An Object-Oriented Microscopic Traffic Simulator

Gustavo K. Andriotti

and

Ana L. C. Bazzan
Instituto de Informática, UFRGS
Caixa Postal 15064
91501-970 – Porto Alegre, Brazil
{fgka,bazzan}@inf.ufrgs.br

Abstract

The aim of this paper is to outline the basic concepts of an object–oriented microscopic traffic simulator under development by us, and report the preliminary results. The motivation behind the development of such a simulator is the increasing utilisation of microscopic simulators based on Cellular Automata. We propose an object–oriented version, integrated with database and visualisation capability, thus facilitating its usage by technicians of the traffic simulation branch. We focus on the classes design, and especially on the driver class, which is integrated in the proposed framework. We then compare our implementation with previous ones at the conceptual level, showing that the object–oriented paradigm permits an easy change of specification of the model of the driver, without having to rewrite significant parts of the code.

1 Introduction

It is a fact that the increasing computational power of the machines available has allowed the development and efficient run of microscopic traffic simulators ([2], [5], [6], [7], [8], [10], [13], [14]). The main advantage of a microscopic simulation is the possibility of describing the components at an individual level, which is normally more realistic.

However, this kind of approach may not be suitable enough to model more complex interactions like for instance giving advice to drivers. This happens due to the lack of a modelling of driver's goals, preferences and intentions when it comes to route choice. We propose to integrate a model of drivers based on multi–agent techniques, i.e., we try to model the driver as close as possible to the cognitive level. For this purpose, we implement a new module in the simulator, which deals with all issues related to the intelligent being behind the wheels of the vehicle.

The remaining of this paper is organised the following way: in the next section we briefly review the model which provide most of the ideas for our implementation. Section 3 presents our proposal and the state of its implementation. In Section 4 a comparison between the two simulators is discussed, and Section 5 presents some application scenarios for our simulator. Section 6 thus concludes discussing future directions of this work.

2 The Nagel–Schreckenberg Model

2.1 Overview

Microscopic models for traffic simulation are in general more complexes than the macroscopic ones. In order to cope with this complexity, cellular–automata (CA) based models have being proposed such as the Nagel–Schreckenberg ([8]) one. In short, each road is divided in unit cells with a fixed length (normally 7.5 meters). This permits the representation of a road as an array where positions show whether a car is present or not, at each given time. Each vehicle travels with a speed which is always represented by the number of cells it can advance at each time step. This characterises the model as discrete and, moreover, most of the quantities are integers, turning the efficiency good enough for real time simulation and control of traffic.

The Nagel–Schreckenberg model ([8]) was originally created to tackle the single–lane, highway scenario. By now, many enhancements were added so that it can also deal with urban, multiple–lane scenarios ([9]). However, for the sake of comprehension, let us focus on the simplest model. The extensions are easily deduced.

As for the network representation, each road is described as a composition of nodes and edges representing intersections and roads. The expression edge is used to refer to directed edges representing one direction of motion on a road, i.e., one road usually consists of two (oppositely directed) edges. The road is subdivided into cells of length 7.5 meters, which can be either empty or occupied by one vehicle. This represents the average space a vehicle takes in a traffic jam, but can be suitably adjusted. It is important to do this adjustment so that each time step in the simulation corresponds to the desired time frame. Every vehicle has a nonnegative integer speed. For one update of the totality of cells, the following four steps are performed simultaneously for all vehicles:

1. Accelerate with probability $1 - p$, where p is the non-acceleration probability (defined on driver);
   (a) If the car will accelerate, then $v_{temp} \leftarrow v_{current} + 1$, where: $v_{current}$ is the current velocity;
   (b) If the car will not accelerate, then $v_{temp} \leftarrow v_{current}$;
2. New velocity will be: $v_{new} \leftarrow min(gap, min(v_{temp}, v_{max}))$, where: v_{new} will be the next velocity, v_{max} is the maximum velocity allowed and gap is the distance (in cells) to the vehicule ahead;
3. Move v_{new} cells foward.

Investigations reported in the literature (references cited above) showed that despite its simplicity the cellular automaton model is capable of reproducing macroscopic traffic flow features including realistic lane changing behaviour (for an overview see [9]). Later on, this basic model was extended to deal with enhanced scenarios ([5], [6], [11]). In the urban traffic scenario, there was the need to add more elements such as traffic lights and more complex intersections. The overall simulation tool consists of different elements explained in detail below.

- Singlelane edges: work as the basic cellular automaton. In addition, turning sections can be connected for each direction at the end of the edge. Vehicles that have to use such a section appear on to the beginning of it; if the vehicle is hindered at entering, it has to wait at a kind of buffer or virtual position until an entering is possible.

- Multilane edges: at the end of multilane roads, the lane–changing behaviour strongly depends on the desired driving direction; for this purpose, multilane edges are subdivided into different regions.
- Direction change: lane changes are unconstrained; a car only changes lane according to the destination directions allowed; sometime vehicles have to wait if they are not on the desired lane, so such waiting vehicles lock positions and can cause deadlocks.

2.2 Priority Rules

Urban road networks differs from freeway networks by various features but one of the most important is the high density of intersections. It is important for the simulation to have realistic throughputs at the intersections, which requires realistic traffic light plans and the consideration of complex priority rules. In the model described in Esser and Schreckenberg ([5]) and Esser et al. ([6]), each edge in the network has a driving direction dependent flag at its end, by which vehicles may be prevented from advancing further. Traffic lights actions are simulated by switching the flags which correspond to predefined plans. Further priority rules are realised by switching leave flags depending on vacant cells on other edges. Here an additional important parameter is the number of gaps prior of cells.

Finally, it is possible that vehicles waiting at an intersection hinder the further advance of each other; in this case one of these is randomly chosen to advance in order to clear the deadlock. A similar problem arises if a vehicle has priority, but cannot drive on (e.g., due to red traffic light or a crowded destination edge); then its priority is neglected for other vehicles.

2.3 Vehicle Types

Vehicles are characterised by their maximum speed, length (number of occupied cells) and a probability of carrying out risky lane changes. In the standard version, the deceleration probabilities are not individual, but assigned to the driver. There are two special kinds of vehicles: those which can be guided periodically along predefined routes following timetables to simulate public transport (bus or tram stops), and artificial vehicles which cover hindrances such as accidents or road works (these vehicles have special length and vmax =0).

2.4 Sources and Sinks

Sources and sinks are a way to model the appearance and disappearance of traffic flow, respectively. Basically, sources and sinks can be linked to every cell in the network. At sources, when vehicles are created, they are also characterised by type, guidance mode and other necessary information on demand. As for sinks, probabilities for leaving the network are attributed to every edge for randomly driven vehicles. This is aimed at various purposes like e.g. the modelling of parking lots.

2.5 Sensors and Detectors

An essential requirement for traffic simulation is to have input data about the traffic state. For this purpose, detection devices like inductive loops or camera sensors are incorporated in the simulation: at measure points. The input are local data like number, flow, or density of vehicles as well as average speed are collected for different vehicle types.

2.6 Simulation Control

The overall simulation tool consists of two main processes. The master controller maintains the overall coordination: it checks static and dynamic network data read from the database for consistency and initialises the scenarios. During the simulation it receives and updates dynamic data like turn counts, handles the simulation output including updates of the graphics and if necessary provides current data to the routing module, which updates of route plans for the vehicle guidance system. The current network dynamics is carried out by the microsimulation process, which involves vehicle motions, traffic light updates, and data collection for statistics. The advantage of this subdivision is twofold: On the one hand, it speeds up the simulation, since the microsimulation can be parallelised. Furthermore, in practice, it facilitates the overall handling because it is possible to rearrange graphical output or data formats, among others, without caring about the complex structure of the microsimulation.

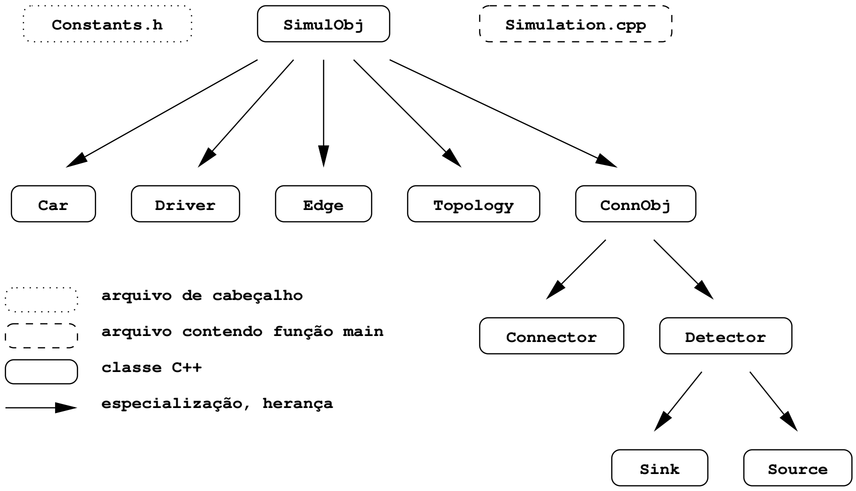

Figure 1: Organisation of Classes

3 The Object–Oriented Implementation of a Microscopic Simulation

3.1 Overview

The traffic simulator proposed here follows a microscopic and discrete simulation approach, combined with an object–oriented paradigm. The simulator is under development using C++/MySQL platform. MySQL was chosen to be the database support for this project phase. The object–oriented paradigm was chosen owing to its ability to hide the implementation details inside classes (there is no need to know how it was designed, just what it does). Besides, one can encapsulate a given behaviour, i.e., all that relates to the object Driver, for example, will be on Driver class and one can create "isolation rings" and solve implementation problems one at time. Moreover, the debugging process is speeded up on this approach as one can (precisely) isolate bugs on certain class. Moreover, with classes it is possible to extend actual classes, implement interfaces (abstract classes) and substitute standard classes. The SQL database model provides an extensive language to manipulate data in form of tables and in a timely fashion. Performance is an issue on real–time simulation, so a database with simple interface and quick response is required. MySQL is a free implementation of SQL/92 standard with a fair time response on database queries.

3.2 The Architecture and Components

All over the simulation, the concept of templates and inheritance were used. Templates are structures which are independent of the data type. For instance, a sum function can be implemented without specifying over which data type that sum function (can be a integer type or floating point type); this will be specified just when it will be necessary. Inheritance is the property to take an already specified behaviour from other classes. Hence, if there is a class that implements sum and one wants to add the subtraction function, one should just extend that class (and add the subtraction function) by allowing it to inherit the properties of the former.

In the programming task, almost every class is a template. This choice was made to increase the flexibility of the simulator and to add different implementations for a given class. This means that classes can be replaced without code rewriting.

3.2.1 Overview of the Organisation of Classes

In the simulator core, there are few classes that are responsible for the entire simulation. This classes are abstraction from reality, inspired on traffic simulator implemented by Nagel, Schreckenberg, and collaborators ([6], [8], [10], [11]). These classes are organised by dependency, not inheritance. Usually, a class needs some methods from other classes. For example, the Driver needs some methods from the Car class in order to know its location and decide what to do next. There is inheritance too, although just to establish a common interface to similar classes (for example, in Connectors, Sources and Sinks there is common methods) and to save time at the programming task.

For all classes, there is an unique ID that obey a mask to quickly identify the class type assigned accordingly to the database or dynamically created (used on Car class).

The whole simulator was designed to use discrete time steps and discrete space segmentation. The entire behaviour of the simulation was designed to use Schreckenberg model ([8]).

3.2.2 Class Topology

Like a centralised manager, this class controls the whole simulation, setting time steps and coordinating all update procedures. On this class resides the network creation and control. All other classes are subordinated to Topology, as depicted in Figure 1.

3.2.3 Class Edge

On Edge we have a homogeneous roadside segment, i.e., the number of lanes, orientation (road hand) are the same for a particular Edge. Actually the number of lanes can vary in the Edge but restricted to a limited number of variations. This lane number variation just can mimic an Edge obstruction, using stopped Cars.

Edge data carry Cars and let them know about its state. This state include its position, gap to nearest Cars, length and its target ending direction (when the car reach the edge's limit). Moreover, edge does not allow Cars to pass through each other, and control turning probability (set by Topology).

3.2.4 Class Connector

Connector simulates a road intersection. In this class we have all priority rules to allow a Car to use that Connector. Basically, Connectors are used to connect Edges in a pairwise way as to compose a road. Connectors have no dimension, i.e., a Car do not consume time to pass through it. This implementation was chosen to avoid deadlock on Connectors. Connectors can have up to 16 Edges connected to each, 8 to get in and 8 to get out, but it can be increased or decreased (just resetting a parameter at compilation time).

3.2.5 Class Detector

On a detector traffic flow controlers and counters are mimic. So it can act like and fine tune traffic flow controle or and data colector. On the first case suppose that there are real traffic flow data available so a detector will remove extra vehicles or add the missing ones. And the other case it will be used to collect data from the traffic network.

3.2.6 Class Source

This class is designed to allow a Car to be inserted on a traffic network. Its function is to insert Cars at a given rate into the Edge that it is connected to. Sources are a special kind of Connector. Moreover, Sources bind Cars to Drivers. These Sources, in the near future, will get their rates from a database and set them on the fly. For purpose of tests, they just have a fixed rate of insertion.

3.2.7 Class Sink

Sinks perform Car collecting. Each Sink collects all Cars that leave the Edge it is connected to, and delete them. On this removing process it guarantees that there is no classes attach to it.

3.2.8 Class Car

This class just carries the car state, i.e., it has all data to precisely locate a car. There are data that indicate which direction that car will take, by asking Edge and Driver for information. Car is a "static" entity, which means that a Car cannot change its state by itself. This will be performed by the Edge and Driver classes. It plays a dummy role on simulator, because it is just act like a temporary memory, or flags, to the system.

3.2.9 Class Driver

The Driver guides the Car direction by setting its properties. It decides whether a Car will change its actual lane or accelerate and, in richer scenarios, which direction it will take at an intersection. A (type of) Driver can control several Cars without limitation, because Cars carry all vital data to decide next state. There are many different aspects, which are handled by the Driver Class, as discussed below.

First, the Driver focuses on accelerating, breaking, bypassing, or changing lanes. This behaviour is usually described by a microscopic traffic flow model such as the Nagel–Schreckenberg model or other car–following model. However, this implementation allows for exchanging these models easily, i.e., the architecture is open.

Second, the Driver manages the navigation through the network. This can be done in two ways: the Driver can statistically decide at every intersection, or it can follow a pre–defined route. For the former approach, turning probabilities have to be determined. The latter approach needs origin–destination data as input. The implementation also allows for assigning a mental map to the Driver, which represents the knowledge about the network.

Third, the Driver has to communicate with for instance traffic control centres, which send traffic messages to help to navigate through the network. These messages need to be processed by the Driver with regard to its mental map. In general, this class will be a repository for different kinds of information: the type of driver (i.e. aggressive, calm, passive, in a hurry, etc.); his/her mental states (i.e. beliefs, preferences, goals, etc.); and possibly some sort of on–board information system (e.g. radio, Internet, or other navigation system).

The main advantage of this implementation is that the models can be exchanged easily and different models can be used at the same time. Our overall goal here is to provide a public class so that the user can insert his model of the driver. For instance, we will incorporate the inference engines which use the information provided by mental states based approaches to model drivers (e.g. [1], [3], [12]), or decision–making under dynamic traffic scenarios ([15], [16]).

3.3 The Database

The database is designed to perform queries to receive all network parameters. The main motivation is that not all the data will fit on a generic database model used on the simulator. To provide a flexibility level on this issue, there are auxiliary programs to automatically generate classes that implement database interface. For example, in the simulator database we expect to deal with probabilistic measurement of flow, but not all provided data correspond to flow (one can have data regarding occupancy rate, density or travel time for instance). To solve this, a function can be specified to convert one system to another, so that the database will be filled with occupancy measures or density, while the simulator receives converted data (flow). Alternatively, we can generate a new database and the auxiliary programs make all proper classes to use this new database.

3.4 Description of the Simulation

Each step of the simulation performs a sequence of procedures as shown above. After that we have the simulation process. All steps above are performed once per simulation cycle, not on every simulation step.

1. Eliminate all cars on *Sinks*;
2. Insert new cars throught *Sources*;
3. Verify all *Detectors*;
4. Update inter-*Edges* gap;
5. Perform lane change on *Edges* (if is an even simulation step perform left-to-right lane change, else right-to-left lane change);
6. Update intra-*Edges* gap (inter-*Cars* gap);
7. Apply *Cars* linear movement.

4 Comparison

Most of the current implementations of CA–based microscopic simulators (see all references of works by Schreckenberg and colleagues as well as TRANSIMS2, [18]) are based on more or less common rules for the evolving of the vehicles in a given network. Our implementation basically follows the Nagel–Schreckenberg model, both the original one as well as the further additions (e.g. to cope with traffic signals, lane change and so on). However, a basic difference is that our version was designed to be as general as possible profiting from the concept of object–orientation and hierarchy of classes.

The object–oriented paradigm allows us to hide the implementation details inside the classes. In practice, this means that any maintenance in the functionality of any class can be easily implemented, without having to rewrite code which is not related to the actual change or addition. This might look simple and obvious but it has been not the case. Our experience shows that most programmers rewrite entire modules from scratch because they cannot understand the existing code. The paradigm of object–orientation, if well understood, might put an end in this possibly inefficient way of software development.

Besides, one can encapsulate a given behaviour inside a single class and solve implementation problems one at time. Hence, the debugging process is speeded up with this approach as one can isolate bugs on certain class. The second point in which our implementation differs from others is in the design of a database especially constructed to this application. This can eliminate the time–consuming tasks of converting units (e.g. occupancy rate to flow of vehicles), and of designing a database for each scenario. Moreover, the database we use relies on open technologies like MySQL, and a tripple-layered architecture that isolate database technology from the simulator. That means that is possible to change the database for another one like: Oracle, DB2, mSQL, etc.

The major difference however, is that we propose a class called Driver. Such class does not exist in other implementations, in which drivers and vehicles are modelled indistinguishably. We depart from the idea that a vehicle can do something by itself. Rather, we open the possibility for the user to model drivers as it pleases. For instance, the easiest way would be to follow the well known version of the so–called driver–vehicle–unit, where both driver and vehicle are seen as a single agent which acts basically in a reactive way using standard models like the car–following.

On the other hand, if necessary, the user of the simulator can implement more complex models for the drivers like BDI–based ones as proposed in Bazzan et al. ([1]) and Rossetti et al. ([12]). To do so, all that is necessary is to design new instances of the class Driver and give the corresponding semantics to the slots of these instances.

Here we see the major advantages of the object–orientation: using standard programming techniques, significant portions of the code have to be rewritten or revised to cope with such a change in the way a driver behaves. Using the object–oriented paradigm, only the class driver has to be modified. And even this modification is much simpler here: only the functionality of the class change, the interfaces with other classes remaining the same. That let methods to be override and have another semantic.

We are aware of the performance of other simulators. It is reported, for instance in [6], that the simulation of the inner ring of Duisburg city using an implementation of the Nagel–Schreckenberg model for that scenario runs a whole day of typical traffic in about 20 minutes on a PC (Pentium P133). This scenario involves more than 100 nodes, about 300 edges, and more than 20000 cells. Although we cannot draw quantitative conclusions at this point, we would say that the performance of our version would be lower due to the need of message passing between objects. This is a trade–off we are willing to make since we know that the computational power of PCs is increasing in a way which permits us to simulate x time steps in less than this time by several orders of magnitude.

5 Results

The scenario depicted in Figure 2 were simulated in four different configurations. These are the result of the combination between a *loop* and a *straight* trajectory, and two situations of traffic flow: jammed and free-flow. In order to gain information about the network we have used a floating car, i.e. a car which travels in the network and returns some information regarding its specific position at each time step (such as speed, position, density of the network in that position).

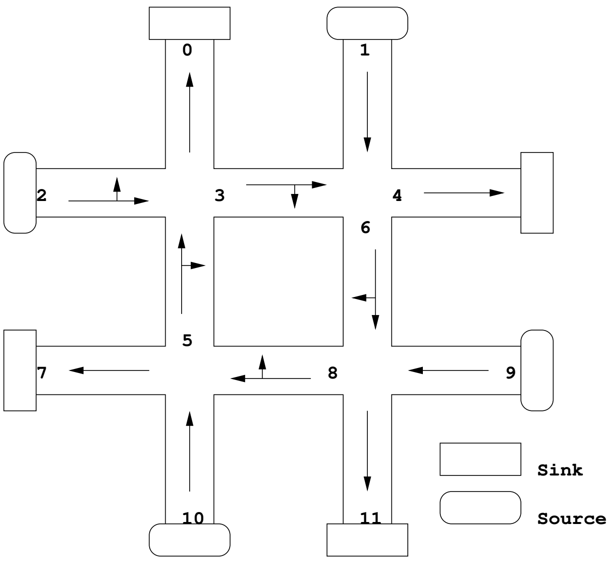

Figure 2: Topology of Simulation

The chosen routes were:

- Loop: passing through *Edges* 2, 3, 6, 8, 5 and back to *Edge* 3 (doing a loop circiut).
- Straight: the *Edges* are 2, 3, 6, 8, 5 and 0 (been eliminated by the *Sink* at the end of *Edge*).

The probabilities of vehicles going to each possible direction, on each *Edge*, is shown on Table 1.

Edge	direction							
	0	1	2	3	4	5	6	7
0	1,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1	0,0	0,0	0,2	0,0	0,8	0,0	0,0	0,0
2	0,1	0,0	0,9	0,0	0,0	0,0	0,0	0,0
3	0,0	0,0	0,4	0,0	0,6	0,0	0,0	0,0
4	0,0	0,0	1,0	0,0	0,0	0,0	0,0	0,0
5	0,3	0,0	0,7	0,0	0,0	0,0	0,0	0,0
6	0,0	0,0	0,0	0,0	0,5	0,0	0,5	0,0
7	0,0	0,0	0,0	0,0	0,0	0,0	1,0	0,0
8	0,2	0,0	0,0	0,0	0,0	0,0	0,8	0,0
9	0,0	0,0	0,0	0,0	0,4	0,0	0,6	0,0
10	0,5	0,0	0,0	0,0	0,0	0,0	0,5	0,0
11	0,0	0,0	0,0	0,0	1,0	0,0	0,0	0,0

Table 1: Table of turning probabilities

To simulate the free-flow scenario a 0.3 insert rate was set on the *Sources*, each simulation step (or, given the simulator semantic, 30% of chance that a *Car* will be inserted on the actual step). And for a jammed scenario a 0.7 insert rate was adopted. The other parameteres, maintained the same on all simulations, are: nonacceleration probability equals 0.0; a 5 cells per time step of maximum velocity; 20 cells of *Edge* length (for all *Edges*); all vehicles with 1 cell of length and all *Edges* with just one lane. *Connectors* are on each *Edge* intersection, on figure 2. *Sources* and *Sinks* are disposed according to that diagram.

6 Further Application Scenarios

The performance of the simulator will be tested in the near future within at least other two of the three following scenarios. First, an urban one where we have continuous data for a dense network of detectors (city of São Paulo, Brazil). The second one is an urban scenario in which the data is sparse and the detectors located far apart, giving us the possibility to mimic a macrosimulation (city of Porto Alegre, Brazil). Finally, a highway scenario composed of two alternative routes to be used for travelling between the cities of São Paulo and Campinas, two important industrial and technological centres in the Southeast of Brazil.

7 Conclusion and Future Work

We have presented the concept of a CA–based microscopic traffic simulator, entirely designed to be object–oriented. We have discussed the function of the main classes, the simulation procedures, as well as the database aimed at storing the data related to each scenario to be simulated. We have shown the advantages of such a concept, especially if we want to give different functionalities to the main actor in this system, the driver. The next step of this work is to simulate those scenarios discussed above, initially with standard models for the driver in order to compare both the results and the performance of the simulator with existing versions. We further plan to integrate a more complex version of the driver, possibly using a BDI–based approach.

References

[1] A. Bazzan, J. Wahle, F. Klgl. Agents in Traffic Modelling – From reactive to Social Behaviour, in: Proc. of KI–99, eds. W. Burgard, A.B. Cremers, T. Christaller, LNAI 1701 (Springer, Berlin, 1999), pp. 303–306.

[2] J. Barcelo, J. Casas, J.L. Ferrer, and D. Garcias. Modelling advanced transport telematic applications with microscopic simulators: The case of AIMSUN2. In W. Brilon, F. Huber, M.Schreckenberg, and H.

Wallentowitz (eds.), Traffic and Mobility: Simulation – Economics – Environment. Springer, Heidelberg (1999).

[3] H. Dia and H. Purchase. Modelling the impacts of advanced traveller information systems using intelligent agents. Road and Transportation Research, 8 (3): 68–73, (1999).

[4] D. Chowdhury.; D. E. Wolf; M. Schreckenberg. Physica A, 235, 417.

[5] J. Esser and M. Schreckenberg. Microscopic Simulation of Urban Traffic based on Cellular Automata. Int. J. of Mod. Phys. C 8 5, 1025 (1997)

[6] J. Esser, L. Neubert, J. Wahle, M. Schreckenberg. Microscopic Online Simulation of Urban Traffic. In: Proc. of the 14th ITS, ed. A. Ceder (Pergamon, 1999), pp. 535–554.

[7] I. Kosonen. HUTSIM – a simulation tool for traffic signal control planning. HUT Transportation Engineering Publication 89, 1996.

[8] K. Nagel and M. Schreckenberg. A Cellular Automaton Model for Freeway Traffic. J. Phys. I France 2, 2221 (1992)

[9] K. Nagel, D.E. Wolf, P. Wagner, and P. Simon. Two–lane traffic rules for cellular automata: A systematic approach. Phys. Rev. E 58, 1425 (1998).

[10] K. Nagel, J. Esser, and M. Rickert. Lagre–scale traffic simulations for transport planning. In: D. Stauffer (ed.) Ann. Rev. Of Comp. Phys. VII, 151–202, Singapore (2000). World Scientific

[11] M. Rickert; K. Nagel; M. Schreckenberg; A. Latour (1997). Physica A, 231, 534.

[12] R. J.F. Rossetti, R. H. Bordini, A. L.C. Bazzan, S. Bampi, R. Liu, and D. Van Vliet. Using BDI Agents to Improve Driver Modelling in a Commuter Scenario. To appear in: Trasnp. Res. C (2001).

[13] P. Wagner, K. Nagel, and D. Wolf, Physica A234, 687 (1996).

[14] P. Wagner, Traffic Simulations Using Cellular Automata: Comparison with Reality, in Traffic and Granular Flow, D. E. Wolf, M. Schreckenberg, and A. Bachem (eds.) (World Scientic, Singapore, 1996).

[15] J. Wahle, A. Bazzan, F. Klgl, M. Schreckenberg. Anticipatory Traffic Forecast Using Multi–Agent Techniques. In Traffic and Granular Flow '99, pp. 87–92, eds. D. Helbing, H.J. Hermann, M. Schreckenberg, D. Wolf (Springer, 2000a).

[16] J. Wahle, A. Bazzan, F. Klgl, M. Schreckenberg. Decision Dynamics in a Traffic Scenario. Physica A 287, 669–681 (2000b).

[17] Provided on MySQL documentation, see www.mysql.com for further information.

[18] TRANSIMS Home Page, http://studguppy.tsasa.lanl.gov

[19] Duisburg Online Simulation Home Page, http://www.comphys.uniduisburg.de/OLSIM/