Implementing a Generic Agent Interpreter in Prolog

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

We describe a simple and practical implementation for a generic agent interpreter in Prolog. To achieve the desirable asynchronicity of distinct agents as well as to allow for actual message-passing among them we make use of standard technology for distributed/parallel computation in Prolog. We also present **LPad** a tool to support the automatic upload and execution of a number of agents using our implementation proposal.

1 Introduction

This document describes a simple and practical way to implement the generic agent interpreter described in [7] using the Linda concept for process communication [2, 3] available in SICStus Prolog [5]. In order to keep our proposal simple, we shall resort to a higher level description of some features of our agents. However, we shall provide our readers with sufficient implementational details so as to allow the reproduction of our experiments and their adaptation and extension.

We are investigating currently available technologies for distributed and parallel computation and their deployment for the implementation of agent-based architectures. We are interested not only in practical issues (for instance, asynchronicity and message-passing among agents) but we also want to explore and assess the adequacy of such technologies when there is a substantial scale-up in the quantity of interacting agents.

In our research project we shall be using computational logic for the specification, analysis and, ultimately, automatic generation of agents. To allow for homogeneity, the agents themselves will also be implemented in a logic programming language. In order to keep our discussion concrete, we have employed SICStus Prolog [5] under a UNIX operating system[1]. Rather than simulate important practical aspects of agent-based programming, oversimplifying and weakening our study, we have decided to provide an accurate execution model featuring actual message-passing and asynchronicity among agents. SICStus Prolog's Linda process communication allows the quick, clean and easy implementation of our agents, as we show below.

Our proposal can be graphically depicted as Figure 1. Each box represents a *process*, that is, an independent

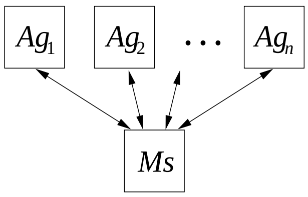

Figure 1: Implementation Proposal for Agents

program which can be running in its own distinct computer – as we explain below, if necessary the processes can be run in different machines geographically apart using Internet links to enable the communication among them. The $Ag_1, Ag_2, \ldots, Ag_n$ boxes are our agent processes the details of which are given below. Process *Ms* is a special message-handling process and it has been included, as we explain below, to allow for more flexible forms of message manipulation and distribution among the agent processes. Agents and message-handling programs are to be implemented as Linda *client* processes as explained below. A *server* process, not shown in the diagram above, must also be running in order to make communication possible – we describe below how such a process can be automatically started.

In the next Section we explain our adopted agent interpreter and define a way for its straightforward implementation. In Section 3 we provide details of how SICStus Prolog's Linda client-server setup can be employed for agent-oriented programming. In Section 4 we explain our message-handling agent. In Section 5 we present a tool to automate the starting up of a number of client-server processes: this tool allows programmers to concentrate on the agents themselves; all other details of setting up the Linda client-server processes is left to our tool. Finally in Section 6 we draw conclusions and give directions for future work.

[1]Additional advantages for this decision are those arising from the use of a higher-level declarative language, *viz.*, conciseness, powerful programming constructs, symbolic manipulation, etc. [6].

2 A Generic Agent Interpreter

In [7] we are presented with a useful proposal for a generic agent interpreter. We have incorporated this proposal because it is simple and robust and sufficient to our needs. However, we have reasons to believe that our experiments can be adapted to other proposals, as we explain in Section 6. We reproduce in Figure 2 the flow diagram for such interpreter (numbers have been added for easy reference to specific portions of the diagram in the ensuing discussion). Each agent repeats the following two steps at regular intervals:

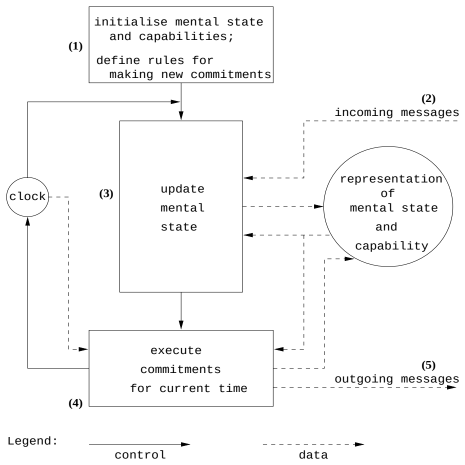

Figure 2: Flow Diagram for a Generic Agent Interpreter (reproduction from [7])

1. read the current messages and update mental state (beliefs and commitments);
2. execute commitments for current time, possibly resulting in further belief change.

We show our implementation for the generic agent interpreter above in Figure 3 (we have added numbers to the left of each relevant line so as to enable easy reference to specific parts of the program). To keep our discussion simple and free from more specific design decisions, we assume that a suitable representation for the mental state and capabilities of the agent should be supplied by the people engineering the agents. Line 0 informs SICStus that the `system` library

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

1    main:-
2        initialiseConnection,
3        initialise(Clock,MentalState,Capabilities),
4        loop(Clock,MentalState,Capabilities),
5        terminateConnection.

6    loop(Clock,MentalState,Capabilities):-
7        terminationConditions(Clock,MentalState,Capabilities).
8    loop(Clock,MentalState,Capabilities):-
9        timeGrain(Tick),
10       sleep(Tick),
11       NewClock is Clock + Tick,
12       checkInMsg(InMsgs),
13       updateMentalState(MentalState,NewMentalState,InMsgs),
14       executeCommitments(Capabilities,NewClock,NewMentalState,NewCapabilities,OutMsgs),
15       sendMsgs(OutMsgs),
16       loop(NewClock,NewMentalState,NewCapabilities).
```

Figure 3: Implementation of Generic Agent Interpreter

of built-ins should be loaded (that is required because we make use of predicate `sleep/1` in the program) and that our program uses `linda/client` process communication built-ins (we explain more about this in Section 3). Predicate `main/0` (lines 1–5) is the start-up of the agent interpreter. It is responsible for initialising the connection between a specific agent interpreter (predicate `initialiseConnection/0` in line 2) at the very onset of the agent's execution and is also responsible for terminating the connection at the end of the agent's run (`terminateConnection/0`, in line 5). These two predicates are not part of the generic agent interpreter flow diagram; they are related to the technology chosen to implement the agents and are explained in more detail below. Once the variables `Clock`, the initial numeric value of the agent's clock, `MentalState` and `Capabilities`, the initial representation for the agent's mental state and capabilities, respectively, are appropriately initialised (predicate `initialise/3`, in line 3, corresponding to component 2 of the flow diagram above), the interpreter `loop/3` (lines 6–16) is started.

The `loop/3` predicate (lines 6–16) is the essence of our generic agent interpreter. Its first clause (lines 6 and 7) checks whether the termination conditions have been reached; if the agent is to perform an infinite loop this clause should be removed or `terminationConditions/3` should be defined simply as `fail`. The second clause (lines

8–16) initially manages any timing restrictions, getting hold of the specification for the time granularity (predicate `timeGrain/1`, declared by the agent's designer), gets the agent to wait (built-in predicate `sleep/1`) in line 10 for an adequate amount of time (given by the time granularity) and updates the clock (line 11). The ensuing lines 12 to 16 comprise a straightforward reading of the flow diagram in Figure 2: line 12 is responsible for retrieving incoming messages `InMsgs`; line 13 updates the agent's mental state based on the incoming messages; line 14 executes commitments, updates the agent's capabilities and prepares outgoing messages, to be sent by `sendMsgs/1` in line 15; line 16 ensures the continuation of the loop with the new values. We give more details about the routines for sending and receiving messages in Section 3.

3 Linda for Process Communication

Linda [2, 3] is a concept for process communication. It consists of an area where any sent messages are stored (that is, the tuple space) and a few simple operations on it. SICStus Prolog [5] incorporates Linda's ideas and offers a repertoire of built-in predicates with which one can write programs that run independently and exchange messages.

We distinguish between two kinds of Linda programs (or processes), the *server* and the *client*. There should be one and only one server process and one or more processes running as clients. The server works as a blackboard onto which clients can write and from which they can read and remove data. If a piece of requested information is not found in the blackboard, alternatives are offered to suspend the requesting process (for a limited time or indefinitely) until that information becomes available. The server and client processes communicate by means of sockets, established via appropriate commands, explained below.

Our Linda server process is the simple program of Figure 4. Line 0 informs SICStus that the `linda/server` library

```
0   :-use_module(library('linda/server')).

1   :-linda((Host:Port)-(tell('messages.addr'),
2                        write('\''),write(Host),write('\':'),write(Port),write('.'),
3                        told))."
```

Figure 4: Linda Server Program `server.pl`

should be loaded. Line 1 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-Goal)` where `Address` is of the form `Host:Port` and `Goal` must be instantiated to a goal [5]. When this predicate is executed, it starts up the server at `Address` and runs `Goal`. 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 in file `messages.addr` – the recording of the file comprises our `Goal`. The single quotes "`\'`" around the host name are *essential* if the installation setting employs machine names containing special characters such as dots. Since they will not interfere when the names are strings of simple characters they should not be removed. If a precise identification for the `Host` is employed (*viz.* the machine name and its domain), the socket will allow the communication between processes running in different machines geographically apart using standard Internet technology.

We may want to hide the server program: 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 by means of 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[2].

It is essential for a client process to have access to the socket information so that a communication channel be established between itself and the server. There is no direct communication among client processes: their messages are sent over to the server and the server is responsible for receiving message requests and for delivering them. A client process can only be connected to one server at a time, but it can be connected to one server at one point, disconnected and then connected 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[3]. 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 when, for instance, a specific item of information was sent and stored as the n-th message and if it is then specifically requested for by a process, it will leapfrog the $n-1$ preceding messages and be delivered first.

3.1 Receiving and Sending Messages via SICStus Prolog Linda

In this section we describe how client processes send and receive messages. It should be pointed out that the server process does not explicitly send nor receives messages, that is, it is not necessary that we program the server. Any messages sent from client processes will be stored in the message area of the server; any requests for messages from

[2]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 in Section 5, however takes care of this, keeping track of processes id numbers and killing them at the user's request.

[3]That is why we have included a message-handling agent to provide for more sophisticated message manipulation.

client processes will be appropriately dealt with and delivered by the server without us having to worry about it. Clients, however, have to be programmed to send and request appropriate messages.

We describe how client processes communicate by means of our agent interpreter. Recall that our program of Figure 2 will be run as a client process. Line 0 of the program in Figure 2 ensures that the appropriate library is loaded. The predicates of Figure 5 invoked from `main/0` and `loop/3` complete our agent interpreter. Predicate

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

 6   terminateConnection:-
 7       close_client.

 8   checkInMsg(msg(Sender,Agent,Msg)):-
 9       agent(Agent),
10       in_noblock(msg(Sender,Agent,Msg)).
11   checkInMsg(nil).

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

Figure 5: Complementary Routines for Agent Interpreter

`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 (that is, the server should be started first!) is read in and then closed. Its contents, of the form `Host:Port`, is used as the argument of built-in `linda_client/1` which is ultimately responsible for establishing the actual connection between the client and the server. Predicate `initialiseConnection` should be invoked 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 `checkInMsg/1` (lines 8–11) checks if there is a message for `Agent`, identified by `agent/1`. It does so by invoking Linda client built-in predicate `in_noblock/1` whose syntax is `in_noblock(Tuple)` and it succeeds if there is a message in the message area of the server (that is, a message that has been sent before) that unifies with `Tuple`. In our implementation, this predicate will succeed if there is a message of the form `msg(Sender,Agent,Msg)`. Since `Agent` gets instantiated by `agent/1` to the identification of the agent, there is a further restriction that the second argument of term `msg` (the addressee) unifies with `Agent`. The format of the message is a design choice. 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 message area, the predicate fails. The timeout period can be set up via the Linda (client) built-in predicate `linda_timeout(OldTimeOut,NewTimeOut)`. If there is no such message, 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` (lines 12–15) receives a list of messages in any 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).

4 The Message-Handling Agent

To allow for more sophisticated forms of message manipulation, we have added a message-handling agent to our proposal. This special client process stores all messages sent to the server, analyses them and send messages to the other clients (via the server). One simple way to implement our message handling agent is shown in Figure 6. Since

```
1   main:-
2       initialiseConnection,
3       messageHandling([]).

4   messageHandling(CurrentMsgs):-
5       in(hnd(NewMsg)),
6       updateMessages(CurrentMsgs,NewMsg,NewCurrentMsgs,OutMsgs),
7       sendMsgs(OutMsgs),
8       messageHandling(NewCurrentMsgs).
```

Figure 6: Message-Handling Agent Interpreter

the message-handling agent is also a Linda client process, the code above should also contain line 0 of Figure 2, but we have left that out to save space, as well as predicates (`initialiseConnection/0` and `sendMsgs/1`. In this edited version of our message-handling agent we have concentrated in more essential aspects of message manipulation: predicate `main/0` (lines 1–3) initialises the connection and starts the `messageHandling/1` loop – since this loop is not supposed to end, there is no concern about terminating the connection with the server.

The `messageHandling/1` predicate (lines 4–8) manages a list of messages (set initially to an empty list upon its initial call in `main/0`) according to some policy. Predicate `in/1` in line 5, a Linda client built-in, checks for a message with the specific syntax `hnd(Msg)` – contrary to the `in_noblock/1` predicate of our agent interpreter, the `in/1` predicate *blocks* the program execution [5], waiting indefinitely for a message with that syntax to be sent. Predicate `updateMessages/4` implements the adopted policy for message handling: using the currently available messages stored in `CurrentMsgs` and the newly received message `NewMsg`, it should prepare and store in `OutMsgs` a batch of messages to be sent and record in `NewCurrentMsgs` the new state of the message area. The newly prepared outgoing messages are sent by `sendMsgs/1` in line 7 and the `messageHandling` loop goes on with the new message area.

We notice that the message-handling agent can also be seen simply as an agent interpreter with the message manipulation being performed as the update of mental state and capabilities. In this case, the mental state should represent the traffic of messages and the capabilities and their update should embed the policy for handling messages.

5 A "Launching Pad" for Agents

The sequence of steps depicted above required to start up a set of Linda client-server processes in SICStus can be rather hindering when one is developing an agent-based solution for a problem. Since a number of attempts have to be made before a stable version of the agents can be designed, 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 tens of agents may be 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 agents can be automatically started up. We show below in Figure 7 a diagrammatic representation for our tool. Starting from a set of specifications

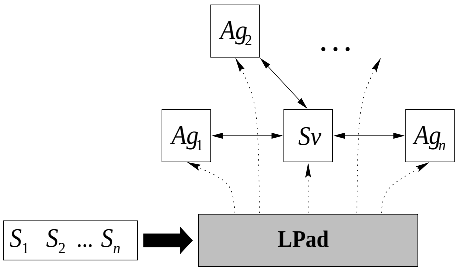

Figure 7: Diagram of Launching Pad **LPad** for Agents

$S_1, S_2, \ldots, S_n$ 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 Figure 4 of Section 3. Then, for each specification S_i, **LPad** automatically synthesises a self-contained, executable SICStus Prolog program, fully compatible with the server process set initially. A specification S is, in fact, a list of Prolog clauses defining an *open program* [4], that is, there are still some predicates referred to but awaiting their definition.

These predicates yet to be defined are synthesised "on-the-fly" by **LPad**. Because they refer to files which will be created by the server process started up initially, they cannot be prepared in advance. We could have agreed on fixed names for the files, thus avoiding the issue, but this would prevent simultaneous (albeit independent) experiments from being held since the file names would clash. The general syntax of a synthesised program for each specification S_i is shown in Figure 8 below. The contents of each specification S_i comprise the definition of `mainClient/0` of line 5 – this predicate should be defined by the agents programmer along the lines of `main/0` in our generic agent interpreter of Figure 3 without the calls to `InitialiseConnection/0` and `terminateConnection/0`. The definition supplied as S_i is inserted in the template where the "`...`" is, as well as any auxiliary predicates. A new `main/0` predicate is employed in our program (lines 1–4) responsible for initialising the client-server connection (line 2), calling up the `mainClient/0` predicate (line 3) and closing the connection (line 4). Since the client process will be read in and interpreted by a SICStus background process, our template must have a command "`:-main.`" (line 28) to instruct the interpreter on what should be initially done, that is, call up predicate `main/0` to start the agent.

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 all client processes are all killed and their corresponding files are deleted. In order to enable the automatic "killing" of processes, their identification had to be stored in a special file (generically referred to as `pIdSFileName`) and routines to insert and delete process id's had to be provided as part of the extra housekeeping routines.

Predicate `initialiseClientServer/0` (line 6–17) refers to a constant `handleFileName` underlined. This constant is automatically set to the name of the file containing the information on the handle, generated by the server process previously started. The information on the host machine and the port through which the communication channels among processes are established is read in (lines 7–9), the connection is established (line 10) and information on the agent's process identification is obtained (built-in `pid/1` on line 11) and included (lines 12–17) in file `pIdSFileName`

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

1   main:-
2       initialiseClientServer,
3       mainClient,
4       terminateClientServer.

5   mainClient:-
        ...

6   initialiseClientServer:-
7       see(handleFileName),
8       read(Host:Port),
9       seen,
10      linda_client(Host:Port),
11      pid(PId),
12      see(pIdSFileName),
13      read(pId(AllPIdS)),
14      seen,
15      tell(pIdSFileName),
16      portray_clause(pId([PId|AllPIdS])),
17      told.

18  terminateClientServer:-
19      close_client,
20      pid(PId),
21      see(pIdSFileName),
22      read(pId(AllPIdS)),
23      seen,
24      delete(AllPIdS,PId,RestPIdS),
25      tell(pIdSFileName),
26      portray_clause(pId(RestPIdS)),
27      told.

28  :-main.
```

Figure 8: Template of Client Process for an Agent

(a constant set to its actual value on the fly). Predicate `terminateClientServer`/0 (line 18–27) closes the connection (line 19) and removes (line 20–27) the process identification from the current list of process identification numbers in file `pIdSFileName`.

We show in Appendix A the complete listing of the source code of our **LPad** tool.

6 Summary, Conclusions and Directions of Research

In this document we present a simple and practical way to implement the generic agent interpreter proposed by Shoham [7] in Prolog. We have made extensive use of Linda [2, 3] process communication concepts offered in SICStus Prolog [5]. Ours is a realistic implementation in the sense that asynchronicity and message-passing is actually realised.

We envisage a scenario in which our agent interpreters are (semi-) automatically designed either by means of a higher-level specification or by user interactions via a suitable interfacing program. In either case once the agent programs are completed they should be properly executed as background UNIX processes. Recall that each agent program in order to run properly has to have access to the file `messages.addr` created by the `server.pl` program above and containing the socket information through which the message passing will take place. A UNIX command along the lines of that shown in Section 3 for the server process can be prepared taking into account the name of the file where the agent interpreter program is stored. Attention should be paid to have copies of the file `messages.addr` made if the agents are to be run in machines with different file storage areas.

A specific order of steps should be followed to get a set of agent interpreters running in our proposal. First of all the Linda server process should be started running the `server.pl` program file (a side-effect of this is the creation of file `messages.addr`). Secondly, the message-handling agent (a Linda client process) should be programmed and started; this program should embed the adopted policy for message-passing. Thirdly and lastly, the agent interpreters (also Linda client processes) should be manually or (semi-)automatically defined and started. To disencumber humans from the responsibility of manually following all the steps (which can be rather hindering if many agents/processes have to be started) a tool **LPad** (for **L**aunching **Pad**) has been developed which automatically starts up and manages a collection of Linda client-server processes specified in a particular format.

To try out our ideas, we have programmed simple agent interpreters which carry out basic negotiations concerning requests and offers for loans. The requesting agents pose a request for a loan and receive a number of offers; they then choose the one with the least interest rate. The agents to offer the loan receive a number of requests and choose the ones with smallest risks based on the requester information. We ran our experiment with a small number of agents (between 2 and 10 of them) and we tried different settings in which agents were running locally and in machines geographically apart (Zurich and Edinburgh).

Agent programming depends entirely on message passing and extreme care must be taken when designing this part of the system: it can be very confusing to manage the sending and receipt of messages. To ensure that messages are received by the correct agent we suggest the use of an appropriate term as a tuple. For instance, we may want to include in the syntax of the message the identification of the agent to which the message is being sent and use this information when checking for messages. By doing so we can guarantee that the right addressee receives the message. Distinct terms for messages may be used to define private "communication channels": in our implementation, we have used `hnd(Msg)` to represent messages sent to the message-handling agent and `msg(Sender,Addressee,Msg)` to represent messages sent to the other agents. The syntax of the terms impedes that messages be wrongly delivered.

Our proposal is only suitable for applications in which fewer agents are required. This is due to the fact that to be executed each agent requires that SICStus Prolog be loaded thus depleting the computational resources of the computer(s) involved. We cannot be more precise about the actual limit in the numbers of agents since this is dependent on the machines in which they are to be run. This issue, however, can be eschewed if *executable* versions of the agents are prepared: if the executable code of each agent is small enough a large number of them could be run at the same time. Although SICStus Prolog offers the possibility of creating executable code from a given program, problems in our installation have prevented us from investigating this issue further.

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.

The two very distinctive properties of agent programming, *viz.* message passing and asynchronicity can be easily captured with our implementation proposal. However, other models of agents different from the one we have adopted could have been used to exploit our ideas. We would like to investigate some of these other models to assess the adequacy and adaptability of our implementation proposal to them.

There are other techniques for parallel/distributed computation available in logic programming languages. SICStus Prolog, for instance, offers another package called Muse for running programs in parallel. CIAO Prolog [1], on the other hand, offers ways to achieve multithreading: portions of a program can be independently executed in parallel. We are currently investigating these (and other) possibilities.

References

[1] D. Cabeza, M. Hermenegildo, and S. Varma. WWW Programming using Computational Logic Systems (and the PiLLoW/CIAO Library). Technical report, Computer Science Department, Technical University of Madrid, 1996.

[2] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[3] N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, April 1989.

[4] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. *LNCS*, 1463:128–??, 1998.

[5] 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.

[6] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994. Invited Paper.

[7] Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60:51–92, 1993.

A Source Code for Launching Pad LPad

In this Section we list the source code for the launching pad **LPad** explained in Section 5. The prototype was successfully run under SICStus version 3.5.

```
% ------------------------------------------------------------------
% launch.pl
% - dates
%     started on 4 July 2000
%     finished on 6 July 2000
% - author
%     Wamberto Vasconcelos (wvasconcelos@acm.org)
% - purpose
%     this program works as a ``launching pad'' to automatically
%   start up a set of Linda client-server communicating processes.
% - input
%     file agents.sp containing a list of agents specifications. Each
%   agent is a Prolog program that will run normally as a background
%   client process
% - output
%     a family of files are generated as part of this program execution.
%   They are, however, deleted at the end, so as to save humans the trouble
%   to delete them by hand. The program uses a prefix name to create the
%   names of the files -- the prefix is of the form
%      fYear_Month_Day_Hour_Min_Sec_
%   where Year, Month, Day, Hour, Min and Sec are numeric values obtained
%   at the begining of the execution. If we call the string above Prefix,
%   the files created (and later deleted) are:
%      1. Prefixserver.pl -- Prolog program to be run in the background as
%         a Linda server process,
%      2. Prefixserver.addr -- file containing the Host:Port info which
%         provides a ``handle'' for Linda client processes
%      3. PrefixpId.pl -- contains the process identification for all
%         processes started up by launch.pl (for later ``killing'')
%      4. Prefixclient_I.pl -- (where 1 =< I =< n, n being the number of
%         agents in file agents.sp) contains the actual Prolog code of
%         each specified agent, to be run in the background as a Linda
%         client process, comunicating via the server (file 1, above)

:- use_module(library(system)),use_module(library(lists)).

% ----------------------------------------------------------------------------

main:-
    preparePrefixNameFiles(Prefix),     % prepare Prefix
    createServerProgram(Prefix,         % creates server program
                        ServerName,
```

```
                        HandleFileName,
                        PIdFileName),
    startServer(ServerName),            % starts up server in background
    startClients(Prefix,                % creates and starts up
                 HandleFileName,        % client processes
                 PIdFileName,
                 ClientFilesNames),
    cleanUp(ServerName,                 % deletes files and kills processes
            HandleFileName,
            PIdFileName,
            ClientFilesNames).

% --------------------------------------------------------------------------
% preparePrefixNameFiles(-Prefix):-
%   prepares Prefix, a string to be added at the beginning of all file
%   names of the system. Prefix is of the form
%       f*Year*_*Month*_*Day*_*Hour*_*Min*_*Sec*_
%   where *Year*, *Month*, etc. are the contents of the variables used to
%   store the year, month, etc. in numeric format.

preparePrefixNameFiles(Prefix):-
    datime(datime(Year,Month,Day,Hour,Min,Sec)),
    concat([f,Year,'_',
              Month,'_',
              Day,'_',
              Hour,'_',
              Min,'_',
              Sec,'_'],Prefix).

% --------------------------------------------------------------------------
% createServerProgram(+Prefix,-ServerName,-HandleFileName,-PIdFileName):-
%   creates a simple Linda server program the contents of which is
% :- use_module(library('linda/server')),
%    use_module(library('system')),
%    pid(PId),
%    tell('*Prefix*pId.pl')),
%    portray_clause(pId([PId])),
%    told,
%    linda((Host:Port)-(tell('*Prefix*server.addr'),
%                        write('\''),
%                        write(Host),
%                        write('\':'),
%                        write(Port),
%                        write('.'),
%                        told)).
%   where *Prefix* is the contents of the variable Prefix. The file is
%   saved as *Prefix*server.pl (the *precise* format above is kept in
%   the file). This predicate also prepares the names for the files
%   containing the server program, the handle information (HandleFileName)
%   and the process numbers (PIdFileName).

createServerProgram(Prefix,ServerName,HandleFileName,PIdFileName):-
    concat([Prefix,'server.pl'],      % prepares files names
            ServerName),
    concat([Prefix,'server.addr'],    % these file names are used in the
            HandleFileName),          % commands of the server program
    concat([Prefix,'pId.pl'],
            PIdFileName),
    tell(ServerName),                 % prepares the contents of server...
    write(':- use_module(library(\'linda/server\')),'),nl,
    write('   use_module(library(\'system\')),'),nl,
    write('   pid(PId),'),nl,
    write('   tell(\''),write(PIdFileName),write('\'),'),nl,
    write('   portray_clause(pId([PId])),'),nl,
    write('   told,'),nl,
    write('   linda((Host:Port)-(tell(\''),
    write(HandleFileName),write('\'),'),nl,
    write('                       '),
    write('write(\''),write(\),write('\'\'),'),nl,
    write('                       '),
    write('write(Host),'),nl,
    write('                       '),
    write('write(\''),write(\),write('\':\'),'),nl,
    write('                       '),
    write('write(Port),'),nl,
    write('                       '),
    write('write(\'.\'),'),nl,
    write('                       '),
    write('told)).'),
    told.

% --------------------------------------------------------------------------
% startServer(+ServerName):-
%   starts up file ServerName as command
%     echo "['*ServerName*']" | sicstus > /dev/null &

startServer(ServerName):-
    concat(['echo "[\'',ServerName,
            '\']." | sicstus > /dev/null &'],
            Command),
    exec(Command,                     % SICStus' built-in
         [null,null,null],
         _).

% --------------------------------------------------------------------------
% startClients(+Prefix,+HandleFileName,+PIdFileName,-ClientFileNames):-
%   starts up a sequence of client processes in the background and
% returns a list with their names (for posterior deletion). The Prefix,
% HandleFileName and PIdFileName are required for naming the files and
```

```
% setting up the programs according to server process above.

startClients(Prefix,
             HandleFileName,
             PIdFileName,
             ClientFileNames):-
    see('agents.sp'),                % file containing the ``specification"
    read(agents(AllAgents)),         % in this format
    seen,
    createClients(AllAgents,         % create a list of clients...
                  1,                 % numbering them from 1 onwards
                  Prefix,            % using this information
                  HandleFileName,    % and this info
                  PIdFileName,       % and yet this info
                  ClientFileNames),  % and return a list of file names
    startAllClients(ClientFileNames). % which is used to start the processes

% ------------------------------------
% createClients(+ListClients,
%               +Number,
%               +Prefix,
%               +HandleFileName,
%               +PIdFileName,
%               -ListClientFileNames):-
%    For each element in ListClients a number (1, 2, etc) is assigned and
% a program file is created with the clauses of the element itself and a
% small repertoire of other clauses responsible for setting up the Linda
% client-server communication channels. The HandleFileName and PIdFileName
% are required to customise the program to the current setup (every run has
% a different server process)

createClients([],_,_,_,_,[]).
createClients([Client|Clients],
               ClientNo,
               Prefix,
               HandleFileName,
               PIdFileName,
              [ClientFileName|ClientsFileNames]):-
    concat([Prefix,
            'client_',
            ClientNo,'.pl'],
           ClientFileName),          % builds program name
    tell(ClientFileName),            % creates file with contents below
    portray_clause((:- use_module(library(system)),
                       use_module(library(lists)),
                       use_module(library('linda/client')))),
    nl,
    nl,
    portray_clause((main:-initialiseClientServer,
                          mainClient,
                          terminateClientServer)),
    writeClient(Client,              % contents of Client especification
                HandleFileName,      % is written out in the file
                PIdFileName),
    told,
    NewClientNo is ClientNo + 1,     % another number for next element...
    createClients(Clients,
                  NewClientNo,
                  Prefix,
                  HandleFileName,
                  PIdFileName,
                  ClientsFileNames).

% ------------------------------------
% writeClient(+Client,+HandleFileName,+PIdFileName):-
%   writes in file (current output stream -- see predicate above) the
% list of clauses of Client. The HandleFileName and PIdFileName are
% passed around because they are needed at the end.

writeClient([],HandleFileName,PIdFileName):-  % when no more clauses...
    nl,
    portray_clause((initialiseClientServer:-  % insert these clauses
                        see(HandleFileName),  % which are ``standard''
                        read(Host:Port),
                        seen,
                        linda_client(Host:Port),
                        pid(PId),
                        see(PIdFileName),
                        read(pId(PIdS)),
                        seen,
                        tell(PIdFileName),
                        portray_clause(pId([PId|PIdS])),
                        told)),
    nl,
    portray_clause((terminateClientServer:-
                        close_client,
                        pid(PId),
                        see(PIdFileName),
                        read(pId(PIdS)),
                        seen,
                        delete(PIdS,PId,RestPIdS),
                        tell(PIdFileName),
                        portray_clause(pId(RestPIdS)),
                        told)),
    nl,
    portray_clause((:-main)).
writeClient([C|Cs],HandleFileName,PIdFileName):-
    nl,
    portray_clause(C),
```

```
    writeClient(Cs,HandleFileName,PIdFileName).

% ------------------------------------
% startAllClients(+ClientFileNames):-
%   given a list of ClientFileNames, it starts up each of them as
% background processes. Their contents, prepared above, ensure that
% the processes employ the right files for the current execution.

startAllClients([]).
startAllClients([ClientName|ClientsNames]):-
    concat(['echo "[\'',ClientName,
            '\']." | sicstus > /dev/null &'],
            Command),
    exec(Command,                     % SICStus' built-in
         [null,null,null],
         _CommandPId),
    startAllClients(ClientsNames).

% --------------------------------------------------------------------------
% cleanUp(+ServerName,+HandleFileName,+PIdFileName,+ClientFileNames):-
%   this predicate asks every 10 seconds if the user wants to kill the
% server process (and the other existing client processes). If the answer
% is ``yes'' then the processes are killed and the files which gave rise
% to the processes are deleted.

cleanUp(ServerName,HandleFileName,PIdFileName,ClientFileNames):-
    sleep(10),
    getAnswer(Kill),
    cleanUp(Kill,ServerName,HandleFileName,PIdFileName,ClientFileNames).

% --------------------------------------------------------------------------

getAnswer(Answer):-
    nl,
    write('Do you wish to kill server process now? [yes/no] '),
    read(TypeIn),
    ( member(TypeIn,[yes,no]) ->
        Answer = TypeIn
     ;
        (nl,
         write('Please answer "yes" or "no"'),
         getAnswer(Answer))).

% --------------------------------------------------------------------------

cleanUp(yes,                         % if answer is yes...
        ServerName,
        HandleFileName,
        PIdFileName,
        ClientFileNames):-
    killProcesses(PIdFileName),
    delete_file(ServerName,[ignore]),
    delete_file(HandleFileName,[ignore]),
    delete_file(PIdFileName,[ignore]),
    deleteClientFileNames(ClientFileNames).

cleanUp(no,                          % if answer is no...
        ServerName,
        HandleFileName,
        PIdFileName,
        ClientFileNames):-
    cleanUp(ServerName,HandleFileName,PIdFileName,ClientFileNames).

% ------------------------------------

deleteClientFileNames([]).
deleteClientFileNames([FileName|FileNames]):-
    delete_file(FileName,[ignore]),
    deleteClientFileNames(FileNames).

% ------------------------------------

killProcesses(PIdFileName):-
    see(PIdFileName),
    read(pId(PIdS)),
    seen,
    killAllProcesses(PIdS).

% ------------------------------------

killAllProcesses([]).
killAllProcesses([PId|PIdS]):-
    kill(PId,9),                       % SICStus built-in
    killAllProcesses(PIdS).

% --------------------------------------------------------------------------
% Auxiliary Routines
% --------------------------------------------------------------------------

% concat(+ListStrings,?Concat):-
%   succeeds if Concat is the concatenation of the strings in ListStrings

concat(ListStrings,Concat):-
    ListStrings = [Str1|Strings],
    concatList(Strings,Str1,Concat).

concatList([],String,String).
concatList([S|Ss],StringSoFar,String):-
    concat(StringSoFar,S,NewStringSoFar),
```

```
    concatList(Ss,NewStringSoFar,String).

% concat(+Str1+,+Str2,?Str1AndSrt2):-
%   succeeds if Str1AndStr2 is instantiated to the concatenation of
%   Str1 with Str2

concat(Str1,Str2,Str1andStr2):-
    name(Str1,ASCStr1),
    name(Str2,ASCStr2),
    append(ASCStr1,ASCStr2,ASCStr1andStr2),
    name(Str1andStr2,ASCStr1andStr2).

% ---------------------------------------------------------------------------
% EOF -- EOF -- EOF -- EOF -- EOF -- EOF -- EOF -- EOF -- EOF -- EOF -- EOF
% ---------------------------------------------------------------------------
```