A NEW GENERAL, FLEXIBLE AND JAVA-BASED SOFTWARE DEVELOPMENT TOOL FOR MULTIAGENT SYSTEMS

Tony Garneau & Sylvain Delisle
Département de mathématiques et d’informatique
Université du Québec à Trois-Rivières
Québec, Canada, G9A 5H7
phone: 1-819-376-5011 + 3832
fax: 1-819-376-5185
email: {Tony_Garneau, Sylvain_Delisle}@uqtr.ca

Keywords: Agent-Based Systems and Designs, Technologies for Distributed Applications.

Abstract
There has been a great interest in using multiagent systems (MAS) in a wide variety of applications in recent years, especially Web-based applications such as those in the field of electronic commerce. Agent-oriented methodologies, development environment and tools have become a priority for the development of large-scale agent-based systems. The work we present here belongs to the disciplines of Software Engineering and Distributed Artificial Intelligence. More specifically, we are interested in software engineering tools involved in the development of multiagent systems (MAS), particularly Java-based software tools that can facilitate the development of MAS. This paper is a logical extension of our recent research on MAS. Initially we studied MAS development methodologies. Then, we considered MAS development tools and environments that we evaluated and compared. This comparative evaluation was carried out in order to identify tools’ and environments’ strong points, as well as weak points, in the hope of determining what a complete development environment should look like. Finally, we undertook the development of a complete MAS development environment essentially meeting the set of requirements identified in our comparative study. Everything related to our MAS development environment/tool in this paper is 100% implemented in Java. At the time of writing, we are conducting a testing phase in which many people participate, all being external to our research team.

1. INTRODUCTION
Although recent years have witnessed a great number of theoretical and practical projects in the field of multiagent systems (MAS), the development of such systems still poses many challenges and thus remains an open and very active field. One crucial aspect is the availability of adequate software engineering methods, and software development environments and tools. These problems are undoubtedly bound to the fact that current MAS methodologies and development tools are incomplete. Another factor is the lack of standardisation in MAS platforms and environments. In previous work [Sabas et al., 2002], we have performed a comparative analysis of MAS methodologies and proposed a first step toward a unification scheme. In this paper, we present a comparative evaluation of software-based tools for the development of MAS and also present the current status of our work on the implementation of a new general, flexible and Java-based tool for the development of MAS systems.

We are about to complete the design and implementation of a MAS software development environment called DMAS Builder. The tool supports the development of totally distributed and OS (operating system) independent MAS and provides a package of classes that facilitates the development of MAS. Many interfaces allow the developer to easily specify different system components. The automatic source code generation (in Java) of all system components allows the developer to save a lot of implementation time and can also fix several potential errors. The system thus developed is totally independent of the specific technological environment in which it will evolve, as long as it supports Java. This is made possible by a mechanism that extracts the necessary classes from the AO (agent-oriented) tool package. Another mechanism allows the automatic creation of archive files (.jar) for each sub-system. Moreover, it is possible to compile and execute the system directly within the development environment.

2. AGENT-ORIENTED AND MAS DEVELOPMENT TOOLS AND ENVIRONMENTS
The emergence of agent-oriented programming has lead to the development of several methodologies and architectures for the modeling of multiagents systems. Here are some: MaSE (multi-agent software engineering) [DeLoach and Wood, 2001], Aaladin or AGIR (agent, group, role) [Ferber and Gutknecht, 1997], RETSINA [Sycara et al.,], dMARS [d’Iverno et al., 1997], OAA (open agent architecture) [Martin et al., 1999], DESIRE [Brazier et al., 1997], Gaia [Zambonelli et al., 2000], Tropos [Mylopoulos et al.,], and Kaos [Bradshaw et al., 1996]. The concept of agent-oriented programming is a very interesting idea and associated methodologies offer a theoretical basis for MAS modeling. However, agent-based systems specified from these methodologies are often difficult to implement directly with standard programming languages like Java or C++. Indeed, there seems to be a wide gap between the very high level of most MAS development methodologies and conventional programming languages.

Several different kinds of agent-oriented programming tools have been proposed recently. Among these tools, ten draw our attention because of their popularity and their relevance: JADE [Rimassa et al., 2000], Zeus [Lee et al., 1999], MadKit [Ferber and Gutknecht, 1997], AgentBuilder [AgentBuilder U. G. 2000], Jack
3. THE CRITERIA DEFINING THE COMPARATIVE EVALUATION

If one wants to develop useful and applicable MAS development tools, several characteristics are essential in practice. Some weaknesses seem to be of chronic nature and research efforts should be made to take them into consideration. Here are several significant characteristics that should, in our view, define a complete MAS development environment:

- Simple support for the deployment of inter-computer and distributed applications.
- Support for the use and integration of databases.
- Significant reduction of the effort required to implement a MAS, especially when considering the important amount of programming usually involved.
- Effective and efficient abstraction in order to allow people less experienced with agent-oriented programming to relatively easily create MAS without in-depth knowledge of all implementation details.
- Sufficient latitude to most experienced users so they can get access to and interact directly with various system components—the tool should add an abstraction level to programming without becoming an obstacle for programmers.
- Implementation (code) is easily extensible.
- Developers do not have to worry about system communications implementation and associated protocols used to transfer messages between agents.
- Availability of a debug utility, of a user interface facilitating development, and of an automatic source code generator.
- The environment is supported by a suitable documentation.

3.1 Objectives and Criteria

The evaluation criteria we selected are directly related to the objectives which, in our opinion, must at least be met to consider a tool a valuable MAS development environment. Obviously, the evaluated tools are not all true MAS development environments and their objectives do differ. Some can possibly obtain a rather poor evaluation according to our grid, but it might be the case that they nevertheless perfectly satisfy the original goals that triggered their creation. The essential objectives of a MAS development environment are:

- Accelerate the development and decrease the programming effort.
- Abstract communication, interaction and coordination mechanisms.
- Allow implementation of relatively complex systems.
- Allow uncomplicated code extensibility.
- Provide support for deployment (and execution) of systems.

3.2 Evaluation Scale

The evaluation scale we originally used in our work was much more detailed. However, we simplified and standardized it so that it would be more intuitive. The final evaluation scale used to indicate how a certain criterion fares for a certain tool is a number ranging between 0 and 4 which is interpreted as follows:

- 4 if the tool meets the criterion very well;
- 3 if the tool meets the criterion well;
- 2 if the tool meets the criterion moderately;
- 1 if the tool meets the criterion a little;
- 0 if the tool does not meet the criterion at all.

3.3 Description of the Evaluation Criteria

The tool’s methodology covers the different development steps (1)

The methodology covers the various stages of the development process. Like the majority of authors, we consider that the MAS development process consists of four main stages: analysis, development, implementation and execution (deployment). Often, the methodology is only loosely coupled with the tool.

Tool learning facility (2)

This criterion is determined according to several features such as documentation quality, components’ complexity and clarity of the concepts used. The knowledge required to correctly use the tool, including the programming language(s), the communication language(s) between agents, the interaction protocol(s) are taken into account.

Simple transition between development and implementation (3)

Facility to progress from the model to its implementation. Several developed methodologies are very interesting at the conceptual level, but not easily applicable, in particular with regard to the implementation details.

Tool flexibility (4)

The tool’s flexibility and the versatility towards the use of its components and its methodology.

Communication between agents (5)

The programmer should not have to be concerned with the implementation of low-level connections between various computers, communication protocols, security management, synchronization, message transport services and so on. These services should be already implemented and at the developer’s service.

Debug utility (6)

Several coordination and synchronization errors are likely to slip into programs during the development of a MAS. The discovery and correction of these errors can be very difficult or even impossible without suitable debugging tools.

Graphic support for development and implementation (7)

The environment proposes graphic interfaces facilitating and accelerating the development and the implementation. These can be used for model creation, agents creation, conversations development, and agents deployment on many platforms.
MAS management support (8)

The tool allows interaction with the system. It allows, for example, dynamically adding, modifying, or removing agents in the system. The interest of this type of management is significant; it can be very useful to study the system at execution, checking and validation levels.

Implementation simplicity and reduction of the required effort (9)

With this criterion, several factors must be taken into account. A language supporting well object-oriented programming, multi-threading and network programming has significant advantages over others that do not. The components must also be easily identifiable (name, package, documentation, parameters, etc.). Moreover, the classes and the services available must be easy to use. The reduction of the required effort in terms of quantity of code to write, components complexity to implement, simplicity to use existing components are also factors to consider.

Database support (10)

Data conservation and protection is a technical task at the programming level. It is interesting to abstract this process as much as possible and provide tools facilitating it.

Automatic code generation (11)

If system specifications are available at the interface level, it is quite useful to be able to generate system source code (at least templates) and its various components.

Code extensibility (12)

Utilities provided by the tools, such as predefined modules, agents or generated code, must be easily modifiable. It is also necessary to be able to easily add code to those existing pieces of code.

Deployment (13)

The possibility of distributing the system on several computers is a very important criterion at the execution level. The tool must also allow for a simple system execution. The execution must be independent of the environment.

Documentation (14)

The available documentation is of good quality. It covers all the components of the tool. Moreover, it is clear, concise and not ambiguous.

Other criteria were considered during our initial comparative evaluation. However, a numerical evaluation of those did not seem necessary or even desirable to us. They were thus taken into account during the global evaluation but are not part of the final grid below. Among these criteria, we had the methodology used for development, the communicating language between agents and the programming language.

4. PERFORMING THE EVALUATION: RESULTS

We evaluated 8 tools/environments for MAS development using the 15 criteria introduced in section 3.3—notice that criterion 7 is twofold. We chose these eight tools primarily because of their current popularity and accessibility ([Garneau 2002]).

4.1 Results Grid

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Tools</th>
<th>JADE</th>
<th>DECAF</th>
<th>AgentBuilder</th>
<th>Zeus</th>
<th>JAFMAS/JIVE</th>
<th>Jack</th>
<th>AgentTool</th>
<th>Madkit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methodology (1)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Learning facility (2)</td>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Step transition (3)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Tool flexibility (4)</td>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Communication (5)</td>
<td></td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Debug utility(6)</td>
<td></td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Development support (7)</td>
<td></td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Implementation support (7)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MAS management (8)</td>
<td></td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Implementation simplicity (9)</td>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Database (10)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code generation (11)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Code extensibility (12)</td>
<td></td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Deployment (13)</td>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Documentation (14)</td>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total /60</strong></td>
<td></td>
<td><strong>29</strong></td>
<td><strong>15</strong></td>
<td><strong>39</strong></td>
<td><strong>42</strong></td>
<td><strong>20</strong></td>
<td><strong>22</strong></td>
<td><strong>26</strong></td>
<td><strong>30</strong></td>
</tr>
</tbody>
</table>

4.2 Global Evaluation

Originally, two additional tools were considered for the evaluation: RMIT and Brainstorm/J. However, these two tools obtained very low scores and we decided to discard them from the evaluation. In fact, it appeared to us that these two tools were developed more like tools intended for internal use within the research team itself. Both RMIT and Brainstorm/J are packages of classes implemented in Java. Both have a significant lack of documentation. Defined architectures and classes are very complex. Moreover, no graphic tool supports the development process. Finally, their components are quite complex, and they lack support at the debugging and deployment levels.

Jack and JADE can be viewed as “frameworks”. They are more complete than RMIT and Brainstorm/J, however these two tools do not have any specification at the methodology level. Moreover, both have great gaps in terms of graphic utilities. Their documentation is good and covers the majority of their respective tools. Both offer great possibilities at implementation level even if it requires a lot of efforts. The main difference between these two
tools is that the development with JADE is done in pure Java contrary to JAL for the development with Jack.

DECAF, JAFMAS (and JiVE), MadKit and agentTool are environments supporting various levels of the development process. DECAF and JAFMAS emphasize planning and interactions between agents but do not deal with implementation and deployment. AgentTool concentrates on the first phases of development but the implementation is not taken into account by this tool. MadKit puts more emphasis on deployment than on other stages of development (even if it is based on the Aalaadin methodology). MadKit offers the best documentation compared to the three others.

Zeus and AgentBuilder obtained the two best scores (42 and 39, respectively) of the eight evaluated tools. They also are the two most complete tools of the group. These tools have an abundant documentation. They offer several graphic interfaces helping the development. Moreover, they have a debug utility. Each one proposes a development methodology. However, these two tools are very complex and they require significant training to be understood reasonably well, and much more to be mastered. AgentBuilder requires knowledge about RADL (Reticular Agent Definition Language), ontology management, the execution engine, the protocols used and the mastering of other interfaces. Zeus also requires a good bit of training, notably to master the “role modeling” technique, as it is essential in Zeus. It is also necessary to understand the various editors that provide several more or less advanced services (ontology, tasks, resources, rules, interaction protocols, system visualization, etc.). Both Zeus and AgentBuilder have weaknesses at the extensibility and flexibility levels.

In conclusion to this comparative evaluation part, we observe that the majority of these tools were developed to exploit, show or prove a particular concept or idea. So the development of these tools neglects, voluntarily or not, the development of several essential dimensions of MAS implementation. This makes their use essentially inappropriate or event impossible for real system development. We suggested above several characteristics that MAS development environments should have. From those, we built a grid of criteria and evaluated eight MAS tools. The evaluation grid enabled us to remark some tendencies between these tools:

- Tools providing graphic utilities for development and implementation have all a lack at extensibility and flexibility levels.
- Tools that provide interfaces for implementation decrease programming effort.
- The majority of tools do not support any mechanisms for data conservation and protection.
- It largely simplifies and accelerates MAS implementation.
- It allows the automatic source code generation of all system components.
- It has been designed to handle the development of relatively complex systems.
- It supports very good code extensibility.
- It allows platform independent development.
- It offers an abstraction of communication mechanisms and protocols between agents and various computers.
- It supports the development of environment (tool) independent applications.
- It allows very simple system deployment.

DMAS Builder is completely implemented in Java and allows MAS development in Java. The environment has two main components: the development environment and the package of classes supporting agent-oriented programming. In this section, we describe the current implementation of our MAS development tool.

### 5.1 Architecture and Methodology Used

One of the evaluation criteria we considered in our work is the development methodology. At the beginning of our research on MAS, we hypothesized that MAS development tools had to include a development methodology. However, after having completed our comparative evaluation and after several discussions with colleagues working in the field of MAS and research teams using this type of tools, we concluded that it was in fact preferable not to impose a specific methodology to the user. The main reason is that the imposition of a specific methodology often constitutes a restriction to both the developer and the types of MAS projects to which it can be profitably applied. However, we want to stress that we do not want, by this standpoint, to minimize the utility and importance of MAS methodologies. We only state that the choice of the specific methodology to be used in a particular project should be left to the user. Thus, the development environment we propose does not impose a specific MAS development methodology.

DMAS Builder expects the development to proceed in subsystems thus allowing the implementation of completely
distributed MAS. We do not allow the creation of more than one subsystem on one JVM because we determined that there was no valid reason to execute more than one subsystem on the same JVM. For this reason, in our case, talking about a JVM or a subsystem is equivalent. Each subsystem is comprised of one or more agents. For each agent, it is necessary to define the set of tasks it can perform. The agent implementation is task-oriented in the sense that behaviours and DF Agent types provides functionalities based on agents tasks capabilities. However, the tool is generic enough to easily use the development environment and the packages of classes with other architectures that are goal-oriented or role-oriented.

Each agent has an independent communication system, and each one can communicate with any other agent: see appendix. Moreover, each agent determines the way by which it will receive messages: an agent can receive messages by RMI, TCP Socket, UDP Socket, multicast Socket or directly (when they are on the same JVM). However, it is often useful and more effective to add a DF Agent (Directory Facilitator Agent) that handles messages routing on a computer when several agents on the same JVM (Java Virtual Machine) want to communicate with other agents located on other computers. Many different types of DF Agent are available offering a variety of functionalities.

5.2 Development Environment

The development environment is a graphic utility having several user interfaces. It has a file editor for files visualization (Java files or others), Java implementation and project compilation. The editor also provides a documentation tool that helps the user find topics in the JavaDoc (environment package classes and Java SDK 1.4.1). The editor specifies file types for all files included in a project:

- Source file: (.java) file added to the project by the developer.
- Generated agent: (.java) file representing the specification (via the system editor) of a particular agent.
- Generated agent type: (.java) file representing the specification of a particular type of agent (system editor).
- Generated knowledge base: (.java) file representing the specification of a particular knowledge base (system editor).
- External resource: any other file added by the user to the project.

Several editors simplify various system parameters initialization such as the path of the required execution program (java), the compiling program (javac), and others.

A help editor supports indexed searches inside the environment JavaDoc (in the package classes), in the Java API 1.4.1 or in both at the same time. In this way, a complete documentation is available in any time to the user. The user determines the API to be used for a more precise and effective search. Separate API in the help avoid to flood the user with useless documentation in particular circumstances.

The main editor is used to make all system specifications; it is the system’s editor. This editor allows the specification of each subsystem (host) and their various components. We determine the type of an agent when it is created. This type is determined according to its behaviour (several kinds of behaviour are available), its architecture (Belief-Desire-Intention or other), and some other specifications. We must specify every task that could be carried out by the agent. At the agent level, we will determine which among the tasks list could be carried out for each agent. The system editor also serves to create agent knowledge bases (if necessary). We must also specify agents for each subsystem (JVM). For each one, we specify various parameters (which have default values) such as the communication mode (RMI, TCP Socket, UDP Socket, Multicast Socket or Direct), the knowledge base used (BDI agent), the agent’s set of tasks (agent abilities), and others parameters. We can add an DF Agent for each JVM. Our development tool provides many kinds of DF Agents offering different types of services and functionalities. It is also possible to indicate that we want an automatic recording of subsystem agents to others JVM. In this case, an instance of RegisterAgent is created for this subsystem.

Four utilities are also provided by this main editor: system validation, automatic source code generation of all system components (agents, agent types, DF agents, register agents, knowledge bases, tasks, behaviours, etc.), creation of necessary source files to allow independent system execution (from the environment), and creation of executable archive (.jar) for each JVM. Let us now have a quick look at those four utilities.

System validation

This option allows the checking of the system’s specification logic. This validation facilitates the finding of inconsistencies inside components or in the relationships between them. Moreover, the validation step checks that words used in the code (names, types and variables) are valid for system source files compilation since it is necessary to use acceptable words as valid Java identifiers. The validation also allows avoiding the introduction of compilation errors during the system’s automatic generation process.

System generation

The system generation allows the automatic creation of all the system’s source files. The generation is done in the following way. First of all, a folder that includes all source files is created for each subsystem. Then, a file is generated for each agent, task, agent type and knowledge base for each JVM. Third, one file (containing the “main method”) being the program entry point is created for each host. The generated files are well commented: this facilitates comprehension of the system’s structure and makes it easier to find places where we should add agent-specific code. Finally, the generated subsystems (applications) can be compiled.

System export

This option allows the creation (copying) of files belonging to the environment package and which are necessary for independent subsystem execution. Each JVM will have the required files for its execution. Needed files for a subsystem are not generated in other JVM. This procedure reduces the size (in bytes) of each subsystem. System exportation allows great code extensibility and makes the development tool totally independent of the specific machine on which it will run. Since the subsystem now contains the necessary environment package classes (because there are copied in the JVM folder), it is possible to bring the JVM folder on another computer and continue the implementation without our development environment. It is also possible to modify the environment source files (those which were copied in the subsystem folder). In this way, it is possible to extend existing functionalities directly inside the MAS package.
**Sub-systems archive**

This functionality allows the creation of an executable file (.jar) for each subsystem (JVM). This executable file contains the JVM binary files and the required MAS package’s files necessary to run the subsystem. A file specifying the JVM entrance point (main class), thus allowing jar file execution, is found there too.

**5.3 AO (Agent-Oriented) Package Classes**

As mentioned above, one of our working hypotheses is that it is useless to run more than one JVM on a computer. Consequently, every agent on a JVM has the same IP address, but each one is totally independent from other agents.

**Agent class**

The “Agent” class is the highest class in the agent hierarchy. This class is a “Thread”; thus each agent runs independently of other agents. Each agent has a name and a GUID (Global Unique Identifier), the latter being unique in the system. The GUID is used to recognize and distinguish each and every agent in a system. The agent is registered at the ANS (Agent Name Server) with its GUID and agents that want to send messages to it must know its GUID. Each agent has a “Mailbox” which serves to send and receive messages. The agent has a tasks list that it can achieve. It is possible to specify the agent’s behaviour by implementing an interface corresponding to the desired behaviour type. The interfaces corresponding to behaviours have methods that normally must be defined in the implementing class. But in this case, it is not recommended to redefine these methods. The “Agent” class redefines all necessary methods to adapt these behaviours. The execution of behaviour methods is dependent on the implemented behaviour sub-interface. Several other utilities are included in this class. For example, a utility allows putting messages in the Java event queue.

**Mailbox class**

The “Mailbox” has two mandates: receive and send messages. The first one is to receive messages arriving from other agents, be they on the same JVM or on another computer. When an agent “Mailbox” receives a message, it adds it to the entering messages file. If there is no message in the file (i.e. if it is the very first message), it notifies the agent about the reception. The message reception by a “Mailbox” can be accomplished in various ways. The direct mode only allows the receiving of messages directly in the incoming box. This implies that the sending agent is on the same JVM or that a DF Agent is on the receiving agent’s JVM. Other reception modes are: RMI (Remote Method Invocation), TCP Socket, UDP Socket, Multicast Socket. These modes support the receiving of messages coming from agents on the same JVM or from other computers (subsystems) even if no DF Agent exists on the host.

The second mandate of the “Mailbox” class is to send agents’ messages to other agents’ mailboxes. To send a message, an agent only has to add the desired message in its mailbox send list. The mailbox determines where (which agent) and how (which communication mode) to send the message. Then, the mailbox sends the message to each recipient with the specified communication mode. For example, if an agent’s mailbox listens to messages via TCP Socket on port 5555, then it sends messages to this recipient via TCP Socket on port 5555. If another agent mailbox waits for messages on a multicast address, then the agent mailbox sends the messages to this address. To allow a better effectiveness and continuity in agent execution, the sending mechanism is running inside a different “Thread”.

This communication mechanism allows the development of totally distributed MAS since each agent is free to communicate with whichever agent it wants without the need for a broker agent, and without the need to share resources. This mechanism is independent of other system entities.

**Task class**

The Task class is a generic class, which allows the creation of jobs that could be added to the system tasks list. This class contains a method “execute” which must be redefined. This method represents the action that must be done to consider the task as being completed. A mechanism supports the passing of parameters to the task before its execution. Another mechanism allows the return of results to the agent after task execution.

**Message class**

This class is the super-class of all messages types that can send messages from an agent to another. It has several useful methods to allow a mailbox to determine the recipients.

**ANS (Agent Name Server) class**

This class is used as a directory (white pages service). It allows the finding of necessary information to communicate with agents. Indeed, it can find the address, the communication mode, the port (if necessary), and the name of an agent. This information about an agent is available, provided we know the agent’s GUID.

**Many classes for building knowledge bases**

It is possible to build knowledge bases thanks to about fifteen classes and three interfaces. We can create them by using user interfaces or directly into the source code. These knowledge bases can be integrated into agents.

**DFAgent (Directory Facilitator Agent) class**

The “DFAgent” class and its subclasses supply interesting services to the JVM: the possibility to find an agent to execute a particular task, and sending or receiving all subsystem agent messages to avoid overload of necessary communication “Threads” (yellow pages service).

**RegisterAgent and MulticastRegisterAgent classes**

These two classes are very useful. They allow the automatic recording of agents from any JVM to any JVM. The “MulticastRegisterAgent” class can execute subsystems without specifying the JVM addresses. The agents’ registration to other JVM is automatically done without external intervention.

**Behaviour interface**

The “Behaviour” interface and its sub-interfaces allow the agents to adopt behaviours already implemented inside the “Agent” class.

**Other classes and interfaces**

The package of classes provided several other utility classes aiming to simplify and accelerate MAS development. For example, a class allows the sending of messages in the Java event queue. These messages can be recovered with a standard AWT event management. Another class (a subclass of “Mailbox”) allows any
Java component to send messages to agents and to receive messages from them.

6. CONCLUSION AND FUTURE WORK

This paper is a logical extension of our recent research on MAS. Initially we studied MAS development methodologies. Then, we considered MAS development tools and environments that we evaluated and compared. This comparative evaluation was carried out in order to identify tools’ and environments’ strong points, as well as weak points, in the hope of determining what a complete development environment should look like. Finally, we undertook the development of a complete MAS development environment essentially meeting the set of requirements identified in our comparative study. DMAS Builder offers several major advantages compared with most other existing tools. Indeed, DMAS Builder offers:

- Graphic specification of all MAS components: subsystems, agents, knowledge bases, tasks, behaviours, communication modes between agents, etc.
- Packages of agent-oriented classes (about 200 classes) allowing agent-oriented development.
- Specifications validation.
- A complete source code generation (in Java) of all system components.
- An exportation mechanism allowing system development and execution independent of DMAS Builder.
- An archive mechanism allowing the creation of an executable archive for each sub-system (JVM)—this allows very simple distributed application deployment.
- Several other options not available on standard development environments (OO or AO), thus simplifying tool preparation and initialization, such as automatic search of the necessary programs (java, javaw, rmic, javac, jar and others), automatic “classpath” and “sourcepath” initialization and much more.
- The environment execution on various operating systems (Windows, Unix, Linux) without any modification.
- Sub-systems deployment (executable jar files) on different operating systems without any modification.
- The capability to develop MAS with OA classes and Java standards API.
- The capability to use an integrated help engine to access all OA and standards Java classes documentation specifications.

Everything related to our MAS development environment/tool in this paper is 100% implemented in Java. At the time of writing, we are conducting a testing phase in which many people participate, all being external to our research team. We expect to offer over the Web the first version of DMAS Builder in the next few months. This one will include the development environment, documentation and examples allowing an easier understanding of the tool’s numerous capabilities. We are very optimistic as to the future of such a tool: as far as we know, it is the first to offer such a package of services for MAS development and deployment.

We are planning to add a distributed debugging tool that will allow the fetching of information in real time on all system agents and other subsystem components. Also, the addition of security utilities, of other behaviours, communication and interaction protocols, and other DF Agent types, will also be part of the next version. Finally, we are planning to support mobile agents (i.e. that will be able to migrate from one JVM to another) in the near future.

ACKNOWLEDGEMENTS

For its financial support, we thank the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES


problèmes et simulation) JFIADSMA’02, Hermes Sciences, 2002, 111-123.


APPENDIX: Generic Architecture