MODELLING AGENTS WITH UML: AN EXAMPLE IN BUILDING SECURITY EVALUTION

JUAN DE LARA

Dept. Ingeniería Informática, Universidad Autónoma de Madrid
Ctra. De Colmenar, km. 15, 28049 Madrid, Spain
e-mail: Juan.Lara@ii.uam.es

Abstract: This paper proposes some extensions to UML for agent-based modelling and simulation, where the agents follow either a purely reactive or a hybrid layered approach (reactivity combined with proactivity). The extensions include notations for sensors, effectors and agent's capabilities and their specification in a formal way. In this work, agent-based simulation is also proposed as a method to help in the evaluation of security in buildings by simulating their evacuation. These simulations allow us to measure evacuation time of different scenarios, to play with different structural properties of the building and test their influence in the building security, which can be useful during its design. In these simulations, people are represented by means of agents using a hybrid layered approach. The lower layer deals with collision avoidance and doors' visualization, while the higher layer builds models of the environment (rooms' connectivity) to help the agent find the exit. Buildings are conceptually modelled as graphs where vertices and edges represent rooms and doors respectively. Rooms are discretized and represented as two-dimensional grids, in which agents can move.

keywords: Agent-Based Simulation, Hybrid Agents, UML, Building Safety, Crowd Simulation.

1. INTRODUCTION

Computer simulation is a valuable tool in situations in which experimentation with the real system is dangerous, expensive, or non-ethical. It is useful for decision-making as it enables the experimentation in multiple different scenarios in an inexpensive way. The increasing speed of today's computers is making possible the simulation of systems described by a large amount of interacting entities. For the modelling of such systems, one usually uses Cellular Automata or Agent-Based techniques [Jennings et al. 1998]. The former is more appropriate when individuals are simple and alike. The latter is more natural when individuals have more complex capabilities, such as sensors, effectors, internal complex states or reasoning abilities. Multi-Agent Systems (MAS) have been used in very different areas such as manufacturing, process control, information management and electronic commerce. In this paper, the concept of MAS is used for the modelling and simulation of a large number of individuals trying to escape from a building. Agents are modelled using a reactive layered architecture. In a purely reactive approach there is no symbolic reasoning, the agent behaviour is expressed as finite state machines [Brooks 1995]. In the approach of the present work, the behaviour is decomposed in layers dealing with reactive and pro-active behaviour.

Although there are many languages for agent-based simulation programming (such as Swarm [Swarm 2003], or OOCSMP [Alfonseca and de Lara 2002]), there is a need for higher-level, graphical, intuitive notations to help in the modelling phase of such systems. Some emerging approaches use UML [Booch et al. 2002] extensions especially devised for the modelling of MAS [aUML, 2003]. These are notations suitable for the design of applications composed of agents, and mainly to specify interaction protocols for the application agents [Bauer at al. 2000]. In opposition, the extensions proposed in this paper are mainly useful for the modelling of *Agent-Based Simulations*. In particular, we propose extensions to model the agent sensors and effectors (to express the agent's interaction with the environment) and the agent capabilities. The extensions proposed for sensors and effectors follow the line of [aUML, 2003], for example, in [Bauer, 2001] interfaces are also used for expressing communication (in the case of an application "*interacting protocols*") between agents. In this paper we also propose the inclusion of the concept of *agent* and *classes of agents* in some of the standard UML diagrams, giving rise to *agent diagrams* (similar to *object diagrams*), *agent class diagrams* (similar to *class diagrams*) and *agent collaboration diagrams* (similar to *collaboration diagrams*). This last kind of diagram is proposed as a means to formalize some of the agent's capabilities, in a similar way as [Engels et al. 2000], but for a different purpose.

The extensions for agent-based modelling proposed in this work are illustrated by means of an example: the simulation of building evacuations. This paper proposes such agent-based simulations as an inexpensive means to help in the evaluation of building security. While

 ISBN 3-936150-25-7

designing the building, simulations can help in architectural decisions which may affect building security, such as the placement of regular and emergency doors, and room's connectivity. In this case, simulation is the only choice, as direct experimentation (real evacuations) cannot be performed. For already existing buildings, simulations can complement evacuation simulations with real people, as they are less expensive and less annoying for the building inhabitants. Simulations make possible to experiment with different scenarios of people density in each room, with different environmental factors (fire, smoke, different degrees of visibility, etc.) or as a means to evaluate the impact of changing some security features in the building; for example, adding new emergency doors, indicators, etc.

2. EXTENDING UML FOR AGENT MODELLING

UML is becoming ever more used in the software community. For that reason, in this work the standard UML is used as much as possible, although extensions have been added to some of the diagrams and are explained in the next subsections.

2.1 Agent Class Diagrams

A class diagram is a graphical view of the static structural model. Here we propose to include *Agent Classes* in this kind of diagrams. These describe the kind of agents that exist in the system (in this paper we only consider reactive or hybrid agents). In standard UML, there is a notation for active objects (with their own thread of execution). Autonomous agents must have their own thread of execution, but they are not mere objects: they have a (partial) knowledge of their environment (by means of sensors) and may act upon it (by means of effectors). They have abilities and can be requested to perform a certain action. This is different from invoking a method on them, as the agent may refuse to perform the action. Figure 2 (Agent class *Runner*) shows the symbol we use for *Agent Classes*. There is a separate box for capabilities and another for actions. Capabilities are arranged in layers, separated by a dotted line. If capabilities or actions are not specified, the corresponding box can be omitted. We also provide symbols for the agent's sensors and effectors (similar to the ones used for interfaces, but filled in black). It is possible to connect other (Agent) classes to these black dots to mean that the agent can sense or act upon that other class. An example of the use of these symbols is given in Figures 2 and 3. Most of the times, *Agent Classes* have one or more associated Statechart diagrams specifying the agent behaviour (see Figure 1).

2.2 Agent Diagrams

A static object diagram is an instance of a class diagram, where objects and their relationships may appear. It shows a snapshot of the state of the system at a point in time. Here we include *Agents* in this kind of diagrams. These are instances of *Agent Classes*, and are represented in a similar way (see Figure 3).

2.3 Agent Collaboration Diagrams

These diagrams show graphs of objects linked to each other, toghether with their communication patterns. Here we use these diagrams to specify some of the agent capabilities in a formal way. In order to specify a capability, one has to provide a number of collaboration diagrams; each one of them can be applied under different circumstances (in a similar way as graph-grammars [Engels at al. 2000] and rule-based programming). Each collaboration diagram is assigned a priority that specifies the order in which the collaboration diagram will be tried. A collaboration diagram is applicable if it is consistent with the state of the system at that moment. That is, if an homomorphism between the system's state and the collaboration diagram can be found. When the collaboration diagram is applied, the nodes and links tagged as *new* and *destroyed* are created and destroyed. Collaboration diagrams are also extended with the capacity to return values (as here they are used as a means to specify functions). An example of a capability specified in this way can be found in Figure 6.

This approach has similarities with graph grammars. These are composed by rules, each having graphs in their left and right hand sides (LHS and RHS). If a rule makes a match with a certain part of an input graph (called host graph) the rule can be applied and the zone of the graph that was matched is replaced by the RHS of the rule. In the extensions to collaboration diagrams that we propose, both LHS and RHS are collapsed into a single graph, and the nodes and links to be created and destroyed are tagged with *new* and *destroyed*. We also use *negative application conditions* in collaboration diagrams, which is a standard notation in graph grammars in the form of crossed-out elements (see Figure 6). It expresses the fact that, in order for the rule to be applied, the crossed-out elements must not be present in the matched graph. A similar idea but applied in another context, and without the possibility to return a value was proposed in [Engels et al. 2000].

An example of the use of these extensions is presented in the following section.

3. EVALUATING BULDING SECURITY WITH AGENT-BASED SIMULATION

3.1. Single room model

This section deals with the simpler case, in which we consider evacuations of single rooms. In our model, rooms are discretized and represented as two-dimensional, rectangular grids. Each cell of the grid can

be empty, contain up to three agents, a wall or a door. Doors can be made wider by concatenating several of them. Time is discretized, in such a way that agents can move to one of the eight neighbour cells at each time step. Agents do not communicate. Once an agent sees a door, its objective is to move towards it in a straight line. If at some moment the shortest path cannot be followed – because there are many agents blocking the way – the agent moves to the least populated neighbour cell. If the agent has not seen a door before, then it moves to the most populated neighbour cell containing at most two agents (that is, he will try to "follow the crowd"). The simulation finishes when all the agents have reached a door. Figure 1 is a Statechart representing this behaviour. Some transitions in the model invoke methods (*lookAround()* and *move(where)*), which should be considered as the agent capabilities. These capabilities make use of the agent's sensors and effectors (simulated, as we are implementing agents in software). In the case of *lookAround*, the agent is interested in visualizing either a door (which sometimes may not be visible), or the most populated place. By means of *tm(1)* we specify that at each time step, the agent should perform a certain action, depending on its current state.

The agent's structure is shown in Figure 2. It shows an agent class (*Runner)*, which has a sensor (*iVisual*) and an effector (*iLocomotion*). Special relationships (*is visible* and *can move in*) come from the classes that the sensors or effectors can act or sense. In this case, the *iVisual* sensor can sense either Doors, *Walls* or other *Runners*. The *iLocomotion* effector can act on *Rooms* (that is, agents can walk in the room). Agent capabilities *move* and *lookAround* are specified in the *Agent class* lower box. Attributes *doorX* and *doorY* are used to store the position of the door the agent is moving towards (in the case he has seen a door before). A *Runner* is situated in a room, and this is expressed with the relationship class *Position*.

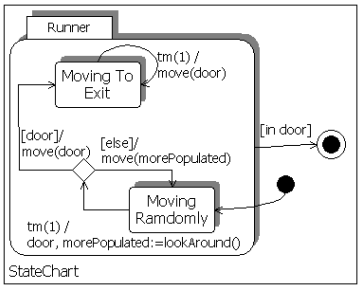


Figure 1: Behaviour of the Runner Agent.

A UML *Package* delimits the environment, which consists of a room made of several doors and walls. Doors are placed in walls (relationship "*has*"). The spatial dimensions of the room are stored in attributes *width* and *length*. The door coordinates are stored in attributes *px* and *py*. The initial and final wall coordinates are stored in attributes *xinit*, *yinit* and *xend*, *yend* respectively. Only walls parallel to the X or Y axis are allowed in the model. The interaction between the environment and the agents is expressed by using the sensor/effector notation introduced in section 2.1.

Figure 3 shows an "*agent diagram*" that reflects the way in which an agent can sense the presence of doors or other agents. The figure shows a situation in which an agent (*r1*) is able to see another agent (*r2*) and a door. The condition for this to happen is that no other visible object must be between *r1* and *r2* or the door.

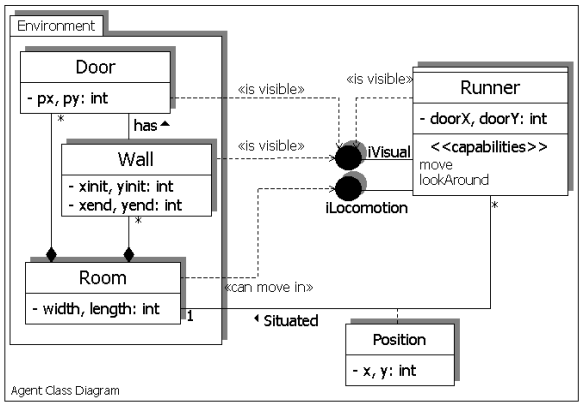


Figure 2: An Agent Class Diagram.

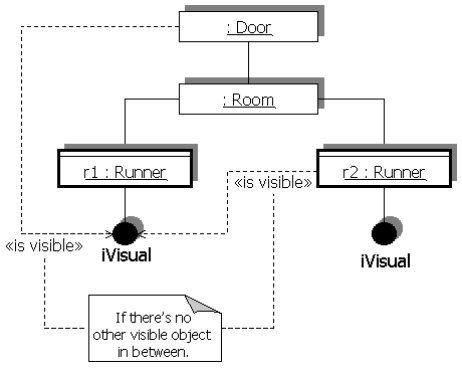
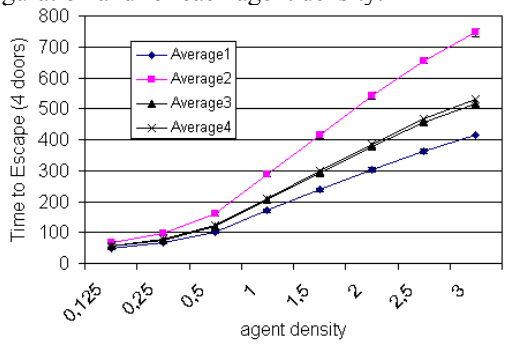


Figure 3: An Agent Diagram

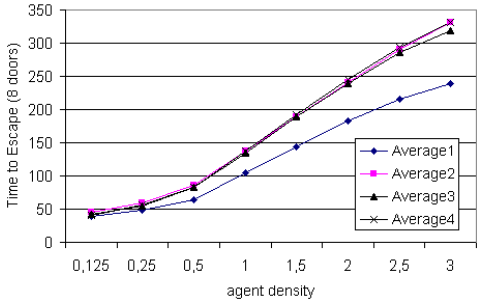
The model has been programmed in C++, as efficiency in time and memory is needed, because thousands of agents will be created in the simulation. Agent programming languages are less efficient than programming directly in C++, because it allows for optimisation of the code by hand. On the contrary, agent languages provide higher-level constructs that make the programming easier. The implementation of the movement in straight line was done using the Bresenham's line drawing algorithm. To illustrate the usefulness of this model the following subsection shows some of the experiments performed.

3.2. Experiments

Two different sets of experiments were performed to evaluate the effect of door placement in the time it takes the agents to escape from the room (of size 42x42). In the first set, four doors were placed in the room, in different configurations, each one tested with different density of agents, from 0.125 to 3 (the maximum, as in each cell at most three agents can be present at the same time). Forty experiments were performed with each room configuration and for each agent density.



(a)



(b)

Figure 4: Time to escape with respect to agent density for different door configurations: 4(a) and 8 doors(b).

Figure 4(a) shows the results of each configuration tested for different agent densities. The X-axis is the agent density; the Y-axis is the time it took the agents to escape (average of the 40 experiments). The first configuration has a door in the middle of each wall. Setting the origin of coordinates at the upper left corner of the room, the second configuration has doors at (5, 0), (35, 41), (41, 35) and (0, 5). The third room has doors at (20, 0), (20, 41), (41, 35) and (0, 6). The fourth room has doors at (20, 0), (20, 41), (41, 19), (0, 19). The first configuration gives the better time, as agglomerations tend to form near the doors, making the escape process more difficult. If two doors are "*too near*" these agglomerations are even bigger. An example of this is configuration 2, which gives the worst results, as it has very close pairs of doors in the room corners. The advantage of configuration 1 is bigger as the agent density goes up, because the effect of the agglomerations as the density of agents increases is bigger.

Figure 4(b) shows the results of the second set of experiments, with eight doors. One of the objectives was to test the efficiency of 8 doors against 4 bigger doors, which can be produced by joining two smaller doors. The first configuration has two doors in each wall, each one placed in an equidistant position to the other door and to the corner of the room. For the second configuration, a big door (composed of two smaller doors) has been placed in the middle of each wall. The third configuration is a room with one big door in the North and in the South, and two smaller ones in the East and West. These are placed at 5 units from the end of the walls. Finally, configuration has two doors in each wall, at 5 units from the end of each wall. The best results have been obtained with the first configuration, for the same reason: if two doors are too near, agglomerations are formed. To reduce this effect, the simulations show that (specially if the room is very crowded) it is better to have numerous small separated doors than a few big doors.

3.3 Extending the model for multiple rooms

In this section, we consider buildings with multiple rooms. The agent structure must be extended with a "*mental*" representation of the rooms' connectivity to guide the agent in his navigation towards the exit. We can experiment with two situations: in the first one the agent does not have any *a priori* knowledge of the building connectivity, he builds his *mental map* while navigating through the building looking for the exit. In the second situation, we assume that the agents have partial or total information about the building. In both cases, the mental map is used by the agent to navigate trough the building.

Left of Figure 5 reflects this situation. Class *Building* has been introduced, composed by a *number* of rooms. Class *Door* has been extended with the attribute *type* indicating if the door is an exit or leads to another room. While *inner* doors are connected to other inner doors leading to other rooms; exit doors are not connected to other doors, as they lead to the outside. The mental map of the environment the agent builds and uses for navigation is shown in a separate package. A relationship of type "*represents*" expresses the fact that the agent is able to recognize a real room if he has been in the room before. The same happens with doors inside rooms. The agent also remembers if he has explored the door before or not. As the mental map is a model of the environment (an abstraction), the agent does not memorize room or wall dimensions, as they are not needed for navigation. The agent capabilities have been extended with the possibility to *memorize* new rooms or doors as they are discovered. Capabilities have been arranged in two layers. The upper layer capability (*getDoor*) is higher-level than the lower layer ones and is used by the agent to decide the most appropriate door to go to, and accesses the mental map.

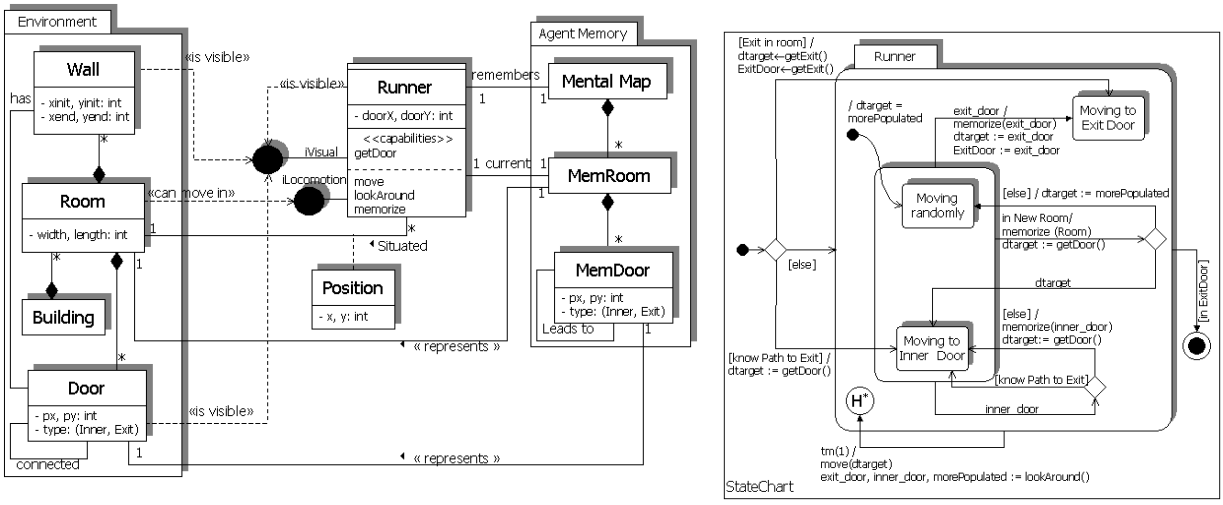


Figure 5: Agent Class diagram with the model for multiple rooms (left). Behaviour of the agent (right).

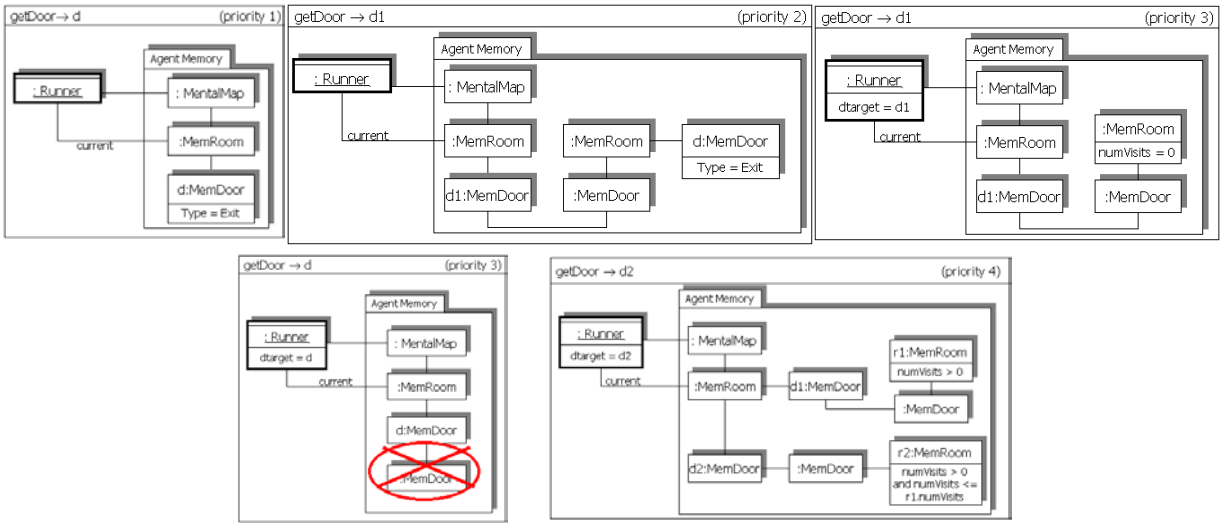


Figure 6: Collaboration Diagrams for Specifying the getDoor capability

If the agent has *a priori* knowledge of the building map, then this capability guides the agent through the shortest sequence of rooms towards the exit. If the agent does not have *a priori* knowledge, then his mental map may not be complete, and several situations can arise. In the easiest case, if he knows an exit door in the current room, this is the most appropriate door. If an exit door is not present in the current room, then the agent searches in his mental map to check if some of the neighbour rooms have an exit door. If this is the case, then the most appropriate door is the one leading to that room. In other case, the agent chooses the door that leads to a non-visited room, and if all rooms have been visited before, he chooses the least visited room. This complex behaviour can be formalized using a number of *agent collaboration diagrams* specifying the expected behaviour of the *getDoor* capability. Each collaboration diagram specifies a situation that, if present at run time, will cause the capability specification to be executed. For example, the first diagram in figure 6 shows the situation in which an agent is in a room with an exit door. In this case, the capability returns this door as the most appropriate. This diagram does not state that the agent must only know one room, but that this is the minimum set of elements that must be present in order for this situation to be valid.

The second diagram specifies the situation in which the exit door is in a neighbour room. The third diagram applies when the agent does not know any exit door in the current or neighbour rooms (or there are not any). In this case the agent chooses a room not visited before. In the fourth diagram the situation is the same, but the agent does not have a complete knowledge of the environment: the map is not complete. If he finds a door which does not have any connection, the door is not explored. The *negative application condition* means that the agent must

be in a room with a door that has not been explored. Finally, the last diagram shows a situation in which the agent chooses the least visited neighbour room.

The Statechart showing the agent's behaviour has to be modified to consider the navigation between rooms and is shown to the right of Figure 5. If the agent does not have *a priori* knowledge of the environment, then from the initial state he moves to the "*Moving Randomly*" state. If the agent has *a priori* knowledge, two situations may arise. In the first one, the agent knows that the exit door is in the current room, so the agent moves to state "*Moving to Exit Door*". In the second one, the agent knows that the exit door is elsewhere, so he selects the most appropriate inner door to move to and moves to state "*Moving to Inner Door*".

4. CONCLUSIONS AND FUTURE WORK

This paper has proposed some extensions to the UML for the modelling of reactive or hybrid agent simulations. The extensions introduce elements similar to interfaces to express agent's sensors and effectors. Special relationships are introduced to express the fact that other agents or objects can be sensed or acted upon by these sensors and effectors. Instances of these relationships and symbols can be found in *agent diagrams* (a kind of diagram similar to object diagrams). Agent capabilities are declared in an extra box in the *agent class* box. Capabilities can be formally specified using a number of *agent collaboration diagrams*, in a similar way as graph grammars rules. Packages are used to separate the environment and the agent memory. The extensions continue the line of the ones proposed by the *aUML* community and have been used to model building evacuations. This kind of simulations is an inexpensive means to test building security, and can be a complement to real evacuation simulations.

We are extending the model with the possibility to evaluate exit signals placement, experimenting with situations of low visibility and communication between agents. We want to test the model with real buildings, validating the simulation results with data from real building evacuations. We are also implementing the proposed UML extensions in the meta-modelling tool AToM^3 [de Lara and Vangheluwe 2002], in such a way that code for some agent programming languages will be generated from the models. We are also constructing a meta-model to allow the users model different kinds of buildings. These models have to be translated into object diagrams for further processing.

Acknowledgement: This paper has been sponsored by the Spanish Ministry of Science and Technology (TIC2002-01948).

REFERENCES

Agent UML (aUML) home page: http://www.aUML.org

Alfonseca, M., de Lara, J. 2002. "*Two level evolution of foraging agent communities*". BioSystems Vol 66, Issues 1-2, pp.: 21-30.

Bauer, B., Müller, J., Odell, J. 2000. "*Agent UML: A Formalism for Specifying Multiagent Interaction*". In Proc. of Agent-Oriented Software Engineering 2000, Springer. pp.: 91-103.

Bauer, B. 2001. "*UML Class Diagrams Revisited in the Context of Agent-Based Systems*". In Proc. of Agent-Oriented Software Engineering (AOSE) 2001, Agents 2001, Montreal. pp.: 1-8.

Booch, G., Rumbaugh, J., Jacobson, I. 1999. "*The Unified Modeling Language User Guide*". Addison-Wesley.

Brooks, R. 1995. "*Intelligence without reason*". In The Artificial Life Route to Artificial Intelligence. Building Embodied, Situated Agents. Lawrence Erlabaum Associates.

de Lara, J., Vangheluwe, H. 2002. "*AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling*". LNCS 2306, p.: 174-188. Springer. AToM^3 home page: http://atom3.cs.mcgill.ca

Engels, G., Hausmann, J. H., Heckel, R., Sauer, S. 2000. "*Dynamic Meta Modeling: A graphical Approach to the Operational Semantics of Behavioral Diagrams in UML*". LNCS 1930, pp.: 323-337. Springer.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. 1999. "*Handbook of Graph Grammars and Computing by Graph Transformation*". Vols.1 and 2. World Scientific.

Jennings, N.R., Sycara, K., Wooldridge, M. 1998. "*A Roadmap of Agent Research and Development*". Autonomous Agents and Multi-Agent Systems, 1, 7-38 (1998). Kluwer.

Swarm development group home page: http://www.swarm.org

BIOGRAPHY

Juan de Lara is an an assistant professor at the Computer Science Department of the Universidad Autónoma in Madrid, where he teaches Software Engineering. He is a PhD in Computer Science, and works in Web based Simulation, Agent based Simulation and Multi-Paradigm Modelling.