

DIMA

Zahia Guessoum

Team Objects and Agents for Simulation and Information Systems (OASIS)

Laboratoire d'Informatique de Paris 6 (LIP6)

8 rue du Capitane Scott, 75015 PARIS

e-mail: dima@poleia.lip6.fr

<http://www-poleia.lip6.fr/~guessoum/DIMA.html>

1 Introduction

To validate the operational platform (*DIMA*), we have developed several applications e.g., a manufacturing process simulator, a multi-agent system to control mechanical ventilation, etc. In this section, we report on some simple examples which illustrate the use of DIMA.

2 Example 1: Circle

In this first example (package `dima.kernel.examples.circle`), we explain the use of the kernel of DIMA. This kernel allows to define simple entities that are pro-active.

The agents aim to form a circle. Each agent knows the center of this circle and its position. The implementation describes the agent environment (the grid) and the agent pro-activity. The grid is described by the class `Ocean` and the agents are described by the class `Shark`. The latter is subclass of `BasicReactiveAgent`.

3 Example 2: Preys/Predators

In this second example (package `dima.kernel.examples.preyspredators`), we have two kinds of agents: Preys and Predators. Each agent is created with an amount of resources and it has a limited perception of its environment. A prey (described by the class `Actor`) moves randomly and is alive while it has resources and has not been captured by a predator. A predator (described by the class `Predator`) is alive while it has resources. The aim of the predators is to capture a lot of preys to have decrease their resources.

4 Example 3: Factorial

This example (see package `dima.kernel.examples.Factorial`) is mainly proposed to illustrate the communication between agents. We consider two kinds of agents:

- **AgentFact**: these agents have the needed behaviors to compute a factorial but they don't have the behavior to multiply numbers.
- **AgentMult**: these agents have a behavior to compute a multiplication.

Agents are implemented as subclasses of `ReactiveCommunicatingAgent` and its subclasses. This class provides the basic communication behaviors:

- **readMailbox**: removes a message from the mail box, if the latter is not empty, and processes it.
- **sendMessage**: sends a message to one or several agents. The message is instance of the class : `Gdima.basiccommunicationcomponents.Message` or its subclasses (KQML messages). The class `Message` is mainly defined by: a content (name of the method), its arguments, the identifier of the receiver.

Note that the algorithm that is used by `AgentFact` is different from the classical one. To compute `factorial(n)`, `AgentFact` creates a list with the `n` number from 1 to `n`. Then it sends requests to `AgentMult` with all the possible couples of numbers. When it receives a result, it puts it in the list. If the latter has more than one number, new requests are then sent `AgentMult`. It repeats this actions while the list contains more than one element or `AgentFact` has not the responses to all the sent requests.

5 Example 4: Eco Problem Solving

This example (see package `Gdima.kernel.EcoResolution`) was developed by Alexis Drogoul. It has then been reused and integrated in DIMA with very small changes. The following description (all this section) has been written by Alexis.

Eco-Problem-Solving is a different approach to problem solving than the more traditional AI or DAI problem-solving methods used so far.

It is strongly related to ALife techniques such as those being studied in Swarm Intelligence or self-organizing systems. Problem solving is seen as the production of stable states in a dynamic system, where evolution is due to the behaviors of simple agents. By "simple" we mean agents that do not make plans but just behave according to their specific program, called eco-behavior, which is just a combination of "tropisms", i.e. behaviors made of reactions to their environment (including the other agents).

Instead of considering a problem as a whole, we then decompose it into a set of small entities, each being considered as an independent reactive agent with its own goal and its own behavior (see figure below). This behavior simply consists in interacting with the other agents to achieve its goal, in a strictly codified way which follows these two simple rules :

- The desire to be satisfied and the possibility to attack other agents to do so,
- The necessity of escaping when being attacked

As all the problem is described using agents, the goal of an agent will always be another agent (e.g. another block in the blocks' world, a stick in Hano towers, etc.). This enables us to describe specific relationships between the agents, called dependencies : an agent will always be dependent on its goal (which means that it will have to wait until its goal is itself satisfied to satisfy itself). It allows us to dynamically (and at any time) have the subgoals being ordered in a correct way. The overall objective (i.e. the solution of the problem) will therefore be achieved when all the agents are satisfied, that is when they all have reached their own goals, and do not have any other agent to attack.

Any EPS application is divided into two main modules: 1) The first one (called the kernel) is independent from the domain of application. It contains the definition of the abstract class `EcoAgent` which defines generic methods allowing the instances of its subclasses to satisfy themselves, flee, etc. 2) The second module is dependent on the application domain. It is made up of concrete subclasses of the

`EcoAgent` class. The instances of these classes will be the agents that populate the problem. The main methods of these subclasses called by the previous ones, which implement the concrete behavior of the agents (i.e. how to satisfy itself, how to flee, how to attack, etc.).

Defining a set of subclasses respectively provided with this set of methods is all what is needed for describing a problem in EPS. However, as in many Alife applications, the tough work is the experimental study of the resulting collective achievement. Though a good analysis of the problem usually provides the designer with the correct classes of agents, their behaviors have to be determined empirically.

However, incredible results have been obtained in solving some classical problems, such as the famous N-Puzzle, for which we have been able to reach very large sizes (99-Puzzle !) while keeping a very good quality of the solutions (*10moves*).

6 Example 5: Agenda

This example was proposed by a French Special Interest Group on multi-agent systems. It is implemented in the package (`dima.kernel.communicatingAgent.examples.TimeTable`) (<http://www-poleia.lip6.fr/~guessoum/asa.htm>)... Sorry it is in french. I'll translate it very soon.

7 Example 6: Auction

We consider an example of simple auction. We have two kinds of agents : Auctioneer and Bidder. Each auction involves one Auctioneer and several Bidders. The Auctioneer has a catalog of products. Before starting the auction, the Auctioneer sends the catalog to all the participants. Then it starts the auction for all the products. The product are therefore proposed sequentially to the participants. The meta-behavior of the Auctioneer is described by the following state machine (see Figure 1). The latter is implemented by the static method: `public static ATN ATNEnchere()` of the class Auctioneer.

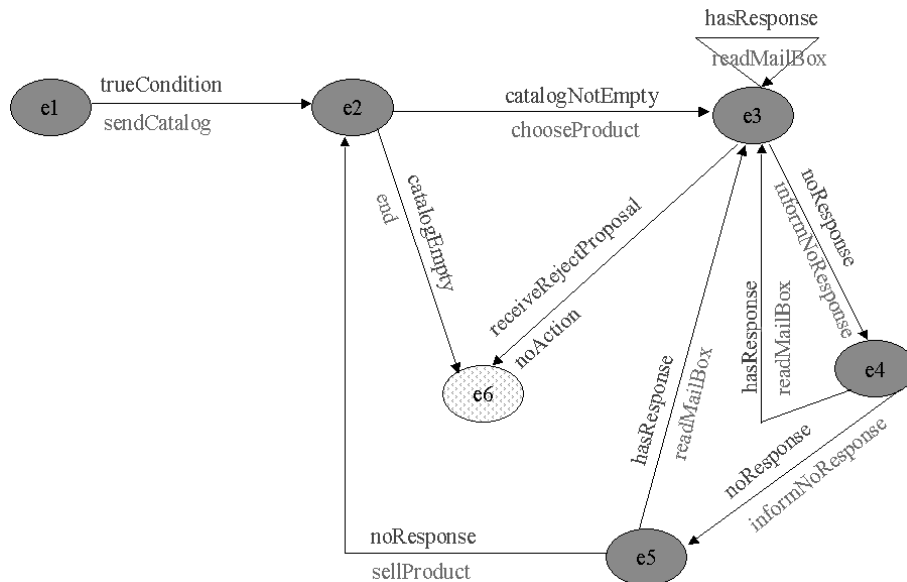


Figure 1: The meta-behavior of the Auctioneer

8 Example 7: e-Agenda

We consider the example of a multi-agent system that helps at scheduling meetings (see packages `eAgenda.*`). Each user has a personal assistant agent which manages his calendar. This agent interacts with:

- the user to receive his meeting requests and the associated information (a title, a description, possible dates, participants, priority, etc.). An interface is provided to interact with the user. The latter can therefore plan all the needed meetings.
- the other agents of the system to schedule meetings.

The agents are described by the class : `eAgenda.mas.AgendaAgent0`.
The main class to run a simulation is : `eAgenda.simulationSimple.SimpleSimulation0`.
The meetings are generated automatically by the method: `generateRandomAction`. The interface of this simulation gives the criticality of agents.