# Using Specification and Description Language to represent users' profiles in OMNET++ simulations

Pau Fonseca i Casas
Dep. of Statistics and Operations Research. Universitat Politècnica de Catalunya. C/ Jordi Girona, 31. Barcelona 08034, Spain
+34 (93) 4017732
pau@fib.upc.edu

Miquel Ramo Niñerola
Dep. of Computer Science. Open University of Catalonia. Rambla Poblenou, 156. Barcelona, 08018, Spain
+34 (93) 3263627
mramo@uoc.edu

Angel A. Juan
Dep. of Computer Science Open University of Catalonia. Rambla Poblenou, 156. Barcelona, 08018, Spain
+34 (93) 3263627
ajuanp@uoc.edu

## ABSTRACT

Omnet++ is a powerful and open-source simulation tool which is basically intended to model discrete-event systems. In particular, Omnet++ is extensively used to model and simulate computer networks. Typically, when a Wide Area Network needs to be modeled, different assumptions are made in order to simplify the complexity associated with human behavior. Nevertheless, human behavior can also be modeled, at least to some extent, by using Multi Agent Systems (MAS). This paper presents a methodology that allows connecting a MAS model –which accounts for human behavior–, with a standard Omnet++ model –which represents the behavior of a computer network. The approach presented here can be useful to obtain a better representation of the human behavior through a MAS model when using Omnet++. Furthermore, our approach simplifies the modeling process by splitting the complexity of a real system into two different parts. Therefore, on the one hand computer scientists can focus on the Omnet++ model while, on the other hand, specialists in human behavior can focus on the MAS model. Finally, our approach also facilitates the distribution of the models among different computers.

## Categories and Subject Descriptors

G.3. [**Simulation and Modeling**]: Simulation Support Systems, Applications, Types of Simulation – *Discrete Event.*

## General Terms

Experimentation, Languages, Human Factors.

## Keywords

Discrete-Event Simulation, Multi-Agent Systems, Computer Networks, Human Behavior, Omnet++.

## 1. INTRODUCTION

The Castelldefels project aims at developing a realistic simulation model of the computer system that gives support to the Virtual Campus of the Open University of Catalonia (UOC). UOC (www.uoc.edu) is an online university that offers e-learning services to thousands of users. During the development of this project, different alternative approaches regarding the implementation of the simulation model have been analyzed. Some of these approaches have used OPNET [1], while others have used Omnet++. The main goal of the project is to provide managers of the computer system with a realistic simulation model that allows them to: (i) analyze the behavior of the current system in order to discover possible performance problems – bottlenecks, weak points in the structure, etc., and (ii) perform what-if analysis regarding future changes in the system, including the addition of new Internet-based services, variations in the number and types of users, changes in hardware or software components, etc.

In order to analyze computer systems' and networks' performance, both analytical and simulation methods can be used. Analytical methods are based upon mathematical analysis that characterizes a network as a set of equations. This approximation usually implies considering several restrictive assumptions, which tend to be not very realistic, since networks are complex systems formed up by hardware and software (protocols, applications, queuing policies, etc.). Alternatively, simulation techniques can also be used to model in detail the dynamic nature of real computer networks [2, 3]. Simulation allows engineers to test different network designs, even before the network physically exists, and to perform what-if analysis with models of the already existent networks without exposing them to failures or inoperative periods.

The Open University of Catalonia (UOC) is an online university located at Barcelona (Spain) with more than 37,000 community members, including students from Spain and Latin America, professors, and managers among others. With this amount of potential intranet users, performance fine-tuning of the computer system that gives support to the UOC Virtual Campus becomes the most important task for system managers. For that reason, a team formed by managers, professors and students started the so-called Castelldefels Project. The main objective of this project is to improve the system performance levels and, consequently, to increase the quality of the service offered to users of the UOC Virtual Campus. This is carried out by selecting appropriate values for configuration parameters such as network topology, hardware devices, queuing and balancing policies, protocols, etc. [4, 5].

## 2. ADDING AGENTS TO THE MODEL

Our approach for modeling users' behavior through agents is described in Figure 1. Agents interact with the Castelldefels system and, as a result, events of a particular type are generated. These events have been classified as shown in Table 1. As will be explained later in this paper, the number of events has been limited to 999.

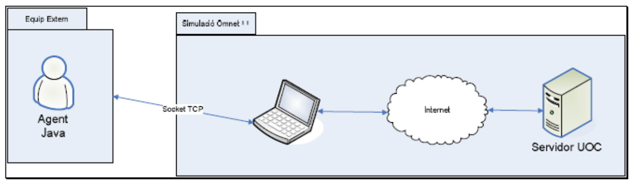


**Figure 1**: Interaction agents-system

**Table 1**. Types of events.

| Event code | Description |
|---|---|
| 1 | BEGIN |
| 2 | END |
| 3 | LOGIN |
| 4 | LOGOUT |
| 5 | OPEN MAIL |
| 6 | READ MAIL |
| 7 | CLOSE MAIL |
| 8 | SEND MAIL |
| 9 | EDIT MAIL |
| 10 | OPEN CLASS |
| 11 | CLOSE CLASS |
| 12 | OPEN FORUM |
| 13 | READ POST |
| 14 | EDIT POST |
| 15 | SEND POST |
| 16 | READ NOTES |

Each one of these events, which must be processed by an agent, has its own state as described in Table 2. This classification is useful in order to represent the errors that can take place during the communication process. These errors are represented in the Omnet++ simulation model.

**Table 2**: Possible states for an event.

| | |
|---|---|
| Pending or scheduled | The event is created in the event list of the agent, but it is waiting for its execution time. |
| In the output queue | The event is in the ChannelWriter to travel to its destination. |
| Sent | ChannelWritter had read the event and it has been send to its destination. |
| On input queue | The event has been processed by the environment and has been returned by the environment. We read the event from the ChannelReader and send the event to the destination agent queue. |
| Received with an error | The agent receives the event, but the environment marked it as wrong. The agent must send again the event. |
| Received | The agent receives the event. |
| Processed | The agent marks the event as processed when it starts the tasks related to other events. |

## 3. AGENTS USED IN OUR MODEL

An intelligent agent receives information through its sensors. Depending on the received information, it executes actions with the "effectors" or "actuators". An intelligent agent adds evolutionary capacity to the whole model, since it is able to implement algorithms that provide it with learning capacities and, therefore, can modify its behavior through time. Different intelligent agents can be considered depending on how the information is processed. Cortés et. alt. [6] proposes the next classification:

- **Simple reflexive agents**: These agents do not have states. Their actions are answers to the perceptions received. The connection between perceptions and actions are based in condition-action rules.
- **Model-based reflexive agents**: These agents have states, which allows for the use of more information when providing an answer. They are based on the idea that sensors do not provide all the necessary information. These agents store relevant information that complements the one provided by the sensors in order to perform the correct action.
- **Goal-based agents**: The agents have specific goals to be achieved. These goals allow selecting the actions to be completed.
- **Utility-based agents**: These agents have goals that do not guarantee by themselves the utility of the agent's behavior. For that reason, a utility function is defined. Actions are then chosen so that this utility function is maximized.

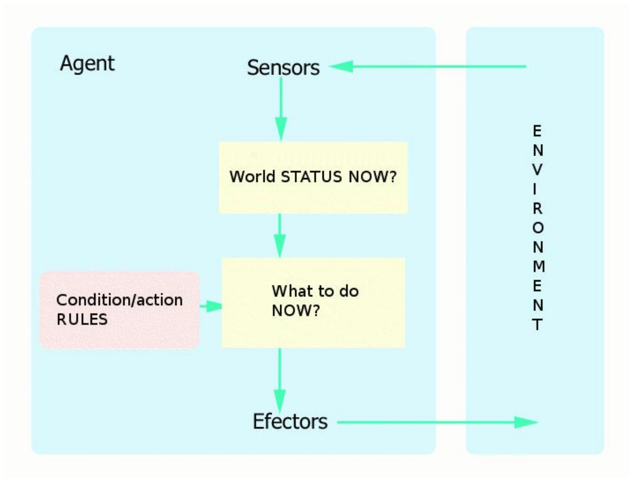


**Figure 3**: Schematic representation of a simple reflexive agent

Our approach proposes the use of simple reflexive agents, which can be represented as shown in Figure 3

The behavior of a simple reflexive agent can be summarized as follows: (i) the agent remains waiting until some environmental information is received; (ii) the agent processes the environmental information received and uses condition/action rules to determine what to do; (iii) the agent can initiate different actions to answer to the stimulus it receives; And (iv) when all due actions have been completed, the agent returns to the waiting state.

In our model agents modify their behavior depending on the information (input events) they receive from the environment. Also, communication with the environment is started by each agent. These agents implement different user profiles using a list of events (that can be fully configured). Each agent has a definition of its chronological list of events (depending on the profile). First, each agent waits until the reception of the "BEGIN" event, which defines the starting point of the simulation. Then, it waits until it comes the time to submit the next event. When all events have been sent, the agent returns to an inactive state. The simulation finishes once all agents have reached the inactive state. Moreover, each agent satisfies the following capacities related to an agent-based model:

- **Autonomy**: Each agent is responsible for its own execution thread, thus being in charge of its own life (of course, its life will also depend on the environment).
- **Reactivity**: Each agent processes the environmental information. Also, each agent possesses an input and output queue.
- **Planning**: The behavior of an agent is planned depending on its profile and the random environmental events.
- **Character**: It is defined by each agent's profile. The main objective is to represent the user's behavior.

Having the previous considerations in mind, the next section proposes a formal representation of these agents using the SDL language.

## 4. SPECIFICATION AND DESCRIPTION LANGUAGE

Specification and Description Language is an object-oriented formal language defined by the International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) (the Comité Consultatif International Telegraphique et Telephonique [CCITT]) on the Z. 100 recommendation. The language was designed for the specification of event-oriented, real-time and interactive complex systems. These systems might involve different concurrent activities that use signals to perform communication [7], [8], [9]. SDL is based on the definition of four levels to describe the structure and the behavior of the models: system, blocks, processes and procedures. In SDL *blocks* and *processes* are named *agents*. The outermost block, the *system* block, is an agent itself. Figure 4 shows this hierarchy of levels.

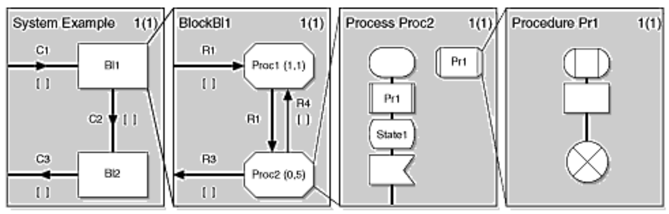
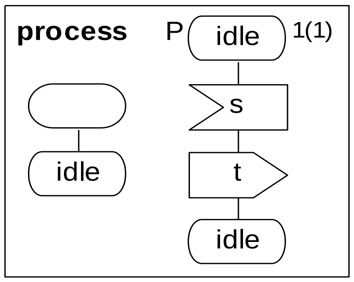


**Figure 4**: A structural vision of an SDL model. 4 main different levels exist (source: http://www.iec.org/online/tutorials/sdl/topic04.html)

The different concepts that the SDL language covers are:

- **System structure**: from the blocks to the processes and their related hierarchy.
- **Communication**: signals, communication paths or channels, parameters that can be carried out by the signals, etc.
- **Behavior**: defined by different processes.
- **Data**: based in Abstract Data Types (ADT).
- **Inheritance**: useful to describe relations between objects and their properties.

Although a textual SDL representation is possible (SDL/PR), this paper focuses on the graphical representation of the language (named SDL/GR). Anyway, the discussion is also valid for its textual representation, since both are equivalents [9] (Figure 5).



```
process P;
 start;
 nextstate idle;
 state idle;
  input s;
   output t;
   nextstate idle;
 endstate idle;
endprocess P;
```

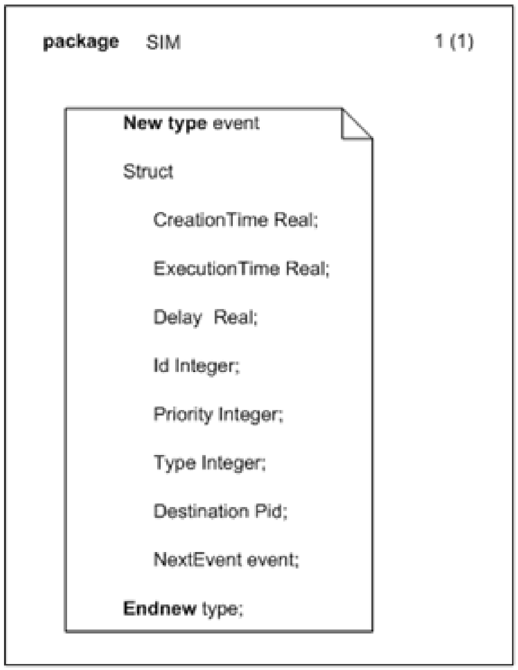
**Figure 5**: A model represented using GS-SDL and PR-SDL

More details about the Specification and Description Language can be found in the recommendation Z.100 [9] or at the web site www.sdl-forum.org. Also, some examples on the use of SDL to represent intelligent agents can be found in [10].

## 5. FORMALIZATION OF USERS USING SDL

In our simulation model, the SDL signals carry a parameter named EVENT that represents the events that rule the model evolution. This solution allows a complete definition of time, since SDL delaying channels cannot be used, because the delay cannot be fully defined [9]. For that reason, the channels to be used will be always non-delaying channels.

The *event* parameter is represented in Figure 6. An event parameter has, at least, three attributes (creation time, execution time and priority). Thanks to this parameter we can fully define the behavior of the model. Also, if needed, it is possible to employ automatic code generation using tools like Tau Telelogic of IBM[11].



```
package SIM                                   1 (1)

    New type event
    Struct
        CreationTime Real;
        ExecutionTime Real;
        Delay  Real;
        Id Integer;
        Priority Integer;
        Type Integer;
        Destination Pid;
        NextEvent event;
    Endnew type;
```

**Figure 6**: SDL SIM package containing the structure event

The formalization of a simple reflexive intelligent agent to represent UOC's users is shown in Figures 7 and 8. The first level of the SDL diagram defines the blocks or processes that compose the system. Only one kind of agents is used, the one enclosed in the *Population* block.

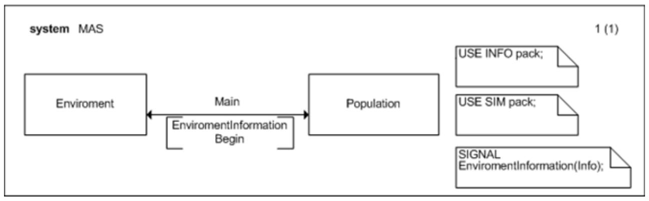
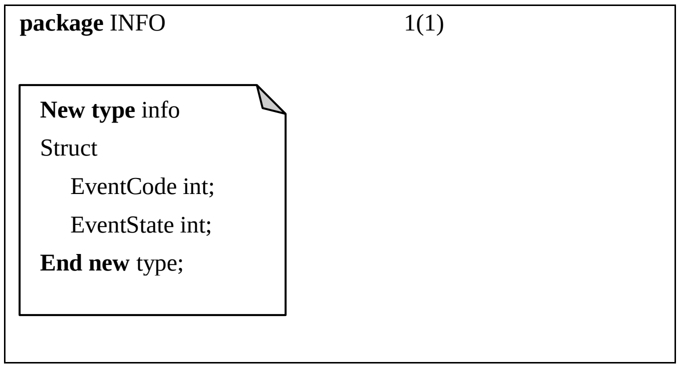


**Figure 7**: System diagram of a MAS model

As Figure 7 shows, agents in the *Population* block receive only two kinds of events: *EnviromentInformation* or *Begin*. The *EnviromentInformation* event has two parameters, i.e. *event* and *info*, being the latter a structure containing the information that agents can perceive. Figure 8 shows the *INFO* package containing the structure *info* with the definition of the different environmental elements that can be perceived by agents. In our approach, the environment is modeled by using Omnet++, while each one of the different intelligent agents represents one potential user of the system.

One major advantage of using a package that contains the *info* structure is modularity. Modularity facilitates to include in a single structure a specification of what can be perceived and what can be modified by agents. This, in turn, simplifies both the verification and validation processes. Notice that no specific behavior has been defined yet in this first level of the model structure. To define the behavior of the model (i.e. agents' behavior), other levels of the SDL language must be employed.



```
package INFO                          1(1)

New type info
Struct
    EventCode int;
    EventState int;
End new type;
```

**Figure 8**: Package "info"

Figure 9 shows the characterization of a *Population* block. Its composition is basically defined by a single process named *PSimpleRAgent*.

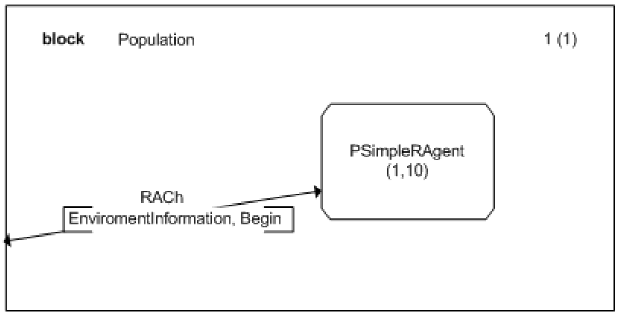


**Figure 9**: Processes contained in the Population block

The numbers (1, 10) define the minimum number of agents (at the beginning of the model execution) and the maximum number of agents allowed in the model (usually created by a *create* operation). At this point, the **structure** of the *Population* block is fully defined and, therefore, we can start the definition of the model **behavior** through the *PSimpleRAgent* process diagram.

A simple reflexive agent reacts to stimulus using its condition/action rules. However some ambiguities still exist in the textual description introduced in the previous section, e.g.: What happens if an agent that is currently executing a given action receives a new stimulus from the environment? Will the agent be able to execute both actions in parallel? Will the agent be forced to wait until its current activity is finished before starting a new one? Will the agent ignore the new stimulus?

Figures 10, 11 and 12 detail the SDL process diagram for a reflexive intelligent agent. Notice that an agent can be in any of the following states:

- **Inactive**: The agent has been created but it is waiting for a BEGIN event to start its simulation. The agent will return to this state after receiving an END signal.
- **Waiting**: The agent is either waiting for new stimulus from the environment or for a target-time to execute a new event from the events list.

- **Executing action**: At least one event is being executed by the agent.

At the beginning the agent is *Inactive*, i.e.: it ignores the entire external stimulus until a *begin* signal is received. Then, the agent changes its state to *Waiting* or *ExecutingAction*, depending on its memory status. The agent's memory stores the agent state before it became *Inactive*. When an *EnviromentInformation* signal is received, the agent starts to process this information and changes its state to *ExecutingAction*. Once the information is processed, and depending on the time defined in the procedure *ReactionTime*, the agent starts its actions, generating new signals according to the procedure *ExecutionTime*.

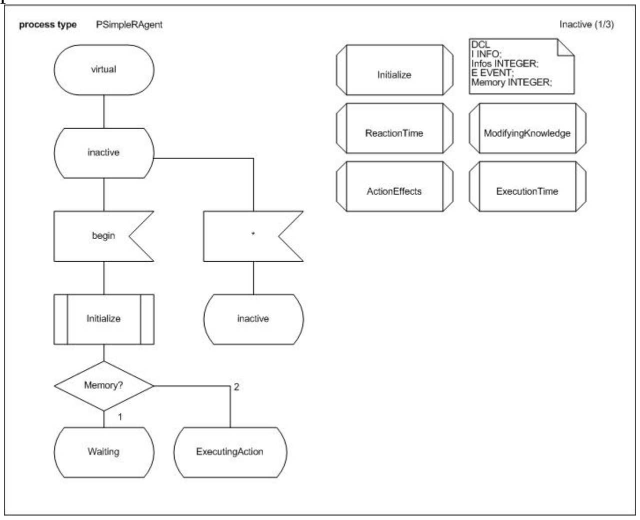


**Figure 10**: Reflexive agent *Inactive* state process diagram

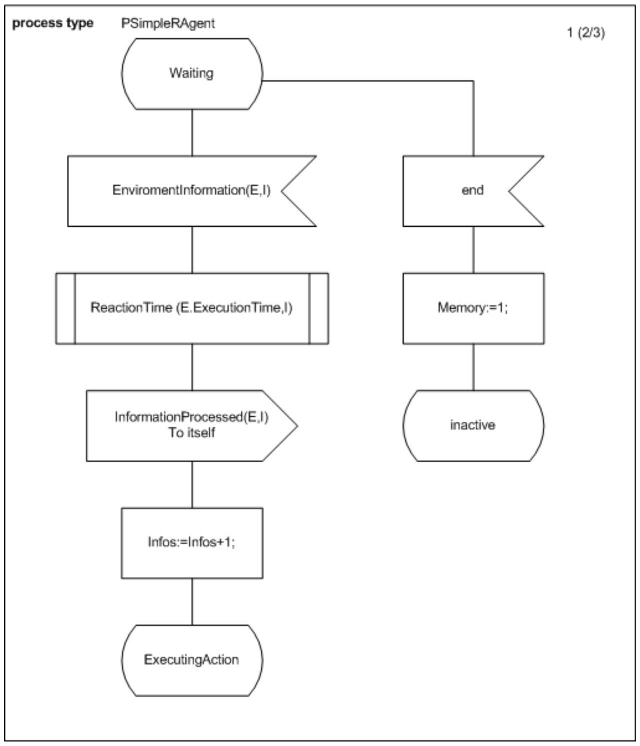

process type PSimpleRAgent
1 (2/3)
Waiting
EnviromentInformation(E,I)
ReactionTime (E.ExecutionTime,I)
InformationProcessed(E,I)
To itself
Infos:=Infos+1;
ExecutingAction
end
Memory:=1;
inactive

**Figure 11**: Reflexive agent *Waiting* state process diagram

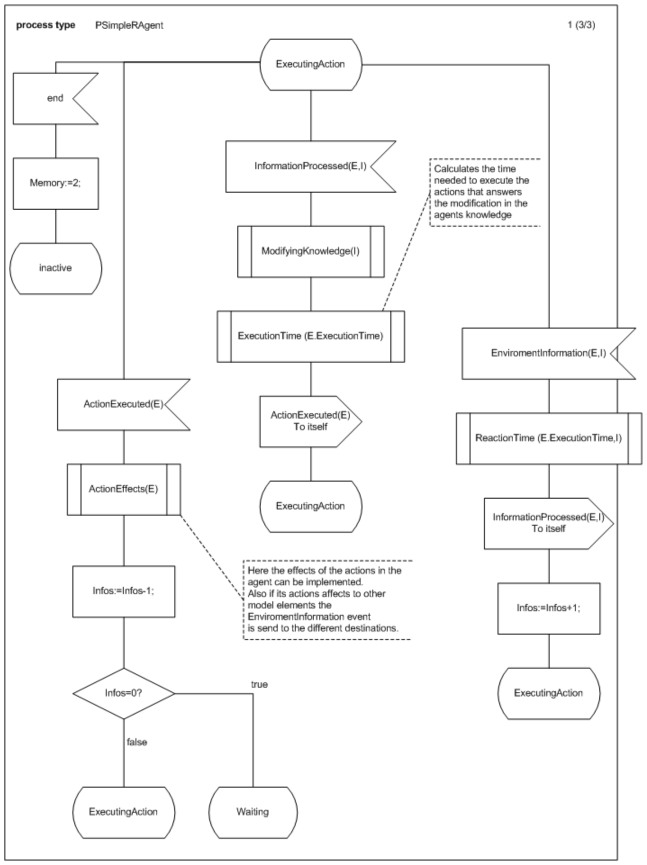

process type PSimpleRAgent
1 (3/3)
ExecutingAction
end
Memory:=2;
inactive
InformationProcessed(E,I)
ModifyingKnowledge(I)
ExecutionTime (E.ExecutionTime)
ActionExecuted(E)
To itself
ExecutingAction
Calculates the time needed to execute the actions that answers the modification in the agents knowledge
EnviromentInformation(E,I)
ReactionTime (E.ExecutionTime,I)
InformationProcessed(E,I)
To itself
Infos:=Infos+1;
ExecutingAction
ActionExecuted(E)
ActionEffects(E)
Infos:=Infos-1;
Here the effects of the actions in the agent can be implemented.
Also if its actions affects to other model elements the EnviromentInformation event is send to the different destinations.
Infos=0?
true
false
ExecutingAction
Waiting

**Figure 12**: Reflexive agent *ExecutingAction* state diagram

During the *Waiting* state, the intelligent agent is not performing any action. Only a new *EnvironmentInformation* signal can modify this state (the agent is a s*imple reflexive agent*, only reacts when it receives an environmental stimulus). As soon as a sensor obtains new information, the agent moves to the *ExecutingAction* state. The attribute *E.ExecutionTime* stores the execution time necessary for the agent to process the information. Notice that the function *ReactionTime* depends on the agent's behavior and that it is specified on the fourth level (procedures) of the SDL language. An *InformationProcessed* output event is generated and sent to the agent itself. This event contains two parameters, i.e. the *event* structure E and the *info* structure I, which defines the information received by the sensor.

As it can be noticed from the diagrams, an agent can perform different actions in parallel, since it supports the reception of new *EnviromentInformation* signals while being in the *ExecutingAction* state. If this behavior is correct for more than one project, only the last level of the SDL language will need to be defined, i.e.: the *ReactionTime* and *ExecutionTime* procedures. This, in turn, facilitates the creation of simulation-objects libraries.

For each *EnviromentInformation* signal received, the agent starts new actions. An *ActionExecuted* signal means that the agent has finished the assigned action.

## 6. THE PROCEDURES

The procedures that exist for this kind of agents are: (i) *ActionEffects*, (ii) *ReactionTime*, (iii) *Initialize*, (iv) *ModifiingKnowledge*, and (v) *ExecutionTime*. This last level of the SDL language can also be expressed by using diagrams. Figure 13 shows the *ReactionTime* procedure as an example.

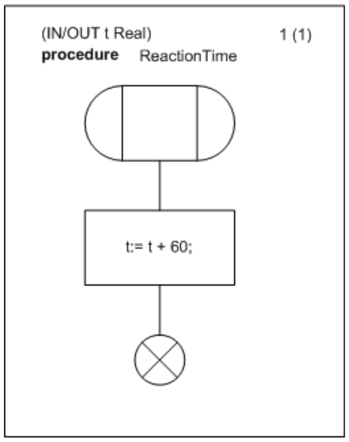
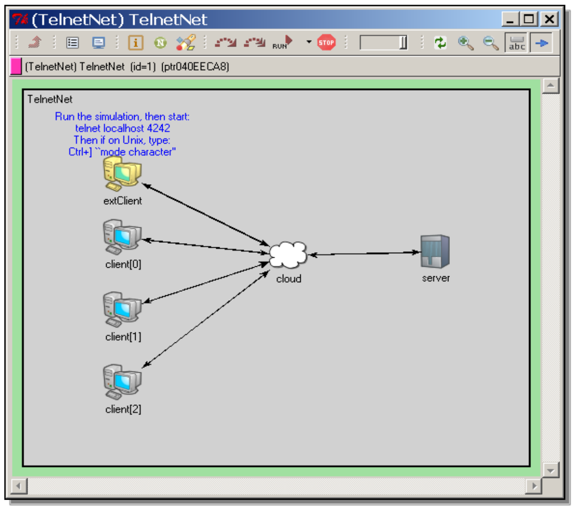


**Figure 13**: *ReactionTime* procedure

Of course, depending on the specific behavior that the agent must reflect it is possible to add new code inside a task block, decisions block, or any other element defined by the SDL language.

## 7. USING JAVA CODE WITH OMNET++

Since users' profiles are defined and implemented in Java, it becomes necessary to connect the resulting Java code with the Omnet++ network model. To this end, we employ TCP/IP using a mechanism based on sockets.



```
(TelnetNet) TelnetNet (id=1) (ptr040EECA8)
TelnetNet
Run the simulation, then start:
telnet localhost 4242
Then if on Unix, type:
Ctrl+] "mode character"
```

**Figure 14**: A network model in Omnet++

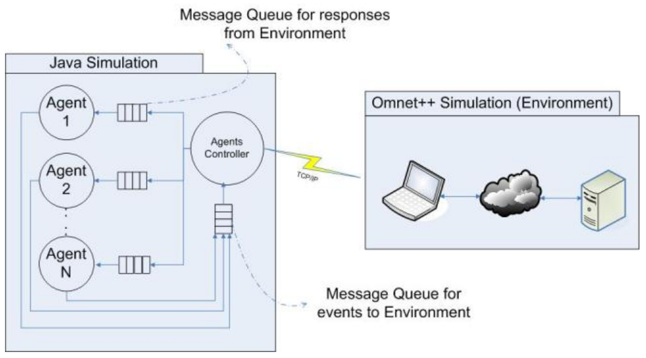
The Omnet++ general documentation includes a set of examples showing how to simulate different networks and scenarios. There are also examples on how to model a real-time network. In one of these examples (a Telnet connection), one of the simulated elements is a group of users that interact with an external model through a TCP socket. This way, external information is entered into the simulation model. This example can be adapted to allow for the simulation model we want to develop. The main goal here is to perform a real-time simulation of the behavior of the connected users on a network (Figure 14). During the simulation, the Omnet++ model receives external information from each Java-coded agent. This information can be used in the simulation model to modify its evolution through time. Also, the Omnet++ model can send information to the different external programs connected to it. The Omnet++ model implements a parser to facilitate the management of the information received. Two different alternatives were analyzed in order to implement the parser:

a) In each stream of text, the long of the stream is specified in its first positions.

b) A special character is defined at the end of each stream.

The first alternative presented some problems since some information was lost due to network errors. According to our experiences, the second alternative is more robust, so we finally used it in the final implementation of our models. However, this second alternative presents some challenges since it is necessary to ensure that the special character defining the end of the stream is not going to be used by the stream to store model information. For that reason, we used ASCII codes (larger than 32) for the initial sequence while using codes lower than 32 for the final sequence. The final-of-stream sequence was defined with the stream "\r\n\r\n". Therefore, once an external agent connects to the Omnet++ model, the communication between both programs follows the aforementioned protocol. In other words, the human behavior of a client that uses its web browser to communicate with the Castelldefels system is modeled and simulated. In the stream we define the kind of event according to the classification from Table 1, but no other information –e.g. data related to the event such as the delays associated with any data transmission– is added. However, the system is ready to automatically integrate this information into the model.

Apart from the external and internal models, special attention must be paid to the communication between external agents and the Omnet++ environment. This communication has been implemented using FIFO queues. First, each agent prepares the events that it must send to the environment. This task depends on its particular profile. For each event, a creation time is defined. Each agent initiates the communication process with the environment as a reaction to the signals received from the model (Figure 15).



Message Queue for responses from Environment
Java Simulation
Agent 1
Agent 2
Agent N
Agents Controller
TCP/IP
Omnet++ Simulation (Environment)
Message Queue for events to Environment

**Figure 15**: System architecture

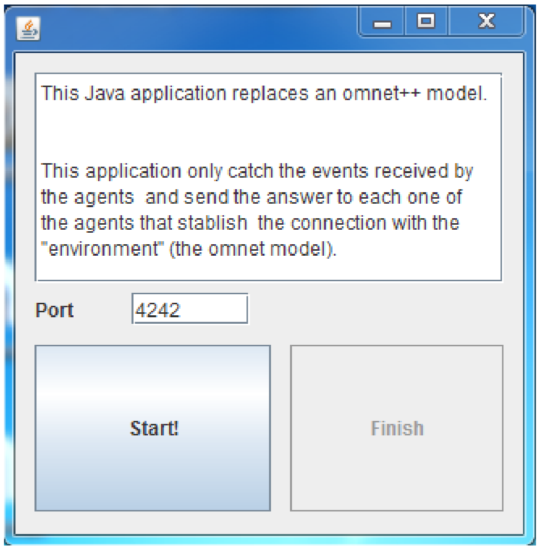
Each stream is characterized by the following set of parameters:

- **Identifier**: Incremental value that identifies the stream.
- **Origin**: Integer value between 1 and 99999 defining the agent that generates the stream.
- **Destination**: Identifies the destination environment.
- **Kind**: Identifies the nature of the message according to Table 1. It is a number between 1 and 999.
- **Priority**: Defines the priority level of each event. It is a number between 1 and 99. In the current version of our model, all events have the same priority (priority 1).
- **State**: Defines the state of each event according to Table 2.
- **Delay**: Defines the maximum amount of time that we can wait for an answer of an event that must be returned by the environment. The agents try to send again the event the times defined in the "retry" value of the event (5 chars).
- **Creation time**: Represents the time at which each event must be processed. It is defined by a 14-character stream (hhmmssddmmaaaa).
- **Execution time**: Stores the time at which each event was executed. It is defined by a 14-character string (hhmmssddmmaaaa).
- **Retry**: Number of times that an event can be resubmitted. It is represented by a 2-character string.

## 8. RUNNING THE MODEL

The model was executed in a platform under the following configuration:

- Software: Windows XP SP3, NetBeans Development IDE 6.7.1, JDK 1.6, Omnet++ 4.0, Mingw.
- Hardware: Processor Pentium 4 with 2Gb of RAM.



This Java application replaces an omnet++ model.

This application only catch the events received by the agents and send the answer to each one of the agents that stablish the connection with the "environment" (the omnet model).

Port 4242

**Figure 16**: The SimOmnet tool

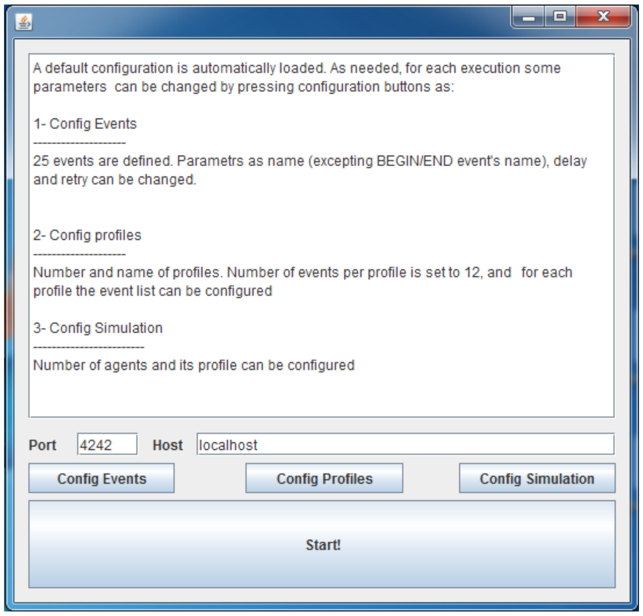
Obviously, executing the Environment (Omnet++ simulator) and the agents (Java code) in the same computer reduces the computer performance. Additionally, if the agents' population increases over 1000 individuals, different errors appear, which are mainly due to an overflow of messages.

Regarding the delays, these are mainly caused by the environment limitations as can be checked by using the SimOmnet tool (Figure 16). SimOmnet is a Java program that emulates the Omnet++ model by simply returning all the messages that come from the agents. In this case, the configuration file used for Omnet++ is:

```
[Config TelnetExample]description = "Telnet model"
network = TelnetNet
**.numClients = 0
**.cloud.propDelay = 0.01s
**.server.serviceTime = 0.01s
**.client[*].sendIaTime = exponential(3s)
```

In the case of the Omnet++ simulation, some errors appear when more than 1000 agents are used. In the case of SimOmnet, similar errors appear when more than 4500 agents are used. Obviously, using a more powerful hardware configuration these restrictions could be softened somewhat. However, some of these problems are caused not by the hardware configuration itself but by the delays in the messages' acknowledgment. Therefore, it could be possible to use a larger number of agents in the simulation by simply assigning less time to the maximum allowed delay.

Figures 17 and 18 show the Java simulator implementing the behavior of the agents.



A default configuration is automatically loaded. As needed, for each execution some parameters can be changed by pressing configuration buttons as:

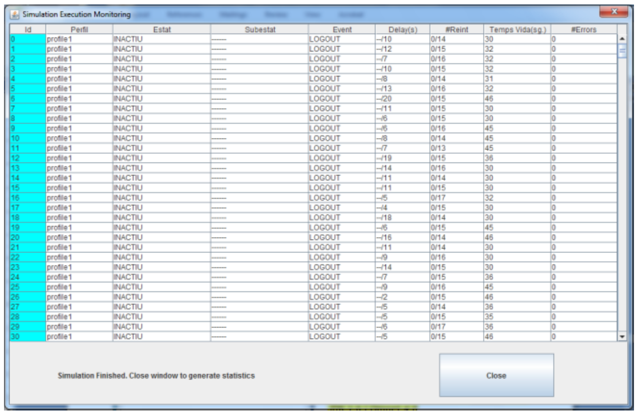
1- Config Events
--------------------
25 events are defined. Parametrs as name (excepting BEGIN/END event's name), delay and retry can be changed.

2- Config profiles
--------------------
Number and name of profiles. Number of events per profile is set to 12, and for each profile the event list can be configured

3- Config Simulation
------------------------
Number of agents and its profile can be configured

Port 4242 Host localhost

**Figure 17**: Java simulator implementing the agents' behavior



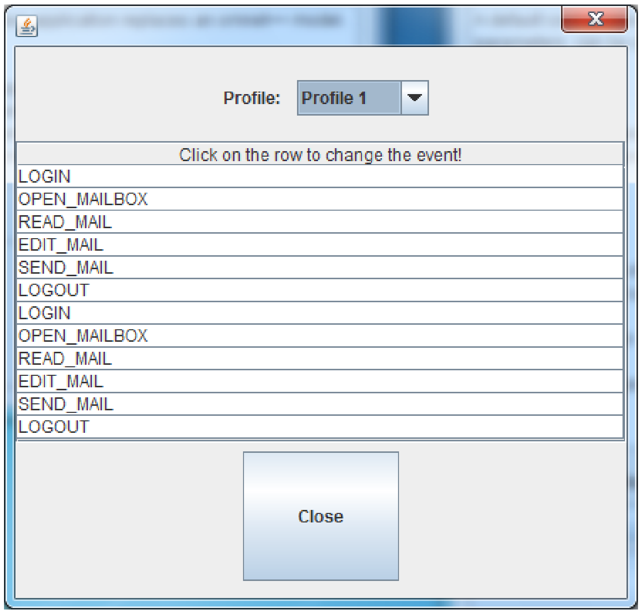
Simulation Execution Monitoring

| Id | Perfil | Estat | Subestat | Event | Delay(s) | #Reint | Temps Vida(sg.) | #Errors |
|---|---|---|---|---|---|---|---|---|
| 0 | profile1 | INACTIU | ------- | LOGOUT | --/10 | 0/14 | 30 | 0 |
| 1 | profile1 | INACTIU | ------- | LOGOUT | --/12 | 0/15 | 32 | 0 |
| 2 | profile1 | INACTIU | ------- | LOGOUT | --/7 | 0/16 | 32 | 0 |
| 3 | profile1 | INACTIU | ------- | LOGOUT | --/10 | 0/15 | 32 | 0 |
| 4 | profile1 | INACTIU | ------- | LOGOUT | --/8 | 0/14 | 31 | 0 |
| 5 | profile1 | INACTIU | ------- | LOGOUT | --/13 | 0/16 | 32 | 0 |
| 6 | profile1 | INACTIU | ------- | LOGOUT | --/20 | 0/15 | 46 | 0 |
| 7 | profile1 | INACTIU | ------- | LOGOUT | --/11 | 0/15 | 30 | 0 |
| 8 | profile1 | INACTIU | ------- | LOGOUT | --/6 | 0/15 | 30 | 0 |
| 9 | profile1 | INACTIU | ------- | LOGOUT | --/6 | 0/16 | 45 | 0 |
| 10 | profile1 | INACTIU | ------- | LOGOUT | --/8 | 0/14 | 45 | 0 |
| 11 | profile1 | INACTIU | ------- | LOGOUT | --/7 | 0/13 | 45 | 0 |
| 12 | profile1 | INACTIU | ------- | LOGOUT | --/19 | 0/15 | 36 | 0 |
| 13 | profile1 | INACTIU | ------- | LOGOUT | --/14 | 0/16 | 30 | 0 |
| 14 | profile1 | INACTIU | ------- | LOGOUT | --/11 | 0/14 | 30 | 0 |
| 15 | profile1 | INACTIU | ------- | LOGOUT | --/11 | 0/15 | 30 | 0 |
| 16 | profile1 | INACTIU | ------- | LOGOUT | --/5 | 0/17 | 32 | 0 |
| 17 | profile1 | INACTIU | ------- | LOGOUT | --/4 | 0/15 | 30 | 0 |
| 18 | profile1 | INACTIU | ------- | LOGOUT | --/18 | 0/14 | 30 | 0 |
| 19 | profile1 | INACTIU | ------- | LOGOUT | --/6 | 0/15 | 45 | 0 |
| 20 | profile1 | INACTIU | ------- | LOGOUT | --/16 | 0/14 | 46 | 0 |
| 21 | profile1 | INACTIU | ------- | LOGOUT | --/11 | 0/14 | 30 | 0 |
| 22 | profile1 | INACTIU | ------- | LOGOUT | --/9 | 0/16 | 30 | 0 |
| 23 | profile1 | INACTIU | ------- | LOGOUT | --/14 | 0/15 | 30 | 0 |
| 24 | profile1 | INACTIU | ------- | LOGOUT | --/7 | 0/15 | 36 | 0 |
| 25 | profile1 | INACTIU | ------- | LOGOUT | --/9 | 0/16 | 45 | 0 |
| 26 | profile1 | INACTIU | ------- | LOGOUT | --/2 | 0/15 | 46 | 0 |
| 27 | profile1 | INACTIU | ------- | LOGOUT | --/5 | 0/14 | 36 | 0 |
| 28 | profile1 | INACTIU | ------- | LOGOUT | --/5 | 0/15 | 35 | 0 |
| 29 | profile1 | INACTIU | ------- | LOGOUT | --/6 | 0/17 | 36 | 0 |
| 30 | profile1 | INACTIU | ------- | LOGOUT | --/5 | 0/15 | 46 | 0 |

Simulation Finished. Close window to generate statistics

**Figure 18**: Executing the model with 1000 agents

Profiles define the behavior of the agents. Different profiles are represented in this model, as shown in Figure 19.



Profile: Profile 1

| Click on the row to change the event! |
|---|
| LOGIN |
| OPEN_MAILBOX |
| READ_MAIL |
| EDIT_MAIL |
| SEND_MAIL |
| LOGOUT |
| LOGIN |
| OPEN_MAILBOX |
| READ_MAIL |
| EDIT_MAIL |
| SEND_MAIL |
| LOGOUT |

**Figure 19**: Agents profile configuration

## 9. CONCLUDING REMARKS

In this paper we propose a methodology that allows to combine users' profiles using a graphical or general-purpose language (SDL or Java) with a complete tool like Omnet++ that models a complete computer network. This way, a complete model, including hardware, software and human factors, can be constructed and simulated.

Thanks to the use of a graphical language to define agents' behavior, different specialists can easily collaborate in the modeling of the human factor. This facilitates the participation of researchers without programming skills in the simulation projects.

We have used Omnet++ to develop a preliminary model of the computer system that gives support to the UOC Virtual Campus. We plan to develop a similar work with other simulation packages, like Opnet, in order to perform a comparative analysis.

## 10. REFERENCES

[1] Juan, A. A., Marquès, J. M., Vilajosana, X., Faulin, J., & Fonseca, P. (2008). MODELLING & SIMULATION OF A VIRTUAL CAMPUS,A Case Study Regarding the Open University of Catalonia. ICEIS 2008 – Tenth International Conference on Enterprise Information Systemsc. Barcelona, Spain: INSTICC Press.

[2] Law, A. M., & Kelton, W. D. (2000). Simulation Modeling and Analysis. McGraw-Hill.

[3] Banks, J., & Gibson, R. (1997, February). Simulation modeling: some programming required. IIE Solutions , 26-31.

[4] Kurose, J., Ross, K., 2005. Computer Networking: A Top-Down Approach Featuring the Internet. Addison-Wesley

[5] Peterson, L., Davie, B., 2003. Computer Networks. A Systems Approach. Morgan Kaufmann

[6] Cortés, U., Béjar, J., & Moreno, A. (1994). Inteligencia Artificial. Barcelona: Edicions UPC.

[7] Reed, R. (2000). SDL-2000 form New Millenium Systems. Telektronikk 4.2000 , 20-35.

[8] SDL Tutorial. (n.d.). Retrieved January 2009, from IEC International Enginyeriing Consortium: http://www.iec.org/online/tutorials/sdl/

[9] Telecommunication standardization sector of ITU. (1999). Specification and Description Language (SDL). Retrieved April 2008, from Series Z: Languages and general software aspects for telecommunication systems.: http://www.itu.int/ITU-T/studygroups/com17/languages/index.html

[10] Fonseca i Casas, P. (2008). SDL, A Graphical Language Useful to Describe Social Simulation Models. In F. J. Quesada (Ed.), 2nd Workshop on Social Simulation and Artificial Societies Analysis (SSASA'08). Barcelona.

[11] IBM. (2009). TELELOGIC. Retrieved 03 31, 2009, from http://www.telelogic.com/