Simulating Multi-Agent Systems using Logic Programming*

Wamberto Vasconcelos
Division of Informatics, University of Edinburgh
80 South Bridge, EH1 1HN, Edinburgh, UK
wvasconcelos@acm.org

Jordi Sabater
Institut d'Investigació en Intel.ligència Artificial
Campus UAB, 08193, Bellaterra, Catalunya, Spain
jsabater@iiia.csic.es

Abstract. In this paper we investigate three distinct approaches to the simulation of multi-agent systems: sequential, multi-processing and multi-threading. A standard inter-agent communication platform has been additionally connected to the multi-threading execution approach making it robust and generic. For each approach we present its advantages and drawbacks and elaborate on the tools we have built.
Keywords: Simulation, logic-based MASs.

1 Introduction

Important aspects of multi-agent systems (MAS henceforth) and distributed AI have been successfully investigated using logics (*e.g.* [16], [18], [20]). However, it is most desirable that there should exist means to simulate the intended behaviours of the agents comprising the MAS. This way, we can benefit from the modelling-simulation feedback loop – the *virtuous cycle* – in which a model is gradually improved through the study of the simulation of its behaviour [12].

In this paper we describe in three means to simulate logic-based MASs: sequential, multi-processing and multi-threading. Each approach has advantages and drawbacks. Ideally, one should be able to run the very same MAS using the different approaches with very little or no need for adaptation, thus exploiting their benefits. We have aimed at simplicity and minimal overhead using existing technologies for logic programming. The objective of this work is the development of tools that allow the controlled study of MASs.

This work regards logic-based MASs as all those employing some logical system to represent or model aspects of its behaviour. A logical system, in this context, is a formal language with a precisely defined semantics and, hopefully, a sound deductive/inferential procedure. Some examples of typical logic-based MASs are those incorporating BDI logics [19], temporal logics [11, 24] or other modal logics. One can prove interesting properties of MASs without having to resort to their simulation [4, 9]. However, important insights can be gained by analysing their simulated behaviour.

This paper concentrates on a particular paradigm, *i.e.* logic programming, for simulating MASs comprising collections of individual components. It focuses further on a specific logic programming language, Prolog [1, 21]. We have investigated technologies available in existing Prolog interpreters, in particular the distributed/parallel execution of programs and communication among processes, and how to harness them for simulating MASs. We have experimented with these technologies and developed simple tools to help people deploying them. We are interested here in those technologies that are readily and widely available in mainstream Prolog systems.

* Work developed within research grant **Sustainable Lifecycles in Information Ecosystems**, sponsored by the European Union, contract no. IST-1999-10948.

A particular distinction between *simulating* MASs and conventional parallel/concurrent *execution* of logic programs concerns the *granularity* of the parallelism. In the latter, a very fine level of parallelism is achieved in which individual commands of the same program are executed in parallel. In the former, a coarser degree of parallelism can be detected in which whole programs (comprising individual components of the MAS) are run in parallel. We have investigated existing technologies for parallel/concurrent execution of logic programs (fine grain) and how these can be employed in our simulations (coarse grain).

In the remainder of this section we introduce and justify some basic assumptions and present a simple notation. In Section 2 we explain the sequential simulation of MASs. In Section 3.1 we introduce our first approach to simulating MASs using parallel logic programming technologies: the agents are individual processes that communicate via a shared memory space. In Section 3.2 we present another way to simulate MASs using an alternative parallel logic programming technology: agents are independent threads that share a common knowledge-base. We exploit a means to make our simulations more flexible and generic in Section 4 and in Section 5 we survey some related work. Finally in Section 6 we draw conclusions and give directions of research.

1.1 Simulation of Logic-Based MASs

We are concerned with the simulation of logic-based MASs. We limit our attention to those logics whose deductive/inferential process can be precisely characterised as an effective procedure, that is, they can be programmed as, for instance, a Turing machine. This restriction is necessary since we want to simulate the deductions as an automatic process.

For any such logic $\mathcal{L}$ with a deduction/inference relation $\vdash_{\mathcal{L}}$ it is possible to encode, in a programming language of the designer's choice, a program $\Pi^{\mathcal{L}}$ which computes it. Logic programming, notably Prolog [1, 21], offers a number of advantages when programming deductive procedures. Being a higher-level and symbolic language, it allows for simple and elegant renditions of theorem provers (for instance, modal logics [17]). In the remainder of our presentation an agent $Ag_i^{\mathcal{L}}$ incorporating logic $\mathcal{L}$ will be represented by $Ag_i^{\mathcal{L}} = \langle \Pi^{\mathcal{L}}, \Sigma_i^{\mathcal{L}}, \boldsymbol{G} \rangle$ that is, a Prolog program $\Pi^{\mathcal{L}}$ (which implements $\vdash_{\mathcal{L}}$) will prove the sequence of goals $\boldsymbol{G}$ using formulae (axioms) $\Sigma_i^{\mathcal{L}}$. We can simplify our notation even further assuming, without loss of generality, that $\Sigma^{\mathcal{L}}$ is encoded as part of program $\Pi^{\mathcal{L}}$, and our agents can be represented as the pair $\langle \Pi^{\mathcal{L}}, \boldsymbol{G} \rangle$. Since it is not relevant for our discussion, we shall drop the $\mathcal{L}$ superscript.

There are fundamental differences between simulating and deploying MASs. When we are simulating MASs, we can experiment with simpler and idealised scenarios, thus allowing us to focus on whatever issues are important. For instance, we can model MASs in which loss of messages and malignant agents are not present and that all agents behave exactly as we intend. This contrasts sharply with building MASs for deployment, in which a plethora of important practical issues must be accounted for, for instance, messages that may get lost, malignant agents, computers and agents that stop working, and so on.

Without a simulation tool, researchers are faced with the substantial overhead of programming the infra-structure to support their experiments. The more researchers want their experiments to resemble actual distributed environments, the more effort they have to spend programming. Various tools and platforms have been proposed for building individual agents and multi-agent systems (see [22], pp. 25–27). However, these are essentially technologies for the deployment of (stable) agents. On the other hand, experiments with research work on intelligent/reasoning agents require more abstract and high-level tools. Simulation can be most useful as a preliminary step before deployment, when experiments can be performed under different controlled conditions and before a stable version of agents are available.

2 MAS as a Sequential Logic Program

A collection of agents implemented as logic programs can be executed sequentially, one complete program, *i.e.* an agent, at a time. However, since agents are most likely to interact by message passing, only a very limited class of agents can be executed in this fashion. Whenever an agent attempted to receive a message sent by another agent no further progress could be made because that message would never be available.

A more prospective approach for sequential simulation is to take turns choosing one goal from a program of our collection and execute it. The execution control would then choose and execute another goal from another program and so on. We describe in **Fig. 1** a simple and generic interpreter implementing this policy in Prolog. A MAS is represented as a list of agents. Each agent is a pair $[\Pi, \boldsymbol{G}]$ where Π is a logic program and $\boldsymbol{G}$ is an initial query. The behaviours of each agent are obtained when trying to prove whether $\boldsymbol{G}$ holds using Π. Predicate `simulation`/2 tries all possible choices of goals in our list of components until it finds one complete sequence of goals successfully executed (via `step`/4). The first clause of `simulation`/2 describes the termination condition: when the queries of all agents are `true` then simulation has come to an end. The second clause of `simulation`/2 defines the general approach: choose an `Agent` (a pair `[Pi,Gs]`) from the list comprising the MAS, execute one step of the deduction `Pi` $\vdash$ `Gs`, appropriately update the list of components with the new program and query, and recursively do the same. A complete sequence of steps is found when only `true` queries are left, as stated in the first clause of `simulation`/2, defining the termination condition. Predicate `step`/4 executes one single step of SLDNF-resolution [1]: the first clause deals with one goal of a multi-goal query, the second clause deals with built-ins and the third clause handles the unification of a goal with the head of a clause. We assume that agents may change their set of axioms or deduction rules, hence the provision for new programs that may be obtained in the execution of `step`/4.

```
simulation(MAS,_):-
  all_members([_,true],MAS).
simulation(MAS,Msgs):-
  choose(Agent,MAS),
  step(Agent,Msgs,NewAgent,NewMsgs),
  replace(Agent,NewAgent,MAS,NewMAS),
  simulation(NewMAS,NewMsgs).

step([Pi,(G,Gs)],Msgs,[NewPi,NewGs],NewMsgs):-
  step([Pi,G],Msgs,[MoreG,NewPi],NewMsgs),
  append(MoreG,Gs,NewGs).
step([Pi,G],Msgs,[NewPi,true],NewMsgs):-
  built_in(G),
  call(G,Pi,Msgs,NewPi,NewMsgs).
step([Pi,G],Msgs,[Pi,Gs],Msgs):-
  copy(Pi,FreshPi),
  member((G:-Gs),FreshPi).
```

Fig. 1: Sequential Simulator for MASs

Predicate `built-in`/1 which appears in the second clause of `step`/4 checks whether `G` is a built-in goal. Message-passing predicates are examples of built-ins:

their meaning is defined by the `call/5` predicate which updates the current messages. We show in **Fig. 2** the definition of message-passing predicates required for our sequential simulator: `send/3` appends a message `M` to the list `Msgs` of current messages; `receive/3` deletes message `M` (if it is found) from the current messages `Msgs`. Predicates which alter the program are also built-ins and hence the need to have the programs (the old and the new versions) as a parameter of `call/5`. Built-ins are goals that are executed completely by `call/5` – their outcome is always a `true` subgoal (and the appropriate changes to the program and/or the messages).

```
send(M,Msgs,[M|Msgs]).
receive(M,Msgs,NewMsgs):-
  member(M,Msgs),
  delete(Msgs,M,NewMsgs).
```

Fig. 2: Message-Passing Predicates

During the sequential simulation message passing has to be implemented in such a way that it allows backtracking to undo the effects of a message being sent or received. In our interpreter above all messages are stored in a list; messages sent are included in this list and a message received is removed from it. Upon backtracking the appropriate reverse effect takes place. The complete sequence of all messages exchanged can be recorded in `Msgs`, if those messages received are not removed, but simply marked as such.

Each complete execution obtained according to this policy consists on an *interleaving* of the independent executions of each program. Backtracking is possible, that is, whenever the execution reaches a point where no further progress can be made (for instance, because a program expects a message that was not sent) the control can go back to the last point where a goal was picked up and try some other goal instead. This guarantees we can find one possible "joint" execution of our collection of programs, or one possible interleaving of all independent deductions. The interpreter above can be augmented to appropriately record the interleavings it finds. This exhaustive search also stops if there is no possible joint execution. The search space for this problem is clearly exponential since all possible interleavings are attempted.

In the sequential approach it is possible to repeat one same simulation of a MAS over and again. This is possible because all the events that happened during a complete simulation can be re-enacted – there are no influences outside the computational model that may affect the simulation. This ability to replicate one same simulation is sometimes very useful to help us understand the workings of a MAS. As we shall see below, as the simulation gets more realistic its replication gets harder to obtain due to influences outside the computational model which play an essential part in defining the ordering of events.

One drawback with sequential simulation happens when the workings of each individual agent of a collection requires time t: a collection with n agents would require time $n \times t$, which, depending on n and t, can be prohibitively long. Another scenario for which sequential simulation is not suitable arises when the programs are sensitive to time, that is, their behaviour is dependent on how long it takes for some event to happen. An artificial account of time would have to be employed in such circumstances.

3 Agents Executed as Parallel Logic Programs

An alternative to the sequential execution of our collection of logic programs as a single program is the parallel execution of each of these programs [14].

We have investigated technologies available in existing Prolog interpreters, in particular the distributed/parallel execution of programs and communication among processes, and how to harness them for executing a collection of agents. We have experimented with these technologies and developed simple tools to help people deploying them.

We have restricted our attention to two different approaches to the concurrent execution of Prolog programs. The first approach, explained in Section 3.1 below, employs the Linda tuple space concept introduced in the early eighties [6]: each agent is an independent Prolog process that communicates via a tuple space. The second approach, explained in Section 3.2 below, uses multi-threading [7]: agents share the same knowledge base but each of them is executed by its own independent thread within one same Prolog session. Although there exist other alternatives for the parallel execution of logic languages [14], we are interested here in those technologies that are readily and widely available in mainstream Prolog systems. We have implemented platforms incorporating each approach and we shall briefly explain them, listing their advantages and drawbacks.

3.1 Agents as Multi-Processes

Our agents can be run as independent process which communicate by some means. We have explored this approach using the Linda process communication mechanism available in SICStus Prolog [15]. Linda [6] is a concept for process communication. It consists of an area, the *tuple space*, where data from a number of different processes can be stored and a few basic operations, write, read and delete, which can be performed on this area. The Linda concept has proven very successful with its simplicity and flexibility. SICStus Prolog [15] incorporates Linda's ideas and offers a repertoire of built-in predicates with which one can write programs that run independently and exchange data.

There are two kinds of Linda programs (or *processes*), the *server* and the *client*. The server process manages the tuple space onto which the client processes may write and from which they may read and remove terms. The server is a very simple Prolog program that simply invokes a built-in predicate to define the tuple space and a *socket* to enable the communication with client processes.

Client processes cannot communicate to each other directly, instead of that, they have to use the tuple space. The access to the tuple space is done through the Server that makes sure that requests from client processes for writing, reading and deleting terms onto/from the tuple space are appropriately performed. A client process can only be connected to one server at a time, but it can be connected to one server at one point, then disconnect from that server and get connected again to another server.

Request for messages are delivered following a precise discipline: the first message that was sent will be delivered first, when requested. However, messages can "leapfrog" one another: if a term was sent and stored as the n-th message and a client process tries to specifically read it, then it will leapfrog the $n-1$ preceding messages and be delivered first. This proposal can be graphically depicted as **Fig. 3**. Initially the *server* process, shown in

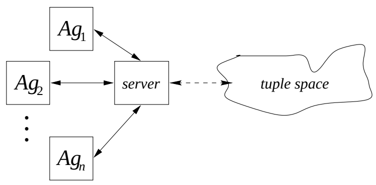

Fig. 3: MASs via a Tuple Space

the middle of the diagram is started. The dotted arrows represent the management of the tuple space done by the server process. After the server has been started the individual client process $Ag_1, \ldots, Ag_n$ shown above can be connected and then use the tuple space. Each client process is an agent, a self-contained Prolog program that may run concurrently, if they are all started up in a single machine with one processor only, or in parallel, if a number of distinct processors are available in one machine or if they are run in different machines.

The sequence of steps depicted above required to start up a set of SICStus Prolog Linda client-server processes can be rather hindering when the experiments are big or they have to be repeated several times. To solve this problem we have developed a "launching pad" with which a number of components can be automatically started up. We show in **Fig. 4** a diagrammatic representation for our tool. A MAS is a list of agents which will be input to **LPadL** (**L**aunching **Pad** for **L**inda) a tool we have developed. Each agent is started up as an independent Prolog process Ag_i, equated here with a process, and represented in the diagram above by the dotted arrows. A preliminary sequence of steps is required to "connect" each process to a Linda *server* process (central *Sv* box in diagram above). This server process is responsible for managing the tuple space and can be automatically generated and started up as a background job and killed at the end of the execution. The management of the tuple space is done by Prolog itself. A suitable definition for message passing predicates is provided, harnessing the technicalities of Linda message-passing.

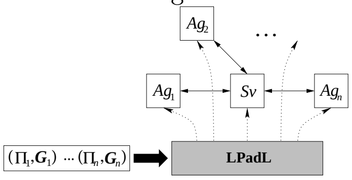

Fig. 4: Simulator **LPadL**

Each pair $\langle \Pi, \boldsymbol{G} \rangle$ will give rise to a process Ag_i. However, some initial steps are required before a process can actually interact with the tuple space. Designers need not worry about this: such a sequence of steps is done automatically by **LPadL** at the onset of the process life. At the end of the execution of each process a procedure to close the connection has to be followed. Again, this has been automated in **LPadL**. We show in **Fig. 5** an example of Linda-based agents. Each agent Ag_i consists of a self-contained Prolog program: to the core components Π_i and $\boldsymbol{G}_i$, **LPadL** inserts the definition of all built-in predicates (such as message-passing primitives, and routines for initialisation and termination of the connection with server) as well as the top-level call that will be made when the agent is started up. The double arrows represent the communication channels, established upon each process `initialise`/0 execution and terminated after the agent runs `terminate`/0. Each agent is thus a stand-alone Prolog program that has all the usual features of an interpreter plus the ability to access the tuple space using built-in communication primitives.

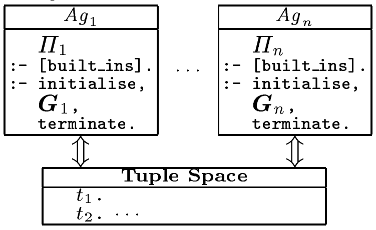

Fig. 5: Agents and Tuple Space

The processes can run in one machine or in many machines – Linda allows processes running in different machines (even geographically apart) to communicate. This feature is important for systems with very large numbers of components: because each process requires a full SICStus Prolog session running, the resources of a machine can get quickly depleted. Large populations can thus be distributed among different machines. Another issue concerns the effect of backtracking on communication primitives: once a message is written onto or read/retrieved from the tuple space the execution control will not undo the action, thus behaving like standard I/O built-ins like `read/1` and `write/1`, that is, skipping over them [15]. A message `M` is sent via SICStus Linda client built-in `out/1` which writes `M` onto the tuple space; a message `M` is received via built-in `in/1` which removes `M` from the tuple space or blocks the execution if `M` is not available.

The tuple space approach has a number of positive features. Since processes are independent self-contained programs, agents sensitive to time can be successfully simulated. Another advantage of this approach is that processes can run in different machines geographically apart, thus allowing for large populations of agents to be executed simultaneously, distributing among many machines the quick depletion of resources caused by having a Prolog session for each agent. Having agents running in distinct machines also confers on the simulations a more realistic aspect since a true parallel/distributed scenario arises. An additional feature of this proposal is that it employs simple and stable technology (from the early 80's) seamlessly incorporated into Prolog. Another useful feature of the multi-process approach is that debugging/tracing of individual agents is possible. That is, we can trace the sequence of steps one program executes during an experiment.

A drawback of this approach lies in its lack of any global account of the parallel execution of its processes. Hence it is not possible to directly obtain the necessary information for a complete representation of the joint execution of processes. There are, however, alternatives to circumvent this shortcoming. For instance, **LPadL** can record all messages in transit – a supervisor process is automatically started at the onset of a simulation and intercepts and stores messages in the order they have been sent. This idea can be extended: we can program interpreters for each process that have their execution monitored by means of message passing. A more sophisticated supervisor process would send messages to each process asking for a report on its execution at a certain point and the processes would send this report (it could be, for instance, a trace of the program up to a certain point) back (also via message passing) to the supervisor process which would collate them together. Connecting agents implemented in languages other than Prolog to a SICStus-Linda session although possible is not straightforward. A stringent issue is that SICStus Prolog [15] is essential for this proposal and since it is not freely available, this proposal cannot be easily replicated by other people.

3.2 Agents as Multi-Threads

An alternative to the multi-process approach explained above is the execution of our collection of components as *multi-threads*. In this proposal, each component is an execution thread. A thread corresponds to an independent program interpreter/evaluator able to execute a goal to completion in a local environment, that is, unaffected by other threads [3]. This approach for concurrent execution of logic programs is explained in details in [7] and has been incorporated to the logic programming system CIAO [3].

We have developed another tool, **LPadT** (**L**aunching **Pad** for **T**hreads), which enables the execution of our components as threads. Each agent $\langle \Pi_i, \boldsymbol{G} \rangle$ is started up as a thread but they all share one same area, the knowledge base *KB*, as shown in **Fig. 6**. **LPadT** reads in all pairs of components but differently from **LPadL** a single program is built as *KB*, joining all the programs $\Pi_1, \ldots, \Pi_n$ together in one big program globally available to all threads. Each one of the queries $\boldsymbol{G}_1, \ldots, \boldsymbol{G}_n$ becomes a thread. Message-passing is performed by means of writing and reading terms onto/from the shared knowledge base.

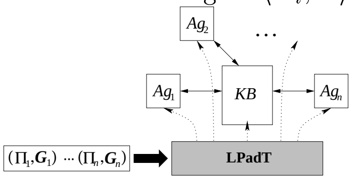

Fig. 6: Simulator **LPadT**

To make sure that an agent will not employ the code of another agent by accident – this is very likely when the MAS consists of dozens of similar programs all sitting together in one large knowledge base – we carry out a simple meaning-preserving program transformation. For each pair $\langle \Pi_i, \boldsymbol{G}_i \rangle$ we add an extra argument to all user-defined literals, obtaining the transformed pair $\langle \Pi'_i, \boldsymbol{G}'_i \rangle$. This extra argument is an identifying tag which ensures that agent i will only access its own code in the shared *KB*. A diagrammatic representation of a running multi-thread session is shown in **Fig. 7**: the transformed programs Π'_i are joined together and each transformed query $\boldsymbol{G}'_i$ is executed as an independent thread. The definition of built-ins, such as message-passing primitives, has to be included in the shared *KB*. The double arrows represent the fact that all agents may at any moment consult and/or alter the shared *KB*.

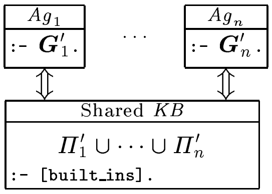

Fig. 7: Agents + Shared *KB*

The main difference between the multi-thread and the multi-process approaches is that in the former the programs are shared by and visible to all agents, whereas in the latter each component has its own private knowledge base. Another distinction concerns the fact that although each thread is technically a process, it requires far less resources from the machine running it [8] since it does not require a full Prolog session. Instead, a lightweight version of a Prolog engine is created for each thread [7].

Many more agents can be started up simultaneously in one machine using the multi-thread approach, since a thread takes up less resources than a complete Prolog session. Another advantage is that, as we explain below, it is possible to connect CIAO with a robust intra-process communication platform. The communication primitives are simple and their behaviour is intuitive.

One initial difficulty with our multi-threading approach concerns debugging. It is not possible to trace a thread: there are no "handles" that programmers can use to trace and debug a thread. Additionally, since a unique global program is shared by all threads, designers must ensure that undesirable interferences do not arise. If an agent changes portions of the shared program, this may accidentally affect the behaviour of other agents. Our adopted solution, *i.e.* to add a tag to clauses to uniquely identify them, has its limitations – if an agent adds clauses to the shared program, then these clauses must have the same tag that has been assigned to other clauses of the agent.

We have employed the logic programming system CIAO [3] to investigate the multi-thread approach to the execution of agents. This is programming environment freely distributed, which means our results can be readily replicated by other researchers. However, CIAO is still under development and some of its features are not yet stable. This is, unfortunately, the case with multi-threading, although the prospects are that we shall soon have a stable and well-defined repertoire of built-ins for multi-threading. It is not likely though that changes in the CIAO features will render our multi-thread approach to agents execution invalid.

It would be useful to be able to run agents in different machines thus confering more realism on our experiments and possibly saving time. Although it is possible to connect different CIAO sessions via sockets, this facility is not readily available and some fairly low-level programming would be made necessary. As we explain below, we have solved this issue – our solution also enables agents programmed in any language to communicate.

4 JADE: a Generic Communication Platform

In our two previous proposals, there are limits on the kinds of communication that can take place. In the tuple space approach, for instance, there can only be one tuple space shared among the agents at a time. Although there could be many different tuple spaces, an agent can only be connected to one of them at a time [15]. The messages must conform to the syntax of Prolog terms and, a more stringent constraint, only SICStus Prolog processes can easily and readily enjoy the communication platform.

The communication among multi-threads cannot be considered technically a platform, since everything happens within one single Prolog session (with many threads). However, we have augmented the multi-thread alternative with a robust and generic communication platform, explained below. This extension allows for agents running as threads within different sessions to communicate, but it also allows agents in *any* language to exchange messages using this platform.

JADE (Java Agent DEvelopment) [2] is a FIPA compliant agent platform implemented in Java. FIPA (Foundation for Intelligent Physical Agents) [10] is a non-profit association that operates through the open international collaboration of member organisations, which are companies and universities active in the agent field. FIPA's purpose is to promote the success of emerging agent-based applications, services and equipment. This goal is pursued by making available internationally agreed specifications that maximise interoperability across agent-based applications, services and equipment. We briefly describe below those parts of the specification that are more relevant for our purposes.

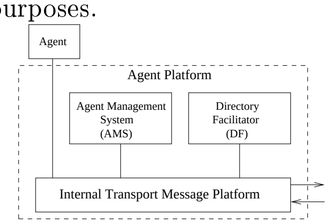

Fig. 8: Management Model

The *Agent Management* part of FIPA specification provides a normative framework within which FIPA compliant agents can exist, operate and be managed. It defines an agent platform reference model as shown in **Fig. 8**. The agent platform has several elements. The directory facilitator (DF) is a mandatory, normative agent that provides "yellow pages" services to other agents. At least one DF must be resident on each agent platform. However an agent platform may support any number of DF's. The agent management system (AMS) is also a mandatory agent of the agent platform which is responsible for managing the activities of an agent platform. These responsibilities include creation of agents, deletion of agents, overseeing the migration of agents to and from platforms, and so on. The AMS also maintains an index of all the agents resident on a platform. Finally, the internal transport message platform provides the communication capabilities between the agents connected to the platform and also with other FIPA platforms.

The *Agent Communication Language* (ACL) part lays out underlying principles and detailed requirements for agents to be able to communicate with each other using messages representing communicative acts. **Fig. 9** shows an example of a message with the main structural elements of a FIPA-ACL message. FIPA defines a set of communicative acts catalogued by categories. For example, the category "Action Performing" contains performatives like `request`, `cancel` or `agree` and the category "Negotiation" performatives like `propose` or `accept-proposal`. Each performative is described in the specification in a formal way to avoid ambiguity. ACL messages also contain a set of one or more parameters. The only parameter that is mandatory in all messages is the `:receiver` parameter, so that the message delivery service can correctly deliver the message. The `:content` of a message refers to whatever the communicative act applies to. In general, the content can be encoded in any language, and that language can be denoted by the `:language` parameter. It is the responsibility of the agents in a dialogue to ensure that they are using a mutually comprehensive content language.

```
(request
  :sender agent-a
  :receiver agent-b
  :content
    (give(agent-b,agent-a,nail)))
```

Fig. 9: Sample ACL Message

By employing JADE as the underlying communication platform, we inherit the benefits of FIPA's standardisation efforts and respect from the mainstream (and not necessarily logic programming oriented) agent research community. Additionally, the use of JADE as an underlying platform ensures a high degree of compatibility with other FIPA compliant agents build by third party programmers. JADE also offers a repertoire of useful tools to visualise and monitor the status of its agents.

The JADE/Prolog combination gets the benefits of the multi-thread approach and solves some of its drawbacks. Agents running in machines geographically apart in distinct CIAO sessions can communicate with each other via JADE. Agents which have been developed in different languages can also be connected to JADE thus giving flexibility to our laboratory – if, in the future, we decide to change the language in which our agents are programmed, we can still employ the same laboratory. Finally, since both JADE and CIAO are freely distributed, our experiments can be replicated by others. One difficulty with this combined alternative is that the connection of non-Java agents, *i.e.*, a Prolog thread, is not trivial and at the moment is done in an indirect manner, via an intermediate agent.

5 Related Work

Swarm [13] is a general-purpose package for simulating concurrent, distributed artificial worlds. The core of Swarm is an object-oriented framework for defining the behaviour of agents and other objects that interact during a simulation. Objects of Swarm are organised into a hierarchy of sub-swarms that may be allocated across multiple processor and communications boundaries. The individual agents are object-oriented programs, *i.e.* written in Objective C or Java. Although Swarm is popular and offers many useful features like tools for visualising experiments and collecting data, writing theorem provers and deductive procedures in non-symbolic languages can be very time-consuming.

Any Prolog interpreter with inter-process communication possibilities can, in principle, be used to simulate MASs. In particular, those interpreters offering Java front-ends such as JINNI [23] and InterProlog [5], allows for agents programmed in Prolog to be enveloped in Java and thus communicate with other programs (in Java, Prolog or other language with a Java front-end). In this scenario, our MASs could have agents programmed in many different languages. However, this mixture can have rather unexpected outcomes: when problems appear (they are inherent in distributed systems) during simulations it may be difficult or even impossible to pinpoint the causes. In this Prolog/Java association there are layers which solely concern Java. When things go wrong their cause may be in these layers thus forcing users of languages other than Java to become acquainted (or more likely, experts) with Java programming.

6 Conclusions

In this paper we have described three approaches to simulate logic-based MASs using available technologies for Prolog. Our simulation approaches provide three main facilities: *i)* an automatic means to start and terminate experiments that may involve large numbers of individual agents; *ii)* a communication means which enables interaction among agents via (simulation of) message-passing; and *iii)* different ways to monitor and record the behaviour of agents.

The sequential approach allows for the exploration of different interleavings of the executions of goals of different components. This approach is not useful when components take into account the time elapsed between events. To circumvent this difficulty, we have also investigated different approaches to the parallel execution of our agents. In one proposal the components of a MAS are equated with a process, that is, an independent Prolog session; these components communicate using a shared tuple space. In our other proposal, the components are threads and they all share one same area where their code is stored. Both parallel approaches have advantages and drawbacks. We have also described how to augment Prolog with a standard communication platform, JADE, allowing our experiments to scale-up in number and diversity of agents. Being able to execute heterogeneous agents implemented in different languages is a desirable feature when simulating MASs since we may employ ready-made agents written in languages other than Prolog.

Ideally, a MAS should be executable in any of our proposals. The degree of realism of each proposal is inversely proportional to the control it offers for the execution, and we might want to experiment with different platforms, considering their advantages and drawbacks. The same experiment specification should be amenable to be executed in each of our proposals, with little or no human intervention to adapt it. Currently, we are implementing a graphic tool to display the results obtained by experiments running over the JADE based platform.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.
2. F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA-compliant Agent Framework. Technical report, CSELT S.p.A, 1999.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System Reference Manual. Tech. report, Facultad de Informática, Universidad Politécnica de Madrid (UPM), Nov. 2000.
4. H.-D. Burkhard. Liveness and Fairness Properties in Multi-Agent Systems. In *Proc. IJCAI-93*, pages 325–330, Chambéry, France, 1993.
5. M. Calejo. InterProlog: A Simple yet Powerful Java/Prolog Interface. *COMPULOG Newsletter*, Dec. 1998.
6. N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4), Apr. 1989.
7. M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *Proc. of the 1999 Int'l Conf. on Logic Programming*, pages 320–334, Cambridge, MA, U.S.A., Nov. 1999. MIT Press.
8. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, U.K., 2nd edition, 1994.
9. R. Fagin, J. Y. Halpern, and M. Y. Vardi. What Can Machines Know? On the Properties of Knowledge in Distributed Systems. *JACM*, 39(2):328–376, 1992.
10. FIPA. The Foundation for Intelligent Physical Agents. `http://www.fipa.org`.
11. M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Proc. 12th Int'l Workshop on Distributed A I*, pages 131–142, Hidden Valley, PA, May 1993.
12. P. A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.
13. P. Johnson and A. Lancaster. SWARM User Guide. SWARM Development Group, `http://www.swarm.org`, 2001.
14. J. C. d. Kergommeaux and P. Codognet. Parallel Logic Program Systems. *ACM Computing Surveys*, 26(3):298–336, Sept. 1994.
15. I. S. Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, Feb. 2000. `http://www.sics.se`.
16. B. v. Linder. *Modal Logics for Rational Agents*. PhD thesis, Utrecht University, The Netherlands, 1996.
17. A. Nonnengart. First-Order Modal Logic Theorem Proving and Functional Simulation. In *Proc. IJCAI-93*, Chambéry, France, 1993.
18. S. Parsons, C. Sierra, and N. R. Jennings. Agents that Reason and Negotiate by Arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.
19. A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. 1st Int'l Conf. on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.
20. J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. Engineering Executable Agents using Multi-Context Systems. *Journal of Logic and Computation*, 2001. to appear.
21. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.
22. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press/AAAI Press, Cambridge, MA, 2000.
23. P. Tarau. JINNI: A Prolog Interpreter in Java for Mobile Agent Scripting and Internet Programming. Available at `http://www.binnetcorp.com/Jinni`, July 1999.
24. M. Wooldridge. Temporal Belief Logics for Modelling Distributed Artificial Intelligence Systems. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons, 1995.