class Modelo {
   public static Delta ambiente = new Delta();
   /**Simulator. Main process. */
   public static void  main(String args[]) {
     GInterface.init("demos.Bioc.toy.Interfaz");
     Glider.setTsim(30);
     System.out.println("Inicio de simulacion");
     // Traza de la simulaci{\'o}n en archivo
     //Glider.trace("DemoCaparo.trc");
     Glider.act(ambiente,0);
     //Procesa la lista de eventos futuros.
     Glider.process();
     // Estadisticas de los nodos en archivo
     //Glider.stat("DemoCaparo.sta");
     System.out.println("La simulacion termino");
  }
}
```

El cambio importante es la identificación de la clase donde se almacenan los métodos de interfaz:

```
GInterface.init("demos.Bioc.toy.Interfaz");
```

Con este cambio, el sistema está listo para simular.

2.2.7 Cómo simular con Galatea. Un juego multi-agente

Como antes, simplemente compilar todos los `.java` y, luego, invocar al simulador:

```
java demos.Bioc.toy.Modelo
```

Disfrute su juego!.

2.3 Asuntos pendientes

No hemos hablado todavía de varios temas muy importantes para la simulación y realizables con Galatea. Galatea es una familia de lenguajes. Es más fácil describir reglas para los agentes en un lenguaje más cercano al humano, tales como:

```
si esta_lloviendo_a_las(T) entonces saque_paraguas_a_las(T).
```

Ese código puede ser incorporado a Galatea de inmediato via Prolog y más adelante, directamente en Java.

El otro tema especial es concurrencia. Java es multihebrado. Galatea también, pero no solamente gracias a la herencia Java. Tenemos previsiones para que una multiplicidad de agentes hebras se ejecuten al mismo tiempo que el simulador principal (y cualquier otra colección de hebras que disponga el modelista) en una simulación coherente.

Modelar para aprovechar Galatea de esa manera requiere un dominio un poco más profundo de la teoría de sistemas multi-agentes a la que se le dedican los siguientes capítulos.

Confiamos, no obstante, que el lector estará motivado, con ejemplos anteriores, para continuar la exploración de las posibilidades que ofrece la simulación de sistemas multi-agente.

Capítulo 3

Teoría de simulación de sistemas multi-agentes

En este capítulo creamos un marco teórico con los conceptos clave que soportan el diseño de la plataforma GALATEA: Primero, en la sección 3.1, presentamos un marco de trabajo y la metodología para el proceso de modelado y simulación de sistemas. En segundo lugar, en la sección 3.2 explicamos lo que entendemos por teoría de simulación de sistemas multi-agentes en el proyecto GALATEA.

3.1 Modelado y Simulación

La frase "modelado y simulación" designa el conjunto de actividades asociadas con la construcción de modelos de sistemas del mundo real y su simulación haciendo uso de computadores.

La figura 3.1 muestra cinco elementos básicos que están asociados a aspectos conceptualmente diferentes de las actividades de modelado y simulación de sistemas. Cuando se discute el modelado y la simulación se definen estos elementos y se establecen las relaciones entre ellos [10]:

El **sistema real** se refiere a la fuente observable de datos, la cual puede ser natural, artificial o una combinación de ambas. En esta etapa del modelado y simulación, la característica importante es la identificación de una porción de realidad y su distinción con el resto del sistema, permitiendo realizar observaciones y mediciones sobre dicha porción.

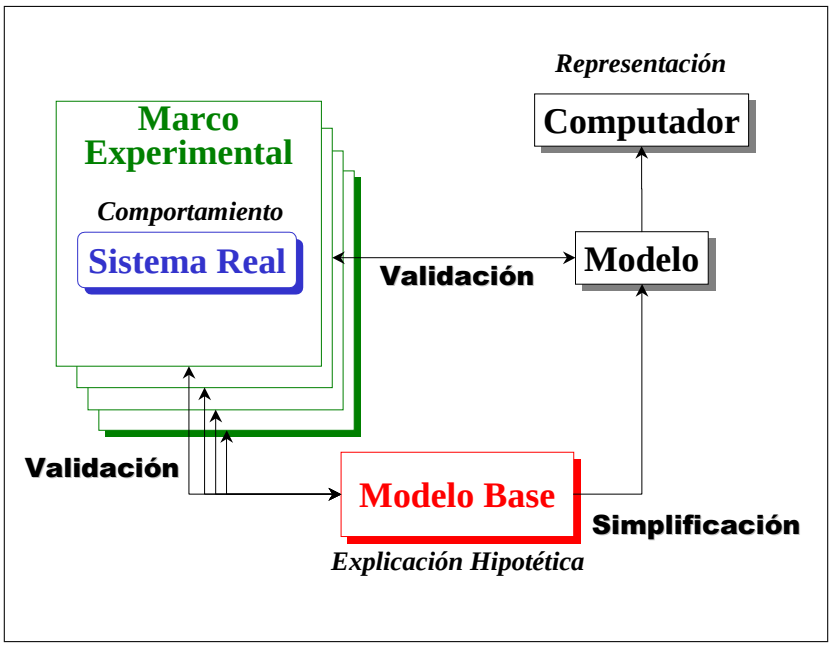

Figura 3.1: Elementos básicos del modelado y simulación de sistemas

El *comportamiento* del sistema real constituye todo aquello que podemos conocer directamente del mundo real. Este comportamiento es representado haciendo uso de variables descriptivas, nombres genéricos para atributos del sistema, las cuales son clasificadas como observables y no observables. Las variables observables son aquellas que corresponden directamente a mediciones, mientras que las variables no observables son aquellas que no pueden ser representadas directamente por una medida o juicio de valor.

Las variables observables se pueden clasificar como variables de entrada y de salida. Las variables de entrada permiten describir perturbaciones o influencias que afectan el sistema desde el exterior. Las variables de salida permiten representar los valores resultantes del proceso de simulación. En otras palabras, las variables de entrada son generalmente consideradas las "causas" y las variables de salida los "efectos".

El avance del tiempo representa uno de los conceptos básicos de simulación. Al agrupar las variables se obtiene el estado del sistema en el transcurso del tiempo. Cada intervalo de tiempo determina los *segmentos* o *trayectorias* del modelo y al conjunto de todos los posibles pares de segmentos entrada-salida obtenidos experimentalmente se les denomina el comportamiento del sistema real.

La validación de un modelo es el proceso de evaluar la validez de dicho modelo. Esta validez es relativa a un marco experimental y a los criterios que permitan calibrar las trayectorias. El **marco experimental** es la definición de un conjunto limitado

de circunstancias bajo las cuales se observa o se experimenta con el sistema real.

El **modelo base** es un modelo que resume el comportamiento del sistema real. Este modelo es válido en todos los marcos experimentales permitidos pues proporciona una *explicación hipotética* completa del comportamiento del sistema real.

En casos reales, la complejidad del modelo base no permite considerarlo para simulación. En estos casos la simplificación tiene como finalidad la definición de un **modelo** (agregado) a partir del modelo base. Este modelo debe ser válido con respecto a un marco experimental de interés.

Como último elemento tenemos el **computador**, el dispositivo de cálculo que ayuda a generar las trayectorias del modelo. Aunque en algunos casos estas trayectorias o sus propiedades pueden ser manipuladas en forma analítica, sin la ayuda del computador, generalmente se desea realizar la *representación* y la manipulación de estas trayectorias paso a paso, es decir, de un instante de tiempo al siguiente. Es a este proceso a lo que se le ha llamado *simulación*.

Las instrucciones para llevar a cabo el proceso de simulación paso a paso están contenidas en el modelo y el computador debe ejecutar correctamente estas instrucciones. El término modelo se refiere al código del programa escrito en el lenguaje utilizado para describir el sistema al simulador (en general, diferente al lenguaje en que se codifica el simulador).

Los motivos o fines que se persiguen con el modelado y la simulación de sistemas pueden verse según dos puntos de vistas diferentes: desde el punto de vista científico, se intenta entender como funciona un sistema, mientras que desde el punto de vista aplicativo lo interesante es notar o explicar que sucede en un sistema cuando se somete a ciertos cambios. Por ende, el modelado y simulación de sistemas constituyen una herramienta básica para la solución de problemas. Como herramienta la simulación ofrece múltiples ventajas:

1. Se apoya en una teoría matemática formal. Esto implica, entre muchas otras cosas, que es posible realizar simplificaciones sistemáticas de las representaciones de un sistema, preservando la estructura, a través de isomorfismos.

2. Es posible representar varios escenarios para el mismo problema, lo cual permite estudiar varias soluciones en forma simultanea.

3. El experimento permanece, no se destruye. Además, la historia generada puede ser extraída completamente.

4. Es posible repetir el experimento, la simulación, tantas veces como sean necesarias con relativa economía.

5. Es posible escalar el tiempo real del sistema haciendo que el tiempo dentro de la simulación transcurra más rápido o más lento que el del sistema real según convenga.

A continuación introduciremos algunos conceptos básicos de sistemas con la intención de presentar los formalismos utilizados comúnmente para la especificación de sistemas, haciendo énfasis en el formalismo de eventos discretos.

3.1.1 Formalismos para especificación de sistemas

La teoría de sistemas [10] proporciona una formalización matemática que permite representar en forma rigurosa los sistemas dinámicos. Esta formalización distingue entre la estructura de un sistema, su constitución propia, y su comportamiento o manifestaciones.

Los sistemas dinámicos se representan mediante una estructura de la forma:

$$S = (X, Y, \Omega, Q, \Delta, \Lambda, T)$$

X : Conjunto de Entradas.
Y : Conjunto de Salidas.
Ω : Conjunto de Segmentos de entrada posibles.
Q : Conjunto de Estados.
Δ : $(Q \times \Omega \to Q)$
Función de Transición de estados.
Λ : $(Q \times \Omega \to Y)$
Función de Salida.
T : Base Temporal. Posibles valores para el tiempo.

donde X, Y y Ω definen las entradas y salidas del sistema, mientras que Q y Δ representan la estructura interna y el comportamiento.

La función de transición de estados Δ debe garantizar que siempre se puede determinar el próximo estado del sistema a partir del estado actual, es decir debe ser un semigrupo.

Por otro lado, visto como una caja negra (figura 3.2) el comportamiento de un

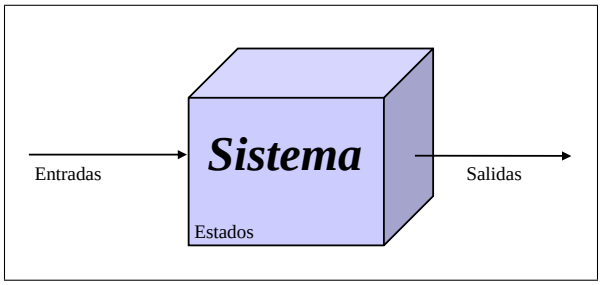

Figura 3.2: Concepto básico de sistemas

sistema es la relación existente entre las entradas y las salidas. Este comportamiento

se representa como una tupla de la forma $(entrada, salida)$ obtenida a partir del sistema real o del modelo. Así mismo, la estructura interna de un sistema incluye el estado del sistema, el mecanismo de transición de estados y el *"mapping"* hacia los estados de salida.

Conocida la estructura de un sistema es posible deducir, analizar o simular su comportamiento. Usualmente en la dirección opuesta (inferir la estructura a partir del comportamiento) no es posible realizar una deducción unívoca. Descifrar una representación válida a partir de un comportamiento observado es uno de los objetivos del modelado y la simulación de sistemas. En el momento en que se definen restricciones sobre las componentes que describen un sistema en el tiempo es posible derivar distintos formalismos del sistema como muestra la figura 3.3 [10, 11].

Tiempo\Variables	Finito	Discreto	Continuo
Discreto	Autómatas de estado finito	Máquinas de estado finito	Ecuaciones en diferencias
Continuo	Sistemas Asíncronos	Eventos Discretos	Ecuaciones Diferenciales

Figura 3.3: Clasificación de los modelos

Los formalismos más utilizados para la representación de sistemas se describen brevemente a continuación.

Ecuaciones Diferenciales (*DESS: Differential Equation Systems Specifications*).

Se emplean ecuaciones diferenciales ordinarias de primer orden para especificar la tasa de cambio de las variables de estado, las cuales generalmente se resuelven utilizando métodos numéricos. La representación de ecuaciones diferenciales mostrada en la figura 3.4 se expresa matemáticamente a través de una estructura de la forma:

$$DESS = (X, Y, \Omega, Q, f, \lambda, \mathcal{R})$$

X : Espacio vectorial ($\mathcal{R}^m$) de las variables de entrada.
Y : Espacio vectorial ($\mathcal{R}^p$) de las variables de salida.
Ω : Conjunto de todos los segmentos de entrada posibles. Estos deben ser contiguos a trozos.
Q : Espacio vectorial ($\mathcal{R}^n$) de estados accesibles del sistema.
f : $Q \times X \rightarrow Q$
Función de transición, representa las tasas de cambio.
λ : $Q \rightarrow Y \ \vee \ Q \times X \rightarrow Y$
Permite obtener la salida en el tiempo t.
$\mathcal{R}$: La base del tiempo son los números reales.

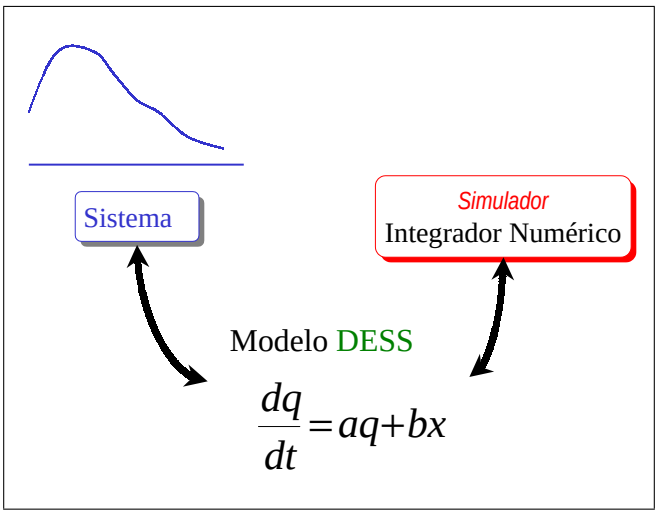

Figura 3.4: Formalismo de Ecuaciones Diferenciales

donde las trayectorias de entrada y salida son trayectorias continuas.

La función de tasa de cambio f debe tener solución única para asegurar que las trayectorias de estado sean únicas.

Ecuaciones en Diferencia (*DTSS: Discrete Time Systems Specifications*).

Se emplea un conjunto de ecuaciones en diferencia para especificar el próximo estado en el tiempo, el cual avanza en forma constante. La representación de este formalismo mostrado en la figura 3.5 es:

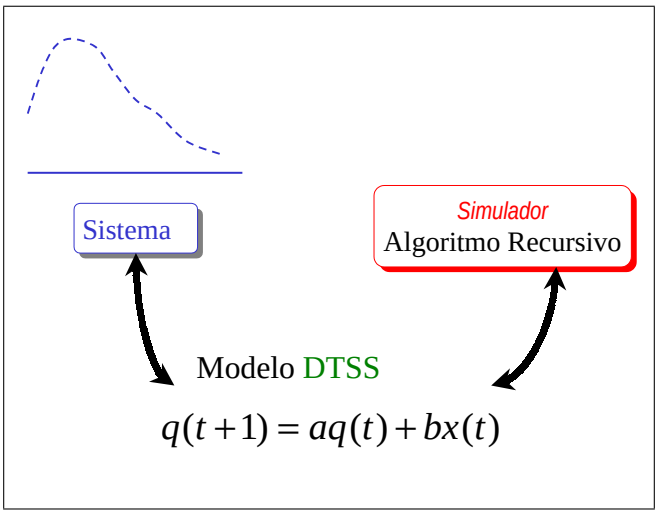

Figura 3.5: Formalismo de Ecuaciones en Diferencias

$$DTSS = (X, Y, \Omega, Q, \delta, \lambda, c)$$

X : Conjunto de Entradas.
Y : Conjunto de Salidas.
Ω : Conjunto de las secuencias admisibles sobre X,Y.
Q : Conjunto de Estados.
δ : $Q \times X \rightarrow Q$
Función de transición de estados.
λ : $Q \rightarrow Y \ \vee \ Q \times X \rightarrow Y$
Función de Salida.
c : Base temporal. Isomorfa a los enteros.

Eventos Discretos (*DEVS: Discrete Event Systems Specifications*).

El sistema es activado por eventos y obedece a transiciones producidas interna y externamente. La representación de este formalismo mostrado en la figura 3.6 es:

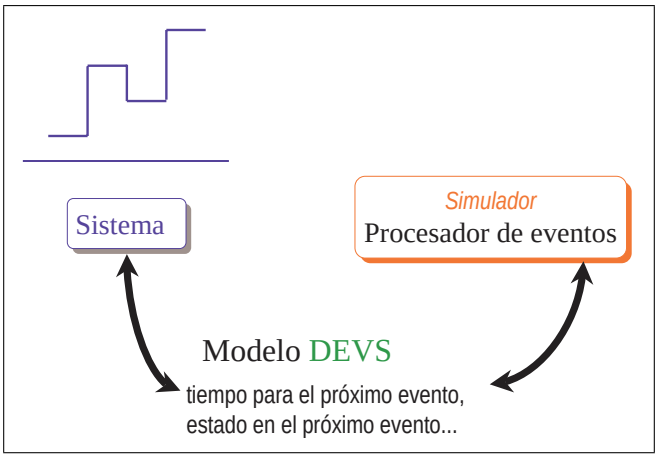

Figura 3.6: Formalismo de Eventos Discretos

$$DEVS = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$$

X : Conjunto de eventos externos de entrada.
Y : Conjunto de eventos externos de salida.
S : Conjunto de estados.
δ_{int} : $S \rightarrow S$
Función de transición interna.
δ_{ext} : $Q \times X \rightarrow S$

Función de transición externa donde
$Q : \{(s, e) \mid \ s \in S,\ 0 \leq e \leq ta(s)\}$
e : tiempo transcurrido desde la última transición

$\lambda :$ $S \rightarrow Y$
Función de salida.

$ta :$ $S \rightarrow \mathcal{R}^+$
Función de avance de tiempo.

3.1.2 Formalismo DEVS

Esta sección introduce el formalismo DEVS para sistemas de eventos discretos que proporcionan un enfoque modular, jerárquico a la construcción de modelos de simulación de eventos discretos [9]. Con este fin se incluyen en el DEVS los conceptos de teoría de sistemas y modelado introducidos en la sección 3.1.

En este documento, mostraremos que DEVS no sólo es importante para modelo de eventos discretos sino que también proporciona una base computacional para implementar comportamiento de sistemas que pueden ser expresados en diferentes formalismos (DESS, DTSS, ...) y que abarcan los enfoques de simulación tanto secuencial como paralela o distribuida. Por otro lado, el simulador DEVS puede usarse para implementar otras estrategias, por ejemplo, el esquema de tiempo discreto se implementa a través de eventos uniformemente distribuidos en el tiempo, tal y como mostraremos a continuación.

Estructura DEVS

DEVS separa el modelado de la simulación y propicia la reutilización y acoplamiento de componentes. Los modelos DEVS capturan la dinámica del comportamiento de los sistemas, es decir, representan los estados del sistema y el cambio de estos estados a través del tiempo. Estos cambios pueden ser causados por la evolución misma del sistema o en respuesta a eventos externos. Los modelos DEVS se clasifican en:

Atómicos: Representación modular de sistemas. Son modelos de bajo nivel basados en la estructura del sistema. Sus elementos son:

* Entradas
* Salidas
* Variables de estado
* Funciones de transición
* función de salida
* Función de avance de tiempo

Acoplados: Representación jerárquica de sistemas. Un modelo acoplado esta compuesto por uno o más modelos atómicos o acoplados. Sus elementos son:

- Componentes,
- Reglas internas de acoplamiento,
- Reglas externas de acoplamiento: Entradas y Salidas.

A cada componente de un modelo DEVS acoplado se le puede representar a través de su equivalente escrito como una composición de modelos DEVS atómicos.

Especificación DEVS

La jerarquía de niveles con los que un sistema puede describirse o especificarse se resume en la figura 3.7. A medida que se incrementa el nivel, se aumenta la especificidad estructural (es decir, nos movemos del comportamiento a la estructura). Cabe mencionar que se incorpora nuevo conocimiento en cada nivel en la jerarquía.

Nivel	Especificación	Formalismo
0	Marco E/S	(T, X, Y)
1	Relación observación E/S	(T, X, Ω, Y, R)
2	Función observación E/S	(T, X, Ω, Y, F)
3	Sistema E/S	$(T, X, \Omega, Y, Q, \Delta, \Lambda)$
4	Sistema Secuencial	$(T, X, \Omega_G, Y, Q, \delta, \lambda)$
5	Sistema Estructurado	$(T, X, \Omega, Y, Q, \Delta, \Lambda)$ con X, Y, Q, Δ, Λ estructurados
6	Sistema no modular acoplado	$(T, X, \Omega, Y, D, \{M_d \mid d \in D\})$ con $M_d = (Q_d, E_d, I_d, \Delta_d, \Lambda_d)$
7	Sistema modular acoplado	$N = (T, X_N, Y_N, D, \{M_d \mid d \in D\}, \{I_d \mid d \in D \cup \{N\}\}, \{Z_d \mid d \in D \cup \{N\}\})$

Figura 3.7: Jerarquía de Especificación de Sistemas

Nivel 0. Describe las entradas y salidas de un sistema en el tiempo.

Nivel 1. Se refiere a registros completamente observacionales de la conducta de un sistema a través de un conjunto de segmentos entrada/salida.

Nivel 2. Se estudia el conjunto de funciones E/S que relacionan las partes de un sistema E/S. Cada función es asociada a un estado inicial del sistema.

Nivel 3. El sistema es descrito por conjuntos abstractos y funciones. Se introducen el espacio de estados y las funciones de transición y salida. La función de la transición describe las transiciones de un estado al estado siguiente causadas por los segmentos de entrada; la función de salida describe la transición de un estado a una salida observable.

Nivel 4. El sistema se describe a través del generador de segmentos de entrada y las transiciones del estado ocasionadas por estos segmentos.

Nivel 5. Se presentan los conjuntos abstractos y las funciones de los niveles más bajos como provenientes del producto directo de conjuntos primitivos y funciones. Esta es la forma en que se describen informalmente modelos.

Nivel 6. Un sistema se especifica como un conjunto de componentes. Cada uno de estos componentes con su propio conjunto de estados y su función de transición. Los componentes se acoplan en forma no modular y la influencia de cada uno recae sobre los otros a través de sus conjuntos de estados y las funciones de transición.

Nivel 7. Un sistema se especifica como un acoplamiento modular de componentes en contraste con los acoplamientos no modulares de nivel 6. Los componentes representan característica técnicas del sistema y se acoplan conectando sus interfaces de entrada y salida.

Con esta especificación, por un proceso de múltiples pasos podemos expresar la conducta asociada con sistemas a cualquier nivel. El proceso de ir de una especificación del sistema a la especificación de su comportamiento es equivalente a la simulación de un modelo por computadora. La máquina obtiene, paso a paso los estados del sistema y la salida. Cada paso involucra el cálculo de las variables de estado componente a componente. Las secuencias de estados y salidas constituyen trayectorias, y los conjuntos de tales trayectorias constituyen la conducta del modelo.

Para los propósitos de la simulación, los niveles más estructurados son muy prácticos. Esta base formal nos servirá ahora para plantear las estrategias de simulación.

Simuladores DEVS

Los simuladores son los responsables de ejecutar la dinámica del modelo (en forma de instrucciones). El simulador puede operar de varias maneras:

* secuencial,
* paralela

∗ distribuida (secuencial/paralela)

Existen algoritmos básicos para los simuladores sobre DEVS que se basan en el manejo de una lista de eventos ordenada según el tiempo de activación de los eventos. Los eventos se extraen en orden secuencial de dicha lista y se ejecutan tomando en cuenta las transiciones de estados.

La ejecución de un evento genera nuevos eventos que deben ser incluidos en la lista de eventos. Así mismo una ejecución puede ocasionar cancelación o eliminación de eventos.

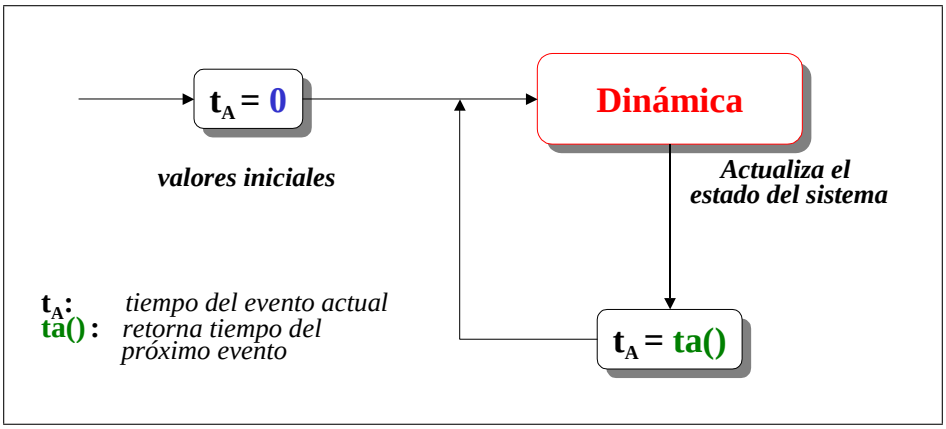

Figura 3.8: Protocolo DEVS básico

Tomando en cuentas estas consideraciones generales, formularemos esquemas para simuladores DEVS siguiendo el modelo básico y el acoplado. En particular, el algoritmo básico para el simulador atómico, mostrado en la figura 3.8, es:

---

1. Asignar valores iniciales al tiempo actual.
$$t_A = 0$$
2. Actualizar el estado del sistema.
   (a) Obtener el estado del sistema para el tiempo actual
   (b) Generar los mensajes de salida
   (c) Ordenar y distribuir los mensajes de salida
   (d) Si ocurren eventos externos ejecutar la función de transición.
3. Calcular el tiempo del próximo evento a partir de la función $ta()$
$$t_A = ta()$$
4. Regresar al paso 2.

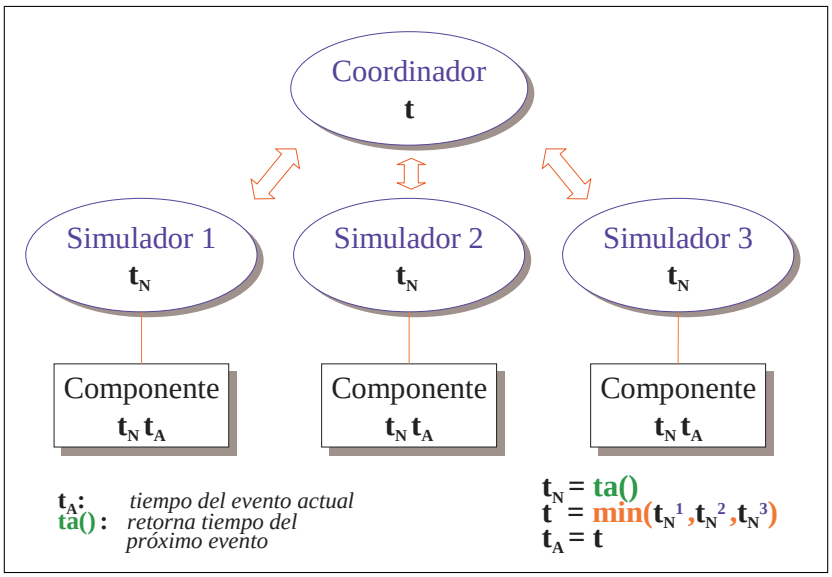

Figura 3.9: Protocolo DEVS acoplado

Por otro lado, el simulador acoplado mostrado en la figura 3.9 muestra una estructura de simuladores y un coordinador. En este caso es importante destacar que existe una única lista de eventos que es controlada por el coordinador. Por ende, los simuladores deben reportar constantemente el estado del sistema y además deben actualizar el tiempo de acuerdo a la lista de eventos general. El algoritmo que simula correctamente este simulador es:

---

1. Asignar valores iniciales al tiempo actual en cada componente.
$$t_A = 0$$
2. Actualizar el estado del sistema en cada componente
   (a) Obtener el estado del sistema para el tiempo actual
   (b) Generar los mensajes de salida
   (c) Ordenar y distribuir los mensajes de salida
   (d) Si ocurren eventos externos ejecutar la función de transición.
3. Calcular el tiempo del próximo evento t_N para cada componente
$$t_N = ta()$$
4. Calcular el tiempo del próximo evento (t global)
$$t = min(t_N^1, t_N^2, \cdots)$$
5. Informar a todos los componentes el t global
$$t_A = t$$
6. Retornar al paso 2.

Modelado y Simulación DEVS

Aplicados a sistemas cada vez más complejos, los proyectos de modelado y simulación de sistemas a gran escala, como muestra la figura 3.10, utilizan toda una infraestructura que involucra equipos interdisciplinarios y múltiples técnicas o herramientas [12].

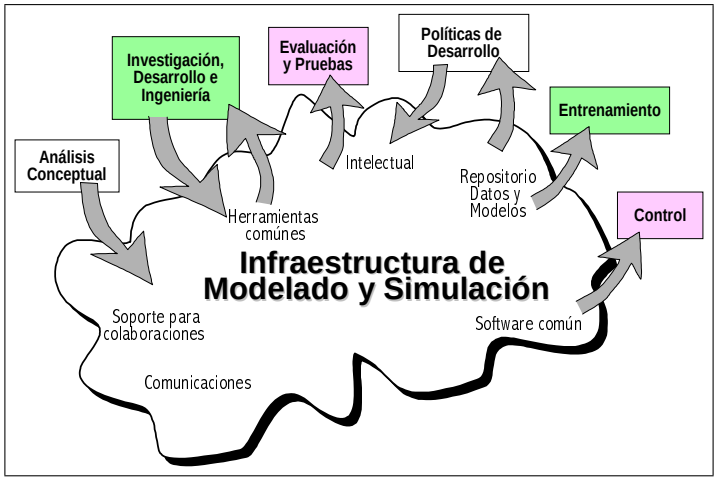

Figura 3.10: Infraestructura para modelado y simulación

La figura 3.11 muestra un esquema de planificación (.ibid) para el modelado y la simulación de sistemas en entornos cooperativos. Este esquema de niveles organiza

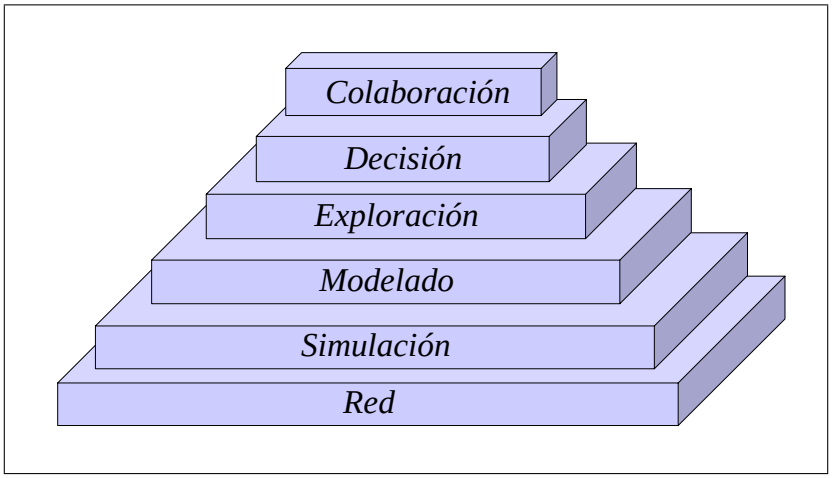

Figura 3.11: Arquitectura para modelado y simulación

los conceptos y las actividades involucradas con el proceso de modelado y simulación, así:

Red: Contiene el hardware y software necesario para soportar todos los aspectos del ciclo de vida de modelado y simulación .

Simulación: Contiene el software que permite la ejecución del modelo con miras a generar el comportamiento del sistema. En esta capa se incluyen:

1. los protocolos que proporcionan la base para simulación distribuida,
2. los sistemas manejadores de bases de datos,
3. software para control, visualización y animación de las simulaciones.

Modelado: Contiene la especificación de los atributos y la dinámica del sistema. Soporta el desarrollo de modelos en múltiples formalismos de manera independiente de la simulación. Acá se incluye la descripción y la construcción del modelo.

Exploración: Soporta la exploración y evaluación del modelo. Implica el estudio de los posibles modelos alternativos. Esta etapa permite ajustar los parámetros del modelo.

Decisión: Involucra la toma de decisiones de acuerdo al dominio de aplicación. Esta capa permite ampliar la capacidad de simular y escoger conjuntos de modelos en el dominio del problema real que sirven de soporte a las tareas de exploración, investigación y optimización.

Colaboración: Permite a cada participante incorporar el conocimiento parcial que posee sobre el sistema. Normalmente este conocimiento esta basado en el área de investigación, disciplina, ubicación, tareas o responsabilidades de los participantes. En este nivel, las perspectivas individuales contribuyen a lograr el objetivo global del modelado y la simulación.

Luego de presentar una breve descripción de los formalismos DESS, DTSS y los detalles del formalismo DEVS a continuación mostraremos una clasificación de los sistemas basada en los enfoques tradicionales de simulación.

3.1.3 Enfoques para Simulación

Como mencionamos anteriormente, uno de los conceptos clave en simulación es la manera en que se realiza el avance del tiempo. Este concepto permite realizar una distinción básica entre las simulaciones basadas en eventos discretos y aquellas basadas en modelos continuos.

Las **Simulaciones de Eventos Discretos** están basadas en modelos que asumen que el sistema no cambia de estado hasta que ocurre un evento. Dicho evento

genera un cambio instantáneo en el estado del sistema. En este caso, el avance del tiempo se realiza de evento a evento y los eventos se ordenan cronológicamente. Estas simulaciones se encargan generalmente de representar sistemas humanos o autómatas que deben ejecutarse en forma sincronizada, generando análisis estadísticos.

Por otro lado, los modelos continuos generalmente se encargan de representar sistemas a través de ecuaciones diferenciales que pueden ser ejecutadas simultáneamente, en lotes, generando tablas o gráficos de funciones continuas. Estos modelos asumen que el tiempo avanza en forma continua y los cambios de estado del sistema son también continuos. El avance de tiempo en este caso se representa con una variable que avanza en incrementos constantes lo suficientemente pequeños para simular en forma precisa el comportamiento continuo del tiempo. Para simular el estado continuo las variables que representan el estado del sistema son actualizadas en cada avance de tiempo. Las simulaciones basadas en estos modelos se denominan **Simulaciones Continuas**.

La **Simulación Combinada**, propuesta por Fahrland [13], abarca la simulación de sistemas que están constituidos por subsistemas discretos y subsistemas continuos que se ejecutan de manera coordinada y cooperativa. Una simulación combinada es una operación concurrente en el tiempo donde el tiempo para las partes discreta y continua debe avanzar de forma sincronizada. La incorporación de subsistemas discretos hace que el sistema completo pueda ser modelado como discreto. De esta forma los segmentos uniformes de la parte continua son incorporados como eventos, y deben ordenarse cronológicamente con los eventos del subsistema discreto.

La simulación combinada es más que una simple agregación de los componentes discretos y continuos ya que proporciona los mecanismos para el acceso a las variables de estado, la activación y desactivación de procesos y la interacción estructural mutua entre los componentes discretos y continuos. Además, al establecer interfaces de comunicación entre los componentes discretos y continuo la simulación combinada permite la construcción de módulos y al organizar dichos componentes en distintos niveles de agregación, se permite la construcción de jerarquías.

Una vez aclarada la forma de representar sistemas basados en modelos continuos y en eventos discretos siguiendo el esquema de eventos discretos, no es necesario hacer distinción entre ambos paradigmas de modelado. Por tanto, a partir de este momento nos concentraremos en el estudio de simulación de eventos discretos.

Otro aspecto ampliamente estudiado en simulación es el uso de múltiples procesadores para realizar un experimento. La creciente disponibilidad de computadores con varios procesadores ha incrementado el interés de explotar el paralelismo natural de algunos sistemas a la hora de realizar el modelado y la simulación. En esta área se han realizado estudios para llevar a cabo el modelado y la simulación de estos sistemas cuidando principalmente la coordinación y la asignación de procesos que pueden ser separados y alojados en procesadores diferentes [14, 15]. Este enfoque

es conocido como **Simulación Distribuida**. Además, como un caso particular se presenta el estudio de sistemas que presentan procesos separables que deben ocurrir simultáneamente. A este enfoque se le conoce como **Simulación Paralela**. [16]

Además de los problemas clásicos de distribución, tales como balance de carga y asignación de procesos a los procesadores, estos enfoques deben sobreponerse al cuello de botella generado al mantener ordenada la única lista de eventos que permite el manejo global del tiempo en el sistema. Comúnmente se requieren los mecanismos de sincronización y de comunicación entre procesos con el fin de mantener una estricta sincronización local y global de los relojes. Sin embargo, alternativamente pueden existir mecanismos de regreso del tiempo que permitan compensar errores en la sincronización.

En general, se conocen dos métodos que permiten resolver el problema del avance del tiempo y del manejo y ordenamiento de la lista de eventos. El primero de ellos sigue la perspectiva *optimista* y supone que cada proceso programa eventos en la lista independientemente del resto de los procesos. En el caso que algún proceso requiera programar un evento en el pasado, es decir, que debe ocurrir antes que el siguiente evento de la lista, este método utiliza los mecanismos de regreso del tiempo para retornar el tiempo y el estado de la simulación al momento en que debe ocurrir el evento que se desea programar.

El segundo método sigue la perspectiva *conservadora* en el cual se utiliza la sincronización de los procesos para detener un avance de tiempo indebido. De esta manera, no se registra un avance de tiempo hasta que se asegura que no ocurrirán programaciones de eventos en el pasado del componente en cuestión.

Otro aspecto importante que ha sido estudiado en simulación es la posibilidad de cambiar la estructura interna del modelo de simulación, la cual normalmente permanece durante el tiempo en que transcurre la simulación. Todos aquellos modelos en los que la estructura cambia deben ser tratados bajo otro enfoque: **Simulación con Cambio Estructural** [17]. En estos modelos, tanto los componentes internos como las relaciones entre ellos pueden mutar ocasionando cambios en la estructura del sistema. Existen diferentes maneras de cambiar la estructura interna del sistema. Generalmente se estudia el cambio motivado al agregar, modificar o eliminar alguna componente. Sin embargo el cambio también puede generarse al modificar las interacciones entre las entidades o las reglas de comportamiento [18].

Por último, es importante que mencionemos otra área de estudio en simulación que incorpora la técnica de reutilización de componentes a la hora de realizar simulaciones. El interés en simulaciones compuestas de diferentes componentes de simulación ha llevado al estudio de la **Simulación Interactiva**, donde *interactividad* se refiere a la habilidad para combinar componentes de simulación sobre una plataforma de computación heterogénea distribuida donde frecuentemente se ejecutan operaciones en tiempo real.

Este enfoque involucra una reformulación de la forma tradicional en que interactúan los componentes de simulación. En lugar de un único programa ejecutándose sobre un computador, es necesario pensar en un número de programas ejecutándose en varios computadores heterogéneos distribuidos que interactúan a través de un sistema operativo distribuido. Por otro lado, como en los enfoques de simulación descritos anteriormente, las variables que representan el estado del sistema se actualizan en cada avance de tiempo.

Los estudios en Simulación Interactiva han conducido al desarrollo del marco de un referencia estándar para el soporte de simulaciones interactivas denominado HLA (*High Level Architecture*).

3.1.4 Marco de Referencia HLA

El DoD[1], bajo la dirección de la oficina de modelado y simulación (DMSO)[2], ha desarrollado HLA [19] en respuesta a las necesidades de su plan maestro de modelado y simulación, el cual requiere una estructura de trabajo común que se pueda aplicar a un amplio rango de aplicaciones potenciales.

Esta sección describe brevemente los principios básicos del marco de referencia HLA que sirve de base en el diseño propuesto para la plataforma GALATEA.

El objetivo del HLA es facilitar la interoperabilidad entre las simulaciones y promover la reutilización de las simulaciones y sus componentes. Este marco de referencia esta basado en la premisa *"una simulación individual o un conjunto de simulaciones desarrolladas para un propósito dado pueden ser aplicadas en otra simulación"* y lo implementa bajo el concepto de federación (un conjunto compuesto de simulaciones entrelazadas). La finalidad de HLA es proporcionar una estructura que soporte reutilización de las capacidades disponibles en simulaciones diferentes. HLA no especifica una implementación en particular, ni se restringe al uso de un software particular o un lenguaje de programación. Esto proporciona cierta libertad al desarrollador.

HLA tiene aplicabilidad en muchas áreas de la simulación incluyendo educación, entrenamiento, análisis, diseño e incluso entretenimiento. Estas aplicaciones ampliamente diferentes nos dan un indicio sobre la variedad de requerimientos que son considerados en el desarrollo y la evolución de HLA.

La figura 3.12 muestra los componentes funcionales de una federación HLA. El primer componente son las **Simulaciones**, las cuales deben incorporar capacidades específicas que le permitan a los objetos de una simulación, *federados*, interactuar con los objetos de otra simulación a través del intercambio de datos soportado por los servicios implementados en la interfaz.

---

[1] DoD: *Departament of Defense*

[2] DMSO: *Defense Modeling and Simulation Office*

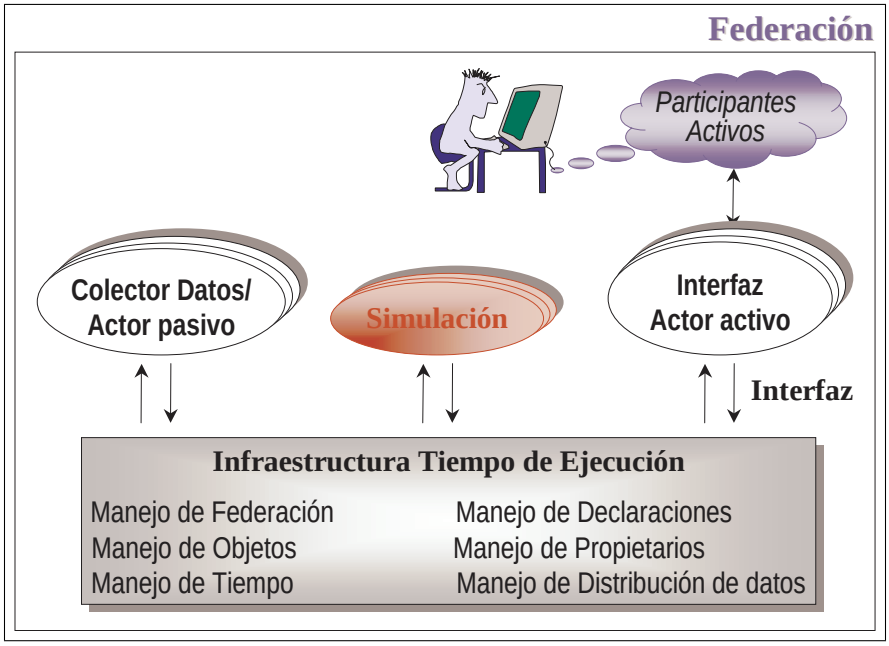

Figura 3.12: Esquema funcional de una federación HLA

El segundo componente funcional es la **Infraestructura en tiempo de ejecución**, RTI[3]. Esta infraestructura proporciona un conjunto de servicios de propósito general que soportan las interacciones entre los componentes de simulación, por lo que puede visualizarse como un sistema operativo distribuido para los federados. Todas las interacciones entre los componentes de simulación se realizan a través de la RTI.

El tercer componente es la **Interfaz** al RTI. Esta interfaz, independiente de su implementación, proporciona una manera estándar para que los componentes de simulación interactúen con la RTI. Además, esta interfaz es independiente de los requerimientos de modelado y de intercambio de datos de los componentes de simulación.

Formalmente, el marco de referencia HLA especifica tres componentes básicos para la simulación:

(a) una *especificación de la interfaz* que define los servicios y las funciones de cada componente.

(b) una *especificación del modelo común para almacenar información.*

(c) un *conjunto de reglas* que deben cumplir los componentes de simulación en forma independiente y como grupo.

[3]RTI: *Runtime Infrastructure*

Especificación de la Interfaz HLA

La especificación de la interfaz describe los servicios en tiempo de ejecución que el RTI proporciona a los federados, y aquellos proporcionados por los federados al RTI. Se destacan seis clases de servicios:

Manejo de federación: ofrece las funciones básicas requeridas para crear/ destruir federaciones, iniciar/detener ejecuciones de una federación e incorporar/separar federados a una federación.

Manejo de declaraciones: crea las relaciones entre solicitante y proveedores de información con las que una federación HLA define aquellos datos que un federado pretende generar (publicación) y aquellos datos que intenta utilizar (inscripción).

Manejo de objetos: provee los servicios para instanciar/eliminar objetos, actualizar/consultar atributos, solicitar/entregar atributos y enviar/recibir mensajes.

Manejo de propietarios: soporta la transferencia dinámica de propietarios de los atributos de un objeto mientras se realiza una ejecución.

Manejo de tiempo: soporta sincronización del intercambio de datos durante la simulación.

Manejo de distribución de datos: soporta el enrutamiento eficiente de datos entre federados durante la ejecución de una federación. Proporciona un mecanismo flexible y extensible para publicación e inscripción.

La especificación de la interfaz define de que manera se acceden estos servicios tanto funcionalmente como en la implementación de la misma en un API[4].

En HLA cada componente de simulación maneja un tiempo local de forma lógica tomando en cuenta ciertas restricciones:

* Es necesario definir los valores inicial y final para el tiempo.
* Los valores asociados al tiempo no están sujetos a ningún sistema de unidades.
* El tiempo actual de simulación es siempre mayor o igual al tiempo inicial.
* El tiempo se maneja en forma discreta.
* Es necesario coordinar la sincronización entre el tiempo lógico y el tiempo real.

[4]API: *Application Programming Interface*

El manejo del tiempo puede realizarse siguiendo uno de los siguientes esquemas:

(a) **Manejo Local.** Cada componente controla el avance del tiempo.

(b) **Sincronización Conservadora.** El avance de tiempo de cada componente se realiza cuando esta garantizado que no se recibirán eventos pasados.

(c) **Sincronización Optimista.** Cada componente es libre de avanzar su tiempo local, pero debe estar preparado para regresar su tiempo lógico si recibe un evento pasado.

(d) **Supervisión de actividades.** Los componentes controlan el tiempo local en periodos. Al final de cada periodo se intercambian mensajes hasta que acuerdan avanzar el tiempo lógico juntos entrando al próximo periodo.

Modelo de objetos HLA

El modelo de objetos HLA es la clave para lograr estas metas de interoperabilidad y reutilización, y representa uno de los principios fundamentales sobre los que se ha definido el estándar. El modelo de objetos HLA esta definido como el conjunto de descripciones, en términos de objetos, de los elementos esenciales que son compartidos entre las simulaciones. En HLA los modelos son entendidos en función de la descripción de los datos que se comparten en una federación.

HLA especifica dos tipos de modelos de objetos: el correspondiente a la federación (FOM)[5] y el correspondiente a las simulaciones (SOM)[6]. El FOM describe el conjunto de objetos, atributos e interacciones que se comparten en la federación, mientras que el SOM describe la simulación (federado) en términos de los tipos de objetos, atributos e interacciones que ofrecen a las federaciones. El SOM proporciona información acerca de las capacidades de una simulación de intercambiar información como parte de una federación. El SOM es esencialmente un contrato para definir el tipo de información que estará disponible para las federaciones. Este facilita la evaluación de cuan apto es un federado para participar en una federación.

El formato de presentación estándar y el contenido del modelo de objetos para HLA es proporcionado por la plantilla OMT[7]. Las categorías de información descritas colectivamente por el OMT definen al modelo de información requerido por todas las federaciones de HLA y los participantes de la federación. Las extensiones a OMT proporcionan categorías suplementarias de información optativa que pueden proporcionar claridad adicional y integridad en algunos tipos de modelos de objetos HLA.

[5]FOM: *Federation Object Model*
[6]SOM: *Simulation Object Model*
[7]OMT: *Object Model Template*

Reglas HLA

Finalmente tenemos el conjunto de reglas que encierran el objetivo principal detrás de HLA: permiten la interacción apropiada entre los componentes y describen las responsabilidades de la simulación y de los componentes. Estas reglas están divididas en dos grupos: las correspondientes a la federación y las de los federados.

Durante el tiempo de ejecución, todas las representaciones de objetos se realizan en los federados y solo un federado es dueño de un atributo de un objeto instanciado a la vez. El intercambio de información se realiza a través de la RTI utilizando la especificación de interfaz HLA.

Como puede observarse, HLA prescribe un modelo genérico, distribuido y orientado por objetos para hacer simulación. Este modelo integra el estado del arte en cuanto a estrategias de simulación de sistemas complejos, compuestos por subsistemas de diversa naturaleza. Este marco de referencia, sin embargo, no prescribe soluciones para un problema que continúan planteándose varias disciplinas asociadas a las tecnologías de la información: cómo incorporar más inteligencia en los componentes del sistema (simulado, en este caso). Una solución a ese problema permitiría, además, aumentar la expresividad de los lenguajes y plataformas de simulación, permitiendo, posiblemente, modelos de simulación mucho más realistas.

Para revisar el estado del arte en la búsqueda de soluciones a ese problema, se dedica la sección 3.1.5 a revisar los últimos aporte de la Inteligencia Artificial y áreas afines: La tecnología de agentes inteligentes.

3.1.5 Simulación Orientada a Agentes

Como mencionamos anteriormente el término agente se asocia generalmente a entidades abstractas que *"perciben su ambiente, a través de sensores, y actúan sobre su entorno, a través de efectores"* [4]. Sin embargo, algunos investigadores han sugerido el estudio de los agentes atribuyéndoles una estructura o estado interno. Para incluir dicha estructura necesario utilizar una definición extendida:

> *"Un agente es una entidad que puede percibir su ambiente, asimilar dichas percepciones y almacenarlas en un dispositivo de memoria, razonar sobre la información almacenada en dicho dispositivo y adoptar creencias, metas e intenciones por sí mismo dedicándose al alcance de dichas intenciones a través del control apropiado de sus efectores" [20].*

La definición anterior no hace referencia explícita a las habilidades sociales. Siguiendo la interpretación de sistemas multi-agentes interactuando simbólicamente [4] es necesario hacer que las percepciones y los comandos a los efectores incluyan el pase de mensajes en un lenguaje compartido por los agentes. En otras palabras, éstas

habilidades sociales pueden ser agregadas a la arquitectura específica, como *actos verbales* [21].

Un agente puede representar a un individuo, a grupos sociales, o a instituciones que se comportan como un individuo, en ciertas circunstancias. Es posible representar sistemas utilizando agentes cuando se decide controlar y analizar el comportamiento de dichos sistemas en términos mentales (creencias, metas, preferencias) y la interacción con su ambiente en términos de acciones y percepciones. Para coordinar y organizar su comportamiento, los agentes, además de utilizar la percepción del ambiente, utilizan su conocimiento sobre las capacidades y creencias de los otros agentes.

El enriquecimiento de las técnicas de simulación con herramientas de la Inteligencia Artificial puede rastrearse hasta el trabajo de Ivan Futó, en Hungría, en los años ochenta del siglo pasado [22] donde se constituyeron diversas extensiones del lenguaje PROLOG para simulación, incluyendo una que permite incluir una concha para sistemas expertos (ALL-EX) en un modelo de simulación en CS-PROLOG (una versión PROLOG para simulación). A pesar de lo completo de ese sistema, no alcanzó mucha popularidad en la comunidad de simulistas.

Quizás la primera asociación sistemática de la noción de agente con simulación se le debe al mismo proponente de la ampliamente conocida teoría de simulación: el profesor Bernard Zeigler. A principio de los años 90, del siglo pasado, Zeigler extiende y sistematiza la teoría de simulación para incluir los principios de orientación a los objetos y las nociones elementales de AGENTE [23]. En ese trabajo, Zeigler explica como algunos modelos no triviales de simulación se pueden constituir con la integración de modelos más elementales (inferior en jerarquía, es decir, que modelan un componente particular de todo el sistema de interés) a través de interfaces bien definidas para intercambio de datos y señales de sincronización. Zeigler asocia esos modelos más elementales, *modelos atómicos* como él los llama, y los propios simuladores a objetos, como en la programación orientada a objetos, y ofrece una cautivadora ilustración de la idea en SCHEME, el lenguaje basado en objetos considerado una extensión de LISP. El sistema se denominó DEVS-Scheme (.ibid) y se ha constituido en una herramienta completa para simulación de sistemas.

Además de proponer esa asociación entre objetos y componentes de simulación y de mostrar como una agregación de modelos puede constituir un modelo de todo un sistema, re-usando modelos más básicos, Zeigler acomoda la noción de agente en simulación. Un agente es un objeto, un modelo atómico, que incorpora un modelo del sistema en el que el mismo agente está imbuido. Esta es la noción de sistema *endomórfico*, en la cual es posible establecer una relación mórfica entre un modelo y un elemento del mismo modelo, el agente en este caso. La idea es, desde luego, que el agente manipule una representación del sistema en el que se encuentra (una imagen de su ambiente) y pueda orientar su actuación con esa representación, tal como postulan las teorías de agente en Inteligencia Artificial. En la propuesta de Zeigler se puede

encontrar, inclusive, una descripción de la función de transición del modelo atómico que corresponde al agente, que se asemeja mucho a la especificación de un motor de inferencia, con reglas escritas en el lenguaje funcional del LISP.

Las idea de Zeigler fueron retomadas por sus estudiantes. En particular, Adelinde Urhmacher, diseño una extensión de DEVS-Scheme que se denomina AgeDEVS y que es presentada por Zeigler en el mismo texto de 1990. Urhmacher orienta su trabajo hacia uno de los problemas más discutidos en la comunidad de los simulistas: La simulación del cambio estructural [24].

Basado en la percepción parcial del mundo, los agentes en AgeDEVS son capaces de manejar en forma flexible ambientes de estructura variable [25, 26]. Es posible implementar diferentes estrategias que modelan el cambio estructural basándose en aquel modelo interno, en el cual el conocimiento de la estructura del sistema, su composición y su acoplamiento son expresados explícitamente. Para describir el cambio estructural DEVS divide el sistema en "cuerpo y mente", en una unidad controlada y una unidad controladora "inteligente" [18]. Como se verá más adelante, estos son mismos principios se adoptan en GALATEA para asociar cambios estructurales en los sistemas, con los agentes.

En particular, con la orientación por agentes es posible representar de múltiples niveles de sociedades, asociando a los actores individuales y colectivos con agentes. Así los agentes sociales, son agregados que pueden incluirse en modelos a la par de agentes individuales. Haciendo uso de este enfoque, en [27] se introduce un modelado de múltiples niveles para modelado y simulación de agentes el cual es independiente del tipo de agentes a involucrar: intencionales o reactivos.

La descripción de entidades y sus interacciones son fundamentales para Simulación Orientada por Agentes. Los sistemas descritos con agentes son arreglados en diferentes niveles de organización donde la composición y el acoplamiento son los principales aspectos a definir, tal como se hace en los sistemas jerárquicos originales de Zeigler. Así mismo, la actividad de un sistema esta restringida a cambios en sus propios estados internos, los cuales dependen de los estados anteriores y de las perturbaciones externas. Las interacciones simbólicas representan a los individuos como entidades dinámicas que evolucionan. Estas entidades actúan guiadas por sus percepciones, las cuales son obtenidas generalmente a través de comunicación con el ambiente o con otras entidades.

Al realizar encapsulamiento de datos y procedimientos, los objetos permiten representar sistemas utilizados en simulación de sociedades. Sin embargo se presentan dos problemas críticos: en primer lugar, las entidades en la simulación orientada por objetos usualmente no involucran las interacciones intencionales que desarrollan los agentes; y en segundo lugar, los objetos generalmente carecen de un modelo interno que le permita comunicarse con su ambiente y con sí mismos. Es este el punto en

donde la orientación por objetos, tan cercanamente asociada a la simulación de sistemas desde su origen, falla en ofrecer mecanismos de representación para sistemas multi-agentes en el mundo real, que queramos modelar por alguna razón.

En estas circunstancias, emprendemos trabajos de investigación como este, con la intención de enriquecer los lenguajes y medios de simulación, con las abstracciones necesarias para modelar agentes.

Uno podría pensar que, así como en ese esfuerzo por mayor expresividad, los lenguajes de simulación dieron origen a la orientación por objetos, convertida ahora en el paradigma más popular en la Ingeniería del Software, quizás esta disciplina y otras (como la Inteligencia Artificial), puedan retribuir con contribuciones a los lenguajes de simulación.

Por esta razón, revisamos a continuación los desarrollos recientes en el espacio común que definen la ingeniería del software y la tecnología de agentes inteligentes.

Agentes como paradigma para el desarrollo de software

La ingeniería de sistemas basada en, u orientada a, agentes es un nuevo enfoque en computación que promete una revolución no necesariamente contraria a la causada por la orientación a objetos. Es de esperar que aquella revolución sea complementaria a la de los objetos y de un impacto profundo en el desarrollo de software complejo o que realice tareas de elevada complejidad. Esta es la conclusión que defienden Nick Jennings y Michael Wooldridge [28], tras más de una década de experiencia usando tecnología de agentes para construir aplicaciones de gran escala en una gran variedad de dominios comerciales e industriales (.ibid).

El argumento en favor de una ingeniería del software orientado por agentes parece, sin embargo, todavía lejos de ser definitivo, a diez años de las primeras menciones a la idea [29] entre los teóricos de la Inteligencia Artificial. Es apenas ahora cuando comienzan a tomar cuerpo metodologías que están basadas en los agentes o que les consideran en el diseño e implementación de software [30] y todavía no se dispone de evaluaciones métricas que permitan evaluar el impacto de tales metodologías [28].

Hay, no obstante, un interés creciente en la transferencia de la experiencia con agentes, que originalmente fue del dominio de la Inteligencia Artificial (Distribuida), a la práctica cotidiana del ingeniería de software. Los investigadores de la antigua Inteligencia Artificial parecen reconocer que sus esfuerzos iniciales por modelar un dispositivo inteligente carecían de un compromiso sistemático (de ingeniería) con la posibilidad de que ese dispositivo "hiciera algo". Ese "hace algo (inteligentemente)" es la característica esencial de los AGENTES (inteligentes).

Esta característica, combinada con el enfoque modular que supone concebir un sistema como constituido por agentes (algo similar a concebirlo constituido por objetos), hacen de los agentes una herramienta muy efectiva para atacar problemas complejos

[7], tales como la gestión de conocimiento [31], la interacción humano-computador [32] y, como ilustramos en esta tesis, el modelado y simulación de sistemas.

Jennings y Wooldridge [28] demuestran que 1) las descomposiciones de sistemas basadas en agentes son un medio efectivo de particionar el espacio problemático de un sistema complejo; 2) las abstracciones asociadas a los agentes (como las aptitudes intencionales: metas, creencias y preferencias), son medios naturales (para usuarios no expertos, en particular) de modelar sistemas complejos; y 3) que la postura orientada a agentes, cuando se trata de identificar y gerenciar relaciones organizacionales, es apropiada para lidiar con las dependencias e interacciones que existen en un sistema complejo.

Precisamente acerca de interacciones, Jennings y Wooldridge explican que una de las ventajas del modelado de agentes es la posibilidad de lidiar con las interacciones entre componentes del sistemas, sin tener que abordar todos los detalles de cada interacción al momento de diseñar . Los agentes son dotados de estrategias generales para interactuar como parte de su diseño, pero los patrones de interacción particulares a cada situación y contextos son decididos por cada agente en tiempo de ejecución. Es decir, los diseñadores pueden concentrarse en describir la lógica de la interacción entre los agentes (y en general la lógica de su comportamiento), dejando a los propios agentes las decisiones acerca de qué hacer en cada situación particular.

Esas explicaciones generales han inspirado descripciones lógicas de lo que es un agente y una sociedad de agentes, [33] [34], algunas de las cuáles son empleadas en las especificaciones de sistemas.

Esas especificaciones lógicas pueden tener un impacto directo sobre el desarrollo e implementación de software. Se ha dicho incluso [28], que una forma de producir conducta concreta en un sistema es tratar aquella descripción lógica como una *especificación ejecutable* que, al ser interpretada, dictará la conducta de cada agente. En GALATEA, aprovechamos este principio. Los agentes son especificados en términos lógicos y con código declarativo que, al ser interpretado, genera su conducta. Hemos, inclusive, realizado algunos experimentos en procura de mecanismos sistemáticos de traducción del código declarativo a procedimental [35], capitalizando en la interpretación procedimental de la lógica clausal usada en la especificación del agente [20]. Creemos que la descripción lógica del agente podrá ser traducida sistemáticamente en un código imperativo, librando al programador de sistemas multi-agentes de un considerable esfuerzo en la codificación.

Así, la ingeniería del software parece compartir un objetivo común con la simulación de sistemas: el manejo de la complejidad con medios computacionales. Y las soluciones que se comienzan a ofrecer en ambos campos tienen por lo menos un elemento común: la especificación y representación de los subsistemas intencionales: los agentes.

3.2 Una teoría de simulación de sistemas multi-agente

En general, una teoría es una *"suposición o sistema de ideas que explican algo"* (Oxford Dictionary). Los matemáticos tienen una definición un poco más precisa: *"Una colección de proposiciones que ilustran los principios de un asunto o materia de conocimiento"* (.ibid). En la simulación de sistemas, el trabajo de los pioneros ha hecho coincidir esas dos definiciones, de manera que en la *teoría de simulación* [10, 9] se encuentra una explicación general de lo que es un sistema, sus componentes y sus reglas de cambio, planteados como una colección de proposiciones matemáticas formales, tal como ilustramos en la seción3.1.1.

El objetivo de Zeigler y los otros al plantear esa formalización, además de proveer la explicación que se espera de una teoría, es proveer a los desarrolladores de **simuladores** de una especificación independiente de la plataforma computacional seleccionada para la implementación.

En esta sección vamos a presentar una teoría de simulación de sistemas multi-agentes exactamente con esos mismos propósitos. Esta teoría, bosquejada originalmente en [8], se ha constituido en la especificación básica de nuestro simulador multi-agente. De esta forma, la discusión sobre lo que significa simular un sistema multi-agente se separa de la discusión sobre la implementación del simulador que la soporta.

Separar la especificación de la implementación es una estrategia de trabajo reconocida y recomendada por los ingenieros de software modernos [36, 37, 38, 39]. Las especificaciones formales no son tan populares en Ingeniería del software. Nuestra insistencia en un formalismo responde a esa tradición en simulación, pero también a la necesidad de integrar la especificación con dos formalizaciones anteriores: la teoría de sistemas multi-agentes de Ferber y Müller [40] y la especificación lógica de un agente inteligente de Kowalski [41, 20]. Estas dos especificaciones son formales por varias razones que iremos ilustrando durante la discusión. No obstante, GALATEA no depende exclusivamente de las formalizaciones lógico-matemáticas que mostraremos en esta sección. En el capítulo 5 presentamos la especificación UML[8] preferida por los ingenieros de software orientado por objetos.

La presentación de la teoría está organizada de la siguiente manera: Primero, ampliamos las razones para desarrollar una nueva teoría y presentamos una revisión de la teoría de sistemas multi-agentes de Ferber y Müller. Luego, presentamos la jerarquía de agentes propuesta por Genesereth y Nilsson a mediado de los años ochenta del siglo pasado, la cual fue empleada por Ferber y Müller para crear una jerarquía correspondiente de sistemas multi-agentes. Para efectos de la teoría, hemos extendido

[8]UML (*Unified Modeling Language*)

esa jerarquía incluyendo un agente basado en lógica [20] que es "reactivo" y "racional" a la vez, finalmente y hemos especificado un nuevo tipo de sistema multi-agente basado en ese agente.

3.2.1 Razones para una nueva teoría

Como dijimos, la nueva teoría pretende servir como especificación formal de una plataforma de simulación de sistemas multi-agentes. Con el proyecto GALATEA, estamos extendiendo un lenguaje de simulación preexistente y maduro: El GLIDER, descrito en la sección 4.1, con las abstracciones necesarias para permitir a los modelistas representar sistemas con entidades autónomas (los agentes) que perciben y actúan sobre su ambiente.

En GLIDER, como se mencionó anteriormente, un sistema es concebido como una colección estructurada de objetos que intercambian mensajes, donde ese intercambio y procesamiento de mensajes está asociado a la ocurrencia de eventos. En GLIDER, modelar un sistema es describir una **red** de nodos, cada uno con una descripción procedimental de su conducta y de cómo, cuando y con cuáles otros nodos intercambiará mensajes.

Uno de los objetivos fundamentales de esta tesis es completar el enriquecimiento de la sintaxis y la semántica GLIDER para permitir la descripción de agentes y de sistemas multi-agentes. Los agentes representan, en el modelo de simulación, a aquellas entidades que en el sistema modelado pueden percibir su ambiente, tienen sus propias metas y creencias y actúan, siguiendo esas creencias, para alcanzar esas metas, posiblemente cambiando el ambiente durante ese proceso.

Enriquecer a GLIDER de esa manera requiere más que un conjunto adicional, *ad hoc*, de constructos lingüísticos. Hemos extendido la teoría de simulación para explicar la conducta de esos objetos especializados: los agentes, basándonos en las formalizaciones de la conducta de agentes presentadas los trabajos en Inteligencia Artificial que citamos al comienzo de esta sección.

Nuestro objetivo final es disponer de una *familia* de lenguajes, apoyada por una única plataforma de computación, que nos permita modelar y simular sistemas multi-agentes. Disponer de lenguajes de diversa naturaleza[9] es, una contribución muy importante a un enfoque *interdisciplinario* para modelado y simulación de sistemas.

3.2.2 Una teoría de influencias y reacciones

Atendiendo a ese interés interdisciplinario, muchos lenguajes y plataformas de simulación han sucumbido a la tentación de combinar diversos formalismos en una misma

[9]desde los lenguajes tradicionales de corte procedimental hasta lenguajes declarativos basados en lógica, pasando por los lenguajes orientados a una red de nodos y los orientados a objetos

herramienta para simulación. Es común, por ejemplo, entre las plataformas para simulación de sistemas dinámicos [42] permitir reglas de selección *if-then-else* como parte del modelo matemático sin mucho reparo en el significado del modelo.

En GALATEA pretendemos servir al enfoque interdisciplinario ofreciendo también una familia de lenguajes, cada uno con su propio basamento formal. Sin embargo, se ha dispuesto para que una misma teoría unifique la semántica de esos lenguajes en un todo coherente y útil para el simulista que desea analizar el significado de sus modelos. Puesto que varios de los lenguajes están orientados al modelado y programación de agentes, la teoría tiene que dar cuenta de lo que es cada agente y todo el sistema multi-agente. Esa es la razón para apoyarnos en una teoría de sistemas multi-agentes.

La razón para escoger la teoría de simulación que presentan Ferber y Müller [40] es que ellos ofrecen una manera de conectar la dinámica interna de cada agente, con la dinámica del universo en el que esos agentes se desenvuelven. Esa manera tiene que ver con una extensión a la forma habitual de describir sistemas dinámicos, que explicaremos a continuación.

Ferber y Müller describen un sistema dinámico usando una especie de estado global enriquecido, con el cual el universo es descrito en término de dos tipos de componentes del estado global: Las *influencias* y el *estado del ambiente*. Las influencias representan *"las acciones propias del agente, con las que intenta modificar el curso de los eventos que ocurrirán"* (.ibid). A través de las influencias los agentes intentan alterar el ambiente y en consecuencia alteran su propia historia. Por otro lado, el estado del ambiente esta representado por las variables de estado del sistema.

Así, las influencias son "indicadores" de lo que los agentes están tratando de hacer en el ambiente y constituyen un registro detallado de los cambios que el agente está "tratando" de causar. Este registro sirve para distinguir esas "intenciones" de su realización. La realización de las intenciones es entendida por Ferber y Müller como la *"reacción"* del ambiente ante las influencias que todos los agentes postulan en un momento dado. De esta forma es que las influencias están asociadas a las intenciones de los agentes y los cambios que de hecho tienen lugar en el ambiente están asociados a la reacción de este ante las influencias de los agentes.

Formalmente, Si definimos Γ como el conjunto de las posibles influencias que postulan los agentes, Σ como el conjunto de los posibles estados del ambiente, Op como el conjunto de los posibles operadores con que cuenta el agente para generar influencias y Λ como el conjunto de las posibles leyes del sistema, tales que:

$$\sigma \in \Sigma \qquad \sigma : \text{Estado actual del ambiente} \tag{3.1}$$

$$\gamma \in \Gamma \qquad \gamma : \text{Influencias postuladas actualmente} \tag{3.2}$$

$$\lambda \in \Lambda \qquad \lambda : \text{Ley} \tag{3.3}$$

$$op \in Op \qquad op : \text{Operador} \tag{3.4}$$

el intento del agente por influenciar al ambiente se define como:

$$\begin{aligned} \mathit{Acción} &: Op \times \Sigma \times \Gamma \rightarrow \Gamma & (3.5)\\ \gamma' &= \mathit{Acción}(op, \sigma, \gamma) & (3.6) \end{aligned}$$

y la reacción del ambiente ante estas influencias se define como:

$$\begin{aligned} \mathit{Reacción} &: \Lambda \times \Sigma \times \Gamma \rightarrow \Sigma & (3.7)\\ \sigma' &= \mathit{Reacción}(\lambda, \sigma, \gamma) & (3.8) \end{aligned}$$

Usando estas definiciones es posible describir la dinámica del sistema como una tupla de la forma

$$< \Sigma, \Gamma, Op, \Lambda, \mathit{Acción}, \mathit{Reacción} >$$

que permite que los agentes ejecuten acciones simultáneas. La evolución del sistema se define como una función infinita recursiva tal que

$$\begin{aligned} \mathit{Evolución} &: \Sigma \times \Gamma \rightarrow \tau & (3.9)\\ \mathit{evolución}(\sigma, \gamma) &= \mathit{evolución}(\mathit{paso}(\sigma, \gamma)) & (3.10) \end{aligned}$$

donde τ se define como una función que no retorna valor o que retorna valores en un dominio de errores, y $\mathit{paso}()$ definida como

$$\mathit{Paso} : \Sigma \times \Gamma \rightarrow \Sigma \times \Gamma \quad (3.11)$$

reúne todas las influencias producidas por los agentes, la cual viene determinada por la conducta del agente mismo, y la dinámica del ambiente. Esta función indica además como reacciona el mundo de acuerdo a las leyes. La especificación de esta función depende del tipo de agentes involucrados en el sistema multi-agente que se describa, por lo cual haremos un paréntesis en el que explicaremos una jerarquía de agentes basada en la conducta y luego continuaremos la descripción de la teoría.

3.2.3 Una jerarquía de arquitecturas de agente

Como dijimos antes, una teoría de sistemas multi-agentes explica lo es que un agente. Un agente es una entidad que despliega una conducta y hace algo sobre su entorno. La experiencia indica, sin embargo, que se deben identificar distintos tipos de agentes para distinguir entre las diversas formas de desplegar las conductas.

En 1988, en su libro de inteligencia Artificial y Lógica, Genesereth y Nilsson propusieron, no sólo una clasificación, sino una jerarquía de agentes, ordenándolos en relación con sus capacidades racionales. La jerarquía está constituida a lo largo de

un sólo eje que lleva desde agentes que apenas perciben su ambiente y reaccionan con acciones reflejos que pueden cambiar el ambiente, hasta agentes que incorporan un sofisticado mecanismo de razonamiento y gestión de conocimiento, entre el percibir y el actuar.

CONOCEDOR Y RACIONAL:
Percibe, registra, razona y actúa
$\langle P_a, S_a, K_a, percibe_a, recuerda_a, razona_a \rangle$

HISTERÉTICO
Percibe, registra y actúa
$\langle P_a, S_a, percibe_a, recuerda_a, decide_a \rangle$

TROPÍSTICO
Percibe y actúa
$\langle P_a, percibe_a, actua_a \rangle$

Figura 3.13: Jerarquía de Genesereth y Nilsson

La figura 3.13 resume la jerarquía de Genesereth y Nilsson mostrando, además, las tuplas con conjuntos y funciones matemáticas que se emplean en caracterizar la máquina abstracta que representa a cada tipo de agente:

P_a: Descripciones parciales del estado del sistema que representan todo aquello que es percibido por el agente. $Pa \in \Sigma$.

S_a: Conjunto de estados internos del agente.

K_a: Conjunto de posibles almacenes del conocimiento acumulado por el agente. Cada uno de estos almacenes contiene la estructura que permite representar percepciones, metas, creencias, preferencias y prioridades.

$percibe_a()$: Función de percepción del agente.

$recuerda_a()$: Función de memorización del agente, actualiza las percepciones del agente incluyéndolas como parte del conocimiento del agente.

***Acción*:** Se refiere a la función que permite postular influencias en el ambiente. La forma de escoger la acción a ejecutar depende del tipo de agente:

$actua_a()$: Asocia directamente percepciones con acciones.

$decide_a()$: Se selecciona la acción que debe ejecutar de acuerdo al estado interno y las percepciones.

$razona_a()$: Se razona sobre la base del conocimiento acumulado por el agente antes de seleccionar la acción.

Ferber y Müller extienden esta jerarquía en su teoría multi-agente con una máquina que ubican en la base. Se trata del *operador*, una máquina que no percibe su entorno, pero que si es capaz de cambiarlo. Esta máquina representa aquellas conductas *naturales* causadas por objetos físicos inmersos en alguna dinámica (p.e. el barril que rueda cuesta abajo: No es un agente, pero puede causar muchos cambios en el ambiente).

Dávila y Tucci han preservado todas estas contribuciones, con algunos ajuste en la noción de operador y en el agente racional. Aquellas conductas naturales ya no son generadas por operadores, sino por componentes del sistema que tienen asociadas ciertas leyes de cambio, a las cuáles obedecen generando esas conductas, no intencionales (en el sentido de que no corresponden a las intenciones de ningún agente), que cambian el ambiente.

Dávila y Tucci incorporan un agente *reactivo-racional* inspirado en la especificación lógica propuesta por Robert Kowalski descrita en detalle en [20].

De esta forma, la jerarquía de agentes que conforma la teoría de soporte de la plataforma de simulación es como muestra figura 3.14, en la que se incluyen nuevos elementos:

Reactivo y racional:
Percibe, registra, razona acotadamente y actúa
$\langle P_a, S_a, K_a, R_a, percibe_a, recuerda_a, planifica_a \rangle$

Conocedor y racional:
Percibe, registra, razona y actúa
$\langle P_a, S_a, K_a, percibe_a, recuerda_a, razona_a \rangle$

Histerético
Percibe, registra y actúa
$\langle P_a, S_a, percibe_a, recuerda_a, decide_a \rangle$

Tropístico
Percibe y actúa
$\langle P_a, percibe_a, actua_a \rangle$

Operador o componente
Actúa
$\langle actua_a \rangle$

Figura 3.14: Jerarquía de Ferber y Müller extendida por Dávila y Tucci

R_a**:** conjunto de valores que representa la restricción del tiempo de razonamiento.

$planifica_a()$**:** el agente razona por un tiempo determinado y determina el conjunto de acciones (estrategia) a seguir.

3.2.4 Un agente reactivo y racional

En las jerarquías presentadas en las secciones anteriores, el agente racional aparece como opuesto al agente cuya conducta es producto de reflejos inmediatos. Las jerarquías parecen sugerir que hay un compromiso entre racionalidad y reactividad. En esta sección discutimos ambos conceptos y una propuesta para reconciliarlos en el diseño de un agente.

Podemos decir que racionalidad en el contexto de los agentes consiste de la capacidad de cada agente para asimilar los datos que recibe de su ambiente y, con esa información y con conocimiento válido de las reglas de cambio de su entorno, decidir como actuar para alcanzar sus metas e intenciones. Dotar a un agente con "racionalidad" significa, para nosotros, proporcionarle los medios de representación de aquel conocimiento e información de su entorno y de los mecanismos para que pueda planificar (decidir sus acciones) para alcanzar sus metas.

La propuesta de Dávila y Tucci [8] se construye sobre la teoría de Ferber y Müller, preservando las definiciones y conceptos de las secciones anteriores. Entre otros ajustes, se establece que la reacción del ambiente es, no sólo función de las influencias, y del estado actual del ambiente sino también de las leyes de cambio del sistema y en general del conocimiento que se tenga sobre el ambiente. Así, es posible definir las ecuaciones (3.6) y (3.8) como:

$$\sigma' = Reacción(\lambda, \sigma, \gamma) \tag{3.12}$$

$$\gamma' = Acción(op, \sigma', \gamma) \tag{3.13}$$

Esto junto con la consideración de las influencias como "mensajes de eventos", que explicaremos más adelante, abren la puerta al uso de esa teoría de sistemas multi-agentes como una teoría de simulación de sistemas.

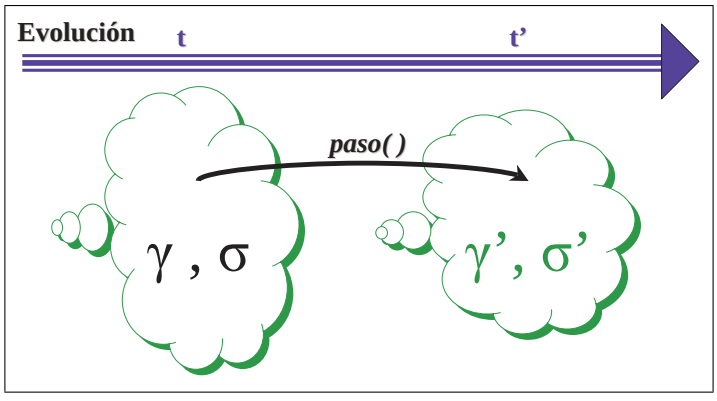

Figura 3.15: Evolución del estado global del sistema

Otro de agregados es la forma en que el sistema dinámico progresa en el tiempo de

t a t' ($t \rightarrow t'$). Este progreso se muestra en la figura 3.15 donde los valores del estado del ambiente (σ) y el conjunto de influencias (γ) evolucionan de acuerdo a conjunto de funciones de transición descritas anteriormente.

Esta descripción operativa (en término de máquinas abstractas) es el fundamento de una semántica operacional para los lenguajes usados en GALATEA (ver sección 5.4.2). En lo siguiente, los valores o variables destacados con el símbolo ($'$) corresponden a t'.

Por las razones mencionadas anteriormente, el agente racional en la teoría de simulación multi-agente está constituido, además de los elementos que ya se discutieron, por una función de memorización, así como por una versión de la función de acción que produce *"planes"* a partir de una representación de la situación actual y de las metas del agente.

Un aspecto de esta concepción de agente cuya importancia es cada vez más apreciada, es el hecho de que la racionalidad de cualquier agente real es *"acotada"*. Hay restricciones concretas en tiempo de procesamiento, espacio de almacenamiento y estructuras cognitivas, que impiden que el agente pueda deducir todas las consecuencias de sus creencias y conocimiento.

En GALATEA, hemos adoptado como modelo inicial de una agente, una especificación de un agente que puede razonar sobre descripciones lógicas de su entorno, metas y conocimiento, en forma acotada [20]. Una de las virtudes de esa especificación es que, aprovechando la estrategia para acotar el razonamiento (y junto con otras estrategias complementarias), ofrece también una forma de reconciliar racionalidad con reactividad en el mismo agente.

Según la especificación de Dávila (.ibid), un agente debe solapar el razonar con el actuar. Para ello, deben existir límites para el tiempo (o el espacio, o ambos) para el proceso de razonamiento y, al alcanzar esos límites, el agente podría tener que actuar siguiendo planes que no han sido completamente elaborados. Es decir, de haber tenido más tiempo, el agente podría haber tomado otro curso de acción.

El detalle clave en esta reconciliación de reactividad con racionalidad es que el agente va a estar siempre listo para la reacción [8], en comparación con un agente que trata de completar todo su racionamiento antes de actuar. Esta concesión, sin embargo, tiene un precio: el agente acotado puede que no tome el mejor plan de acción.

En términos formales, los límites en el razonar son restricciones numéricas sobre el tiempo de razonamiento. Sin embargo, para mostrar los detalles tenemos que presentar la descripción lógica o matemática del agente. Tomando la vía matemática, decimos que el agente está caracterizado por una función de conducta, que como mencionamos anteriormente debe estar incluida en la definición de *paso*() (ec. 3.11):

$$Conducta \quad : \quad \mathcal{T} \times R_a \times K_a \times \Gamma \rightarrow K_a \times \Gamma \tag{3.14}$$
$$< k'_a, g'_a, \gamma'_a > \quad = \quad conducta_a(t, r_a, k_a, g_a, \gamma) \tag{3.15}$$

con

$$k_a, g_a \in K_a, \qquad \gamma_a \in \gamma, \qquad r_a \in R_a$$

t : tiempo actual.
r_a : cantidad de tiempo para razonamiento del agente.
k_a : base de conocimientos del agente.
g_a : conjunto de metas del agente.
γ_a : conjunto de influencias que postula el agente.

donde $conducta_a()$ hace referencia a

$$Actualiza \quad : \quad \mathcal{T} \times \Gamma \times K_a \rightarrow K_a \tag{3.16}$$
$$k'_a \quad = \quad actualiza_a(t, percibe_a(\gamma), k_a) \tag{3.17}$$

y

$$Planifica \quad : \quad \mathcal{T} \times R_a \times K_a \rightarrow \Gamma \times K_a \tag{3.18}$$
$$< \gamma'_a, g'_a > \quad = \quad planifica_a(t, r_a, k'_a, g_a) \tag{3.19}$$

La función $actualiza_a()$ describe como la base de conocimiento del agente k_a, se actualiza con el registro de un conjunto de "perceptos" que el agente obtiene del entorno con la función $percibe_a()$, donde la representación de los perceptos podría ser

$$obs(P, t)$$

para indicar que el agente observa la propiedad P en el tiempo t.

El planificador de tareas del agente, representado por la función $planifica_a()$, reduce las metas del agente g_a en un nuevo conjunto de metas g'_a y las influencias que serán postuladas al ambiente γ'_a empleando para ello las reglas de conocimiento e información sobre el entorno registradas en k'_a. El proceso de planificación toma en cuenta el tiempo actual t y no puede prolongarse por más de r_a unidades de tiempo. Alterando el factor r_a, podemos forzar una conducta más reactiva (un r_a más pequeño hace que el agente observe su entorno con mucha frecuencia y por ende tardará más en llegar a "conclusiones profundas" acerca de su forma de actuar) ó más racional (un r_a más grande hace que el agente razone más profundamente antes de revisar su conocimiento del ambiente).

$planifica_a()$ es la forma matemática del predicado *demo*() descrito en detalle, como un programa lógico, en [20]. Es importante notar, sin embargo, que el predicado *demo*() es sólo una forma posible (correcta y conveniente) de dotar al agente de un motor de inferencia. Otras soluciones, particularmente una que tome en cuenta otras capacidades mentales para el agente, además del razonamiento temporal y planificación (como el aprendizaje automático) también podrían ser incorporadas eventualmente.

Con esta especificación general de un agente reactivo y racional, nos preparamos ahora para describir todo un sistema multi-agente.

3.2.5 El sistema multi-agente con ese agente racional

En las secciones anteriores explicamos que todo sistema es descrito por una función que especifica su progreso a través del tiempo (ec. 3.9). La función *evolución*() para nuestro sistema multi-agentes con n agentes racionales puede describirse como:

$$Evolución \quad : \quad S \times \mathcal{T} \times \Sigma \times \Gamma \rightarrow \tau \tag{3.20}$$

$$evolución(< s_1, s_2, \cdots, s_n >, t, \sigma, \gamma) \quad =$$

$$evolución(paso(< s_1, s_2, \cdots, s_n >, t, \sigma, \gamma)) \tag{3.21}$$

donde

$$s_a =< k_a, g_a > \tag{3.22}$$

describe el estado interno del agente a y puede obtenerse a partir de 3.15.

Note que se trata simplemente de una función recursiva que se invoca a sí misma, luego de invocar a la función *paso*() que describe la transición de un estado global al siguiente.

Desde luego, los detalles interesantes de cómo un estado global se trasforma en otro tomando en cuenta la evolución de los agentes, las influencias que estos postulan y la reacción del ambiente a dicha influencia, tienen que ser incluidos en la definición de *paso*(). Por tanto

$$Paso : S \times \mathcal{T} \times \Sigma \times \Gamma \rightarrow S \times \mathcal{T} \times \Sigma \times \Gamma$$

puede ser representada como

$$<< s'_1, s'_2, \cdots, s'_n >, t', \sigma', \gamma' >= paso(< s_1, s_2, \cdots, s_n >, t, \sigma, \gamma) \tag{3.23}$$

la cual hace referencia a varias funciones, entre ellas a la función $conducta_a()$ de la

ecuación (3.15) para obtener los s_a y los γ_a y la función *reacción*() definida como

$$Reacción : \Lambda \times \beta \times \mathcal{T} \times \Sigma \times \Gamma \rightarrow \Sigma \times \Gamma \tag{3.24}$$

$$< \sigma', \gamma' >= reacción(\Lambda, \beta, t, \sigma, \gamma \cup_a \gamma_a) \tag{3.25}$$

donde

Λ : descripción del sistema a través de las leyes de cambio.
β : estructura e información de soporte acerca del sistema.
γ_a : conjunto de influencias que postula el agente a.

Esta descripción matemática dice, en esencia, que el sistema evoluciona debido a dos causas generales:

(a) por la respuesta del ambiente al estado actual y a las influencias de los agentes, procesadas en *reacción*() usando descripciones del cómo debe cambiar el sistema (Λ, β).

(b) por el progreso de los agentes en la producción de las influencias a través de la función $conducta_a()$.

Resta por presentar las transformaciones que conducen a Λ, β y a las descripciones del estado global actual en un nuevo estado (la definición de *reacción*()). Vamos a postergar esa discusión hasta la sección 5.4, la cual presenta el diseño detallado del lenguaje de modelado y de la plataforma de simulación. Allí aprovecharemos la definición de *reacción*() para postular una explicación operativa del significado de Λ y β: produciremos las bases para una *semántica operacional* del lenguaje que denominamos GALATEA.

Capítulo 4

Tributo al ancestro: GLIDER en GALATEA

Tomando en cuenta que GALATEA es una extensión de la plataforma GLIDER, en la sección 4.1 presentamos un breve resumen de los fundamentos de la plataforma de simulación GLIDER.

Entre los enfoques formales para modelado y simulación de eventos discretos, optamos por el formalismo DEVS, desarrollado por Zeigler [10]. Como vimos en la sección 3.1, éste formalismo proporciona una base matemática que soporta la especificación modular de sistemas de eventos discretos en forma jerárquica, lo que nos permite realizar extensiones que permitan incluir otros formalismos partiendo del análisis de comportamiento y de la estructura del sistema original [23].

En éste capítulo describiremos los detalles de diseño e implementación del simulador DEVS, desarrollado en Java, el cual se adecúa al paradigma de orientación por objetos y permite procesar modelos diseñados de acuerdo a la semántica del GLIDER.

Para la presentación de este diseño escogimos UML el cual nace del intento de unificar los métodos de Grady Booch [43] y James Rumbaugh [44]. La elección de este lenguaje de modelado se realizó tomando en cuenta que UML permite especificar, construir, visualizar y documentar los componentes de un sistema de software orientado a objetos. Sin restringir a un modelo estándar de desarrollo UML define un lenguaje de modelado que recomienda utilizar los procesos que otras metodologías tienen definidos.

4.1 Plataforma de simulación GLIDER

La plataforma de simulación GLIDER [45] esta compuesta por:

* El lenguaje de simulación GLIDER.
* Un compilador de programas GLIDER.
* Un ambiente o entorno de desarrollo para la construcción de modelos de simulación en GLIDER.

Hasta ahora el proyecto GLIDER ha abarcado estudios en distintos aspectos de simulación vinculados con simulación Continua y Discreta [6].

En [46, 47] se estudia el uso de la programación por objetos en simulación, y en particular su aplicación al GLIDER. En [48] se presenta un prototipo para el compilador GLIDER desarrollado en C++ que propone el uso de programación por objetos para realizar una extensión del GLIDER que permita el soporte a la simulación del cambio estructural.

En [49] se propone un método basado en simulación estructural para generar escenarios para procesos de cambio estructural en sistemas sociales donde los cambios son generados por interacción de actores con diferentes percepciones de la situación y diferentes repertorios de posibles metas y tácticas. En este método, los actores seleccionan las metas de acuerdo a sus percepciones de la estructura y el comportamiento del sistema. De estas percepciones y metas se seleccionan las tácticas a través de las cuales la estructura del sistema puede ser cambiada. Estos cambios son simulados por medio de reglas con parámetros, que pueden ser afectados por cambios aleatorios o experimentales. Los resultados de la simulación son escenarios que pueden ayudar en la toma de decisiones.

El prototipo del compilador desarrollado en [48] es evaluado en [50] para verificar el soporte a simulación combinada discreta-continua. En [50] se presentan modificaciones al GLIDER con la finalidad de dar soporte a la especificación de modelos de simulación orientados por objeto, y a la programación visual de modelos de simulación. Además se propone el uso de meta-nodos que representen a subsistemas con interfaces bien definidas para ser enlazados dentro de sistemas más generales. Esta propuesta permite la construcción de modelos multiformalismos jerárquicos y modulares basándose en conceptos de la teoría de sistemas.

4.1.1 Lenguaje de simulación GLIDER

El lenguaje GLIDER permite la especificación de modelos de simulación para sistemas continuos y de eventos discretos. En su diseño, se han tomado en cuenta facilidades para la ejecución simultánea y cooperativa de ambos tipos de sistemas.

GLIDER representa el sistema modelado como un conjunto de subsistemas que intercambian información (mensajes). Cada subsistema puede almacenar, transformar, transmitir, crear y eliminar información del estado del sistema, a través de un código enlazado. Este código especifica como cambian los estados del sistema cuando ocurre un evento en un subsistema. Ese intercambio y procesamiento de mensajes está asociado a la ocurrencia de eventos tal como se establece en la teoría de simulación de sistemas a eventos discretos (DEVS) de Zeigler [10]. En ese sentido, GLIDER está basado en DEVS.

Las características fundamentales del GLIDER son:

* Los subsistemas son representados por nodos de una red.
* Los nodos pueden ser de diferentes tipos de acuerdo a su función. Los tipos básicos son:

  G (Gate) Controla el flujo de mensajes.
  L (List) Simula disciplinas de colas.
  I (Input) Genera mensajes de entrada.
  D (Decision) Selecciona mensajes de acuerdo a reglas de selección indicadas.
  E (Exit) Destruye los mensajes.
  R (Resource) Simula recursos usados por los mensajes (entidades).
  C (Continuous) Resuelve sistemas de ecuaciones diferenciales ordinarias de primer orden.
  A (Autonomous) Permite programar eventos.
  O (Other) Permite al usuario definir sus propios nodos.
* La información es descrita y ejecutada a través de un código asociado a cada nodo.
* El intercambio de información entre los nodos es realizado a través del código haciendo uso de pase de mensajes, variables y archivos compartidos.
* La información se almacena en variables, archivos o listas de mensajes.
* Cada nodo tiene asociado un conjunto de nodos predecesores de los que puede recibir mensajes y un conjunto de nodos sucesores hacia los cuales puede enviar mensajes.

	Encabezado
`TITLE`	*Título del modelo*
`NETWORK`	*Descripción de la Red*
`INIT`	*Valores iniciales para variables*
`DECL`	*Declaración de variables*
`END.`	

Figura 4.1: Estructura de un programa GLIDER

Estructura de un programa GLIDER

La estructura de un programa GLIDER (figura 4.1), consta de cinco secciones:

Encabezado: Esta sección permite incorporar la descripción del sistema, así como también algunos comentarios.

Título: Esta sección permite incorporar un título al modelo.

Red: La sección de Red se inicia con la etiqueta `NETWORK`. En esta sección se describen las relaciones entre los nodos y el código que simula su comportamiento.

Los nodos poseen dos partes un *encabezado* donde se definen el nombre, el tipo, la multiplicidad, los sucesores y las variables locales y un *código* compuesto de elementos PASCAL y GLIDER.

Iniciación: La sección de Iniciación se indica con la etiquetar `INIT`, acá se realizan asignación de valores que serán usados en tiempo de ejecución de la simulación, tales como:

- Valores iniciales a arreglos.
- Valores iniciales para la capacidades de los nodos tipo R.
- Valores de funciones multivaluadas.
- Valores de parámetros en tablas de frecuencia.
- Valores de arreglos a partir de tablas de base de datos.
- Activación inicial de nodos.

Declaración: La sección de Declaraciones se inicia con la etiqueta `DECL` y según el tipo de declaración a realizar es necesario utilizar las etiquetas:

`TYPE`: Tipos de datos definidos por el usuario

`CONST`: Constantes

`VAR`: Variables
`NODES`: Nodos definidos por el usuario
`MESSAGES`: Mensajes
`GFUNCTIONS`: Funciones multivaluadas definidas por el usuario
`TABLES`: Tablas de frecuencia
`DBTABLES`: Tablas provenientes de base de datos
`PROCEDURES`: Procedimientos y funciones definidos por el usuario
`STATISTICS`: Solicitud de estadísticas para nodos y variables

El código GLIDER, como todo programa PASCAL, debe terminar con la palabra reservada `END.`

4.1.2 El compilador GLIDER

El Compilador para el lenguaje GLIDER ha sido desarrollado en PASCAL y su sintaxis esta basada en TurboPascal 6.0 Borland©. Este compilador permite que el usuario además de las estructuras, procedimientos y funciones GLIDER incorpore sus propias funciones y procedimientos PASCAL.

Como mencionamos anteriormente, los componentes básicos del simulador son los mensajes y los nodos. en esta sección revisamos los aspectos fundamentales de estos componentes y luego mostramos el algoritmo general del compilador.

Mensajes GLIDER

Los mensajes poseen varios atributos que clasificaremos según su función en:

Identificación. Un *nombre* que determina el tipo de mensaje y un *número* que indica cuantos mensajes del tipo indicado han sido creados.

Un mensaje se identifica unívocamente dentro de la red a través de su nombre y su número.

Ubicación. La ubicación de los mensajes en la red se registra almacenando el tiempo en que los mensajes pasan de una lista a otra.

El momento en que el mensaje entra en el sistema se identifica como *tiempo de creación*. Así mismo, el momento en que entra a la lista actual es *tiempo de entrada* y el momento en que debe salir de dicha lista es *tiempo de salida*.

Información. Es posible almacenar información dentro de los mensajes, esta información puede ser alterada en el transcurso del mensaje en la red a juicio del modelista.

La estructura de datos para almacenar la información es la *lista de campos*, la cual contiene el *nombre* del campo y el *valor* del mismo.

Nodos GLIDER

Como se mencionó anteriormente, cada nodo se especifica mediante el nombre, el tipo, la lista de sucesores y el código asociado.

El código asociado es ejecutado por los mensajes procesados por la red. Este código incluye:

* Instrucciones PASCAL.
* Instrucciones GLIDER.
* Procedimientos o Funciones GLIDER.
* Instrucciones del sistema.

El procesamiento de los mensajes dentro de los nodos se hace a través de listas, existiendo dos tipos básicos de lista: `EL` (*Entry List*) que contiene los mensajes a ser procesados por un nodo y `IL` (*Internal List*) que contiene los mensajes durante su permanencia en el nodo.

La ejecución del código de un nodo es llamada **activación del nodo**. Es posible programar la activación de ciertos nodos en el tiempo de simulación. El esquema de activación de los nodos se almacena en la lista denominada `FEL` (*Future Event List*).

Cada elemento de la `FEL` representa un evento pendiente y contiene una referencia al nodo que debe ser activado y el tiempo en el que la activación debe ocurrir.

El orden de activación de los nodos representa la secuencia de eventos que ocurren en la simulación. La activación de un nodo puede ocasionar cambios en las variables de estado, en la ejecución de procedimientos y en el procesamiento de mensajes.

Las características básicas de los nodos son:

`G`: Gate.

* Controla flujo de mensajes.
* Posee `EL`.
* Se activa y se recorre.
* Retiene los mensajes en la `EL`.
* La instrucción `STATE` se usa para cambiar el estado de la compuerta cuando el nodo es activado por la `FEL`.

`L`: List.

* Simula disciplinas de colas.

- Posee `IL` y `EL`.
- Se activa y se recorre.
- Los mensajes que entran a la `EL` son pasados a la `IL` de donde salen según el orden especificado FIFO, LIFO, Ascendiente o Descendiente.

`I`: Input.

- Genera mensajes de entrada.
- No posee ni `IL` ni `EL`.
- Se activa.
- Crea el mensaje y lo envía al nodo sucesor.

`D`: Decision.

- Selecciona mensajes de acuerdo a reglas de selección indicadas.
- Posee una `EL` por cada predecesor.
- Se activa y se recorre.
- Los mensajes son sacados de las `EL` y enviados a los nodos sucesores de acuerdo a las reglas de selección indicadas.

`E`: Exit.

- Destruye los mensajes.
- Posee `EL`.
- Se activa y se recorre.
- Se procesan los mensajes de la `EL` se ejecuta el código y se destruye el mensaje.

`R`: Resource.

- Simula recursos usados por los mensajes (entidades).
- Posee `IL` y `EL`.
- Se activa y se recorre.
- Se le asocia un valor real que determina la capacidad del recurso.
- Los mensajes en la `EL` representan entidades que demandan cierta porción de la capacidad del recurso durante cierto periodo de tiempo.
- Se examina la `EL` y se verifica la disponibilidad del recurso, si hay capacidad los mensajes se envían a la `IL`, si no permanecen en la `EL`.
- La salida del recurso se programa automáticamente, el mensaje sale de la `IL` y se envía al sucesor. Si se trata de un sucesor múltiple se envía una copia del mensaje a cada sucesor.
- El código posee dos partes asociadas a los mensajes de entrada y salida respectivamente.
- El código de salida se activa a través de la `FEL`.
- El código de entrada puede activarse por revisión .

`C`: Continuous.

- Da solución a sistemas de ecuaciones diferenciales ordinarias de primer orden.
- No posee ni `IL` ni `EL`.
- Se activa.
- El proceso de solución de la ecuación se inicia al activarse el nodo.
- Es posible detener y reanudar el proceso de solución de la ecuación a través de instrucciones GLIDER.
- Durante la activación del nodo se programan automáticamente las activaciones correspondientes a cada paso de integración. Estas activaciones no causan revisión de la red.

`A`: Autonomous.

- Permite programar eventos.
- No posee ni `IL` ni `EL`.
- Se activa.
- Puede activarse a sí mismo y a otros nodos, permite realizar cambios en las variables y enviar mensajes.

`O`: Other.

- Permite al usuario programar sus propios nodos.
- No posee ni `IL` ni `EL`. Sin embargo el usuario puede requerirlas explícitamente si las necesita.
- Se activa y se recorre.
- El manejo de listas debe realizarse a través de instrucciones y procedimientos GLIDER.

Algoritmo General de un programa GLIDER

El procesamiento de la red se inicia cuando un nodo es activado por un evento, el código del nodo es ejecutado y se inicia un recorrido exhaustivo de la red.

En el recorrido aquellos nodos que no poseen `EL` o cuya `EL` no contiene elementos son ignorados. El resto de los nodos se activan, es decir, se ejecuta su código. Cuando el último nodo es revisado, el recorrido continúa con el primer nodo hasta que se alcanza el nodo activado al inicio del evento.

Este recorrido permite considerar todas las consecuencias que el evento actual ocasiona sobre el resto de los nodos. Si durante el recorrido ocurre algún movimiento de mensajes se reinicia el recorrido. Si ocurre un recorrido completo sin movimiento de mensajes se da por finalizado el evento actual y se extrae un nuevo evento de la `FEL`.

Al extraer un evento de la `FEL` se actualiza el tiempo de simulación y se continúa el proceso de simulación hasta que se cumple la condición de fin de simulación proporcionada por el usuario.

4.1.3 Entorno de Desarrollo GLIDER

El entorno de desarrollo del GLIDER [45] pretende proporcionar al usuario un ambiente de trabajo para las tareas de desarrollo de un modelo de simulación que se inicia con el diseño del modelo mediante el uso de esquemas gráficos, continua con la programación y pruebas y finaliza con la experimentación.

El ambiente consta de varios componentes:

Monitor. Permite seleccionar las tareas a ejecutar en el sistema y facilita la manipulación de archivos.

Editor de Texto. Permite edición en múltiples ventanas, diálogos para selección de archivos, manejo de teclado y ratón, etc.

Editor Gráfico. Permite la creación del modelo a través de selección de iconos gráficos.

Animador. Una vez ejecutada la simulación es posible obtener una animación diferida.

Controlador. Permite controlar la experimentación.

El algoritmo implementado en el simulador GLIDER se basa en DEVS. Sin embargo, no existe una especificación formal que relacione o explique la implementación de GLIDER en DEVS. Con GALATEA procuramos corregir este lapsus metodológico basando el algoritmo del simulador (ver sección 5.2.1) en la teoría de simulación de sistemas multi-agentes que mostraremos a continuación.

4.2 Aspectos de Diseño

Como mencionamos anteriormente, en el proceso de modelado y simulación de sistemas, el simulador es el encargado de generar las trayectorias del modelo permitiendo la manipulación y el análisis de dichas trayectorias con el fin de representar el comportamiento del sistema modelado bajo ciertas condiciones.

Uno de los objetivos fundamentales de esta tesis es completar el enriquecimiento de la sintaxis y la semántica GLIDER para permitir la descripción de agentes y de sistemas multi-agentes. Para cumplir con dicho objetivo realizamos una redefinición del simulador que incluye aspectos de diseño, tales como:

* Definición de la estructura de datos.

* Definición de los métodos para manipular dicha estructura.

* Desarrollo de las librerías que permitirán simplificar las tareas del compilador.

Como se describe en la sección 4.1 los componentes básicos del GLIDER son los mensajes y los nodos. En nuestro caso redefinimos[1] estos componentes e incorporamos una variedad del componente evento como componente esencial. El diagrama de

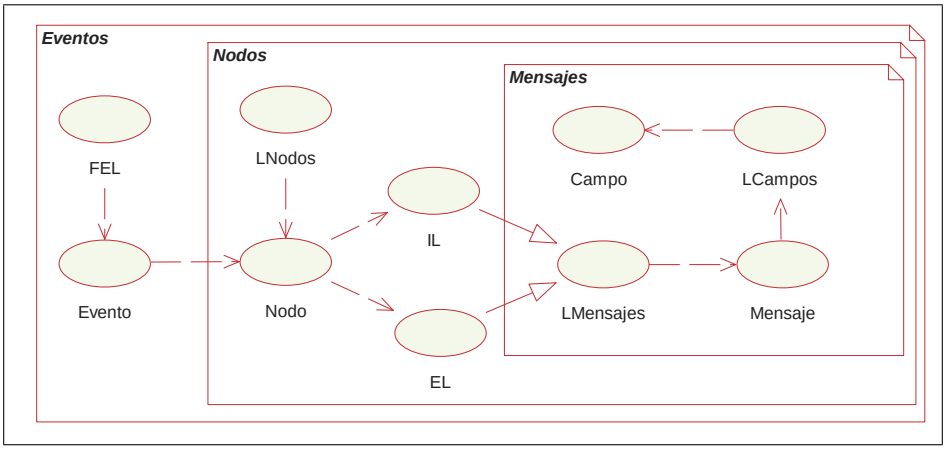

Figura 4.2: Componentes básicos del simulador

casos de uso de la figura 4.2 muestra las relaciones entre los componentes:

Eventos.

Los eventos, como mencionamos anteriormente, se utilizan para controlar la evolución del sistema modelado en el tiempo. Estos eventos son almacenados en una lista de eventos ordenada denominada `FEL` (*Future Event List*).

Cada evento asocia el nodo a ser activado con el tiempo en el que debe ocurrir dicha activación.

Nodos.

Representan los subsistemas componentes del sistema a modelar. Cada nodo esta definido por dos listas que contienen mensajes: la lista externa, `EL`, donde permanecen los mensajes antes de ser procesados en el nodo y la lista interna, `IL`, que almacena los mensajes que están siendo tratados.

Los nodos se almacenan en una lista, `LNodos`, a la que tendrá acceso el simulador para activar el nodo indicado en el momento preciso.

[1]Esta redefinición de los componentes obedece a la necesidad de extender la funcionalidad del simulador para incorporarlo a la nueva plataforma de simulación

Mensajes.

Representan las entidades que viajan a través de la red y que almacenan información en las listas `LCampos`. Estas entidades se almacenan en listas, `LMensajes`, y el recorrido que realizan en la red se representa a través de un cambio de una lista a otra.

En este diagrama, como se sugiere en [48], utilizamos Listas para almacenar la información. Esta selección se realizó tomando en cuenta que las listas permiten una manipulación eficiente y eficaz de su contenido y que por ser estructuras dinámicas se adaptan perfectamente a las exigencias del modelista facilitando además la interacción dentro del simulador.

4.2.1 Listas

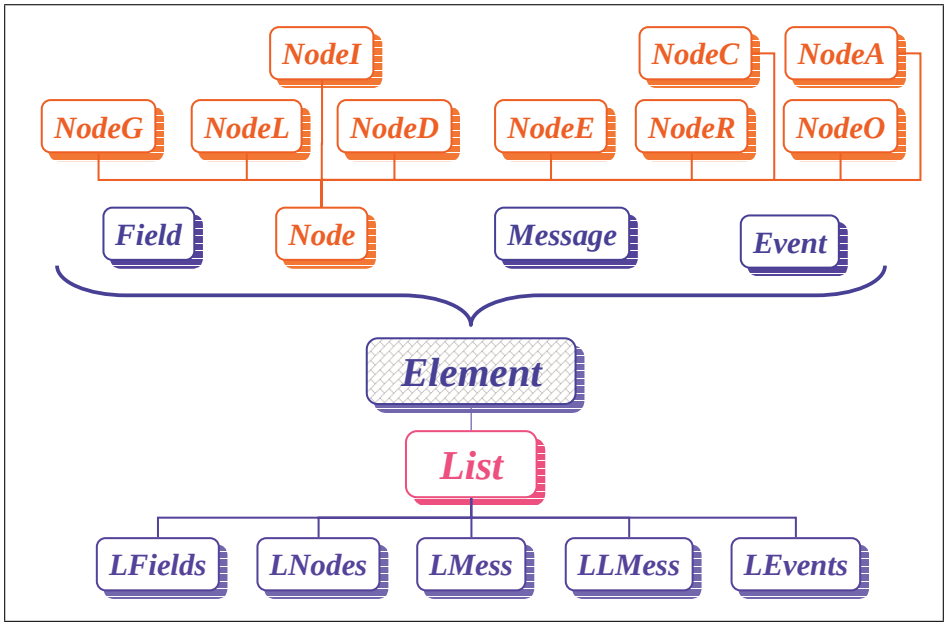

Figura 4.3: Relación entre los componentes del simulador

Como describe el diagrama de uso (figura 4.2) las listas corresponden a la estructura sobre la que se apoya el simulador. Por otro lado, la figura 4.3 muestra como se relacionan los componentes básicos del simulador con las listas:

* los componentes del simulador denominados eventos, nodos, mensajes y campos son representados por `Event`, `Node`, `Message` y `Field` respectivamente, los cuales pueden encapsularse en `Element`. Este encapsulamiento permitirá a dichos componentes formar parte de listas.

* `List` puede extenderse para implementar `LEvents`, `LNodes`, `LMess`, `LLMess` y `Field` que corresponden a los componentes tipo lista del simulador que almacenan eventos, nodos, mensajes, listas de mensajes y campos respectivamente.

* `Node`, que representa al nodo, puede extenderse para implementar los tipos de nodos reconocidos por el simulador (G,L,I,D,E,R,C,A,O).

En particular, el diagrama de la figura 4.4 muestra el esquema básico para las lista utilizadas, que se destacan por ser circulares y poseer enlace simple entre sus elementos.

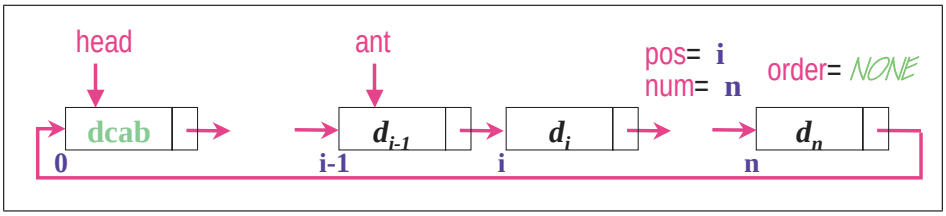

Figura 4.4: Diagrama básico de lista

Cada elemento de la lista esta compuesto por un dato genérico y un apuntador al siguiente elemento. Básicamente, esta forma en que se define el elemento permite que cualquier objeto pueda pertenecer a una lista, como muestra la figura 4.3.

Para facilitar la identificación de las listas se les asigna un nombre (`name`), y para facilitar el recorrido y las operaciones propias de manipulación de datos, las listas se indexan haciendo uso de un apuntador a la cabecera de la lista, un indicador de la posición del elemento actual, un apuntador al elemento anterior al que esta en uso actualmente, un indicador del número de elementos (`num`) y un indicador del orden relativo de la lista.

El elemento cabecera de la lista apuntado por `head` contiene información relativa a todos los elementos de la lista. Gracias al apuntador que indexa este elemento, el acceso a los datos es directo.

El indicador de la posición actual (`pos`) y los apuntadores al elemento anterior (`ant`) y siguiente (`next`) permiten acceder fácilmente a los elementos anterior, actual y siguiente de la lista.

Es posible definir el orden de la lista a través del indicador (`order`) que puede tomar varios valores:

NONE: Desordenada

FIFO: Los elementos son almacenados según el orden de llegada y se extraen de la lista en el mismo orden. Se identifica con el acrónimo de *First Input First Output* (primero en entrar, primero en salir).

LIFO: Los elementos son almacenados según el orden de llegada y se extraen de la lista en el orden inverso. Se identifica con el acrónimo de *Last Input First Output* (último en entrar, primero en salir).

ASCEN: Los elementos son almacenados usando el orden ascendente. Por lo tanto, es necesario definir el atributo de ordenamiento. La extracción de los elementos no sigue un patron pre-determinado.

DESCEN: Los elementos son almacenados usando el orden descendente. Por lo tanto, en este caso también necesario definir el atributo de ordenamiento y la extracción de los elementos tampoco sigue un patron pre-determinado.

Los métodos para manipular listas incluyen los métodos básicos de consultar, definir y actualizar los atributos antes mencionados. Los métodos restantes se pueden clasificar de acuerdo a la función que cumplen en:

Básicos.

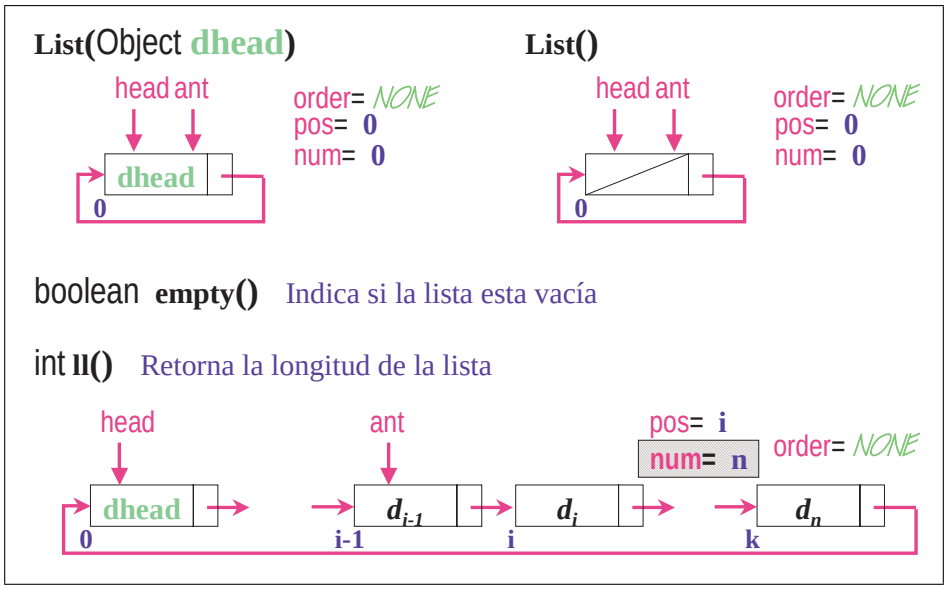

Figura 4.5: Métodos básicos de lista

Corresponde a los constructores que permiten definir objetos tipo Lista y los métodos que permiten haciendo uso del indicador del número de elementos de la lista (`num`), consultar el estado de la lista, indicando si la lista esta vacía, `empty()`, y la longitud actual de la lista `ll()`. Cabe destacar que el elemento que ocupa la

cabecera es considerado parte de la información de la lista, por tanto, no se considera su presencia al solicitar la longitud de una lista o al consultar si dicha lista está vacía.

Existen dos formas de definir una nueva lista: el constructor nulo corresponde a una lista sin cabecera, y el constructor con información que corresponde a una lista que en el elemento cabecera contiene información referente a la lista.

Adicionalmente, incluimos constructores que permiten definir valores para los indicadores del nombre de la lista, el tipo de orden.

Consultas.

Corresponde a los métodos para realizar consultas de la información almacenada en la lista. Esta operación no cambia el contenido de la lista, solo retorna información relativa al contenido de la lista. La figura 4.6 muestra los métodos básicos de consulta:

`getHead()`

Permite consultar el elemento cabecera de la lista. Retornando la información asociada a la lista o un apuntador nulo en caso de que la cabecera este vacía.

`getDat()`

Estos métodos permiten consultar el contenido de los elementos de la lista. En el caso básico, es posible consultar el elemento actual. Si se desea consultar el elemento de una posición particular el método se desplaza en la lista hasta la posición dada, si es válida, consulta el elemento como en el caso anterior para luego retornar a la posición en que la lista se encontraba al inicio del proceso. En ambos casos si no se encuentra el dato se retorna un apuntador nulo.

Es posible también consultar la existencia de un dato en particular, en este caso el método consulta si la posición del dato dado es válida y si es válida retorna que el dato existe.

`getPos()`

Estos métodos permiten consultar la posición de los elementos de la lista. En el caso básico, es posible consultar la posición del elemento actual, si se desea consultar la posición de un dato particular el método consulta la posición del dato dado, si es válida se desplaza en la lista hasta la posición obtenida, y consulta la posición del elemento como en el caso anterior para luego retornar a la posición en que la lista se encontraba al inicio del proceso. En ambos casos si no se encuentra el dato se retorna posición nula.

Adicionalmente, el método `getName()` permite consultar el nombre de la lista, el método `getOrder()` retorna información asociada al tipo de orden de la lista, el método `getAnt()` consulta el dato del elemento anterior y `getNext()` consulta el dato

del elemento siguiente. Estos dos últimos métodos respetan el hecho de que la lista es circular y por tanto en los extremos están conectados el primer y el último elemento.

Avances.

Corresponde a los métodos `setPos()` para cambiar las referencias respecto al elemento actual en la lista. Esta operación no cambia el contenido de la lista, solo reubica el apuntador al elemento anterior y actualiza el indicador de la posición del elemento actual. La figura 4.7 muestra el método básico para avanzar al elemento siguiente en una lista: el apuntador al anterior se ubica en el elemento actual y el indicador de posición avanza.

Es importante destacar que en caso de alcanzar el extremo de la lista, dado que la lista es circular, el próximo elemento es el que ocupa la posición 1, por tanto, el apuntador al elemento anterior queda ubicado en la cabecera y el indicador de posición se actualiza al primer elemento.

En el caso en que se desee avanzar hasta una posición dada el método verifica que la posición deseada es válida y en caso de serlo, se recorre la lista de un elemento al siguiente, como se explico anteriormente, a partir de la cabecera o del elemento actual, seleccionando el recorrido de menor longitud.

Adicionalmente es posible avanzar hasta la posición que ocupa un dato particular el método debe consultar la posición que corresponde al dato proporcionado, si el dato existe, se procede a avanzar al siguiente elemento como se explico anteriormente.

Agregar.

Corresponde a los métodos `add()` que permiten incluir elementos en la lista. La figura 4.8 muestra el método básico para agregar un elemento en una lista desordenada: el dato almacenado se encapsula en un elemento de lista, el elemento anterior apuntará al nuevo elemento, el nuevo elemento ocupará la posición del elemento actual y se incrementa el número de elementos de la lista.

En el caso de listas ordenadas el método debe consultar la posición que corresponde al dato proporcionado, se desplaza en la lista hasta encontrar la posición obtenida y procede a agregar el elemento como se explico anteriormente para luego retornar a la posición en que la lista se encontraba al inicio del proceso.

Eliminar.

Corresponde a los métodos `remove()` que permiten extraer elementos de la lista. La figura 4.9 muestra el método básico para eliminar el elemento que ocupa la posición actual de la lista: el elemento anterior apuntará al elemento siguiente y el dato contenido en el elemento extraído es retornado.

Además, se muestra el caso en el que se elimina un dato particular, en este caso el método debe consultar la posición que corresponde al dato proporcionado, si este existe, se desplaza en la lista hasta encontrar la posición obtenida y procede a extraer el elemento como se explico anteriormente para luego retornar a la posición en que la lista se encontraba al inicio del proceso.

Modificar.

Corresponde a los métodos internos que el simulador utiliza para modificar el contenido de la información almacenada en la lista. La figura 4.10 muestra los métodos básicos para modificar el contenido de la lista:

`setHead()`

Permite actualizar el contenido de la cabecera de la lista ubicando el dato proporcionado como cabecera.

`setDat()`

Permite actualizar el contenido del dato actual ubicando en su posición el dato proporcionado.

`setName()`

Permite actualizar el nombre actual ubicando en su posición la cadena de caracteres proporcionada.

Tomando en cuenta que el uso de estos métodos puede provocar inconsistencias en el simulador no son de acceso al modelista.

Una vez descritas las listas que se utilizarán en el simulador, pasamos a describir detalladamente cada uno de los componentes del simulador.

4.2.2 Componentes básicos del simulador

Como muestran el diagrama de casos de uso de la figura 4.2 y la figura 4.3 los componentes básicos del simulador interactúan con otros componentes. En esta sección explicaremos con detalle cada uno de los componentes y su interacción.

Eventos.

La figura 4.11 muestra el Objeto `Event` que representa los eventos. Este objeto contiene dos atributos: el primero, `comp`, almacena un apuntador al componente que va a ser activado cuando el evento se este procesando y el segundo, `ta`, es el indicador del tiempo en el cual se debe procesar dicho evento, denominado comúnmente en simulación tiempo de activación.

Para el caso del simulador básico los nodos son los únicos componentes que pueden encapsularse como eventos. Sin embargo, como veremos en el próximo capítulo, los agentes y sus influencias también serán incluidos en la descripción de la evolución del sistema que se esta simulando. Este hecho justifica la modificación de la estructura de datos para eventos con respecto a la utilizada en GLIDER (ver sección 4.1).

Además de los métodos básicos de consulta, definición, inicialización y modificación de los atributos, el objeto `Event` debe incluir algunos métodos:

`equals()`

Permite verificar el orden parcial de los eventos. Este orden se lleva a cabo de acuerdo al atributo asociado al tiempo de activación del evento. Este orden parcial permitirá mantener ordenada la lista de eventos.

`getNode()`

Retorna, en caso de existir, el nodo asociado a un evento dado.

Los eventos, como muestra el diagrama de casos de uso de la figura 4.2 se almacenan en una lista de eventos denominada `FEL`, la cual sirve de programador u organizador de los eventos que se procesaran durante la simulación. Esta lista sigue las pautas descritas en la sección anterior para las listas del simulador siendo definida como una lista ordenada ascendentemente (*ASCEN*)

Nodos.

La figura 4.12 muestra los objetos asociados a los nodos en el simulador. El objeto `Node` representa al nodo como tal, mientras que el objeto `HNode` representa la cabecera de la lista de nodos.

El objeto `Node` contiene diez atributos: El atributo `name` que almacena el nombre del nodo y el atributo `ino` que indica el índice del nodo facilitan la identificación unívoca de cada nodo. Dado que es posible tener nodos múltiples que comparten el mismo nombre el índice permite distinguirlos.

Los atributos que corresponden al recorrido y activación de los nodos son: `succ[]`, que apunta a la lista de nodos sucesores y facilita el envío de mensajes; y los atributos `IL` y `EL` que apuntan a lista de mensajes que corresponden a la listas interna y externa del nodo respectivamente.

El atributo `type` que representa el tipo asociado al nodo, este indicador puede tomar varios valores: *G* (Gate), *L* (List), *I* (Input), *D* (Decision), *E* (Exit), *R* (Resource), *C* (Continuous), *A* (Autonomous), *O* (Other).

Los atributos que determinan particularidades en los nodos son: `nmess`, que indica el número de mensajes generados en el nodo, `cap`, que indica la capacidad máxima del nodo y `use` que indica la cantidad de recurso utilizado.

Además de los métodos básicos de consulta, definición, inicialización y modificación de los atributos, el objeto `Node` debe incluir algunos métodos:

`act()`

Permite iniciar y programar la activación de los nodos. El tipo de nodo determinará las funcionalidades del mismo al momento de completar el proceso de activación del nodo.

`create()`

Asocia un nuevo mensaje al nodo.

`fscan()`

Método genérico de recorrido del nodo. El modelista sobre-escribe el método para que incluya las instrucciones particulares que el nodo debe ejecutar en el momento de su recorrido.

`fact()`

Método genérico de activación del nodo. El modelista sobre-escribe el método para que incluya las instrucciones particulares que el nodo debe ejecutar en el momento de su activación.

`it()`

Programa la próxima activación del nodo con respecto al tiempo actual de simulación.

`nt()`

Programa la próxima activación del nodo.

`scan()`

Permite iniciar el recorrido de un nodo. El tipo de nodo determinará las funcionalidades del mismo al momento de completar el proceso de recorrido de la red.

`sendto()`

Permite el tránsito de los mensajes a través de los nodos. Envía el mensaje que se está procesando al nodo indicado.

`stat()`

Permite calcular las estadísticas sobre el nodo.

`stay()`

Programa el tiempo de permanencia de un mensaje en el nodo

Adicionalmente, incluimos métodos que permiten manipular conjuntos de mensajes: `assemble()`, que permite ensamblar mensajes y `deassemble()` que permite restituir mensajes ensamblados.

Los nodos se almacenan en una la lista de nodos, `LNodes`, la cual tiene como cabecera un objeto del tipo `HNode` que cuenta con dos atributos: el primero, `fa`, indica la posición del primer elemento de la lista en ser activado, y el segundo, `ls`, indica la posición del último elemento de la lista en ser recorrido. Estos atributos facilitan el recorrido de la lista al momento de realizar los procesos de activación y recorrido de los nodos.

Esta lista tiene un orden particular. Como mencionamos anteriormente según el tipo es posible identificar cuales se activan, cuales se recorren, y cuales se activan y se recorren. Esta propiedad es utilizada para ordenar la lista de forma que los nodos que se recorren luego de la ocurrencia de un evento ocupan los primeros lugares de la lista, seguidos de aquellos nodos que sólo se activan desde la lista de eventos. Este orden responde al hecho de que los nodos se consultan con mayor frecuencia durante el recorrido de la red que por activación de eventos. Esto asegura que los elementos más consultados se encuentren con mayor celeridad.

La lista de nodos hereda todos los métodos de lista e incorpora los métodos básicos de consulta, definición, inicialización y modificación de los atributos de la cabecera.

Mensajes.

La figura 4.13 muestra los objetos asociados a los mensajes en el simulador. El objeto `Message` representa al mensaje como tal, mientras el objeto `LMess` representa lista de mensajes y el objeto `HMess` la cabecera de dicha lista. Por otro lado, El objeto `Field` forma parte del mensaje y el objeto `LLMess` representa la lista de listas de mensajes.

El objeto `Message` contiene nueve atributos: Los atributos, `name` y `number` permiten la identificación del mensaje, mientras que `where` indica la ubicación y el atributo `use` indica la cantidad de recursos que el mensaje está consumiendo. Generalmente, el mensaje hereda el nombre del nodo donde es creado, por eso es necesario que el identificador contenga dos indicadores.

Los atributos `gt`, `et` y `xt` almacenan valores tipo tiempo que indican el momento en que se creo el mensaje, el momento en que ingreso en la lista actual y el momento en que aspira salir de la lista actual respectivamente. Estos indicadores proporcionan información básica para el cálculo de estadísticas y para el control interno en el momento de activar y recorrer los nodos.

El objeto `Message` posee dos atributos que permiten acceso a listas: El primero, `LA`, apunta a una lista que contiene mensajes ensamblados. En el caso en que un

mensaje este conformado por un grupo de mensajes que se ensamblaron en un nodo anterior debe ser posible restituir los mensajes originales (por eso se almacenan). Por otro lado el segundo atributo, `LF`, apunta a una lista de campos, que permite asociar información a los mensajes.

Además de los métodos básicos de consulta, definición, inicialización y modificación de los atributos, el objeto `Message` debe incluir algunos métodos:

`addAssemble()`

Permite agregar un mensaje o una lista de mensajes a la lista de ensamblados.

`addField()`

Permite agregar un campo a la lista de campos del mensaje.

`copy()`

Crea una copia del mensaje.

El objeto `HMess` contiene doce atributos. Para obtener estadísticas acerca del sistema, se almacena información en los indicadores: `ne()`, indica el número de mensajes que ha salido de la lista, mientras que `suml()`, `sumll()`, `maxl()` y `nl()` contienen la suma, la suma de cuadrados, el valor máximo y el número de registros para las longitudes de la lista y `sumt()`, `sumtt()`, `maxt()` y `nt()` son sus equivalentes para almacenar el tiempo de permanencia de los mensajes en la lista.

También, se almacenan valores de variables de tiempo que indican: `te()`, el momento en que la lista queda vacía, `tm()`, el momento en que se realizó el último cambio y `tf()`, el tiempo que la lista ha permanecido vacía.

Además de los métodos básicos de consulta, definición, inicialización y modificación de los atributos, el objeto `HMess` debe incluir algunos métodos que reflejan la ocupación de las listas:

`statLong()`

Permite obtener estadísticas sobre la longitud de la lista.

`statTime()`

Permite obtener estadísticas sobre el tiempo de permanencia de los mensajes en la lista.

La lista de mensajes, `LMess`, hereda todos los métodos de lista e incorpora además de los métodos básicos de consulta, definición, inicialización y modificación de los atributos de la cabecera los métodos:

`medl()`, `dmedl()`, `mstl()`, `dmstl()`

Permiten obtener la media y la desviación estadística de la longitud de la lista y del tiempo de espera de los mensajes en la lista.

`maxl()`, `entr()`, `tfree()`

Permiten obtener la longitud máxima alcanzada, el número de mensajes procesados, y el tiempo libre de la lista.

`scan()`, `fscan()`

Permiten examinar la lista, a través el método `fscan()` el usuario incluye las instrucciones particulares que controlan la revisión de la lista.

`stat()`

Permite obtener las estadísticas de la lista.

Las listas de listas de mensajes, `LLMess`, se utilizan para representar listas externas de nodos cuyos predecesores son múltiples y necesitan tratar los mensajes en el nuevo nodo haciendo referencia a la lista donde provienen.

El objeto `Field` es el integrante de la lista de campos, `LFields` y contiene dos atributos: `name` y `value` que representan el nombre de una variable y el valor que esta almacena. Esta lista hereda todos los métodos de lista e incorpora además los métodos básicos de consulta, definición, inicialización y modificación de los atributos de los objetos `Field` que la componen.

A continuación, incluiremos la descripción de un grupo de componentes que complementan la especificación del simulador GLIDER.

4.2.3 Otros Componentes

Para incorporar algunas funcionalidades al simulador deben incluirse un grupo de componentes. Estos componentes permiten entre otras incorporar rutinas para generación de números aleatorios, `GRnd` y manipular archivos, `GStr`.

El componente `GRnd` hace uso de las distribuciones estadísticas comúnmente usadas en simulación: Bernoulli, Beta, Erlang, Exponencial, Gamma, Gaussiana, Log-Normal, Normal, Poisson, Triangular, Uniforme, Weibull para generar números aleatorios. Este componente permite al modelista hacer uso de estas distribuciones en el modelado de sistemas.

El componente `GStr` permite manipular los archivos propios de la simulación que contienen la traza y las estadísticas generadas por el simulador, y además facilita la definición y manipulación por parte del usuario de archivos que almacenan información del modelo.

Tomando en cuenta que la estructura de datos diseñada facilita su expansión resulta conveniente agregarle al simulador algunos componentes tales como:

* `GEDO`, que permite incorporar en el modelo la descripción de sistemas haciendo uso de sistemas ecuaciones diferenciales ordinarias,
* `GDB`, que permite incorporar funcionalidades de bases de datos,
* `GIS`, para sistemas de información geográfica.

El componente que controla el comportamiento del simulador, `Glider`, sera presentado en la siguiente sección.

En definitiva, el diseño del simulador debe integrar los componentes mencionados en las últimas secciones. El diagrama de clases de la figura 4.14, generado a partir del diagrama de casos de uso de la figura 4.2 y las especificaciones de los componentes del simulador descritos anteriormente, incluye todos los componentes descritos y presenta en detalle las dependencia entre los objetos que componen el simulador.

A continuación mostraremos los detalles de comportamiento del simulador a partir de la interacción de sus componentes que permitirán asegurar el correcto funcionamiento del simulador desarrollado.

4.2.4 Comportamiento del simulador

La especificación de la dependencia entre los objetos es un paso necesario más no es suficiente para describir el comportamiento del simulador. A continuación especificaremos la interacción entre los componentes.

El diagrama de secuencia de la figura 4.15, muestra la interacción a lo largo del tiempo entre los objetos que contribuyen al proceso de ejecución del simulador haciendo referencia a los procesos que despliega cada objeto. Acá se especifica la secuencia que corresponde a la programación y activación de eventos que sigue el algoritmo descrito en la sección 4.1.2, la cual incluye el proceso recursivo de revisión de la red y la activación de los nodos presentes en el sistema.

Los componentes asociados al proceso de ejecución de la simulación son:

FEL.

Contiene, cronológicamente ordenados, los eventos que deben "tener lugar" en la simulación del sistema. El objeto FEL realiza las siguiente operaciones:

1. incluye en la lista de eventos los eventos que recibe,
2. si se le solicita, entrega el evento que debe ser procesado.

Control.

Representa el controlador del proceso de simulación realizando las siguientes actividades:

1. solicita la programación de los eventos que permiten el inicio del proceso de simulación,
2. solicita el próximo evento y controla que se procese el evento,
3. programa los nuevos eventos.

Network.

Representa el conjunto de subsistemas (nodos) presentes en el sistema a simular. Tiene como función controlar la interacción y activación de los nodos y lleva a cabo las siguientes actividades:

1. controla la activación del nodo actual,
2. controla la revisión de la red.

Además, en el diagrama de colaboración de la figura 4.16 se muestran los intercambios de datos disparados por los eventos que han sido activados en el sistema. En este diagrama, se evidencia que el componente `Control` es quien tiene a su cargo el simulador, y ordena las actividades que se llevan a cabo:

1. Solicita la incorporación de los eventos que inician el proceso de simulación haciendo uso del método `add()`.
2. Solicita el evento a procesar a través del método `extract()`.
3. Recibe un objeto tipo `Event`.
4. Extrae del evento el tiempo de activación y actualiza el reloj del simulador a través del método `advTime()`.
5. Solicita la activación del nodo asociado al evento a través del método `act()`.
6. Desencadena la ejecución del método `fact()` del nodo.
7. Recibe un objeto `List` que contiene eventos que deben programarse a causa de la activación del nodo.
8. Solicita la incorporación de los nuevos eventos haciendo uso del método `add()`.
9. Controla la revisión recursiva de toda la red, `scan()`, con el fin de reflejar los cambios causados por el procesamiento del evento. Esta actividad se realiza hasta asegurar que no es posible reflejar cambios causados por el recorrido de la red.

10. Desencadena la ejecución del método `fscan()` de los nodos de la red
11. Recibe de cada uno de los nodos que se ha revisado un objeto `List` que contiene los eventos que deben programarse a causa de dicho recorrido.
12. Solicita la incorporación de los nuevos eventos haciendo uso del método `add()`.

Cabe destacar que los pasos 10 y 11, que reflejan el proceso de revisión de la red, descrito en la sección 4.1.2, se repiten tantas veces sean necesarias hasta reflejar todos los cambios causados en el sistema. Así mismo el proceso de activación, que se inicia en el paso 2 y abarca el resto de pasos, se repite hasta finalizar la simulación: cuando se alcanza el tiempo determinado para simular, cuando no quedan eventos o cuando el simulista lo indique.

Una vez que completamos la descripción del comportamiento del simulador procedemos a presentar los detalles de implementación del prototipo desarrollado.

4.3 Aspectos de implementación

En la implementación del prototipo del simulador GLIDER se utilizó el lenguaje Java [51] como lenguaje matriz en reemplazo del PASCAL. La selección del lenguaje se realizó fundamentalmente por la posibilidad de hacer de GALATEA una plataforma de simulación independiente de la plataforma física en donde se ejecute. Por otro lado, tomando en cuenta el método escogido para la desarrollo del prototipo pretendemos aprovechar las características de modularidad y flexibilidad que presenta el Java, el cual además, como lenguaje orientado por objetos, nos permitirá emplear los mecanismos de reutilización de componentes.

Para el desarrollo de este prototipo se escogió el método **iterativo e incremental basado en casos de usos**, el cual se funda en el perfeccionamiento gradual del prototipo a través de múltiples ciclos de análisis, diseño y construcción y además estimula la generación temprana de una arquitectura del sistema. Las actividades relacionadas con la creación, presentación y mantenimiento del prototipo son:

Fase de Planificación y Elaboración.

1. Planificación de Requerimientos.
2. Definición de Requerimientos.
3. Planificación del prototipo.

Fase de Construcción.

1. Diseño del prototipo a partir de los requerimientos de caso de uso.
2. Construcción del prototipo siguiendo el ciclo de vida iterativo, basado en incorporación de nuevas funciones en cada ciclo.

Fase de Aplicación.

1. Transición de la implementación del prototipo a su uso.

En esta sección mostraremos los detalles del prototipo que implementamos con Java haciendo uso de los respectivos diagramas UML que nos permiten:

* Modelar el prototipo utilizando los conceptos de la orientación a objetos.
* Utilizar un único lenguaje que entienden máquinas y personas para la descripción del prototipo.
* Establecer un acoplamiento explícito de los conceptos y los artefactos ejecutables.

El diagrama de clases de la figura 4.14 muestra los componentes del prototipo del simulador, mientras que la figura 4.17 muestra como es la dependencia y las relaciones entre las clases.

De cada una de estas clases se implementaron en primera instancia los métodos básicos que permiten la manipulación de los objetos instanciados:

Constructores.

Son métodos que contienen la forma de declarar un objeto perteneciente a la clase y configurar valores iniciales a los atributos de dicho objeto. Estos métodos permiten definir o instanciar objetos particulares según se determinen valores para los atributos.

Métodos de Consulta.

Comúnmente conocidos entre los programadores como métodos `get()`. Estos métodos permiten realizar consultas de los atributos presentes en la clase. Su función principal restringir el acceso al atributo para evitar consultas no deseadas que provoquen fuga de información.

Métodos de Configuración.

Comúnmente conocidos como métodos `set()`. Estos métodos permiten realizar modificaciones o alteraciones de manera controlada en los atributos de la clase. Su función principal es evitar inconsistencias en el prototipo.

Método de visualización. Estos métodos muestran en formato básico de cadena el contenido del objeto. Su función principal es dar a conocer el estado actual del objeto instanciado. Se conocen comúnmente como métodos `toString()`.

Posteriormente, se implementaron los atributos y métodos particulares de cada clase siguiendo los detalles de diseño explicados a lo largo de este capítulo y las pautas descritas en 4.1. Además, tomando en cuenta que queremos obtener una versión compatible con el actual simulador GLIDER, se incorporaron a este prototipo muchas de las instrucciones, procedimientos y funciones que descritas en el manual del GLIDER [52].

El apéndice B contiene una breve descripción de la implementación de las clases (atributos y métodos) que representan cada componente. Además, la descripción detallada de las clases se encuentra en la documentación del prototipo. Conocer estas descripciones le permitirá al usuario simplificar la tarea de escribir el modelo en el código apropiado para el prototipo del simulador.

Cabe destacar que en la implementación de este prototipo, se explotaron cuatro de las características de la programación orientada por objetos:

* Encapsulamiento
* Polimorfismo
* Vinculación dinámica
* Herencia

Estos conceptos proporcionan funcionalidad y control en el diseño del simulador.

Encapsulamiento.

El encapsulamiento es una forma de ocultar y proteger la información. Las clases implementadas, realmente ocultan sus datos de otras clases. Además el encapsulamiento nos permite descomponer el simulador en clases perfectamente diferenciables, donde se agrupan los datos y métodos relacionados, que se implementan de manera independiente, lo cual facilita su posterior mantenimiento.

Polimorfismo.

El polimorfismo se refiere a la naturaleza de los objetos. Los mensajes enviados de un objeto a otro dan como resultado un comportamiento distinto entre los diversos objetos. Así, el comportamiento del objeto es dependiente de la naturaleza misma del objeto. El polimorfismo contribuye a que los objetos sean reutilizables. En nuestro caso lo que parece ser la misma operación aplicada a diferentes objetos mantiene en realidad un comportamiento único en forma transparente para argumentos de diferente tipo.

Vinculación Dinámica.

Relacionada con el polimorfismo está la vinculación dinámica. Esta se refiere al hecho de que los objetos se envían mensajes entre sí, y los aceptan sin saber realmente el tipo específico del objeto con quien están comunicándose. La vinculación dinámica implica que la búsqueda de métodos aplicables se decide en el momento de la ejecución. Esta característica nos permite implementar métodos generales del comportamiento de los componentes del modelo de simulación que posteriormente el modelista sobre-escribe para que dicho componente incorpore las funcionalidades particulares del modelo que representan su comportamiento en el sistema real.

Herencia.

La herencia es un concepto importante que nos permite definir nuevas clases a partir de clases existentes. Estas nuevas clases *heredan* las características y funcionalidades de la clase antigua. Además la herencia nos permite implementar la especialización de objetos.

Estas características, junto con la reutilización de código se explotaron en la implementación del prototipo, la cual como muestra el diagrama de la figura 4.17, presenta el simulador como un paquete o biblioteca de programas que permite la simulación de sistemas. Al empaquetar el simulador favorecemos su visualización como un modulo independiente que puede interactuar perfectamente con los programas del usuario y en vista de que ha sido desarrollado en Java, aseguramos además su portabilidad. La modularidad y la portabilidad presentes en el prototipo serán explotadas posteriormente en el diseño de la plataforma GALATEA mostrado en el capitulo 5.

En resumen, la biblioteca de clases del prototipo del simulador GLIDER proporciona una colección de estructuras de datos suficientes para cubrir las necesidades de los simulistas GLIDER.

La figura 4.18 muestra la estructura del sistema de archivos que contiene la biblioteca de clases de nuestro prototipo. Conocer la manera en que están organizados los archivos, permitirá a los usuarios del prototipo entender la estructura sobre la que se soporta el mismo.

A partir del directorio base, **Root(->)**, se encuentran los archivos que conforman el prototipo del simulador:

* El subdirectorio demos contiene algunos ejemplos, descritos detalladamente en la sección 4.4, que servirán de guía a los usuarios en el uso del prototipo.

* La documentación del prototipo del simulador y de los ejemplos se encuentra en los subdirectorios docs/galatea/glider y docs/demos respectivamente. Esta

documentación se generó con el documentador del lenguaje Java denominado **Javadoc**.

* Los archivos que contienen el código java, comúnmente denominados archivos fuente, se encuentran en el subdirectorio galatea/glider. Este directorio es de acceso restringido a los miembros del grupo de desarrollo de GALATEA.

* Finalmente, los archivos que contienen el conjunto de clases necesarias para la utilización del prototipo, comúnmente denominados ejecutables entre los diseñadores de sistemas, se encuentran en archivo galatea.jar. Este archivo agrupa todas las herramientas necesarias, y su acceso debe ser sin restricciones.

Una vez implementado el prototipo, se realizaron un conjunto de pruebas de verificación y validación que nos permiten asegurar su correctitud.

4.3.1 Pruebas del prototipo

Para cumplir los requisitos del grupo de desarrollo de la plataforma GALATEA, la biblioteca de clases del prototipo debe cumplir con las características de completitud, adaptabilidad, eficiencia, seguridad, simplicidad y extensibilidad. El procedimiento de pruebas que seguimos para verificar que se alcanzan las exigencias antes mencionadas incluyó:

* Pruebas de Métodos. Se realizó una prueba exhaustiva de cada uno de los métodos definidos en las clases construidas. Esta prueba fue orientada a verificar el cumplimiento de las funcionalidades de cada método y a validar el comportamiento de los mismos ante situaciones extremas (datos de tensión).

* Pruebas de Clases. Se realizaron pruebas para comprobar la integración entre los métodos que pertenecen a una misma clase, verificar la consistencia de los métodos, y validar la consistencia de la estructura de datos diseñada. Además se realizaron las pruebas de funcionalidad con datos comunes, siguiendo el esquema detallado en la sección 4.2.

* Pruebas de Integración. Estas pruebas permitieron verificar la integración entre las diferentes clases que conforman el prototipo. Estas pruebas permiten validar que el prototipo sigue el comportamiento descrito en la sección 4.2.4.

* Pruebas de Uso. Estas pruebas permitieron verificar la completitud de la biblioteca desarrollada.

Con la finalidad de mostrar el uso del prototipo de simulador descrito anteriormente, a continuación presentaremos algunos ejemplos básicos de modelado y simulación de sistemas.

4.4 Modeloteca

Probablemente, la mejor manera de introducir la semántica de un nuevo lenguaje es a través de ejemplos. Haciendo uso de este recurso, en esta sección se muestra el proceso de modelado y simulación de tres sistemas básicos:

1. Sistema simple de tres taquillas [48].
2. Sistema de ferrocarril [52].
3. Sistema de supermercado.

El proceso de modelado y simulación se realizó haciendo uso de las herramientas actualmente desarrolladas para la plataforma GALATEA: El modelado de cada sistemas se realizó con el IDE para GALATEA [3], un prototipo de un ambiente de desarrollo de modelos de simulación que hemos diseñado, mientras que la simulación se llevo a cabo con el prototipo de simulador desarrollado como parte de este trabajo.

4.4.1 Sistema simple de tres taquillas

Se desea simular el flujo de personas durante un día laboral en una oficina de atención al publico haciendo uso de un sistema de taquillas, tal y como ocurre en una entidad bancaria. Las consideraciones del proceso son:

- La dependencia presta sus servicios durante 300 unidades de tiempo ininterrumpidas cada día.
- Las personas, clientes, llegan a la dependencia aproximadamente cada 10 unidades de tiempo.
- La dependencia cuenta con tres (3) taquillas, cada una de las cuales es atendida por un cajero y puede prestar servicio a un único cliente a la vez.
- El sistema envía a los clientes a la taquilla correspondiente a la cola más corta con la intención de que sean atendidos lo más rápido posible.
- El tiempo promedio de servicio se estima en 45 unidades para cada cliente.

* Se asume que las taquillas prestan múltiples servicios y el cliente sale de la dependencia una vez que es atendido.

Para modelar el sistema descrito asociamos los clientes que llegan al banco con mensajes y la estructura de la entidad bancaria la asociamos con una red de tres (3) nodos:

Entrada.

Representa la entrada del banco. En este nodo se programan y registran los mensajes (clientes) que llegan al sistema. Como muestra la figura los mensajes son enviados a las taquillas de atención al cliente.

Taquilla.

Nodo múltiple que representa las tres taquillas de atención a cliente. En la taquilla se retiene al mensaje (cliente) mientras es atendido y luego se envía a la salida.

Salida.

Representa la salida del banco. Este nodo registra los mensajes (clientes) que salen del sistema.

Para simular el proceso se sigue el modelo descrito, el cual se muestra en la figura 4.19. Este modelo puede ser escrito en GLIDER (ver código 12). En este código, se especifican los detalles del sistema. En este caso en el código GLIDER se presentan cuatro secciones (ver sección 4.1.1):

Código 12 Código GLIDER: Sistema simple de tres taquillas

```
TITLE
      Sistema simple de tres taquillas
NETWORK
      Entrada (I)  Taquilla::
        IT:=10;
        SENDTO(Taquilla[MIN]);
      Taquilla (R) [1..3] Salida::
        STAY:=45;
      Salida (E) ::
INIT
      TSIM:= 300;
      ACT(Entrada,0);
DECL
      STATISTICS ALLNODES;
END.
```

1. El título del modelo va precedido de la etiqueta `TITLE`.

2. La red que describe el sistema esta precedida de la etiqueta `NETWORK` y consta de tres nodos:

   (a) El nodo `Entrada` es de tipo `I` (Input). Este nodo controla la llegada de mensajes al sistema. Cada vez que se recibe un mensaje se programa la llegada del próximo mensaje en `10` unidades de tiempo usando la instrucción `IT`. Posteriormente, el mensaje se envía a la taquilla que está más desocupada usando la instrucción `SENDTO`.

   En nuestro caso, la indicación de enviar el mensaje (cliente) a la taquilla cuya cola externa sea menor se realiza a través de la etiqueta `MIN`.

   (b) El nodo `Taquilla` es de tipo `R` (Resource). Es un nodo múltiple que representa las tres taquillas de atención al cliente. Al recibir un mensaje, se programa la permanencia en el recurso en `45` unidades de tiempo a través de la instrucción `STAY`. Al culminar el servicio, el mensaje es automáticamente enviado al nodo salida.

   (c) El nodo `Salida` es de tipo `E` (Exit). Este nodo registra la salida de mensajes del sistema.

3. Los valores de las variables que almacenan las condiciones iniciales del estado del sistema van precedidos de la etiqueta `INIT`. En nuestro caso, se asignan `300` unidades al tiempo total de simulación a través de la instrucción `TSIM` y se programa la llegada del primer mensaje en el tiempo `0` a través de la instrucción `ACT`.

   Es posible cambiar los valores para las variables globales del sistema (`TSIM`) en momento de ejecución. Esto permite configurar escenarios alternativos que facilitan el análisis de resultados de la simulación.

4. La declaración de estadísticas en los nodos del sistema va precedida de la etiqueta `DECL` y se realiza a través de la instrucción `STATISTICS`.

El código 13 muestra el equivalente Java del código 12 para el prototipo del simulador desarrollado. La sintaxis de los atributos y métodos correspondientes al simulador se incluye en el apéndice B.

En este código se muestra la especificación de los nodos asociados al modelo (clases `Entrada`, `Taquilla` y `Salida`) y la especificación modelo (clase `Taquilla3`).

En las clases que representan los nodos se incluyen los detalles de identificación (a cada nodo se asocian el nombre, la múltiplicidad y el tipo) y de comportamiento (si es necesario se sobre-escriben los métodos de activación y recorrido).

En la clase que representa el modelo se incluyen los detalles de construcción de la red de nodos y los que corresponden a la ejecución del proceso de simulación:

Código 13 Código Java: Sistema simple de tres taquillas

```
import galatea.glider.*;
class Entrada extends Node{
  Entrada(Node s[]){
    super("Entrada",'I',s);
    Glider.nodesl.add(this); }
  public void sendto(Message m){
    sendto(m,Taquilla3.taquilla,Glider.MIN); }
  public boolean fact(){
    it(10);
    return true; } }
class Taquilla extends Node{
  private static int mult = 1;
  Taquilla(Node s){
    super("Taquilla",mult,'R',s);
    Glider.nodesl.add(this);
    mult++;}
  public boolean fscan(){
    stay(45);
    return true; } }
class Salida extends Node{
  Salida(){
    super("Salida",'E');
    Glider.nodesl.add(this); } }

public class Taquilla3{
  // construccion de la red
  public static Salida salida = new Salida();
  public static Taquilla taquilla[] = { new Taquilla(salida),
                                        new Taquilla(salida),
                                        new Taquilla(salida) };
  public static Entrada entrada = new Entrada(taquilla);
  // Simulador
  public static void  main(String args[]){
            // Inicia las variables globales
    Glider.setTitle("Sistema simple de tres taquillas");
    Glider.setTsim(300);
            // Traza de la simulacion en archivo
    Glider.trace("Taquilla3.trc");
            // programa el primer evento
    Glider.act(entrada,0);
            // Procesamiento de la red
    Glider.process();
            // Estadisticas de los nodos en archivo
    Glider.stat("Taquilla3.sta"); } }
```

- se asignan valores para las variables globales,
- se programa el primer evento,
- se solicita el procesamiento de la red y

En este código Java a diferencia del código GLIDER asociado se le indica al simulador que las estadísticas y la traza de la simulación se registran en archivo.

4.4.2 Sistema de ferrocarril

Se desea simular un día de trabajo en un tramo de un sistema de ferrocarril simple. El ferrocarril tiene múltiples trenes y cada tren tiene varios coches. Las consideraciones del proceso son:

* El número de vagones que se ensamblan en cada tren se desconoce pero se sabe que se encuentra entre `16` y `20`.
* El tramo consta de dos estaciones: una estación de partida y una de destino.
* Los trenes salen de la estación de partida cada `45` minutos.
* El recorrido del tramo a estudiar dura `25` minutos, luego de los cuales el tren llega a la estación destino.
* Los trenes permanecen en la estación destino en espera de un nuevo recorrido.
* El recorrido completo del ferrocarril es circular. Por ende, los trenes se encuentran tiempo completo viajando entre estaciones.

Para modelar el sistema asociamos un minuto a una unidad de tiempo de simulación, representamos los trenes con mensajes y la estructura del tramo ferroviario la representamos a través de una red de tres (3) nodos:

Partida.

Representa la estación de partida de los trenes. En este nodo se programan y registran los mensajes (trenes) que llegan al sistema. Como muestra la figura los trenes son enviados a la ruta.

Ruta.

Representa la ruta a seguir por los trenes. En esta ruta se retiene al mensaje (tren) mientras realiza el recorrido y luego se envía a la estación destino.

Código 14 Código GLIDER: Sistema de ferrocarril

```
TITLE
      Sistema de Trenes
NETWORK
    Partida (I) Ruta ::
        Vagones:= UNIFI(16,20);
        IT:=45;
    Ruta (R) Destino ::
        STAY:=25;
    Destino (E)  ::
        FILE(REGISTRO,TIME:5:2,Vagones);
INIT
    TSIM:=   1440;
    ACT(Partida,0);
DECL
    MESSAGES
        Partida(Vagones:INTEGER);
    STATISTICS ALLNODES;
END.
```

Destino.

Representa la estación destino de los trenes. En este nodo se registran los mensajes (trenes) que salen del sistema.

El modelo del ejemplo se muestra en la figura 4.20. El código GLIDER asociado (ver código 14) es similar al ejemplo anterior. En este caso, se destacan algunas instrucciones:

1. En el nodo `Partida`, de tipo `I`, a cada mensaje (tren) se le asocia un atributo que designa el número de vagones que están presentes durante el recorrido, el cual se obtiene aleatoriamente con una distribución uniforme entre `16` y `20` a través de la instrucción `UNIFI`. Luego, se programa la llegada del siguiente tren en `45` minutos y finalmente, se envía el tren a la ruta.

2. En el nodo `Ruta`, de tipo `R`, se programa la permanencia del tren en la ruta en `25` minutos.

3. En el nodo `Destino`, de tipo `E`, se programa el registro de llegada de trenes a través de la instrucción `FILE`. En este registro se indica el tiempo de llegada del tren y el número de vagones que posee.

4. En la sección de declaraciones se define el tipo de dato asociado al atributo `Vagones` de los mensajes que pasan por el nodo `Partida`. Esta definición se realiza haciendo uso de la instrucción `MESSAGES`.

Este ejemplo muestra el uso de campos para almacenar información detallada del mensaje, la variable `Vagones` del nodo `Entrada` almacena el número de vagones

Código 15 Código Java: Sistema de trenes

```
import galatea.glider.*;
import java.io.*;
class Partida extends Node{
  public static String vagones = "Vagones";
  Partida(Node s){
    super("Partida",'I',s);
    Glider.nodesl.add(this); }
  public boolean fact(){
    Glider.mess.addField(vagones,GRnd.unif(16,20));
    it(45);
    return true; } }
class Ruta extends Node{
  Ruta(Node s){
    super("Ruta",'R',s);
    Glider.nodesl.add(this); }
  public boolean fscan(){
    stay(25);
    return true; } }
class Destino extends Node{
  public static PrintStream registro = GStr.open("REGISTRO");
  Destino(){
    super("Destino",'E');
    Glider.nodesl.add(this); }
  public boolean fscan(){
    GStr.add(registro,
      Glider.getTime() + " " +
      Glider.mess.getValue(Partida.vagones));
    return true; } }
public class Trenes {
        // construye de la red
  public static Destino destino = new Destino();
  public static Ruta ruta = new Ruta(destino);
  public static Partida partida = new Partida(ruta);
       // Simulador
  public static void  main(String args[]) {
           // Inicia las variables globales
    Glider.setTitle("Sistema de Trenes");
    Glider.setTsim(1440);
           // programa el primer evento
    Glider.act(partida,0);
           // Procesamiento de la red
    Glider.process();
           // Estadisticas de los nodos
    Glider.stat();  } }
```

que conforman el tren (mensaje). Adicionalmente, muestra el uso de distribuciones estadísticas para generar números aleatorios y la forma en que se registran en archivos valores asociados a variables del sistema y a campos en mensajes.

El código 15 muestra el equivalente Java del código 14 que es interpretado por el simulador desarrollado. Este código a diferencia del código 13 indica al simulador que las estadísticas sobre los nodos únicamente se muestran al simulista, no se registran en archivo.

En este código se muestra el uso de cosntantes asociadas a un nodo como el nombre del campo (`vagones`) asociado al mensaje creado en el nodo `Partida` y el nombre del archivo (`registro`) asociado al nodo `Destino`.

4.4.3 Sistema de supermercado

En este caso, se desean simular `15` horas de trabajo continuo en un supermercado. Las consideraciones del proceso son:

* Se desea modelar la actividad de los clientes del supermercado y de los vigilantes, sabiendo que el supermercado cuenta con `nVig` empleados de vigilancia.
* Una hora antes del cierre real del supermercado, se prohíbe la entrada de clientes.
* Los clientes llegan al establecimiento según una distribución exponencial con media `t0`. De los cuales un porcentaje (`p`) intentan extraer productos del supermercado.
* El supermercado cuenta con `nCaj` puestos de cobranza.
* Se tienen `nPas` pasillos, que son visitados en forma aleatoria por los clientes quienes permanecen en los pasillos siguiendo una distribución Gaussiana con media `45` minutos y desviación `10` minutos.
* Los clientes permanecen en el establecimiento un tiempo limite `LP` durante el cual estiman que pueden realizar sus compras. Sin embargo salen antes del establecimiento si cubren su cuota estimada de artículos que desean comprar (`EC`) o robar (`ER`).
* Las personas que pretenden robar verifican que no existan vigilantes en el pasillo para proceder a sustraer productos.
* Los vigilantes revisan los pasillos durante `5` minutos, si encuentran algún cliente con productos robados proceden a detenerlos y llevarlos ante el gerente.

Para modelar el sistema descrito (ver figura 4.21) hacemos equivalencia entre una unidad de tiempo de simulación y un minuto. Además, asociamos las personas que llegan al supermercado con mensajes y la estructura del mismo la asociamos con una red de siete (7) nodos (subsistemas):

Vigila.

Representa la entrada de los empleados de vigilancia al supermercado. Los vigilantes son enviados inmediatamente al nodo `Pasillo`

Entrada.

Representa la entrada de los clientes al sistema. Como muestra la figura los mensajes son enviados al nodo `Pasillo`.

Pasillo.

Nodo múltiple que representa los pasillos del supermercado. En este nodo se retiene al mensaje. Durante su permanencia en el pasillo los clientes compran o roban artículos mientras que los vigilantes realizan su ronda. Si un vigilante le encuentra a un cliente artículos robados lo detiene. Al cumplir su permanencia los mensajes se envían al nodo `Decide`.

Decide.

Realiza la selección del destino de la persona:

* Si el cliente ha sido detenido se envia al `Reten`
* Si es un vigilante se determina aleatoriamente a que pasillo se dirige y se envía a dicho pasillo.
* Si el cliente ha cumplido su limite de tiempo, o sus cuotas de compra o robo tiene dos alternativas: es enviado a `Salida` si no reporta artículos comprados o al nodo `Caja` con la cola más corta para registrar el pago por los artículos.
* Si el cliente permanece en el supermercado se determina aleatoriamente el pasillo al cual se envía.

Salida.

Representa la salida del supermercado. Este nodo registra los mensajes (personas) que salen del sistema.

Caja.

Nodo múltiple que representa los puestos de cobranza del supermercado. En este nodo se retienen los clientes mientras realizan el pago de los artículos comprados. La permanencia de los mensajes en este nodo depende del tiempo medio de servicio del cajero y del número

de artículos comprados. Al cumplirse la permanencia los mensajes se envían al nodo `Salida`.

Reten.

Representa la oficina de revisión. Retiene los clientes que han sido detenidos por los vigilantes. La permanencia de los mensajes en este nodo depende del número de artículos robados.

El código 16 muestra el modelo en GLIDER. En este ejemplo, similar a los anteriores, mostramos algunas de las facilidades de GLIDER para modelado y simulación de sistemas. Tomando en cuenta que en este caso modelamos con más detalles el sistema, consideramos necesario destacar algunas particularidades de la implementación:

1. En el nodo `Vigila` no es necesario incluir la creación de nuevos mensajes. Esta creación se hace a través de la activación del nodo de la sección `INIT` en el tiempo `0` de la simulación. En este nodo a cada mensaje creado se le asocia el valor constante `Vig` al campo `Tipo` y se envía al pasillo `1`. El campo `Tipo` permitirá identificar el tipo de mensaje que transita en la red.
2. En `Entrada`, la identificación del mensaje como cliente se hace asociando la constante `Cli` al campo `Tipo` (`p` por ciento de los clientes se re-identifican con `Lad`, y representan a los clientes que intentan robar en el supermercado). La creación de los nuevos mensajes se hace a través de la variable `t0`.

   En este nodo, se inician los contadores del número de artículos comprados/robados (`NAC`/`NAR`), se indica cuantos artículos espera comprar/robar (`EC`/`ER`) y cuanto tiempo esta dispuesto a permanecer en el supermercado (`LP`). Todos estos valores son propios de cada cliente y por tanto se representan como campos del mensaje.

   En la asignación de valores a estos campos se hace referencia a las variables GLIDER `TSIM` (tiempo total de simulación), `GT` (tiempo de creación del mensaje), y `TIME` (tiempo actual de simulación) y a las constantes `LAC` (límite de artículos comprados), `LAR` (límite de artículos robados).

   Los mensajes son enviados aleatoriamente a uno de los pasillos.
3. El nodo `Pasillo` define `10` pasillos, sin embargo, dentro del código asociado al nodo se toman en cuenta `nPas` pasillos. El considerar un número variable de pasillos es motivado al hecho de generar escenarios alternativos del modelo . Al asociar una variable a la instrucción

Código 16 Código GLIDER: Sistema de supermercado

```
TITLE Supermercado
NETWORK
  Vigila (I) Pasillo[1] ::
    Tipo:= Vig;
  Entrada (I) Pasillo ::
    Tipo:= Cli;
    IF (TSIM-TIME>60) THEN IT:= EXPO(t0);
    ER:= 0;                NAR:= 0;
    EC:= UNIFI(0,LAC);     NAC:= 0;
    LP:= GT + MIN(tc*EC,TSIM-GT);
    IF (BER(p/100)) THEN BEGIN Tipo:=Lad; ER:= UNIFI(1,LAR);END;
    k:= UNIFI(1,nPas); SENDTO(Pasillo[k]);
  Pasillo (R) [1..10]  Decide ::
    IF ((Tipo=Lad) AND (NAR<ER)) THEN BEGIN
      libre:= TRUE;
      SCAN(IL_Pasillo[INO]) IF (O_Tipo=Vig) THEN BEGIN
         libre:= FALSE; STOPSCAN; END;
      IF libre THEN NAR:= NAR + UNIFI(1,ER-NAR); END;
    IF (Tipo=Vig) THEN SCAN(IL_Pasillo[INO])
      IF ((O_Tipo<>Vig) AND (O_NAR>0)) THEN O_Tipo:=Det;
    IF (NAC<EC) THEN NAC:= NAC+UNIFI(0,EC-NAC);
    IF (Tipo=Vig) THEN STAY:=GAUSS(5,1) ELSE STAY:= GAUSS(45,10);
  Decide (G) Reten,Caja,Salida,Pasillo ::
    IF (Tipo=Det) THEN SENDTO(Reten)
    ELSE IF (Tipo=Vig) THEN BEGIN
      k:= UNIFI(1,nPas); SENDTO(FIRST,Pasillo[k]); END
    ELSE IF ((TIME>=LP) OR (NAR>ER) OR (NAC>EC)) THEN
      IF (NAC=0) THEN SENDTO(Salida) ELSE SENDTO(Caja[MIN])
    ELSE BEGIN k:= UNIFI(1,nPas); SENDTO(Pasillo[k]); END;
  Caja (R) [1..10] Salida ::
    STAY:= tCaja[INO] * NAC;
  Reten (R) Salida ::
    STAY:= MIN(TSIM-TIME,tDet*NAR);
  Salida (E) ::
INIT
   TSIM:= 15*60;
   nVig:=3;    nPas:= 5;   nCaj:= 3;
   t0:= 5;     tDet:= 5;   tc:=1.5;    p:= UNIF(0.0,30.0);
   INTI nVig:3:cantidad de vigilantes;
   INTI nPas:3:cantidad de pasillos;
   INTI nCaj:3:cantidad de cajeros;
   FOR k:= 1 to nCaj DO tCaja[k]:= UNIF(0.2,1.5);
   FOR k:= 1 to nPas DO Pasillo[k]:= UNIF(7,15);
   FOR k:= 1 to nVig DO ACT(Vigila,0);
   ACT(ENTRADA,0);
DECL
   CONST  Vig = 0; Cli = 1; Lad = 2; Det = 3; LAR = 15;  LAC = 50;
   VAR
      k, nVig, nPas, nCaj: INTEGER;
      t0, tDet, tc, P: REAL;
      libre: BOOLEAN;
      tCaja: ARRAY[1..10] OF REAL;
   MESSAGES
      Entrada(Tipo,EC,NAC,ER,NAR: INTEGER; LP: REAL);
      Vigila(Tipo: INTEGER);
   STATISTICS ALLNODES;
END.
```

`INTI` de la sección `INIT` podemos definir nuevos valores para la variable a la hora de realizar la simulación. De la misma forma, el nodo `Caja` define `10` puestos de cobranzas y en este caso se toman en cuenta `nCaj` puestos.

4. En el nodo `Pasillo` tanto los mensajes tipo `Lad` y `Vig` realizan revisión de los pasillos a través de la instrucción `SCAN()` los mensajes `Lad` consultan la existencia de vigilantes en el pasillo si encuentran alguno posponen su intención de robar artículos. Por otro lado, los vigilantes consultan el número de artículos robados por los clientes del pasillo, si encuentran clientes con artículos robados cambian el tipo del mensaje a `Det` indicando que el cliente ha sido detenido.
5. En la sección `INIT` se asigna como capacidad de cada pasillo un número entero aleatorio entre `7` y `15`, como tiempo promedio que tarda cada cajero en cobrar un producto se asigna un número aleatorio entre `0.2` y `1.5`

El código 17 muestra una porción el código Java equivalente, los nodos que se no se muestran en dicho código se implementan de manera similar a los nodos de los ejemplos anteriores (códigos 13 y 15):

* El nodo `Vigila` se implementa como el nodo `Partida` del ejemplo del ferrocarril. En este caso se define el campo `Tipo` en el mensaje. Como no es necesario definir la creación de nuevos mensajes no se escribe el método `fact()`.
* El nodo `Entrada` se implementa como el nodo `Partida` del ejemplo del ferrocarril. En este caso es necesario definir varios campos en el mensaje: `Tipo`, `EC`, `NAC`, `ER`, `NAR`.
* El nodo `Caja` y el nodo `Reten` se implementan como el nodo `Ruta` del ejemplo del ferrocarril.
* El nodo `Salida` se implementa como el nodo `Salida` del ejemplo del banco.

4.5 Simulación y Resultados

Estos ejemplos elementales, ilustran algunas de las características tradicionales de los modelos de simulación. El modelista describe la estructura del sistema con detalles y el simulador se encarga de generar la traza de la simulación de tal forma que describa el comportamiento del sistema.

Código 17 Código Java: Sistema de supermercado.

```
class Pasillo extends Node{
  private static int mult = 1;
  Pasillo(Node s){
    super("Pasillo",mult,'R',s);
    Glider.nodesl.add(this);
    mult++;}
  public boolean fscan(){
    if ((Glider.mess.getValue(Tipo) == Super.Lad) &&
        (Glider.mess.getValue(NAR) < Glider.mess.getValue(ER))){
      Super.libre = true;
      void il.fscan(){
        if (mess.getValue(Tipo) == Super.Vig){
            Super.libre = false;
            scan = false;}}
      if (Super.libre){
        Glider.mess.setValue(Entrada.NAR,
          Glider.mess.getValue(NAR)
          + Grnd.unif(1,
            Glider.mess.getValue(ER) - Glider.mess.getValue(NAR)));}}
    if (Glider.mess.getValue(Tipo)=Super.Vig){
      void il.fscan(){
        if ((mess.getValue(Entrada.Tipo)!=Super.Vig) &&
            (mess.getValue(Entrada.NAR)>0)){
          mess.setValue(Entrada.Tipo,Super.Det);}}}
    if (Glider.mess.getValue(NAC)<Glider.mess.getValue(EC)){
      Glider.mess.setValue(Entrada.NAC,
          Glider.mess.getValue(NAC)
          + Grnd.unif(0,
            Glider.mess.getValue(EC) - Glider.mess.getValue(NAC)));}
    if (Glider.mess.getValue(Tipo) == Super.Vig){
      stay(Grnd.gauss(5,1));}
    else {
      stay(Grnd.gauss(45,10));}
    return true;}}
```

El simulador desarrollado sigue el algoritmo básico de eventos discretos descrito en la sección 4.1.2. Para explicar el proceso de simulación y los resultados obtenidos con el simulador vamos a hacer referencia al ejemplo del sistema simple de tres taquillas descrito anteriormente.

4.5.1 Estadísticas sobre los nodos

El simulador a solicitud del modelista, quien incluye la instrucción `STATISTICS` en la sección `DECL` del modelo, presenta estadísticas sobre los nodos del sistema. En particular en el ejemplo de la figura 4.19, correspondiente al código 12, se indica que se analicen todos los nodos (`ALLNODES`), por tanto el simulador muestra la siguiente información:

```
Salida[1]
  EL=18:0:1:0,50:0,50:5,0:0,27778:1,14531:45,0
Taquilla[3]
  EL=10:3:3:1,23529:0,94118:85,0:40,71429:31,33102:50,0
  IL=7:1:1:0,53846:0,49852:45,0:45,0:0,0:20,0
Taquilla[2]
  EL=10:3:3:1,47059:0,97725:90,0:47,85714:31,03652:30,0
  IL=7:1:1:0,53846:0,49852:45,0:45,0:0,0:10,0
Taquilla[1]
  EL=11:4:4:1,61111:1,11250:90,0:49,28571:31,78371:30,0
  IL=7:1:1:0,53846:0,49852:45,0:45,0:0,0:0,0
Entrada[1]  31
```

Las estadísticas en cada nodo incluyen:

* Nombre e índice del nodo.
* Estadísticas de las listas del nodo:
  - Número de mensajes que entraron a la lista.
  - Longitud final de la lista.
  - Longitud máxima alcanzada por la lista.
  - Promedio de la longitud de la lista.
  - Desviación del promedio de la longitud de la lista.
  - Tiempo máximo de permanencia en la lista.
  - Promedio del tiempo de espera en la lista.
  - Desviación del promedio del tiempo de espera en la lista.
  - Tiempo máximo que permaneció desocupada la lista.

Estos datos nos indican que llegaron 31 clientes al banco, de los cuales 11 clientes van a la cola 1, 10 clientes a la cola 2 y 10 clientes a la cola 3. Al final de la simulación se registra que:

* 18 clientes fueron atendidos: seis en cada taquilla
* están siendo atendidos 3 clientes: uno en cada taquilla.
* quedan sin atender 4 clientes en la cola 1, 3 clientes en la cola 2 y 3 clientes en la cola 3.

Además, se obtienen datos de desocupación, de la longitud de las colas y del tiempo de espera de los mensajes en las mismas.

4.5.2 Traza de la simulación

A solicitud del simulista es posible obtener la traza de la simulación. Para el ejemplo de tres taquillas de la figura 4.19 el simulador proporciona la traza mostrada en la figura 4.22. La traza nos da datos del proceso de simulación, mostrando la lista de nodos y los detalles que ocurren al procesar cada evento:

1. Creación de mensajes.
2. Trayectoria de los mensajes.
3. Activación de los nodos.
4. Recorrido de los nodos.
5. Estado de la lista de eventos.

El formato de salida del simulador es el siguiente:

Lista de Nodos.
* Nombre de la lista.
* Referencias de la lista.
  - Número de nodos.
  - Posición del nodo actual.
  - Posición del primer nodo que se activa.
  - Posición del último elemento que se recorre.
* Nodos presentes: tipo, nombre e índice.

Lista de Eventos.
* Nombre de la lista.
* Referencias de la lista.
  - Número de eventos.
  - Posición del elemento actual.
* Eventos presentes
  - Nodo asociado: tipo, nombre e índice.
  - Tiempo de activación

Proceso. La etiqueta `Act` indica la activación de un nodo y la etiqueta `Scan` indica la revisión o recorrido. Estas etiquetas vienen seguidas de la información asociada a él(los) nodo(s) asociado(s).

- Tipo, nombre e índice del nodo.
- Etiqueta `Node`
- Etiqueta `***->` seguida del método a ejecutar
  Creación: el método `create()` y el número del mensaje.
  Activación: el método `act()` tiempo de activación del nodo.
  Envío: el método `sendto()`

En el caso particular del ejemplo de tres taquillas, la traza comienza con la lista de nodos

$$LNodes = Salida \rightarrow Taquilla[3] \rightarrow Taquilla[2] \rightarrow Taquilla[2] \rightarrow Entrada$$

y luego se muestran los detalles del proceso de simulación mostrando la activación de cada evento:

---

1. En el tiempo 0 se programa la activación del nodo `Entrada` y se actualiza la lista de eventos
   $$FEL = (Entrada, 0)$$
   (a) Se activa el nodo *Entrada*
   - Se crea un mensaje.
   - Se programa la activación del nodo *Entrada* en el tiempo 10
   - Se programa el envío del mensaje actual al próximo nodo.

   (b) Se revisan los nodos *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$.
   (c) Se recorre el nodo $Taquilla[1]$.
   - Se programa la activación del nodo $Taquilla[1]$ en el tiempo 0.
   - Se programa la salida del mensaje que ocupa el nodo $Taquilla[1]$ en el tiempo 45.

   (d) Se revisan los nodos *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$
2. Se actualiza la lista de eventos
   $$FEL = (Taquilla[1], 0) \rightarrow (Entrada, 10) \rightarrow (Taquilla[1], 45)$$
   (a) Se activa el nodo $Taquilla[1]$.
   (b) Se revisa el nodo $Taquilla[1]$.
3. Se actualiza la lista de eventos
   $$FEL = (Entrada, 10) \rightarrow (Taquilla[1], 45)$$
   (a) Se activa el nodo *Entrada*.
   - Se crea el mensaje 2.
   - Se programa la activación del nodo *Entrada* en el tiempo 20
   - Se programa el envío del mensaje 2 al próximo nodo.

(b) Se revisan los nodos *Salida*, $Taquilla[3]$, $Taquilla[2]$.

(c) Se recorre el nodo $Taquilla[2]$.

* Se programa la salida del mensaje que ocupa el nodo $Taquilla[2]$ en el tiempo 55.

(d) Se revisan los nodos $Taquilla[1]$, *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$

4. Se actualiza la lista de eventos
$$FEL = (Entrada, 20) \rightarrow (Taquilla[1], 45) \rightarrow (Taquilla[2], 55)$$

⋮

7. Se actualiza la lista de eventos
$$FEL = (Taquilla[1], 45) \rightarrow (Entrada, 50) \rightarrow (Taquilla[2], 55) \rightarrow (Taquilla[3], 65)$$

(a) Se activa el nodo $Taquilla[1]$.

* Se programa el envío del mensaje actual al próximo nodo.

(b) Se recorre el nodo $Taquilla[1]$.

* Se programa la salida del mensaje que ocupa el nodo $Taquilla[2]$ en el tiempo 90.

(c) Se revisan los nodos *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$, *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$.

⋮

50. Se actualiza la lista de eventos
$$FEL = (Entrada, 300) \rightarrow (Taquilla[1], 315) \rightarrow (Taquilla[2], 325) \rightarrow (Taquilla[3], 340)$$

(a) Se activa el nodo *Entrada*

* Se crea el mensaje 31.
* Se programa la activación del nodo *Entrada* en el tiempo 310
* Se programa el envío del mensaje 31 al próximo nodo.

(b) Se revisan los nodos *Salida*, $Taquilla[3]$, $Taquilla[2]$, $Taquilla[1]$.

Existen varios aspectos que consideramos necesario resaltar respecto a la secuencia de activación de nodos:

- En el evento 2 la activación del nodo $Taquilla[1]$ no indica la ejecución de ningún método. Esto ocurre debido a que en el modelo (ver código 12) para este nodo la única función de activación descrita corresponde a la programación de la salida del mensaje actual y en este caso la taquilla esta vacía.

- El caso en que la revisión de un nodo no indique la ejecución de algún método indica que no ocurrieron cambios en la lista de mensajes y por eso no se revisan.

- En los eventos 3, 4 y 7 se muestra que algunos nodos se revisan dos veces. Esto se hace para asegurarse que no se presenten cambios en las lista de mensajes.

- Dado que en el evento 50 el reloj de la simulación alcanzo las `300` unidades de tiempo deseadas (ver código 12) y el próximo evento ha sido programado para el tiempo `310` se detiene el proceso.

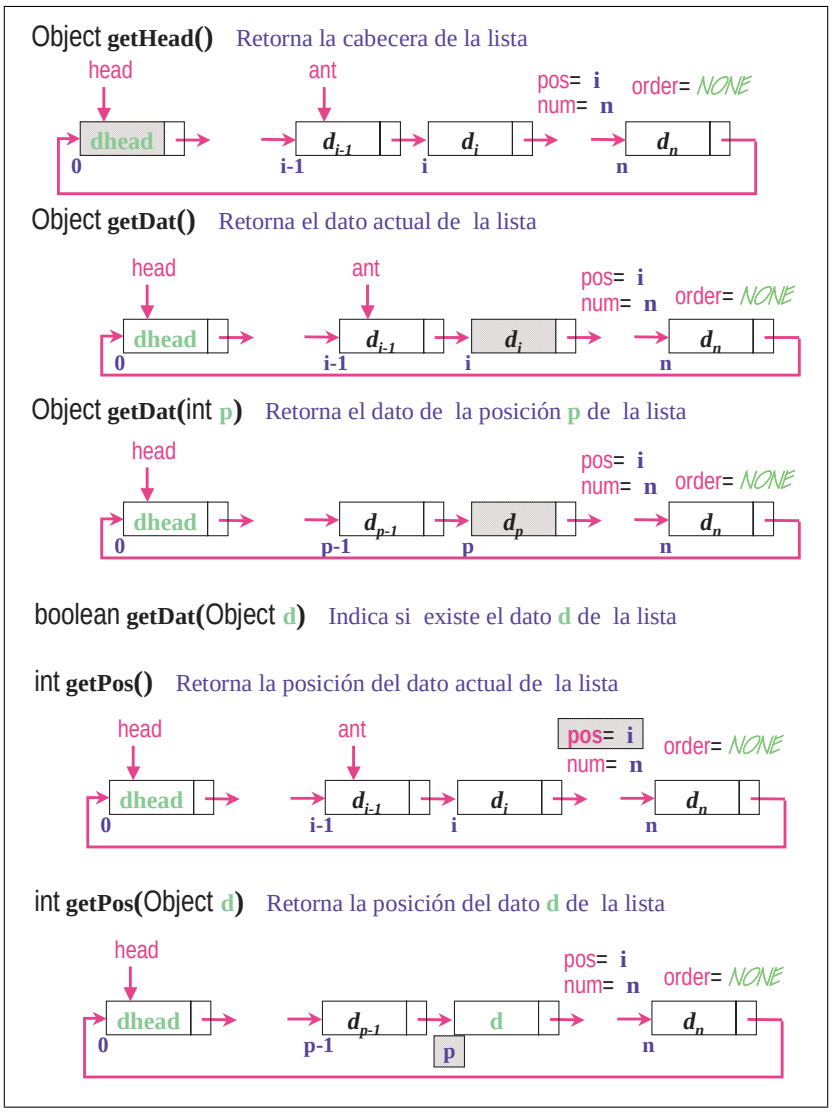

Figura 4.6: Métodos para consultar en lista

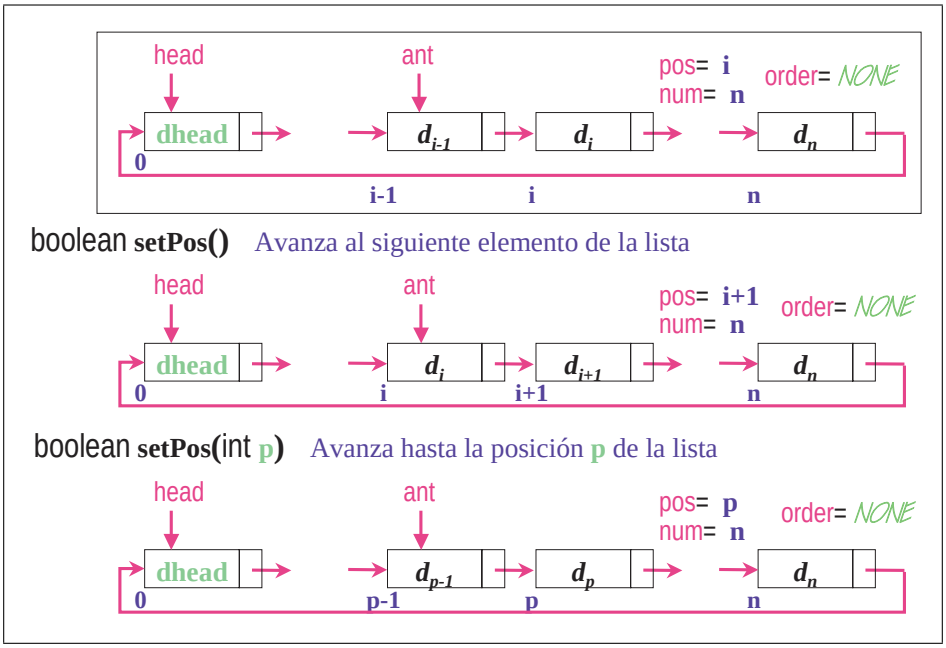

Figura 4.7: Métodos para avanzar en lista

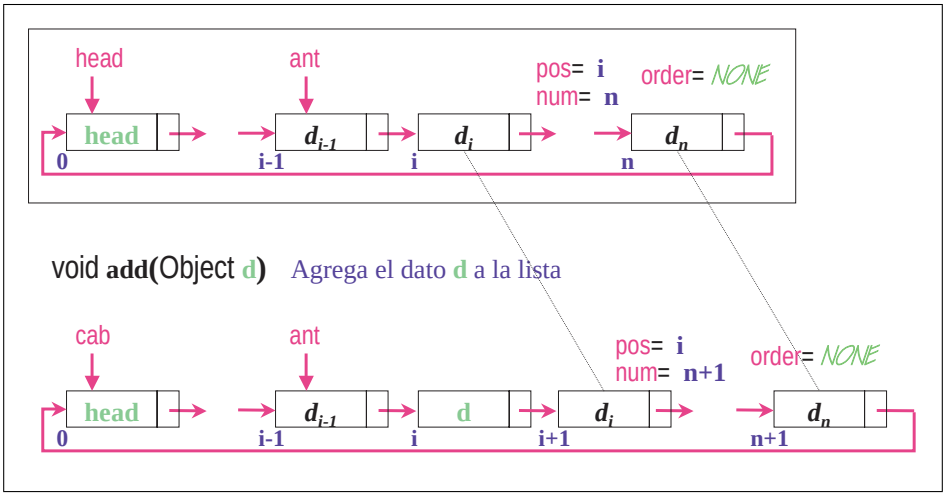

Figura 4.8: Métodos para agregar en lista

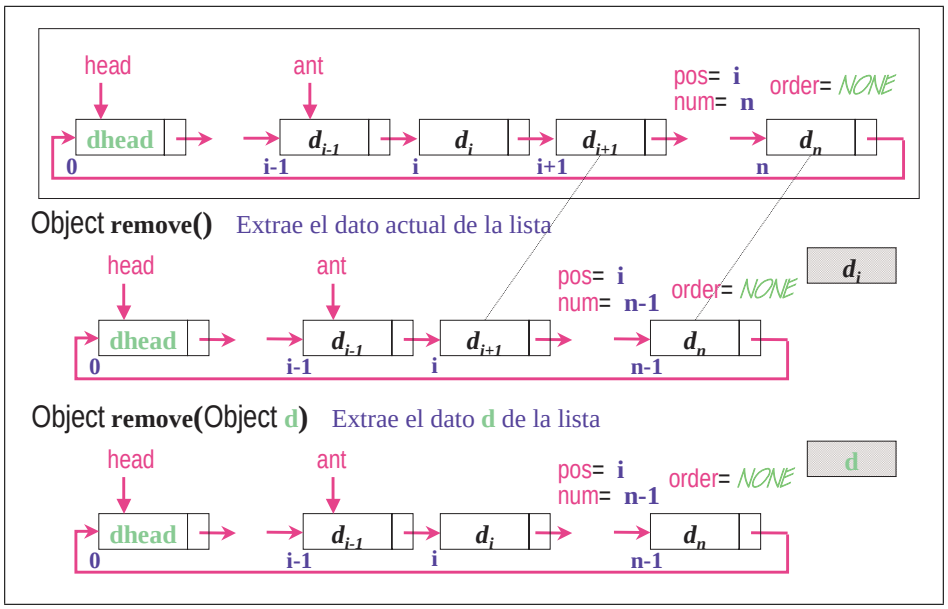

Figura 4.9: Métodos para eliminar en lista

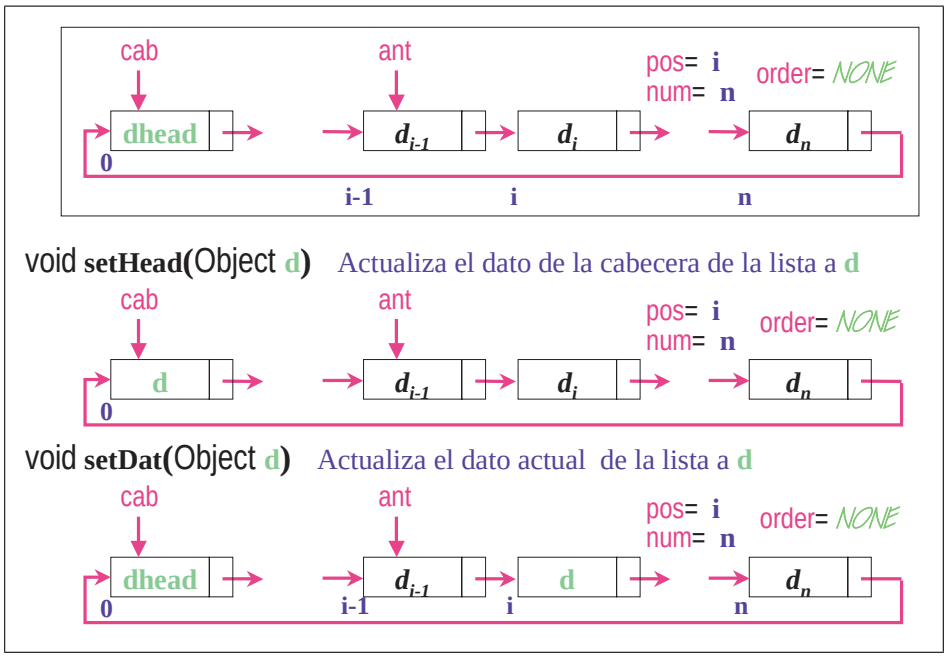

Figura 4.10: Métodos para modificar en lista

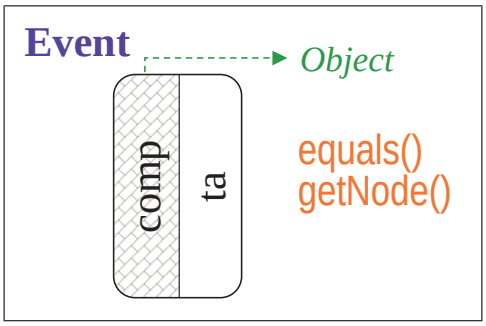

Figura 4.11: Objetos asociados al componente evento

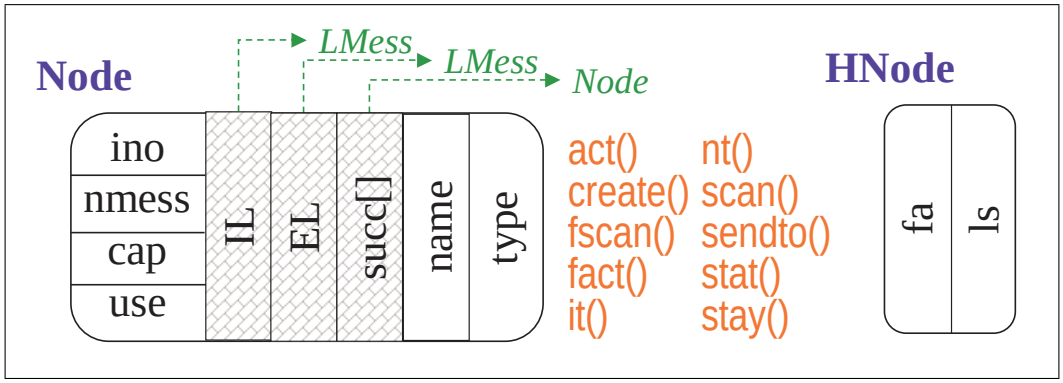

Figura 4.12: Objetos asociados al componente nodo

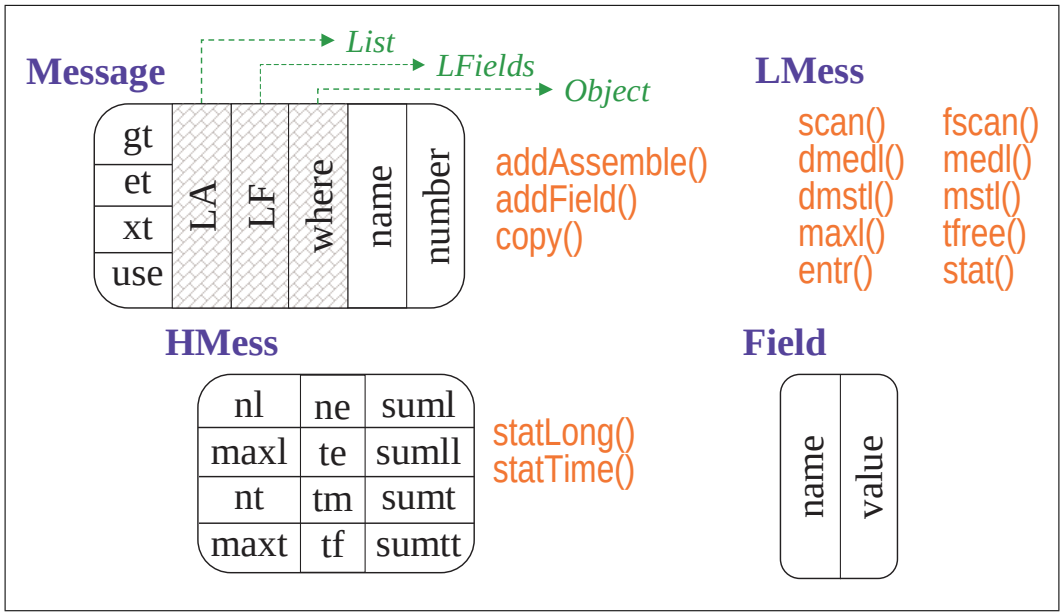

Figura 4.13: Objetos asociados al componente mensaje

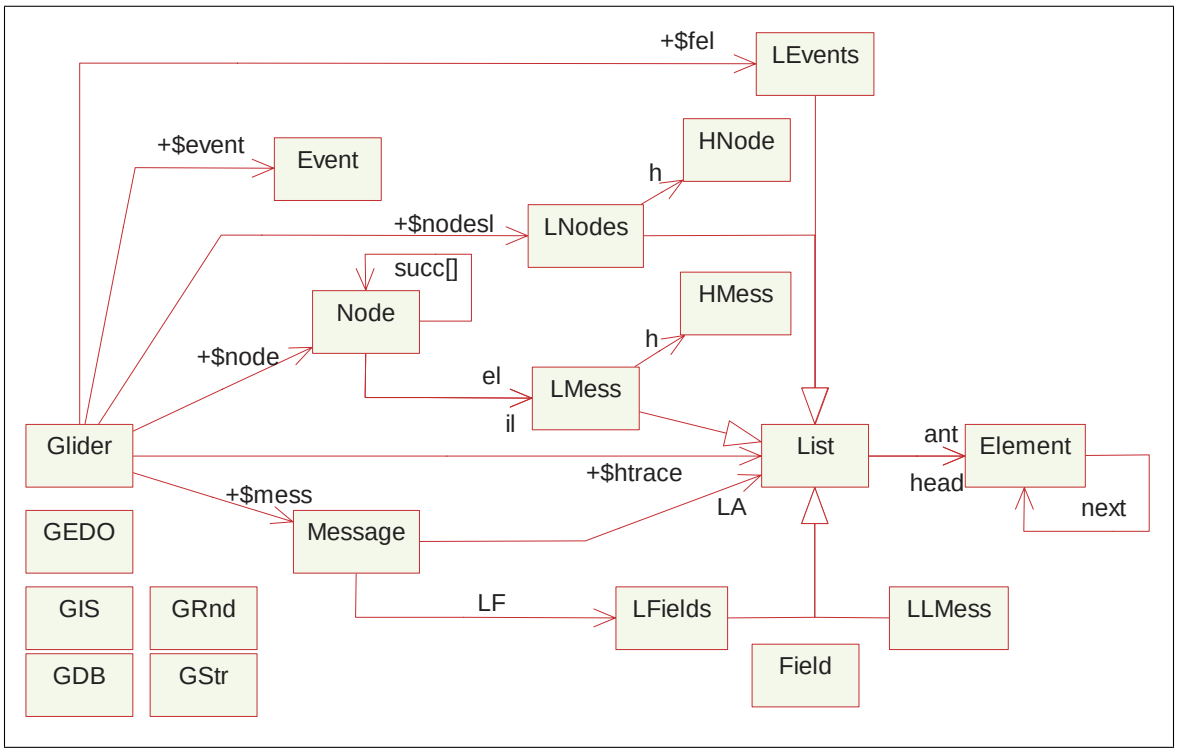

Figura 4.14: Diagrama de clases del simulador

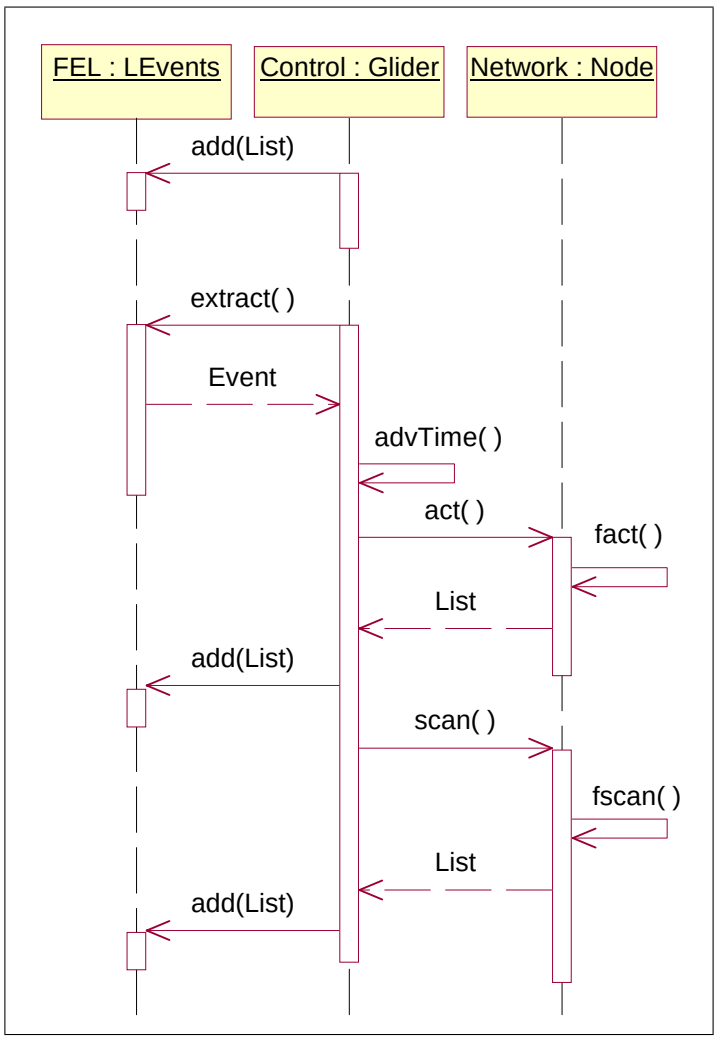

Figura 4.15: Diagrama de secuencia para el simulador

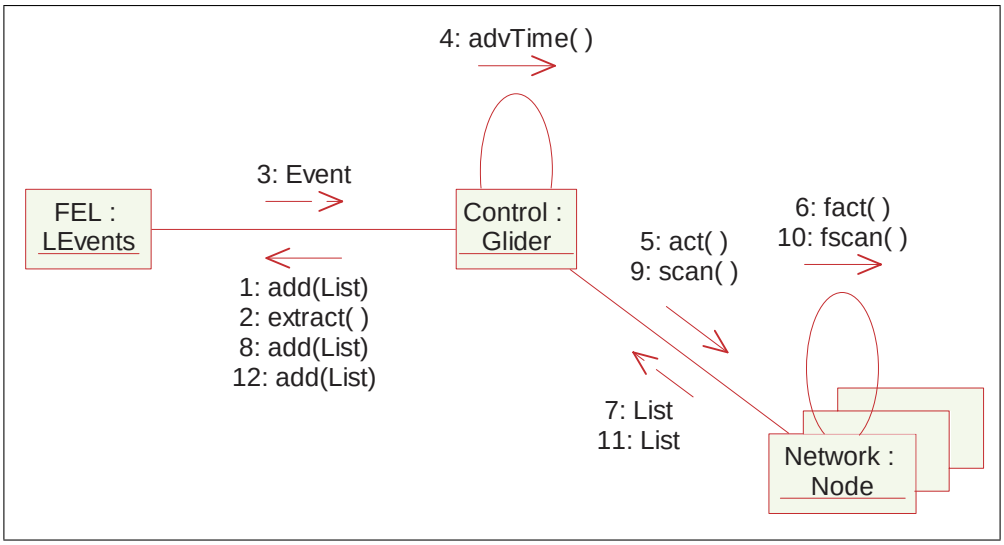

Figura 4.16: Diagrama de colaboración para el simulador

Figura 4.17: Jerarquía de clases del simulador

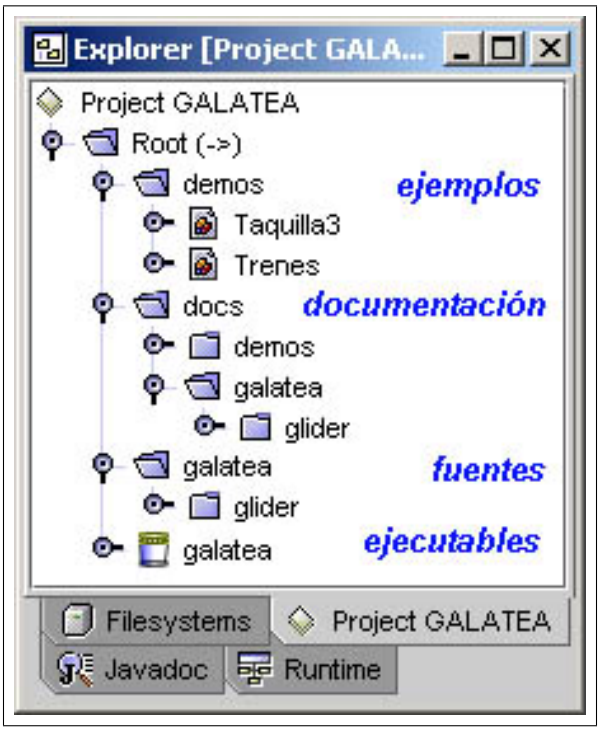

Figura 4.18: Estructura de archivos del simulador

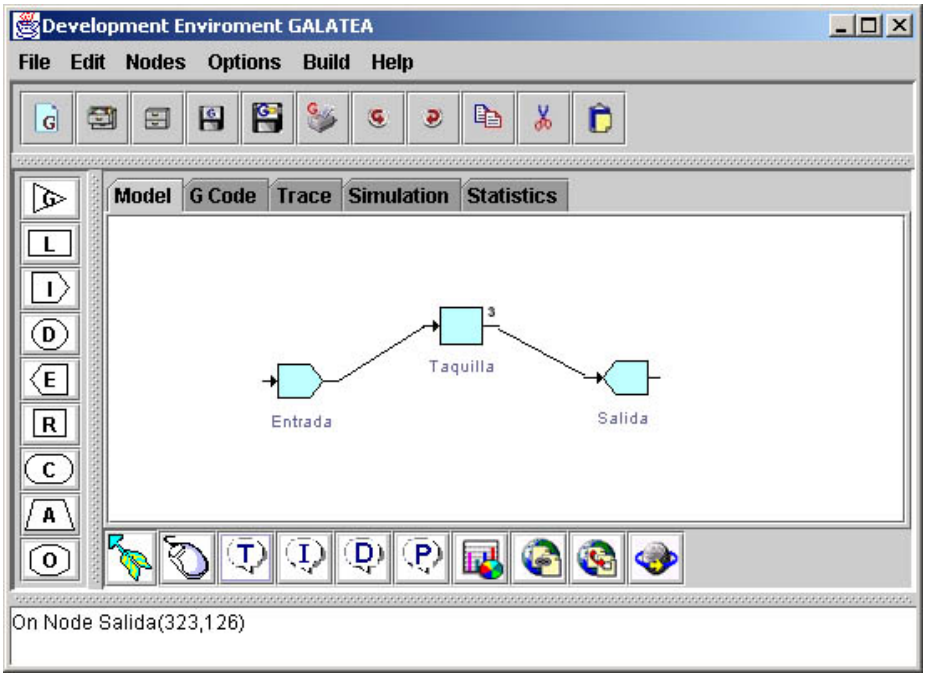

Figura 4.19: Modelo del sistema simple de tres taquillas

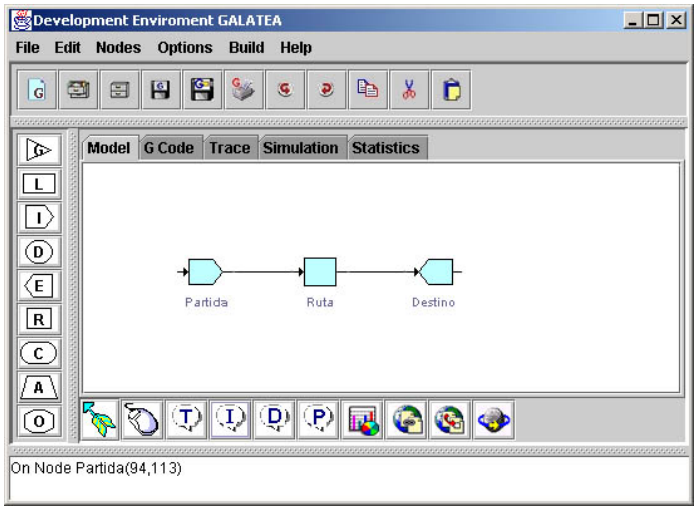

Figura 4.20: Modelo del sistema de ferrocarril

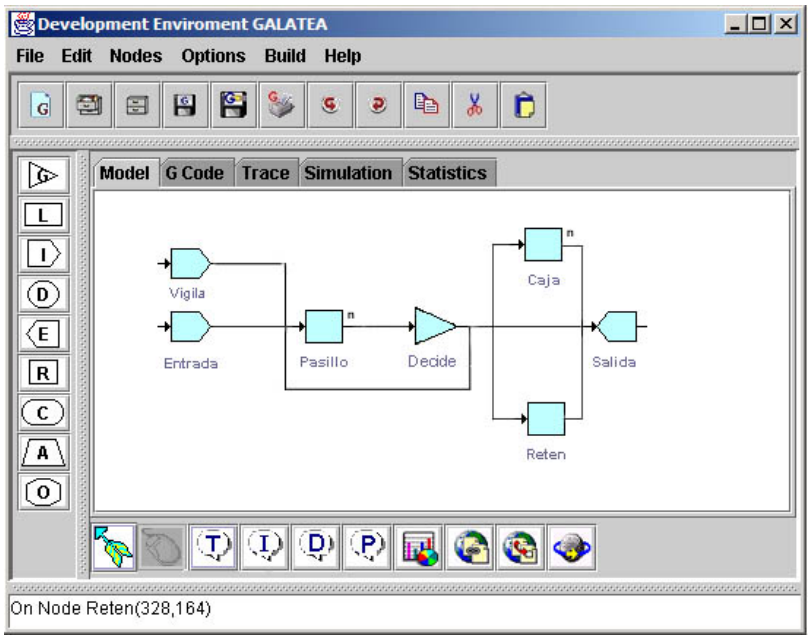

Figura 4.21: Modelo del sistema de Supermercado

```
 LNodes=[5;1;2;4] -> (E)Salida[1] -> (R)Taquilla[3] -> (R)Taquilla[2] ->
                     (R)Taquilla[1] -> (I)Entrada[1]
    (I)Entrada[1]Node ***->act()    0.0
>>>>>  1  ********************************
 FEL=[1;1;/] -> (I)Entrada[1],0.0
 Act (I)Entrada[1]
     (I)Entrada[1]Node ***->create() 1
     (I)Entrada[1]Node ***->act()    10.0
     (I)Entrada[1]Node ***->sendto()
 Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2]
 Scan (R)Taquilla[1]
      (R)Taquilla[1]Node ***->act()   0.0
      (R)Taquilla[1]Node ***->act()   45.0
 Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2] Scan (R)Taquilla[1]
>>>>>  2  ********************************
 FEL=[3;3;/] -> (R)Taquilla[1],0.0 -> (I)Entrada[1],10.0 -> (R)Taquilla[1],45.0
 Act (R)Taquilla[1]
 Scan (R)Taquilla[1]
>>>>>  3  ********************************
 FEL=[2;1;/] -> (I)Entrada[1],10.0 -> (R)Taquilla[1],45.0
 Act (I)Entrada[1]
     (I)Entrada[1]Node ***->create() 2
     (I)Entrada[1]Node ***->act()    20.0
     (I)Entrada[1]Node ***->sendto()
 Scan (E)Salida[1] Scan (R)Taquilla[3]
 Scan (R)Taquilla[2]
      (R)Taquilla[2]Node ***->act()   55.0
 Scan (R)Taquilla[1] Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2]
 Scan (R)Taquilla[1]
>>>>>  4  ********************************
 FEL=[3;3;/] -> (I)Entrada[1],20.0 -> (R)Taquilla[1],45.0 -> (R)Taquilla[2],55.0
 Act (I)Entrada[1]
     (I)Entrada[1]Node ***->create() 3
     (I)Entrada[1]Node ***->act()    30.0
     (I)Entrada[1]Node ***->sendto()
 Scan (E)Salida[1]
 Scan (R)Taquilla[3]
      (R)Taquilla[3]Node ***->act()   65.0
 Scan (R)Taquilla[2] Scan (R)Taquilla[1] Scan (E)Salida[1] Scan (R)Taquilla[3]
 Scan (R)Taquilla[2] Scan (R)Taquilla[1]
   .
   .
   .
>>>>>  7  ********************************
 FEL=[4;2;/] -> (R)Taquilla[1],45.0 -> (I)Entrada[1],50.0 ->
                (R)Taquilla[2],55.0 -> (R)Taquilla[3],65.0
 Act (R)Taquilla[1]
     (R)Taquilla[1]Node ***->sendto()
 Scan (R)Taquilla[1]
      (R)Taquilla[1]Node ***->act()   90.0
 Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2] Scan (R)Taquilla[1]
 Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2] Scan (R)Taquilla[1]
   .
   .
   .
>>>>>  50  *******************************
FEL=[4;4;/] -> (I)Entrada[1],300.0 -> (R)Taquilla[1],315.0 ->
               (R)Taquilla[2],325.0 -> (R)Taquilla[3],340.0
Act (I)Entrada[1]
    (I)Entrada[1]Node ***->create() 31
    (I)Entrada[1]Node ***->act()    310.0
    (I)Entrada[1]Node ***->sendto()
Scan (E)Salida[1] Scan (R)Taquilla[3] Scan (R)Taquilla[2] Scan
(R)Taquilla[1]
```

Figura 4.22: Traza para el sistema simple de tres taquillas

Capítulo 5

Simulación de sistemas multi-agentes con ejemplos

Este capítulo describe los detalles de la plataforma de simulación GALATEA y presenta además recomendaciones técnicas específicas para su implementación.

Introducción

La plataforma de la simulación GALATEA integra los conceptos y herramientas que permiten simular sistemas bajo los enfoques distribuido, interactivo, continuo, discreto y combinado. Además, en esta plataforma, incorporamos soporte para el modelado y la simulación eficiente de sistemas multi-agentes.

Nuestra intención es hacer de la plataforma una herramienta que facilite las tareas de modelado y simulación de sistemas, pensando en esto, proponemos que GALATEA este conformada por:

* una familia de lenguajes, y sus respectivos compiladores, que permiten simular sistemas multi-agentes,
* un simulador
* un ambiente para la construcción de modelos.

Esta plataforma está desarrollándose en Java [51] y podrá ser ejecutada desde aquellos ambientes que soportan aplicaciones Java. Hasta ahora las implementaciones asociadas a esta plataforma han sido parcialmente evaluadas en Windows y Linux:

* En [20] se muestra una especificación detallada de los lenguajes.
* En el presente documento (capítulo 4), se muestran los detalles de diseño e implementación de un prototipo funcional del simulador.
* En [3] se muestran los detalles de diseño e implementación del prototipo funcional para la interfaz gráfica del ambiente de desarrollo.

GALATEA propone una reformulación de la manera tradicional de manejar la relación entre los componentes de simulación con miras a la simulación interactiva, por ello es necesario pensar en varios programas ejecutándose en computadoras heterogéneas y distribuidas que interactúan a través de un sistema operativo distribuido. Nosotros aprovechamos estas características para permitir el modelado y la simulación de sistemas multi-agentes.

Como mencionamos anteriormente, los adelantos en Simulación Interactiva han permitido el desarrollo de un marco de referencia estándar que facilita la integración de múltiples componentes de simulación. Este marco de referencia, denominado HLA (ver sección 3.1.4), nos sirve como referencia para el diseño de la plataforma.

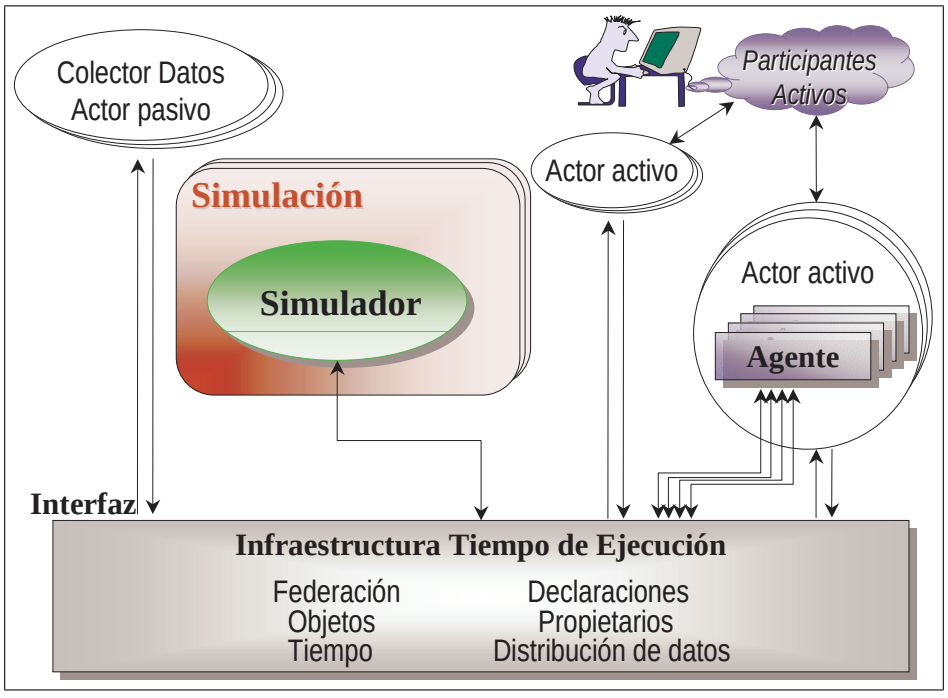

Figura 5.1: Arquitectura de la plataforma de simulación GALATEA

La figura 5.1 muestra la estructura de la federación HLA, usada en GALATEA,

dividido en sus componentes funcionales principales. El primer componente, la **Simulación**, reúne todas las simulaciones (como se define en HLA) el cual debe incorporar capacidades específicas que permiten a los objetos de una simulación, *federados*, conectarse con los objetos de otra simulación. El intercambio de datos entre federados es soportado por los servicios implementados en la interfaz. Nosotros particularizamos la simulación HLA para que cada federado mantenga su propio subconjunto del modelo del mundo e incluimos:

1. un federado que representa el simulador principal que controla todos los eventos. El uso de un único simulador puede rendir beneficios sustanciales, incluyendo:
   * facilita la validación del comportamiento de los sistemas existentes y la reutilización de los modelos existentes,
   * proporciona una infraestructura para desarrollar nuevos sistemas,
   * facilita el estudio de sistemas de gran escala de interacción en un ambiente controlado, y
   * facilita la comparación de resultados.
2. un grupo de federados que representa los agentes del sistema. Estos agentes poseen algún grado de comportamiento autónomo y comunicación con su ambiente y con los otros agentes para alcanzar sus metas. El comportamiento de los agentes causa cambios en el ambiente y en consecuencia el ambiente reacciona.

El segundo componente funcional de la arquitectura es la **Infraestructura de Tiempo de Ejecución** (RTI: *Runtime Infrastructure*). Esta infraestructura proporciona un grupo de servicios de propósito general que soportan las interacciones entre los componentes de simulación. Todas las interacciones entre los componentes de la simulación se llevan a cabo a través del RTI.

El tercer componente es la **Interfaz**. Esta interfaz, independiente de su aplicación, mantiene una manera estándar de relacionar los componentes de simulación con el RTI. La interfaz es independiente de los requisitos para el modelado y del intercambio de datos entre los componentes de la simulación. En particular, en nuestro caso, la interfaz además de controlar la sincronización del tiempo global, debe proporcionar el conjunto de servicios necesarios para que el intercambio de datos entre los agentes y la simulación a la que están asociados fluya.

A continuación, introduciremos cada uno de los componentes de la arquitectura que proponemos en la figura 5.1: la interfaz, el simulador y los agentes. Esta descripción incluye los detalles que deben tomarse en cuenta en la implementación.

5.1 Interfaz

En general en GALATEA cada una de las entidades de nuestra federación HLA se asocia a un objeto. Estos objetos interactúan a través de pase de mensajes los cuales son vistos como eventos puntuales. Como mencionamos anteriormente tanto los modelos de simulación como los agentes corresponden a federados en nuestro modelo HLA y por ende siguen el siguiente ciclo de vida:

1. Se incorpora a la federación.
2. Establece sus requerimientos de datos.
3. Revisa y encuentra las instancias de objetos que necesita.
4. Actualiza los valores de sus atributos.
5. Avanza el reloj.
6. Elimina objetos.
7. Se separa de la federación.

Los federados necesitan sincronizar sus operaciones a medida que avanza el tiempo, por tanto en cada actualización cada federado repite los pasos 2-6 y cada actualización es potencialmente síncrona con respecto a un tiempo virtual compartido.

Con la finalidad de reducir el tráfico de mensajes y limitar la cantidad de interrupciones en cada federado HLA propone mecanismos que permiten reducir el número de mensajes:

Publicación. Al iniciarse la ejecución de una federación cada simulación registra sus objetos y atributos en la RTI.

Inscripción. Además debe registrar que tipo de atributos externos necesita para llevar a cabo su tarea. También es posible definir un rango para los valores que puede asumir dicho atributo.

El objetivo de estos mecanismos es extraer tanta información como sea posible de las simulaciones. Tanto la inscripción como la publicación son mecanismos dinámicos y pueden ser alterados durante la ejecución de la sesión. Cabe destacar que el RTI hace distinción entre direccionamiento de datos (establece la conexión necesaria) y el envío propiamente dicho de los datos. La meta al definir el RTI es establecer la red de conexiones necesarias para minimizar el retardo y maximizar el desempeño. Este factor es de vital importancia en el caso en que la RTI es utilizada por federados que se encuentran distribuidos en una red de computadores.

En vista de que nuestros federados intercambian atributos, necesitamos definir un mecanismo para compartirlos. Por lo pronto podemos obligar a que cada atributo sea

controlado por el federado que instancia el objeto que lo contiene. No obstante, el federado que es dueño del atributo es el único que puede alterarlo durante la ejecución.

Con respecto al manejo del tiempo, de momento sugerimos que la sincronización del tiempo global sea realizada a través de la supervisión de actividades y la concurrencia se resuelva en el modelado. Los puntos de control de simulación se consideran eventos. Éstos puntos de control son dispositivos especiales que permiten controlar tanto las percepciones y las acciones de los agentes como los progresos de la simulación. Además, como en simulación DEVS clásica, se actualizan las variables que representan el estado del ambiente después de cada avance de tiempo.

Dado que en [9] se muestra una implementación natural para DEVS haciendo uso de un marco de referencia orientado por objetos, la cual se utiliza en [53] para crear simulaciones HLA, explicando como es el ambiente DEVS/HLA, como se realiza el mapeo DEVS/HLA, y además se presentan algunos detalles de la implementación en C^{++} del protocolo de simulación DEVS en HLA, proponemos que se sigan estos trabajos al momento de implementar la interfaz en lenguaje Java para nuestra plataforma.

5.2 Simulador

Como mencionamos anteriormente el simulador de la plataforma GALATEA es una extensión del simulador GLIDER. Por tanto, la presente descripción estará basada las modificaciones necesarias para que el simulador presentado en el capítulo 4 pueda integrarse a la plataforma.

5.2.1 Algoritmo general para el simulador

La simulación de un sistema consiste en generar y activar una sucesión de eventos en el ambiente simulado. Por consiguiente, el simulador, esencialmente, activa dichos eventos y ejecuta las piezas de código asociadas con cada uno. Se puede argumentar que el disparo o activación de un evento consiste, precisamente, en la ejecución de esa pieza de código. Sin embargo, hay casos que no encajan con esta definición. Así, en GLIDER un evento es la activación de un nodo que puede o no implicar la ejecución de su código.

Una simulación de un modelo GLIDER es la activación de los eventos relacionados a los nodos y la posterior revisión de la red, buscando el código a ser ejecutado y probando sus pre-condiciones estructurales. Si estas pre-condiciones se alcanzan, el código debe ejecutarse. Para garantizar que los efectos (cambios) inducidos en cada evento se propaguen a través de todo el sistema, la activación de un nodo (la ocurrencia de un evento identificado con el nombre del nodo) puede activar nuevos

eventos y además puede implicar cambios en el estado global del sistema o pases de mensajes.

Así, una revisión de la red conduce a la evolución de la simulación del sistema. Cuando un nodo se activa, su código se ejecuta y se inicia la revisión del resto de la red. Durante la revisión, aquellos nodos que no tienen una lista externa (`EL`: *External List*), lista de espera para los mensajes, o cuya `EL` está vacía se ignoran. En estos nodos no se reflejan cambios debido a que no contienen ninguna entidad.

El proceso de revisión continua visitando nodos (reflejando los cambios ocurridos en aquéllos nodos que contienen entidades) hasta que se alcanza el nodo donde se inicio la revisión (el nodo activado) luego de completar un ciclo completo sin movimientos de mensajes.

Una vez que la revisión termina, el simulador busca otra entrada de la lista de evento futuros (`FEL`), para determinar el próximo evento en ser ejecutado. La `FEL` es, por supuesto, una colección de eventos: etiquetas de los nodos ordenadas según el tiempo fijado para su ejecución.

Éste es el algoritmo general de GLIDER. GALATEA, conserva esta estrategia general, excepto que, basado en la especificación proporcionada por la teoría de simulación multi-agentes [8], proporciona la ejecución de un motor de inferencia asociado a cada agente. Todos estos motores se ejecutan concurrentemente con el simulador principal y su ejecución se intercala cuidadosamente, para intercambiar información y permitir la simulación de un sistema multi-agentes en forma efectiva. Los motores de inferencia proporcionan la simulación del ciclo percibir-razonar-actuar de cada agente mientras el simulador controla la evolución física del sistema.

Este enfoque en el que la simulación de un sistema se realiza a partir de los componentes distribuidos no es una idea nueva. De hecho, como mencionamos anteriormente, estamos aprovechándonos de una especificación existente para simulación distribuida, incorporando la simulación en una "federación". El estándar HLA especifica las condiciones mínimas que un conjunto de componentes debe cumplir para ser convertidos en una federación. En nuestro caso, el simulador y cada uno de los agentes es un federado. Ellos se agrupan a en una federación multi-agentes que usa una estrategia simple, centralizada, de manejo de tiempo para controlar su propia evolución. En particular, en GALATEA los intercambios de información son siempre actualizaciones de la lista de eventos futuros, `FEL`, (o conjunto de influencias como las llamamos en otras partes de este documento). Los agentes incorporan sus intenciones (acciones que quieren ejecutar) en la `FEL` y el simulador decide el desenlace de las mismas. De igual forma, el simulador determina y programa las percepciones apropiadas para cada agente en la `FEL`, y cada agente "decide" lo que quiere ver.

Este algoritmo general para GALATEA, nos ha permitido especificar los detalles que deben regir el comportamiento del simulador. A continuación daremos una breve

descripción del comportamiento del simulador.

5.2.2 Comportamiento del simulador

El diagrama de secuencia de la figura 5.2 corresponde al proceso de ejecución en la plataforma GALATEA de una simulación que incluye agentes. Este diagrama muestra la interacción a lo largo del tiempo entre los objetos que participan, destacándose las actividades recursivas de activación y recorrido de la red de nodos y la activación de los agentes que participan en el sistema. Los componentes asociados al proceso de simulación son:

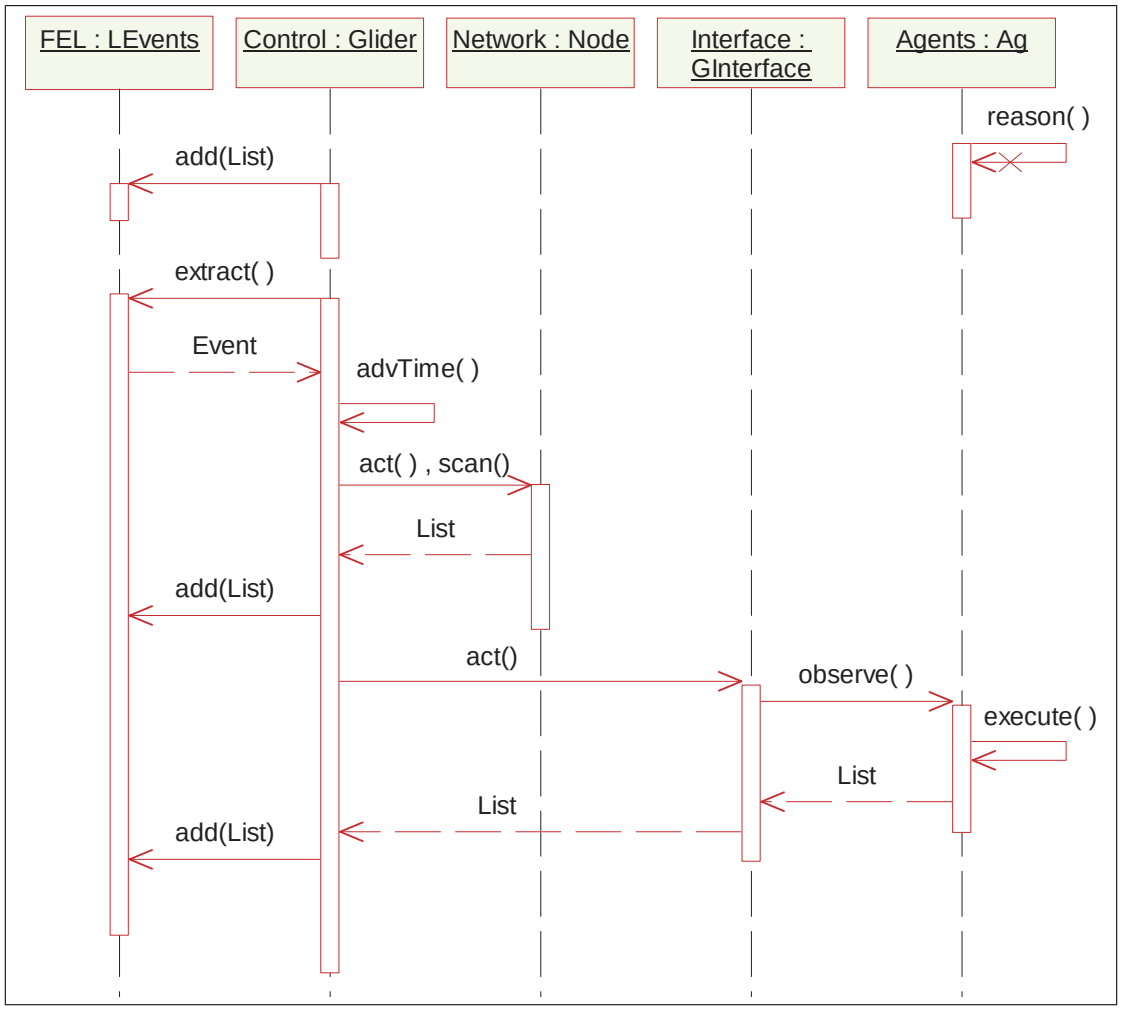

Figura 5.2: Diagrama de secuencia para GALATEA

FEL.

Contiene, cronológicamente ordenados, los eventos que deben "tener lugar" en la simulación del sistema. El objeto FEL realiza las siguiente operaciones:

1. incluye en la lista de eventos los eventos que recibe,
2. si se le solicita, entrega el evento que debe ser procesado.

Control.

Representa el controlador del proceso de simulación realizando las siguientes actividades:

1. solicita la programación de los eventos que permiten el inicio del proceso de simulación. Este procesamiento incluye la activación de la red de nodos y de los agentes,
2. solicita el próximo evento y controla que se procese el evento,
3. programa los nuevos eventos,

Network.

Representa el conjunto de subsistemas (nodos) presentes en el sistema a simular y por lo tanto, tiene como función controlar la interacción y activación de los nodos. Este componentes debe cumplir las especificaciones descritas en la sección 4.2 y debe llevar a cabo las siguientes actividades:

1. controla la activación del nodo actual,
2. controla la revisión de la red,
3. controla la activación de los agentes.

Interface.

Sirve de intermediario entre los agentes y el simulador. Este componente debe cumplir con las pautas descritas en 5.1.

Agents.

Representa al grupo de agentes presentes en el sistema, que siguen las especificaciones que mostraremos con detalle en la sección 5.3. Este componente tiene la función de controlar las actividades propias del agente:

1. razonar. Implica asimilar percepciones, observaciones, recuerdos, metas, creencias y preferencias para tomar decisiones sobre que afectarán su conducta,
2. observar. Percibir el ambiente,

3. actuar. Intentar modificar el ambiente de acuerdo a las decisiones tomadas.

Además, en el diagrama de colaboración de la figura 4.16 se muestran los intercambios de datos disparados por los eventos que han sido activados en el sistema. En este diagrama, se evidencia que el componente `Control` es quien tiene a su cargo el simulador, y ordena las actividades que se llevan a cabo, mientras que el componente `Interface` sirve de intermediario entre el simulador y los agentes. En el diagrama se destacan las siguientes colaboraciones:

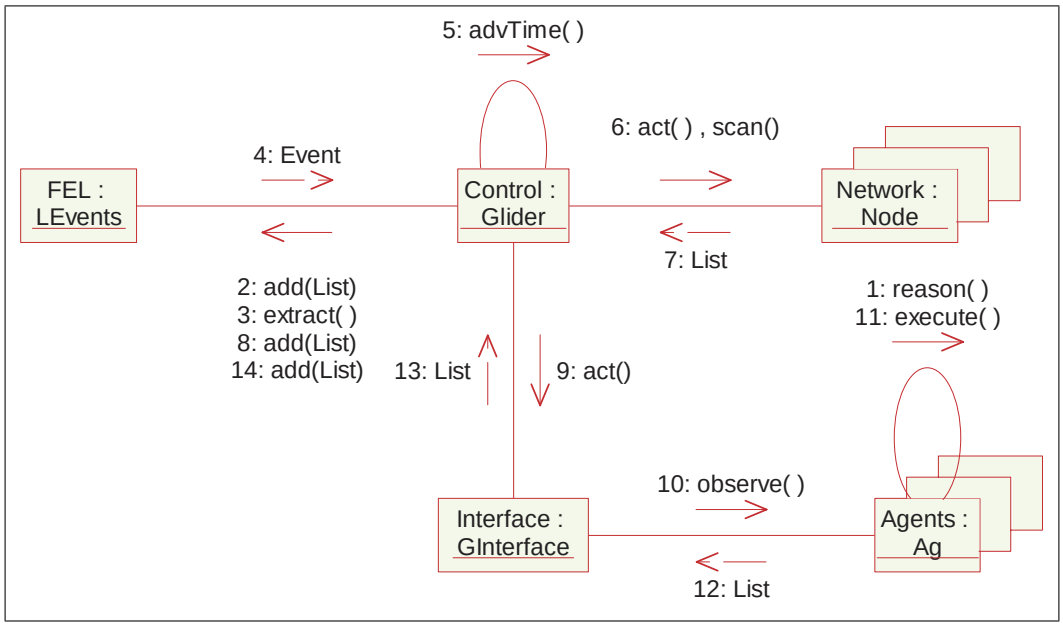

Figura 5.3: Diagrama de colaboración para GALATEA

1. Se inicia el proceso de razonamiento de los agentes invocando el método `reason()`. Este proceso se mantiene activo durante la ejecución de la simulación y solo es interrumpido momentáneamente mientras el agente actúa.
2. Solicita la incorporación de los eventos que inician el proceso de simulación haciendo uso del método `add()`.
3. Solicita el evento a procesar a través del método `extract()`.
4. Recibe un objeto tipo `Event`.
5. Extrae del evento el tiempo de activación y actualiza el reloj del simulador a través del método `advTime()`.

6. Si el evento esta asociado a un agente extrae la lista de acciones que deben ejecutarse en este momento y las ejecuta, pero si el evento esta asociado a un nodo se solicita la activación de dicho nodo a través del método `act()` y el recorrido de la red de nodos a través del método `scan()`. Los detalles de activación y recorrido se muestran en los diagramas del simulador de las figuras 4.15 y 4.16.
7. Recibe objetos `List` que contienen los eventos que deben programarse a causa de la activación del nodo y del recorrido de la red.
8. Solicita la incorporación de los nuevos eventos haciendo uso del método `add()`.
9. Solicita la activación de los agentes a través del método `act()`.
10. `Interface`, a solicitud de `Control` invoca en cada agente el método `observe()`.
11. `Interface` desencadena en cada uno de los agentes la ejecución del método `execute()` que determina las acciones que ha de intentar llevar a cabo el agente de acuerdo al razonamiento alcanzado hasta el momento.
12. `Interface` recibe de cada uno de los agentes que se ha activado un objeto `List` que contiene las influencias que deben programarse a causa del intento de los agentes de modificar su ambiente.
13. Recibe objetos `List` que contienen los eventos que deben programarse a causa de la activación de los agentes.
14. Solicita la incorporación de los nuevos eventos haciendo uso del método `add()`.

Cabe destacar que el paso 6 refleja el proceso recursivo de revisión de la red, descrito en la sección 4.1.2, que se repite secuencialmente hasta asegurar que se han reflejado los cambios causados en el sistema por la activación del nodo.

Por otro lado, el proceso de activación de los agentes, que corresponde a los pasos 9-13, se ejecuta simultáneamente en todos los agentes presentes en el sistema y está fuertemente vinculado al proceso de razonamiento de cada agente, presentado en el paso 1, el cual esta ejecutándose continuamente y es sólo interrumpido un momento mientras se realiza la activación del agente.

El ciclo de la simulación se inicia en el paso 3 y abarca el resto de pasos, se repite hasta finalizar la simulación: cuando se alcanza el tiempo determinado para simular, cuando no quedan eventos o cuando el simulista lo indique.

Con esta descripción del comportamiento culminamos la presentación del simulador, describiremos a continuación los agentes que pueden ser incluidos en una simulación en GALATEA.

5.3 Agentes del tipo reactivo-racional

Como mencionamos en la sección 3.2, nuestros agentes son de tipo reactivo-racional. En esta sección proporcionamos una descripción general de los detalles de construcción de este agente reactivo-racional [8] siguiendo la descripción de la sección 3.2.4.

Además, con la intención de completar la descripción de nuestra arquitectura, mostraremos los detalles que nos permitirán incluir varios agentes en nuestro modelo de simulación según las especificaciones de la sección 3.2.5.

5.3.1 Detalles del agente

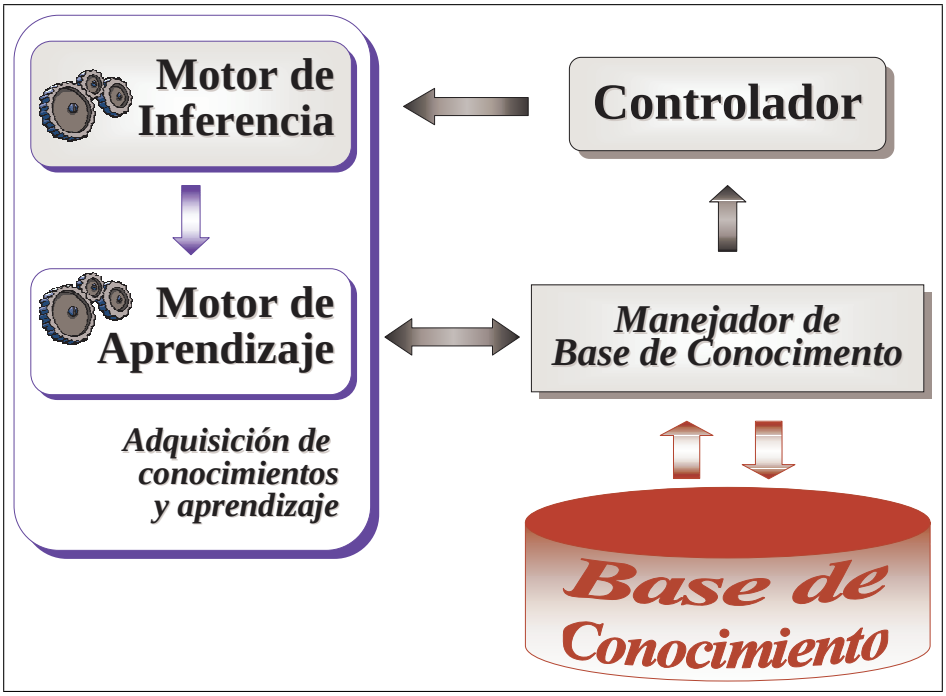

Figura 5.4: Componentes del agente reactivo-racional

La figura 5.4 muestra la arquitectura interna del agente. Esta arquitectura muestra que es posible definir un agente a través de sus componentes.

Como principal componente se destaca la **Adquisición de conocimiento y aprendizaje**. Este componente incluye un motor de aprendizaje y un motor de inferencia cuyo comportamiento esta asociado al formalismo en el cual representamos la base de conocimientos, así como también la metodología para construir la base de conocimientos automáticamente. Nuestro agente racional incluye dos características fundamentales: la adquisición de conocimiento y la capacidad de aprendizaje a las

que se le agrega alguna capacidad elemental para resolver problemas intrínsecos del aprendizaje y una base de conocimientos.

El componente **controlador** se encarga de ejecutar algunas operaciones independientes del dominio tales como comparar la sintaxis y las reglas, mientras que el **manejador de la base de datos** permite el acceso y la modificación del contenido de la base de datos.

Es necesario destacar que el agente debe ser implementado de forma tal que pueda ser configurado para un dominio de aplicación en particular. De esta forma, el modelista podrá representar algunas particularidades del razonamiento del agente.

Representación del Conocimiento

La representación del conocimiento se refiere a la correspondencia entre el dominio de aplicación externa y el sistema de razonamiento simbólico. En otras palabras, es la información que tiene el agente del mundo externo. Esta representación está conformada por una estructura de datos para almacenar la información y los métodos que permiten manipular dicha estructura. De esta forma, para cada elemento relevante del dominio de aplicación del agente existe una expresión en el dominio del modelo del agente que representa dicho elemento.

La representación del conocimiento se define con el fin de facilitar las operaciones básicas que involucran aprendizaje. Este mapeo entre los elementos del dominio de aplicación y del dominio del modelo permite a los agentes razonar acerca del dominio de aplicación al ejecutar procesos de razonamiento en el dominio del modelo y transferir las conclusiones de vuelta al dominio de aplicación.

Una característica básica de una representación es su significado semántico el cual en el caso de nuestro agente esta basada en lingüística. Cada elemento en el dominio de aplicación es representado en el dominio del modelo como un símbolo. En el mismo orden de ideas, una entidad compleja esta representada en término de sus elementos y su significado está dado por el significado de los símbolos que representan dichos elementos.

Al definir la representación de conocimientos para un agente racional es necesario considerar cuatro características importantes relacionadas con los componentes de un agente: representabilidad, inferenciabilidad, eficiencia al resolver problemas, aprendizaje eficiente.

Representabilidad. Se refiere a la habilidad de representar las clases de conocimiento presentes en cierto dominio de aplicación.

Inferenciabilidad. Se refiere a la habilidad de representar los procedimientos de inferencia necesarios en el dominio de aplicación. Los procedimientos de inferencia manipulan la estructura de representación con el fin de derivar nuevas estructuras correspondientes al

nuevo conocimiento inferido a partir del conocimiento anterior. Estos procedimientos se utilizan durante la solución de problemas y durante el aprendizaje.

Eficiencia al resolver problemas. Se refiere a la habilidad de representar eficientemente los métodos que permiten resolver los problemas propios del agente. Un ejemplo podría ser incorporar en la estructura de conocimiento información adicional que pueda ser usada en enfocar los mecanismos de inferencia en la dirección más prometedora.

Aprendizaje eficiente. Se refiere a la habilidad para adquirir conocimiento e incorporar nueva información a la base de conocimientos y a la habilidad de modificar la estructura existente con miras a mejorar la representación actual del dominio de aplicación.

Como se muestra en la figura 5.4 la representación del conocimiento y el razonamiento del agente se fundamentan en la base de conocimientos. A continuación mostraremos los detalles de implementación de la base de conocimientos.

Base de Conocimientos

La base de conocimientos contiene la información disponible para el agente. Existen muchas alternativas de implementación de este almacén de conocimientos. Sin embargo, en el desarrollo de una base de conocimientos se pueden distinguir tres fases:

1. Incorporación
2. Refinamiento
3. Reformulación

Durante la etapa de incorporar conocimiento se define el conocimiento básico del agente. Es importante seleccionar el esquema de representación de conocimiento y desarrollar la terminología básica y la estructura conceptual de la base de datos. El resultado de esta fase es una base de conocimientos inicial, generalmente incompleta y compuesta por conocimiento parcialmente incorrecto que posteriormente será refinado y mejorado durante las siguientes etapas del desarrollo.

En la fase de refinamiento se extiende y se depura la base de conocimientos. El resultado de esta fase debe ser una base de conocimientos sustancialmente completa y correcta que permita proporcionar soluciones correctas a los problemas que el agente debe resolver.

Durante la fase de reformulación se organiza la base de conocimientos para mejorar la eficiencia del controlador a la hora de contrastar las reglas.

La descripción del agente incluye varias aspectos y hasta ahora hemos presentado sólo la arquitectura interna del agente. A continuación mostraremos como realizar una descripción de un agente utilizando lógica.

Descripción Lógica del Agente

Esta sección presenta una especificación para un agente inteligente en lógica clásica. Este tipo de especificación puede traducirse sistemáticamente en un código ejecutable que describa al agente. Esto nos permite con un único lenguaje establecer la teoría, implementar la arquitectura y programar el agente.

Normalmente se acepta que la descripción de un agente requiere " modalidades" para representar nociones intencionales como conocimiento, creencias y incluso las metas [54]. Debido a esto las lógicas modales resultan muy atractivas: las descripciones en los lenguajes de lógica modal son muy cercanas a los lenguajes naturales. Los modelistas pueden indicar una modalidad para cada noción (conocimiento, creencias, metas, etc.). Con este conjunto de modalidades se pueden declarar sentencias directamente traducible en la lógica modal como:

> Don Quijote cree que Rocinante es un caballo magnífico.
> Don Quijote quiere ser un caballero.
> Don Quijote cree que él sabe quién es el amor de Dulcinea.

Es cierto que la lógica modal es muy cercana al lenguaje natural, y por consiguiente resulta fácil traducir especificaciones técnicas. Sin embargo, este tipo de lógica posee una semántica más compleja: la semántica de los mundos posibles. Los constructos fundamentales de esta semántica son los mundos. Estos mundos son configuraciones conceptuales del universo donde cada uno representa un estado posible en el que el universo pueda encontrarse.

En esta semántica algo es posiblemente verdad si es verdad en algunos de los mundos y algo es necesariamente verdad si es verdad en todos los mundos posibles.

Los lógicos modales modernos [55] explican que es posible adaptar esta semántica para permitir además de la posibilidad y la necesidad otras modalidades tales como la intencionalidad. Lamentablemente, esto no es aplicable en todos los casos. Por ejemplo, si K es la modalidad intencional para el conocimiento, el axioma

$$KKx \leftarrow Kx$$

puede generar que se ejecute una formulación en forma infinita

yo se que yo se que $\cdots$

obviamente esto no sucede con un agente real, y en particular no ocurre con los humanos.

Los lógicos plantean que la lógica modal permite describir un agente "ideal" capaz de deducir todas las consecuencias de sus creencias y su conocimiento y que dicho agente "ideal" no debe ser confundido con los agentes reales.

Aunque en el diseño e implementación de teorías o arquitecturas de agentes esta inconsistencia es pasada por alto, al momento de modelar y de implementar sistemas multi-agentes los modelistas desean describir un agente más real, que no se exponga a estos ciclos infinitos.

Para explicar como utilizar la lógica clásica para establecer una especificación más realista de un agente en [56] y [41] se muestran programas lógicos que permiten especificar un agente reactivo-racional. Esta especificación, extendida con el procedimiento de prueba mostrado en [57] se incorpora al razonamiento del agente en [20] que se emplea en la teoría de simulación de sistemas multi-agentes [8] en la que se apoya GALATEA.

Para la descripción de un agente en este contexto, es necesario:

* Una base de conocimiento con definiciones de la forma si y sólo si que será usada para reducir metas a sub-metas.
* Un conjunto de metas dadas en forma de átomos o implicaciones, tales que:
  - Sea posible representar metas condicionales como reglas de condición-acción generalizadas y prohibiciones.
  - Los átomos puedan representar acciones complejas o atómicas.
  - Las observaciones se incorporen a las metas.
* Representar explícitamente el tiempo haciendo uso del cálculo de eventos de la forma

  se_tiene(paso_libre,25/07:10:10) .
* Especificar la forma en que avanza el tiempo. En cada paso de la iteración es necesario limitar el razonamiento a pequeñas porciones de tiempo y permitir que se pueda reanudar en la próxima iteración.

La evolución en el tiempo que describe la vida de un agente esta determinada por el predicado *ciclo*() presentado en [56], que ha sido actualizado y mejorado en los trabajos antes mencionados. Nuestra versión para este predicado especifica que en T deben cumplirse los siguientes pasos:

---

1. Observar la porción del estado del ambiente a la cual tiene acceso en T, asimilar la observación como una meta y registrar dicha observación en la base de conocimientos.

2. Ejecutar el procedimiento de reducción de metas a submetas en $T+1$. Este procedimiento permite propagar las observaciones, durante R unidades de tiempo
3. Seleccionar entre las metas que se ofrecen como alternativa acciones atómicas que puedan ser ejecutadas y postular dichas acciones en el tiempo $T+R+2$.
4. Repetir el ciclo en el tiempo $T+R+3$.

Usando este predicado, en [20] se muestra que es posible realizar una descripción basada en lógica de un agente reactivo e inteligente. Esta descripción, denominada GLORIA[1], explica el comportamiento de un agente a través de los predicados:

$$
\begin{array}{l}
cycle(KB, Goals, T) \\
\quad \leftarrow\ demo(KB, Goals, Goals', R) \\
\quad \wedge\ R \leq n \\
\quad \wedge\ act(KB, Goals', Goals'', T+R) \\
\quad \wedge\ cycle(KB, Goals'', T+R+1) \qquad \textbf{[GLOCYC]} \\
act(KB, Goals, Goals', T_a) \\
\quad \leftarrow\ Goals \equiv PreferredPlan \vee\ AltGoals \\
\quad \wedge\ executables(PreferredPlan, T_a, TheseActions) \\
\quad \wedge\ try(TheseActions, T_a, Feedback) \\
\quad \wedge\ assimilate(Feedback, Goals, Goals') \qquad \textbf{[GLOACT]} \\
executables(Intentions, T_a, NextActs) \\
\quad \leftarrow\ \forall A, T(do(A,T)\ is\_in\ Intentions \\
\qquad \wedge\ consistent((T = T_a) \wedge\ Intentions) \\
\qquad \leftrightarrow\ do(A, T_a)\ is\_in\ NextActs) \qquad \textbf{[GLOEXE]} \\
assimilate(Inputs, InGoals, OutGoals) \\
\quad \leftarrow\ \forall A, T, T'(action(A, T, succeed)\ is\_in\ Inputs \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals \\
\qquad \rightarrow\ do(A, T)\ is\_in\ NGoal) \\
\quad \wedge\ \forall A, T, T'(action(A, T, fails)\ is\_in\ Inputs \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals \\
\qquad \rightarrow\ (\textbf{false} \leftarrow\ do(A, T))\ is\_in\ NGoal) \\
\quad \wedge\ \forall P, T(obs(P, T)\ is\_in\ Inputs \\
\qquad \rightarrow\ obs(P, T)\ is\_in\ NGoal) \\
\quad \wedge\ \forall Atom(Atom\ is\_in\ NGoal \\
\qquad \rightarrow\ Atom\ is\_in\ Inputs \\
\quad \wedge\ OutGoals\ \equiv\ NGoal\ \wedge\ InGoals \qquad \textbf{[GLOASSI]} \\
A\ is\_in\ B\ \leftarrow\ B\ \equiv A\ \wedge\ Rest \qquad \textbf{[GLOISN]} \\
try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment... \qquad \textbf{[TRY]}
\end{array}
$$

Figura 5.5: GLORIA: Especificación Lógica de un agente

cycle(), correspondiente a nuestro predicado *ciclo*().

demo(), corresponde al paso 2 de nuestro predicado *ciclo*(). Permite reducir metas *Goals* a submetas *Goals'* a partir de la base de conocimientos KB controlando además el uso de la cantidad de recursos R disponibles para llevar a cabo este proceso de razonamiento.

[1] **G**eneral-purpose, **L**ogic-based, **O**pen, **R**eactive and **I**ntelligent **A**gent

act() corresponde al paso 3 de nuestro predicado *ciclo*(). Cambia la estructura mental del agente, si en T_a el plan preferido del agente $PreferredPlan$ contiene acciones $TheseActions$ que puedan ser ejecutadas en T_a, estas acciones se postulan en paralelo y se incorpora nuevas metas con la información obtenida como respuesta.

executables() permite obtener el grupo de acciones $NextActs$ correspondiente a la lista de intenciones $Intentions$ que pueden ser ejecutadas en T_a.

try() Intenta ejecutar la lista de acciones $Output$ en T obteniendo como respuesta $Feedback$.

assimilate() corresponde al paso 1 de nuestro predicado *ciclo*(). Permite actualizar la base de conocimientos ya que asimila las entradas $Inputs$ las incorpora con las metas del agente $InGoals$ para obtener el nuevo conjunto de metas $OutGoals$ que incorpora la información correspondiente a la entrada dada.

Los agentes inmersos en un sistemas están rodeados de información. Esta información se genera en forma continua durante el ciclo de vida del agente. Por este motivo, es necesario evaluar, resumir y sintetizar esta información con el fin de facilitar y agilizar la toma de decisiones por parte del agente.

Razonamiento y Aprendizaje

Los agentes deben ser lo suficientemente flexibles para adaptarse a cambios en el ambiente y a cambios en sus requerimientos. Las estrategias de razonamiento les permiten a los agentes anticipar las consecuencias de las posibles acciones a ejecutar y así escoger la acción más "racional".

El razonamiento es una operación que consume recursos. Nuestros agentes combinan el razonamiento con un patrón de comportamiento reactivo que le permite reaccionar a tiempo según la dinámica del sistema.

Es difícil definir con precisión en que consiste el aprendizaje de un agente. Una de las razones es que el término *aprendizaje* es utilizado comúnmente bajo diferentes connotaciones en campos como psicología, educación, ciencia de conocimiento, inteligencia artificial, etc. Sin embargo, la definición más ampliamente utilizada es la dada en términos de la habilidad del agente para mejorar su propio desempeño en algún dominio, basándose en su experiencia pasada.

Abducción

El razonamiento abductivo se basa en que el agente adopta una hipótesis que explica sus observaciones. En este caso para registrar aprendizaje el agente necesita una observación que no se siga deductivamente de la base de conocimientos y una base de conocimientos que explique parcialmente dicha observación.

El razonamiento abductivo permite incorporar nuevo conocimiento en la base de conocimientos. A través del proceso de refinamiento se logra extender la base con nuevas piezas de conocimiento y usualmente esto mejora la eficiencia del aprendizaje.

Generalmente, en el proceso de razonamiento abductivo se utiliza la representación clausal del conocimiento ya que esta facilita la exploración que se realiza para explicar la observación. En este caso la entrada debe ser un hecho y la base contiene conocimiento clausal relativo al hecho. El objetivo es incorporar una nueva pieza de conocimiento que podría ser considerada para nuevas entradas.

En nuestro agente, usamos abducción en un contexto diferente. La usamos para "explicar" metas. En el proceso de reducción de metas a sub-metas, arribamos a una sub-metas atómicas que no podemos deducir, pero que si podemos postular como condiciones que de cumplirse implicarían el logro de las metas superiores de donde se obtuvieron. Estas submetas atómicas son las acciones y un conjunto de estas acciones obtenidas para una meta, constituyen un *plan*. Así es como abducción se convierte en el mecanismo de planificación del agente. Así, lo que el agente conoce de su ambiente determina sus planes de acción.

Además de los complejos procesos internos que hemos mostrado del agente, al incorporar los agentes a un sistemas donde convive con otros agentes se presentan algunos inconvenientes que debemos resolver. A continuación presentaremos los detalles de nuestro Sistema Multi-Agentes.

5.3.2 Detalles del Sistema Multi-Agentes

El problema de coordinación central en un sistema multi-agentes como el mostrado en la figura 5.6 es fundamental, ya que se puede incluso opacar las habilidades individuales de cada agente impidiendo que el modelo de simulación represente el sistema real. Por lo tanto, el esquema de comunicación entre los agentes y de éstos con el ambiente debe verificarse según con dos aspectos:

1. Cualitantitativamente la comunicación debe ser tal que globalmente los agentes puedan llevar a cabo las tareas para los cuales fueron modelados.

2. Cuantitativamente debe asegurar rapidez y eficiciencia en el desempeño de los agentes.

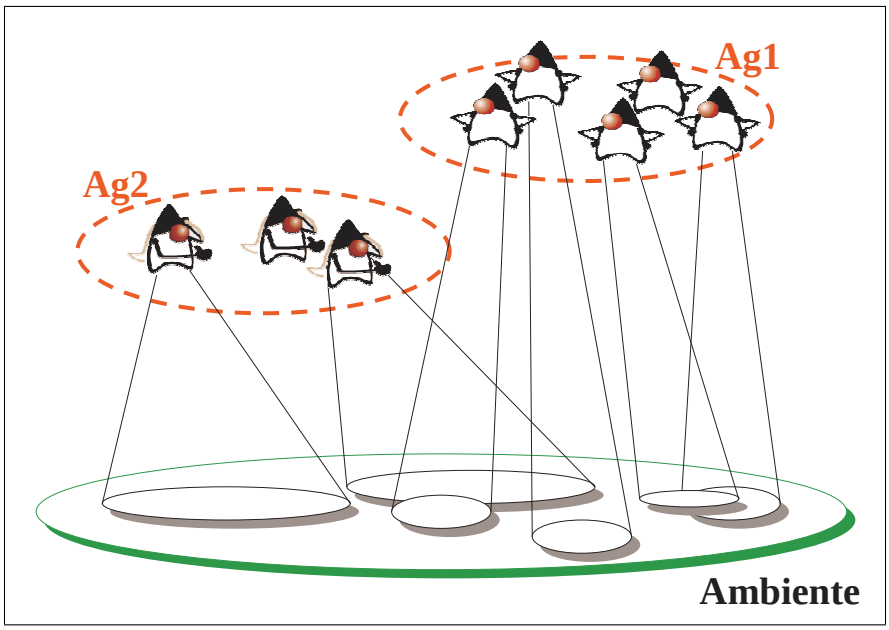

Figura 5.6: Ejemplo de sistema multi-agentes

Comunicación entre agentes

La comunicación entre nuestros agentes se realiza a través de pase de mensajes utilizando el ambiente como intermediario. Inicialmente la comunicación se realizará a través de "actos verbales", es decir, influencias especiales que pueden modificar el estado estado interno de los agentes, sin variar el ambiente, aunque este sirva de medio de transmisión.

Comunicación con el ambiente

La comunicación de nuestros agentes con el ambiente esta vinculada a las nociones de percepción, de estado interno y de influencias de los agentes que mencionamos en 3.2. A continuación aclararemos la procedencia de dichas nociones:

En [58] se utiliza una función de percepción propia de cada agente

$$Percibe : \Sigma \rightarrow P_a \tag{5.1}$$

donde P_a es una partición de Σ, tal que si

$$\Sigma = \{(encendio, caliente), (encendido, frío), (apagado, caliente), (apagado, frío)\}$$

es posible tener $P_a = \{encendido, apagado\}$ que indica que el agente a no percibe

todo su entorno, solo percibe una partición del conjunto de estados internos en que se puede encontrar el ambiente.

Por otro lado, en [40] se adopta una función de percepción diferente en la que separan las influencias de las reacciones. Esta función es del tipo

$$Percibe : \Gamma \rightarrow P_a \tag{5.2}$$

donde la percepción puede estar perfectamente "localizada": el agente percibe todo aquello que lo influencia y no percibe los estados internos del ambiente. En este caso es necesario identificar quien causa cada influencia y a quienes afecta.

Nuestra función de percepción permite al agente percibir el estado del ambiente y aquello que lo influencia. Esta forma es muy conveniente al momento de la representación del agente ya que nos permite establecer que el agente percibe una forma estática del sistema, el ambiente, y un componente dinámico del mismo, las influencias.

$$Percibe : \Sigma \times \Gamma \rightarrow P_a \tag{5.3}$$

Las influencias están asociadas con marcas de tiempo, este hecho tiene efectos sobre la percepción: todo aquello que el agente percibe puede ser configurado a partir de la historia de las influencias anteriores, por deducción a partir de datos en diversos instantes de tiempo. Así, por ejemplo, el agente podría "percibir" movimiento, analizando perceptos sobre la misma propiedad en instantes consecutivos.

Para efectos de representación, la información percibida será registrada en la base de conocimiento como afirmaciones atómicas con la forma:

$$obs(P, T)$$

para indicar que el agente observa la propiedad (fluente, cambiante) P en el instante T. Esta representación puede usarse para modelar lo que entra al agente desde el exterior. Pero el agente puede también "observar" eventos y acciones con la forma:

$$do(Alguien, Acción, T_0, T_f)$$

que indica que el agente observa que *Alguien*, algún otro agente, realizó la acción *Acción* entre T_0 y T_f.

Más aún, y esto es clave para los agentes que se emplean en tareas de vigilancia de su entorno, la misma representación puede ser usada, desde el agente y hacia el ambiente, para programar acciones de observación en el agente. Por ejemplo:

$$do(yo, obs(botónA), 1, T)$$

puede ser tomada por el agente como una instrucción para que el agente por si mismo

mismo (*yo*) observe el objeto indicado a partir del tiempo 1.

En el procedimiento de prueba descrito en [20], estos tres usos de la representación se integran coheremente en el sistema de planificación del agente.

Para culminar la descripción de nuestro sistema multi-agentes queremos hacer notar que la aparición y desaparición de agentes, la variación de sus estructuras de comunicación son típicos en sociedades de agentes autónomos. Un patrón de interacción dinámico y una composición dinámica de la sociedad pueden ser representados en un modelo que permita cambio estructural [17]. Con la plataforma GALATEA pretendemos proporcionar las facilidades para crear y eliminar agentes, para incluir y excluir agentes en grupos y para alterar la estructura del ambiente en que están inmersos dichos agentes asegurando la consistencia del modelo. Sin embargo, para llevar a cabo este propósito, es necesario extender la teoría de simulación de sistemas multi-agentes [8] que esta formulada para un grupo constante de n agentes.

5.3.3 Implementación del agente

La figura 5.7 muestra nuestra primera implementación del agente (código 18) donde se reflejan los métodos que que corresponden los predicados mostrados en la descripción lógica del agente (sección 5.3.1) y a las funciones matemáticas de la jerarquía de agentes de la sección 3.2:

`cycle()`. Este método corresponde al predicado *cycle*(), y a la función *Evolución* (ec. 3.20).

`observe()`. Informa al agente el estado del ambiente. Este método esta asociado al predicado *act*() y a la función $Percibe_a$ (ec. 5.3).

`execute()`. Comunica al ambiente las intenciones del agente. Este método esta asociado al predicado *act*() y a la función *Acción* (ec. 3.13).

`reason()`. Implementa los mecanismos de adquisición de conocimiento del agente. Este método corresponde al predicado *demo*() y a la función $Planifica$ (ec. 3.18).

Por otro lado, en la figura 5.7 se muestran también las estructuras de datos para almacenar las metas (`Goals`), las creencias (`Beliefs`), las preferencias (`Preferences`), las prioridades (`Priorities`) y una meta permanente (`permGoal1`) del agente. Además, se evidencia que el manejo de la información se hace a través de listas: `observations` contiene las percepciones del agentes mientras que `influences` contiene las influencias que postula el agente.

Código 18 Implementación del agente

```
import galatea.hla.*;
public class Ag extends Agent implements Runnable {
  public Ag() { Ag(null,null) ; }
  public Ag(List obsArea, List inflArea){
    list body = new List();
    body.add("it rains");
    Goal permGoal1 = new Goal("carry umbrella",body);
    super(obsArea,inflArea,new Goals(),null,null); }
  /** The cycle/locus of control. */
  public void cycle() {
    observe();
    reason();
    List result = execute();
    result.toString() ;
    cycle() ; }
  /** Tries to execute its intentions */
  public List execute() {
    return influences ; }
  /** Update its knowledge of its environment.*/
  public void observe() {
    // Get inputs from the environment.. somehow..
    List obs = new List() ;
    obs.add("it rains");
    // Update its records.
    observations = obs ; }
  /** Reasoning engine of the agent. */
  public void reason() {
    // Every goal must be checked against observations..
    if (permGoal1.fired(observations)) {
        goals.activateGoal(permGoal1) ;
    };
    influences = new List() ;
    influences.add(goals.allGoals[goals.intention]); }
  /** */
  public void run() {
      this.permGoal1.toString();
      this.cycle() ;
  }
  /** Test the agent*/
  public static void main(String argv[]) {
    Ag agent = new Ag() ;
    agent.permGoal1.toString();
    agent.cycle() ; }
}
```

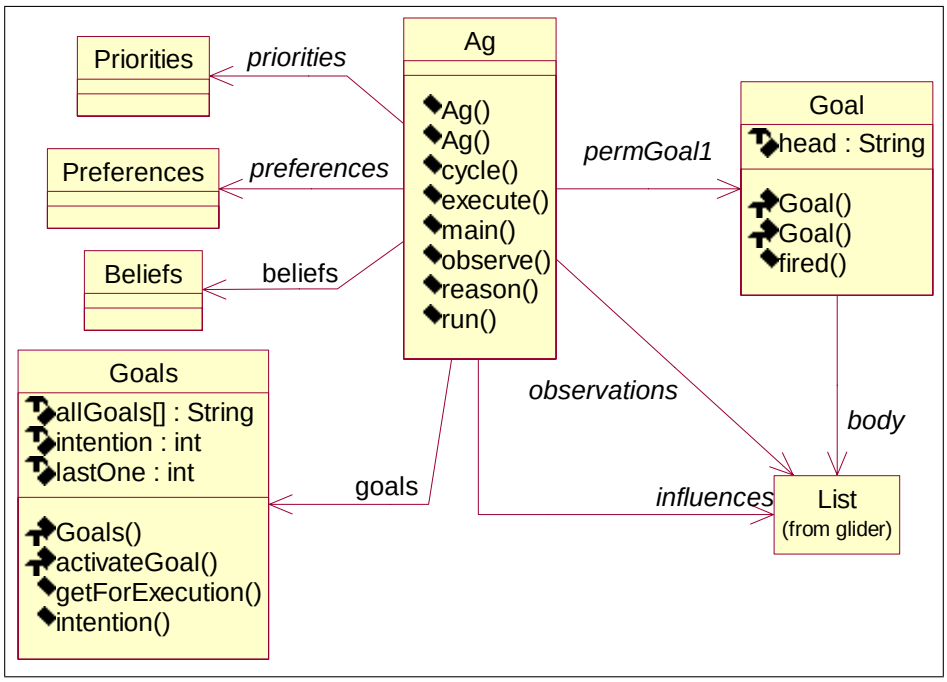

Figura 5.7: Diagrama de clases de la implementación del agente

Para hacer que los agentes haga lo que se espere de ellos, es preciso codificarle conocimiento operacional. Esto, a su vez, requiere de lenguajes de programación para agentes y por supuesto en nuestro caso también necesitamos un lenguaje que me permita incorporarlos en la plataforma de simulación.

5.4 Familia de Lenguajes

La familia de lenguajes que conforman GALATEA incluye:

1. Los fundamentos básicos del lenguaje GLIDER, descritos con detalle en [52]
2. Un grupo de lenguajes basados en lógica, como se muestra en la sección 5.3.1, que permiten describir a los agentes
3. Una extensión a la sintaxis y a la semántica GLIDER que permite incorporar agentes al modelo,
4. Las facilidades y la estructuras del lenguaje Java en lugar del lenguaje PASCAL.

Para familiarizarnos con estos lenguajes, a continuación mostraremos la estructura de un programa escrito para GALATEA, la cual será utilizada posteriormente para presentar algunos ejemplos en la sección 5.5. Posteriormente, en la sección 5.4, mostraremos la semántica operacional bosquejada para GALATEA.

5.4.1 Estructura de un programa GALATEA

La figura 5.8 muestra la estructura de un programa de GALATEA, el cual consta de siete secciones básicas:

```
                        Encabezado
TITLE
                        Título del modelo
NETWORK
                        Descripción de la Red
AGENTS
                        Descripción de los Agentes
    GOALS                   Metas
    BELIEFS                 Creencias
    PREFERENCES             Preferencias
INTERFACE
                        Descripción de las relaciones entre
                        los agentes y el ambiente
INIT
                        Valores iniciales para variables y
                        estructuras
DECL
                        Declaración de variables
END.
```

Figura 5.8: Estructura de un programa GALATEA

Las secciones que corresponden a las etiquetas `TITLE`, `NETWORK`, `INIT` y `DECL` se comportan como las secciones equivalentes de un programa GLIDER (ver sección 4.1.1). En GALATEA, es necesario incorporar las instrucciones para representar aquellos mensajes tipo agentes y los agentes implícitos que no recorren la red presentes en el modelo.

A la especificación del programa GLIDER se anexan dos secciones:

Agentes: Esta sección empieza con la etiqueta `AGENTS` y contiene descripción de cada tipo de agente, como se muestra en [8]. Esta sección contiene la especificación interna del agente, el estado mental (base de conocimiento, procedimientos, metas y preferencias). El código ubicado aquí permite particularizar a cada tipo de agente.

La sección `AGENTS` por si misma es una colección de dispositivos de representación de conocimiento. Para cada entrada, hay una subdivisión (`GOALS`) para describir las metas del agente, una para describir las creencias (`BELIEFS`) y otra para describir las preferencias (`PREFERENCES`). Cada una de las subdivisiones utiliza su propio lenguaje lógico. La riqueza de los lenguajes y la conveniencia de la lógica para capturar conocimiento declarativo y procedural justifica la presencia de la lógica en esta parte del modelo del sistema. Además, creemos que estos lenguajes (discutidos en [20]) son la mejor forma de incluir piezas de conocimiento humano en un motor.

Interfaz: Esta sección empieza con la etiqueta `INTERFACE`. En esta sección se incluye la descripción de las relaciones entre los agentes y el resto del sistema. Esta sección contiene el código Java que describe el comportamiento del simulador para permitir a los agentes percibir y actuar. Es decir, se relaciona las acciones de los agentes y las percepciones de los estados actuales del mundo. Aquí el modelista define las acciones que el agente ejecutará o cuando y por qué un agente percibirá ciertas propiedades de su ambiente.

Aquí, con toda la expresividad del marco de trabajo OO, los programadores pueden especificar los efectos reales de las acciones de los agentes, incluyendo esas acciones que involucran a muchos agentes y, por consiguiente, puede involucrar la acción combinada de múltiples influencias. Además acá el programador declara todo aquello que perciben los agentes en cada circunstancia en la que podrían estar involucrados.

5.4.2 Semántica operacional

Una semántica operacional asocia los constructos de un lenguaje de programación (las instrucciones, declaraciones y cualquier otro tipo de anotación que uno pueda hacer en el lenguaje) con transformaciones del estado de una máquina ideal, diciendo que esas instrucciones *tienen por significado* a esas transformaciones. De esta forma, es posible discutir sobre el significado del lenguaje de programación, independientemente de la plataforma particular de computación sobre la que se le emplee. Las máquinas ideales que soportan las semánticas operaciones son especies de máquinas virtuales: un conjunto de instrucciones y estados que han sido acordados, por los computistas, como la base funcional de ciertos procesos de cómputos.

Desde el inicio del proyecto GALATEA, hemos querido aproximarnos a la especificación formal de una semántica operacional para el lenguaje de simulación. Nos ha parecido muy importante que los modelistas y simulistas puedan discutir sobre el

cómo funciona el simulador y cómo se le codifican los modelos, con total independencia (esto es, sin tener que conocer los detalles) de las implementaciones particulares (software y hardware) del simulador. Así es posible concentrarse en los aspectos de expresividad del lenguaje y computabilidad del modelo, aparte de los detalles de cómo se le hace trabajar en los experimentos particulares.

La teoría de simulación que se presentó brevemente en el capítulo 3 viene a prestar un servicio fundamental en la especificación de la semántica de GALATEA. Aunque es todavía trabajo en progreso, podemos usar la teoría (que es esencialmente la descripción de una máquina abstracta) para explicar como funciona el simulador, dado un modelo de simulación escrito para GALATEA.

En la ecuación (3.25) se muestra como la función *reacción*() produce un nuevo estado global, a partir del estado anterior y de las influencias producidas por los agentes en la transición en curso, $\cup_a \gamma_a$. Sin embargo, para producir el siguiente estado global la función debe ser provista también de un conjunto de leyes y de cierto conocimiento adicional. Estos dos elementos (Λ y β) son obtenidos a partir del código suministrado por el modelista en la forma de un modelo de simulación.

La definición de *reacción*() debe completarse entonces con:

$$\Lambda = selecciona(Network, \xi) \tag{5.4}$$
$$\xi = sigEvento(\gamma) \tag{5.5}$$
$$t' = tiempo(\xi) \tag{5.6}$$
$$\beta = interpreta(InitDecl) \tag{5.7}$$

La función *selecciona*() obtiene la lista de "leyes de cambio" del sistema, a partir la sección `NETWORK` del modelo. Así, $Network$ es una descripción del sistema orientada a la red, como se dice en simulación, y es como vimos en la sección 4.1 el elemento lingüístico principal de un modelo GLIDER.

La función *interpreta*() obtiene la información de soporte del modelo GALATEA. $InitDecl$ representa el conocimiento que se tiene de las condiciones iniciales del sistema y de los parámetros que permiten definir escenarios al momento de realizar la simulación. Además proporciona información acerca de estructura del sistema. Esta información es tomada de las secciones `INIT` y `DECL` del modelo.

Las funciones *sigEvento* y *tiempo* son bien conocidas en la simulación tradicional, pues caracterizan al paradigma DEVS [9]. La primera selecciona el proximo evento a ocurrir a partir de la llamada lista de eventos futuros, la cual, gracias a nuestra simulación, podemos caracterizar perfectamente como la lista total de "influencias".

Estas dos funciones completan la caracterización de un simulador DEVS y, en cuanto a la semántica, nos permiten definir el significado de la declaración `NETWORK` en un modelo de simulación GALATEA.

Para continuar la fundamentación semántica, completaremos la definición de la función *reacción*(): Si Λ representa el conjunto de todas la leyes de cambio del sistema y tanto **scan** como **noscan** son etiquetas suministradas como argumentos para indicar si procede o no la revisión de la red (ver secciones 4.1 y 5.2) la función *reacción*(), correspondiente a la reacción del ambiente, se define como

$$\mathit{Reacción} : (\Lambda \cup \{;\}) \times (\Lambda \cup \{;\}) \times \{\textbf{scan,noscan}\} \times B \times \mathcal{T} \times \Sigma \times \Gamma \rightarrow \Sigma \times \Gamma,$$

Esta definición conduce a una versión más detallada de la función *reacción*() la cual puede escribirse como un programa lógico:

$$\mathit{reacción}(\varepsilon, \Lambda, \textbf{noscan}, \beta, t, \sigma, \gamma, \sigma, \gamma). \tag{5.8}$$

$$\begin{aligned}&\mathit{reacción}(\varepsilon, \Lambda, \textbf{scan}, \beta, t, \sigma, \gamma, \sigma', \gamma') \leftarrow \\ &\quad \mathit{reacción}(\Lambda, \Lambda, \textbf{noscan}, \beta, t, \sigma, \gamma, \sigma', \gamma').\end{aligned} \tag{5.9}$$

$$\begin{aligned}&\mathit{reacción}([\lambda \mid R], \Lambda, \mathit{Etiqueta}, \beta, t, \sigma, \gamma, \sigma'', \gamma'') \leftarrow \\ &\quad \lambda =< ID_\lambda, PreConds_\lambda, PreInfl_\lambda, Reduc_\lambda > \wedge \\ &\quad PreConds_\lambda(\sigma) \wedge \\ &\quad PreInfl_\lambda(\gamma) \wedge \\ &\quad reduce(ID_\lambda, Reduc_\lambda, \beta, t, \sigma, \gamma, \sigma', \gamma') \wedge \\ &\quad \mathit{reacción}(R, \Lambda, \textbf{scan}, \beta, t, \sigma', \gamma \cup \gamma', \sigma'', \gamma'')\end{aligned} \tag{5.10}$$

$$\begin{aligned}&\mathit{reacción}([\lambda \mid R], \Lambda, \mathit{Etiqueta}, \beta, t, \sigma, \gamma, \sigma'', \gamma'') \leftarrow \\ &\quad \lambda =< ID_\lambda, PreConds_\lambda, PreInfl_\lambda, Reduc_\lambda > \wedge \\ &\quad \neg(PreConds_\lambda(\sigma) \wedge PreInfl_\lambda(\gamma)) \wedge \\ &\quad \mathit{reacción}(R, \Lambda, \textbf{scan}, \beta, t, \sigma, \gamma, \sigma'', \gamma'')\end{aligned} \tag{5.11}$$

donde ε corresponde a una lista vacía de leyes, mientras que λ y R corresponden a leyes del sistema tales que $\Lambda = \lambda \cup R = (\lambda \mid R)$ y $\mathit{Etiqueta} \in \{\textbf{scan}, \textbf{noscan}\}$.

Cada λ es una ley, un conjunto de instrucciones (con un único identificador ID_λ), que conduce a cierto estado global, siempre que las pre-condiciones, *preConds*, y pre-influencias, $prefInfl$, para ese estado global se cumplan en el estado actual (σ, γ).

La variable *Reduc* contiene el conjunto de leyes que son reducibles y corresponde a un fragmento de código (sección `NETWORK`) que puede ser "reducido" sistemáticamente, haciendo uso de la función *reduce*(), a un conjunto de declaraciones acciones que transforman el estado global.

Esta estrategia de reducir código procedimental a declaraciones de las acciones de un agente, es muy similar a la reducción de cláusulas lógicas a átomos "abducidos", como en programación lógica abductiva [41] y puede ser usada para proveer la semántica operacional para el lenguaje usado para escribir el código procedimental,

como se explica en [59].

Operacionalmente, el sistema interpreta las entradas en código GALATEA como un programa que sirve de guía a la simulación. Declarativamente, para el ejemplo del sistema de ferrocarril de la sección 4.4, es posible utilizar:

```
Network = [ Partida (I) Ruta::
                Vagones:= UNIF(16,20);
                IT:= 45;
            Ruta (R) Destino::
                STAY := 25;
            Destino (E)::
                FILE(REGISTRO,TIME:5:2:Vagones); ]
```

como entrada para la función *selecciona*(). Esta función reorganiza esta lista de forma tal que el nodo que esta siendo activado se ubica en la cabecera de la lista. Para dar inicio a la simulación se ejecuta la primera instrucción de activación `ACT(Partida,0)`, en este momento $Network = Leyes$.

Posteriormente, la función *reacción*() permite reducir las *Leyes* a un conjunto de instrucciones que cambian el componente estático del sistema σ y se programa los próximos eventos agregándolos a λ, por tanto cada entrada de la lista debe tener la forma:

$$\texttt{do(}componente\texttt{,}acción\texttt{,}t\texttt{)}$$

donde *componente* indica que componente intenta realizar la acción *acción* contiene información sobre la acción que se va a ejecutar y el momento en que debe ser ejecutada y t el tiempo de máquina en que se postula la acción.

Para nuestro ejemplo es posible obtener algo como:

```
λ'' = [ do(simulator,set(Vagones = UNIFI(16,20), now),t0),
        do(simulator,set(IT = 45, now), t0+1),
        do(simulator,ACT(Partida, now+IT), t0+2),
        do(simulator,SENDTO(Ruta, now), t0+3),
        do(simulator,ACT(Ruta, now+25), t0+4)),··· ]          (5.12)
```

Este nuevo conjunto de influencias indica que el simulador en `t0` programa la asignación de un valor aleatorio entero entre `16` y `20` proveniente de una distribución uniforme a la variable `Vagones` en el tiempo actual de simulación. De la misma forma

en `t0+1` programa la asignación `45` a la variable `IT`, en `t0+2` programa la activación del nodo `Partida` en el tiempo `now+IT` y así sucesivamente.

Este, es un registro declarativo de todo lo que hace el simulador y de hecho, todo lo que hacen todos los agentes en la simulación (el simulador contado entre ellos). Este registro independiente de implementación puede ser de mucha utilidad al verificar los modelos de simulación y al probar la correctitud de nuestro simulador multi-agentes.

De la misma forma, es posible introducir influencias provenientes de los agentes. En este caso *componente* contiene el nombre del agente que postula la influencia, mientras que *acción* la acción que desea postular y el tiempo en que debe ser ejecutada y como en el caso anterior t contiene el tiempo de máquina en que se postula dicha influencia.

Para introducir la semántica del nuevo lenguaje, en la siguiente sección mostraremos algunos ejemplos que incluyen agentes en el modelo del sistema.

5.5 Modeloteca

Los ejemplos mostrados en la sección 4.4, ilustran algunas características de las técnicas de simulación tradicional. Un modelista describe completamente la estructura definida para el sistema. Dicha estructura tiene una dinámica asociada que se lleva a cabo con la ejecución de cierta pieza de código predefinida. Generalmente, esta estructura permanece igual a lo largo de la simulación (hasta que el modelista cambie la descripción, es decir, el modelo).

Esta sección muestra el proceso de modelado y simulación de tres sistemas que incluyen agentes:

1. Sistema simple de tres taquillas.
2. Sistema de supermercado.
3. Sistema elevador.

5.5.1 Sistema simple de tres taquillas con agentes

Con GALATEA, estamos apuntando a otro nivel de descripción: aquel que tiene en cuenta a los agentes y, a través de ellos, permite cierto tipo de cambios en la estructura del sistema. El modelado de agentes y el cambio estructural son conceptos interesantes pero difíciles de discutir. Por tanto, preferimos argumentar ilustrando un ejemplo: una extensión del ejemplo sistema simple de tres taquillas (sección 4.4.1).

El ejemplo del código 12 ha sido modificado para incorporar un Gerente y para incluir a éste y a los Clientes cierta "racionalidad" (código 19). En esta modificación, los clientes no son entidades dependientes que se mueve en la red, en vez de ello los

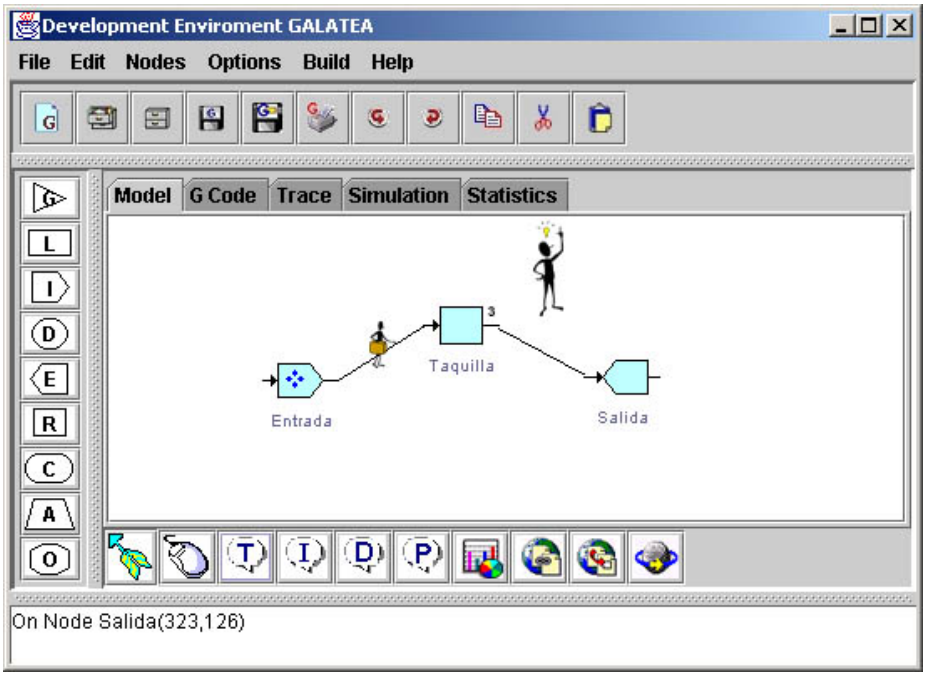

Figura 5.9: Modelo del sistema simple de tres taquillas con agentes

clientes pueden razonar basándose en las percepciones y su estado interior (específico para cada entidad representada) y entonces toman decisiones y actúan en el ambiente (en este caso, se realizan actos verbales muy elementales). El gerente, un agente implícito, también tiene racionalidad y capacidad de actuar y su influencia es mucho más transcendental que la de los clientes ya que puede cambiar la estructura real del sistema agregando o eliminando taquillas.

Al realizar un contraste con el modelo de taquillas original se pueden apreciar ciertos detalles. La sección `NETWORK` tiene simplemente un cambio: la instrucción `setAgent(Cliente)` le indica al simulador que la entidad que se genera en el nodo `Entrada` es un agente de tipo Cliente, cuyo estado interior (mental) incluye el código guía que se describe en la sección `AGENTS` (precisamente seguido de la etiqueta `Cliente::`) y los planes que dicho agente construya haciendo uso de este código. Ésta es la manera en que se transforman las entidades que cruzan la red (mensajes/clientes) en agentes. Esto implica que el sistema de simulación está asignando, para cada mensaje a ser tratado, un motor de inferencia que corre en paralelo con el simulador. El motor de inferencia se encarga del razonamiento del agente.

En GALATEA permitimos además una segunda manera, menos explícita, de introducir agentes en un modelo de la simulación: es posible declarar entradas en la

Código 19 Código GALATEA: Sistema simple de tres taquillas con agentes

```
TITLE
   Sistema simple de tres taquillas
NETWORK
   Entrada (I) ::
      setAgent(Cliente);
      IT:= 10;
      SENDTO(Taquilla[MIN]);
   Taquilla [1..nCaj] (R) ::
      STAY:= 45;
   Salida (E) ::
AGENTS
   Cliente (AG) ::
      GOALS
         revisa_cola and
         if cola_larga then (queja)
   Gerente (AG) Static ::
      GOALS
         revisa_cola and
         if cola_larga then (crear_taquilla) and
         if taq_vacias then (elim_taquilla)
INTERFACE
   revisa_cola() {
      if ((where.getName() == "Taquilla") && (where.el.ll() > 5))
         input.add("cola_larga", time);}
   queja() {
      System.out.println(name + ": Cola larga!!"); }
   crear_taquilla() {
      if (Taquilla.mult > Taquilla.maxMult)
         System.out.println("Banco lleno!!");
      else Taquilla.addInstance(); }
   taq_vacias() {
      for (int i = 0, j = 0; i < Taquilla.mult; i++) {
         if (Taquilla[i].il.ll() + Taquilla[i].el.ll() = 0) {
            j++;
            if (j > 1) input.add("taq_vacias", time); }}}
   elim_taquilla() {
      int i = 0;
      while ((i < Taquilla.mult  - 1) &
             (Taquilla[i].il.ll() + Taquilla[i].el.ll() > 0)) {i++;}
      if (Taquilla[i].il.ll() + Taquilla[i].el.ll() = 0)
         Taquilla[i].delInstance(); }
INIT
   TSIM:= 300;
   nCaj:= 3;
   ACT(Entrada, 0);
   INTI nCaj:3:0 :cantidad de cajeros;
DECL
   VAR nCaj: INTEGER;
   STATISTICS ALLNODES;
END.
```

sección `AGENTS` como "`static`" (como hacemos con el Gerente). Esto significa que tenemos un agente con motor de razonamiento que no se ata a ninguna entidad que se mueve en la red, pero que también tiene su propio estado interior, razona y puede cambiar el estado del mundo con sus acciones.

Por motivos de simplicidad, nosotros hemos presentado el ejemplo con un conjunto muy trivial de reglas: El agente de tipo Cliente es controlado por el precepto que la cola es demasiado larga, en cuyo caso el acto relacionado (queja) se activa y se ejecuta inmediatamente. De momento, hemos restringido las reglas al caso proposicional, pero en uno de los ejemplos siguiente haremos uso del poder expresivo de la lógica clausal.

Finalmente, la sección `INTERFACE` enlaza las percepciones y acciones del agente al ambiente. En nuestro ejemplo de taquillas, el agente de tipo Gerente es manejado por las percepciones de una cola larga o demasiadas colas vacías. En cada caso, el agente tiene una respuesta específica en forma de una acción: crear o eliminar una taquilla. Note que estas acciones, como se detallan en la `INTERFACE`, una vez que se ejecutan exitosamente cambian la estructura del modelo original (al agregar o eliminar taquillas).

5.5.2 Sistema de supermercado con agentes

El modelo GLIDER del código 16 ha sido modificado para asociar agentes a los mensajes que representan los vigilantes del supermercado, como muestra el modelo de la figura 5.10.

Nuestra intención con este ejemplo es mostrar como el agente además de controlar su ubicación y desplazamiento en la red puede modificar el contenido de los mensajes que transitan la red de nodos.

En la sección `NETWORK` del código 20 se pueden apreciar los cambios entre el código GALATEA y el código GLIDER asociado:

1. En el nodo `Vigila` se asocia a cada mensaje un agente del tipo `Vigilante`
2. En los nodo `Entrada` y `Pasillo` el uso la distribución uniforme es `UNIF()` tanto para valores reales como enteros, el tipo de dato a retornar lo determinan los argumentos con que se invoca.
3. La multiplicidad de los nodos `Pasillo` y `Caja` esta definida por las variables `nPas` y `nCaj` respectivamente. Ya que el nuevo simulador permite incorporar y eliminar instancias de nodos múltiples, no es necesario estimar la multiplicidad máxima.
4. Los nodos `Pasillo` y `Decide` no incluyen la revisión del pasillo que realiza el vigilante ni la decisión del pasillo que debe revisar. En este caso tanto el traslado del mensaje asociado al vigilante como

Código 20 Código GALATEA: Sistema de supermercado con agentes

```
TITLE Supermercado
NETWORK
  Vigila (I) Pasillo[1] ::
    Tipo:= Vig;
    setAgent(Vigilante);
  Entrada (I) Pasillo ::
    Tipo:= Cli;
    IF (TSIM-TIME>60) THEN IT:= EXPO(t0);
    ER:= 0; NAR:= 0;
    EC:= UNIF(0,LAC);  NAC:= 0;
    LP:= GT + MIN(tc*EC,TSIM-GT);
    IF (BER(p/100)) THEN BEGIN Tipo:=Lad; ER:= UNIF(1,LAR);END;
    k:= UNIFI(1,nPas); SENDTO(Pasillo[k]);
  Pasillo (R) [1..nPas]  Decide ::
    IF ((Tipo=Lad) AND (NAR<ER)) THEN BEGIN
      libre:= TRUE;
      SCAN(IL_Pasillo[INO])
        IF (O_Tipo=Vig) THEN BEGIN libre:= FALSE; STOPSCAN; END;
      IF libre THEN NAR:= NAR + UNIF(1,ER-NAR); END;
    IF  (NAC<EC) THEN NAC:= NAC+UNIF(0,EC-NAC);
    STAY:= GAUSS(45,10);
  Decide (G) Reten,Caja,Salida,Pasillo ::
    IF (Tipo=Det) THEN SENDTO(Reten)
    ELSE IF ((TIME>=LP) OR (NAR>ER) OR (NAC>EC)) THEN
      IF (NAC=0) THEN SENDTO(Salida) ELSE SENDTO(Caja[MIN])
    ELSE BEGIN k:= UNIF(1,nPas); SENDTO(Pasillo[k]); END;
  Caja (R) [1..nCaj] Salida ::  STAY:= tCaja[INO] * NAC;
  Reten (R) Salida :: STAY:= MIN(TSIM-TIME,tDet*NAR);
  Salida (E) ::
AGENT
   Vigilante (AG) static::
     GOALS cuida_pasillo and if (hora_cambio) then (cambia_pasillo)
INTERFACE
    cuida_pasillo(){
      il.fscan(){ if ((O_Tipo<>Vig) and (O_NAR > 0)){ O_Tipo=Det; }}}
    cambia_pasillo(){ k = UNIF(1,nPas); SENDTO(FIRST,Pasillo[k]); }
    hora_cambio(){ return (time = et+5); }
INIT
   TSIM:= 15*60;
   nVig:=3;    nPas:= 5;   nCaj:= 3;
   t0:= 5;     tDet:= 5;   tc:=1.5;    p:= UNIF(0.0,30.0);
   FOR k:= 1 to nCaj DO tCaja[k]:= UNIF(0.2,1.5);
   FOR k:= 1 to nPas DO Pasillo[k]:= UNIF(7,15);
   FOR k:= 1 to nVig DO ACT(Vigila,0);
   ACT(Entrada,0);
DECL
   CONST  Vig = 0; Cli = 1; Lad = 2; Det = 3; LAR = 15;  LAC = 50;
   VAR
      k, nVig, nPas, nCaj: INTEGER;
      t0, tDet, tc, p: REAL;
      libre: BOOLEAN;
      tCaja: ARRAY[1..nCaj] OF REAL;
   MESSAGES
      Entrada(Tipo,EC,NAC,ER,NAR: INTEGER; LP: REAL);
      Vigila(Tipo: INTEGER);
   STATISTICS ALLNODES;
END.
```

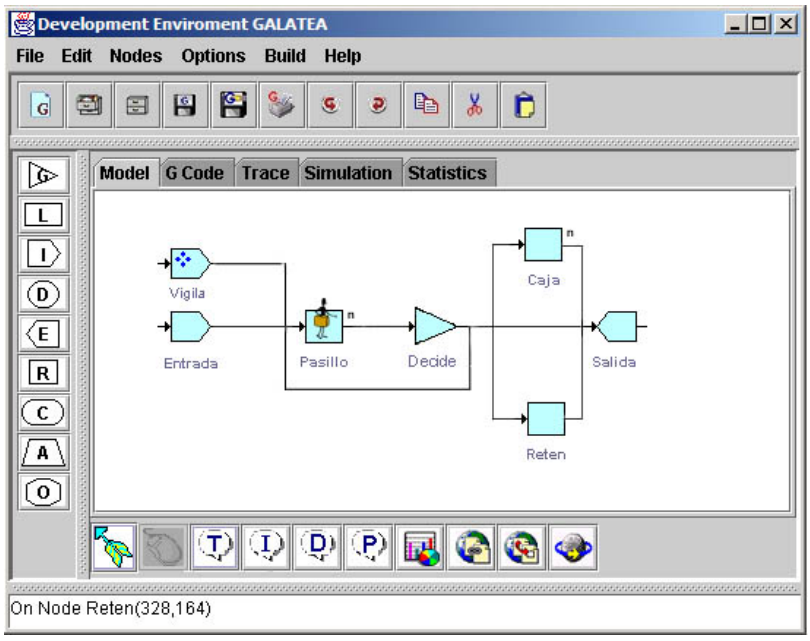

Figura 5.10: Modelo del sistema de supermercado con agentes

el comportamiento "racional" del mismo están controlados por el agente.

En la sección `AGENTE` mostramos de nuevo una descripción proposicional del código que guía el proceso de planificación de nuestro agente.

En la sección `INTERFACE` incluimos el código que permite guiar la percepción del agente. Acá se muestra como el "cuerpo" del agente (el mensaje que viaja en la red de nodos) permanece en el nodo `Pasillo` y además se dan las instrucciones para que el agente revise el pasillo en que se encuentra y en caso de encontrar clientes con artículos robados indique que el cliente debe ser detenido.

El resto de las secciones no presenta cambios.

5.5.3 Sistema elevador con agentes

Se desea simular en forma inteligente el comportamiento de un elevador mientras un grupo de personas esta haciendo uso del mismo. En este caso, las personas no son consideradas agentes, sino simplemente entidades que se desplazan sobre el edificio siguiendo algunas leyes probabilísticas. El agente inteligente en este ejemplo es el controlador del ascensor. Las consideraciones del proceso son:

* El edificio posee cinco pisos y para efectos del modelo, la única forma de acceso es a través del elevador.
* Las personas entran en el edificio según una ley estadística exponencial con media `TLleg`.
* En promedio las personas permanecen entre 5 y 20 minutos en un piso y luego deciden a que piso se dirigen.
* Una vez invocado, el elevador tratará de dirigirse hacia el piso donde fue llamado.

Para modelar el sistema descrito hacemos equivalencia entre una unidad de tiempo de simulación y un minuto. Además, asociamos las personas que llegan al edificio con mensajes y la estructura del edificio la asociamos con una red de cinco (5) nodos (subsistemas):

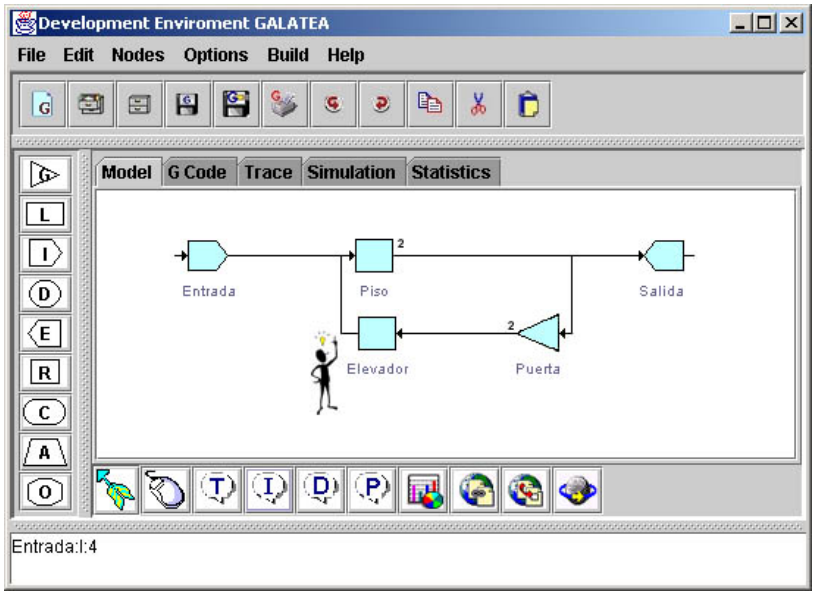

Figura 5.11: Modelo del sistema elevador con agentes

Entrada.

Representa la entrada al edificio. En este nodo se programan y registran los mensajes (personas) que llegan al sistema. Como muestra la figura los mensajes son enviados al nodo `Piso`. En este caso al piso 1.

Piso.

Nodo múltiple que representa los pisos del edificio. En este nodo se retiene al mensaje (persona) indicando la permanencia en minutos de cada persona en el piso según una distribución estadística uniforme entre 5 y 20.

La selección del destino de la persona una vez que ha cumplido su tiempo de permanencia en el piso se hace a través de una distribución uniforme que retorna un valor entero entre 0 y 5 donde 0 indica que la persona desea salir del edificio y un valor entre 1 y 5 indica el piso al que desea dirigirse.

Una vez establecido el destino de la persona, se envía el mensaje al nodo `Puerta` correspondiente al piso en que se encuentra. En caso de escoger salir del edificio y encontrarse en el piso 1 el mensaje es directamente enviado al nodo `Salida`

Salida.

Representa la salida del edificio. Este nodo registra los mensajes (personas) que salen del sistema.

Puerta.

Nodo múltiple que representa las puertas del ascensor. En este nodo se retienen los mensajes hasta que el elevador este disponible y luego los envía al nodo `Elevador`.

Elevador.

Representa al elevador. Retiene los mensajes (personas) hasta que pueda ubicarlos en el piso que seleccionaron como destino. Al ubicarse en un piso deja allí los mensajes que tienen como destino el piso en cuestión.

Para simular el proceso se sigue el modelo descrito, el cual se muestra en la figura 5.11. Este modelo puede ser escrito en GALATEA (ver código 21).

Como mencionamos anteriormente, la sección `NETWORK` incluye el conjunto de leyes que gobiernan el sistema. Cada entrada en la sección representa uno de los subsistemas y por ende le corresponde una ley que permite definir como se cambia el estado del mismo. En este ejemplo, a diferencia del ejemplo de tres taquillas esta sección no incluye instrucciones que asocien mensajes con agentes.

En la sección `AGENT` se define un único agente, denominado `Controlador`, que permite representar el comportamiento del simulador. En este caso, en la sección `GOALS` se muestra un conjunto de reglas condición-acción que representan metas condicionales que relacionan las entradas recibidas del ambiente con las acciones que el agente

Código 21 Código GALATEA: Sistema de elevador con agentes

```
TITLE
    Elevador. Esquema de un edificio
NETWORK
     Entrada (I)  Piso[1] ::
        IT:= EXPO(TLleg);
     Piso [1..nPisos] (R) Puerta[INO], Salida ::
        STAY:= UNIF(5,20);
        Destino:= UNIFI(0,nPisos);
        if ((INO = 1) AND (Destino = 0)) then SENTO(Salida);
     Puerta [1..nPisos] (G) Elevador ::
        if ((PosElevador = INO) AND Abierta AND (F_Elevador>0)) then
            SENTO(Elevador);
     Elevador (R) Piso[PosElevador]::
        STAY:= TSIM;
        IF (Destino = PosElevador) THEN SENDTO(Piso[PosElevador])
     Salida (E) ::
AGENTS
    Controlador (AGENT) static ::
    GOALS
       if (piso(M) at T) and (on(N) at T) then (
          if (N = M) then ((abrir; off; cerrar) at T) and
          if (N < M) then (abajo at T) and
          if (M > N) then (arriba at T)) and
       (verPisos at T) and
       (verBotones at T)
INTERFACE
// Se implementan los efectos de las acciones sobre el ambiente.
// Para reducir espacio se ha omitido el codigo asociado a los metodos
// piso(), on(), abrir(), off(), cerrar(), abajo(), arriba(),
// verPisos() y verBotones().
INIT
   TSIM:= 100;
   ACT(Entrada, 0);
   Tlleg:=5;
   Elevador:= 8;
DECL
     CONST
        nPisos = 5;
     VAR
        TLleg: REAL;
        PosElevador: INTEGER;
        Abierta: BOOLEAN;
     MESSAGES Entrada(SigPiso:INTEGER);
END.
```

debe ejecutar si las condiciones se cumplen. Note que esta acción puede ser múltiple, en cuyo caso se representa como una secuencia de acciones atómicas y reglas.

La sección `INTERFACE` contiene el código que describe los efectos de las acciones del agente sobre el sistema. En la sección `INIT` se configuran las condiciones iniciales del sistema: La simulación dura `100` unidades de tiempo, la primera persona entra al sistema al tiempo `0`. Se inicializa el tiempo entre llegadas a `5` y se inicializa la capacidad para el elevador en `8` personas.

En la sección `DECL` se indica la estructura de datos a utilizar para las variables globales, y el campo `sigPiso` asociado a los mensajes que contiene la información asociada al piso que desea visitar la persona. Además, se indica que el número de pisos estará almacenado en la constante `nPisos`, el uso de esta constante permite que posteriormente se pueda configurar escenarios donde el número de pisos sea diferente a `5`.

5.6 Detalles de Implementación

5.6.1 Requerimientos generales

La biblioteca de clases de la plataforma de simulación GALATEA debe ser:

* Completa. Debe incluir una familia de clases que presente al usuario alternativas para el modelado de sistemas.
* Adaptable. Evitar los aspectos específicos de la plataforma de ejecución, en caso de no poderse evitar es necesario identificarlos, documentarlos y aislarlos de forma que sea fácil realizar las sustituciones locales.
* Eficiente. Los componentes deben ser fáciles de ensamblar, debe imponerse la eficiencia en términos de recursos de compilación y de ejecución.
* Segura. No debe permitir que se violen la semántica dinámica de una clase. Las suposiciones estáticas sobre el comportamiento de una clase deben ser reforzadas al momento de la ejecución.
* Simple. La biblioteca debe usar una organización clara y consistente que facilite la identificación y selección de las clases apropiadas.
* Extensible. Los desarrolladores deben ser capaces de añadir nuevas clases independientemente, conservándose al mismo tiempo la integridad en la arquitectura de la plataforma.

Al igual que con el prototipo del simulador, cada uno de los módulos de la plataforma de simulación debe ser probados exhaustivamente siguiendo las pruebas de métodos, clases, integración y uso descritas en la sección 4.3.1

5.6.2 Ambiente de desarrollo

El ambiente de desarrollo para GALATEA debe contener ciertas cualidades:

* Apoyarse en una interfaz gráfica amigable.
* Facilitar las operaciones de construcción, codificación, simulación y análisis de modelos.
* Facilitar la integración de los usuarios GLIDER a la nueva plataforma.
* Proporcionar facilidades gráficas para el modelado de sistemas, para la presentación de resultados y para la animación del modelo.
* Facilitar la inclusión de nuevos módulos a la plataforma.
* Incorporar herramientas que faciliten el análisis de los resultados.

Hasta ahora en paralelo con el desarrollo del prototipo del simulador, se llevo a cabo el desarrollo de un prototipo para la interfaz gráfica [3], que se muestra en la figura 5.12, en la cual se identifican cinco porciones bien definidas:

Menú de Opciones. Permiten navegar a través de menús que despliegan acceso a las funcionalidades generales de todo ambiente de trabajo: manejo de archivo, edición, ayuda y las funcionalidades particulares del simulador.

Ambientes de Trabajo. Determina la forma de interactuar con el modelo su simulación, se presentan varias alternativas: Modelo gráfico, Código, Traza, Animación de la simulación y Estadísticas.

Area de Trabajo. Area donde se visualiza e interactúa con el modelo. Esta área cambia según el ambiente de trabajo.

Area de mensajes. Permite enviar mensajes al usuario que lo informen sobre las actividades que esta ejecutando sobre el modelo.

Menú de Barras. Contienen iconos que facilitan el acceso a las funcionalidades de la plataforma.

* Barra de herramientas. Corresponde a las funcionalidades generales de manejo de archivo y edición.

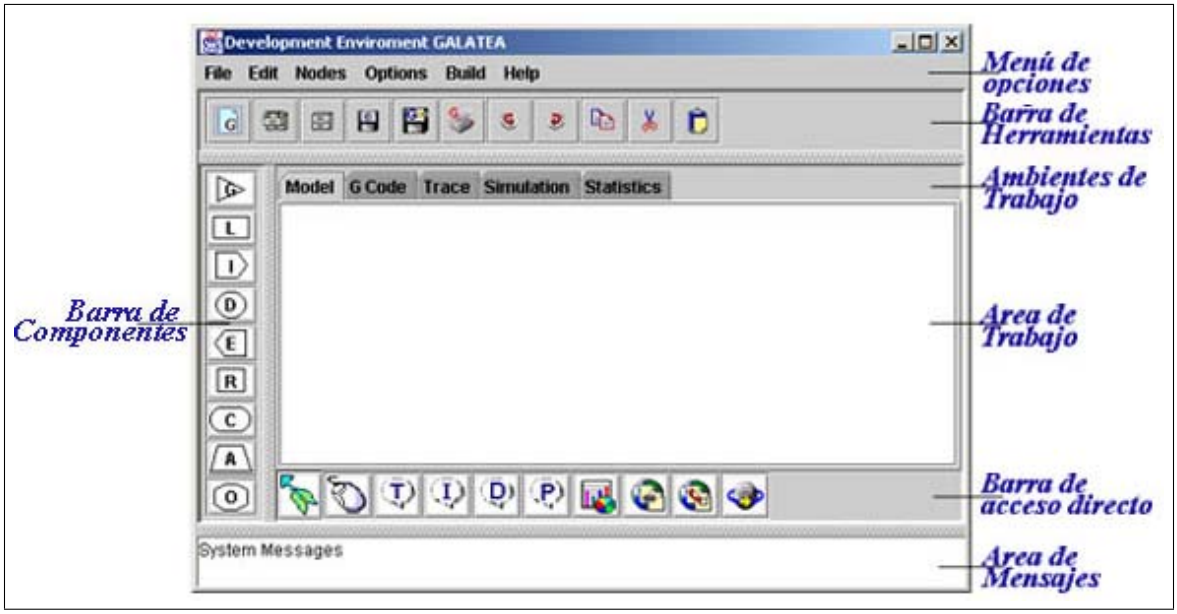

Figura 5.12: Ambiente de desarrollo para GALATEA

* Barra de componentes. Corresponde a los componentes predefinidos del sistema, en el caso del prototipo construido, permite incorporar nodos al modelo.
* Barra de acceso directo. Combina múltiples funcionalidades:
  - Permite incorporar conexiones y seleccionar objetos dentro del área de trabajo.
  - Permite incorporar la información correspondiente a las secciones `TITLE`, `INIT`, `DECL`, `PROCEDURES` al modelo.
  - Permite activar los procesos correspondientes a: visualización de resultados, compilación, ejecución y animación del modelo.

El desarrollo de este prototipo sirvió como escenario de las pruebas de correctitud del diseño del simulador presentado en el capítulo 4.1.2, sobre todo en la parte de integración entre los módulos.

5.6.3 Estructura de archivos

Conocer la manera en que están organizados los archivos permitirá a los usuarios y a los miembros del grupo de desarrollo de la plataforma entender la estructura sobre la que se soporta la misma.

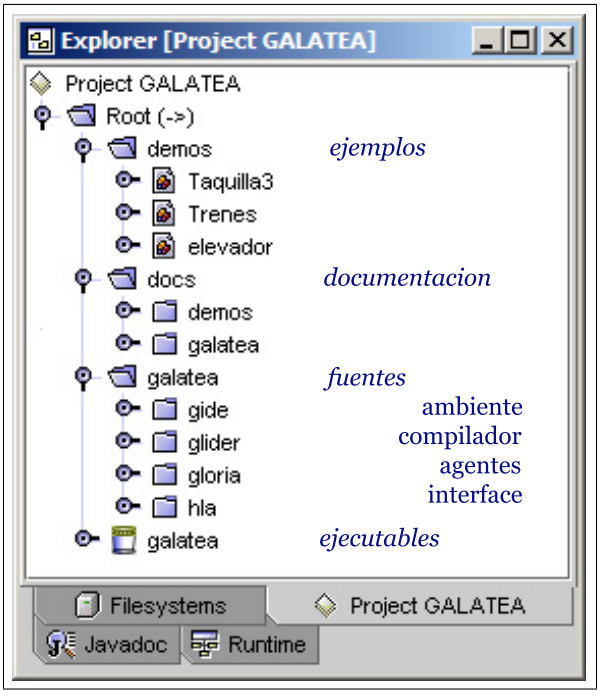

Figura 5.13: Estructura de archivos para GALATEA

Como muestra la figura 5.13, la estructura del sistema de archivos acordada para contener la plataforma GALATEA se define a partir del directorio base, **Root(->)**, allí se encuentran los archivos que conforman la plataforma:

* El subdirectorio demos debe contener algunos ejemplos, como los mostrados en las secciones 4.4 y 5.5, que servirán de guía a los usuarios en el uso de la plataforma.

* La documentación de las clases que conforman la plataforma y de los ejemplos se encuentra en los subdirectorios docs/galatea y docs/demos respectivamente. Esta documentación se generó con el documentador del lenguaje Java.

* Los archivos que contienen el código java, comúnmente denominados archivos fuente, se encuentran en el subdirectorio galatea. Este directorio es de acceso restringido a los miembros del grupo de desarrollo de GALATEA y ha sido subdividido con la finalidad de que cada módulo de la plataforma este perfectamente diferenciado:

    - galatea/gide, debe contener los fuentes correspondientes al modulo del ambiente de desarrollo.

- galatea/glider, debe contener los fuentes correspondientes al módulo simulador.
- galatea/gloria, debe contener los fuentes correspondientes al módulo de agentes.
- galatea/hla, debe contenerlos fuentes correspondientes al módulo de interfaz.

* Los archivos que contienen el conjunto de clases necesarias para la utilización de la plataforma, comúnmente denominados ejecutables entre los diseñadores de sistemas, se encuentran en archivo galatea.jar. Este archivo agrupa todas las herramientas necesarias, y su acceso debe ser sin restricciones.

Capítulo 6

El Compilador Galatea

Capítulo 7

Un ambiente de desarrollo para simulación con GALATEA

Referencias

[1] G. Birtwistle, *Discrete Event Modelling on Simula*. Macmillan Press, 1979.

[2] J. Vaucher, "The simula web site," 1998. [Online]. Available: http://www.jsp.umontreal.ca/~simula/

[3] M. Cabral, "Prototipo del módulo GUI para la plataforma de simulación galatea," Universidad de Los Andes. Mérida. Venezuela, 2001, tutor: Uzcátegui, Mayerlin.

[4] S. J. Russell and P.Ñorvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, Inc, 1995.

[5] O. García and R. Gutiérrez Osuna, "An update approach to complexity from and agent-centered artificial intelligence perspective," in *Encyclopedia of Library and Information Science.* Marcel Dekker. Inc, vol. 68, no. 31.

[6] C. Domingo, "GLIDER, a network oriented simulation language for continuous and discrete event simulation," in *International Conference on Mathematical Models*, Madras, India, August, 11-14 1988.

[7] N. R. Jennings, "An agent-based approach for building complex software systems," *Communications of the ACM*, vol. 44, no. 4, pp. 35–41, April 2001.

[8] J. A. Dávila and K. A. Tucci, "Towards a logic-based, multi-agent simulation theory," in *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000].* Mérida, Venezuela: AMSE & ULA, October, 22-24 2000, pp. 199–215. [Online]. Available: http://citeseer.nj.nec.com/451592.html

[9] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modelling and Simulation*, 2nd ed. Academic Press, 2000.

[10] B. P. Zeigler, *Theory of modelling and simulation*, ser. Interscience. New York: Jhon Wiley& Sons, 1976.

[11] H. Präehofer, "System theoretic formalisms for combined discrete-continuous system simulation," *International Journal of General Systems*, vol. 19, no. 3, pp. 219–240, 1991.

[12] NRC, *Thecnology for the United States Navy and Marine Corps, 2000-2035 Becoming a 21st-Century Force*, ser. Modeling and Simulation. National Academic Press, 1997, vol. 9.

[13] D. A. Fahrland, "Combined discrete event continuous systems simulation," *Simulation*, February 1970.

[14] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198–206, April 1981.

[15] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, March 1986.

[16] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, October 1990.

[17] B. P. Zeigler and H. Präehofer, "System theory challenges in the simulation of variable structure and intelligent systems," *In: CAST-Computer-Aided Systems Theory, Lecture Notes, Springer-Verlag*, pp. 41–51, 1989.

[18] A. M. Uhrmacher and B. P. Zeigler, "Variable structures in object-oriented modeling," *International Journal of General Systems*, vol. 24, no. 4, pp. 359–375, 1996.

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

[20] J. A. Dávila, "Agents in logic programming," Ph.D. dissertation, Imperial College of Science, Technology and Medicine, London, UK, June 1997.

[21] J. L. Austin, *How to do things with words.* Oxford Universitty Press, 1976.

[22] I. Futó and T. Gergely, *Artificial Inteligence in Simulation*, ser. Series in Artificial Intelligence. Ellis Horwood LTD, 1990.

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

[24] C. Domingo and G. Tonella, "Towards a theory of structural modeling," R. Scazzieri, Ed., vol. 10(3). Elsevier, 2000, pp. 1–18.

[25] A. M. Uhrmacher, "Concepts of object- and agent-oriented simulation," *Transactions of the Society of Computer Simulation*, vol. 14, no. 2, pp. 56–67, 1997.

[26] A. M. Uhrmacher and B. Schattenberg, "Agents in discrete event simulation," 1998.

[27] A. M. Uhrmacher, "Object-oriented, agent-oriented simulations: Implications for social science applications," *Social Science Micro Simulation - A Challenge for Computer Science. Springer Lecture Notes in Economics and Mathematical Systems*, 1996.

[28] N. R. Jennings and M. Wooldridge, "Agent-oriented software engineering," *Springer-Verlag Lecture Notes in AI*, vol. 1957, January 2001.

[29] Y. Shoham, "Agent-oriented programming," Stanford University, Stanford, CA 94305, Technical Report STAN-CS-1335-90, 1990.

[30] M. Wooldridge and P. Ciancarini, "Agent-oriented software engineering: The state of the art," *Springer-Verlag Lecture Notes in AI.*, vol. 1957, January 2001.

[31] F. Ibáñez-Cruz, R. Valencia-García, R. Martínez-Béjar, and F. Martín-Rubio, "A tool for comparing knowledge elicitation techniques (in spanish)," in *8th Conference of the Spanish Artificial Intelligence Association*, ser. Knowledge Engineering and Agents, Murcia, España, 1999.

[32] E. Pollitzer and J. A. Dávila, "Application of agent architecture to modelling human-computer interactions," in *ESSLLI'97 Symposium on Logical Approaches to Agent Modelling and Design*, Aix-in-Provence, France, 1997. [Online]. Available: http://citeseer.nj.nec.com/444483.html

[33] P. R. Cohen and H. J. Levesque, "Intention is choice with commitment," *Artificial Intelligence*, vol. 42, pp. 213–261, 1990.

[34] M. Wooldridge, "The logical modelling of computational multi-agent systems," Ph.D. dissertation, Department of Computation, Manchester Metropolitan University, Manchester, UK, October 1992.

[35] J. A. Dávila and M. Uzcátegui, "Agents' executable specifications," in *Logic for Programming, Aritificial Intelligence and Reasoning [LPAR 2002].* Poster Section: Kurt Goedel Society, October 14-18 2002, to be published.

[36] R. Oirfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[37] J. Martin and J. J. Odell, *Object-Oriented Methods: A Foundations*. Prentice Hall Inc, 1996.

[38] G. Booch and J. Rumbaugh, *Unified Method for Object Oriented Development*. Rational Software Corp, 1996.

[39] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc, 1997.

[40] J. Ferber and J.-P. Müller, "Influences and reaction: a model of situated multi-agent systems," in *ICMAS-96*, 1996, pp. 72–79.

[41] R. A. Kowalski and F. Sadri, "Towards a unified agent architecture that combine rationality with reactivity," in *LID'96 Workshop on Logic in Databases*, D. Pedreschi and C. Zaniolo, Eds., San Miniato, Italy, July 1996. [Online]. Available: http://www-lp.doc.ic.ac.uk/UserPages/staff/rak.html

[42] I. Mathworks, *SIMULINK, User's Guide*. Prentice Hall, Inc, 1996.

[43] G. Booch, *Object-Oriented Analysis and Design with Applications*, second edition ed. Addison-Wesley, Inc, 1996.

[44] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice-Hall, Inc, 1991.

[45] C. Domingo, "Proyecto GLIDER. e-122-92. informe 1992-1995," CDCHT, Universidad de Los Andes. Mérida. Venezuela, Tech. Rep., December 1995.

[46] C. Domingo, G. Tonella, H. Herbert, M. Hernández, M. Sananes, and J. Silva, "Use of object oriented programming ideas in a new simulation language," in *Summer Computer Simulation Conference*, Boston, USA, July, 19-22 1993, pp. 137–142.

[47] G. Tonella, C. Domingo, M. Sananes, and K. Tucci, "El lenguaje GLIDER y la computación orientada hacia objeto," in *XLIII Convención Anual ASOVAC*. Mérida, Venezuela: Acta Científica Venezolana, November, 14-19 1993.

[48] K. A. Tucci, "Prototipo del compilador GLIDER en C++," Universidad de Los Andes. Mérida. Venezuela, 1993, tutor: Tonella, Giorgio.

[49] O. Terán, "Simulación de cambios estructurales y análisis de escenarios," Master's thesis, Maestría en Estadística Aplicada, Universidad de Los Andes. Mérida. Venezuela, 1994.

[50] F. J. Palm, "Simulación combinada discreta/continua orientada a objetos: Diseño para un lenguaje GLIDER orientado a objetos," Master's thesis, Maestría en Matemática Aplicada a la Ingeniería, Universidad de Los Andes. Mérida. Venezuela, 1999.

[51] Sun Microsystems Inc, "The source for the java technology," Since 1994. [Online]. Available: http://java.sun.com

[52] C. Domingo, G. Tonella, and M. Sananes, *GLIDER Reference Manual*, 1st ed., CESIMO–IEAC.Universidad de Los Andes, Mérida, Venezuela, August 1996, cESIMO IT-9608.

[53] B. P. Zeigler, "Creating simulations in HLA/RTI using the DEVS modelling framework," in *SIW'99*, ser. Tutorial, Orlando, FL, March 1999.

[54] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, June 1995.

[55] Moore, *Logic and Representation.* 333 Ravenswood Avenue, Menlo Park, CA 94025: Center for the Study of Language and Information (CSLI), 1995, iSBN 1-881526-15-1.

[56] R. A. Kowalski, "Using metalogic to reconcile reactive with rational agents," in *Meta-Logics and Logic Programming*, K. Apt and F. Turini, Eds. MIT Press, 1995. [Online]. Available: http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/recon-abst.html

[57] T. H. Fung and R. A. Kowalski, "The iff proof procedure for abductive logic programming," *Journal of Logic Programming*, July 1997.

[58] M. R. Genesereth and N.Ñilsson, *Logical foundations of Artificial Intelligence.* California. USA: Morgan Kauffman Pub., 1988.

[59] J. A. Dávila, "Openlog: A logic programming language based on abduction," in *PPDP'99*, ser. Lecture Notes in Computer Science. 1702. Paris, France: Springer, 1999. [Online]. Available: http://citeseer.nj.nec.com/64163.html

[60] J. A. Dávila and M. Uzcátegui, "Galatea: A multi-agent simulation platform," in *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000].* Mérida, Venezuela: AMSE & ULA, October, 22-24 2000, pp. 217–233. [Online]. Available: http://citeseer.nj.nec.com/451467.html

[61] B. P. Zeigler, *Objects and Systems. Principled Design with Implementations in C++ and Java*. Springer-Verlag, Inc, 1997.

[62] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, Inc, 1998.

[63] R. Lutz, "A comparison of HLA object modeling principles with traditional object-oriented modeling concepts," in *SIW'97*, ser. Spring, 1997.

[64] G. Tetuci, *Building Intelligent Agents. An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies*. San Diego, California: Academic Press, 1998.

[65] G. Weiss, Ed., *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. Cambridge, Massachusetts: MIT Press, 1999.

[66] J. Josephson and S. Josephson, *Abductive Inference*. Cambridge, Massachusetts: Cambridge University Press, 1994.

[67] D. Flanagan, *Java in a Nutshell. A Desktop Quick Reference for Java Programmers*. O'Reilly & Associates, Inc, 1996.

Apéndice A

Applet Montecarlo

Apéndice B

Clases del Simulador GALATEA

La figura B.1 muestra el diagrama de clases del prototipo de simulador para la plataforma GALATEA. A continuación mostraremos los detalles de las clases que integran el prototipo, las cuales fueron escritas siguiendo los detalles de diseño e implementación mostrados en el capítulo 4.

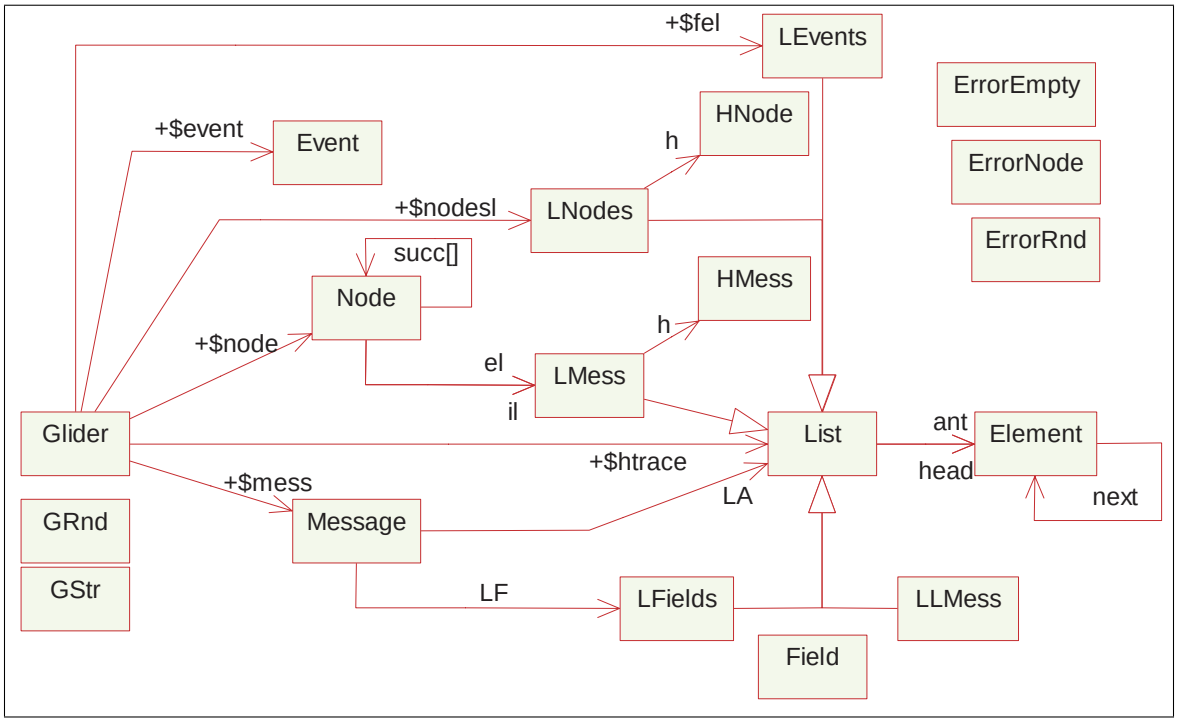

Figura B.1: Diagrama general de clases del simulador

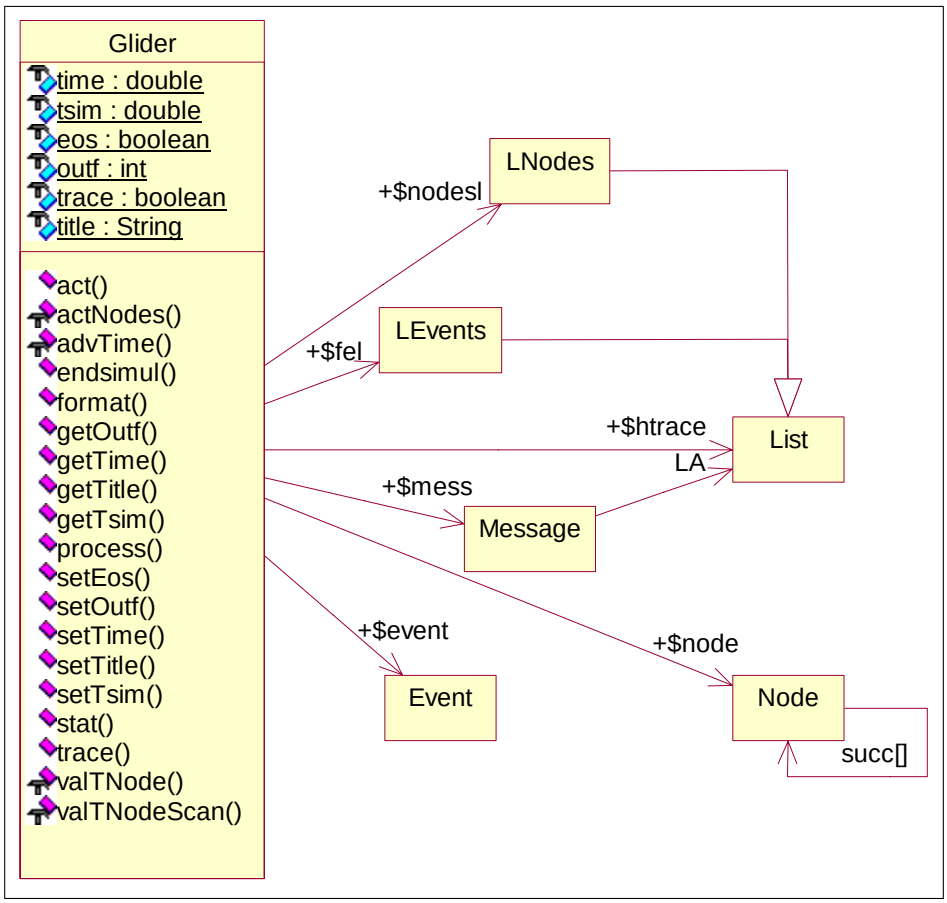

Figura B.2: Diagrama de clases para el simulador

B.1 Clase Glider

Clase básica para el simulador.

Esta clase contiene las variables de ambiente del simulador e implementa los métodos generales del algoritmo básico del comportamiento del simulador.

El diagrama de la figura B.2 muestra los detalles de la clase `Glider`.

Atributos públicos

Event event	Evento actual
LEvents fel	Lista de eventos futuros
List htrace	Registro de la lista de eventos futuros
Message mess	Mensaje actual
Node node	Nodo actual
LNodes nodesl	Lista de nodos de la red

Atributos privados

boolean eos	Indica el fin de la simulación

int outf	Indica el número máximo de decimales para el formato de salida
boolean trace	Indica si se realiza registro de traza
double time	Tiempo actual de simulación
String title	nombre del modelo
double tsim	Tiempo máximo de simulación

Métodos públicos

void act(Node n, double ta)	Método para activar un nodo
void endsimul()	Indica el fin de la simulación
String format()	Da formato a un número real
int getOutf()	Obtiene el número máximo de decimales de la salida
double getTime()	Obtiene el tiempo actual de simulación
double getTsim()	Obtiene el tiempo total de simulación
String getTitle()	Obtiene el nombre del modelo
void process()	Procesa la lista de eventos futuros
void setEos(boolean end)	Configura el fin de simulación
void setOutf(int f)	Configura el número máximo de decimales de la salida
void setTime(double t)	Configura el tiempo de simulación
void setTitle(String n)	Configura el nombre del modelo
void setTsim(double t)	Configura el tiempo máximo de simulación
void stat()	Registrar las estadísticas de la simulación
void stat(String fname)	Registrar las estadísticas de la simulación
void trace()	Registrar la traza de la simulación
void trace(String fname)	Registrar la traza de la simulación

Métodos privados

boolean actNodes()	Activa la lista de nodos
void advTime()	Avanza el tiempo de simulación
boolean valTNode(char type)	Determina si es un tipo de nodo válido
boolean valTNodeScan(char type)	Determina si es nodo a recorrer

B.2 Clase Element

Clase para el manejo de Elementos.

Esta clase implementa los atributos y métodos para un elemento de lista. El elemento es el componente básico para las listas del simulador. Este componente, mostrado en el diagrama de la figura B.3, permite que cualquier objeto pueda incorporarse como elemento de una lista.

Atributos privados

Object dat	Dato almacenado
Element next	Apuntador al siguiente elemento

Constructores

Element ()	Crea un elemento nulo que se apunta a sí mismo

Element (Object dat)	Crea un elemento a partir del dato dado que se apunta a sí mismo
Element (Object dat, Element next)	Crea un elemento a partir del dato dado que apunta al elemento dado

Métodos públicos

String toString ()	Visualización del objeto Element

Métodos privados

Object getDat ()	Retorna el dato
Element getNext ()	Retorna el siguiente elemento
boolean leq (Element e)	Indica si existe un orden parcial de elementos
void setDat (Object dat)	Asigna un dato al elemento
void setElement (Object dat, Element next)	Configura el elemento
void setNext (Element next)	Asigna el apuntador al siguiente elemento

B.3 Clase List

Clase para el manejo de Listas.

El diagrama de la figura B.3 incluye la clase `List`. Esta clase hereda los atributos y métodos básicos de la clase `List` e implementa los atributos y métodos para las listas de l simulador. Las listas utilizadas por el simulador son circulares y poseen enlaces simples entre sus elementos.

En este caso, las rutinas de manipulación de los elementos respetan el orden de la lista (*NONE*, *FIFO*, *LIFO*, *ASCEN DESCEN*).

Atributos privados

Element ant	Apuntador al elemento anterior
Element head	Apuntador a la cabecera
String name	Nombre de la lista
int num	Número de elementos de la lista
byte order	Tipo de orden de la lista
int pa	Posición del elemento anterior en la lista
int pos	Posición del elemento actual de la lista

Constructores

List()	Crea una lista FIFO sin cabecera
List(byte order)	Crea una lista ordenada sin cabecera
List(Object dhead)	Crea una lista FIFO con cabecera
List(Object dhead, byte order)	Crea una lista ordenada con cabecera
List(Object dhead, String name)	Crea una lista FIFO con nombre y cabecera
List(Object dhead, String name, byte order)	Crea una lista ordenada con nombre y cabecera
List(String name)	Crea una lista FIFO con nombre sin cabecera
List(String name, byte order)	Crea una lista ordenada con nombre y sin cabecera

Métodos públicos

void add(List l)	Agrega los elementos de la lista dada
void add(Object dat)	Agrega un elemento con el dato indicado
boolean empty()	Indica si la lista esta vacía
boolean equals(Object o)	Indica si dos listas son equivalentes
Element getAnt(Object dat)	Retorna el elemento anterior al dato indicado
Object getDat()	Retorna el dato del elemento actual
Object getDat(int pos)	Retorna el dato según una posición dada
boolean getDat(Object dat)	Indica si el dato existe
Object getHead()	Retorna el dato cabecera
String getName()	Retorna el nombre de la lista
Element getNext(Object dat)	Retorna el elemento siguiente al dato indicado
byte getOrder()	Retorna el tipo de orden
int getPos()	Retorna el la posición del elemento actual
int getPos(Object dat)	Retorna la posición del dato indicado
int ll()	Retorna la longitud
Object remove()	Extrae un elemento
boolean remove(Object dat)	Extrae el elemento del dato dado
boolean setPos(int pos)	Avanza a la posición dada
boolean setPos(Object dat)	Avanza a la posición del dato dado
String stat()	Retorna las estadísticas sobre la lista
String toString()	Visualización del objeto List

Métodos privados

Object extract()	Extrae el elemento actual
void insert(Object dat)	Agrega un elemento con el dato indicado en la posición actual
void insert(Object dat, int pos)	Agrega un elemento con el dato indicado en la posición indicada
boolean next()	Avanza a la posición siguiente
void setDat(Object dat)	Configura el dato del elemento actual
void setDat(Object dat, Object dnew)	Configura el dato del elemento dado
void setHead(dat)	Configura la cabecera de lista
void setList(Object dat, String name)	Configura la lista
void setName(String name)	Asigna el nombre a la lista
boolean valPos()	Indica si la posición actual es válida
boolean valPos(int pos)	Indica si la posición dada es válida

B.4 Clase Event

Clase para el manejo de eventos.

Esta clase, mostrada en el diagrama de la figura B.4, implementa los atributos y métodos que definen un evento. Además, incorpora un orden entre los eventos, el cual viene dado por el valor del campo `ta` que representa el tiempo de activación del evento.

Atributos privados

Object comp	Componente asociado al evento
double ta	Tiempo de activación del evento

Constructor

Event(Node n, double ta)	Crea un nuevo evento

Métodos públicos

setEvent(Node n, double ta)	Configura un evento
String toString()	Visualización del objeto Event

Métodos privados

boolean equals(Object o)	Indica si existe un orden parcial de eventos
Node getNode()	Retorna el Nodo asociado al evento
double getTa()	Retorna el tiempo de activación asociado al evento

B.5 Clase LEvents

Extiende la clase List.
Clase para el manejo de lista de eventos.

Esta clase hereda los atributos y métodos básicos de lista e implementa los atributos y métodos, mostrados en el diagrama de la figura B.4 para la lista de eventos. Corresponde a una lista ordenada en forma ascendente.

Constructores

LEvents()	Crea una nueva lista de eventos
LEvents(String name)	Crea una nueva lista con nombre

Métodos públicos

void add(Node n, double ta)	Agrega un nuevo evento
boolean remove(Node n, double ta)	Elimina un evento

B.6 Clase Node

Clase para el manejo de nodos.

Esta clase, incluida en el diagrama de la figura B.5, implementa los atributos y métodos para nodos. En particular implementa los métodos generales para activación y revisión de los nodos predefinidos de la red (G,L,I,D,E,R,C,A,O), y define los respectivos métodos que pueden ser re-escritos por el modelista para incluir detalles del modelo en cada nodo particular.

Atributos privados

double cap	Capacidad del nodo
LMess el	Lista Externa de mensajes
LMess il	Lista Interna de mensajes
int ino	Indice del nodo
String name	Nombre del nodo
int nmess	Número de mensajes generados
Node[] succ	Nodos sucesores
char type	Tipo del nodo
double use	Cantidad de recursos utilizados

Constructores

Node(name, char type)	Crea un nodo no múltiple cuyo sucesor es el nodo siguiente
Node(String name, char type, Node succ)	Crea un nodo no múltiple con sucesor único
Node(String name, char type, Node[] succ)	Crea un nodo no múltiple con sucesores
Node(String name, int ino, char type)	Crea un nodo cuyo sucesor es el nodo siguiente
Node(String name, int ino, char type, Node succ)	Crea un nodo con sucesor único
Node(String name, int ino, char type, Node[] succ)	Crea un nodo

Métodos públicos

boolean act()	Activación del nodo
void act(double ta)	Permite programar la activación del nodo
Message create()	Asocia un nuevo mensaje al nodo
boolean fact()	Método de activación genérico de un nodo
boolean fscan()	Método de recorrido genérico de un nodo
List getEl()	Retorna la lista externa del nodo
List getIl()	Retorna la lista interna del nodo
int getIno()	Retorna el índice del nodo
String getName()	Retorna el nombre del nodo
char getType()	Retorna el tipo del nodo
void it(double it)	Programa la próxima activación del nodo respecto al tiempo actual
void nt(double nt)	Programa la próxima activación del nodo
void sendto(Message m)	Envía un mensaje al nodo sucesor
void sendto(Message m, LMess l)	Envía un mensaje a una lista
void sendto(Message m, Node n)	Envía un mensaje a un nodo

void sendto(Message m, Node[] n)	Envía un mensaje a cada instancia del nodo múltiple
void sendto(Message m, Node[] n, int ino)	Envía un mensaje a alguna instancia del nodo múltiple
void setCap(double cap)	Asigna la capacidad al nodo
String stat()	Retorna las estadísticas sobre el nodo
void stay(double xt)	Programa la permanencia del mensaje actual en el nodo
void stay(Message m, double xt)	Programa la permanencia de un mensaje en el nodo
String toString()	Visualización del objeto Node

Métodos privados

boolean scan()	Recorrido del nodo
void setIno(int ino)	Asigna el índice del nodo
void setName(String name)	Asigna el nombre del nodo
void setNode(String name, int ino, char type)	Configura el nodo
void setType(char type)	Asigna el tipo del nodo

B.7 Clase HNode

Clase para el manejo de cabeceras de lista de nodos.

Esta clase, incluida en el diagrama de la figura B.5, implementa los atributos y métodos necesarios para manipular la cabecera de la lista de nodos. Esta cabecera define un orden parcial entre los elementos que serán incluidos en la lista: los primeros elementos corresponden a aquellos nodos que sólo se recorren, los elementos siguientes se recorren y se activan y por último se encuentran los nodos que sólo se activan. Este orden facilita los procesos de activación y de recorrido de la red.

Atributos privados

int fa	Posición del primer nodo que se activa
int ls	Posición del último nodo que se recorre

Constructores

HNode()	Crea una cabecera de lista de nodo
HNode(int fa, int ls)	Crea una cabecera de lista de nodo dadas las posiciones de activación y recorrido

Métodos públicos

String toString()	Visualización del objeto HNode

Métodos privados

int getFa()	Retorna el primer nodo a activar
int getLs()	Retorna el último nodo a recorrer
void setFa(int fa)	Asigna la posición del primer nodo a activar
void setHNode(int fa, int ls)	Configura las posiciones de activación y recorrido
void setLs(int ls)	Asigna la posición del último nodo a recorrer

B.8 Clase LNodes

Extiende la clase List.
Clase para el manejo de lista de nodos.

Esta clase, incluida en el diagrama de la figura B.5, hereda de la clase `List` los atributos y métodos básicos e implementa los atributos y métodos necesarios para manipular la lista de nodos. Estos métodos, respetan el orden parcial de la misma, definido en la cabecera. Por ser una lista con cabecera, además implementa los métodos para acceder a la información almacenada en la cabecera.

Atributos privados	
HNode h	Cabecera de lista
Constructores	
LNodes() LNodes(String name)	Crea una lista de nodos Crea una lista de nodos con nombre
Métodos públicos	
void add(Node n)	Agrega un nodo a la lista.
Métodos privados	
int getFa() int getLs() void insert(Node n) void setFa(int fa) void setLs(int ls)	Retorna la posición del primer nodo en ser activado Retorna la posición del último nodo en ser recorrido Agrega un nodo a la lista Configura la posición del primer nodo en activarse Configura la posición del último nodo en recorrerse

B.9 Clase Field

Clase para el manejo de campos.

Esta clase, mostrada en el diagrama de la figura B.6, implementa los atributos y métodos para manipular campos de los mensajes.

Atributos privados	
String name Object value	Nombre del campo Valor asociado al campo
Constructores	
Field(String name, Object value)	Crea un campo

Métodos públicos	
boolean equals(Object o)	Comparación de objetos Field
String getName()	Retorna el nombre del campo
Object getValue()	Retorna el valor del campo
String toString()	Visualización del objeto Field
Métodos privados	
void setField(String name, Object value)	Configura un campo
void setValue(Object value)	Asigna un valor al campo

B.10 Clase LFields

Extiende la clase List.
Clase para el manejo de listas de campos.

Esta clase hereda de la clase lista los atributos y métodos básicos e implementa los atributos y métodos necesarios para manipular la lista de campos y los campos que la conforman que se muestran en el diagrama de la figura B.6.

Constructores	
LFields()	Crea una lista de campos
LFields(String name)	Crea una lista de campos con el nombre dado
Métodos públicos	
Object getValue(String name)	Retorna el valor asociado a un campo
Métodos privados	
void add(String name, Object value)	Agrega un campo
void insert(Object dat)	Agrega un dato
void setField(String name, Object value)	Configura un campo

B.11 Clase Message

Clase para el manejo de mensajes.

Esta clase, mostrada en el diagrama de la figura B.6, implementa los atributos y métodos necesarios para el manejo de mensajes en el simulador.

Atributos privados	
double et	Tiempo de entrada a la lista actual
double gt	Tiempo de generación del mensaje

List LA	Lista de campos ensamblados del mensaje
LFields LF	Lista de campos del mensaje
String name	Nombre del mensaje
int number	Número asociado al mensaje
Object where	Ubicación del mensaje
double use	Cantidad de recurso que consume el mensaje
double xt	Tiempo de salida de la lista de mensajes actual

Constructores

Message(String name, int number)	Crea un mensaje y le asocia el número dado
Message(String name, int number, LFields lfields)	Crea un mensaje, le asocia el número y la lista de campos dada

Métodos públicos

void addAssemble(List l)	Agrega una lista a la lista de ensamblados
void addAssemble(Message m)	Agrega un mensaje a la lista de ensamblados
void addField(List l)	Agrega una lista a la lista de campos
void addField(String name, int value)	Agrega un campo entero al mensaje
void addField(String name, Object value)	Agrega un campo al mensaje
Message copy()	Crea una copia del mensaje
double getEt()	Retorna el tiempo de entrada a la lista actual
double getGt()	Retorna el tiempo de generación del mensaje
String getName()	Retorna el nombre del mensaje
int getNumber()	Retorna el número asociado al mensaje
double getUse()	Retorna la cantidad de recursos consumidos por el mensaje
Object getValue(String name)	Retorna el valor asociado a un campo del mensaje
Object getWhere()	Retorna la ubicación del mensaje
double getXt()	Retorna el tiempo de salida de la lista actual
void setField(String name, Object value)	Configura un campo al mensaje
String toString()	Visualización del objeto Message

Métodos privados

void addField(Field f)	Agrega un campo al mensaje
void setEt(double et)	Asigna el tiempo de entrada a la lista actual
void setMessage(String name, int number, LFields lfields)	Configura un mensaje
void setUse(double use)	Asigna la cantidad de recurso a utilizar
void setWhere(Object w)	Asigna la ubicación del mensaje
void setXt(double xt)	Asigna el tiempo de salida de la lista actual

B.12 Clase HMess

Clase para el manejo de cabeceras de la lista de mensajes.

Esta clase, mostrada en el diagrama de la figura B.6, implementa los atributos y

métodos necesarios para el manejo de la cabecera de lista de mensajes. Esta cabecera almacena la información que permite obtener estadísticas de las listas de mensajes.

Atributos privados

int maxl	Longitud máxima de la lista
double maxt	Tiempo de máximo de permanencia en la lista
int ne	Número de entradas a la lista
int nl	Número de muestras para la longitud de la lista
double nt	Número de muestras para el tiempo de permanencia en la lista
double suml	Acumula longitudes de la lista
double sumll	Acumula longitudes de la lista
double sumt	Acumula tiempos de permanencia en la lista
double sumtt	Acumula tiempos de permanencia en la lista
double te	Tiempo de entrada a la lista
double tf	Tiempo que la lista ha permanecido vacía
double tm	Tiempo de la última modificación

Constructores

HMess()	Crea una cabecera para de lista de mensajes

Métodos públicos

String toString()	Visualización del objeto HMess

Métodos privados

void statLong(int l)	Calcula estadísticas sobre la longitud de la lista
void statTime(double t)	Calcula estadísticas sobre el tiempo de permanencia en la lista

B.13 Clase LMess

Extiende la clase List.
Clase para el manejo de Listas de mensajes.

Esta clase, incluida en el diagrama de la figura B.6, hereda de la clase `List` los atributos y métodos básicos e implementa los atributos y métodos necesarios para manipular la lista de nodos y acceder a la información almacenada en su cabecera.

Atributos privados

HMess h	Cabecera de la lista de mensajes

Constructores

LMess()	Crea una lista FIFO de mensajes
LMess(byte order)	Crea una lista ordenada de mensajes
LMess(String name)	Crea una lista FIFO de mensajes con nombre
LMess(String name, byte order)	Crea una lista de mensajes ordenada con nombre

Métodos públicos

double dmedl()	Retorna la desviación de la longitud
double dmstl()	Retorna la desviación del tiempo de espera
void fscan()	Método de recorrido genérico de una lista
int entr()	Retorna el número de entradas
int maxl()	Retorna la longitud máxima
double medl()	Retorna el promedio de longitud
double mstl()	Retorna el promedio de tiempo de espera
void scan()	Permite recorrer la lista
String stat()	Retorna las estadísticas sobre la lista
double tfree()	Retorna el tiempo libre

Métodos privados

void add(Message m)	Agrega un mensaje
Object extract()	Extrae el mensaje actual

B.14 Clase LLMess

Extiende la clase List.
Clase para el manejo de Listas de listas de mensajes.

Esta clase, incluida en el diagrama de la figura B.6, hereda de la clase `List` los atributos y métodos básicos e implementa los atributos y métodos necesarios para manipular la lista de listas de mensajes y para la información almacenada en sus cabecera.

Constructores

LLMess()	Crea una lista de listas de mensajes
LMess(String name)	Crea una lista de listas de mensajes con nombre

Métodos públicos

String toString()	Visualización del objeto LLMess

Métodos privados

void add(Message m, Node n)	Agrega un mensaje a la lista asociada
void add(Node n)	Agrega una lista asociada a un nodo predecesor
add(Node[] pred)	Agrega una lista asociada por cada nodo predecesor
void add(Node[] pred, byte order)	Agrega una lista asociada por cada nodo
void add(Node n, byte order)	Agrega una lista asociada a un nodo predecesor
Object extract(Node n)	Extrae el mensaje actual de la lista asociada
LMess getList(Node n)	Retorna la lista asociada al nodo dado
String stat()	Método GLIDER que proporciona las estadísticas sobre la lista

B.15 Clase GRnd

Clase para el manejo de distribuciones estadísticas

Esta clase, mostrada en el diagrama de la figura B.7, implementa los atributos y métodos que permiten el uso de distribuciones estadísticas comúnmente usadas en simulación: Bernoulli, Beta, Erlang, Exponencial, Gamma, Gaussiana, LogNormal, Normal, Poisson, Triangular, Uniforme, Weibull.

Atributos privados

int[] rn	Almacén de semillas

Métodos públicos

boolean ber(double p)	Distribución de Bernoulli para semilla 1
boolean ber(double p, int s)	Distribución de Bernoulli
double beta(double a, double b)	Distribución Beta para semilla 1
double beta(double a, double b, int s)	Distribución Beta
double erlg(double u, long k)	Distribución de Erlang para semilla 1
double erlg(double u, long k, int s)	Distribución de Erlang
double expo(double u)	Distribución exponencial para semilla 1
double expo(double u, int s)	Distribución exponencial
double gamma(double u, double d)	Distribución Gamma para semilla 1
double gamma(double u, double d, int s)	Distribución Gamma
double gauss(double u, double d)	Distribución normal positiva para semilla 1
double gauss(double u, double d, int s)	Distribución normal positiva
void inisem()	Inicializador del semillero
double lognorm(double u, double f)	Distribución LogNormal para semilla 1
double lognorm(double u, double f, int s)	Distribución LogNormal
double norm(double u, double d)	Distribución normal para semilla 1 Metodo de la relación (Ripley: Stocasting Simulation)
double norm(double u, double d, int s)	Distribución normal
int poisson(double u)	Distribución de Poisson para semilla 1
int poisson(double u, int s)	Distribución de Poisson
double rnd(int s)	Generador de números aleatorios
double tria(double a, double b, double c)	Distribución triangular para semilla 1
double tria(double a, double b, double c, int s)	Distribución triangular
double unif(double a, double b)	Distribución uniforme sobre valores punto flotante para semilla 1
double unif(double a, double b, int s)	Distribución uniforme sobre valores punto flotante
int unif(int a, int b)	Distribución uniforme sobre valores enteros para semilla 1
int unif(int a, int b, int s)	Distribución uniforme sobre valores enteros
double weibull(double e, double f)	Distribución de Weibull para semilla 1

double weibull(double e, double f, int s)	Distribución de Weibull.

B.16 Clase GStr

Clase para el manejo de archivos.

El diagrama de la figura B.7 muestra los atributos y métodos que permiten la manipulación de archivos en el simulador.

Atributos públicos

PrintStream fstat	Archivo de estadísticas de la simulación
PrintStream ftrace	Archivo de traza de la simulación

Métodos públicos

void add(PrintStream file, Object o)	Método para agregar información a un archivo de salida
void close(PrintStream file)	Método para cerrar un archivo de salida
PrintStream open(String fname)	Método para configurar un nuevo archivo de salida

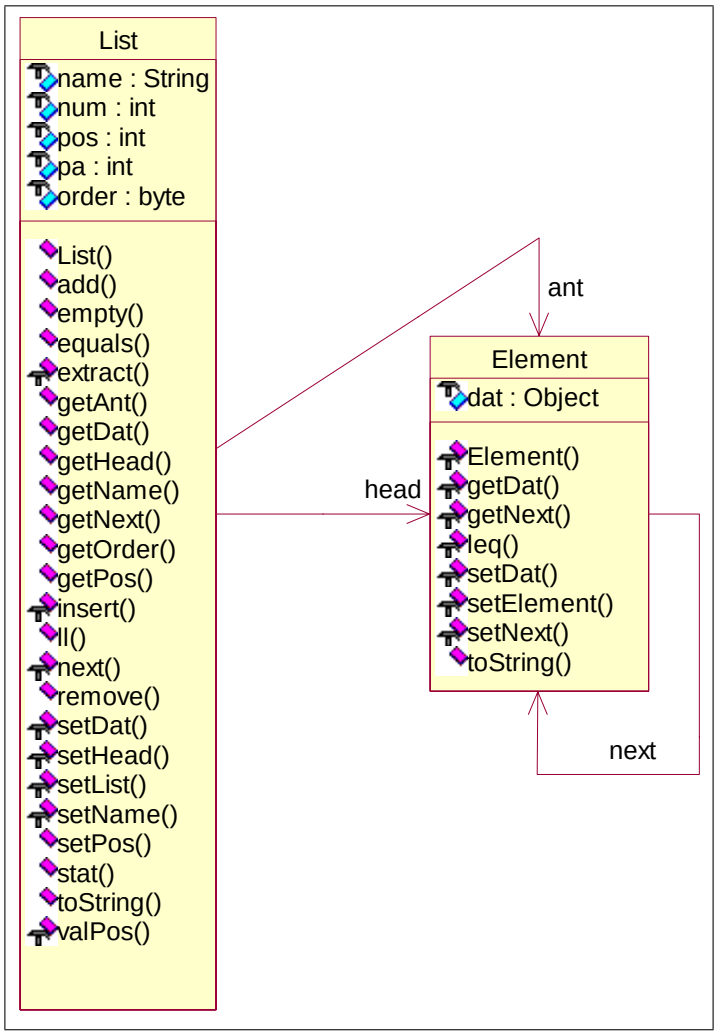

Figura B.3: Diagrama de clases para listas

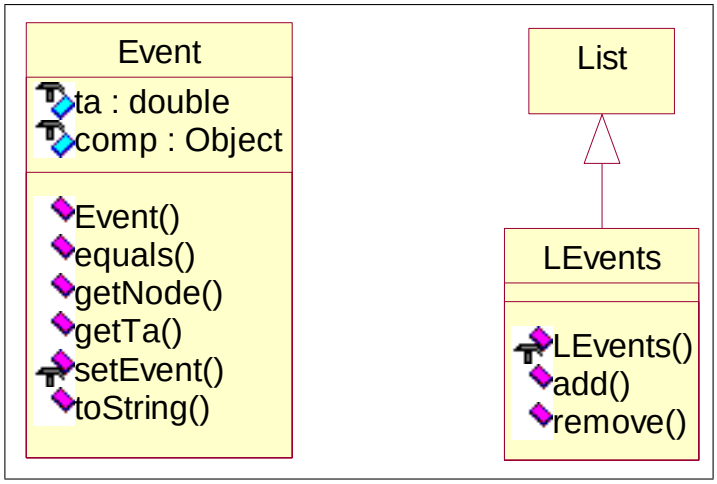

Figura B.4: Diagrama de clases para eventos

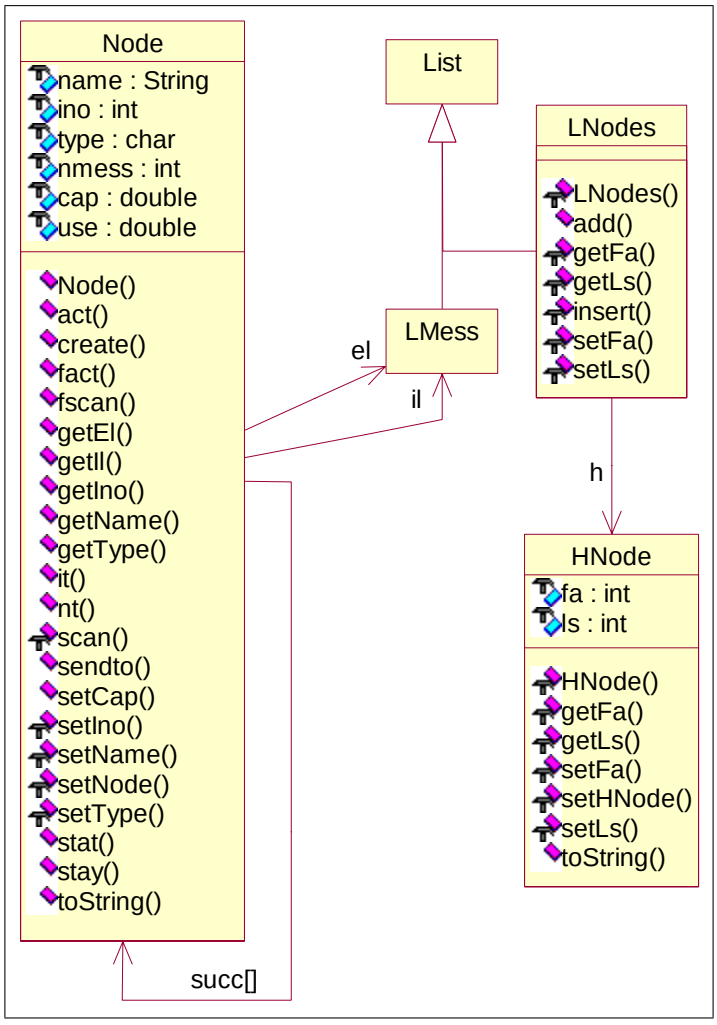

Figura B.5: Diagrama de clases para nodos

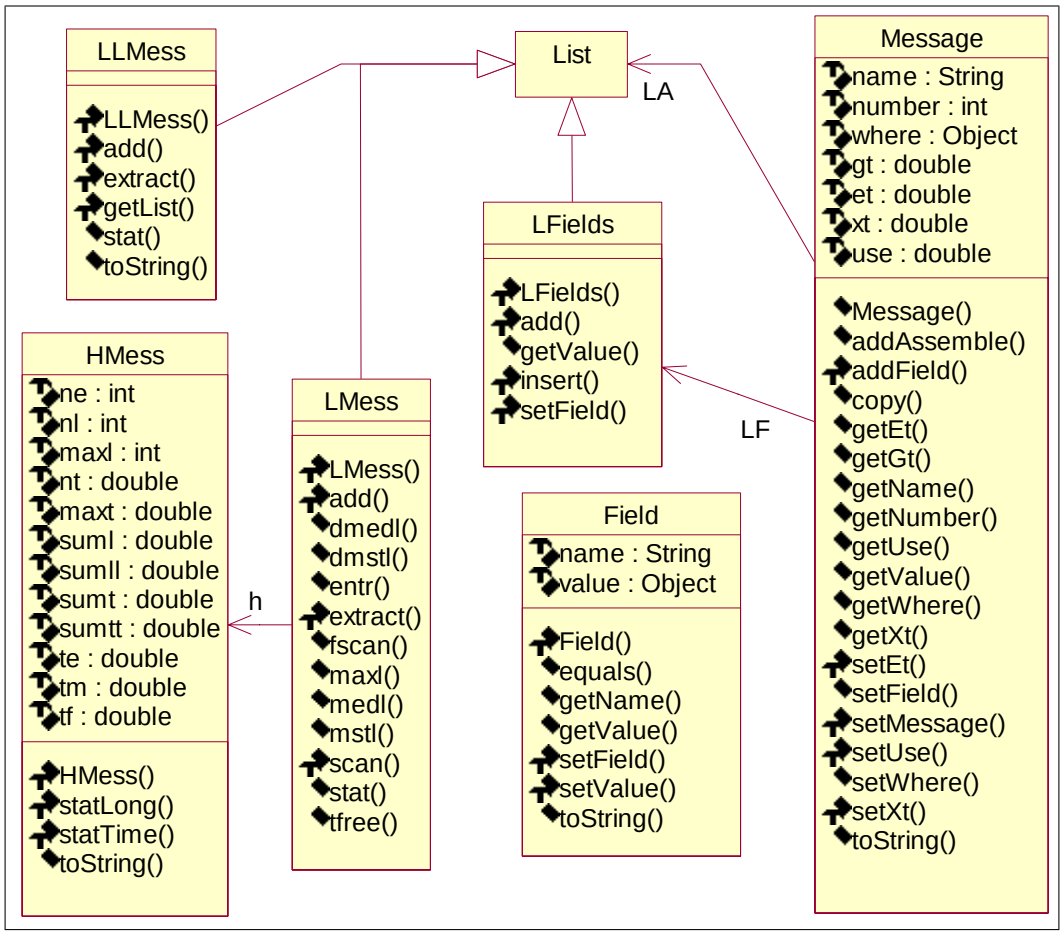

Figura B.6: Diagrama de clases para mensajes

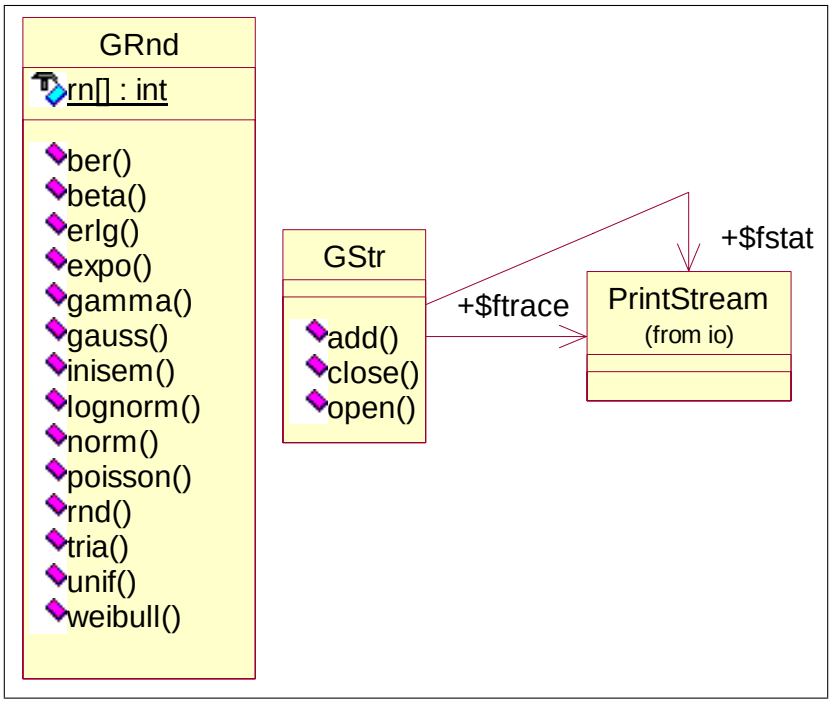

Figura B.7: Diagrama de clases para métodos generales

Glosario

Agente (*Agent*)
: Entidad que puede percibir su ambiente, asimilar dichas percepciones y almacenarlas en un dispositivo de memoria, razonar sobre la información almacenada en dicho dispositivo y adoptar creencias, metas e intenciones por sí mismo dedicándose al alcance de dichas intenciones a través del control apropiado de sus efectores.

Ambiente (*Environment*)
: Normalmente le llamamos así a el entorno en donde se desenvuelven los agentes. En simulación se procura construir un modelo de ese ambiente y de su dinámica de cambio. En ambiente también puede verse como un agente más, con ciertos privilegios de acción independiente.

API (*Application Programming Interface*)
: Interfaz para Programación de Aplicaciones.

Aprendizaje abductivo
: En el proceso de aprendizaje se adoptan hipótesis que explican una observación.

Arquitectura
: Una metodología particular para construir agentes. Generalmente este término se utiliza para denotar la estructura de datos, algoritmos y flujo de control de los datos que usa el agente al momento de decidir la acción a ejecutar. La arquitectura por tanto describe como las metas, creencias y preferencias de un agente son representadas, actualizadas, y procesadas para determinar las acciones del agente.

Autonomía

Generalmente el término se aplica a control intrínseco. Específicamente, se asume que el agente tiene control sobre su estado interno y sobre las acciones que ejecuta.

Base de Conocimientos
: Una colección de información básica, que contiene reglas y heurísticas específicas a cierta área del conocimiento.

Creencias (*Beliefs*)
: Un concepto que describe la información que el agente posee del ambiente, que no necesariamente es cierta. Entre las creencias se encuentran aquellos estados del ambiente entre los cuales el agente no puede discriminar.

DESS (*Differential Equation Systems Specifications*)
: Especificación de Sistemas con Ecuaciones Diferenciales.

DEVS (*Discrete Event Systems Specifications*)
: Especificación de Sistemas de Eventos Discretos.

DoD (*Departament of Defense*)
: Departamento de Defensa de los Estados Unidos.

DTSS (*Discrete Time Systems Specifications*)
: Especificación de Sistemas con Ecuaciones en Diferencias.

Federación
: Grupo de federados que conforman una comunidad. Estos federados intercambian información en forma de objetos y mensajes.

Federado
: Componente de una federación. Existen varios tipos de federados: modelo de simulación, colector de datos, simulador, agentes autónomos o espectadores.

FOM (*Federation Object Model*)
: Modelo de Objetos para Federación. Define las posibles interacciones entre los objetos que conforman una federación.

GALATEA (***G****LIDER with* ***A****utonomous,* ***L****ogic-based* ***A****gents,* ***TE****mporal reasoning and* ***A****bduction*)
: Plataforma de simulación cuyo diseño y prototipo funcional se describen en este documento.

GLIDER

plataforma de simulación que permite la especificación de modelos de simulación para sistemas continuos y de eventos discretos.

GLORIA (***G****eneral-purpose,* ***L****ogic-based,* ***O****pen,* ***R****eactive and* ***I****ntelligent* ***A****gent*)
Descripción lógica de un agente de propósito general reactivo e inteligente.

HLA (*High Level Architecture*)
Arquitectura de Alto Nivel. Marco de un referencia estándar para el soporte de simulaciones interactivas desarrollado por el DoD.

Influencias
Representan las acciones propias del agente, con las que intenta modificar el curso de los eventos que ocurrirán.

MARS (*Multi Agent Rational System*)
Sistema Multi-Agentes compuesto por agentes racionales.

MAS (*Multi Agent System*)
Sistema compuesto de múltiples agentes que interactúan entre ellos y con su ambiente.

Metas (*Goals*)
Un concepto que describe la predisposición del agente en ver realizadas ciertos deseos. Las metas deben ser mutuamente consistentes.

Modelado
Proceso de construir modelos que permiten describir un sistema.

OMT (*Object Model Template*)
Patrón del Modelo de Objetos.

OOD (*Object Oriented Design*)
Diseño Orientado a Objetos.

Planificación
El proceso por medio del cual un agente plantea la solución a un problema. Normalmente equivale a derivar los pasos y subtareas que deberán cumplir uno o varios agentes, para alcanzar la consecusión de una tarea mayor o un estado deseado.

Planificación abductiva
El uso del razonamiento hipotético y la abducción para generar planes dirigidos a alcanzar ciertas metas pre-establecidas.

Preferencias (*Preferences*)
: Un conjunto de heurísticas que sesgan las decisiones de un agente en dirección a ciertos conjuntos de acciones. Le permiten al agente incorporar criterios de *urgencia* e *importancia* en el proceso de planificación.

RTI (*RunTime Infrastructure*)
: Infraestructura de Tiempo de Ejecución.

Semántica
: Las estructuras que otorgan significado a un lenguaje.

Simulación
: Proceso en el que haciendo uso del computador se generan y manipulan las trayectorias que describen paso a paso el sistema, es decir, de un instante de tiempo al siguiente.

Simulación HLA
: Sesión en la que participan un conjunto de federados, en la que cada una de las entidades que la conforman se asocia a un objeto.

SMA (*MAS*)
: Sistema Multi-Agentes.

SMAR (*MARS*)
: Sistema Multi-Agentes Racional.

SOM (*Simulation Object Model*)
: Modelo de Objetos para Simulación. Define las aptitudes o capacidades de un federado para ser compartido entre varias federaciones.

UML (*Unified Modeling Language*)
: Lenguaje Unificado para Modelado que permite especificar, construir, visualizar y documentar los componentes de un sistema de software orientado a objetos.

Índice de Materias

Abducción, 157
Agentes
  definición, 66
    sistemas multi-agentes, 66
  desarrollo de software, 69
  estado del ambiente, 73
  GALATEA, 150
  influencias, 73
  inteligentes, 69
  jerarquía de Genesereth-Nilsson, 74
  jerarquía Ferber-Müller extendida por Dávila-Tucci, 76
  planificación, 157, 166
  reactivo-racional, 77
    ejemplo, 168, 171, 173
    GALATEA, 150
    sistema multi-agentes, 80
  simulación orientada a, 66
  teoria de simulación de, 71

GALATEA
  agentes, 150
  ambiente de desarrollo, 178
  interfaz, 143
  lenguajes, *véase* Lenguajes
  plataforma de simulación, 140
  semántica operacional, 164
  simulador, *véase* Simulador
  teoria de simulación SMA, 71
GLIDER
  entorno de desarrollo, 90
  lenguaje de simulación, 83
  plataforma de simulación, 83
  simulador, 86, *véase* Simulador

HLA, 62
  componentes, 63
    interfaz, 63
    RTI, 63
    simulaciones, 62
  federación, 62
  federados, 62
  interfaz, 64
  manejo del tiempo, 65
  modelo de objetos, 65
  objetivo, 62
  reglas, 66

Lenguajes
  GALATEA, 162
  GLIDER, 83

Modelado, 46, 48
  arquitectura, 58
  ciclo de vida, 59
  de agentes, 68, 70
  DEVS, 53, 58
  infraestructura, 58
  motivos, 48, 50

Plataforma de simulación
  GALATEA, 140
  GLIDER, 83

Simulación, 46, 48
  arquitectura, 58

ciclo de vida, 59
DEVS, 58
enfoques, 59
cambio estructural, 61
combinada, 60
continua, 60
distribuida, 61
eventos discretos, 59
interactiva, 61
paralela, 61
HLA, *véase* HLA
infraestructura, 58
motivos, 48
orientada a agentes, 66
teoría SMA, 71
Simulador
clases, 191
Element, 193
Event, 195
Field, 200
Glider, 191
GRnd, 204
GStr, 206
HMess, 202
HNode, 199
LEvents, 197
LFields, 201
List, 194
LLMess, 204
LMess, 203
LNodes, 200
Message, 201
Node, 198
DESS, 50
DEVS, 52, 53, 55
acoplado, 57
básico, 56
DTSS, 51
GALATEA
algoritmo general, 144
componentes, 97
diseño, 90
modeloteca, 168
GLIDER, 86
algoritmo, 89
estadísticas, 123
modeloteca, 110
traza, 124