GEAMAS V2.0: an Object Oriented Platform for Complex Systems Simulations.

JC. Soulié, P. Marcenac, S. Calderoni and R. Courdier
MAS2 Team - IREMIA, University of La Réunion, BP 7151
97715 St Denis Cedex 9 - France
Email: {soulie, marcenac, calde, courdier}@univ-reunion.fr
Url: http://www.univ-reunion.fr/~mas2

Abstract
This paper presents the object oriented design and implementation of GEAMAS V2.0, a toolkit for virtual simulations of complex systems. GEAMAS V2.0 is structured in three modules: the Kernel, the Generation Environment and the Simulation Environment. The Kernel implements an object model for agents and provides generic classes. The Generation Environment allows the graphical design of applications. The Simulation Environment enables the observation of the simulation’s evolution via Graphical User Interface tools. The implementation uses Java 1.1. We applied GEAMAS V2.0 to the sand-pile automaton problem. This reference application, which is easily modeled with GEAMAS V2.0, provides a first validation of our system architecture and simulation mechanism.

1. Introduction
The great complexity of natural systems constitutes a challenge for computer science modeling. This kind of tools is meant for researchers who need to validate complex models, without having to implement the whole application by their own. This work tries to answer some lacks in the framework of complex systems modeling, by providing a development environment reducing complexity of the system tackled. Resulting systems then provide behavior emerging from local interactions between simple components. Because of this complexity, our team has developed a generic architecture based on agent modeling called GEAMAS, acronym of GEneric Architecture for Multi-Agent Simulation.

Our team obtain promising results [4] [7] [10] with dedicated application developed with agent technologies on Smalltalk-80. This was the first prototypal version of GEAMAS called GEAMAS V1.0. Two simulation applications of physical processes have developed: the first one investigated volcano eruptions predicting; the second one reports a very interesting experience dealing with the emergence of natural structures by simulation. But with the growth of Internet, we want to develop software systems whose can be accessible by everyone on the net. That's why we decided not only to port our model in Java but to create a set of tools whose can help scientists or researchers working on the agent approach. This new version of GEAMAS is called GEAMAS V2.0.

GEAMAS V2.0 is structured in three modules: the Kernel which provides generic agents and society, the Generation Environment which allows the graphical design of new applications, and the Simulation Environment which allows the observation of the simulation’s evolution. These two environment relied on the GEAMAS V2.0’s core: the Kernel. The design of GEAMAS V2.0 has been tackled with an OMT approach [14] and has led to implement three abstraction level: a micro-level, describing atomic and d determinist behaviors in simple (reactive) agents; a medium-level, arranging intermediate structures; and a macro-level called society, describing the whole system where emerging behavior will be observed and analyzed.

Our proposal relies on the fact that the asked question is not "what happened?" or even "what might have happened?", but rather "what are sufficient conditions for a given result to be obtained?". The aim of the platform is then to provide a complete toolkit as a virtual laboratory [12] for designing a large scope of dynamic systems, and providing generic interfaces to set and control the simulation. This kind of tool is meant for researchers needing to simulate complex systems, without to implement the whole application by their own.

This paper is organized as followed. Section 2 describes briefly the engineering and the abstraction level of GEAMAS V2.0. Section 3 details GEAMAS’s Kernel, including OMT models, and Java code samples. Section 4 details the Generation Environment and the Simulation Environment of GEAMAS V2.0.

2. Engineering and Abstraction Level
The objectives are here to separate language and agents features from application definition, and finally application from simulation represented by instances

Figure 1. gives an overview of the GEAMAS V2.0 which is briefly exposed in this section, before being detailed in the next one.

Figure 1. An overview of GEAMAS V2.0

Horizontally, GEAMAS V2.0 is based on three software engineering layers:
The Language Layer, describing Java as an object-oriented language [8] [15], which has been chosen to implement GEAMAS V2.0. There are three leading reasons of this choice:

- The clarity and cleanliness of the language according to the object-oriented philosophy (all-object, static and dynamic inheritance, dynamic overloading, polymorphism, ...) as well as useful utilities for agent's implementation, such as threads;
- Java program are compiled to an architecture neutral byte-code. A Java application can run on any system which implements the Java Virtual Machine (JVM);
- The necessity to deliver our applications through Applets, which is small, subordinate or embeddable application to be run within the context of a larger application such as a web browser for instance [13]. Moreover Applets are secured: they can not access to the local file system. Such kind of access are managed by the SecurityManager class.

The Agent Layer, adding agents capacities to Java, that is independence, autonomy and evolution features. Independence allows the agent to process tasks without explicitly receiving the order. Action are performed through passing asynchronous messages. At the same time, an agent is given an autonomous part which can control its behavior. The autonomous part of an agent determines its self-governing and decentralizes the control of the system behavior: an agent is able to take an initiative and exercise a flexible degree of control over its own actions, by dynamically choosing which action to invoke. An agent also evolves during its life, due to relationships kept with other agents.

Finally, the GUI Layer, hiding the complexity of the environment to the user, by introducing interfaces needed to setup applications and derive simulations. The GUI Layer contributes to integrate the platform as a toolkit and a virtual laboratory for complex systems simulation purposes.

The platform is also vertically based on four knowledge abstraction layers:

- The Object Layer is the language layer. It is composed of each Java class being used or derived to implement the sub-abstraction layer. Figure 1. illustrates some of these classes, for instance: Thread allowing parallel tasking, Observer indicating that a class is going to be observed, or Vector which allows to dynamically manage list of instance. The GEAMAS Layer which constitutes the heart of GEAMAS V2.0. It includes a specific architecture (composed of a macro-agent, medium-agents and micro-agents) as well as agent and society models.

- The Application Layer can be seen as the result of the GEAMAS Layer applied on a specific domain. An application is therefore understood as a real world translation in terms of agents following the model implemented in the GEAMAS Layer. Building an application then consists in describing the real world by deriving some or all classes of the previous layer by using The Generation Environment. The Application Layer is a set of classes, whose definitions are built by the generator.

- The Simulation Layer is the instance level of the application. It defines a set of instances of classes. Instanciation of such classes is graphically set with the help of the simulator editor, that proposes some fill-in-the-blanks windows to set each parameter value.

The next section describes the heart of the GEAMAS Layer named Kernel in the GEAMAS V2.0 architecture. We detail internal mechanisms, the OMT model, and the Java implementation.

3. Architecture and Implementation of the GEAMAS V2.0's Kernel

3.1 Architecture and Mechanisms

The architecture of the Kernel is based on three abstraction levels [11]. Figure 2. gives an overview of the three levels.

Each successive level represents a higher level of abstraction. Each abstraction level describes a degree of knowledge complexity and applies a model of agents through the expression of knowledge, behavior and evolution capabilities. Each level is independent executing processes asynchronously and representing a higher level of abstraction than the one below it. Furthermore, the architecture implements the recursion property to greatly reduce the design of the system, by applying the same agent-model to coarse-grain and fine-grain agents. To tackle the relation between individual and collective, two fundamental mechanisms are introduced: Decomposition and Recomposition. Decomposition allows to transfer informations to lower levels, until Recomposition allows to transfer information to higher levels [10].

Figure 2. Architecture with three abstraction levels

- The Macro-Level: describes a coarse-grain agent, as a related element in which the global behavior will be observed. It holds on the underlying agent's organization in a society. In multi-agent systems, such an organization has no reason to be if it is unable to represent interactions facilities between agents. The society is therefore organized as a network of acquaintances, allowing agents to interacts together and is named macro-agent.

- The Micro-Level: describes fine-grain-agents, that is reactive agents [5] [6], called micro-agents. Each micro-agent generates behavior without explicit representation of
the domain neither global tasks constraints. At run time, micro-agents interact in a concurrent way with their surroundings. The Micro-Level then masks the complexity of the whole system by encapsulating basic actions. The Medium-Level: however, as we seen it before, a complex system integrates a chaotic dimension: the critical state of the system leads to "catastrophes". Our assumptions are then to consider the critical state as assigned to particular states of micro-agents, and the catastrophe to a spatio-temporal aggregation, in some particular context and environment, and under specific conditions, of micro-agents taking part to the phenomenon. Those agents are named medium-agents and represents cognitive-agents. Our proposal is to consider the result of this spatio-temporal aggregation as an emergent phenomenon, created by self-organization in the multi-agent system. Self-organization is a dimension of complexity and an important feature of evolutionary systems. It is assigned to the appearance in a specific context and in an active environment, of new structures not previously identified, and from now on irreversible within a system of interacting entities. Therefore, the main issue is to represent and characterize self-organization result, born from local interactions between agents. Considering a two-levels architecture in the system makes the catastrophes modeling tasks difficult in the system, and this for two reasons. On the one hand, as macro-level is described by fine-grain agents, the organization of new structures can not be processed in such a microscopic level, having not enough knowledge to do so. Beyond this microscopic dimension, emergent structures can only arise in one macroscopic level, disposing of missing knowledge. On the other hand, it is often impossible for a macro-level to characterize and correctly identify the whole set of emergent structures. Indeed, available knowledge in macro-agent is too imprecise, and as a matter of fact, a significant number of micro-agents could intervene in the whole emergence process. It is then skillful to introduce an intermediate grain level between the macro-level and the micro-level, embedded as sub-organizations. This medium level is designed before the simulation begins by giving sub-organizations' structure and behavior. This point of view enforces the necessity of finding the most efficient abstraction when designing this level, which is totally dependent of context tackled [2]. However, such medium-agents are not dynamically pre-defined in the system because they spontaneously appear during the simulation as the result of self-organization. When self-organization arises (meaning that the system's critical state is reached), a new medium-agent will be created within the system to model the "catastrophe". This approach seems handy, because it allows to consider the "catastrophe" as an entire entity able to compute its own properties. Moreover, it is at the right place to distribute new computed values after the critical point in order to constraint the affected part of the system. Medium-agent's properties are computed and set when the organization has been built. Such properties are then "re-introduced" in underlying lower-level agents as constraints to apply in their own structure, this mechanism is called "back-propagation". That's why, a medium-agent is playing a causal role for the rest of the simulation, by constraining a part of the system. 3.2 Working 3.2.1 Global working: two fundamental mechanisms are identified: firstly, a set of decomposition mechanisms which allows a global input event to be distributed over the network in order to be performed by lower level agents; secondly, the part of the self-organization mechanisms needed to create, manage, and preserve consistency of medium-agent during phenomena emergence. Decomposition general mechanisms: three tasks are carried out when distributing events over the networks. Firstly, a process called Decomposition, able to propagate a message to underlying agents. Decomposition allows informations to be transferred in lower levels of the architecture. When receiving a Decomposition message, the agent gets information given by its society. A Decomposition message could specify that an external event is to be performed by lower level agents. This type of message helps micro-agents to be ready to perform the event. The event is thus divided up, and "sub-events" are transferred to micro-agents which are the best able to process them. Conversely, there may be laws or knowledge on a macro-level that must be observed by micro-agents, for instance, knowledge that constrains micro-agents moves. A society is at the right place to express such knowledge; it may constraint part of the state and behavior of their subordinate micro-agents, either by acting on the world structure or on individual micro-agents. Secondly, a process able to find the handling underlying agents which can perform an event, according to those able to receive external solicitations in the organization. It defines a process responsible for localizing the necessary agents to perform an event type that remains consistent with the current goal of the society. Thirdly, a filtering function receives an external in input and provides "discretized" sub-events in output. Such a function is application-relative, as the way to transform an event in such tiny simulated events can not be generic anyway. Global self-organization mechanisms: when self-organization is needed to represent emergent phenomena, the global mechanism is used to create a medium-agent aggregating some micro-agents, or adding them to an existing one. Beyond this mechanism, back propagation aims a feedback reaction at medium-agent's members, by computing appropriate values to the new agent's state. Setting new properties in a medium-agent becomes easy at this point, mainly because the global creation mechanism can pick up information from lower-level. Therefore, when the emergent structure back-propagates a constraint, it forces the behavior of underlying agents by applying them some of its properties. This view is very close to reality: during an eruption for instance, each magma lens releases energy and moves into the system. The remaining energy is then low, and such micro-agents should be properly re-initialized so that the whole system behavior might be modified accordingly. 3.2.2 Local Working: recomposition is the "bottom-up" mechanism which is at the root of emergent behaviors of the system. It transfers informations on the agent's stability from micro-level to macro-level. State parameters values evolve during the simulation, based on multiple local interactions between micro-agents. When a threshold is reached, the agent is assumed to be unstable, and information is then transferred to the agent's society with a Recomposition message. Therefore, a Recomposition message is provided by micro-agents to alert the society that something unusual is happening. Shifting from the micro-level up to the upper-level, the society collects data on microbehaviors and combines them to determine macrobehaviors and then adapts itself to the situation. Recomposition then requires the society some abilities to interpret informations at low-levels. A higher level behavior then emerges from such ability. In such sense, the dynamic of low-level agents governs emerging behaviors. Local self-organization mechanism: a local detection mechanism, distributed over micro-agents, is locally implemented to detect and identify an effective cooperation between interacting agents leading to an emergent phenomenon. At the time an interaction occurs, an agent A looks for agent B's intentions. Then a local observer mechanism compares agent
A’s intention with those of agent B. As the similarity is detected, a cooperation is then established, and both agents get organized for a common purpose. From this self-organization, locally emerges an organizational structure that has never been described in the system before [2]. When the phenomenon stops, a mechanism in charge of ending the process examines the neighborhood to look at potential agents not enough stable (and able to propagate again the phenomenon). This mechanism is called an observer. In traditional multi-agent systems, the observer is usually implemented as a global loop inserted over the system or as a part of the society, to inspect the stability of underlying agents forming the structure. In our architecture the programmed observer is distributed and locally defined [10].

3.3 Implementation

The heart of the Kernel is composed of the Society Model Unit describing the global view of the multi-agents system, the Agent Model Unit describing local behaviors and interactions. Another important part of the Agent Model Unit is an unit called Agent System Unit. This unit allows the Simulation Environment to observe the multi-agents system’s evolution.

3.3.1 Society Model Unit: the macro-level manages agents in the system in order to match global specifications of the application and is inscribed in a model called the society model. The society model describes the role, the interface and the whole organization of the systems: the system’s role is expressed through behaviors (global events and global constraints); the interface with the external world (software, human, ...) describes input and output parameters for simulation needs; and the organization is managed through a structure of inter-related agents.

The society model has a global vision of all the micro-agents and medium-agents. And those agents are evolving and interacting together during a simulation. The organization is defined as the description of agent’s dynamic relationships used to structure interactions within the system. The structure is then organized as a network of acquaintances, forming a competence network, where agents correspond to nodes and interaction possibilities (acquaintances relationships) to edges. The choice of a network as data structure to organized agents is justified by its flexibility and its general nature to be adapted to several situations [10]. In addition it allows the description of some real world topologic representation, where the geometry can be implicitly represented. Indeed, the acquaintances number of each agent determines the topology, and constitutes a very interesting feature, showing how a real world could be designed as a three dimensional network. The parameter is called the agent’s connectivity. As connections determine the ability of an agent to communicate, a part of the communication protocol is then defined by setting this parameter.

In addition, some more information has been added in the graph: firstly, edges could be strengthened to take the signal intensity between two agents into account. This feature allows a certain priority of messages among agents to be expressed by the application designer. Indeed, the intensity of a signal emitted by an agent decreases as a function of the distance between agents, and agents’ behavior is strongly dictated by their relative position in the topological structure. Secondly, some agents could be defined to receive external solicitations. This kind of agent will be first considered when an external event has to be performed in the system. In descriptive real worlds such as Physics for instance, such agents generally model the system’s side. This kind of agent are called input-agents and the graph can enumerate the set of this input-agents. Thirdly, with the same idea, some agents are shown as able to provide results, modeling for instance the opposite side of the system. This information allows the system with computing simulation results or adapting behavior. This kind of agent are called output-agent, and the graph can retrieve them too.

The Figure 3 below illustrates the OMT class diagram of the SocietyModel Unit and the AgentModel Unit.

3.3.2 Agent Model Unit: the agent model is the most important part of the Kernel, as it defines agent as the atomic entity handled in the GEAMAS V2.0’s Kernel. Agent’s knowledge is represented with a “state vector”, from which each coordinate describes an internal property. A subset of these internal properties, called state parameters, are limited by threshold defining the critical state. A specific behavior is then associated with each threshold. The agent’s stability is defined as a satisfaction state with regards of its role. An agent is then stable when the state parameters values do not reach the threshold. The agent behavior can be internal or external. An internal behavior describes the agent life-cycle that performs actions continuously without external solicitations. Internal behaviors ensure the independent part of the agent. For this reason, agent’s behavior should be implemented as a Thread [8] (in the Java sense of it, that is parallel process).

Figure 3. OMT diagram of the SocietyModel Unit and the AgentModel Unit

External behavior are provided by the agent when receiving external solicitations (such as events). For instance, one of the scenarios for such an external behavior can be: (1) update the state vector, (2) compute the rest of constraint to propagate in neighborhood, (3) choose agents to which propagate the event and (4) really propagate the event.
The autonomous part of the agent is embedded in an unit called Autonomous Unit. This unit owns two classes: the Consciousness class and the IndividualConsciousness class. The first one inherits from Thread (given in the Java Development Kit and allows parallel processing). It is an abstract class because it will never be instantiated. As it is a subclass of Thread, it should implement the method named `run()`. The second one is very important for the agent. It manages them. It inherits from Consciousness and defines methods such as: `behaviorControl()`, which launch a new behavior; this mechanism allows to keep room for decision making, since an agent's behavior is dynamically self-adapted during the simulation; `perform()` which allow to call a method only with his selector and his parameters, this method simulates the `perform` we can find in Smalltalk [9] which does not exist in Java. This is very useful functionality to setup a generic method invocation mechanism for autonomous agents.

Class In
- extends Observer
- Agent agent
- Graph acquaintancesList
- EventsList listRemainingEvents
- catchMessage()
- takeFirst()
- update()

Class Out
- implements Observable
- Agent agent
- EventsList messageList
- sendAllAcquaintances()
- sendDirectMessage()
- setChanged()
- notifyObservers()

Figure 4. Class description of asynchronous message passing part in GEAMAS V2.0

Message Passing with Java: an important part of agents is its asynchronous part. This part is derived from the use of two mailboxes: In and Out. The In class stores messages that should be performed by the agent ordered by intensity and arrival. The Out class stores each message to be propagated in the agent's neighborhood. To implement this mechanism, we use two pre-defined classes of the Java Development Kit: `Observer` and `Observable` [8]. The "observable" object concept has been borrowed from Smalltalk [9] with the Model-View-Controller (MVC) approach. In this language, an object can be observed by another one. In Smalltalk this mechanism is embedded in the `Object` class and not in Java. In Java the observed object should inherit from the pre-defined Observable class. When it changes, it invokes two methods: firstly, a method called `setChanged()` which marks the object as changed; secondly a method called `notifyObservers()` which notifies all observer objects. That’s why the Out class implements the Observer interface and the In class inherits from the Observable class.

In conclusion of the asynchronous message passing part, the Figure 4. above describes the Java class description of the In and Out mailboxes.

3.3.3 Agent System Unit: because the GEAMAS V2.0's Kernel is able to run without any attached GUI application, it has been necessary to create a link between the Kernel and GUI Applications. This link is then very important and we have brought a particular attention to it. Two essentials roles are affected to this link. Firstly, the unit Agent System realizes the link between the Kernel and GUI applications; secondly, this link as been implemented for learning user's comportment and predict his action. For this reason, an Agent System is a kind of Agent. With an inheritance link, an Agent System inherits from all Agent's capabilities. Finally, it owns two mailboxes. For instance, an agent can send the message "my state has changed" to the Agent System that will forward this message to the attached GUI application.

Figure 5. below describe the OMT model of the Agent System Unit.

In our Kernel, we have chosen to implement this link at the upper level, precisely in the Society unit. We have added an aggregation link between the society and the AgentGUI class.

More, the AgentGUI class is able to save a simulation into a file, thanks to the serialization’s utilities given by the Java Development Kit. A description of the serialization concept and implementation can be found in [8]. A class can be saved if and only if it implements the Serializable interface. That’s why all classes in the Kernel implement this interface. This facility is interesting for two reasons: firstly, with only one call of method, all the society and its dependencies can be save; secondly, the file can be compressed during the writing file operation. By the same way a serialized file can be read and all objects in it
are retrieved. The only default is that links, such as observer links, are not saved and the AgentGUI class should rebuild them when a simulation is loaded by covering all the acquaintances’ graph.

The next section details a tool developed over the Kernel. This tool named "Generation Environment" is a GUI Application which generate Java code.

4. The Generation Environment and the Simulation Environment of GEAMAS V2.0

4.1 The Generation Environment

Section 3 reports the generic Kernel of GEAMAS V2.0. But our goal is to reduce the complexity of application designing. That’s why the Generation Environment has been developed. It comes in help a user who wants to create a simulation.

With the generation environment, the user can define, through GUIs, different agents, a macro-agent, medium-agents, and micro-agents. More, the Generation Environment should be able to: describe different types of agents and general comportment for this types, and generate associated code in Java.

As our Kernel is generic, we can build a specific application only with deriving classes from the Agent class or the Society class and overriding methods. The creation of a new application has to be realized in four steps: firstly, identification of requirements; secondly, definition of the appropriate micro-agents, medium-agents, and macro-agent; thirdly, in each type of agent, the definition of the consciousness, behaviors, input events, and output events; fourthly interactions between agents.

More precisely, this environment owns two main windows: on the one hand the Agent Designer Window, one the other hand the Society Designer Window. The first one allows to define the following parts: the agent’s name, the description of this agent, the agent’s consciousness, agent’s external and internal behaviors, agent’s external and internal parameters. And the second one allows to define the following parts: the society’s name, the society’s description, the agent’s part of the society (remember that a society is a kind of agent), the society’s filtering function, global constraints of the system, and input and output events of the system.

4.2 The Simulation Environment

The Simulation Environment is a GUI application which provides tools to set and control a simulation created with the Generation Environment. This tool is able to: set the global context of a system, set locally each agent in the system, dynamically control the global evolution of the system, dynamically control the local evolution of each agent in the system, and dynamically observe system’s evolution, and precisely observe the medium-agent’s emergence.

Our Simulation Environment uses the notion of Bean. According to the Sun Microsystems’ definition, a Bean is "a reusable software component that can be manipulated visually in a builder tool" [4]. Our purpose is not to develop a builder tool but a Bean which is able to read information in a class. The Simulation Environment reads and initializes classes defined by the generator, and for instance, the Simulation Environment can dynamically modify the value of a state vector of a society or its input event.

Moreover, our environment is able to define the acquaintances’ graph. In a frame you can define the number of agents at the beginning of a simulation, if an agent is an input agent or not, or if it is an output agent or not. You can also define the acquaintance relation between two agents and the strength of their relations.

After this stage, the first operation to do is to initialize all values of the simulation, for instance the input event value. The second one is realized by the application, it instantiates the society, the graph structure, micro-agents, medium-agents and the macro-agent; it assigns each agents’ state vectors; and finally the environment assigns the structure composed of micro-agents and medium-agents to the society.

5. Conclusion

This paper has presented a model and object implementation features of GEAMAS V2.0 which is structured in three modules. The first one is a kernel which implements generic agents and societies. The second one is a code generation module to model graphically new applications of non linear complex systems and generate the associated code. And finally, the third one is a simulation module providing the ability of initializing and observing the simulation’s evolution. We successively described the kernel implementation and its particularities, in particular the asynchronous messages passing between agents, the control of the agent’s behavior, and the connection between the Kernel and a Graphical User Interface; the Generation Environment and the Simulation Environment.

We point out the fact that oriented object approach is very advantageous to model agent and its associated specificity. Inheritance and aggregation allow to build easily and quickly complex applications. Moreover, as we have implemented GEAMAS V2.0 with Java, precisely with the "100% pure Java" standard, our platform is architecture neutral and portable on every systems implementing a Java Virtual Machine.

We have applied GEAMAS V2.0 to “sand-pile automaton” problem[1] and we have obtained very encouraging results for instance, we have obtained similar result than in-situ observations. At this moment, we are initializing a new survey on “ecological phenomenon simulation for the Reunion Island” in cooperation with the CIRAD International Co-operation center for agronomic research and development.

At last, we are now working on an agent based approach of the Environmental Unit which is, at this date, an empty part of GEAMAS’S Kernel. This new version of the GEAMAS’s Kernel will be tested with a simulation of lava flows on the Piton de la Fournaise volcano (Island of La Réunion).

Acknowledgment

It is a pleasure to acknowledge the support of the Institut de Recherche En Mathématiques et Informatique Appliquées (IREMIA) for the stimulating intellectual environment it provided to us, as well as financial support of this project.

References


