Multiagent Traffic Control System

Francisco Javier Guzmán

Centro de Inteligencia Artificial
Instituto Tecnológico y de Estudios Superiores de Monterrey
Sucursal de Correos "J"
64849 Monterrey, N.L.
MEXICO

May 31, 2007

Abstract

The 20th century witnessed the worldwide adoption of the automobile as a primary mode of transportation.Present-day traffic networks are unable to efficiently handle the daily movements of traffic through urban areas. Multi agent systems are an excellent way of doing microscopic simulation and thus provide possible solutions to the traffic problem. On this study we'll discuss a different approach to simulate traffic networks and optimize them via a traffic light bidding process.

1 Summary

2 Introduction

The car traffic transiting through local urban networks is growing each day. Thus, government and organizations need to put more effort in optimizing this process. It is worthless to talk about network re-configuration given that it is very expensive and it only solves part of the problem. Therefore the principal concern is to optimize the traffic lights.

In this study, we explain the base of a system that simulates traffic at a micro-level (that is at a vehicle level) using multi agent technologies. We also propose new ways of optimizing the traffic light system by a bidding system. We explain the methodology that we had followed so far and the future plans for this project.

3 The Multi Agent based Traffic Simulation

3.1 Traffic Problem

The 20th century witnessed the worldwide adoption of the automobile as a primary mode of transportation. Coupled with an expanding population, present-day traffic networks are unable to efficiently handle the daily movements of traffic through urban areas. Improvements to urban traffic congestion must focus on reducing internal bottlenecks to the network, rather than replacing the network itself. Of primary concern is the optimization of the traffic lights, which regulate the movement of traffic through the various intersections within the environment.

At present, traffic lights may possess sensors to provide basic information relating to their immediate environment. The use of such sensors provides greater flexibility within traffic lights, since more appropriate patterns can be calculated for the current situation. However, these sensors are incapable of monitoring external events, or providing slow and gradual changes to the traffic flow. In addition, sensor information only passes to one specific intersection. As such, neighboring intersections are unaware of outside concerns, and may allow traffic to flow in a detrimental manner. When these problems occur, the processes within a traffic light will remain oblivious. The resulting congestion will eventually overwhelm neighboring intersections, increasing the congestion of the overall network.

Therefore, any solution to the traffic problem must handle three basic criteria, including: dynamically changing traffic patterns, occurrence of unpredictable events, and a non-finite based traffic environment.

3.2 Traffic Simulation

Traffic simulation has been in existence for many years. They all had to use simplified models of traffic flow in order to produce results within practical timescales. A typical assumption is to represent traffic flow in a particular road as a single quantity. Such models are generally called *macroscopic* models, simulation being called macroscopic simulation. Unfortunately, such models do not properly represent real traffic behavior in congested situations, and do not reproduce the fluctuating nature of real world situations. Models that model traffic at the individual vehicle level are called *microscopic* models. Microscopic simulation enables more accurate study of congestion formation and dispersion and emphasizes the insight into the nature of road traffic flow. During each time step, the vehicles are moved towards its destinations, and from road to road if necessary, as in real-life.

3.3 Multi agent based Traffic Simulation

Multi agent systems are an excellent way of doing microscopic simulation and thus provide possible solutions to the traffic problem, while meeting all necessary criteria. Agents are expected to work within a real-time, non-terminating environment. As well, agents can handle dynamically occurring events and may posses several processes to recognize and handle a variety of traffic patterns.

By definition, a multi agent system consists of several agents working in cooperation within a single environment, towards a universal goal. When applied to the urban traffic problem, individual agents provide the computational abilities of specific intersections, while others monitor the overall environment and coordinate actions between intersections.

The coordination between agents is the key to maintaining a balance between optimized events at a global and local level. Equilibrium between the two levels must be achieved to ensure that improvements in one area of the network do not overwhelm and damage other aspects of the same environment.

Although there are several approaches to developing a multi agent traffic system, our model stresses the importance of finding a balance between the desires of the local optimum against a maintained average at the global level. Unfortunately, our system in its actual state doesn't guarantee a global balance. However, local agents are fully capable of determining their own local optimum. Therefore, in the future it will be possible the creation of a hierarchical structure in which a higher-level agent monitors the local agents, and will be able to modify the local optimum to better suit the global concerns.

3.4 Introduction to Multi Agent Systems

Before going deeper in the definition of the MAS we need to define some concepts that are necessary to build such definition. These concepts are the Agent and the Environment.

3.4.1 The Agent

The agent idea it's a very intuitive concept and it can be understood, as a comparison, to a physic agent: let's suppose that a person wants to organize a trip, he (or she) can make telephone calls to different travel agencies, compare prices, consult travel guides, buy an insurance, get foreign exchange, etc. This person could also ask another person – a travel agent- to do this work for him. This agent it's similar to a software agent, only that the later lives and works in a computer.

Thus, a software agent can be considered as an entity intelligent enough to function in an autonomous way, with the only purpose of accomplishing the objectives set by the programmer in the conception stage. Nevertheless, even if the AI (Artificial Intelligence) community can't agree in a definition for the "agent" term, it actually states that an agent differentiates from any other software in its inherent autonomy, given that its behavior is not controlled from the outside. This notion is the key of the MAS (Multi Agent Systems), because all this agents exist in an environment, an independent agent itself, with which they interact.

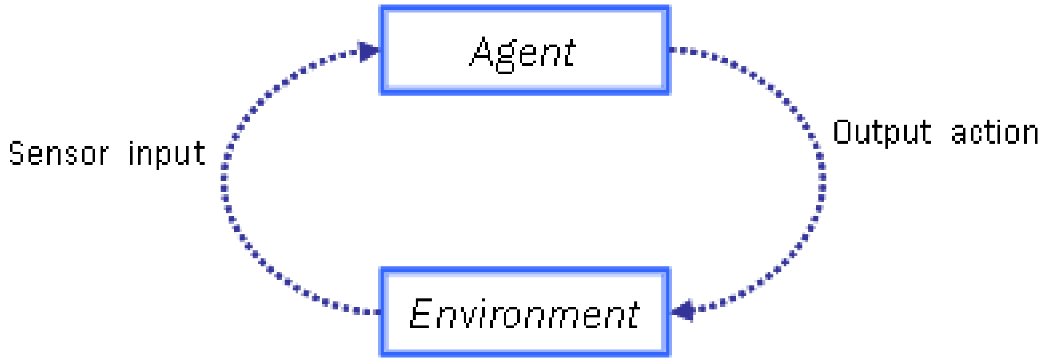


Figure 1: An agent and it's environment

The ability of an agent to modify its environment is limited by the actions that it is allowed to do. Also, there are different kinds of agents:

1. Logical agents: their decisions are made based on their logical deductions
2. Reactive agents: their decisions are made based in a relation between the situations and the actions
3. The BDI agents (Belief-Desire-Intention): the agent decide the actions to do in function of their internal state, expressed under the form of beliefs, desires or intentions
4. Multy level agents: The internal knowledge of this type of agent is organized in different levels of abstraction allowing different levels of treatment.

3.4.2 The environment

The complexity of the agents' decisions is mostly based on the environment's characteristics. The later can be:

1. Accessible: the agent can get complete information and updates of the environment state.
2. Deterministic: an action has a warranted effect on time.

3. Episodic: the agent can't predict the evolution of the environment because the observed events don't show any relation amongst them and they don't correspond to any known rule.

4. Static/dynamic: a dynamic environment has processes that affect its internal state, independently of the agent's control.

5. Discrete/continuous: a discrete environment has a number of actions and perceptions which are finite and fixed.

3.4.3 From the environment to MAS

The concept of MAS is actually an extension of two concepts: it describes a system in which the agents (that interact with each other and with the environment) evolve in order to solve a common problem. But that doesn't mean that the MAS are only a bunch of agents. In fact, their interactions are defined by the use of communication protocols, without the imposition of a specific model: the programming language choice it's open.

Summarizing:

1. An agent it's a reactive piece of software, meaning that it modifies its internal state in function of its environment and its goals.

2. It is proactive, meaning that it is capable, by its own initiative, of doing actions to accomplish its objectives.

3. Finally, an agent is also characterized by its capability to communicate with other agents or even humans, which gives it a social dimension.

If the agent responds to all this criteria of flexibility, then it is called an "intelligent" agent. The intelligence notion is associated to the capacity of a program to perform actions flexible and autonomous, always with a specific objective.

3.5 Choosing a Multi Agent System

In order to properly develop a Multi agent simulation, the first question we had to ask, is which tool will allow us to create a multi agent system that best suits to our needs:

1. Distributable, so we could run our simulation in different machines if needed.

2. Light weighted

3. Flexible and customizable, for our application to grow in time

4. With a Simulation tool, to develop our project.

To answer that question, we started a quest to discover the most known MAS platforms. Here is what we've found out:

3.5.1 NetLogo

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of independent "agents " all operating in parallel. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals. NetLogo has extensive documentation and tutorials.

Anyway, this environment is very limited in the extension of possible applications. Is not as much customizable and agents are based in functional code.

3.5.2 Swarm

Swarm is a multi agent platform with reactive agents. The inspiration of this agent model is inspired on artificial life. SWARM is the preferred tool of the American community and the researchers on artificial life. The environment offers an ensemble of libraries that allow the implementation of multi agent systems with a great number of simple agents that interact in the same environment. Many applications have been developed using SWARM, which is currently available in many languages (Java, Objective-C). Even if this is a much concurred platform, it didn't suit for our needs, given that we wanted our application to be flexible and to grow with time, also to be distributable. And that is a functionality that swarm does not provide.

3.5.3 MadKit

MadKit is a Java multi-agent platform developed by the " Laboratoire dÍnformatique, de Robotique et de Microélectronique de Montpellier (LIRMM) " from the Montpellier University. It was built upon an organizational model (Aalaadin). It also provides general agent facilities (lifecycle management, message passing, distribution), allows high heterogeneity in agent architectures and communication languages, and various customizations. MadKit communication is based on a peer to peer mechanism, and allows developers to quickly develop distributed applications using agent principles.

MadKit is free software based on the GPL/LGPL license. Even if there is not a lot of information available for this platform (let's say it's not as popular as the others), it suited to our needs, given that it is distributable, it has a very flexible model, is written in java (which makes it very customizable), and comes with a lot of libraries (including a Simulation tool) which made it the best option to develop our project.

Those are the reasons why we chose to start developing our application with MadKit. Even it doesn't have a lot of documentation or tutorials to aid the newbies to get the feeling of this environment, it is developed in Java, which makes it a lot easier to use.

Note that we did search other platforms, but since we didn't have the time to test them all, we decided to stick to MadKit.
To better understand how this platform works, in the following page's we'll discuss the Aalaadin model, and MadKit's architecture.

3.6 The Aalaadin Model

Under their thesis project Olivier GUTKNECHT and Jacques FERBER developed a generic meta-model of a multi-agent system. This model called Aalaadin allows simply describing organizational and interactive structures within a multi agent system. These structures allow the Aalaadin model to answer to the heterogeneity problems, frequent among the SMA. Therefore, it is convenient to go deeper in the fundamental principles of this model.

The organizational structure of the meta-model Aalaadin is based on three principal concepts: the agent, the group and the role. This way of modeling systems allows Aalaadin to successfully split every entity of a system.

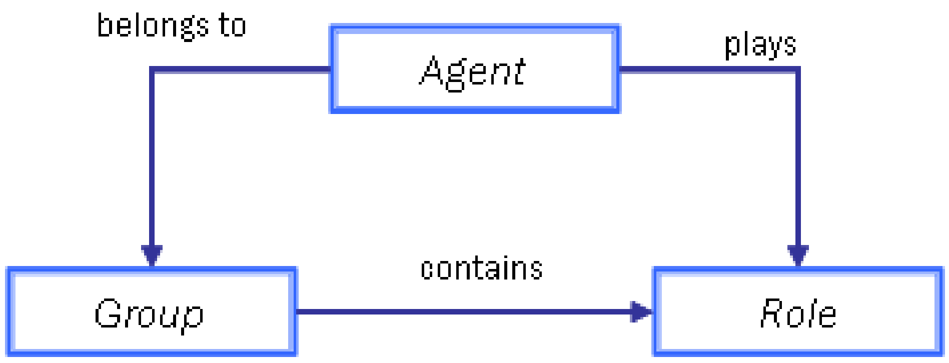


Figure 2: The AGR Model

3.6.1 Agent

The architecture and the behavior of an agent aren't based on a particular model. The Aalaadin model only defines agents as autonomous entities able to communicate, playing a role within a specific group. Aalaadin grants little semantic to an agent, fact that allows in one hand to define entirely its behavior during conception phase, and in the other hand, not to restrict the definition of an agent under deterministic criteria. These criteria would, again, trigger the discussion about the true definition of an agent.

3.6.2 Group

This model it's based in a very simple concept: the group notion. As its name indicates, it allows to assembly and to classify agents in order to divide the system into micro systems (system atomization). A group is defined by the way it communicates; also the belonging to a specific group would be constraint to some conditions as it is the knowledge of a particular language. Even that the group notion identifies agents within a system; it doesn't stop them from belonging to different groups.

3.6.3 Role

Aalaadin doesn't identify an agent based only in the group of which they are members but also it identifies it based in the function they have assigned to accomplish. Therefore, a role is an abstract representation that identifies an agent inside its group. Nevertheless, one agent can have different roles or the same role can be given to different agents. Usually, the group manager is the group creator. If the requirements of a role are not met, the manager will reject the request. These requirements may vary from a specific ability, structure or the group state.

These three concepts give Aalaadin a social dimension which is less remarkable in other agent oriented models.

3.7 MadKit: the concretization of Aalaadin Model

In order to implement Aalaadin Model, a tool was required. This tool should be capable of correctly adapt to any model of agent in whichever domain of application. It should also be customizable as much as possible in order to adapt to different needs

Contrary to most multi agent platforms, which were conceived under the " Zeus " agent model or a specific domain of application (simulation, mobility, etc...), the MadKit (Multi-Agent Development Kit) was conceived under Aalaadin model, which doesn't care about an agent's internal architecture but the organizational aspect.

That's one of the principal originalities of this platform, given that it is based on the model that it intends to implement. In fact, MadKit uses agent groups to structure its platform.

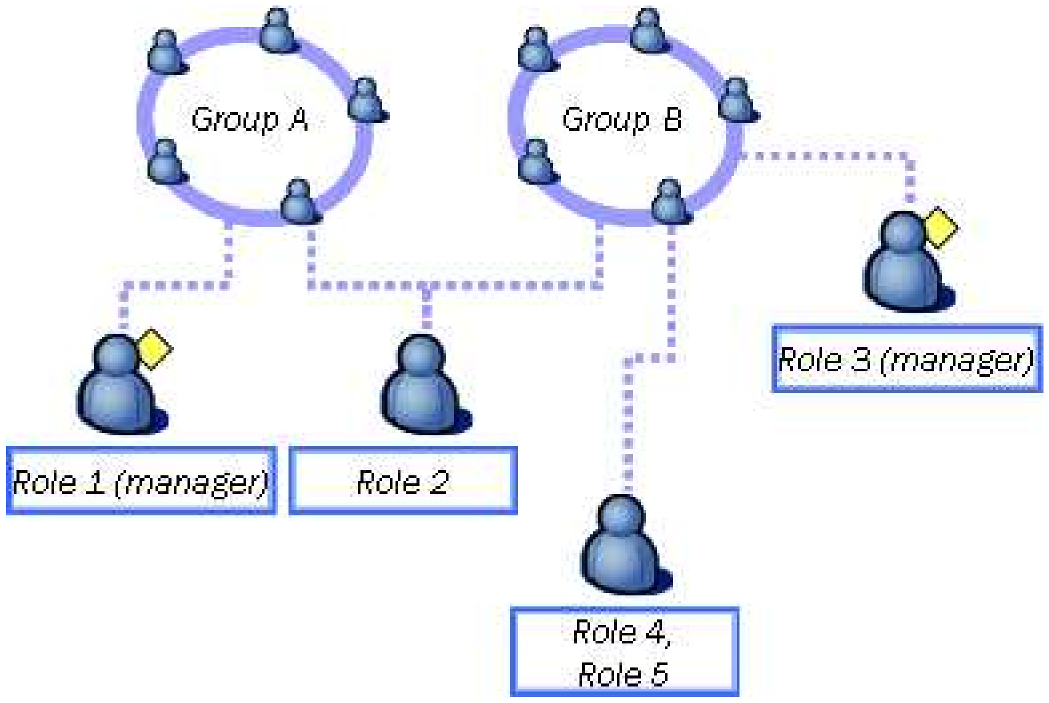


Figure 3: An agent may belong to several groups, and even play different roles on each one. Accordingly, each group has to have only one manager

3.7.1 Base classes in MadKit

3.7.1.1 The agents

A MadKit agent extends the AbstractAgent class which defines the basic functionalities:

Life Cycle:

Each agent has for possible states (creation, activation, execution, destruction). It also has the possibility of launching other agents in the local platform.

Communication:

The communication is set under asynchronous mode. Each agent has a mailbox identified by the AgentAdress. An agent can send a message to another particular agent by this mean or it can broadcast the message to any agent from a group which has a specific role.

Organization:

Every method that allows the organization of agents has already been defined. An agent can observe the local organization (existence of groups or roles) and react (foundation of a group, request of subscription to a "group/role").

Interface:
An agent can define entirely its GUI.

3.7.1.2 Messages

A message should extend the Message class which simply defines the sender/receiver notions. Some message types had already been defined as such, as objects or even as common interoperability protocols (XML, KQML).

3.7.1.3 Synchronous Engine

To simplify the development of autonomous agents, the base class is declined into a class (Agent) which associates a thread to an agent. An application such as a simulation needs the concurrent execution of a great number of agents, what can result into a synchronization problem. The synchronous engine is based under the principle of using external agents to control the order (Scheduler) and their execution (Activator). Another kind of agent (Watcher) allows spying the properties of the synchronized agent by the means of probes to have a global vision of our SMA

3.7.1.4 Service agentification

Any other functionality offered by MadKit which are not managed by the Kernel, is supported by an agent (i.e. the communication in distributed mode). This service agentification allows a great modularity.

Summarizing MadKit it's an ensemble of java packages that implement all the later functionalities. This language has been chosen given its portability, its great number of base classes and by the Beans, a mean good of standardization.

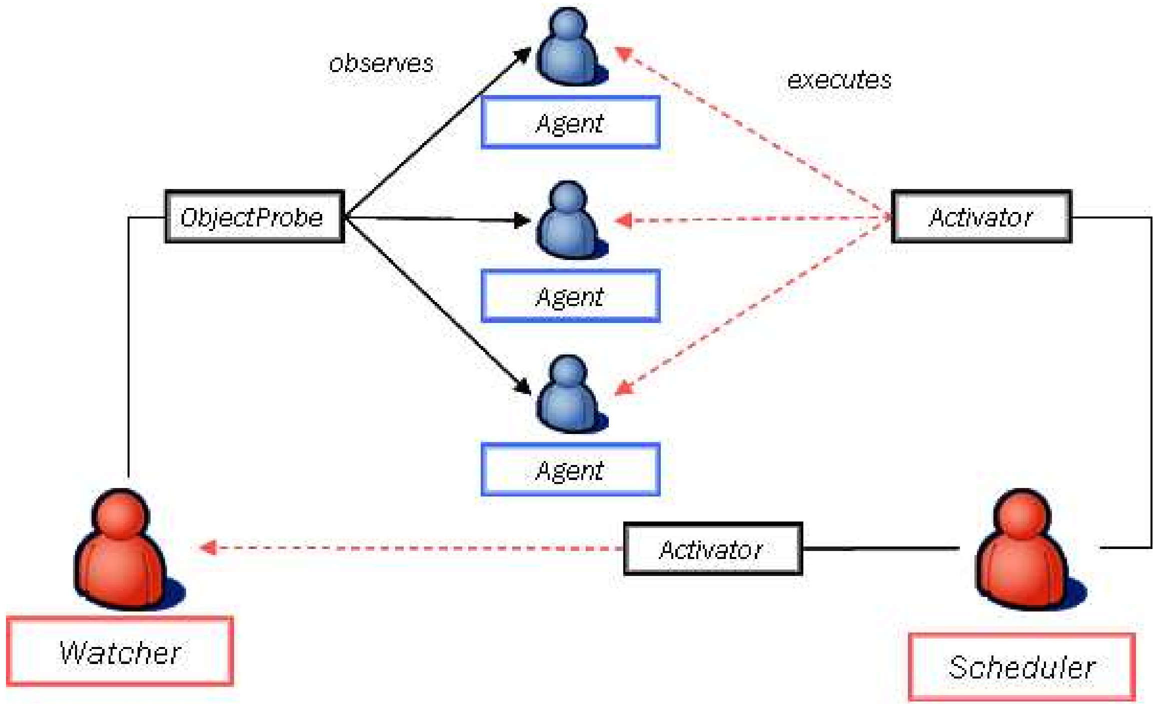


Figure 4: The synchronous engine architecture

3.8 Software Architecture

3.8.1 The Synchronous Engine

To implement our traffic simulation, we stuck to the Synchronous engine model of MadKit. The overall System Architecture is represented by the next figure:

3.8.1.1 Starter The starter class is the one in charge of the synchronization of the simulation. It also launches the principal agents. It has associated two GUIs, which will be discussed later. Its principal method is start, an infinite loop that uses Activators to execute agents.

It is also charged for the save and restore of simulation configuration files using XML.

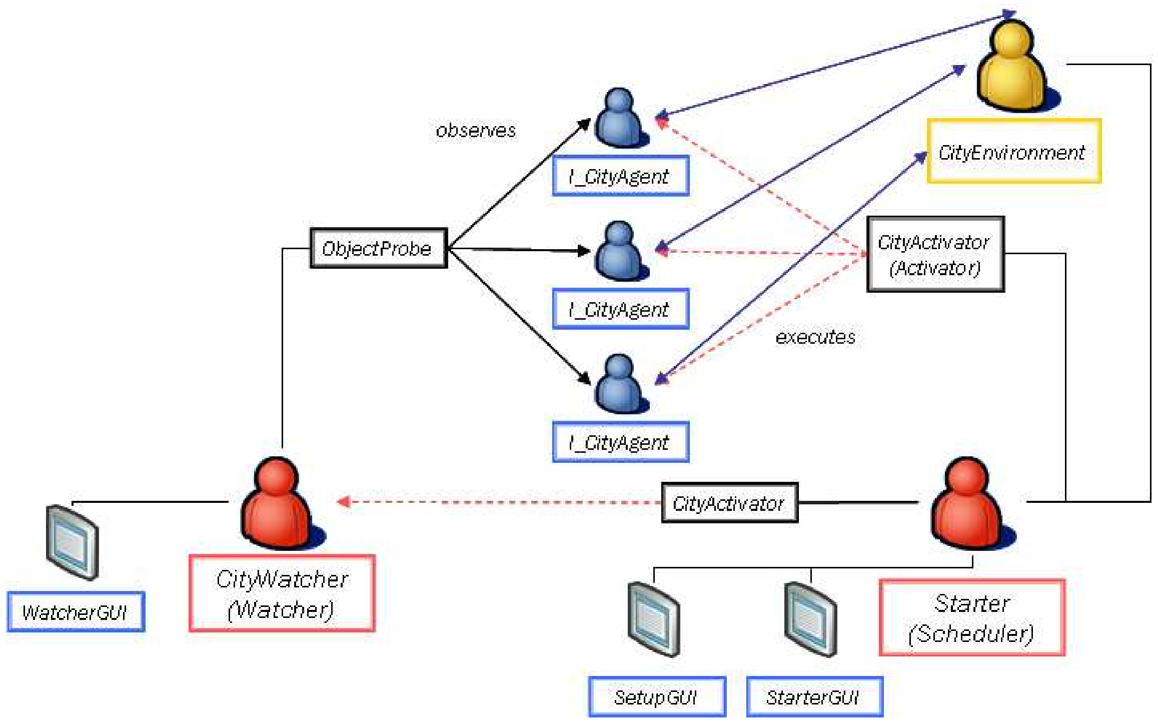


Figure 5: Our flavored implementation of the synchronous engine

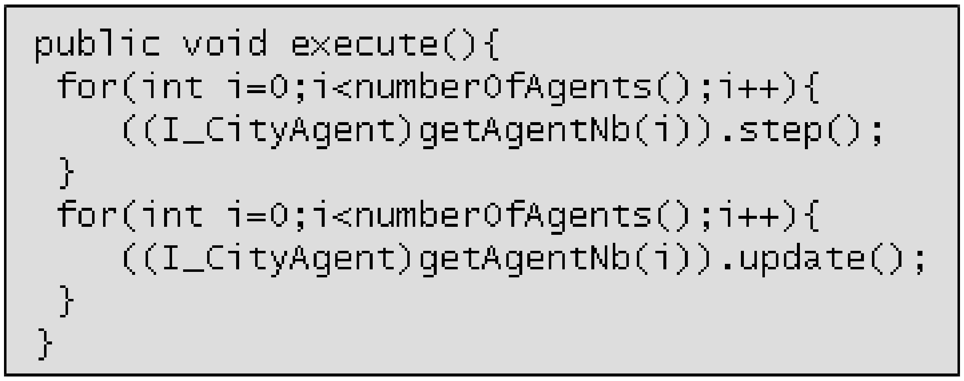
3.8.1.2 CityActivator This tool class defines a basic scheduling policy. It works in conjunction with the scheduler to get a list of schedulable agent. An activator is configured according to a group and a role. On operation, it dynamically discovers the implementation classes of the agents having the given group and roles. These direct references can be used afterwards to directly invoke operations on the agents. Subclasses have to implement the abstract operation, which might be invoked by a Scheduler agent.

3.8.1.3 CityWatcher The principle of the {Watcher agent resembles the Scheduler one. A Watcher manages a list of probes, and combines their input in something meaningful. But in contrast with the scheduler, a watcher is not a threaded agent and has to be executed from the outside, with an associated scheduler. It has also has an associated GUI.

```
public void start(){
 println("Lifecyle of the starter");

 a1= new CityActivator("traffic","car");
 CityActivator a2= new CityActivator("traffic","source");
 CityActivator a4= new CityActivator("traffic","crossway");
 SingleMethodActivator a3= new SingleMethodActivator("observe",
                                               "traffic","observer");
 CityActivator a5= new CityActivator("traffic","lights");

 addActivator(a2);
 addActivator(a1);
 addActivator(a3);
 addActivator(a4);
 addActivator(a5);

 while(!dead){
  pause(100);
  if(!paused){
    a4.execute();
    a5.execute();
    a2.execute();
    a1.execute();
    a3.execute();
    update();
    }
  }
pause(3000);
}
```

Figure 6: Code showing lyfecicle of the scheduler agent (starter)

3.8.1.4 The Probe class The Probe tool class is the basic code for exploring code of a ReferenceableAgent agent. It is configured with a group, and a role. On update( ) operation, it dynamically discovers the implementation classes of the agents having the given group and roles. These direct references can be used afterwards to directly invoke operations on the agents. Subclasses have to implement the actual probing mechanism. This mechanism is close to the probe mechanism implemented in other platforms, like Swarm or Cormas.



```
public void execute(){
 for(int i=0;i<numberOfAgents();i++){
    ((I_CityAgent)getAgentNb(i)).step();
 }
 for(int i=0;i<numberOfAgents();i++){
    ((I_CityAgent)getAgentNb(i)).update();
 }
}
```

Figure 7: Code showing execution of an activator agent

3.8.2 The Agents

The Agent architecture that we've implemented looks like the following:

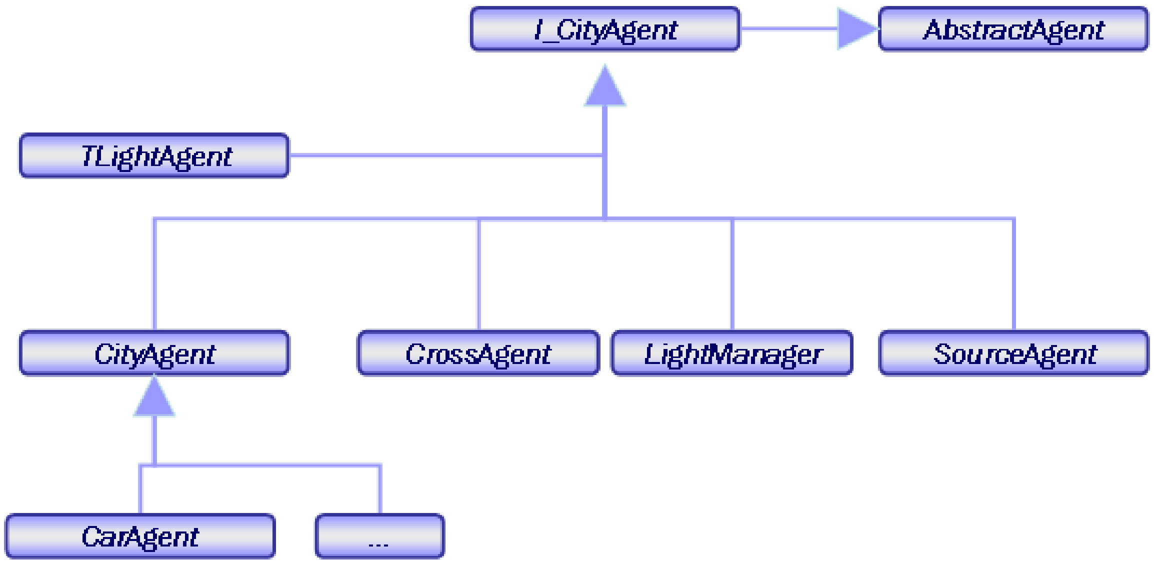


Figure 8: Agents' architecture

Most agents extend (inherits) from I_CityAgent class. This configuration allows treating most agents transparently, given that Java doesn't allow multiple inheritance.

3.8.2.1 I_CityAgent This abstract (virtual) class extends the AbstractAgent behavior. It concentrates the method common to most agents such as step( ), update( ), end( ) which simplifies the implementation of methods where the nature of the agent is not important.

3.8.2.2 LightManager This agent is in charge of the coordination of lights from every light. It has associated a GUI that helps configuring start and finish times of lights.

```
public void step(){
 TLightAgent k;
 if(wait==compt){

  for(Iterator it=lights.iterator();it.hasNext();){
     k= (TLightAgent)it.next();
     if(k.getStart()==cursor)k.setColor(TLightAgent.T_GREEN);
     if(k.getFinish()==cursor)k.setColor(TLightAgent.T_RED);


  }

  if(cursor==cycle)restart();
  else cursor++;

  compt=0;
 }
 else
 compt++;
}
```

Figure 9: Code showing the main behavior of a LightManager

3.8.2.3 Source Agent This agent controls the creation of CarAgents, which are created at a λ rate. This rate is communicated to the TLightAgents in order to make the proper statistic treatment. It has associated a GUI that helps to configure the creation rate as well as the turning direction for the cars created by this source.

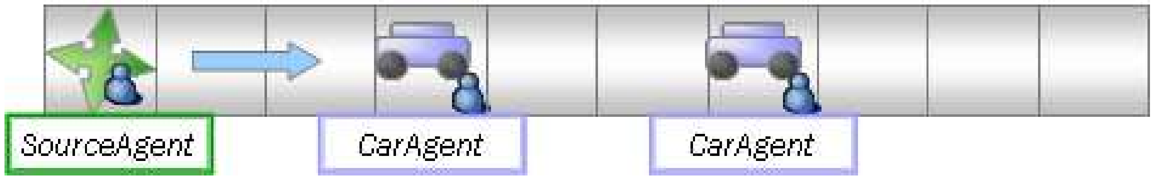


Figure 10: A source agent in action

3.8.2.4 CrossAgent This agent's only purpose is to setup the crossways during the setup phase.

3.8.2.5 TLightAgent This agent has several internal states that represent the different colors possible in a traffic light. For this version, the only colors available are red and green. This agent also makes a statistical treatment in order to improve the traffic flow. The way it communicates with other light agents is via XML messages.

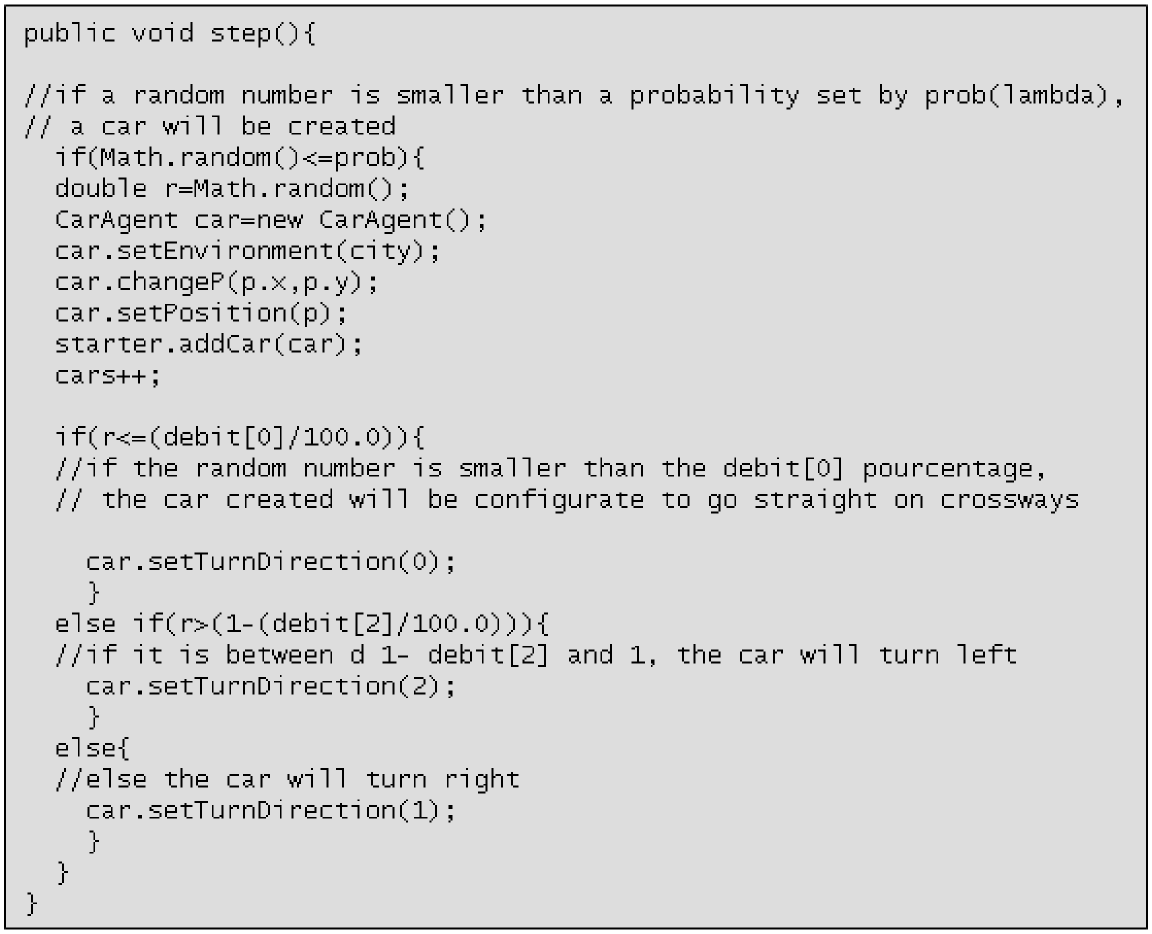
3.8.2.6 CityAgent This agent class is an generalization of the CarAgent class which centralizes the common behavior of any vehicle. This class has been implemented in order to permit further enhancement of the model by including different vehicles other than cars.

3.8.2.7 CarAgent This agent is perhaps the most important agent in the simulation. The actual functionalities of this agent are to accelerate, decelerate, turn, and stop. This agent is purely reactive. It obtains its data by communicating with the environment. It is aware of other cars, crossways and traffic lights.

The model that rules this agent is based on a linear acceleration approach. For this, the car increases its speed constantly until it reaches the maximal speed allowed by the street on which it is traveling.

Another enhancement for this model would be to include an aggressiveness factor, which would allow each car to behave differently from the others, by speeding, over passing other cars, etc.

This agent also keeps track of its performance, so the user could analyze the simulation's output data.



```
public void step(){

//if a random number is smaller than a probability set by prob(lambda),
// a car will be created
  if(Math.random()<=prob){
  double r=Math.random();
  CarAgent car=new CarAgent();
  car.setEnvironment(city);
  car.changeP(p.x,p.y);
  car.setPosition(p);
  starter.addCar(car);
  cars++;

  if(r<=(debit[0]/100.0)){
  //if the random number is smaller than the debit[0] pourcentage,
  // the car created will be configurate to go straight on crossways

    car.setTurnDirection(0);
    }
  else if(r>(1-(debit[2]/100.0))){
  //if it is between d 1- debit[2] and 1, the car will turn left
    car.setTurnDirection(2);
    }
  else{
  //else the car will turn right
    car.setTurnDirection(1);
    }
  }
}
```

Figure 11: Code for a source agent

3.8.3 The Environment (CityEnvironment)

The environment describes a collection of states. It is represented by a matrix of N x M cells of size C. It is in charge of the creation of streets, the accommodation of crossways, traffic lights, etc. For doing this it is organized on different layers of information. On the first layer, we find the sNetwork, which is a collection of streets, their directions and software. This grid allows cars to navigate.

The second layer (grid) is made out of CarAgents. It contains position of cars as well as their reference. This grid allows a car to meet its neighbors.

The last layer (kgrid) is made out of SourceAgents and TLightAgents.. This grid allows cars to look for traffic lights and to regulate their speed.

Even if the grid and kgrid layers manage the same kind of data (I_CityAgent), we decide not to make only one layer but two. This will diminish the task of class verification and the chances of having a ClassCastException.

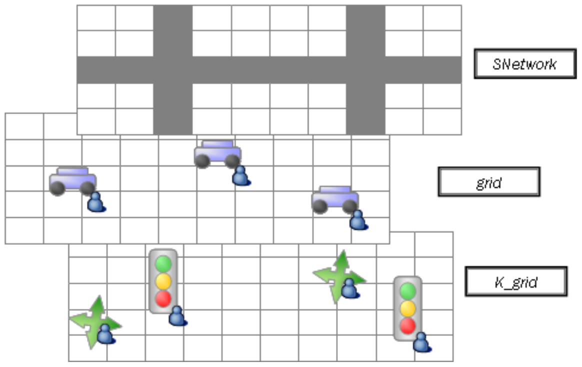


Figure 12: Environment constitution

3.8.4 The GUI Architecture

To have a better feeling of what's happening in our simulation, it was necessary to build a Graphical User Interface. The fact that our simulation is completely customizable is in fact achieved to this part of our architecture.

In MadKit any graphical interface is embedded into the main program's frame. So, any component may inherit from JPanel. Therefore, our architecture looks like figure:13

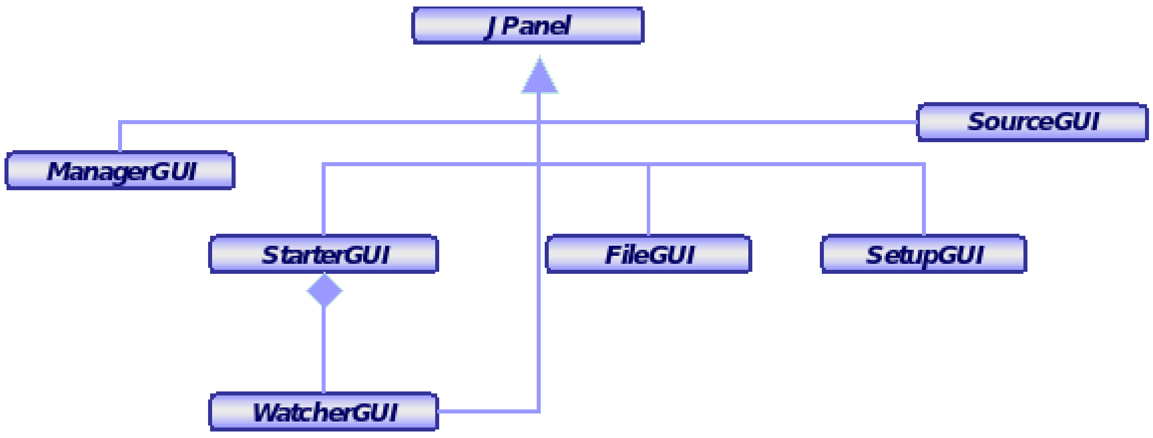


Figure 13: GUI Architecture

Note that any agent may load its own graphical interface, as is the case of WatcherGUI and StarterGUI by the method.

```
public void initGUI(){
  gui = new StarterGUI(this,envsize*cellsize,envsize*cellsize);
  setGUIObject(gui);
```

Figure 14: Code showing InitGUI Method

On the other hand, we've implemented an agent that is in charge of loading interfaces dynamically. This Dialog Agent is launched by the Starter and its only purpose is to launch panels and manage their events.

Now, we'll do a brief description of each Panel.

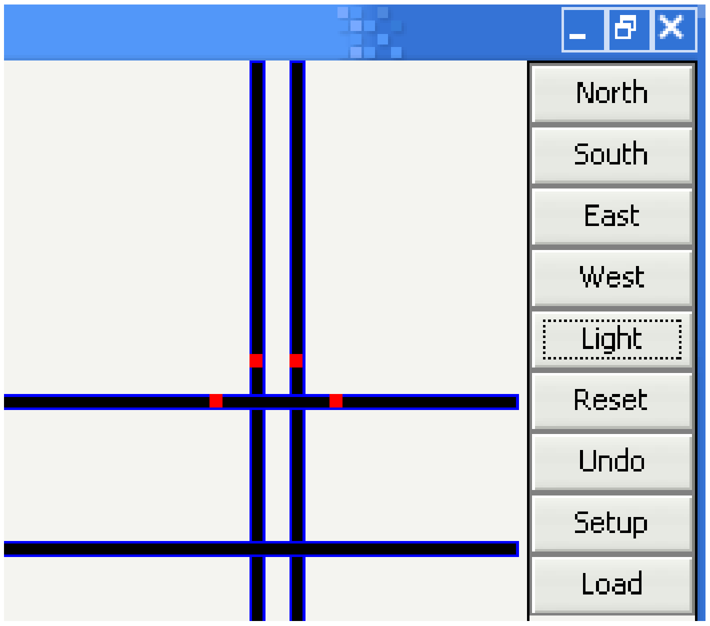


Figure 15: SetupGUI

3.8.4.1 SetupGUI This component helps the user to setup a scenario by drawing streets, or by the setup of lights. With this interface you could also load a saved file. If anytime you make a mistake, don't worry: it has an undo function.

3.8.4.2 StarterGUI This component acts as a "player" of our simulation. It instantiates a WatcherGUI which allows us to see the state of our simulation. It also allows us to setup different sources and format traffic light times

3.8.4.3 WatcherGUI As we said previously, this component "spies" the state of the system and displays it.

3.8.4.4 SourceGUI This component allow us to modify a source's car spawn probability and also the percentage of cars that will turn in different directions

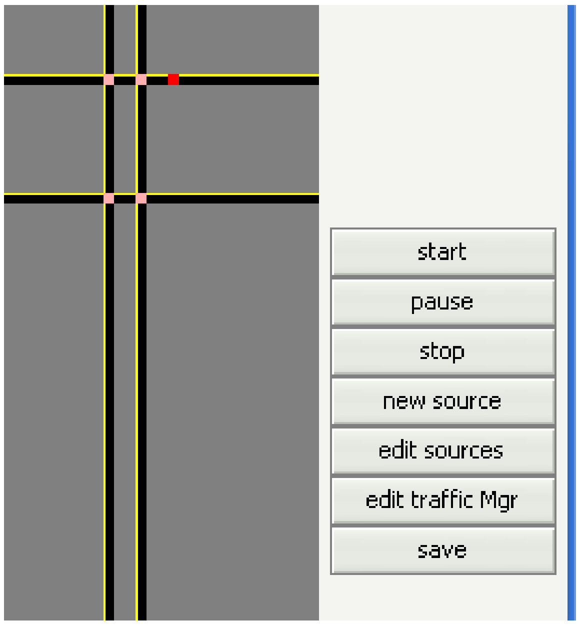


Figure 16: Starter GUI

3.8.4.5 ManagerGUI This component allows configuring every traffic light present in the system.

3.8.4.6 FileGUI This component allows the user to load or save previous configurations.

In the next section we'll se how this happens.

3.8.5 Save and Restore

For our application to be easier to use and to be capable of loading any previous configuration done, we've implemented the Save and Restore functionality making use of the JDOM libraries of XML.

The way it works its easy, once the Setup stage finished, the user is able to save the scenario he has just created. A XML Document is created by the Starter class and then it passes it to each Class so they can save their important data by the function Serialize( );

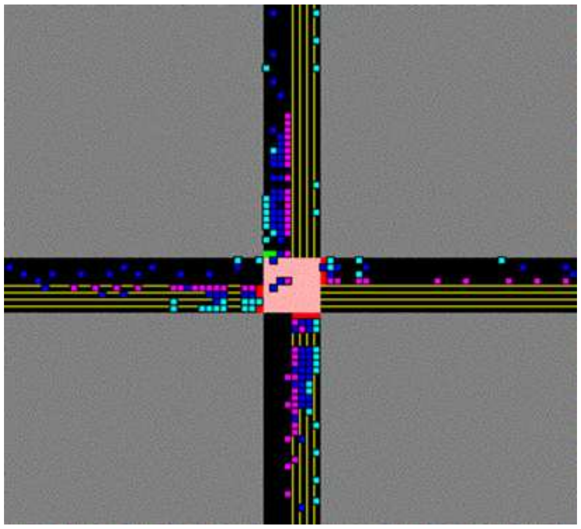


Figure 17: Watcher GUI

But if a user has already stored a Document, it can during the Setup stage, load it and restore all the data he had archived. It works similarly to the Serialize function, but this time, the Starter Builds up an XML Document based on the raw data from the file opened, and it restores classes depending on the content of that Document.

3.8.6 Messaging

In order to improve communication between agents, we implemented a different XMLMessage class than the provided by MadKit. This XMLMessage is also based on JDOM libraries. The advantage of this tool is that we can send classified information and not only Strings.

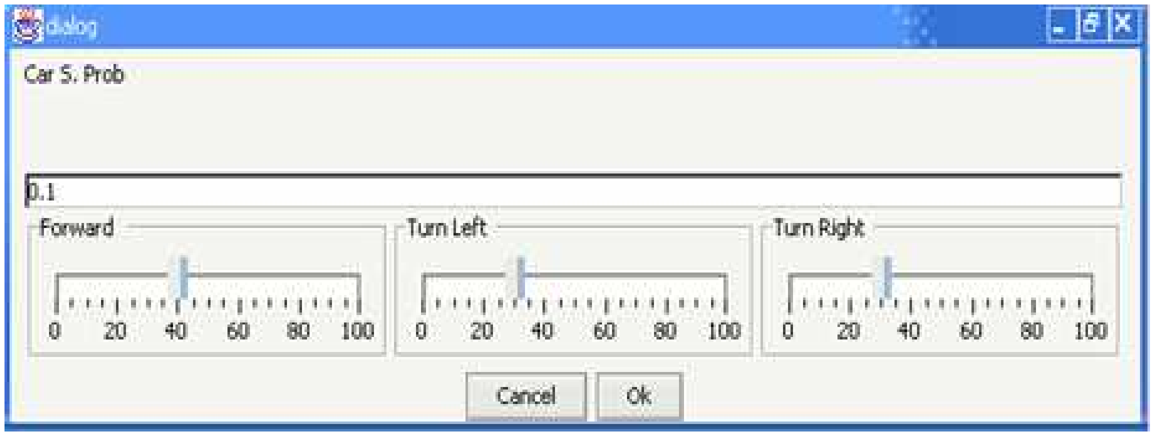


Figure 18: Source GUI

3.9 Traffic Light Bidding System

3.9.1 Optimizing utilization

Let's suppose a scenario: We have four streets: one in each direction, and each one holding. At the beginning of each street we have a source agent and just before the crossway, a traffic light. Let's also consider that each light's start and end times are independent one of each other.

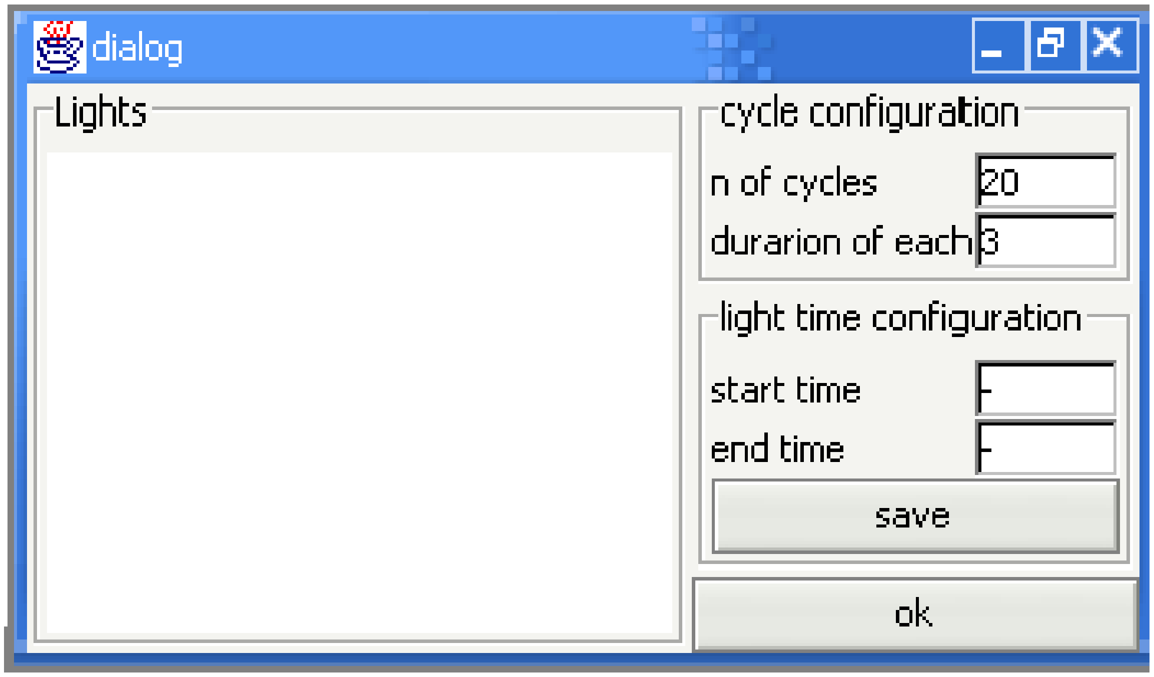
Then, each lane of this system could be considered as an M/M/1 queuing system with an arrival rate of λ, one server (the traffic light) and a departure rate of μ. (crf. Appendix A)

In this traffic system, we're interested in minimizing the time a car has to wait in order to be served (advance).If we recall the mean waiting time can be written as:

$$W = \frac{1}{\mu - \lambda} = \frac{1}{\mu(1-\rho)}$$

Where λ can is a parameter of simulation (a characteristic of each source). Anyway, to determine μ it's a little more complicated.

Let's suppose a depart rate distribution from one light as described in figure 23.



n of cycles 20
durarion of each 3
start time
end time

Figure 19: Manager GUI

Where T is the total light cycle time, M(t) represents the quantity of cars that are served by a light by time step, μ_c is the maximal amount of cars that one light can serve on a time step and τ is the amount of time a light remains on service par cycle.

Then, average depart rate can be calculated as:

$$\bar{\mu} = \frac{1}{T} \int_0^T M(t)\,dt \simeq \frac{1}{T} \int_{start}^{finish} \mu_c dt$$

which results in

$$\bar{\mu} = \frac{\mu_c \tau}{T}$$

then we can write the utilization rate as:

$$\rho = \frac{T\lambda}{\mu_c \tau}$$

from here we can observe that (what is obvious) the utilization rate of a light will be smaller as the greater is its service time τ.

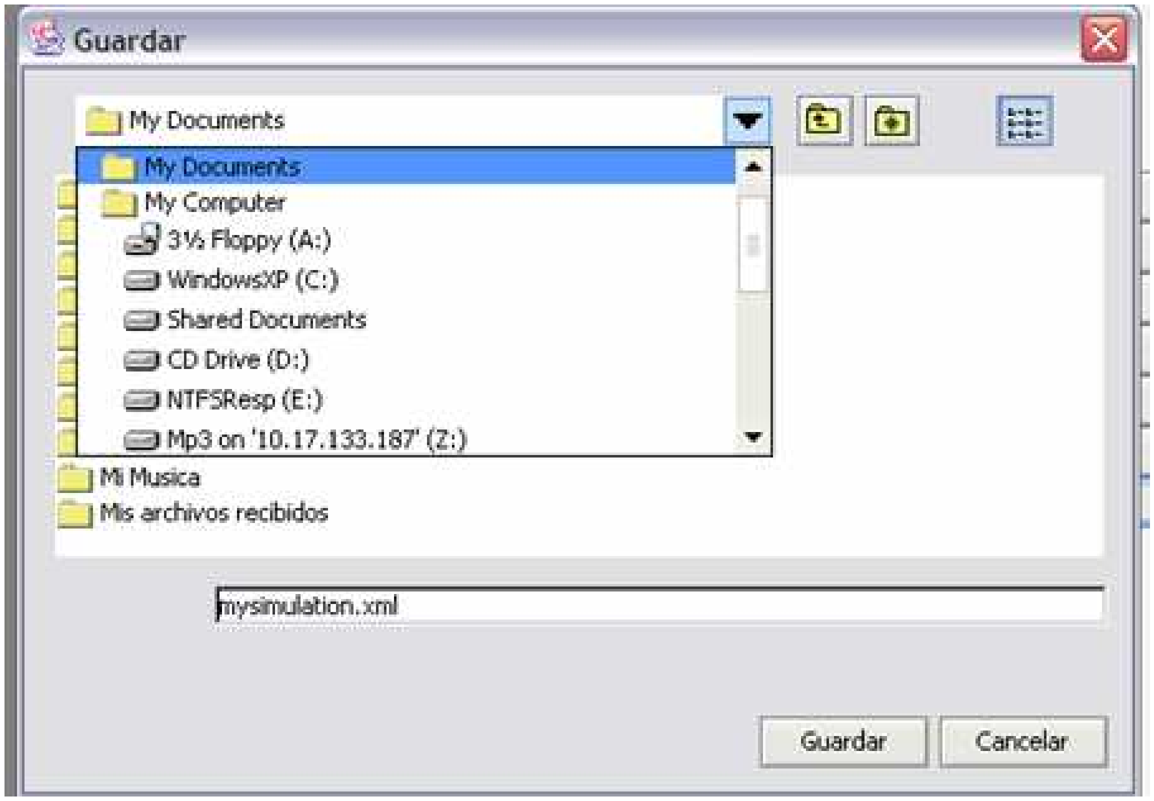


Figure 20: Save/Load Dialog

3.9.2 The bidding system principle

In order to diminish the amount of time that a car may wait in a red light, we introduced a bidding system, in which each light is continuously checking its utilization rate ρ .If this rate is greater than a "tolerance" rate (ρ_c) then it sends a message with subject "open_bid" to all the other traffic lights in the neighborhood asking them for the time that they may not need, and sending them the name of the requester. It also creates a new group called bid_group, in which this demanding agent is the manager and has the role of owner.

Once the other agents receive the message that a bid is open, they subscribe to the bid as bidders. They propose a time quota, in function of their actual state. That quota needs to be the greater enough to be considered by the owner as the best proposition but small enough to leave the light agent still operational (that means $\rho > \rho_c$).

After an agent has calculated the bid time (time quota) then it sends a message to the owner under with the subject "reply_bid" which contains the amount of time proposed and

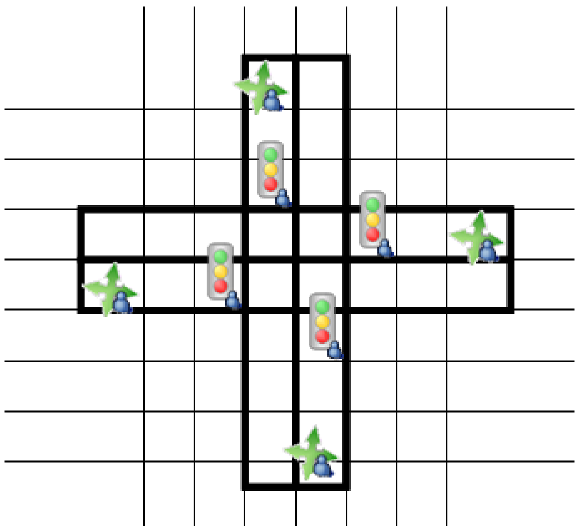


Figure 21: Baseline scenario

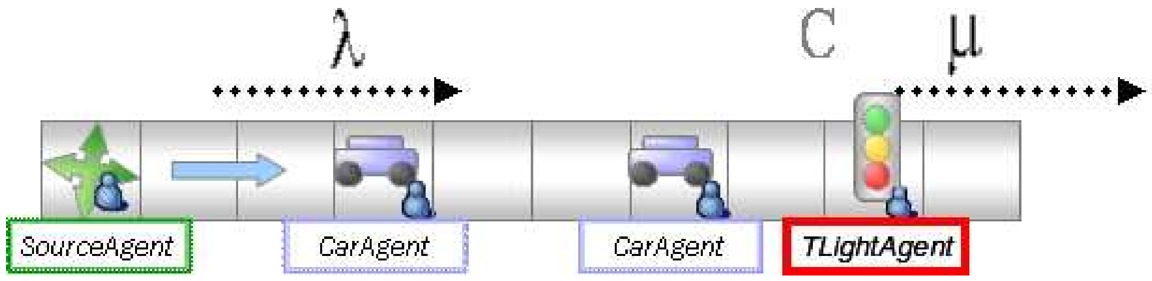


Figure 22: M/M/1 queue

the address of the sender, for the owner keeps track of the bids and keeps only the best one by the function makeContest( ).

A Light time cycle passed, the owner closes the bid, announcing to the whole group the winner, the amount of time proposed by the winner and start and stop time references in for the bidders to update their times.

The time updates are done in the following way:

Any light agent whose start and end times are between the Zone A limits, that is between the owner's finish time and the winner's start time, will remain unchanged. In the

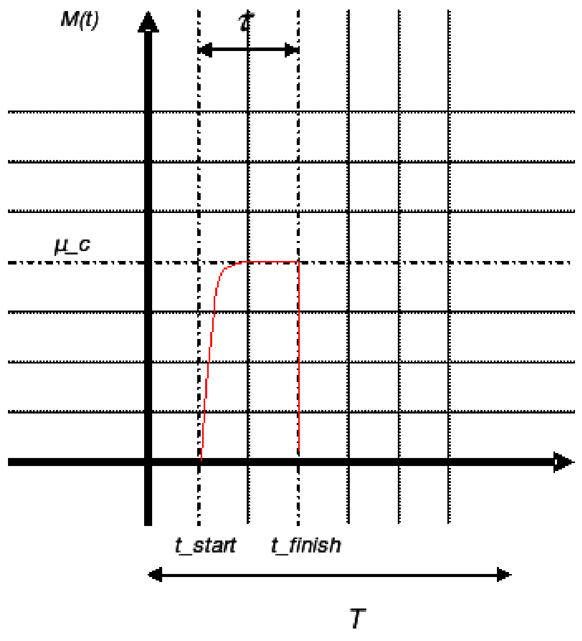


Figure 23: Departure rate for a given traffic light

other hand, agents whose start and end times are found in Zone B, will be decaled to the left an amount of time τ.

Note that this is true under any time conditions since the traffic time cycle is treated as circular (toroidal). The only condition to be accomplished is that start and finish times of different agents be overlapped.

3.9.3 Results

During the experimentation phase of our model, we have been only able to test one kind of scenario: when only one light is congested. In order to test other kind of scenarios and since the bidding system is synchronous, we would have to make a reservation mechanism, in which a bidder wouldn't be able of offering the same amount of time to different bid owners.

Here we have some results given by our bidding system. We have a scenario where one light is congested (source λ =0.3), and the other lights are almost free (λ =0.01). Under

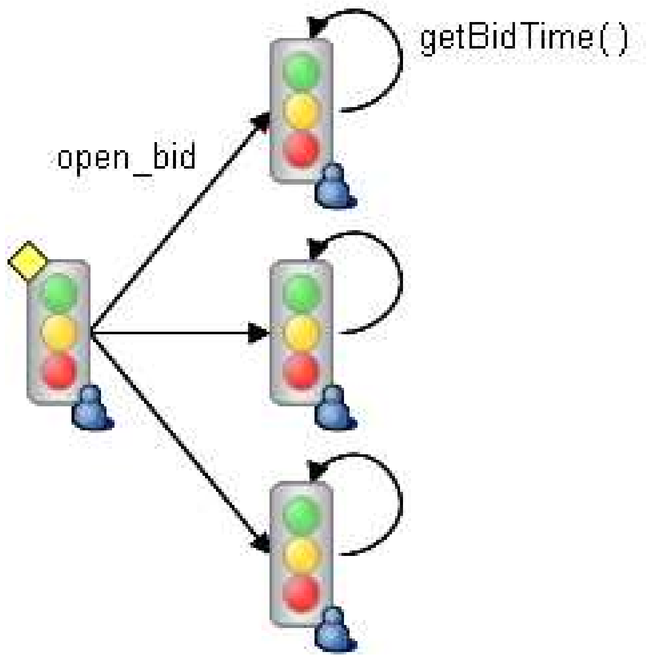


Figure 24: Auction announcement

this scenario we could see that we had a lot of response from the bidders. That made the system converges to a solution.

To even think of implementing this system in a real world situation, we would have to make a more exhausting calibration that would allow us to demonstrate that this system would be effective in such situations.

Anyway, this system has a weakness; it is restricted to systems where we have an opportunity of optimizing (at least one lane is less congested than the others). Otherwise, the lights would ask indefinitely for extra time to their neighbors but no positive answer will come as they are congested as well.

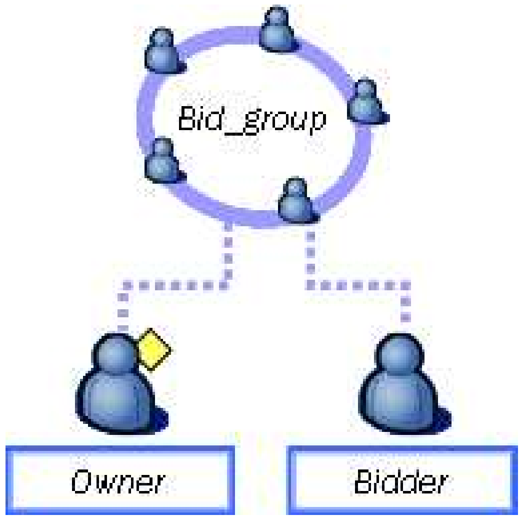


Figure 25: Bidding group

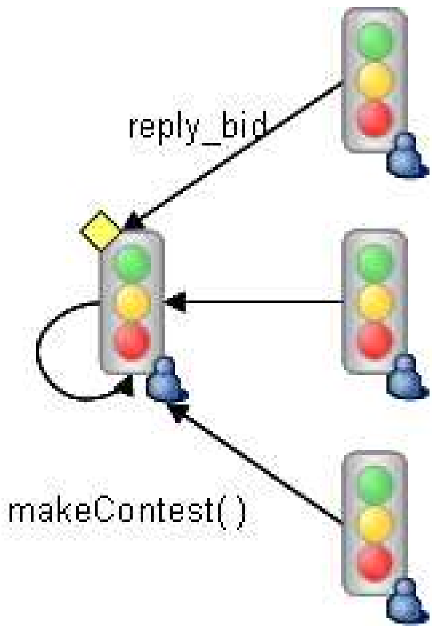


Figure 26: Other agents place a bid

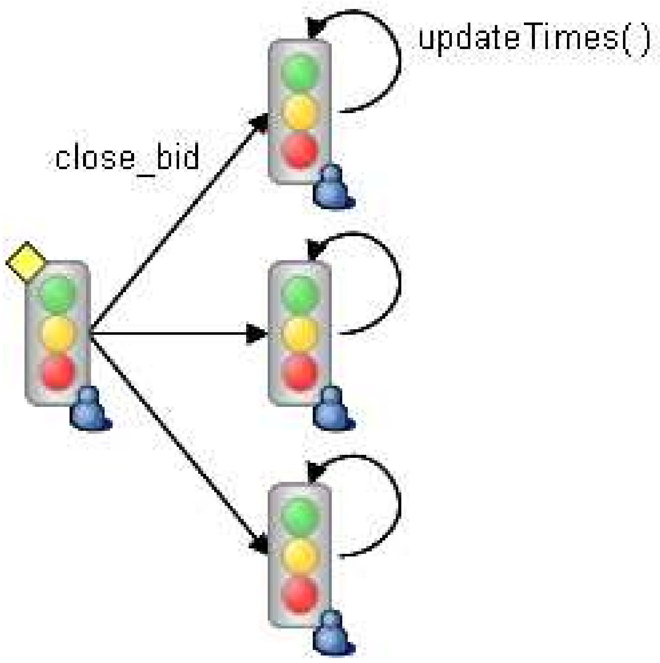


Figure 27: Auction conclusion

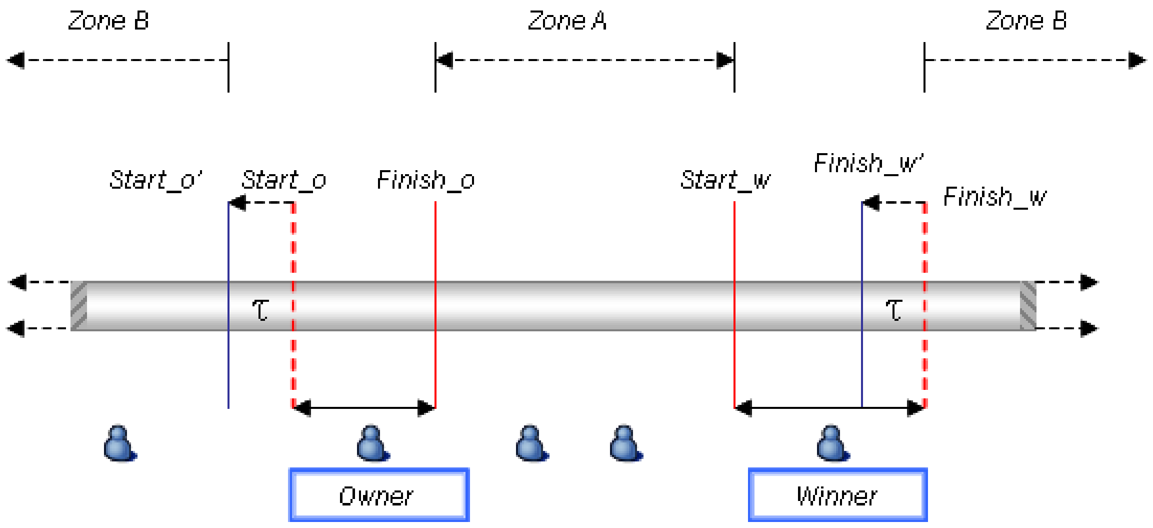
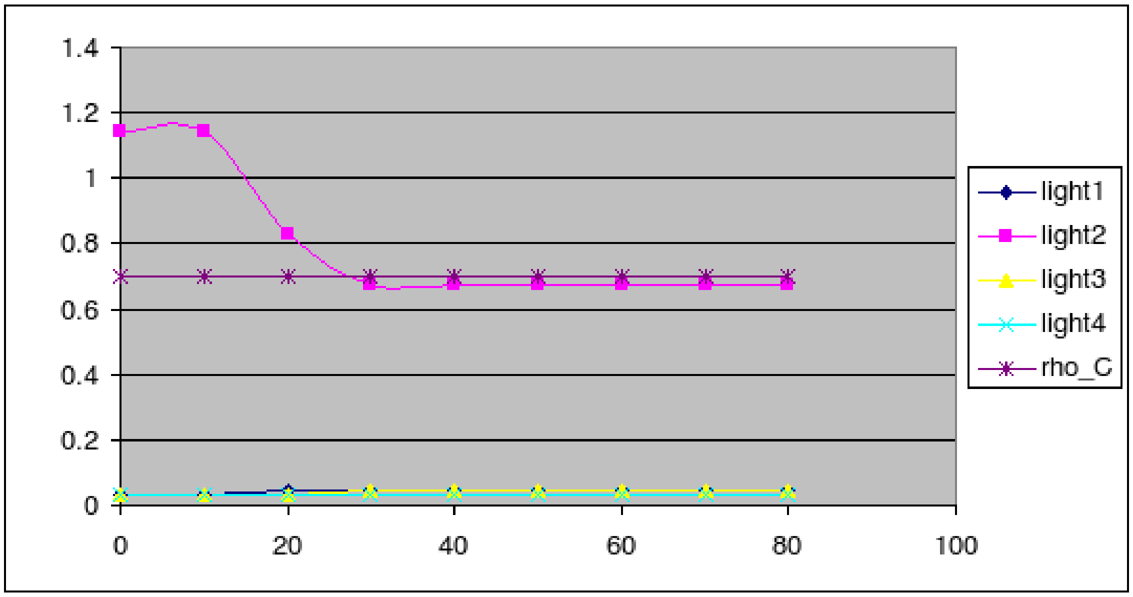


Figure 28: Time update schema


1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
20
40
60
80
100
light1
light2
light3
light4
rho_C

4 Methodology & Experimentation

To successfully build a Simulation, we needed to stick to a methodology. First we defined our system: urban traffic, more specific Monterrey's metropolitan area's traffic. Then, we had to make observations about how was Monterrey traffic behaving. So, we chose a specific crossway in the city, we analyzed it and we modeled the behavior of cars, traffic lights, and "virtual sources" (that's because, in reality, the sources that we modeled and that we have explained before do not exist, but simplify the treatment). Then we made some assumptions (i.e. traffic is constant at determinate time, cars respect speed limits, etc). A then we developed our application using Multi agent technology. Finally, compared the results out of our simulation to those we observed, and made corrections to our model.

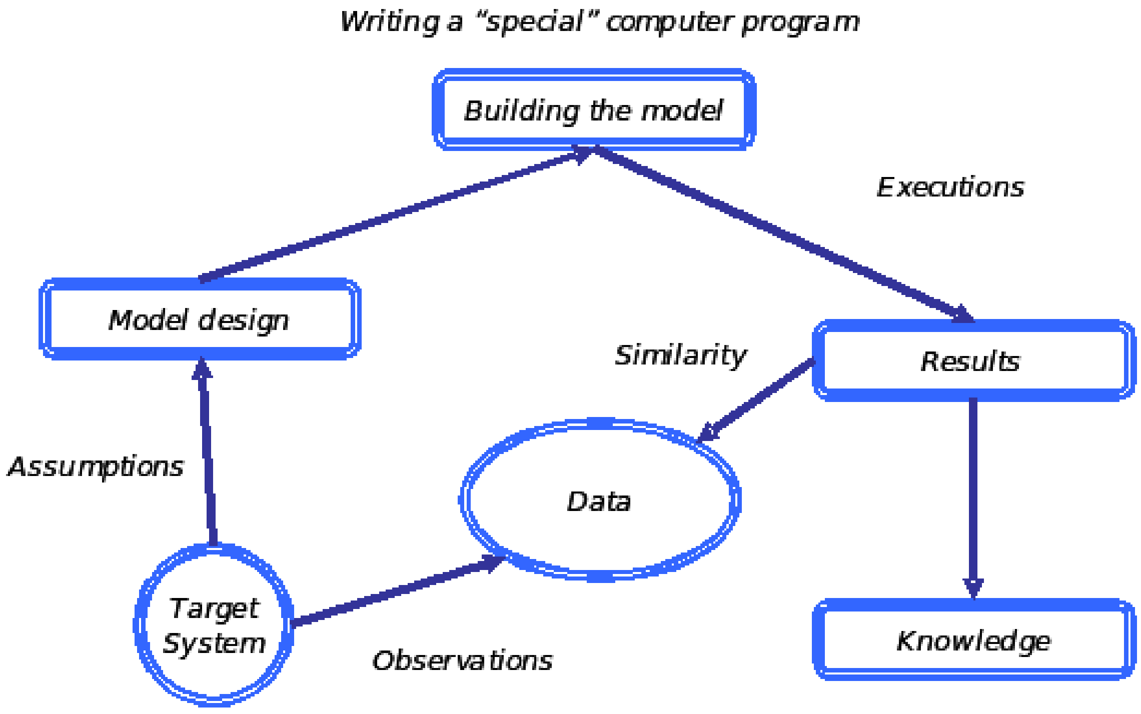


Figure 29: Methodology

(Gilbert, N. & Troitzch K. G., 1999, Simulation for the Social Scientist, Open University Press, Buckingham)

4.1 Choosing Scenarios

As we previously said, we needed to focus into a particular spot of the city to observe traffic behavior. The crossway we chose at the intersection of E. Garza Sada and Luis Elizondo avenues. We decided to pick this one out given that

1. It's close to our laboratory
2. It's a very important crossway
3. It's about to be modified to construct a overpass bridge. (Here, we found an excellent opportunity for our simulator to predict the future behavior of traffic)

4.2 Obtaining Data

The way our field research was done was quite simplistic. We took a house camera and a hand counter. We filmed several blocks of 5 min each at different times of the day, at different days of the week. After that, we analyzed the movie and extracted data from it. We weren't looking forward to have a time dependent traffic flow model, but rather a stochastic one. The data we obtained was: start and end time of each traffic light, arrival rate of each lane, and the distribution of turning directions. The synthesized data is in the following table.

4.3 Conceiving a model

This is perhaps the hardest part when building a simulation. We need to analyze the physical behavior of each component of the simulation. Modeling lights was quite easy, because they only possess three states (green, yellow, red). In the other hand, modeling a car is quite difficult due to the complexity of the mechanics that rule a car. Other complexity comes when we try to adapt a continuous physical model (as simple we decide to make it) to a time step model.

Fortunately, when doing multi agent based simulation, the goal is not to have a very sharp model, since what we're interested on is the interaction of the overall system, but to have a simple, but still representative model.

The model we've implemented suits for any type of scenario, since it is completely customizable by the user. The only restriction it posses is to have straight streets (by the moment).

4.4 Programming software

Even if programming with JAVA it makes it a lot easier since we have a lot of libraries; this stage, together with the testing and debugging stages may be the most time consuming. Given that we were using MadKit as a platform to run our simulation, we didn't have a lot of debugging tools. Also, the class load for MadKit cannot be made on the go, which means that if we modify our code, and recompile it, we would need to close and reopen the platform in order to execute it.

4.5 Calibrating our model

Once we had an application that was stable, we started to run some simulations in order to compare it with data obtained from previous observations of the system. Then we modified the convenient data in our model (light's times, sprouting rate (λ), turning directions, etc).

Anyway, this stage of our project needs more time to be finished. We need to plug in more data and run a lot of simulations to warranty the system's reliability.

4.6 Delivering results

As we have said previously, we are still in the calibration stage. In order to deliver convenient results, we need to verify that our model sticks to reality as best as possible. Otherwise any data that comes out from our simulator will be erroneous and will not reflect a real system.

4.7 Publishing knowledge

The CSI Department is very interesting on publishing the relevant results coming from our simulation. This would be a way of sharing information with other people that may be doing research on the same are as we do. This also may attract directives from the institution to invest even more on research, and then with a consolidated team of researchers, we could have a more developed project that in fact would be useful and applicable to real traffic networks.

5 Future Work

During the development of our project, we realized that there were a lot of interesting points to study, but since the time was short; we focused into the most important functionalities and left the others for a possible future development. Here are some of the points that we could enhance in our system.

Implementation of time dependent model

In order to simulate an environment in which the arrival rate changes dynamically with time, it will be necessary to change the statistical model under which the lights are ruled. This will allow our simulations to stick more to reality (i.e. the change from a rush hour to a quiet hour) and in that fashion we could evaluate the possible implementation of a Bidding system in real world.

Implementation of bidding systems including interleaved lights

For our simulation to have a richer variety of possible scenarios, we'll need to change the way streets are created (only straight streets) and also the implementation of bidding systems where two or more traffic lights are on at the same time.

Implementation of a higher level manager

As we've proved, a bidding system may optimize the traffic flow in an specific area (crossway). Anyway, the global maximum is not guaranteed. A solution to this, would be the implementation of a higher level Traffic Manager that would warranty the optimization of a whole neighborhood (or even a city).

Distribution of the simulation

Even if MadKit allows us to make distributed simulations, in this version of the software, they have not been implemented. Thus, a possible enhancement would be to split a more complex scenario into different machines, by the implementation of migrating agents.

Calibration with more data

As we said before, we need to keep plugging in external data to refine our model and thus make it more accurate. Once this stage is finished we could also extend our model to predict the behavior of new traffic situations.

6 Conclusions

During the development of this project, we have built a simulation system that permits a user to design setup very varied sorts of traffic scenarios. Also the system architecture is designed to be flexible and adaptable to any kind of needs.

Our application simulates at a micro-level using simple car models and multi-agent technologies. Anyway, we're still in the calibration stage, which one finished will allow us to simulate real networks and even predict future behavior of traffic networks.

We also found a way of optimizing traffic lights at a local level. This algorithm (while still need to be extensively tested) will allow to improve the actual networks significantly.

Finally, there is still a lot of work to do: we've just opened a door for future research...

7 Appendix A.- Theory of Queues

7.1 The M/M/1 queuing system

The M/M/1 model is characterized by the following assumptions:

1. Jobs arrive according to a Poisson process with parameter λt, or equivalently, the time between arrivals, t, has an exponential distribution with parameter λ, i.e., for t ≥ 0 , the probability density function is

$$f(t) = \lambda e^{-\lambda t} \tag{1}$$

2. The service time, s, has an exponential distribution with parameter μ, i.e., for s≥ 0 , the probability density function is

$$g(t) = \mu e^{-\lambda t} \tag{2}$$

3. There is a single server;
4. The buffer is of infinite size; and
5. The number of potential jobs is infinite.

Let's define the utilization, ρ, is the average arrival rate x average service time.

By 1, the distribution of inter-arrival times is exponential, hence the average inter-arrival time,

$$\bar{t} = \frac{1}{\lambda} \tag{3}$$

By 2, the distribution of service times is exponential, hence the average service time,

$$\bar{s} = \frac{1}{\mu} \tag{4}$$

and thus, by 1 and 4

$$\rho = \frac{\lambda}{\mu}$$

Also, it is important to assume that $\rho < 1$ otherwise, the queue would grow without limit

Let's consider the following state diagram:

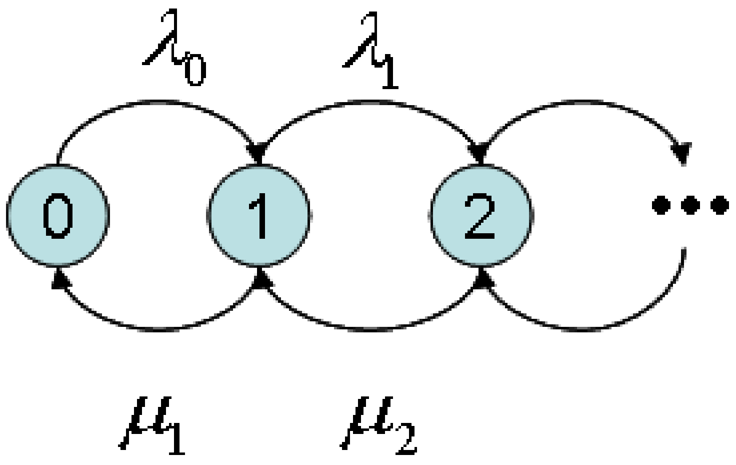


Figure 30: State diagram

In a steady state, the expected number of transitions from n up to n+1 must equal the number of transitions from n+1 down to n, or

$$\lambda_n P_n = \mu_{n+1} P_{n+1}$$

So we can express an equilibrium equation for the initial state as:

$$\lambda_0 P_0 = \mu_1 P_1 \tag{5}$$

But since there is a continuous flow, the arrival rate and the depart rate remain the same all through the process. In that fashion we can express the first relation as:

$$P_1 = \left(\frac{\lambda}{\mu}\right) P_0 = \rho P_0 \tag{6}$$

Following with the next state, we can express the equilibrium equation as:

$$(\lambda + \mu)P_1 = \lambda P_0 + \mu P_2$$

or

$$P_2 = \frac{\lambda(P_1 - P_0)}{\mu} + P_1$$

Then, substituting the equation 6, we get

$$P_2 = \rho\left((\rho P_0) – P_0\right) + (\rho P_0) = \rho^2 P_0$$

From where we can obtain an generalized form

$$P_n = \rho^n P_0 \tag{7}$$

Also, we know that

$$\sum_{n=0}^{\infty} P_n = 1 \tag{8}$$

and substituting equation 7

$$P_0 \sum_{n=0}^{\infty} \rho^n = 1$$

from where we can get the expression for $P_0 = 1/\sum_{n=0}^{\infty} \rho^n$
but since $\sum_{n=0}^{\infty} \rho^n = 1/(1-\rho)$ for any$\rho < 1$ we get

$$P_0 = 1 - \rho$$

And finally

$$P_n = \rho^n (1 - \rho)$$

The expected number of jobs in the system (either in queue or process) is :

$$L = \sum_{n=0}^{\infty} nP_n = \sum_{n=0}^{\infty} n\rho^n(1-\rho)$$

Simplifying we have

$$L = \rho(1-\rho)\sum_{n=0}^{\infty} \frac{d}{d\rho}\rho^n = \rho(1-\rho)\frac{d}{d\rho}\rho\sum_{n=0}^{\infty}\rho^n$$
$$L = \rho(1-\rho)\frac{d}{d\rho}\left(\frac{1}{1-\rho}\right) = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda}$$

Since there is a single server, the expected number of jobs in the queue is

$$L_q = \sum_{n=0}^{\infty}(n-1)P_n = \sum_{n=1}^{\infty}(n-1)\rho^n(1-\rho)$$
$$L_q = \rho(1-\rho)\sum_{n=1}^{\infty}(n-1)\rho^{n-1} = \frac{\rho^2}{1-\rho} = \frac{\lambda^2}{\mu(\mu-\lambda)}$$

Little's formula: In steady state, the average time spent waiting in the queue,

$$W_q = \frac{L_q}{\lambda}$$

And the average time spent in the system

$$W = \frac{L}{\lambda}$$

Applying Little's Formula,

$$W = \frac{1}{\mu-\lambda} = \frac{1}{\mu(1-\rho)}$$

and

$$W_q = \frac{\lambda}{\mu(\mu-\lambda)}$$

From figure 31, we can understand why it is undesirable that $\rho \to 1$. As the utilization rate goes up to 1 the average waiting time goes to infinity.

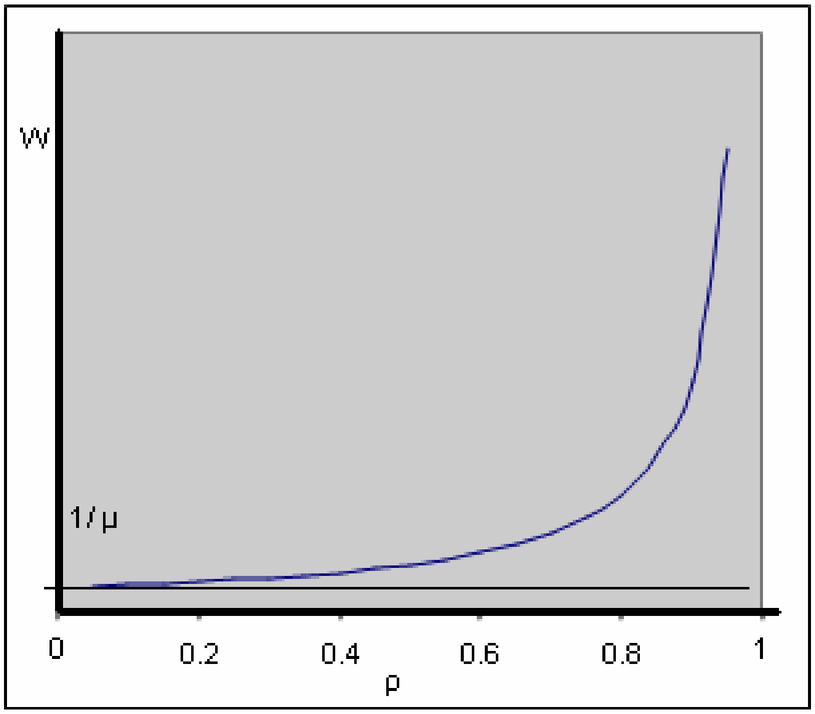


Figure 31: As the utilization approaches to 1, the waiting time trends to infinity

References

[1] A. Drogoul, D. Vanbergue, and T. Meurisse. Multi-agent based simulation: Where are the agents? In *Multi-Agent Based Simulation Conference (MABS'02)*, Bologna, Italy, July 2002. LNCS, Springer-Verlag.

[2] John France and Ali A. Ghorbani. A multiagent system for optimizing urban traffic. In *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, page 411, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Andrej Tibaut. Dynamic load balancing for parallel traffic simulation. In *V: Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed Processing University of Westminster*, pages 134–140, London, United Kingdom, 1997. IEEE Computer Society Press.

[4] Lieurain M. Simulations distribues et multi-agents. Master's thesis, Universit Montpellier II, 1998.

[5] *Teoría de colas y teletráfico.* Universidad Politécnica de Valencia, 1995.

[6] M. Baglietto and R. Parisini, T.and Zoppoli. Distributed-information neural control: the case of dynamic routing in traffic networks. pages 485–502. IEEE Transactions on Neural Networks.

[7] Birgit Burmeister, Afsaneh Haddadi, and Guido Matylis. Application of multi-agent systems in traffic and transportation. *IEE Proceedings - Software*, 144(1):51–60, 1997.

[8] D. Roozemond. Using intelligent agents for urban traffic control control systems, 1999.