MADS Overview

Serban Georgescu, Okuda Laboratory

November 29, 2006

1 Introduction

1.1 From common models and patterns to a new simulation framework

In order to design an useful, efficient and coherent simulation framework, we analyzed the previously selected models and simulations in which these models were used. We discovered several similarities and common simulation design patterns. The significant findings and the ideas that followed from them are the following:

- most simulations combine data and models from various sources → we designed a distributed simulation infrastructure in which simulation modules are combined in the main simulation
- for a realistic simulation, quite a large number of such modules are needed → we designed a service-oriented centralized data/model management system
- data acquisition takes a lot of time → the management system was designed so that previously entered data is available to all framework users, independent of location
- almost all aspects considered in the simulation and the result itself vary with time and this evolution is what drives adoption → we considered the base element of the simulation to be a *service function*, a special kind of time series; all simulation modules are represented by these service functions, which can be created in the simulation or received from remote modules (hence the name *service)*
- a single-product market with a (large) group of consumers and sometimes a (small) group of suppliers is usually considered; interaction between consumers and between suppliers is sometimes taken into account; interaction networks are usually the standard ones (grids, small world networks, random graphs)→ we considered a standard market composed of one product in a number of variants, characterized as attributes represented as service functions, a network of consumers and a network of suppliers; we pre-defined all standard network topologies.
- plotting is always required → we added integrated plotting capabilities

1.2 From Service Oriented Architecture to Distributed Simulation

The Service Oriented Architecture (SOA) paradigm assumes that various services, provided by loosely-connected systems, are combined to make a complete application. In order to function together properly, the services need a message parsing system and some way of advertising their functions and usage.

Usually, the message parsing system is provided by the SOAP [6] protocol, which exchanges XML-based messages over a computer network, normally using HTTP. In this way, SOAP provides a basic messaging framework that more abstract layers can build on. In conjunction with SOAP, the Web Services Description Language (WSDL) [7] is used as a standard way of describing web services. These services can be searched and found using UDDI (Universal Description, Discovery, and Integration) [5] which provides white and yellow pages services.

Another widely used implementation of SOA is the CORBA (Common Object Request Broker Architecture) [13] technology. Generally, CORBA acts as a wrapper for an object written in some language by providing it with a IDL (Interface Definition Language) interface. In this way, any remote system, probably using another programming language, can use the received object. CORBA does not use SOAP, but a different message parsing system, using binary data instead of XML. CORBA is designed to handle a very big number of requests and provides load balancing and various tools for these situations.

At a more abstract layer, making use of SOAP, CORBA and other similar technologies, we can find infrastructures for building powerful large-scale distributed systems. JINI [4] is such a framework, which is able to link various hardware and software services. It is usually used for mission-critical system (e.g. Search & Rescue operations) where different elements need to cooperate seamlessly in order for the whole system to work.

Regardless the method used or implementation type, all these systems have two main elements in common:

1. Provide means for SOA computation

2. Are very general and complex, so that they can be used in all kinds of applications

While using a service-oriented approach is indeed very useful in a wide range of fields, including simulation and in particular technological simulation, their generality and complexity makes these systems difficult to use in small scale simulations. This complexity is well justified by the wide range of applications that these technologies make possible. For example CORBA objects are very ofter behind a whole range of web application, like on-line shopping sites. But developing a CORBA application is not an easy task, and there is not so much sense in using such a technology for simple simulation.

Because of this, having just simulation in mind, more specialized service-oriented system have been developed. This is what is usually called Distributed Simulation (DS). Although more specialized than the first systems, these are also too complex for most small/medium size simulations. One example is the D'Agent system [3], designed with the idea of using remote and dynamical data-sources. Although it was successfully tested on Search & Rescue mission scenarios, it is too laborious to construct all the parts needed for enabling a distributed simulation. Another example is a web-based federated simulation system [8], built using JINI, whose main feature is the possibility of defining in a very flexible way access rights and access lists regarding objects and resources. But these features also increase the work required to design an object and to create a simulation, making the system not so efficient for a small/medium scale simulation.

It seems there is no distributed system simple enough so that any researcher, with a moderate programming skill, can easily use. This fact can be also inferred from the lack of mention to such systems in papers regarding, for example, technology diffusion, field where various numerical simulations are a common thing.

Because, as mentioned before, in the field of technology diffusion numerical simulations are an indispensable tool, having a fairly simple and very easy to use system would save

a lot of time. This system need and should not be so general as to enable the creation of web application, not even to be able to do real-time simulation. Various load balancing and network tools are also not necessary. Also, security and fault tolerance are not so important, as it is not to be used in mission-critical simulations. What is really important here is the simplicity of using and maintaining it and the advantages it brings for the simulation process. The proposed framework aims to be such a system.

1.3 Naming conventions

In this thesis we are proposing a new diffusion simulation framework. This framework will be referred to as MADS, which stands for *Multi-Agent Diffusion Simulation* framework.

In the course of this thesis we will be referring to *agents*. This word will be used here with two meanings, depending on the context. When speaking about MADS infrastructure, agent represents autonomous programs that communicate and provide services to the simulation. When speaking about the diffusion process, agents will represents the actors (households, groups of similar households) that participate (decide) in the process of adoption.

2 The MADS Infrastructure

2.1 Introducing the Service Function

A general characteristic of economical and social simulation (but not limited to these fields) is the fact that usually the result of the simulation is a time series, a function of time. This represents the evolution through time of some variable under study, for example the price of oil or of some other asset, the number of adopters of fuel cell vehicles, etc. Various variables, in order to be computed, need several other parameters, so formally they are not functions of only time, but one can make a distinction between time and the rest of parameters, so that for fixed parameter values the variable becomes only a function of time. Another important characteristic is the fact that these "other" parameters are not constants, but themselves functions of time. The lifetime cost of a car depends, among other things, on the cost of gasoline and the fuel consumption. One can take these as being constants, but that would be a big mistake as usually the evolution of these parameters is what drives the process of adoption in the first place.

The central idea of MADS is to have such functions of time in the core of the system. This means that everything in the simulation will be a special kind of function, called a *service function*. There are no constants, just functions that have constant value through time. These functions will be called service functions because, while from a mathematical point of view they represent a function of time, from the simulation's point of view they represent various services that are provided (usually) by an external source and used internally in the simulation. They are services in that they provide the simulation with the service of computing something, in particular a time series. For example, a service function may compute the price of oil, under some conditions given as parameters. In this context, the functions provide the simulation with the service of computing somehow, not important from the point of view of that particular simulation, the cost of oil at various points in time.

2.2 The benefits of a Multi-Agent infrastructure

Dividing a simulation into modules is a very efficient practice but having various modules that make various computations is not enough. In the field of economical and social simulation, a large amount of information and models, therefore simulation modules, are required for a more realistic simulation. For example, in our simulation of fuel cell vehicle diffusion, more than 50 such simulation parts had to be used in order to account for the most important aspects of the adoption process. Therefore one needs an efficient way of managing and incorporating these parts into the simulation.

The easiest way would be to just copy-paste the code from one simulation to another, but this would make simulations big and hard to understand and the modules themselves hard to find and to maintain. Also, the usage of the code would be restricted to the designer of the functions, the only one that can understand that code (and for how long?).

Another, cleaner solution, would be to put them inside packages (e.g. Java packages). From the simulation's point of view it would be OK, but, as functions get more and more numerous, they would become hard to find. How much information can one store in a file-name? To share the code, the designer would have to share the whole or portions of the function library, accompanied by documentation. After sharing the code, it has no longer any control over its usage. Data and model updates would also be a problem as users will have to verify every time that the package contains the most recent data. Also, modules contributed by other researchers would be difficult to add as these would have to be sent to a central location and distributed during the next library release. This is a very lengthy and time-consuming process.

The way proposed here is to share the code using a different kind of function library, one that uses *agents*, autonomous programs that communicate and provide service to the simulation via service functions. Agents are connected and registered to a server, so anyone connecting to the server can have access to their services. Although agents are required to be registered on the same server, they can actually run anywhere, as long as there is a way of accessing the server. Thus, every researcher can easily add his own modules to the systems by adding or just registering them to the central server. The rest of the users will have access to the added modules instantaneously.

The advantages of this agent based approach are:

- the simulation modules (service functions) can be very easily found
- documentation is very easy to obtain, being displayed when the agent is found
- there is no need of library sharing
- as soon as a service function is added or updated, all users have access to the new function or the new version
- the creator of a particular service function has full control regarding its usage, as it can always remove a particular function by stopping the corresponding agent
- data privacy is also taken care of as the agent does not contain source code, the only visible things being the results

2.3 The infrastructure

2.3.1 Implementation

We chose to implement MADS in the JAVA programming language (version 1.5). There are several reasons for this choice:

- portability: there is a version of JAVA running on all major operating systems (Windows, Unix/Linux, Mac OS, Solaris, etc.) therefore MADS can be used simultaneously on a wide range of platforms
- security: providing services in the form of executable code (the service functions are sent to the application in binary format) represents a potential security risk if the programming language is not prepared to deal with this kind of situations; being created exactly for this purpose, JAVA is probably the most secure programming language there is
- reliability: JAVA is much less prone to error in comparison with other programming languages like C because of the Garbage Collector and Exception Handling mechanism
- productivity: thanks to the large amount of libraries and tools and to the presence of the Garbage Collector, developing an application in JAVA is roughly two times faster than it would be to develop the same application in C++
- speed: combined with the JIT (Just In Time) Compiler, JAVA is almost as fast as C

2.3.2 Communication infrastructure

The next step up into de implementation hierarchy is the choosing of the communication infrastructure. This will handle the communication between the service providing agents and the main simulation. For this task, we selected the JADE ([11]) framework (Fig. 1).

JADE is one the best distributed agent simulation systems. It provides agent-to-agent communication transparent of the agent's location and a powerful agent life-cycle management system. Communication is done using FIPA (Foundation for Intelligent Physical Agents) standards. JADE agents reside into one or more layers which can be on a single machine or distributed. Agents can be created in a layer, deleted, or moved from a layer to another. JADE makes sure the agent management process and the communication are independent of the location of the layers, the operating system the layers reside in or the type of connectivity between the layers. A layer can be created in any place in which JAVA can be run. Thanks to JAVA's portability, not only layers can be created on almost any operating system but also on portable devices as PDAs or mobile phones. Layers can also be created inside a JAVA Applet running inside a web browser, opening the possibility of running code (via agents) on a remote computer, inside the web browser or migrating agents from a central repository to a user's computer (via the web browser) and back.

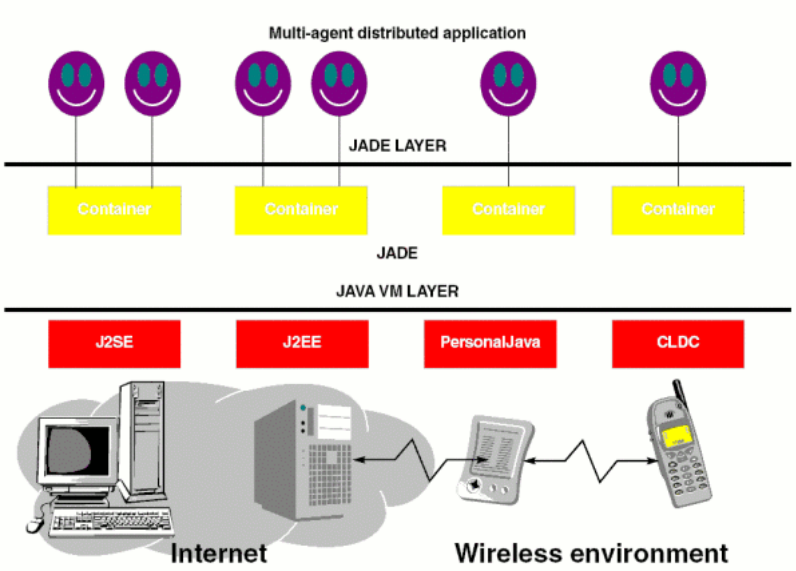

Figure 1: JADE Layout. On the lowest level there we find a wide range of platforms that support a JAVA Virtual Machine. Inside the virtual machines, independent of the physical system, JADE containers can be created. In each container JADE agents can be run.

2.3.3 MADS Infrastructure

Building on the JADE framework, we implement the MADS multi-agent infrastructure for data/model management. We thus provide MADS with all advantages that come from JAVA and JADE. In order to use JADE as communication infrastructure, all parties that create the simulation must be JADE agents. We therefore define two basic types of agents derived from JADE agents and corresponding to the two parties involved in creating the simulation: a *simulation agent type* (referred to as simulation agent) which is responsible for managing the main simulation (on the user's computer) and a *service agent type* (referred to as service agent) responsible for managing the remote service function providing agents.

In a typical simulation (Fig. 2), one simulation agent will request and receive services from a number of service agents. Each service agent has an unique name (the naming convention is global, not local to each container). A simulation agent will send a standard service request, using the name of the agent. The request will be forwarded to the service agent, independent of the agent's location, which will answer by sending the service function. The communication mechanism is asynchronous; the main simulation will wait for a certain time for the answer to arrive and will generate a timeout error if no answer is received.

2.3.4 Finding and requesting an agent

Agent management is a very important task. We transform the agent server into a virtual simulation module database by providing each agent with extensive description of its functionality and by creating a web interface for searching and browsing through these descriptions. A service agent has to provide the following information:

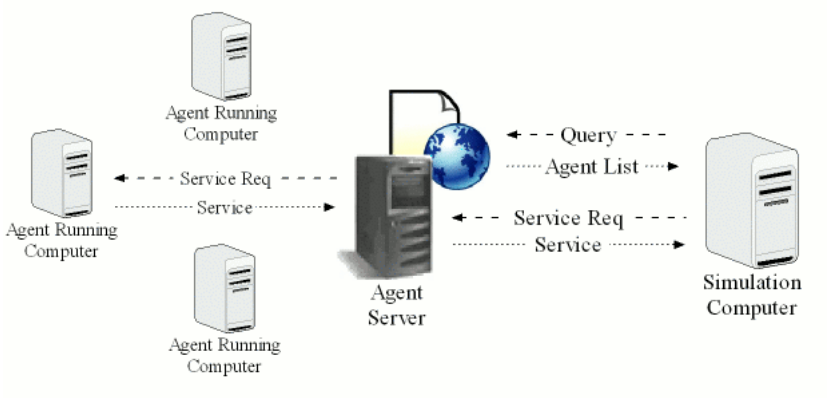

Figure 2: JADE Infrastructure layout

- name (globally unique)
- description; a comprehensive description of the agent's functionality
- time range; the years between which the service is valid (e.g. a forecast for gasoline price might be valid between 2000 and 2030)
- list of parameters; zero or more parameters required for the agent in order to produce the requested data
- references; one or more references to literature related to the data or theory of the provided service

Providing this information is not optional but a requirement of the system. Using such strong policy, the framework makes sure the services are well documented so that searching the database will give valid results and every user knows how to use the service correctly.

Searching and browsing the agent database is done via a web interface (Fig. 3). In fact, the interface itself is a special agent that runs inside the user's web browser, in a JAVA applet. The interface broadcasts a query message to all agents which answer with the information described above.

After browsing the database and locating the necessary agents, the user will call the agent using its name (displayed in the blue frame and underlined at the beginning of the description). Obtaining a service function from a service agent inside the main simulation is very easy and requires just a line of code. For the agent displayed above, the user would have to do something like the following:

```
sfElectricityCost = findRemoteServiceFunction("steamBoilerElectricityCost",
                                              new ServiceFunction[]{pe, le, r, g});
```

where the name of the service agent is written between columns, in blue color, and all parameters, exactly as specified in the agent description, are given between brackets.

Usually, service agents validate the data using the rules specified in the description. If, for example, only three parameters or a negative price would be given, an error message would return instead of the result.

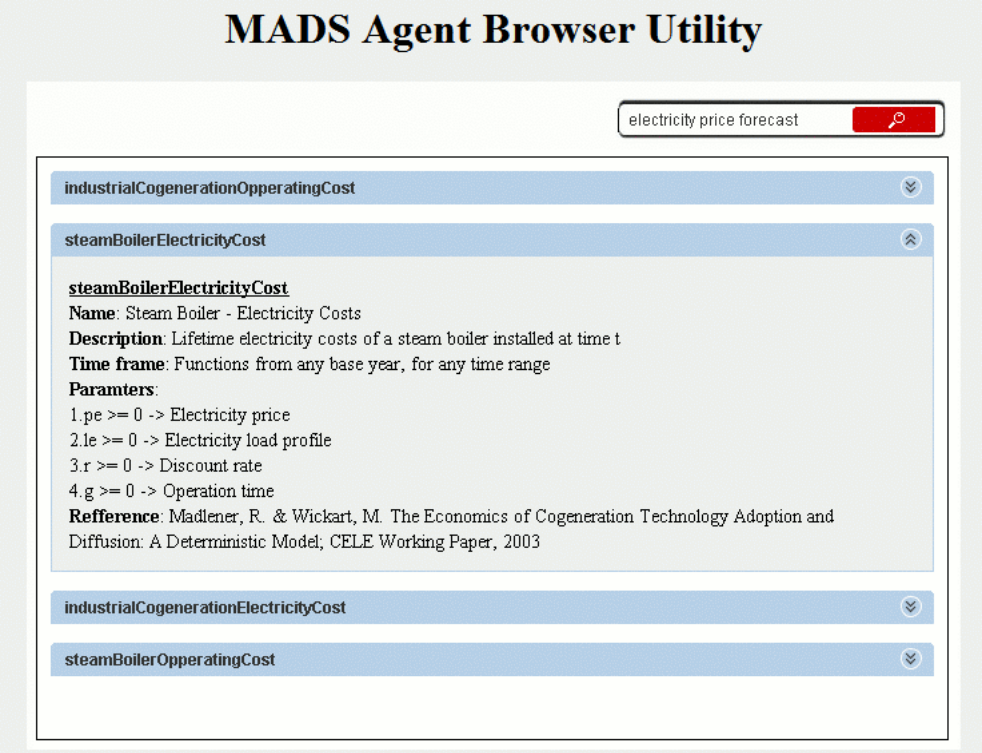

Figure 3: MADS Agent Browser

3 Operations on Service Functions

Service functions are in the core of the MADS framework. All time series received from simulations modules have the form of service functions and so are all parameters and data used throughout the simulation. For this reason MADS contains an extensive list of operations with and on service functions.

3.1 Domain of definition

As service functions represent predictions, trends, data and so on, a time interval of validity must be considered. In MADS, for every service function we define the *base* and *range*. The *base* represents the starting year from which the service function is valid. For example, for a gas price forecast computed for from year 2000 to year 2030, *base* would equal 2000. The *range* represents the number of years, starting from the base, from which the service function stays valid. In the previous example, the *range* would be 30.

For some service functions, time does not matter (e.g. constant functions). For these, the *base* defaults to 0 and the *range* defaults to 10000, which is also the maximum possible range. As time is usually counted in years, the value of 10000 is more than sufficient.

3.2 Atrithmetics

Service function artihmetics is included in MADS. We define an arithmetical operation between two service functions *sf1* and *sf2* as an arithmetical operation made on every pair *(sf1(t), sf2(t))* for every point in time t in which both functions are defined. If the domains of definition of the functions involved in the operations are not the same, the result service function's domain will be the intersection. This insures that operations on

service functions are restricted to the time interval where all involved time series are valid. As the result of an arithmetical operation is also a service function, multiple arithmetical operation can be combined (chained) using the same functions.

The following arithmetical operations are included: *addition, subtraction, multiplication* and *division.*

In Fig. 4, a sample addition between two service functions is presented. The first service function represents the forecast for gasoline price on 2000 - 2030 while the second represents a potential tax on pollution. Both service functions are received from service providing agents. We combine them inside the simulation by on one line of code and obtain the taxed gasoline price for 2000 - 2030.

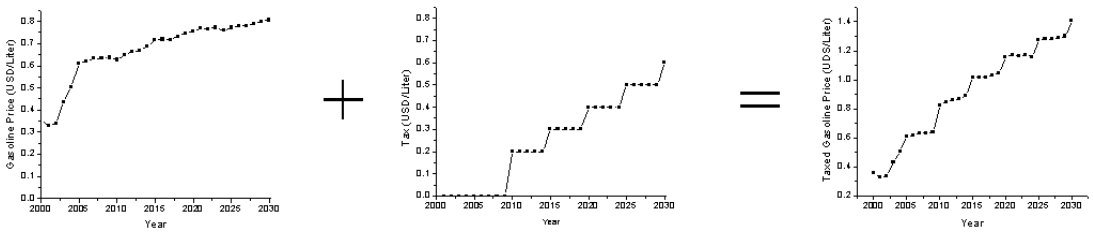

Figure 4: Sample service function addition

3.3 Other operations

The points and values where the service function reaches its maximum and minimum values can be found using the *min* and *max* functions. The mean value on the whole domain is given by the function *mean.*

Usually, some elements from a simulation are generated according to some distribution function. Example of such elements are consumer characteristics, firm strategies, connections from interaction topologies and so on. In these cases, a simulation has to be run for a number of times and the results of each simulation have to be averaged. As will be later presented, the result of a simulation is itself a service function therefore this averaging of results reduces to an averaging of service functions. We have implemented this using the *AverageServiceFunction* class, which defines a service function having at every time step the average value of all service functions given as parameters. The usage is the following:

```
for (i = 0; i < N; i++)
{
  // Compute some result for each iteration
  sfResult[i] = compute_some_service_function(parameters)
}
sfFinalResult = new AverageServiceFunction(sfResult);
```

3.4 Writing to file

Sometimes service functions need to be saved for later reference, plotting in a specialized software, further processing, etc. All information stored inside a service function can be saved to a text file by simply calling a *saveToFile* function.

3.5 Plotting

In almost every kind of simulation plotting is necessary. This might be to check the input data, verify the data received from service agents, test the evolution of the system at various points either to see whether it is evolving as it is supposed to or just to get some insight in the process at it evolves, plot the final results, etc. Exporting data and plotting it in an external software would be a great waste of time as this operation is performed extremely often. We therefore incorporated simple service function plotting into MADS. As almost everything in a MADS simulation is a service function, almost everything can be directly plotted. A plot resulting from this procedure will look like the one from Fig. 5.

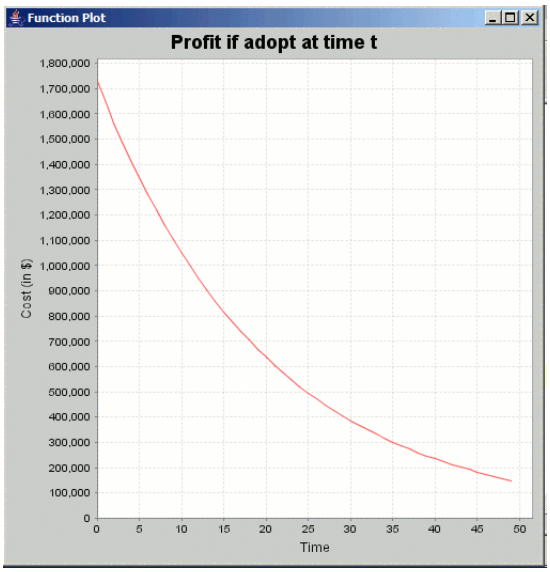

Figure 5: Sample service function plot for a co-generation system adoption simulation

3.6 Creating service functions

Until now we showed that service functions can be obtained by one of the following ways:

- received from a remote agent using *getRemoteServiceFunction*
- obtained as a result of an arithmetical operation
- obtained as a result of an averaging procedure via *AverageServiceFunction*

In the following, we will show how these service functions can be created inside a simulation.

The simplest way of creating a service function is by directly instantiating the base *ServiceFunction* class. One will obtain a new service function with a specified time domain (and the default one if none is specified) and having all values equal to zero. Values can then be added for each point in time using a *set* function. One might want to create such a service function to perform some custom operation on data, or simply to store data in some form than can be easily saved and plotted.

Another more practical way to create a service function is by using a predefined function type. MADS provides the following service function patterns: *constant service function,*

linear service function, logistic service function, gaussian service function and *discount service function.*

A *constant service function* will probably be the most used service function pattern in a simulation. As in MADS all data and parameters must be given in service function format, constants will be represented by service functions that have no time domain restrictions and whose value is the same at any point in time.

The second in order of usage in a diffusion simulation is the logistic service function. This is because the progress (development) of some technology or diffusion process has generally the form of a logistic (sigmoid) function. A logistic function has the following formula:

$$f(x) = A2 + \frac{A1 - A2}{1 + e^{(x-x_0)/dx}}$$

where *A1* is the initial value, *A2* is the final value, x_0 is the center (the point of inflection) and dx represents the width.

As mentioned before, logistic functions often appear in the context of technological evolution. Providing such a service function template becomes very handy in diffusion simulation. One usually finds data or forecasts regarding the evolution of some technology, fits the data with a logistic function and obtains the desired coefficients. As an example, the evolution of the time needed by a generic FCV to accelerated from 0 to 100km/h, from the year 2000 to 2030, can be written with a single line of code:

```
ServiceFunction sf = new LogisticServiceFunction(12.00546, 9.29999, 2006.55497, 1.05672)
```

In some cases, a *gaussian* function fits the data better than a logistic function. For these cases, a *gaussian service function* has been defined. A gaussian function can be described by the following formula:

$$f(x) = y_0 + \frac{A}{w \cdot \sqrt{\frac{\pi}{2}}} e^{-\frac{2(x-x_0)^2}{w^2}}$$

where y_0 represents the baseline offsetn, A is the total area under the curve from the baseline, x_0 is the mean and $w/2$ represents the standard deviation.

An example where a gaussian function fits the data better than a logistic function is in the case of the weight of hybrid and gasoline vehicles. The evolution of the weight of a generic hybrid vehicle, from 2000 to 2030, can be compressed into the following line of code:

```
ServiceFunction sf = new GaussianServiceFunction(1148.52736, 2008.52977, 2008.52977, 2008.52977)
```

Another useful function defined in MADS is the *linear function,* with the well known formula

$$f(x) = ax + b$$

The last service function pattern is the more special *discount service function,* often used in economic simulation. This function represents the exponential decrease in time of a value of some asset due to inflation or risk aversion. The notion of discount represents the fact that a certain amount of money is more valuable now than it would be in future both because of an uncertainty regarding the future and because of the possibility of investing the money, if possessed now, and making a profit. Having an asset or an amount of money whose value in time is represented by a service function *sf*, the real (discounted) value will

be easily computed in MADS by simply multiplying sf with a discount service function. The formula for a discount service function is the following:

$$d(x) = e^{-rx}$$

where r represents the discount rate (usually 0.05, which represents a decrease in value by 5% every year). One can apply the discounting operation in a single line by chaining the multiplication and service function creation, as in the following piece of code:

```
sf.multiply(new DiscountServiceFunction(-0.05))
```

4 Networks

In the previous chapter we argued that social interactions are important and should be taken into account in diffusion simulation. For the interaction models, support for networks has been included in the framework. The MADS graph handling infrastructure is provided by JUNG ([1]).

4.1 Service Network Functions

To provide maximum flexibility, networks are defined in the form of *service network functions*, in analogy to service functions. This is because we consider networks to be a special kind of service functions whose value at any time step is represented by a network topology, a graph. Network service functions can be created in the simulation, usually using generators, or received from remote agents. Although usually topologies are considered to remain unchanged with time, in some particular cases the evolution of the network might prove important. This is the reason why we chose to define the intersection topologies as graph time series as opposed to simple graphs.

As in the case of service functions, the value at a particular time t (in this case a graph) can be obtained using a *get* function.

There is no arithmetics associated with service network functions as in this case it does not make sense.

4.2 Creating service network functions

Although service network functions can be directly created by instantiating the *ServiceNetworkFunction* class and the network handcrafted by adding vertices and edges using the tools provided by JUNG, most simulations restrict themselves to using standard, well known network topologies. MADS provides *network generators* for all standard topologies: *grids, small world networks, random graphs* and *scale-free random graphs.*

4.2.1 Grids

The most often used interaction topology, the 2D grid, is generated by the *TwoDimLatticeGenerator* generator. A service network function can thus be created using the following (chained) syntax:

```
ServiceNetworkFunction snf = (new TwoDimLatticeGenerator(N, true)).generateNetwork()
```

where N represents the size of the grid and the second parameter specifies if the lattice is toroidal or not (usually is).

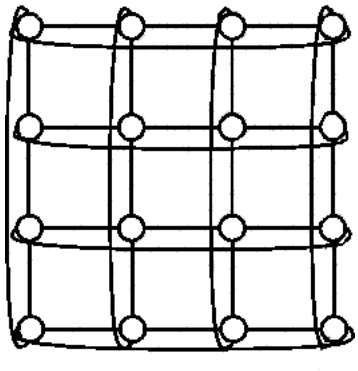

Figure 6: 2D toroidal grid with size 4 (16 vertices)

4.2.2 Small world networks, 1D latices and random graphs

Small world networks[12] are considered to be the best representation for the the relations in a society (e.g. the small world phenomenon, six degrees of freedom). Small world networks bridge the gap between networks with high clustering (e.g. grids) and networks with small average path length (e.g. random graphs). A simple and intuitive way of creating such networks is by starting from a grid topology and then making a few random shortcuts. This can be also seen as an analogy with the fact that there is dense connectivity between a person and his/her entourage but there is also a weak connectivity between very different (very distant) entourages through some common acquaintances. While decisions are usually strongly influenced by the ones of the entourage, information travels fast between entourages via these weak connections.

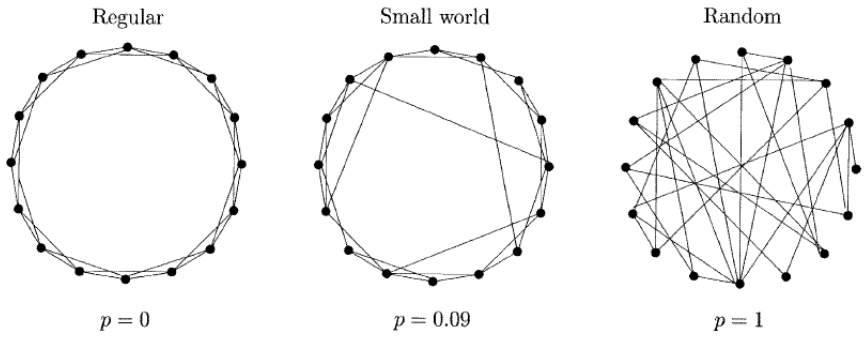

Figure 7: 1D small world network, as in Watts. p represents the shortcut probability

In MADS, two different small world generators, one for one dimensional networks using the Watts beta function ([12]) and another for the two dimensional version, using Kleinberg's algorithm ([9]) are provided. Other generation methods are also possible [10]. Their syntax are the following:

```
ServiceNetworkFunction snf = (new WattsSmallWorldNetworkGenerator(N, p, deg)).generateNetwork()
ServiceNetworkFunction snf = (new KleinbergSmallWorldNetworkGenerator(M, d)).generateNetwork()
```

where N represents the number of vertices, p the shortcut creation probability, *deg* the degree of the vertices, M represents the size of the 2D lattice and d represents a clustering coefficient.

It should be noted that by setting the shortcut probability p to 0, a *1D toroidal lattice* will be obtained. Also, if p is set to 1, the result will be a *random graph.* The values of p for which a small world network is obtained are in the range [0.01, 0.1]. This transition lattice → small world network → random graph while varying the parameter p from 0 to 1 will prove very useful for the sensitivity analysis.

4.2.3 Scale-free random graphs

Scale free networks represent networks in which the probability of a new connection from a source node to a destination node is not uniform, but proportional to the number of nodes already connected to the source node. Such networks are centered around "hubs" having a very large number of connections. Samples of scale free networks are the World Wide Web and any kind of networks that involve preferential attachment or strong influence by central individuals.

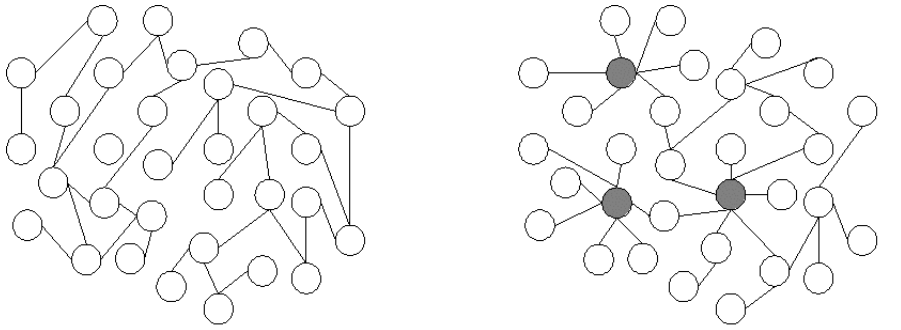

Figure 8: Difference between a random graph and a scale-free random graph

In MADS, scale-free random graphs are generated with the most commonly used Barabasi-Albert algorithm ([2]) using the following syntax:

```
ServiceNetworkFunction snf = (new BarabasiAlbertRandomScaleFreeGenerator(N, E)).generateNetwork()
```

4.3 Plotting

Plotting was implemented also in the case of service network functions. Because the type of information these objects carry (graphs and not simple numbers) two types of plots were implemented. While one is for visualizing the graph associated with a particular point in time, the other is for viewing, step by step, the evolution of the states associated with the vertices for the whole simulation.

The first type of plot can be created by simply calling the *plot* routine associated with the service network function. By default, only the graph structure will be plotted. If the

states associated with the vertices are also necessary, a states map that links the vertices with the states must be provided (such a map can be received from the object holding the simulation results, which will be presented in the next sections).

Sample plots obtained by both methods are shown in Fig. 9.

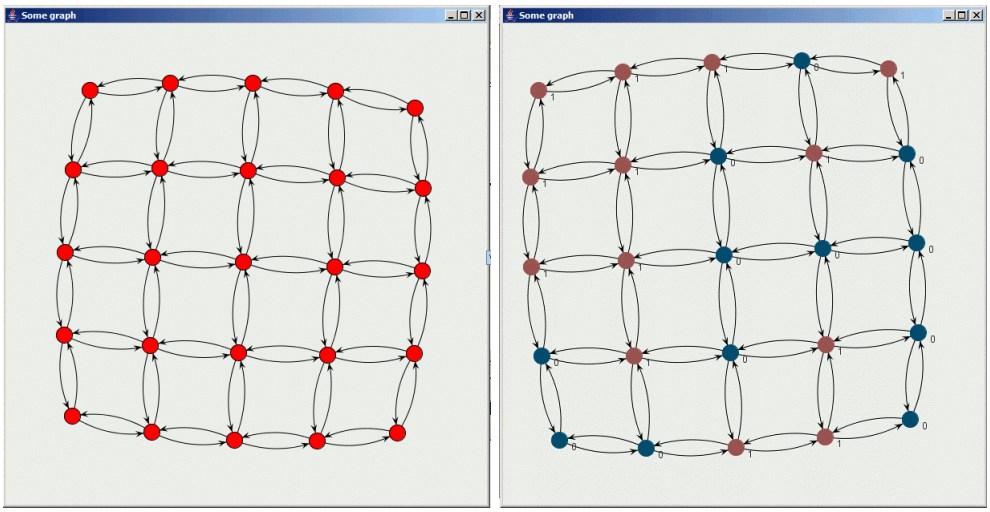

Figure 9: Sample plots for a 2D grid. In the left side, just the connectivity is shown. In the right side, the state information can also be observed.

The second type of plotting can be accessed from the result file returned after the simulation has ended (*NetworkSimulationResult* class*)*. A sample plot created this way can be seen in Fig. 10. At the bottom of the plot navigation buttons and a box displaying the current time step can be seen. As the user goes forward in time, the evolution of the system can be clearly seen.

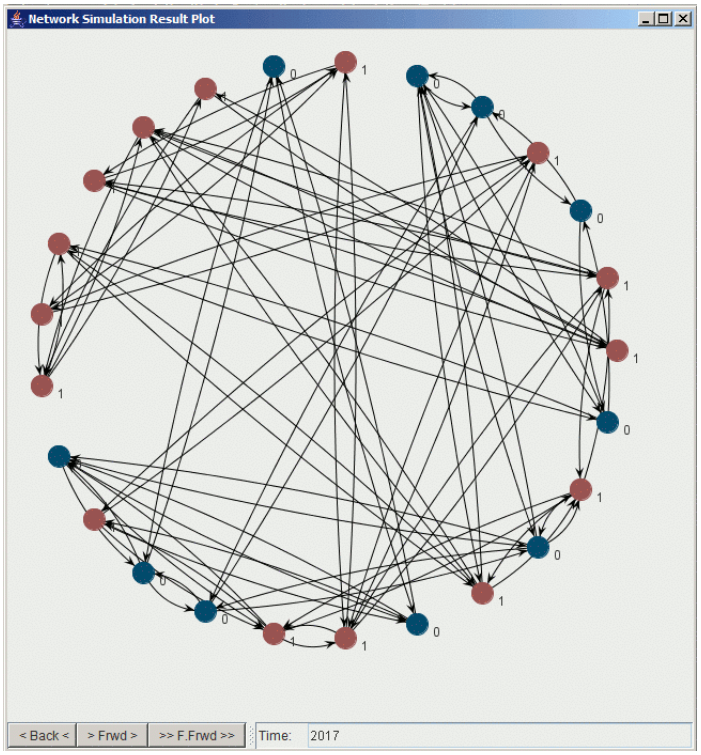

Figure 10: Sample plot

5 Simulations

This section shows how simulations can actually be done in MADS. There are three basic simulation types: *disconnected*, *interaction-based* and *complex-interaction-based*. Although all simulations can be implemented as *complex-interaction-based*, we preferred to keep things simple where possible.

The *disconnected* type deals with agents that do not interact directly nor indirectly. This particular feature makes the simulation very simple to design, consisting only of service function arithmetics.

In *interaction-based* simulations, simple interaction between agents is considered but the supply side is not taken into account. As it deals with interactions, it has to be done in an iterative way.

Complex-interaction-based simulations are the most general kind, including both direct and indirect interactions between agents and a supply side.

5.1 Disconnected simulations

Disconnected simulations include the *rank* model, usually used for reference cases. In a disconnected simulation, a perfectly-rational and informed profit-maximizing consumer chooses to adopt a new technology, if profitable, at exactly the time when the profit is greater. As there is no interaction between agents, this type of simulations cannot be considered multi-agent.

5.1.1 Usage

In MADS, a disconnected simulation can be implemented in four steps:

1. find necessary data for the product's characteristics and encapsulate it into service agents
2. call the data in the form of service functions
3. do arithmetics with service functions to arrive at the profit
4. find the time when the profit has the maximum value; this is the adoption time

5.2 Social interaction simulations

Social interaction simulations include statistical mechanics inspired models and all other models which assume only demand side and interaction between agents. Thus, they are multi-agent simulations. A graphical representation is shown in Fig. 11.

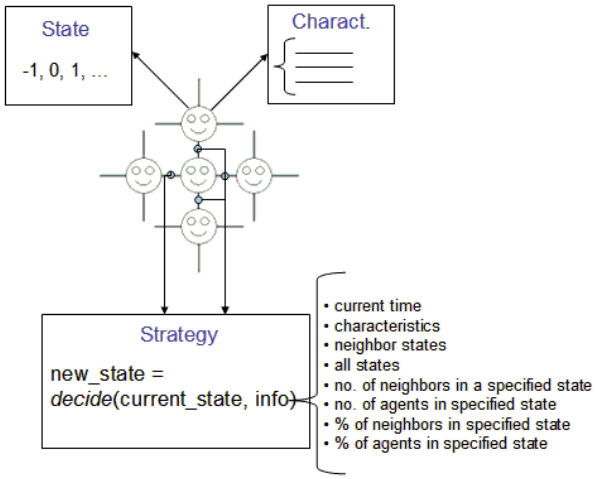

Figure 11: Social interaction simulation layout

In such simulations, agents (consumers) are considered to have *characteristics* and *states*. By agent characteristics we understand all "personal" attributes that influence an agent's process of selection. These attributes change with time, so we express them as service functions. An agent also has a state, representing the choice the agent made. The choice is considered to be an integer number. For example, if the agent has to decide between a gasoline vehicle, a hybrid and an FCV, a 0-state might denote the current possession of a gasoline vehicle, a 1-state the possession of a hybrid and a 2-state the possession of a FCV.

To decide who interacts with whom, a network topology must be specified. This is done by providing the simulation with a network service function.

A state represents the result of a decision process, of applying a decision strategy. In MADS, the decision process associated with an agent is encapsulated into a *strategy* object. This object receives from the simulation data regarding the agent's current state, state and characteristics of the neighbors and so on.

When and in which order agents get to decide is specified in a *run regime*. Three run regimes have been implemented:

- *runRandomized* - execute the decision procedure for a specified number of agents, chosen randomly
- *runSequentialSweep* - execute the decision procedure for all agents, in sequential order
- *runRandomSweep* - execute the decision procedure for all agents, in random order

After the simulation is run, results are stored in a *networkSimulationResult* structure. This object can perform the following functions:

- return a service function representing the evolution of a state, from the start time to the end time of the simulation; the evolution is given as the percent of agents being in the specified state at each point in time
- plot the graph of the system at a particular time step
- plot the entire evolution of the system

5.2.1 Usage

To implement a social interaction simulation in MADS, one must use the following steps:

1. find a model for the consumer (relevant characteristics that affect decision)
2. find necessary data for the product's characteristics and encapsulate it into service agents
3. call the data and set the consumers' characteristics
4. decide the simulation time range
5. set the initial conditions
6. set the interaction network
7. run the simulation
8. extract the results

5.3 Complex interaction simulations

Complex interaction simulations must be used in the cases where both demand and supply have to be considered. This type of simulation will be used for our FCV diffusion simulation. A complex interaction simulation contains the following elements:

- a product
- a network of consumers
- a network of suppliers
- an automated market agent to handle the trading between consumers and suppliers
- a bank for loans, in the cases where complex supplier models are employed (e.g. the evolutionary supplier presented in Section **??**)

A complex interaction simulation is divided in trading cycles, one each year. In each trading cycle, suppliers make offers and consumers evaluate and buy. In order to avoid using inventories, it is considered that production takes place on-demand, only after all consumers have ordered. A trading cycle has the following steps (Fig. 12):

1. suppliers post offers to the market agent for the product, containing the characteristics and price
2. consumers get all the offers from the market
3. consumers analyze the offers, choose the ones that are acceptable and post them to the market agent in the order of preference
4. the market agent does the matching (chooses consumers in random orders and, if possible, matches their first preferences with the corresponding offer; if this is not possible further rounds are conducted, where lower level options try to be satisfied)
5. the matched orders arrive to the suppliers
6. if something was bought then what was bought arrives to the consumers

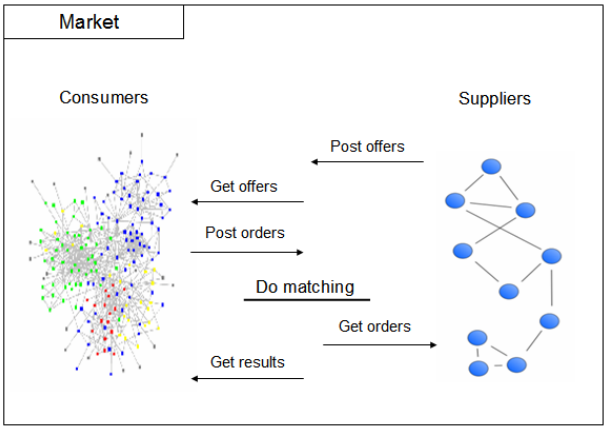

Figure 12: A trading cycle

A trading cycle is automatically done my MADS, provided a collection of consumers and a collection of suppliers. Depending of the purpose of the simulation, various models for consumers and suppliers can be used. All the actions from the steps 1 to 7 are automatically done by MADS, but there are still functions that have to be implemented by the user.

For the consumer, the user has to define an *utility evaluator* which given a product (which includes all characteristics relevant for the decision, in the form of service functions) and considering the consumer's preferences (included in the consumer's characteristics, also as service functions) will return a real number. The system will then rank the offers received from suppliers according to this number and post orders to suppliers.

For the supplier, the user has to devise a way in which the supplier decides the price of the product and the quantity made available, for each year. In the case of the virtual supplier (Section **??**), the quantity will be infinity and the price will be given as a service function. In the case of the simple adaptive supplier (Section **??**) the price will also be given

as a service function but the quantity will be decided according to the number of offers received in the previous year. Finally, in the case of the evolutionary supplier (Section **??**), both price and quantity will be internally computed following a rather complex algorithm.

5.3.1 Usage

To implement a complex interaction simulation in MADS, one must use the following steps:

1. Define the product
2. Find a utility function for the consumer
3. Find a model for the supplier
4. Find necessary data to be used as parameters both inside the consumer's utility function and inside the supplier's model and encapsulate it into service agents
5. Decide the simulation time range
6. Set the initial conditions
7. Set the interaction network
8. Run the simulation
9. Extract the results

References

[1] Java universal network/graph framework.

[2] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[3] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and performance of a mobile-agent system. *Software— Practice and Experience*, 32(6):543–573, 2002.

[4] http://www.jini.org/. Jini.

[5] http://www.uddi.org/. Universal description, discovery, and integration (uddi).

[6] http://www.w3.org/TR/soap/. Simple object access protocol (soap).

[7] http://www.w3.org/TR/wsdl. Web service definition language (wsdl).

[8] Xueqin Huang and John A. Miller. Building a web-based federated simulation system with jini and xml. In *34th Annual Simulation Symposium (SS01)*, 2001.

[9] Jon M. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.

[10] M. E. J. Newman. Models of the small world. *Journal of Statistical Physics*, 101:819–841, 2000.

[11] TILAB. Jade (java agent development frameowk). 1999.

[12] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Letters to Nature*, 393:440–442, 1998.

[13] www.corba.org. The common object request broker architecture (corba).