A Prolog Simulation Platform for Logic-Based Multi-Agent Systems

Wamberto W. Vasconcelos*
School of Artificial Intelligence, Division of Informatics, The University of Edinburgh
80 South Bridge, EH1 1HN Edinburgh, Scotland, UK
wvasconcelos@acm.org

Abstract. Logic, in its broadest sense and diversity, has been successfully employed to investigate important issues of multi-agent systems. However, it is most desirable that we offer means to simulate the intended behaviours of the systems modelled via such logical approaches. In this paper we present an extensible simulation platform in Prolog for logic-based multi-agent systems.

1 Introduction

Important aspects of multi-agent systems (MAS henceforth) and distributed AI have been successfully investigated using logics (*e.g.* [15], [17], [21], [23] and [28]). 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 in which a model is gradually improved through the study of the simulation of its behaviour [11].

In this paper we describe in detail one means to simulate logic-based 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 [22], temporal logics [7, 10, 27] or other modal logics. One can prove interesting properties of MASs without having to resort to their simulation [3, 8]. 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, 24]. 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. In this work, we restrict our attention to one approach, *viz.* concurrent processes sharing a tuple space. 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.

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

* Sponsored by the European Union, contract IST-1999-10208, research grant **Sustainable Lifecycles in Information Ecosystems** (SLIE).

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). We have supplied enough details so as to allow the reproduction of our experiments and their customisation to more specific needs.

In the remainder of this section we introduce and justify some basic assumptions and present a simple notation. In Section 2 we present our execution approach and its details. In Section 3 we compare our approach with related work. In Section 4 we draw conclusions and give directions for future work.

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 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, 24], 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 [18, 19]). In the rest of our presentation an agent $Ag_i^{\mathcal{L}}$ incorporating logic $\mathcal{L}$ will be represented by $Ag_i^{\mathcal{L}} = \Pi_{\mathcal{L}}(\Sigma_i^{\mathcal{L}}, \boldsymbol{G})$ that is, a Prolog program $\Pi_{\mathcal{L}}$ (which implements $\vdash_{\mathcal{L}}$) will prove the sequence of goals $\boldsymbol{G}$ using formulae $\Sigma_i^{\mathcal{L}}$.

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 [25], 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 A Simulation Platform for Logic-Based MASs

Any MAS simulation platform must provide at least these three features:

1. *Individuality of components* – the overall behaviour/properties of MASs must arise from the interactions among their individual agents. It is thus essential that individual agents correspond to executing units with their own control (see item 3 below)
2. *Message-passing facilities* – message-passing is the only means of communication among agents. Basic services for sending and receiving messages must hence be offered to MASs designers.
3. *Distributed/concurrent control of execution* – to ensure the appropriate simulation of time-sensitive agents, *i.e.*, those agents whose behaviour depends on time, an appropriately distributed or concurrent execution mechanism must be employed.

We have adopted a "lightweight" approach though nonetheless all these features are catered for. Only the bare essentials are presented here, but with enough details so as to allow their reproduction. Ideally, a means to visualise, log data and trace the simulation should be part of a user-friendly simulation platform but we do not exploit these issues in this paper.

Our platform is based on Linda [5, 6], an inter-process communication proposal. Agents are individual Prolog processes that run independently and communicate via a shared memory space. This is a typical blackboard architecture studied in depth in distributed AI circles. In the remaining of this section we describe our platform in its details.

2.1 Linda: Processes and a Tuple Space

Linda [5, 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) can be performed on this area. The Linda concept has proven very successful with its simplicity and flexibility. SICStus Prolog [16] incorporates a Linda library and offers a repertoire of built-in predicates with which one can write programs that run independently and exchange data. We have investigated the adequacy of this technology for simulating MASs.

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. We show in **Fig. 1** a means to define a SICStus Prolog Linda server. The first line informs SICStus that the `linda/server` library should be loaded. The second line starts up the server process by means of the built-in predicate `linda/1` from the `linda/server` library. The syntax and semantics of this built-in is `linda(Address-Goals)` where `Address` is of the form `Host:Port` and `Goals`

```
:-use_module(library('linda/server')).
:-linda((Host:Port)-
        (tell('messages.addr'),
         write(Host),write(':'),
         write(Port),write('.'),
         told)).
```

Fig. 1: Linda Server `server.pl`

must be instantiated to a sequence of goals [16]. When this predicate is executed, it starts up the server, sets up a socket at `Address` and runs `Goals`. Our program thus establishes a communication socket `Host:Port` through which client processes can communicate with the server. Since the socket information ought to be given to the client processes, it is recorded as a term in the file `messages.addr`.

It is essential that a client process has access to the socket information so that a communication channel be established between itself and the server. Client processes cannot communicate to each other directly – producer processes writes terms onto the tuple space defined by the server and consumer processes read or delete them. The server 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.

The server process is always as simple as the one above. No built-in predicates are offered to manage the message area[1]. Client processes, on the other hand, can have access to the message area but only by means of built-in predicates, explained below. 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 retrieved first.

2.2 Messages in the Tuple Space

We now describe how client processes can use the tuple space to send and receive messages. We equate the recording of a term onto the tuple space with a message being sent; the retrieval of a term (by reading or deleting it) from the tuple space is equated with the receipt of a message. Messages, hence, are not delivered to a private area of the recipient process. Rather, they are recorded in the tuple space awaiting collection and all processes are able to read/delete them.

Our Linda client processes are the actual components of our MASs. The server process is not really part of our MAS, it being required due to the way SICStus Prolog incorporates Linda techniques. However, it should be pointed out that the server process does not explicitly manage the tuple space, that is, it is not necessary that we program the server. Request from client processes to read or delete from and to write onto the tuple space will be appropriately dealt with by the server without us having to worry about it. Clients, however, have to be programmed as to when and how terms are recorded or retrieved from the tuple space.

Basic services have to be provided to each individual process/agent. These services concern the establishment of the connection (*i.e.* the socket) to communicate with the server process, the possible termination of the connection, the receipt of messages and sending them. We show in **Fig. 2** a simple way of implementing these services (we have added numbers to the left of relevant lines so as to enable easy reference to specific parts of the program).

[1] We can include a message-handling agent as a client process which provides for more sophisticated message manipulation.

```
 0 :-use_module(library(linda/client)).

 1 initialiseConnection:-
 2   see('messages.addr'),
 3   read(Host:Port),
 4   seen,
 5   linda_client(Host:Port).

 6 terminateConnection:-
 7   close_client.

 8 readMsg(msg(Sdr,Ag,M)):-
 9   agentId(Ag),
10   in_noblock(msg(Sdr,Ag,M)).
11 readMsg(nil).

12 sendMsgs([]).
13 sendMsgs([M|Ms]):-
14   out(M),
15   sendMsgs(Ms).
```

Fig. 2: Communication Predicates

Line 0 of the program ensures that the appropriate library is loaded. Predicate `initialiseConnection/0` (lines 1–5) establishes the communication channel between the agent and the server process: the file `messages.addr`, recorded previously by our `server.pl` program (the server should be started first) is read in and then closed. Its contents, of the form `Host:Port`, are used as the argument of built-in `linda_ client/1` which is ultimately responsible for establishing the actual connection between the client process and the server. Predicate `initialiseConnection/0` should be invoked once at the very beginning of the client execution. Predicate `terminateConnection/0` (lines 6 and 7), on the other hand, closes the connection with the Linda server by means of built-in `close_client/0`; it should only be invoked at the very end of the agent's execution or when the client process does not need to interact with other processes anymore.

Predicate `readMsg/1` (lines 8–11) checks if there is a message for `Ag`, identified by `agentId/1`. It does so by invoking the built-in predicate `in_noblock/1` whose syntax is `in_noblock(Tuple)` and it succeeds if there is a term in the tuple space which matches/unifies with the pattern supplied in the call. To make our discussion concrete, we have employed the construct `msg(Sdr,Ag,M)` as our message tuples – this format is a design choice and should be consistently used by all agents. Variable `Ag` gets instantiated by `agentId/1` becoming the (unique) identification of the agent and hence the second argument of term `msg` (the addressee) gets the same value. Predicate `in_noblock/1` fails if there is not such a message. A waiting period can be set up in advance so that its failure is delayed: the message is waited for for some time and if it is not made available in the tuple space, the predicate fails. The timeout period can be set up via the Linda (client) built-in predicate `linda_timeout(OldTimeOut,NewTimeOut)`. If no such message is available within the set period, then the first clause of `checkInMsg/1` fails and upon backtracking the second clause will be employed to assign `nil` to the message received. Finally, predicate `sendMsgs/1` (lines 12–15) receives a list of messages prepared in advance in a suitable format and sends each of them to the server (second clause) via predicate `out/1` (line 14) until there are no more messages to send (first clause).

Each client process is a self-contained SICStus Prolog program, and it runs in its own Prolog session. The source code for each client must be stored in a file which is then loaded and executed. SICStus Prolog offers a repertoire of built-in predicates that agents may employ to read/delete from and write onto the tuple space. They are [16]:

- `out(Term)` – stores `Term`, a Prolog term, in the tuple space.
- `in(Term)` – removes `Term` from the tuple space if it is there. If not, the predicate blocks until `Term` is available, that is, another process performs an `out/1` with the appropriate term.

- `in_noblock(Term)` – removes `Term` from the tuple space if it is there. If not, the predicate fails. This predicate can fail due to a timeout.
- `in(TermList,Term)` – as `in/1` but succeeds when either of the terms in TermList is available. Term is unified with the fetched tuple. If that unification fails, the term is not reinserted in the tuple space.
- `rd(Term)` – succeeds if `Term` is available in the tuple space, suspends otherwise until it is available. Contrasting with `in/1`, the `rd/1` predicate does not remove `Term`, that is, it remains in the tuple space.
- `rd_noblock(Term)` – succeeds if `Term` is available in the tuple-space, fails otherwise. This predicate can fail due to a timeout.
- `rd(TermList,Term)` – as `in/2` but does not remove any term.
- `bagof_rd_noblock(Template,Term,Bag)` – `Bag` is the list of all instances of `Template` such that `Term` exists in the tuple space. The behaviour of variables in `Term` and `Template` is as in `bagof/3`. The variables could be existentially quantified with `^/2` as in `bagof/3`. The operation is performed as an atomic operation. This predicate can fail due to a timeout.

2.3 Simulating MASs via a Tuple Space

We now explain how to harness the technicalities of SICStus Prolog Linda client-server processes to simulate MASs. Our proposal can be graphically depicted as **Fig. 3**. Initially the *server* process, shown in the middle of the diagram is started. The dotted arrows represent the management of the tuple space done by the server process. We may want to hide the server process: rather than having it running in the foreground, taking up a window, we may wish to make our client-server interaction "cleaner" and have our server process running in the background. If we store the server program above in the file `server.pl`, we can get it running in the background via the UNIX command "`echo "[server]." | sicstus > /dev/null &`" which invokes SICStus and loads in program `server.pl`, diverting any outcoming system messages to the file `/dev/null`, the standard Unix dump file. After this command is executed, the SICStus server process will stay running permanently in the background – killing our background server process would require one to know the process identification number and to be appropriately authorised to do so. Our tool, explained below, however takes care of this, keeping track of processes id numbers and killing them at the user's request.

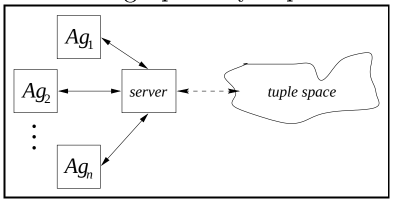

Fig. 3: MASs via a Tuple Space

After the server has been started, the individual client processes $Ag_1, \ldots, Ag_n$ shown above can be connected and then use the tuple space. As discussed previously, 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. They may also run in parallel, if a number of distinct processors are available in one machine or if they are run in different machines. Each agent must have the communication routines shown in **Fig. 2** and they should all invoke the predicate `initialiseConnection/0` at the very onset. Each client process runs independently and it may synchronise with other processes via predicates `in/1` and out/1 and their variants. Ideally, each client process should run quietly in the background, avoiding cluttering up the machine(s) with windows.

In our proposal, MASs consist of a set of components running as Linda client Prolog processes. MASs are simulated by starting up the server, then each of its component in turn. Each component should be designed bearing in mind how messages are sent, how they are stored and what should be done in order to receive them. In special it should be noticed that backtracking over built-in predicates in/1 and out/1 does not have the expected effect of "undoing" the deed: a term shall remain stored in the tuple space even though the execution backtracks over the out/1 predicate that caused the term to be stored.

Each MAS component must be stored in a separate file. The files should be loaded and appropriate initial queries posed. We can automate the start-up of a set of agents: if we are given a list with the names of the files (or the contents of each agent) then we can start each of them as background processes (thus saving on unnecessary open windows). We can incorporate standards to the design of individial components/agents thus allowing for their homogeneous processing. **Fig. 4** shows a partial standardised definition of a component. Our adopted design standardises the topmost predicate main/0 (lines 1–4): it initialises the connection between the client process and the server (initialiseConnection/0, line 2, as defined in **Fig. 2**), then it invokes the agent's main predicate mainAgent/0 (line 3) and finally it terminates the connection (terminateConnection/0, line 4, as defined above). Predicate mainAgent/0 (line 5) is responsible for the agent's actual behaviour and hence it is not defined in our template. At the very end of the file, as the last line n, a command is defined: upon loading the file, predicate main/0 will be invoked.

```
0 :-use_module(library(linda/client)).

1 main:-
2     initialiseConnection,
3     mainAgent,
4     terminateConnection.

5 mainAgent:-
      ...

n :-main.
```

Fig. 4: Template for Agents

The template above should be completed with an appropriate definition for mainAgent/0 and for any auxiliary predicates. The definitions of **Fig. 2** should also be part of the agent's code. If an agent implemented as the template above is stored in file agent.pl, it can be started as a background process via the UNIX command "echo "[agent]." | sicstus > /dev/null &" which, similar to the start-up of our server, invokes SICStus and loads in program agent.pl, diverting any outcoming system messages to the file /dev/null, the standard dump file.

2.4 A "Launching Pad" for MASs

The sequence of steps depicted above required to start up a set of SICStus Prolog Linda client-server processes can be rather hindering when one is experimenting with simulations for MASs. Since a number of attempts are likely to be made, much time and energy will have to be spent setting up files to run in the background, killing processes that went awry and deleting files of previous versions. If large numbers of components are required then following the steps above manually makes experimentation virtually impossible.

To solve this problem we have developed a "launching pad" with which a number of components can be automatically started up. We show below in **Fig. 5** a diagrammatic representation for our tool. Starting from a set of specifications $S_1, S_2, \ldots, S_n$ with the actual Prolog code for each agent, our **LPad** tool starts up (denoted by the dotted arrows in our diagram) Linda client processes $Ag_1, Ag_2, \ldots, Ag_n$, one for each agent specification S_i, and a process Sv, the Linda server. **LPad** automatically starts up a Linda server process, along the lines of the program `server.pl` shown in **Fig. 1**. Then, for each specification S_i, **LPad** automatically synthesises a self-contained, executable SICStus Prolog program, fully compatible with the server process set initially. The common routines that define the connection and communication with the server are added to the definition of the agent making it a complete definition of a self-sufficient program to be executed as a client process.

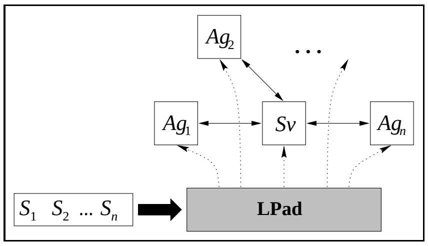

Fig. 5: Launching Pad **LPad**

A collection of files are managed by **LPad** during an execution. For each Linda client and server processes **LPad** creates a file with an actual Prolog program. These programs will be read in and executed, each one by an independent SICStus background process. A unique prefix for the names of those files created in a run of **LPad** is assembled with details of the date and time when the run starts. The execution of the processes in the background can be terminated at any moment by the user of **LPad**: if the user decides to stop the execution, then the server and client processes are all killed and their corresponding files are deleted. In order to enable the automatic "killing" of processes, their identification is obtained and stored and routines to insert and delete process id's had to be provided as part of the extra housekeeping routines.

3 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 [26] and InterProlog [4], 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 such layers thus forcing users of such languages to become acquainted (or more likely, experts) with Java programming.

4 Conclusions and Directions of Research

We have presented an approach to simulating logic-based MASs. This presentation has enough details so as to allow the reproduction and adaptation of our ideas. In our proposal the components of a MAS are self-contained Prolog programs with their own control of execution (that is, with identity and asynchrony) and that exchange message (that is, a means for communication is offered) by means of a shared memory. Stable technology of SICStus Prolog Linda client-server processes has been employed to build a tool, **LPad**, a launching pad for logic-based MASs.

The homogeneity of our approach is an obvious advantage: the very same language, Prolog, is used to program both the simulation infrastructure and the agents themselves. Since there are no gaps caused by mixing different technologies (as for instance, Prolog with Java front-ends) all technical details are visible and no extra training is required from users. Ideally people wanting to simulate logic-based MASs should not be aware of intricate low-level details of the underlying machinery but in case customisation or special services are required all details of our platform are available. Because only Prolog is actually used, tracing and debugging is also possible.

An important advantage of using SICStus Prolog Linda client-server modules lies in the possibility of running agents in geographically remote machines using standard Internet technology. As far as the program themselves are concerned there are no differences between running them locally or over the Internet. This allows for realistic scenarios to be exploited. If individual agents are run in different machines true asynchrony is achieved. Additionally, alternative means to handle messages can be programmed, for instance, by delaying the delivery of messages or "losing" some of them. Agents can be programmed to manage the tuple space and implement any policies for message delivery. Distinct "communication channels" can be defined by using different terms in messages. The tuple space can also be organised differently via an appropriate encoding of the format of messages.

A larger number of people would benefit from our ideas if the Prolog interpreter we used was freely available. Another restriction concerns the agents that can partake in an experiment: only agents programmed in SICStus Prolog can join in. Additionally, problems may arise when the MAS is simulated with a large number, *i.e.* hundreds, of components. Each component is implemented as an independent Linda client process with its own Prolog session and if too many of them are started up simultaneously, they may deplete the resources of the machine in which they are being run. The maximum number of Prolog sessions that can be run is of course dependent on the specification of each machine and hence we cannot be more precise. However, different machines can be employed to run the client processes and we can set up a group of machines (and add as many machines to this group as we need) thus increasing the number of agents we can cope with.

To try out our ideas, we have programmed different MASs, carrying out different kinds of auctions [20] and negotiations [9] as well as a larger scale MAS with more than 150 agents modelling a supply network [12]. Special messages were used to log data during experiments: a supervisor agent was responsible for retrieving these messages and writing the data in an appropriate format.

The service to log data could be incorporated to the basic resources of our **LPad**. Useful visual aids could also be provided allowing users to follow (and perhaps intervene in) a simulation. We would like to offer, in particular, visual means to follow a simulation in terms of message-passing and internal actions of agents. If each agent stores a suitable account of its execution we can *re-enact* a run by finding one appropriate ordering for these accounts – this would solve the problem one faces when executing communicating processes when a run may never be replicated again.

There are other techniques for parallel/distributed computation available in logic programming languages [14]. CIAO Prolog [2], for instance, offers ways to achieve multithreading: portions of a program can be independently executed in parallel. We are currently investigating this possibility. We have also detected two areas where we can apply our approach. One concerns providing legacy software in Prolog with an agent wrapper. The other area is the synthesis of web-sites: a logic-based web-site specification can be augmented with logic-based agents that would be held responsible for parts of the site. We are carrying out work on these topics.

Acknowledgements: Thanks to C. Walton for proofreading earlier versions of this text. Any remaining mistakes or imprecisions are the responsibility of the author only.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System Reference Manual. Technical Report, The Ciao System Documentation Series CLIP 3/97.1, Version 1.6#3, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid (UPM), November 2000.
3. H.-D. Burkhard. Liveness and Fairness Properties in Multi-Agent Systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 325–330, Chambéry, France, 1993.
4. M. Calejo. InterProlog: A Simple yet Powerful Java/Prolog Interface. *COMPULOG Newsletter*, December 1998. Updated version available at `http://www.declarativa.com/InterProlog/default.htm`.
5. N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.
6. N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, April 1989.
7. C. Dixon, M. Fisher, and M. Wooldridge. Resolution for temporal logics of knowledge. *Journal of Logic and Computation*, 8(3):345–372, 1998.
8. R. Fagin, J. Y. Halpern, and M. Y. Vardi. What Can Machines Know? On the Properties of Knowledge in Distributed Systems. *Journal of the ACM*, 39(2):328–376, 1992.

9. P. Faratin, C. Sierra, and N. R. Jennings. Negotiation Decision Functions for Autonomous Agents. *Robotics and Autonomous Systems*, 24(3–4):159–182, 1998.
10. M. Fisher and M. Wooldridge. Executable temporal logic for distributed A.I. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (IWDAI-93)*, pages 131–142, Hidden Valley, PA, May 1993.
11. P. A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.
12. C. M. Harland. Supply Chain Management: Relationships, Chains and Networks. *British Journal of Management*, 7:63–80, 1996.
13. P. Johnson and A. Lancaster. SWARM User Guide. SWARM Development Group, `http://www.swarm.org/release-docs.html`, 2001.
14. J. C. de Kergommeaux and P. Codognet. Parallel Logic Program Systems. *ACM Computing Surveys*, 26(3):298–336, September 1994.
15. M. Klusch, editor. *Intelligent Information Agents.* Springer-Verlag: Berlin, Germany, 1999.
16. Intelligent Systems Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, available at `http://www.sics.se/isl/sicstus2.html#Manuals`, February 2000.
17. B. van Linder. *Modal Logics for Rational Agents.* PhD thesis, Utrecht University, The Netherlands, 1996.
18. Andreas Nonnengart. First-order modal logic theorem proving and standard prolog. Mpi-i-92-228, Max-Plank-Institut für Informatik, July 1992.
19. Andreas Nonnengart. First-order modal logic theorem proving and functional simulation. In *IJCAI-93*, 1993.
20. P. Noriega and C. Sierra. *Auctions and Multi-agent Systems*, pages 153–175. Springer, 1999.
21. 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.
22. A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.
23. 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.
24. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 2nd edition, 1994.
25. V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems.* MIT Press/AAAI Press, Cambridge, MA, 2000.
26. P. Tarau. JINNI: A Prolog Interpreter in Java for Mobile Agent Scripting and Internet Programming. Available at `http://www.binnetcorp.com/Jinni/index.html`, July 1999.
27. 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.
28. M. Wooldridge and A. Rao, editors. *Foundations of Rational Agency.* Kluwer Academic Publishers: Boston, MA, 1999.