Revista Ciencia e Ingeniería. Edición Especial: "Jornada de Modelado y Simulación" pp. 107-116. ISSN 1316-7081

Desarrollo de modelos de tráfico en GALATEA.
Development traffic models in GALATEA.

Tucci, Kay[1,2,*] **y Pérez, Carlos**[1]
[1]Laboratorio SUMA, Departamento de Física,
Facultad de Ciencias;
[2]Centro de Simulación y Modelos,
Facultad de Ingeniería,
Universidad de Los Andes. Mérida 5101, Venezuela
[*]kay@ula.ve

Resumen

A pesar de que lo vivimos a diario, el tráfico vehicular es un fenómeno que no está todavía totalmente comprendido. Hay muchas dificultades para su estudio, pero hoy en día ya tenemos suficiente información estadística y capacidad de cálculo para comenzar a desarrollar modelos realistas y detallados de tráfico vehicular. En este trabajo implementamos algunos modelos de tráfico, que se pueden utilizar como prototipo, para extender el sistema de modelos de demostración de la plataforma de simulación Galatea, desarrollada en el Centro de Simulación y Modelos de Universidad de Los los Andes. Se implementaron tres tipos de modelos microscópicos de tráfico vehicular: con automatas celulares, con mapas acoplados y con los sistemas de ecuaciones diferenciales ordinarias; para cada uno de ellos se estudiaron tres situaciones diferentes: una situación base de un carril unidireccional, la incorporación de vehículos al carril y un semáforo en el carril.

Palabras Clave: Tráfico, simulación, autómata celular, mapas acoplados, EDOs.

Abstract

Despite that we live everyday, the traffic of vehicles is a not yet compleatly understood phenomenon. There are many difficulties for its study, but nowadays we already have enought statistical information and capacity of calculation to begin to develop realistic and detailed models of vehicles traffic. In this work some models of traffic are implemented, the ones that can be used as prototype, in order to extend the set of demoinstration models of the simulation platform Galatea, developed in Centro de Simulación y Modelos of Universidad de Los Andes. Three types of microscopic models of traffic of vehicles were implemented: with cellular automatas, with coupled maps and with systems of ordinary differential equations; for each of them three different settings are studied: a base setting of a unidirectional lane, the incorporation of vehicles to the lane and a traffic light on the lane.

Keywords: Traffic, simulation, cellular automata, coupled maps, ODEs.

1 Introducción

Una de las cualidades que se ha observado durante la evolución de las sociedades ha sido el desarrollo de la capacidad de movilidad, hecho que ha incidido directamente en el aumento de la calidad de vida de sus integrantes. Pero con el incremento poblacional en el mundo comenzaron las dificultades de movilidad que, a su vez, mermaron la libertad, produciendo efectos nocivos en el normal desenvolvimiento de las actividades socio-económicas. Para estudiar y comprender este problema se han desarrollado modelado que describen algunos fenómenos de tráfico como lo son las ondas de arranque-frenado en una congestión, o la evolución de un sistema en atascos.

Las descripciones matemáticas de la dinámica de flujo de tráfico a nivel científico comienza en los años treinta del siglo pasado (Greenshields 1935), donde se estudiaron las relaciones fundamentales entre flujo, velocidad y densidad de tráfico. Posteriormente en los años cincuenta se describió la física de propagación de los flujos de tráfico por medio de los modelos de dinámica macroscópica y microscópica (Lighthill y Whitham 1955; Richards 1956), pero no fue sino hasta los años noventa que el número de científicos en esta área de investigación comienza a hacerse importante. Actualmente, con el incremento de la cantidad de información estadística del tráfico y del poder de cómputo, es posible simular modelos que preserven muchos de los detalles del tráfico real (Helbing et al. 2009). Mientras que los modelos macroscópicos hacen énfasis en cantidades como flujos y densidades de vehículos con

escalas de tiempo que van desde los minutos a los años (Newell 1961; Prigogine et al. 1979), los modelos microscópicos individualizan a cada vehículo tomando en cuenta su posición, velocidad y aceleración con escalas de tiempo que van desde las fracciones de segundo a las horas (Chopard et al. 1998; Nagel et al. 1992).

Con la finalidad de estudiar la capacidad y expresividad de la plataforma de simulación `GALATEA` (GALATEA 2010), en este trabajo se muestran tres modelos distintos de simulación de tráfico microscópico: autómatas celulares, mapas acoplados y sitemas de ecuaciones diferenciales ordinarias; mediante un modelo base de un carril unidireccional y dos variantes de éste: la incorporación de vehículos al carril y el funcionamiento de un semáforo en el carril. `GALATEA` es una familia de lenguajes, compiladores e interpretadores, que contiene distintas herramientas de simulación entre las que se tiene un integrador de ecuaciones diferenciales ordinarias, un simulador `DEVS` (Zeigler et al. 2000), y una colección de bibliotecas de apoyo y plantillas genéricas para el desarrollo de modelos de simulación de eventos discretos, sistemas continuos y sistemas multi-agentes que permite explorar alternativas para la integración con un formalismo general de modelado y simulación (Uzcategui et al. 2007). Las simulaciones con `GALATEA` se implementan a través de nodos; en un sistema de modelado un nodo es un conjunto de sub-sistemas que intercambian mensajes, que en nuestro caso serán los vehículos. Cada subsistema puede almacenar, transformar, transmitir, crear y eliminar mensajes del estado del sistema, a través de un código enlazado. Existen distintos tipos de nodos según su función, entre los que tenemos: los nodos tipo **`Gate`** que controlan el flujo de mensajes, los tipo **`Input`** que generan mensajes de entrada, los tipo **`Resource`** que simulan recursos usados por los mensajes, los nodos **`Exit`** que son los nodos que destruyen los mensajes, los nodos **`Autonomous`** y este permite programar eventos (Dávila et al. 2004).

Mediante el uso de nodos junto con el paquete `gSpace` (Laffaille 2005) se construye un modelo de simulación capaz de reproducir eventos discretos con estados continuos. El `gSpace` recrea el espacio, que en nuestro caso es el carril, con sus cualidades físicas y ubica en él a los mensajes. La herramienta permite programar una función de movimiento (`Regla`) para la interacción de los mensajes que se encuentran en el o los distintos espacios. Los espacios se definen mediante el uso de límites impenetrables (`Walls`) y límites penetrables (`Doors`), que son modificables en tiempo de ejecución.

2 Modelo base

Un carril unidireccional, como el de la Fig. 1, puede ser representado en la plataforma de simulación `GALATEA` mediante el diagrama de la Fig. 2 donde se observa un

Fig. 1. Ejemplo del sistema base a modelar.

nodo tipo **`Input`** encargado de generar la entrada de vehículos al sistema, un nodo tipo **`Exit`** encargado de eliminar los vehículos que salen del sistema y un `GSpase`, de color gris, que representa el carril por donde circulan los vehículos. Este último está compuesto por un nodo tipo **`Autonomous`**, un nodo tipo **`Gate`** y un nodo tipo **`Resource`**, llamados `move`, `wait` y `door` respectivamente.

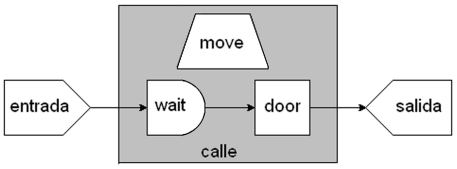

Fig. 2. Diagrama del modelo base en Galatea.

Es intuitivo imaginar el modo como se mueven los vehículos en una carretera de un solo carril como la que se muestra en la Fig. 1; por ejemplo, los vehículos no se adelantan pues físicamente es imposible. A partir de ésta y otras ideas se construye el diseño que sustenta nuestros modelos. En general, los vehículos en la carretera se comportan de la siguiente forma:

- Una vez que los vehículos se incorporan en la vía conservaran el orden en que entran a la carretera hasta el momento de la salida, el vehículo i-esimo siempre será el siguiente del $(i-1)$-esimo.
- Los vehículos ocupan un área determinada y, por tanto, una posición en el espacio sólo puede ser ocupada por un vehículo.
- Los vehículos deben tener aceleración, velocidad, y capacidad de frenado finita.
- En los modelos no habrá estudio de colisiones ni se hará simulaciones que las incluyan.
- La carretera es unidireccional, por lo que todos los vehículos se movilizan de izquierda a derecha.
- Los vehículos guardan una distancia mínima s_i^* entre sí que no dependerá directamente del modelo implementando.
- La posición, velocidad y aceleración de cada uno de los vehículos son individuales, es decir, el vehículo i tendrá posición x_i, velocidad v_i y aceleración a_i.

Los modelos base entán compuestos por cuatro clases. La primera de ellas es la clase principal `Via` cuyo código se muestran en la Fig. 3. Allí se instancia la clase `Nodo` para crear los nodos `entrada` y `salida` que generan y eliminan respectivamente a los mensa-

```
import galatea.glider.*;
import galatea.gspaces.*;
public class Via {
 static double l;
 static Entrada entrada=new Entrada();
 static Space calle=new Space("calle",
                   0,6.0,6.0,1.0, 1.0,"Regla");
 static Salida salida = new Salida();
 public static void main(String[] args){
   setExperiment(args);
   calle.addWall(0.0, 0.0, l, 0.0);
   calle.addWall(0.0, 12.0, l, 12.0);
   calle.addWall(0.0, 0.0, 0.0, 12.0);
   calle.addWall(l, 0.0, l, 12.0);
   calle.addDoor(l, 0.0, l, 12.0,
                salida,'U',1,0.0,0.0);
   calle.build();
   Glider.trace("Via.trc");
   Glider.stat("Via.sta");
   Glider.setTsim(10000);
   Glider.act(entrada, 0);
   Glider.act(calle.getMove(), 1);
   Glider.process();
 }
 static void setExperiment(String[] args){
   args = Galatea.checkArgs(args);
   l = Galatea.doubleArg(args, "Longitud", "01000.0");
   Glider.addPath(Galatea.getExperDir());
 }
}
```

Fig. 3. Código Java de la clase `Via`

jes (vehículos) del sistema. También, se instancia a la clase `Space` para crear el espacio, `calle`, por donde circulan los mensajes. La clase tiene dos métodos `main` y `setExperiment`. El primero es el método principal, que se invoca al ejecutar la clase y es el encargado de: llamar al método `setExperiment`, que es el que procesa los parámetros con los que se correrá el modelo. Luego, el método `main` establece los límites de `calle` e inicia la simulación asignando, entre otras cosas, el tiempo de simulación, el nombre de los archivos de salida y las primeras activaciones de nodos.

La siguiente clase es `Entrada` que extiende a la clase `Nodo` de Galatea del tipo **I**. Como se muestra en la Fig. 4, ésta clase implementa al constructor y al método `fact` que programa su próxima activación, asignan los valores iniciales a los campos del los mensajes (vehículos) y por último, envía al mensaje creado al espacio `calle`. Entre los campos tenemos a las posiciones en cada eje `x` e `y`, la velocidad deseada `v0`, y las velocidades iniciales en cada eje `vvx` y `vy`.

La tercera clase es `Salida` que también extiende a `Nodo` de `GALATEA` y cuyo tipo es **E**. Esta clase, como se muestra en la Fig. 5, se implementa su constructor y el método `fscan`, que de ser necesario puede procesar al mensaje saliente.

La última de las cuatro clases es `Regla`. En ella se implementan las reglas de movimiento de cada modelo.

```
import galatea.glider.*;
import galatea.gspaces.*;
class Entrada extends Node {
    Entrada() {
      super("entrada", 'I');
      Glider.nodesl|.add(this);
    }
    public void fact() {
      it(GRnd.unif(1, 4));
      Glider.mess.addField("x", 0.1);
      Glider.mess.addField("y", Via.calle.getYIn());
      Glider.mess.addField("v0", 12);
      Glider.mess.addField("tU",Glider.mess.
                              getNumber()%2);
      Glider.mess.addField("p", null);
      Glider.mess.addField("c", 0);
      Glider.mess.addField("s", 0);
      Glider.mess.addField("d", null);
        Glider.mess.addField("cut", null);
        Glider.mess.addField("vvx", 0);
        Glider.mess.addField("vy", 0);
/* ********************************** */
/* EL CODIGO PARTICULAR DE CADA MODELO */
/* ********************************** */
        GSpace.sendto(Glider.mess, this, Via.calle);
    }
}
```

Fig. 4. Código base en Java de la clase `Entrada`

```
import galatea.glider.*;

class Salida extends Node {
    Salida() {
      super("salida", 'E');
      Glider.nodesl.add(this);
    }
    public void fscan() {
/* ******************************** */
/* CODIGO PARTICULAR DE CADA MODELO */
/* ******************************** */
    }
}
```

Fig. 5. Código base en Java de la clase `Salida`

En general tiene la forma que se muestra en la Fig. 6, pero el cálculo de las nuevas posiciones y velocidades de cada mensaje dependerá de cada modelo en particular. La clase `Regla` se asocia con `calle` al momento de instanciarlo en la clase `Via` como se muestra en la Fig. 3.

2.1 Modelo con autómatas celulares

En los modelos con autómatas la vecindad es factor determinante para realizar movimientos en el espacio que está compuesto por celdas. Para cada instante discreto de tiempo los vehículos solamente se desplazaran a una nueva posición si se cumplen las condiciones para

```
import galatea.glider.*;
import galatea.gspaces.*;

public class Regla {
  public Regla(){}
  public double[] move(Move e, Message m, Cell c) {
    double[] nCoord = new double[4];
/* ****************************** */
/* CODIGO PARTICULAR DE CADA MODELO */
/* ****************************** */
    nCoord=e.dChecking(m,c,nCoord, e.getDoor(1));
    return nCoord;
  }
}
```

Fig. 6. Código base en Java de la clase `Regla`

que esto ocurra. En este modelo la posición del vehículo i en el instante de tiempo $(t + it)$ viene dada por

$$x_i(t + it) = x_i(t) + v_i(t + it) \, , \tag{1}$$

donde, $x_i(t)$ y $v_i(t)$ son su posición y la velocidad en el instante t y $it = 1$ es el paso de tiempo del autómata.

Para este modelo en el método `move` de la clase `Regla` hay que agregar el código de la Fig. 7. En este código, en cada iteración se llama al método `desplazamiento`, también de la clase `Regla`, que asigna las velocidades de todos los vehículo de acuerdo a sus posiciones y velocidades anteriores. Luego cambiar la posición del vehículo según la ecuación (1) mediante la asignación de los campos del arreglo `nCoord`.

```
 if(lastUpdate<Glider.getTime()){
    lastUpdate=Glider.getTime();
    desplazamiento();
}
double[] nCoord = new double[4];
nCoord[0] = m.getdoubleValue("x")+
            m.getdoubleValue("vvx");
nCoord[1] = m.getdoubleValue("y");
```

Fig. 7. Código Java particular del método `move` para el modelo de autómatas celulares. Este código hay que agregarlo al de la Fig. 6

El método `desplazamiento`, que se muestra en la Fig. 8, revisa para cada vehículo las celdas que se encuentran delante de el y determina cuanto puede desplazarse.

```
void desplazamiento(){
  Cell[] c=Via.calle.getMove().getCells();
  Message m;
  int i;
  int v0;
  int vx;
  int vx1;
  for(i=0;i<c.length;i++){
    if(!c[i].isEmpty()){
      m = (Message)(c[i].getAg().getDat(1));
      v0 =m.getintValue("v0");
      vx = m.getintValue("vvx");
      vx1 = 0;
      while(vx1<v0&&vx1<vx+1){
        if(c[i+vx1+1].isEmpty())
          vx1++;
        else
          break;
      }
      m.setField("vvx", vx1);
    }
  }
}
```

Fig. 8. Código del método `desplazamiento` de modelo de autómatas celulares

2.2 Modelo con mapas acoplados

Este modelo se describe mediante el sistema de mapas acoplados

$$x_i(t + it) = x_i(t) + v_i(t)it + \frac{a_i(t + it) \; it^2}{2} \tag{2}$$

$$v_i(t + it) = a_i(t + it) \; it + v_i(t) \tag{3}$$

$$a_i(t + it) = a_i^* \left[1 - \left(\frac{v_i(t)}{v_i^*} \right)^2 - \left(\frac{s_i^*(t)}{s_i(t)} \right)^2 \right] \tag{4}$$

donde $x_i(t)$, $v_i(t)$ y $a_i(t)$ son la posición, la velocidad y la aceleración respectivamente del i-esimo vehículo en el instante t, it es paso de tiempo de las iteraciones, a_i^* es la aceleración máxima, v_i^* la velocidad deseada, $s_i = x_{i-1}(t) - x_i(t)$ es la brecha actual entre el vehículo i y el anteior a él y s_i^* es la brecha deseada, que viene dada por

$$s_i^*(t) = s_0 + v_i T + \frac{v_i(v_i - v_{i-1})}{2\sqrt{a_i^* b}} \, , \tag{5}$$

donde s_0 es la brecha mínima entre vehículos a velocidad cero y b es la desaceleración cómoda.

Para simular este modelo, en la clase `Entrada` se añaden varios campos a los nuevos mensajes como muestra la Fig. 9 y se declara e inicializa la variable `Message ant = null`, que hace referencia al vehículo inmediatamente anterior al actual. Luego, si existe un vehículo anterior, es decir `anterior` no es nulo, se asigna el campo `posterior` para este mensaje.

```
Glider.mess.addField("a", 0.0);
Glider.mess.addField("b",0.9);
Glider.mess.addField("a0",11.1);
Glider.mess.addField("g0",4.0);
Glider.mess.addField("T",1.5);
Glider.mess.addField("anterior",ant);
if(ant!=null)
 ant.addField("posterior",Glider.mess);
ant = Glider.mess;
```

Fig. 9. Código Java particular de la clase `Entrada` para el modelo de mapas acoplados. Este código hay que agregarlo al de la Fig. 4

Para liberar la memoria ocupada por aquellos vehículos que ya salieron del sistema, en el método `fscan` de la clase `Salida:` se anula el atributo `anterior` del mensaje `posterior` al mensaje que está saliendo, como se muestra en la Fig. 10.

```
Message post = (Message)(Glider.mess
               .getValue("posterior"));
if(post != null) post.setField("anterior", null);
```

Fig. 10. Código Java de método `fscan` de la clase salida para el modelo de mapas acoplados

Como el método `move` de la clase `Regla` depende de cada modelo, la fracción particular del código del método para el modelo de mapas acoplados se muestra en la Fig. 11. El método actualiza la velocidad de todos los vehículos invocando al método `desplazamiento`. Luego se actualiza el arreglo `nCoord` con las nuevas posiciones `x` y `y` según lo establecen las ecuaciones (2) - (4).

```
if(lastUpdate < Glider.getTime()){
  lastUpdate=Glider.getTime();
  desplazamiento();
}
nCoord[0]=m.getdoubleValue("x")
  +m.getdoubleValue("vvx")*e.getIt()
  +0.5*m.getdoubleValue("a")*Math.pow(e.getIt(),2);
nCoord[1]=m.getdoubleValue("y");
```

Fig. 11. Parte del código Java del método `move` para el modelo de mapas acoplados. Este código hay que agregarlo al de la Fig. 6

La Fig. 12 muestra el método `desplazamiento` de la clase `Regla`. El método calcula el desplazamiento de cada vehículo teniendo en cuenta si delante de éste hay otro vehículo o no. En caso de tener a un vehículo adelante, es decir `ant!=null`, se debe calcular el desplazamiento teniendo en cuenta la brecha entre vehículos `g`. En cambio, si no hay vehículo `anterior` simplemente se calcula la aceleración para un vehículo libre en la vía. Finalmente se actualizan los campos aceleración `a` y velocidad `vvx` del vehículo.

```
void desplazamiento(){
  double t=Via.calle.getIt();
  for(int n=1;n<=Via.calle.getWait().getEl().ll();n++){
    Message m=(Message)Via.calle
              .getWait().getEl().getDat(n);
    Message ant=(Message)m.getValue("anterior");
    double v_m=m.getdoubleValue("vvx");
    double T=m.getdoubleValue("T");
    double b=m.getdoubleValue("b");
    double a0=m.getdoubleValue("a0");
    double x_m=m.getdoubleValue("x");
    double v0=m.getdoubleValue("v0");
    double g0=m.getdoubleValue("g0");
    double stop=m.getdoubleValue("g0")+10;
    double a=m.getdoubleValue("a");
    cd Articulos/Mios/carlosDaniel/
    if(ant!=null){
      double x_ant=ant.getdoubleValue("x");
      double v_ant=ant.getdoubleValue("vvx");
      double g=x_ant-x_m;
      if(g>stop){
        double gg=g0+v_m*T+v_m*
                  ((v_m-v_ant)/(2*Math.sqrt(a0*b)));
        a = a0*(1-Math.pow((v_m/v0),2)
            -(Math.pow((gg/g),2)));
      }else{
        if(ant.getdoubleValue("vvx")!=0){
            a = 0;
        }else{
            a = 0;
            v_m = 0;
    } } } else
      a=a0*(1-Math.pow((v_m/v0),2));
    if(a >= 0)rticulos/Mios/carlosDaniel/

      m.setField("a", a);
    vx = a*t + v_m;
    m.setField("vvx", vx);
} }
```

Fig. 12. Código del método `desplazmiento` de la clase `Regla` para el modelo de mapas acoplados

2.3 Simulación con ecuaciones diferenciales

Éste es un modelo de espacio y tiempo continuos, el cual actualiza la posición y velocidad de cada vehículo resolviendo el sistema de ecuaciones diferenciales ordinarias,

$$\frac{dx_i}{dt} = v_i \tag{6}$$

$$\frac{dv_i}{dt} = a_i^* \left[1 - \left(\frac{v_i}{v_i^*} \right)^2 - \left(\frac{s_i^*}{s_i} \right)^2 \right] \tag{7}$$

donde x_i es la posición del vehículo i, v_i es su velocidad, a_i^* es su aceleración deseada, v_i^* su velocidad deseada, s_i^* su brecha deseada y s_i la brecha entre el i-esimo vehículo y el anterior a él. Para resolver los sistemas de ecuaciones diferenciales ordinarios se utiliza el integrador de `GALATEA`. Para ello, por cada uno de los sistemas de ecuaciones ordinarios se crea una clase que extiende la clase

Cont de GALATEA tal y como se muestra en la Fig. 13.

```
class vOde extends Cont {
  Message m;
  vOde(String l, double ic,Message m){
     super(l,ic);
     this.m = m;
  }
  public double feval(double x, Vector y){
    double T = m.getdoubleValue("T");
    double b = m.getdoubleValue("b");
    double g0 = m.getdoubleValue("g0");
    double v0 = m.getdoubleValue("v0");
    double v_m = m.getdoubleValue("vvx");
    double a = m.getdoubleValue("a");
    Message ant = (Message)m.getValue("anterior");
    double x_m = m.getdoubleValue("x");
    double x_ant;
    double v_ant;
    double g;
    double gg;
    double Vode = 0;
    if (ant!=null) {
      x_ant=ant.getdoubleValue("x");
      g=x_ant-x_m;
      v_ant=ant.getdoubleValue("vvx");
      gg=g0+v_m*T+v_m*((v_m-v_ant)/(2*Math.sqrt(a*b)));
      Vode=a*(1-Math.pow((v_m/v0),2)
           -(Math.pow((gg/g),2)));
    } else {
      Vode=a*(1-Math.pow((v_m/v0),2));
    }
    return Vode;
    }
}
class xOde extends Cont {
  Message m;
  xOde(String l, double ic, Message m){
    super(l,ic);
    this.m=m;
  }
  public double feval(double x,Vector y){
    double Xode=m.getdoubleValue("vvx");
    return Xode;
  }
}
```

Fig. 13. Implementación de las ecuaciones diferenciales del sistema de ecuaciones (6)-(7)

En la clase Entrada se añaden, como lo muestra la Fig. 14, los atributos al mensaje entre los que se encuentra anterior, posterior y una instancia ode del integrador de ecuaciones diferenciales Odes de GALATEA. Notese que a la instancia de Odes, ode se le agregan las instancias de las clases xOde y vOde, descritan en la Fig. 13, las cuales representan respectivamente a las dos ecuaciones del sistema de ecuaciones (6)-(7). La clase Salida es igual a la del modelo de mapas acoplados cuyo método fscan corresponde al de la Fig. 10.

Por último el método move de la clase Regla debe incluir el código de la Fig. 15. Note que el método toma de cada vehículo (mensaje) el integrador de sus ecuaciones de movimiento ode, le asigna un paso de integracion

```
Glider.mess.addField("a", 2.8);
Glider.mess.addField("b",0.9);
Glider.mess.addField("g", 0);
Glider.mess.addField("g0", 10.0);
Glider.mess.addField("T",1.5);
Glider.mess.addField("anterior",ant);
if(ant != null )
  ant.addField("posterior", Glider.mess);
Odes ode = new Odes(true, 0.0, 0.01);
ode.addEqs(new vOde("vvx",0.0,Glider.mess));
ode.addEqs(new xOde("x",Glider
          .mess.getdoubleValue("x")));
Glider.mess.addField("ode",ode);
ant = Glider.mess;
```

Fig. 14. Parte del código Java de la clase Entrada del modelo de ecuaciones diferenciales

y luego las integra numéricamente mediante el método ode.solve. Por último se asigna la solución de la primera ecuacion al campo de velocidad representado por vvx y se asigna la segunda solución al campo nCoord[0] que finalmente actualiza la posición del mensaje.

```
Odes ode = (Odes)m.getValue("ode");
ode.setStep(e.getIt());
ode.solve(Glider.getTime() + e.getIt());
m.setField("vvx", ode.getState(0));
nCoord[0] = ode.getState(1);
nCoord[1] = m.getdoubleValue("y");
```

Fig. 15. Parte del código Java del método move para el modelo de ecuaciones diferenciales

La Fig. 16 muestra los resultados de una realización del modelo base utilizando los tres modelos microscópicos de tráfico. La diferencia entre las pendientes de las líneas se debe a la diversidad de los valores asignados a los parámetros de cada vehículo. Notese que, como era de esperarse, en ninguno de los tres modelos los vehículos se adelantan. En las tres gráficas de la figura también se puede observar como cuando un vehículo se desplaza más lentamente, lo que se aprecia en los gráficos como líneas de menor pendiente, se acumulan los vehículos más veloces que vienen detrás de él. También, la Fig. 16 muestra la diferencia que hay entre el uso de un autómata celular (gráfica izquierda), en el que se observan cambios bruscos de las velocidades, y los otros dos micromodelos (gráficas centro y derecha), en los que la velocidad varía con mayor suavidad.

3 Modelo con incorporacón de vehículos

Esta variante del modelo base, representada en el diagrama de la Fig. 17, simula la incorporación de un vehículo en un punto intermedio de la vía. En ella se añade el nodo autónomo Incorporacion. La Fig. 18 muestra un ejemplo de los resultados de una realización

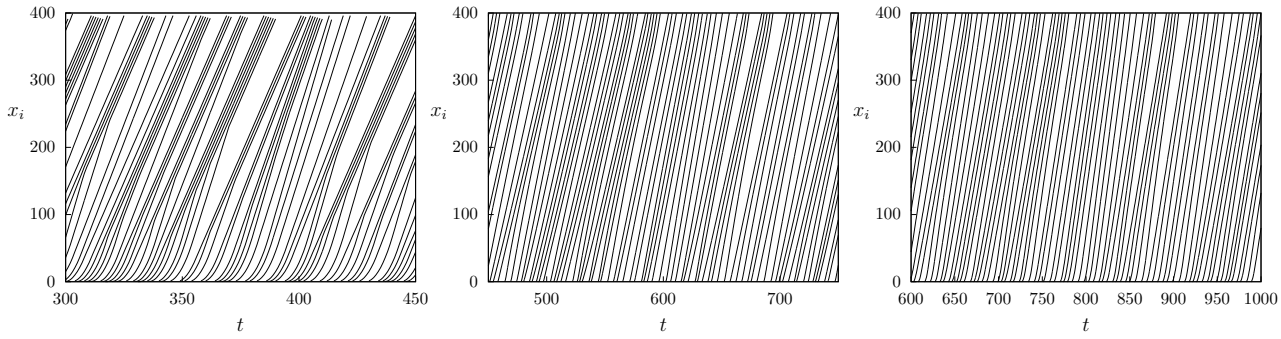

Fig. 16. Ejemplo de evolución temporal de la posición de los vehículos x_i en los tres micromodelos para el caso base. Cada línea representa la evolución de la posición de un vehículo. Para modelar diversidad en el comportamiento de los vehículos los parámetros de cada uno de ellos se asignaron en forma aleatoria. Izquierda: Autómata celular. Centro: Mapas acoplados. Derecha: Ecuaciones diferenciales ordinarias.

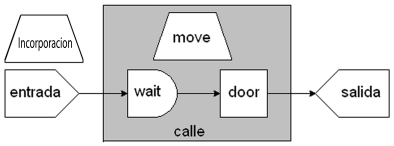

Fig. 17. Diagrama Galatea del modelo con incorporación de vehículos

```
static Incorporacion incorporacion=new Incorporacion();
  /* ****************** */
  /* ACA VA CODIGO BASE */
  /* ****************** */
  Glider.act(incorporacion, 60);
```

Fig. 19. Fragmento de código de la clase `Via`

de este modelo en la que se incorporan vehículos a baja velocidad en el punto medio de la vía cada $\Delta t = 60$ unidades de tiempo.

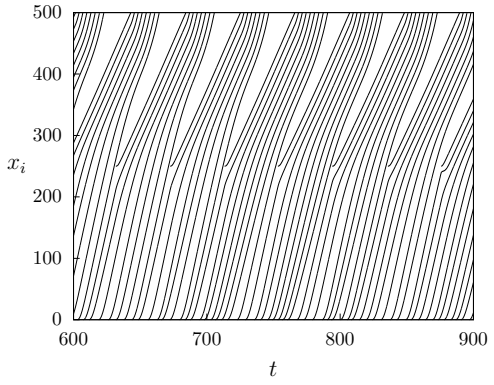

Fig. 18. Ejemplo de evolución temporal de la posición de los vehículos x_i en el modelo con incorporación de vehículos. Cada $\Delta t = 60$ unidades de tiempo se incorpora un vehículo lento en la mitad de la vía. Este gráfico, que corresponde al micromodelo de ecuciones diferenciales, no difiere significativamente de lo que se observa con los otros dos micromodelos.

En la Fig. 19 se muestran los cambios, con respecto al modelo base, hechos en el método `main` de la clase principal `Via`, responsable de instanciar y activar al nodo `Incorporacion`. También se modificó el nodo `Entrada` para ubicar a los vehículos que se incorporan. En este caso en particular se escogió la mitad de la longitud de la vía. La Fig. 20 muestra las líneas de código adicional del método `fact` de la clase `Entrada` requeridas para este fin.

```
/* ****************** */
/* ACA VA CODIGO BASE */
/* ****************** */
if(Incorporacion.activa){
  Incorporacion.activa=false;
  Glider.mess.addField("x", Via.l/2);
  Message p=Via.incorporacion.getPosterior();
  if(p != null){
    Message a = (Message)(p.getValue("anterior"));
    Glider.mess.setField("anterior",a);
    Glider.mess.setField("posterior", p);
    p.setField("anterior", Glider.mess);
    a.setField("posterior", Glider.mess);
  }
}
```

Fig. 20. Parte del código del método `fact` de la clase `Entrada` para la variante de los modelos con incorporación de vehículos

Por último, para el modelo con incorporación de vehículos se implementó la clase `Incorporacion` como se muestra en la Fig. 21 para los micromodelos de mapas acoplados y ecuaciones diferenciales. En ella se implementa el método `fact`, que, aunque conceptualmente tiene la misma función para todos los modelos, para el modelo de autómatas celulares se implementa con ligeras diferencias al mostrado aquí, para así conservar el paradigma de autómata celular. En todos los

casos, en primer lugar se determina que haya suficiente espacio para que resulte la incorporación. De ser así entonces se activa el nodo `entrada` con el atributo `incorporacion.activa` en verdadero. En cambio, de no ser suficiente el espacio entonces se reprograma la activación del nodo `incorporacion` hasta lograr la incorporación del vehículo.

```
import galatea.glider.*;
import galatea.gspaces.*;

class Incorporacion extends Node{
  static boolean activa;
  Message posterior;
  Incorporacion(){
    super("incorporacion", 'A');
    activa=false;
    Glider.nodesl.add(this);
  }
  public void fact(){
    posterior = null;
    Message m;
    int ll = Via.calle.getWait().getEl().ll();
    double x;
    for(int n=1; n<=ll;n++){
      m = (Message)Via.calle
           .getWait().getEl().getDat(n);
      x = m.getdoubleValue("x");
      if(x<=Via.l/2){
        Message a=(Message)(m.getValue("anterior"));
        if(x<Via.l/2-10&&(a==null||
              a.getdoubleValue("x")>Via.l/2+10)){
          posterior = m;
          Glider.act(Via.entrada, 0);
          it(60);
          activa=true;
          return;
        } else {
          it(0.1);
          return;
        }
      }
    }
    it(60);
    return;
  }
  public Message getPosterior(){
    return posterior;
  }
}
```

Fig. 21. Clase `Incorporacion`

4 Modelo con semáforo

La Fig. 22 muestra el esquema `GALATEA` de la variante de modelo base que incorpora un semáforo en la vía. El modelo está conformado ahora por dos espacios distintos, uno antes del semáforo llamado `calle` y otro despues llamado `otraCalle`. Además, se incluye un nodo autónomo `semaforo` que controla las acciones del semáforo en la vía, este último representado por el atributo `Door` de `calle` que envía los vehículos a espacio `otraCalle`. Las clases `Entrada` y `Salida` quedan sin cambios con respecto al modelo base de cada modelo en particular. En cambio las clases `Via` y `Regla` si se ven modificadas.

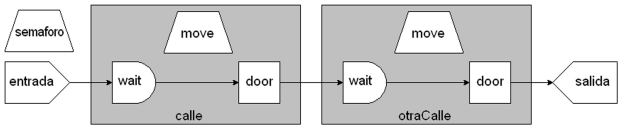

Fig. 22. Diagrama del la variante del modelo con semáforo

En la clase `Via`, que se muestra en la Fig. 23, se instancian un nuevo espacio y un nuevo nodo que llamamos `otraCalle` y `semaforo` respectivamente, se añaden la geometría referente al nuevo espacio y finalmente se programa el tiempo de la primera activación de `semáforo` y de `otraCalle`.

```
static Space otraCalle=new Space("otraCalle",0,
                    3*l/4,6.0,1.0,0.1,"Regla");
static Semaforo semaforo = new Semaforo();
  /* ****************** */
  /* ACA VA CODIGO BASE */
  /* ****************** */
  otraCalle.addWall(l/2, 0.0, l, 0.0);
  otraCalle.addWall(l/2, 12.0, l, 12.0);
  otraCalle.addWall(l/2, 0.0, l/2, 12.0);
  otraCalle.addWall(l, 0.0, l, 12.0);
  otraCalle.addDoor(l, 0.0, l, 12.0,
                    salida,'U',1,0.0,0.0);
  otraCalle.build();
  display.addSpace(otraCalle);
  /* ****************** */
  /* ACA VA CODIGO BASE */
  /* ****************** */
  Glider.act(otraCalle.getMove(), 0);
  Glider.act(semaforo,0);
```

Fig. 23. Parte del código Java de la clase `Via` para la variante de los modelos con semáforo

La Fig. 24 muestra los resultados de tres simulaciones de este modelo utilizando los tres modelos microscópicos de tráfico. En todas estas simulaciones el semáforo está ubicado a en la mitad del trayecto simulado, a $l/2 = 100$ unidades de distancia Se aprecia como en la simulación con autómatas celulares (gráfica de la derecha) los cambios de velocidad son más abruptos que con mapas acoplados (centro), siendo los más suaves los del micromodelo de ecuaciones diferenciales (izquierda).

En el método `move` de la clase `Regla`, además de la información de cada modelo en particular (ver Fig. 6, 7, 11 y 15) se añade el código de la Fig. 25, que reduce a cero la velocidad del vehículo que trate de salir del espacio `calle` cuando su `Door` esté cerrada, lo que representa al semáforo en rojo.

Finalmente, la clase `Semaforo`, que extiende a `Nodo`

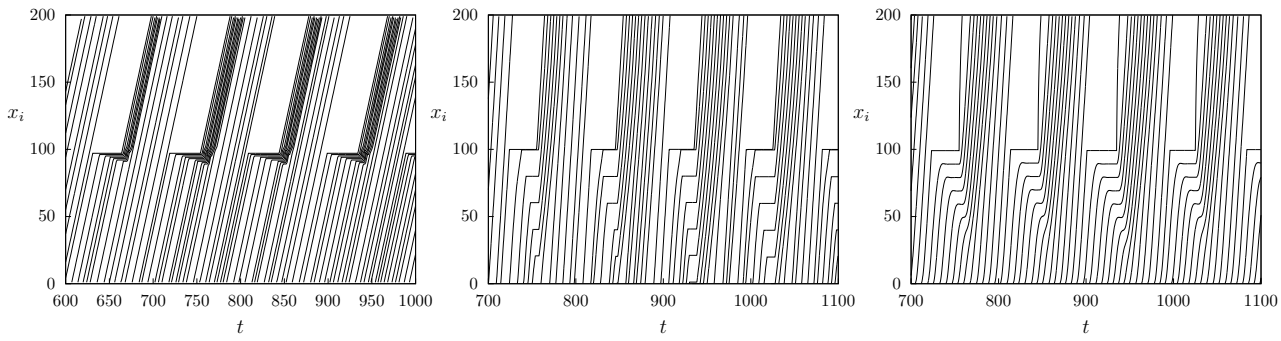

Fig. 24. Ejemplo de evolución temporal de la posición de los vehículos x_i en los tres micromodelos para el caso con semáforo. Cada línea representa la evolución de la posición de un vehículo. El semáforo está ubicado en la mitad del trayecto, a $l/2 = 100$ unidades de distancia. Izquierda: Autómata celular. Centro: Mapas acoplados. Derecha: Ecuaciones diferenciales ordinarias.

```
if(nCoord[2]==1 && e.getDoor(1).isClose()){
  m.setField("vvx", 0);
}
```

Fig. 25. Parte del código añadido al método `move` de la clase `Regla` para la variante con semáforo

de tipo **A** se muestra en la Fig. 26. En ella se implementa el método `fact` cuya única función es abrir y cerrar la clase `Door` de `calle`. Como ejemplo, en el código la clase se mantiene abierta por 55 unidades de tiempo de simulación y se cierra por 35, que equivaldrían aproximadamente a 55 y 35 segundos respectivamente de acuerdo a los valores de los parámetros relacionados con velocidades y aceleraciones del modelo.

```
import galatea.glider.*;
class Semaforo extends Node{
  Semaforo(){
    super("semaforo",'A' );
    Glider.nodesl.add(this);
  }
  public void fact(){
    Via.calle.getMove().getDoor(1).setClose(
        !Via.calle.getMove().getDoor(1).isClose());
    if(Via.calle.getMove().getDoor(1).isClose()){
      it(35);
    }else{
      it(55);
    }
  }
}
```

Fig. 26. Código de la clase `Semaforo`

5 Conclusiones

Se implementaron, sobre la plataforma de simulación Galatea nueve modelos de tráfico vehicular en una vía de un solo carril: un modelo base, un modelo con incorporación de vehículos y un modelo con semáforo, cada uno de ellos utilizando tres paradigmas miroscópicos de tráfico diferentes: autómatas celulares, mapas acoplados y ecuaciones diferenciaes. La plataforma demostro tener la expresividad y el rendimiento suficiente para simular los nueve modelos con más de 100 vehículos circulando simultaneamente.

Agradecimientos

Este trabajo fue financiado en parte por el proyecto C-1696-10-05-B del Consejo de Desarrollo Científico, Humanístico, Tecnológico y de las Artes de la Universidad de Los Andes.

Referencias

Dávila J, Uzcátegui M y Tucci K, 2004, Universidad de Los Andes, Mérida.

Chopard B y Droz M, 1998, Cellular automata Modeling of Physical Systems, Cambridge University Press, Cambridge.

Galatea, http://galatea.sourceforge.net/Home.htm, septiembre de 2010.

Greenshields BD, 1935. A study of traffic capacity. In Proceedings of the Highway, Highway Research Board, Vol. 14, No. 1, pp. 448-477.

Helbing D, Kesting A y Treiber M, 2009, Agents for Traffic Simulation, in Multi-Agent Systems:cw Simulation and Applications, Editores Uhrmcher A y Weyns D, Cap. 11, pp 325-356, CRC Press, Boca Ratón.

Lighthill MJ y Whitham GB, 1955, On kinematic waves: II. A theory of traffic on long crowded roads, Proc. Roy. Soc. of London A, Vol. 229, pp. 317-345.

Nagel K y Schreckenberg M, 1992, A cellular automaton model for freeway traffic, J. Phys. I France, Vol. 2, pp. 2221-2229.

Newell GF, 1961, Nonlinear effects in the dynamics of

car following, Operations Research, Vol. 9, pp. 209.
Prigogine I y Herman R, 1979, A Two-fluid Approach ot Town Traffic, Science Reprint Series, Vol. 204, pp. 148-151.
Richards PI, 1956, Shock waves on the highway, Operations Research, Vol. 4, pp. 42-51.
Uzcátegui M, Dávila J y Tucci K, 2007, GALATEA: Plataforma de Simulación de Sistemas Multi-Agentes, Memorias de VI Jornadas científico técnicas de la Facultad de Ingeniería, Universidad de Los Andes, Mérida, Venezuela. pp 923-930.
Zeigler BP, Praehofer H y Kim TG, 2000, Theory of modeling and simulation: integrating discrete event and continuous, Academic Press, San Diego.
Laffaille K, 2005, Gspace, meta-modelo para simular desalojos de Espacios Urbanos y Arquitectónicos basado en GALATEA, Tesis de Maestría. Universidad de los Andes, Mérida.

Recibido: 23 de mayo de 2011
Revisado: 30 de julio de 2011