GALATEA: A multi-agent, simulation platform

Jacinto Dávila[1] and Mayerlin Uzcágegui[1,2]

[1] CeSiMo (Centro de Simulación y Modelado),
[2] SUMA (Sistema Unificado de Microcomputación Aplicada),

Universidad de Los Andes, Mérida 5101, Venezuela

Abstract. This paper presents a new simulation platform: GALATEA, which is offered as a language to model multi-agent systems to be simulated in a DEVS, multi-agent platform. GALATEA is the product of two lines of research: simulation languages based on Zeigler's theory of simulation[Zeigler: 1976,Zeigler: 1990] and logic-based agents[Dávila: 1997]. There is in GALATEA a proposal to integrate, in the same simulation platform, conceptual and concrete tools for multi-agent, distributed, interactive, continuous and discrete event simulation. It is also GALATEA a direct descendent of GLIDER[Domingo et al: 1985,Domingo: 1988], a DEVS-based simulation language which incorporated tools for continuous modelling as well. In GALATEA, GLIDER is combined with a family of logic programming languages specifically designed to model agents[Dávila: 1997].

1 Introduction.

The possibility of modelling and simulating entities that perceive their environment, reason upon those perceptions and certain background of knowledge, and then act upon that environment (i.e. as Agents in AI[Russell and Norvig: 1995]) opens the door to a number of extensions to the established uses of simulation. Economic modelling can, with agents, take into account new and, arguably, more precise characterisations of human beings. This way of modelling economic agents may become an alternative to more traditional mathematical models employed in economics. Those traditional descriptions of human beings normally exclude, for the sake of tractability, fundamental aspects such as qualitative descriptions of the agents' goals and intentions, beliefs and other attributes of human reasoning (e.g. bounded rationality).

However, we do no aim at replacing traditional modelling techniques with a new modelling language, but instead we try to integrate well-known approaches (DEVS modelling and its extensions) together with a new approach that allow us to include, in a simulation model, those reasoning-acting entities: the agents. By appealing to engineering techniques from Artificial Intelligence and Logic Programming, we are capable of producing a light reason engine to simulate each agent observe-reason-act process. With as many agent engines as required for the particular application and with the traditionally conformed simulator for the environment in which those agents are placed (the main simulator), we create a federation (In HLA's sense[HLA: 1995]) which serves as the simulator for the whole multi-agent system.

GALATEA architecture is based on objects. Both the agents and the main simulator are designed according to a OO layout to support distribution (of objects), modularity, scalability and interactivity as demanded by the HLA specification. Are aiming at a flexible platform from the software engineering point of view (which is, arguably, inaccessible to final users: the modellers) but based on a family of modeller-friendly languages with enough expressiveness to allow the modellers to describe a multi-agent system in a way that makes feasible its simulation. We think that this possibility is critically dependent on domain and application specific trade-offs. Thus, we allow the modellers to describe systems in which there are agents, but in which not everything is an agent and in which traditional discrete-event or continuous modelling techniques are good enough for most purposes (such as dealing with subsystems that required very aggregated models to make their simulation feasible at all).

This paper is structured as follows. We start by presenting GALATEA OO layout, in UML. We then briefly introduce the syntax of the simulation language and, to illustrate the intended contribution,

we transform a DEVS model of a teller system (as modelled with GLIDER), into a multi-agent model (with GALATEA).

2 GALATEA Simulation Platform.

The simulation platform GALATEA integrates the concepts and tools that allow simulating systems under the distributed, interactive, continuous, discreet and combined focuses. Also, in this platform, we incorporate support for modelling and simulation of multi-agents systems.

The simulation platform GALATEA consist of:

- a family of languages that allow to simulate systems multi-agents,
- an compiler
- a framework for the construction of models

This platform is being developed in pure Java and runs on every platform supporting a Java implementation. It has been tested on Windows and Linux.

GALATEA requires a reformulation of the traditional way of managing the relationship among the simulation components. Aiming at interactive simulation, it is necessary to think of a number of programs executing in heterogeneous and distributed computers and interacting with each other through a distributed, operating system. We exploit these features to allow modelling and simulation for multi-agents systems.

Advances in Interactive Simulation have allowed the development of a standard framework simulations with many different simulation components. This framework is called HLA (*High Level Architecture*) [HLA: 1995,Dahmann et al.: 1999], we used it as reference for the design of our platform.

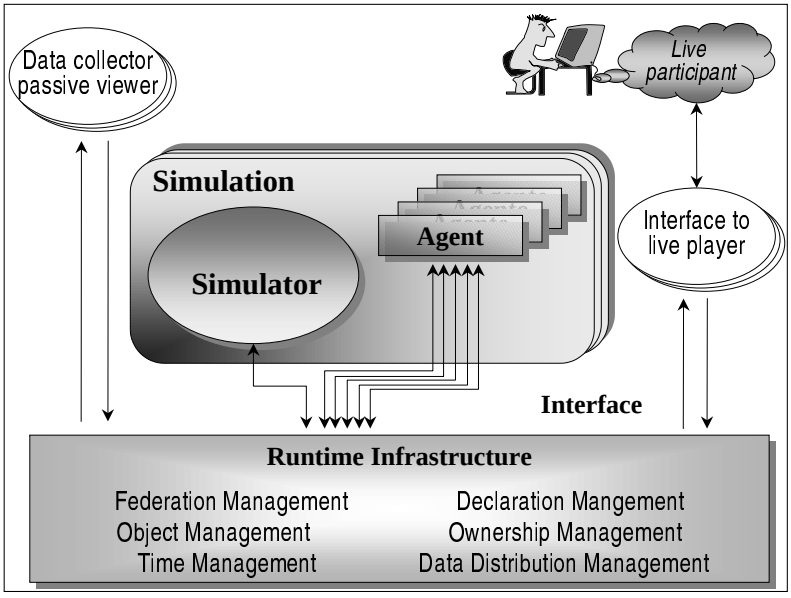

Fig. 1. Architecture of the simulation platform GALATEA

Figure 1 shows the structure of the federation HLA, used in GALATEA, divided in their main functional components. The first component aggregates all the simulations (as defined by HLA), which should incorporate specific capacities that allow the objects of a simulation, *federates*, to be

connected with the objects of another simulation. The exchange of data among federates is supported by the services implemented in the interface. We customise HLA simulation to include:

1. a federate that represents the unique simulator controlling all the events. The use of a unique simulator can yield substantial benefits, including:
   - improved validation of the behaviour of existing systems and reused of existing models,
   - a rich infrastructure for developing new systems,
   - the opportunity to study large-scale systems interaction in a controlled environment, and
   - easier comparison of results across research efforts.
2. a group of federates that represent the agents of the system: A agent in GALATEA is a entity that perceives its environment, assimilates perceptions, remembers, reasons on what it knows, adopts beliefs, goals and intentions for itself and it acts, modifying its environment through its effectors.

The second functional component of the architecture is the Runtime Infrastructure (*RTI*). This infrastructure provides a group of general-purpose services that support the interactions among the simulation components. All the interactions among the simulation components are carried out through the RTI.

The third component is the interface to the RTI. This interface, independent of their implementation, provides a standard way for the simulation components to relate with the RTI. The interface is independent of the requirements for modelling and exchange of data from the simulation components. In our case, in particular, it is necessary to incorporate the form in which the agents perceive and modify their environment. The unique simulator controls all data exchange between an agent and its environment. This allows a first, easy solution for control of concurrent events and agent interaction.

Additionally, HLA specifies three basic aspects for the simulations. First, there is a set of rules that simulation components must comply separately and as a group. Second, a specification of the interface that defines the services and the functions of each component and, finally, a common model to store information.

In HLA, each simulation component manages a logical local time, counter, under certain restrictions:

- it is necessary to define the initial and final values for the counters.
- the values associated to the time are independent of the real-time units.
- the current time during a simulation is always greater than the initial time.
- time's management is in discreet form.
- it is necessary to co-ordinate the synchronisation between the logical time counter and the real time.

In general, handling of the time can be done according one of the following strategies:

Local Management. Each component controls the advance of the time.

Conservative Synchronisation. The advance of time of each component is carried out when this guaranteed that last events wouldn't be received.

Optimistic Synchronisation. Each component is free of advancing its local time, but it should be prepared to return its logical time if it receives a last event.

Supervision of activities. The components control the local time in periods. At the end of every period messages are exchanged until they agree to advance the logical time together entering to next period.

GALATEA, for the time being, is handling time by supervision of activities wherein concurrence is dealt with at modelling's time. Simulation control points are regarded as events. These control points are special devices to manipulate agent's perceptions and actions as the simulation progresses.

Also, as in the classic DEVS simulation, variables that represent the state of the system are updated after each advance of time.

The figure 2 shows, in a UML sequence diagram, the simulation execution process of GALATEA. It specifies the sequence corresponding to scheduler and activation of events (denoted with *), including the recursive process of scanning of the network (denoted with **) which includes ineurl the exchanges triggered by the activated event in the system. The components associated the simulation execution process are:

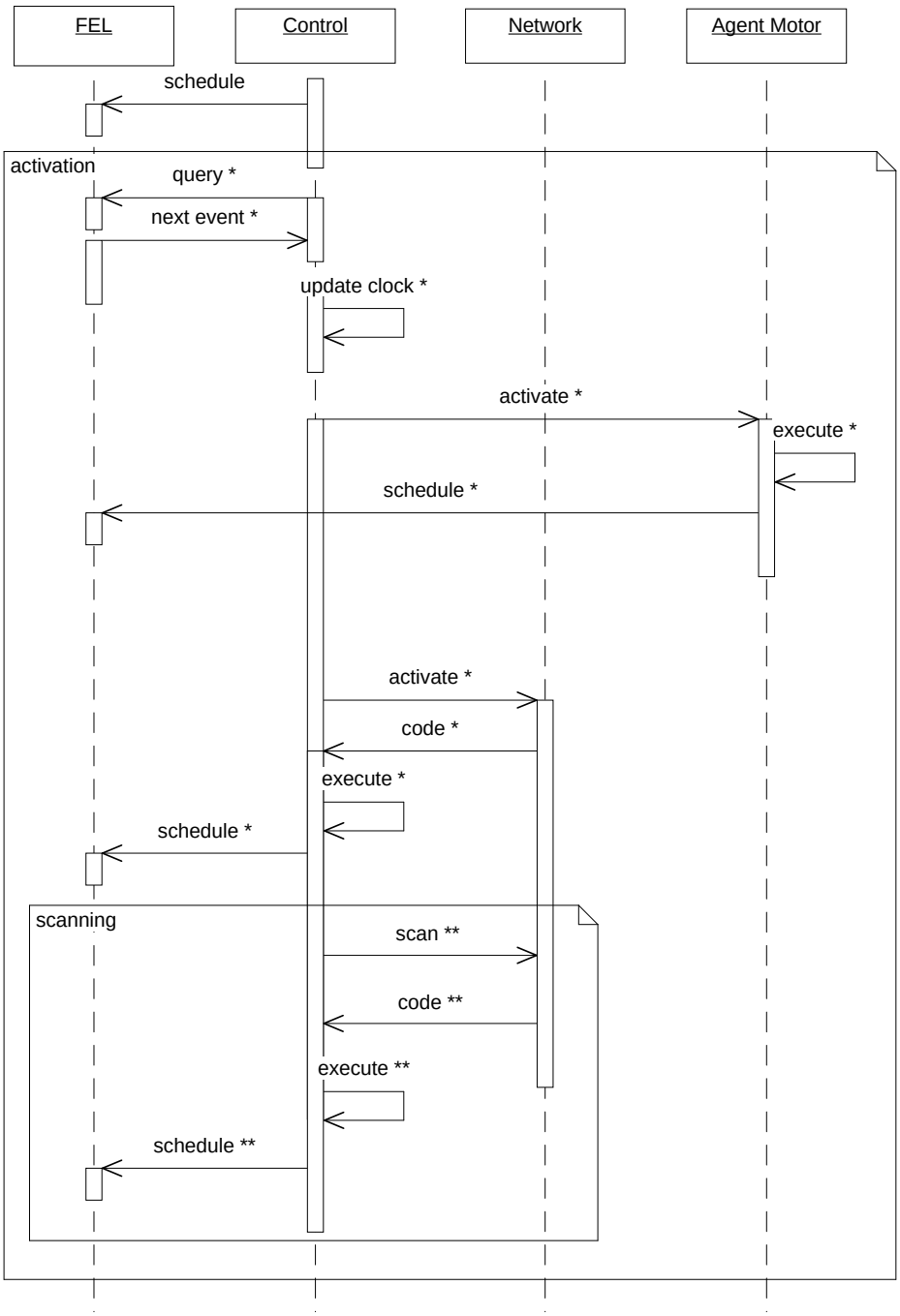

Fig. 2. GALATEA simulation process

FEL (*Future Event List*). Component that contains the events in chronological order.
Control. This component controls the simulation process:

- query next event
- update clock
- activate the Agent Motor
- activate the Network
- execute the code associated to the networks' nodes
- scheduler new events
- management the Network recursive scanning

Network. This component represents the subsystems of the simulation systems (see section 3). This has function of management the node interaction and activation.

Agent Motor. There are one for each agent in the system. These components have the function of controlling the activities associated with the agents:

- perception of the environment
- assimilation of the perception
- remember
- reasoning
- adoption of the beliefs, goals and intentions
- modification of its environment

3 Languages and Simulator.

The family of languages that GALATEA conforms includes the basic foundations of the language GLIDER and a group of languages based on logic that will allow describing the agents.

3.1 Foundations of GALATEA.

The language GLIDER allows the specification of simulation models for continuous systems and of discreet events. In their design, they have taken in account facilities for the simultaneous execution and co-operative of both types of systems.

GLIDER represents the modelled system as a group of subsystems that exchange information. Each subsystem can store, transform, transmit, create and eliminate information from the system state, as each has a procedural code linked to it. This code specifies how the system state must change when an event on a subsystem occurs.

The fundamental characteristics of the GLIDER are:

- Nodes of a net represent the subsystems.
- The nodes can be of different types according to their function. The following are the basic types of nodes:
  - `G (Gate)`: it controls flow of messages.
  - `L (List)`: it simulates queuing disciplines.
  - `I (Input)`: it generates input messages.
  - `D (Decision)`: it selects messages according to suitable selection rules.
  - `E (Exit)`: it destroys the messages.
  - `R (Resource)`: it simulates resources used by entities in the system (messages).
  - `C (Continuous)`: it solves a system of first-order, ordinary differential equations and, in the process generates the behavioural states of a system modelled by that continuous model.
  - `A (Autonomous)`: it can be customised to simulated special events.
  - `O (Other)`: it is similar to A nodes, but with no structural constraint on execution.
- The information is described and executed through a code associated to each node.

– The exchange of information among nodes can be performed through global variables, shared files and message passing.
– The information is stored in variables, files or lists of messages.
– Each node has associate a group of predecessors nodes from which can receive messages and a group of successors nodes towards which it can send messages.

The first compiler for GLIDER was developed in PASCAL and, this made that the languages syntax is based on TurboPascal 6.0 Borland©. The design allows the user-programmer to introduce its own pieces of code (functions and procedures) besides predefined structures, procedures and functions from GLIDER.

Compiler's prototype for GALATEA, developed in Java, preserves most the syntax of the language GLIDER and incorporates the facilities and structures of Java programming instead of PASCAL.

3.2 GALATEA program structure.

The figure 3 show the structure of a GALATEA program. It consists of six sections:

```
            header
NETWORK
            description of the net
AGENTS
            description of the agents
INTERFACE
            description of the relationships between agents and the environment
INIT
            initialising of variables and structures to be used in the simulation
DECL
            declaration of Variables
END.
```

Fig. 3. Structure of a GALATEA program

Header: This section contains a title and description of the system, as well as some comments.

Net: This section begins with the `NETWORK` label. It is where the nodes, the relations among them and the code that guides the simulation are described.
Each node has two parts: 1) a *header* where the name, the type, the multiplicity, the successors and the local variables are defined and 2) a *code* made up of constructs from Java and GLIDER.

Agents: This section begins with the `AGENTS` label and contains description of each agent type. Each agent type, in turn, contains for a particular (type of) agent, as explained in [Dávila and Tucci: 2000]

Interface: This section begins with the `INTERFACE` label. In this section the description of the relationships between agents and the rest of the system is included.
This section contains Java code that describes what the simulator does to let the agents perceive and act.

Initiation: This section is indicated with the `INIT` label, here they are carried out assignment of values that they will be used in time of execution of the simulation, such as:
– Values initials to arrangements.
– Values initials for the capacities of the nodes type R.
– Values of functions multivalued.
– Values of parameters in charts of frequency.

– Values of arrangements starting from database charts.
– Initial Activation of nodes.

Declaration: This section begins with the `DECL` label and according to the declaration type to carry out is necessary to use the label:

`TYPE`: Types of defined data for the users.
`CONST`: Constant.
`VAR`: Variables.
`NODES`: Nodes defined by the users.
`MESSAGES`: Messages.
`GFUNCTIONS`: Multivalued functions defined by the users.
`TABLES`: Charts of frequency.
`DBTABLES`: Charts coming from database.
`PROCEDURES`: Procedures and functions defined by the users.
`STATISTICS`: Application of statistics for nodes and variables.

As all program GLIDER should culminate with the reserved word `END.`

3.3 A general algorithm for the simulator.

A simulation of a system is, essentially, the production of a sequence of events in the simulated environment. Therefore, what the simulator does is, essentially, the triggering of those events and the execution of pieces of code associated with each of them. On could argue that an event is precisely the execution of that piece of code. However, there are cases that do not fit to this definition. Thus, in GLIDER an event is the activation of a node that may or may not imply the execution of its code.

A simulation of a GLIDER model is the triggering of events related to the nodes and then the scanning of the network, looking for the code to be executed and testing its structural preconditions. If these preconditions hold, the code must be executed to guarantee that effects (changes) induced by each event are propagated through all the system. The activation of a node (the occurrence of an event with the node name) may trigger new events and may also imply changes on the system global state and movement of messages.

Thus, the scanning of the network drives the evolution of the system simulation. When a node is activated, its code is executed and the scanning of the rest of the network begins.

During the scanning, those nodes that do not have a external, waiting list for messages (an *EL*) or whose *EL* is empty are ignored. These are the nodes where nothing can happen (as there are no entities in them).

The scanning proceeds visiting nodes (an executing those with entities in them) until it reaches the node where the scanning started (the activated node, provided it completes a whole round without movements of messages (As this may imply that other nodes are then ready for execution).

Once a scanning is finished, the simulator looks for another entry from the Future Event List (*FEL*), to see which is the next event to be executed. The *FEL* is, of course, a collection of events: nodes labels ordered by the time-points for their scheduled execution.

This is GLIDER's general algorithm. GALATEA preserves this general strategy, except that, based on the specification provided by the multi-agent simulation theory in [Dávila and Tucci: 2000], it provides for the running of an inference engine for each agent. All these engines run concurrently with the main simulator and their execution is carefully interleaved, so that they can exchange information and effectively simulate a multi-agent system. The engines provide the simulation of the cycle of perceiving, reasoning and acting by each agent. The simulator controls the evolution of the (physical) system.

This arrangement in which a simulation system is made up from distributed components is not a new idea, of course. In fact, we are profiting on an exciting specification for distributed, interacting

simulator arranged in a “federation”. HLA (High Level Architecture) specifies the minimal conditions a set of component must fulfill in order to become a federation. In our case, the simulator and each of the agents is a federate. They joint up on a multi-agent federation using a simple, centralised, time-management strategy to control their own evolution. In particular, information exchanges are always updates to the Future Event List, *FEL* (or set of influences, as we called elsewhere in this volume. Agents feed their intentions (actions they wanted executed) into the *FEL* and the simulator decides the outcome. Similarly, the simulator put the appropriated percepts for each agent in the *FEL*, and each agent will “decide” what they want to see from it.

This is the general algorithm for GLIDER and for GALATEA. We now turn to illustrate the architecture and later to explain how it actually works, with one example.

4 The Teller example.

4.1 A very elementary GLIDER example.

In GLIDER, we can simulate a teller system (e.g. a bank) with a model like as show in the figure 4.

```
NETWORK
   Input (I) ::
      IT:= 10;
      SENDTO(Teller[MIN]);
   Teller [1..3] (R) ::
      STAY:= 45;
   Exit (E) ::
INIT
   TSIM:= 300;
   ACT(Input, 0);
END.
```

Fig. 4. Teller system

The `NETWORK` section describes the system’s structure. ”Nodes” represent components of that structure. In the example, the inputting subsystem, responsible for the arrival of entities into the system (e.g. clients getting into the bank) is modelled by the `Input` node (of type I). The node has a set of associated instructions, which dictate what happens when an input event occurs. In this case, for every arrival, another input event is scheduled to occur exactly `IT` units of times after the current time (`IT`: interarrival time). Every arriving message is then sent to (by `SENDTO(Teller[MIN])`) to the teller which is represented by an `R` (Resource) node.

The statement `Teller[1..3] (R) ::` says that there are three tellers in the system. As soon as a message traverses the `Input` node is placed into the waiting line in front to the teller with the minimum number (the `MIN` in `Teller[MIN]`) of messages already queued. Thus, the teller is a queue plus a serving unit. The simulationist tells the simulator that each message must stay (`STAY=45`) for `45` units of time while is served, once they reach the head of the queue. As soon as they are served, messages are delivered to the `Exit` node where they get discarded from the system.

This very elementary example illustrates the several features of traditional simulation techniques. A modeller describes a perfectly defined structure for the system. The structure has an associated dynamics which gets produced by the execution of certain predefined pieces of code (the instructions next to the `::`). This structure remains the same throughout the simulation (and until the modeller changes the description, that is, the model).

With GALATEA, we are aiming at another level of description: one which takes AGENTS into account and, through them, allows certain kind of changes in the structure of the system. Agent

modelling and structural change are exciting but difficult concepts to discuss. We prefer an argument illustrated by an example: an extension of the Teller example. We explain below how it incorporates AGENT modelling and changes of structures.

4.2 The teller example revised.

The more fundamental different in GALATEA with respect to GLIDER is the addition of agents to the semantics. This has many subtle, but important, implications: 1) With agents, we have a concurrent model of computation for simulation; 2) we need to explain what agents are, both at an abstract level (for the modeller) and at concrete level (for the simulationist); 3) we have to device languages and strategies to program agents to do things.

As we explained before, in GALATEA we preserve the `NETWORK`, `INIT` and `DECL` sections as in GLIDER models. We add the sections `AGENTS` and `INTERFACE`. The former contains the specification of the agent's internal, mentalistic state (knowledge base, procedures, goals and preferences). Here is where the code to guide each type of agent is placed. The `INTERFACE` section relates agents' actions and perceptions to the actual states of the world. Here the modeller describes what it means for and agent action to be executed or when and why an agent perceives such and such properties of its environment.

We have modified the Teller example to include the Clients and a Manager's "rationalities" (figure 5). Clients are no longer dependent entities that are moved around in the network. Instead, they can reason upon perceptions and an internal state (specific for each represented entity) and then act upon the environment (in this case, very elementary speech acts are performed). The manager, an implicit agent, also has reasoning and acting capabilities. Its influence is much more transcendental than that of the clients. The manager can actually change the structure of the system by adding or eliminating tellers.

Let us see how it works in detail (and by contrasting it against the original teller model above). The `NETWORK` section has just one change: the instruction `setAgent(Client)` tells the simulator that the entity just passing by (the message going through the node) is an agent of type Client, whose internal, mentalistic state is described in the `AGENT` section (precisely next to the `Client ::` label). This is GALATEA's way to transform entities that traverse the network (normal messages) into `AGENTS`. This implies that the simulation system is allocating, for each message so treated, an inference engine that runs in parallel with the simulator itself. The inference engine is (in charge of) the agent's reasoning.

There is, however, a second less explicit way to introduce agents into a simulation model (in GALATEA): one can declare agent entries in the `AGENT` section as "static" (as we do to the Manager). This means that we now have an agent-reasoning engine which is not attached to any entity that moves around, but that also has its own internal state, reasons with it and can change the state of the world with its actions.

The `AGENT` section itself is a collection of knowledge representation devices. For each `AG` entry, there is a subsection to describe agent's `GOALS`, another to describe agent's `BELIEFS` and also (though not used now), a section to describe agent's `PREFERENCES`. Each one of these subsections uses its own, specific, logical language. Richness of the languages and the suitability of logic to capture declarative and procedural knowledge justify the ubiquity of logic in this part of the system model. We believe these languages (not presented here due to space limitations but discussed in [Dávila: 1997]) are the best way to embed useful pieces of human knowledge into an artefact. For the sake of simplicity, we have furnished the example with a very trivial set of rules. The Client agent, for instance, is driven by the percept that the queue is too long, in which case the related (speech) act is triggered and executed immediately. For the time being, we have restricted the rules to the propositional case only. But we are aiming at the full expressive power of clausal logic.

```
NETWORK
   Input (I) ::
      setAgent(Client);
      IT:= 10;
      SENDTO(Teller[MIN]);
   Teller [1..3] (R) ::
      STAY:= 45;
   Exit (E) ::
AGENTS
   Client (AG) ::
      GOALS
         if long_queue then complain
   Manager (AG) Static ::
      GOALS
         if long_queue then create_teller;
         if two_empty_teller and then eliminate_teller;
INTERFACE
   long_queue(Node where, Agent who) {
      if (where.el.length() > 5) who.input.add("long_queue ", now);
   }
   complain(Node where, Agent who) {
      System.out.println(who.name + " : It's a long queue!!");
   }
   create_teller(Node where, Agent who) {
      if (Teller.multiplicity > Teller.maxMult)
         System.out.println("Ouch!, the place is full");
      else Teller.addInstance();
   }
   two_empty_teller(Node where, Agent who) {
      int j = 0;
      for (int i = 0, i < Teller.multiplicity, i++ ) {
         Teller currentTeller = Teller.instance(i);
         if ( currentTeller.il.ll() + currentTeller.el.ll() = 0 ) {
            j++;
            if (j > 1) who.input.add("two_empty_teller ", now); }}
   }
   eliminate_teller(Node where, Agent who) {
      int i = 0;
      while ((i < Teller.multiplicity  - 1) &
              (currentTeller.il.ll() + currentTeller.el.ll() > 0)) {
         i++;
         if (currentTeller.il.ll() + currentTeller.el.ll() = 0)
            NodeList.extract(currentTeller); }
   }
INIT
   TSIM:= 300;
   ACT(Input, 0);
END.
```

Fig. 5. Teller system revised

Finally, the INTERFACE sections links agent perceptions and actions to the environment. Here, with all the expresivity of the underlining object-oriented framework, the simulationist can specify the actual effects of the agents' actions, including those actions which involve many agents and, therefore, might be the subject of synergistics influences. Also in here, the simulationist states what the agents perceive in each circumstance in which they might be involved. In the teller example, for instance, the Manager agent is driven by the perceptions of a long queue or of too many empty queues. In each case, the agent has an specific response in the form of an action: either create or eliminate a teller. Notice that these actions, as detailed in the INTERFACE, once successfully executed, will change the structure of the original model (by adding or eliminating tellers).

5 Conclusions and further work.

In this paper we have presented the general structure of the platform GALATEA for multi-agent simulation. GALATEA is an extension of the DEVS oriented platform GLIDER and, like the later, supports general-purpose modelling and simulation. By introducing agents though, GALATEA also supports more expressive descriptions of systems in which decision making, acting and changing structures are the key elements. We plan to complete the development of GALATEA, implementing logical interpreters for each of the logical languages used to program the agents.

References

[Dávila: 1997] Dávila, J. A. (1997). *Agents in Logic Programming*. PhD thesis, Imperial College of Science Technology and Medicine., University of London. London. UK.

[Dávila and Tucci: 2000] Dávila, J. A. and Tucci, K. A. (2000). A multi-agent simulation theory.

[Domingo: 1988] Domingo, C. (1988). Glider, a network oriented simulation language for continuous and discrete event simulation. International Conference on Mathematical Models.

[Domingo: 1995] Domingo, C. (1995). Proyecto glider. e-122-92. informe 1992-1995. Technical report, CDCHT, Universidad de Los Andes. Mérida. Venezuela.

[Domingo et al: 1985] Domingo, C. and Hernández, M. (1985). Ideas básicas del lenguaje glider. Technical report, Instituto de Estadística Aplicada y Computación, Universidad de Los Andes. Mérida. Venezuela.

[Domingo et al.: 1993] Domingo, C., Tonella, G., Hoeger, H., Hernández, M., Sananes, M., and Silva, J. (1993). Use of object oriented programming ideas in a new simulation language. *Proceedings of Summer Computer Simulation Conference*, pages 137–142.

[Dahmann et al.: 1999] Dahmann, J. S., Calvin, J. O., and Weatherly, R. M. (1999). Flexible, interacting simulations provide a foundation for successful dmt applications. a reusable architecture for simulations. *Communications of the ACM*, 42(9):79–84.

[HLA: 1995] Modeling, D. and Simulation (1995). High level architecture (hla). *http://www.dmso.mil*.

[Palm: 1999] Palm, F. J. (1999). Simulación combinada discreta/continua orientada a objetos: Diseño para un lenguaje glider orientado a objetos. Master's thesis, Maestría en Matemática Aplicada a la Ingeniería, Universidad de Los Andes. Mérida. Venezuela.

[Russell and Norvig: 1995] Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall, Inc.

[Tonella et al.: 1993] Tonella, G., Domingo, C., Sananes, M., and Tucci, K. (1993). El lenguaje glider y la computación orientada hacia objeto. *Proceedings of XLIII Convención Anual ASOVAC*.

[Tucci: 1993] Tucci, K. A. (1993). Prototipo del compilador glider en c++. Tesis de Grado. Escuela de Ingeniería de Sistemas. Facultad de Ingeniería. Universidad de Los Andes. Mérida. Venezuela.

[Zeigler: 1976] Zeigler, B. P. (1976). *Theory of Modelling and Simulation.* John Wiley & Sons, Inc.

[Zeigler: 1990] Zeigler, B. P. (1990). *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press, Inc.