CaseLP: a Complex Application Specification Environment based on Logic Programming

Maurizio Martelli, Viviana Mascardi, Floriano Zini
DISI - Università di Genova
via Dodecaneso, 35 - 16146 Genova (Italy)
tel: +39 10 3536727 - fax: +39 10 3536699
e-mail: {martelli, mascardi, zini}@disi.unige.it

Abstract

More and more complex applications need to cope with the integration of different kinds of information, the reuse of existing software, the integration of well established tools and systems (such as databases). This and the distributed nature of many applications made it very appealing to use multi-agent technology. Rapid prototyping and executable specifications could be very important for the development of these applications and Logic Programming can prove itself extremely appropriate for this task.

The paper presents CaseLP, a specification environment for Multi-Agent Systems based on Logic Programming. Many of the desirable properties and features of an ideal system have already been implemented in CaseLP, which, as a first prototype, has already been proven very useful in the case of some real applications. The paper outlines the general features of the system, describes some aspects of the implementation and presents two applications that have been specified with CaseLP.

1 Introduction

The Agent Oriented Paradigm [Sho93] has notably increased in diffusion and importance in less than five years. The interest of the researchers has moved during these years from the theoretical formalization of one agent, to the study of societies of agents, called Multi-Agent Systems (or MAS for short) and in the last period to the possible applications of this technology to cope with real-world problems. A reason of the MAS success is that they provide an high level tool for developing applications in which many different components interact to coordinate themselves and to share and integrate knowledge. Until now, however, almost all the MAS for these applications have been developed in a bottom-up manner, resulting into many problem-dependent incomparable models, difficult to verify using formal established approaches. For this reason many researchers are beginning to focus their efforts towards some more established technology for building such systems and for filling the gap between theory and practice (see Section III in [MWJ96] for a discussion on this topic). Nevertheless, tools and techniques for designing MAS, which are becoming more and more necessary with the increasing of the MAS technology, aren’t yet very widespread and only few prototypes have been developed by researchers (see for example [BDKJT96] and [IGV95]). For example, due the inherent complexity of a MAS, it would be useful to define a formal specification of the system before the stage of implementation and it would be of great help to have some tool which assists the designer with the task of developing the specification. Further, if this specification could be executed, showing the effective behaviour of the system, the programmer could debug and improve it.

Our final aim is to develop a tool (we called it CaseLP, i.e., Complex Application Specification Environment based on Logic Programming) which addresses the following issues:

1. Providing a methodology for the development of complex software applications realized as Multi-Agent Systems.

2. Investigating how the integration of heterogeneous software modules and data (indispensable in real-world application) can be carried out in the context of communicating agents that have to exchange information.

3. Providing a set of specification languages for the definition of the behaviour of the agents as well as a methodology to translate/integrate them in a unique executable specification.
4. Defining a general agent architecture, to allow standardization and reusing in the development of applications based on MAS.

5. Providing tools for building an executable specification of an application, putting together different agents and for testing such a specification.

We are implementing the CaseLP system using the CLP language Eclipse [ACD+95]; we chose a logic programming paradigm for various reasons: first of all, it can be considered (as it is well known) a powerful and executable specification language; second, it easily allows to program agents that maintain a symbolic description of the world and to perform high-level reasoning over this description; third, the integration of both different software modules and multiple specification languages in a MAS can be made simpler using meta-programming techniques and easier to carry out in a logic programming paradigm. Finally, even if logic programs containing extra-logic primitives as assert and retract haven’t a well-defined semantics, it is possible to explore the semantics of a system involving dynamic agents which change their state by means of Linear Logic: this topic is under study at our Department [Del97].

In this paper we address some of the issues above. CaseLP is an ongoing project and we will describe a first real implementation of it; the prototype is still open to many integrations and has some limitations and rigid solutions that we hope to improve in the future; nevertheless we think of it as a real experimental research activity and we will modify it according to the needs of new complex applications which will be specified using CaseLP. The systems has proven itself very satisfying as a tool for simulating the behaviour of a MAS; in this paper we stress this feature: most of the work done up to now concerns CaseLP as a MAS simulation tool.

In Section 2 we describe the architecture and the behaviour of the agents that can be built using the CaseLP tool and how the integration of external software/data can be carried out; Section 3 concerns the specification of the single agents and ACLPL, (i.e., Agent Constraint Logic Programming Language), the language we have defined for programming the agents; Section 4 describes how the agents can be inserted in a context of a Multi-Agent System. A brief description of the implementation of the tool follows (Section 5); we also describe two real-case applications in which the use of our tool has been proven useful (Section 6) and the article ends with some considerations about the related and future work (Section 7).

2 Architecture of CaseLP agents

A CaseLP entity may be defined as an agent for four main reasons:

- the state is composed by facts which represent the world in a symbolic way; our case study applications didn’t require that the agents were characterized by sophisticated states, however it is still possible to define the agent’s state expressed in terms of mental attitudes such as Beliefs, Desires and Intentions;

- the behaviour may be complex; logical rules allow to define in a simple way the complex behaviour that intelligent agents must show;

- the reasoning mechanism is carried out in high level way, i.e. theorem proving;

- CaseLP entities are virtually distributed and communicate in an Agent Communication Language to cooperate and achieve a common goal; even if, for the time being, agents reside on a single processor, a physical distribution may be easily realized.

In our model the communication is carried via message passing, with messages written in KQML [MLF95]. KQML specifies both the messages format and the protocol to handle them. The syntax of KQML messages is based on a list, whose first element is the performative of the message, and the following tokens are the arguments of the performative.

An example of KQML message is the following:

```
(ask-all
 :content "price(ibm, [Price, Time])"
 :receiver stock-server
 :language standard-prolog
 :ontology NYSE-TICKS)
```

This message requires the set of all answers to the query

What is the price of an IBM stock?
The ontology assumed by the query is identified by the token NYSE-TICKS, the receiver is a server identified by stock-server, and the content of the query is written in standard Prolog.

Since the language we use to implement our agents is based on logic programming, the syntax of the messages is slightly different: they are terms with the performative as principal functor and the other tokens as arguments.

KQML distinguishes between the message and the content of the message; the last may be written in any language, from C to Prolog, or any other one. In our first prototype, however, the content language is unique to ensure the homogeneity of the messages and an easier integration of agents implemented by different programmers: the content of a message is always a Prolog term of the form $F(A_1, ..., A_n)$. The following differentiation of the kinds of agents allows to recover the simplification of the KQML model.

To face the applications in which the agents prove really useful, i.e., those where high-level reasoning, sharing of knowledge and integration of information from different sources are required, we individuate at least two types of agents: logical agents, which show capabilities of complex reasoning, and interface agents which only provide an interface between external modules and the agents in the system. It is possible to think of a third kind of agents, which both supply an interface towards external modules and behave in an intelligent way: the hybrid agents. An external module could be for example a Data Base, or a C module which evaluates mathematical functions, or a graphic tool which furnishes primitives to draw lines, polygons and so on. Those software components usually aren’t agents; they provide some functionalities possibly complex, but they aren’t intelligent since they don’t act autonomously, they don’t perceive the environment in which they are working, and they haven’t any social ability: they only answer, in a fixed way, when an agent requires their services. There are some external modules which can be easily integrated in the MAS, namely those for which an Eclipse interface already exists: C, Tcl/Tk, the Eclipse Data Base and obviously the Eclipse modules.

At the moment the agents are written in ACLPL, however the integration of external agents implemented in different languages and communicating via KQML messages is still possible: an interface between our system and the outer world populated by those heterogeneous agents is under study.

In our more general model the agents share a common architecture whose main components are:

- an updatable set of facts, defining the state of the agent,
- a fixed set of rules, defining the behaviour of the agent,
- two mail-boxes for public and private messages and
- an interpreter.

Not all these components are present in all agents: the interface agents include neither the mailbox for private messages nor the state, whereas logical agents don’t include the interpreter. The interface agents receive messages and use an interpreter to translate the content of the message, written in the homogeneous syntax known by every agent, into a syntax recognizable by the component to which it is interfaced; the interpreter interacts with this external component, gets the result from it, translates the result in the standard form comprehensible to the agents, and gives back the translated result to the agent which will provide to forward the result to the sender. On the other hand, the logical agents have a state usually defined by a wide set of facts and show an intelligent behaviour established by complex rules. Figure 1 shows an example of MAS in which a logical agent interacts with two interface agents; dotted lines represent KQML communication.

Let’s now see how an agent behaves when it receives a message. The behaviour of an interface agent is simple: the agents in the system don’t need to know anything about the way the interpreter associated with it works, or if it is interfaced with a C function, a Pascal procedure, an Eclipse module or any other component for which an interface exists. They only need to know that a certain interface agent will provide to them some functionality accessible by sending a message related to the functionality. For example, an interface agent would be able to evaluate the mean of ten numbers when receiving a message ask with content $\text{eval} \text{.} \text{mean}(A_0, ..., A_9)$. Depending on what module is interfaced to the agent, the interpreter would translate this content in a different way. For example, the external component could be a C module exporting a function mean with ten arguments, or a Pascal module exporting the function eval$\textunderscore mean$ with only one argument which is an array of ten elements, or an Eclipse module which exports the predicate $\text{mean} \text{.} \text{value}$ with one argument which is a list of values. The interpreter will translate $\text{eval} \text{.} \text{mean}(A_0, ..., A_9)$ in the appropriate way, depending on the type of the module and the functionalities it provides. The result
evaluated from the module, which is written in the module language, has to be translated back by the interpreter into the common agent content language, before the agent forwards it to the sender.

As far as the logical agents are concerned, the management of a message usually leads the agent to change its state and to forward messages to other agents in the system; the behaviour an agent shows is defined by its fixed set of rules. To change its state, the agent must update the facts of its knowledge base; this is implemented via the update primitives `assert_state` and `retract_state`. The use of update primitives can introduce incoherence in the knowledge base if they appear within a failing goal. To avoid undesirable effects, the system handles the update task in a sophisticated way characterized by two steps:

1. While the agent is handling a message, it records all the state changes, putting them into a temporary buffer.

2. The managing of a message, implemented as proving the message itself, may terminate with a success or a failure; if there is a failure, the user could prefer to skip the updates met while trying to prove the goal, to avoid an incoherence within the agent’s state. In this case the agent simply ignores the updates recorded in the buffer. Indeed, if the message has been handled with success, or if the user desires to make effective the state changes even in presence of a failure, the agent gets the updates from the buffer, and for each of them sends a private message to itself, to perform the update. Before receiving and processing any external message, the agent gets and handles its private messages to update its state in a coherent way.

The user is forced to express his preferences on how the to handle the side-effecting update procedures when he loads the agent into the system.

A problem, similar to the one we have with the side-effecting update primitives, arises when an agent wants to send a message: if the primitive used to send messages is part of a message body whose management fails, the send has to be made effective or not? Also in this case, we let the user choose the strategy which best matches with the intended agent behaviour.

Another main feature of the agent’s behaviour is the way it handles unknown messages. When an agent receives a message, three situations may happen:

1. The agent cannot recognize the performative; for example, an agent which handles only `ask` and `reply` receives a message of type `achieve`. 

Figure 1: A MAS: a logical agent interacts with two interface agents.
2. The agent recognizes the performative, but not the content; for example, an agent which handles only messages with performative ask and content information_about(X) receives a message of type ask with content evaluate_expression(sin(pi)).

3. The agent recognizes both the performative and the content; this doesn’t necessarily mean that it can succeed in achieving the goal issued by the message, since it could understand what it has to do without knowing how to do it.

The behaviour of the agent in these three cases is the following:

1. If the agent doesn’t recognize the performative, it simply ignores the message.

2. If the content C of the message is unknown to the agent, it can ignore the message or reply to the sender forwarding to it a message of type tell with content content_unknown(C). The performative tell is known by every agent by default. It is the user of the system who can decide what kind of behaviour the agent must show in this situation.

3. If both the performative and the content of the message Msg are known by the agent, it tries to prove the goal Msg looking for clauses in the knowledge base whose heads match with Msg, and trying to prove their bodies. Obviously a failure could happen even at this stage. Suppose for example that an agent has been programmed to perform the division between X and Y, if Y is not zero, when it receives the message ask(content(divide(X,Y)), .....). If Y becomes zero, and no clause defines what to do in this case, the management of this message fails even if the agent recognizes both the ask performative and the divide content. It is a task of the programmer to define the behaviour of the agent exhaustively and correctly.

Example

A simple application. This example is very simple; in a real case it would be silly to use the CaseLP system and the MAS technology to solve it: we have chosen it only to show the main features of our system.

Suppose that a user wants to have some information about students and votes of some courses at the University of Genova. The data he is interested in are the best, worst and average votes and students of every course. We may think to have a Data Base which contains the information about the students of the various courses, and their votes. Three C procedures, min, max and avg, are able to evaluate the minimum number in an array, the maximum one and the average of the array values. An application of this type should be simulated using four agents: the agent user, a logical agent which asks information about the courses and handles them for its purposes; the courses information provider, a logical agent which receives the user’s request, and executes a plan to satisfy it; the mathematical function provider, an interface agent which is interfaced with the C procedures min, max and avg, and the courses data provider, an interface agent which is interfaced with the Data Base of University courses.

In our simulation, the user agent is simply capable of sending requests and of receiving messages, without managing them. The courses information provider, indeed, is more complex. It receives a request from the user and, according to the type of request, the behaviour will be different. If user wants to know the names of the best, or worst, or average student of a course, the information provider asks the courses data base provider to know the list of votes of that course; when the answer arrives, it asks the mathematical function provider to evaluate the maximum, minimum or average value of the list; when the mathematical function provider gives the result, it asks the courses data provider again to know who are the students of the course, who got the vote evaluated by the mathematical function provider, and when it receives the list of names, it sends it to the user. If the user only wants to know the best, or worst, or average vote given in a certain course, the behaviour of the information provider is simpler. It asks the courses data provider to know the list of votes of that course; when the answer arrives, it asks the mathematical function provider to evaluate the maximum, minimum or average value of the list; the result provided by the mathematical function provider is sent to the user.

While the interface agents are simple, the interpreters associated with them, which translate the contents of the messages into queries to the Eclipse Data Base, or into C procedures calls, are a bit more complex.

3 Specifying and programming an agent

To give a specification of the behaviour of an agent we have defined a function

$$\text{behaviour} : \text{Agent} \times \text{State} \times \text{Message} \rightarrow \text{State} \times \mathcal{P}(\text{Message})$$

5
The message $M$, received in the state $S$ by agent $A$, drives the receiver in a state $S_1$, and causes it to send the messages $M_1, \ldots, M_n$, iff

$$\text{behaviour}(A, S, M) = (S_1, \{M_1, \ldots, M_n\})$$

To obtain the code of the agent from the specification we translate it into ACLPL, the language we use for programming agents; at the moment this translation is still “hand-made” (and originates ACLPL programs following the scheme we’ll explain below), but we are thinking to automatize this task.

Let’s do some considerations about the static specification expressed by the function behaviour. As far as the communication is concerned, we think this function gives a sufficient detail of specification (the set of messages the agent has to send is given explicitly), whereas the actions the agent has to do when it receives a message (that lead it into a new state) aren’t in general sufficiently defined. In fact, if the actions are limited to control the current state and to update it according to the result of the control, this is easily programmed directly in ACLPL (using the standard scheme we’ll explain below), but if the agent has to perform more complex actions we need a more accurate high-level specification. Further, ACLPL is a logical language, so for a non-logic programmer it can be not so easy to program an agent, even following the standard scheme. For these reasons we are conceiving CaseLP as a multi-language specification environment, in which a set of specification languages (useful for describing different kinds of behaviour) can be integrated using meta-programming techniques; in this way, a specification written in a high-level language could be mapped into an ACLPL executable specification by means of a suitable interpreter for it.

An executable agent is just a simple piece of software written in ACLPL. ACLPL is Eclipse extended with five procedures:

- **receive_one**, to get only one message from the public communication mail-box associated with the agent;
- **receive_all**, to get all messages present in the public mail-box;
- **send(Message, Receiver)** and **send(high_priority, Message, Receiver)** to send a message to an agent, respectively with no or high priority;
- **assert_state(Clause)**, to modify the state of the agent by adding Clause to it, and
- **retract_state(Clause)**, to delete the first clause in the state that unifies with Clause.

When the agent is loaded into the system, using the primitive load described in the following section, its state and behaviour are filled with the facts and the rules defined by the user in the program. The agent gets the structures necessary to receive messages (the two mail-boxes, public and private), and the rules which define its behaviour with respect to the updates, the forwardings, and the unknown messages. All this information is automatically provided: the user must only define what the agent has to do; many of the details of how to do this, are let to the system.

The form of a program defining an agent is the following:

```
activate :-
  <Actions to do when the agent is activated>

  <Initial state of the agent>

  <Behaviour of the agent>
```

The action to do when the agent is activated is to perform the ACLPL procedures **receive_one** or **receive_all**. The state of the agent may be empty if the agent is an interface agent; otherwise it will be composed by Eclipse facts representing the initial situation of the agent. This state will change over time due to update procedures. The definition of the behaviour for an interface agent has the following form:

```
ask(
  content(X),
  ontology(O),
  sender(S),
  receiver(R)
) :-
  interpret(X, Answer),
  Answer != none,
  send(
    reply(
      content(Answer),
      ontology(O),
```
This rule interfaces the agent with its interpreter. The agent and the corresponding interpreter are associated at loading time. The interpreters must have a standard form: they define a procedure \textit{interpret}(X, Answer) in which \(X\) is the content of the message and \textit{Answer} is the result elaborated by the external module and translated into the standard syntax by the interpreter; this result may be \textit{none} if some problem occurs. Notice that, at the moment, only the ACLPL agents are able to use the communication primitives; neither the interpreter, nor the external software modules can do it.

As far as the logical agents are concerned, their behaviour is defined in the following way:

\[
< \text{Rules to handle the messages} > \\
< \text{Auxiliary predicates} >
\]

The rules to handle messages will have the general form:

\[
\text{Performative(} \\
\text{content}(X), \\
\text{Optional Arguments}) :- \\
\text{Body}
\]

\textit{Performative} may be \textit{ask}, \textit{reply}, \textit{tell}, \textit{evaluate}, and every other performative over which there is accordance between the agents. The Optional arguments are usually the ontology assumed by the message, the sender and the receiver. The body is a set of ACLPL predicates: the procedures \textit{assert\_state}, \textit{retract\_state} and \textit{send} will appear among other user-defined primitives.

**Example (cont.)**

**Source code of the agents.** We give some fragments of the code of the \texttt{courses information provider} agent, whose name is \texttt{logic_students}, since it is a logic agent concerning information about students.

**Activation**

\[
\text{activate} :- \text{receive\_all.}
\]

This rule defines what the agent does when it is activated: it gets all the messages in its mail-box.

**Initial state**

\[
\text{request\_identifier}(0), \\
\text{my\_name}(\text{logic\_students}).
\]

Those facts define the initial state of the agent.

**Behaviour**

\[
\text{ask(} \\
\text{content(best\_student(Course)),} \\
\text{ontology(students),} \\
\text{sender(S),} \\
\text{receiver(R)} \\
\text{)} :- \\
\text{request\_identifier(Id),} \\
\text{retract\_state(request\_identifier(Id)),} \\
\text{Id1 is Id + 1,} \\
\text{assert\_state(request\_identifier(Id1)),} \\
\text{assert\_state(associated(Id1, best\_student(Course), S)),} \\
\text{send\_ask(} \\
\text{content(votes(Course, Id1)),} \\
\text{ontology(students),} \\
\text{sender(R),} \\
\text{receiver(interface\_db) ),} \\
\text{interface\_db).}
\]

This rule defines the behaviour of the agent when it receives a message requiring information about the best students of the course \textit{Course}; the agent gets the last number used to identify a request (line 1 of the body), updates the state with the information of the new used identifier (lines 2, 3, 4), associates this to the current request (line 5) and sends to the agent \textit{interface\_db} a message in which it asks information about the votes of the \textit{Course} students.
The code of the interface agents is very short, and it is the same for both the courses data provider, called *interface_db* and the mathematical function provider, called *interface_c_math*. We don’t write it here, since it is the code written when describing the definition of the behaviour of an interface agent. As far as the activation is concerned, when activated the interface agents call the predicate `receive_all`. As an example of the code of the interpreters, let us show about half code of the interpreter associated to the *interface_db* agent:

```
interpret(votes(Course, Id), votes(List, Course, Id)) :-
    use_module(db_functions),
    ((Course = artificial_intelligence; Course = data_bases);
    Course = operating_systems),
    query5(List, Course).

interpret(votes(Course, Id), votes([], Course, Id)).
```

where `query5` is a procedure exported by the Eclipse Data Base, loaded in the module `db_functions`, which we have built for this application.

The interpreter associated with the *interface_c_math* agent is:

```
:- external(max/2, p_max).
:- external(min/2, p_min).
:- external(avg/2, p_avg).

interpret(min(List, Id), min(M, Id)) :-
    min(List, M).
```

... similar clauses for max and avg

The first three goals are used to make available to the interpreter the procedures `max`, `min` and `avg` which are mapped into the C procedures `p_max`, `p_min` and `p_avg`.

## 4 Building and executing the specification of the MAS

Now that we have learned how to build an agent, we would like to insert it in a context in which other agents are present, and to start the simulation of the system when all the agents have been loaded into it. Two primitives are provided for this purpose: `load` and `initialize`.

The first primitive is used to insert an agent into the system. The arguments of `load` are six:

- **FN**: name of the file containing the source code of the agent.
- **AN**: name to assign to the agent.
- **Int**: name of the file containing the source code of the interpreter associated with the agent; if it has no interpreter associated because it is a logical agent, the value of this argument must be set to `none`.
- **Flag ignore unknown message**: `ignore_unknown_message` and `tell_unknown_message` are the two values it can assume. In the first case, when the agent receives a message whose content is unknown, it will ignore it. In the second case, instead, the agent will forward a message to the sender, telling that it didn’t recognize the content.
- **Flag update only if success**: it can assume two values: `update_success` or `update_however`. In the first case, the side-effects tied to the primitives `assert_state` and `retract_state` are made effective only if the management of the message succeeds. In the second case, all the `assert_state` and `retract_state` met while trying to handle a message will be done.
- **Flag send only if success**: in a similar way as for the preceding flag, `Flag send only if success` may assume the values `send_success` or `send_however`: if it is `send_success`, the requests of sending a message are satisfied only if the management of the message succeeds. Otherwise, all the `send` met while trying to prove a goal are made effective.
After all the agents have been loaded, the simulation of the obtained MAS can start. It will be necessary to initialize the mail-boxes of some agents putting some messages into them. The primitive `initialize` allows the user to do this. It has two arguments: the first is a list of couples (Message, Receiver) and triples (high priority, Message, Receiver), which defines what message has to be put in what mail-box, and if the message has high priority. The second argument of `initialize` is the name of the file in which the messages exchanged during the simulation are to be saved: this permits an off-line control of the evolution of the system. A synthetic form of every message exchanged is however printed on the video at run time.

The primitive `initialize`, beside filling some mail-boxes and defining the name of the output file, starts the simulation: every agent is in turn activated and the actions contained in its `activate` definition are performed.

The detailed description of how the simulation is carried out is contained in Section 5.

After the simulation has been completed, the user can analyze the messages exchanged using the primitive `see_output`. The only argument of it is the name of the file in which the messages have been saved (it is the name used as second argument of `initialize`). This primitive allows the user to see all the messages sent by an agent, or all the messages received by an agent, or all the messages exchanged between two agents, or the whole set of messages exchanged during the simulation. This off-line check has proven useful to trace the execution of the system and to get more information about it, since during the simulation only a part of the messages (its performative, the sender, the receiver and the main functor of the content) is displayed, while the off-line check displays the complete messages.

**Example (cont.)**

**Loading the agents and inspecting the output.** The first thing to do to execute the specification is to make available the CaseLP primitives, compiling the source of the system with the command:

```
use_module(caselp).
```

Then the agents must be loaded and the external modules compiled:

```
load('agents/logic.pl', logic_students, none, 
tell_unknown_message, update_success, send HOWEVER),
load('agents/user.pl', user, none, 
tell_unknown_message, update Scrolls however, send Success),
load('agents/interfac.pl', interface db, 'interpr/intpl.db.pl', 
tell_unknown_message, update however, send Success),
load('agents/interfac.pl', interface c math, 'interpr/intpl.math.pl', 
tell_unknown_message, update however, send Success),
compile('interpr/db.pl').
```

The file `interpr/db.pl` contains the code for the Eclipse Data Base, with the definition of our procedures `query1`, . . . , `query6` to access to the information contained in the students Data Base. To start the simulation, the `initialize` primitive must be used, for example with content:

```
initialize([ 
  (ask( 
    content(best student(data bases)), 
    ontology(students), 
    sender(user), 
    receiver(logic_students) 
  ), logic_students 
), 
  output 
]).
```

While the execution is running, messages of type

```
agent user == agent logic_students
REQUEST: best student / 1
```

appear on the screen. After the execution has stopped, it is possible to see the complete messages exchanged using `see_output`; it is possible for example to ask for all messages received by `user`. In this case the answer is:

```
reply(
  content( 
```

9
5 Implementation of CaseLP

To implement an agent as a logic theory, we have used the Eclipse Module System, which allows to define a module (i.e., a set of clauses) and to specify the interface of this module towards other modules (i.e., the procedures that the module exports and the ones it imports). The module which defines the procedures described in the previous section, load, initialize, see_output, and the other ones, is called caselp and is contained in a file named caselp.pl. The user must compile this file to make available all the primitives he needs. Also the agents are loaded into the system as Eclipse modules. This is completely transparent to the user, who writes the agent source code in a simple way without having to know anything about Eclipse modules.

The translation from the simple ACLPL source code into a complex module, which contains not only the user-defined clauses, but also system clauses which define the behaviour of the agent relatively to assert_state, retract_state, send, and the messages with unknown content, is made by the load(FN, AN, Int, Flag_ignore_unknown_message, Flag_update_only_if_success, Flag_send_only_if_success) primitive.

The main steps of load are:

1. Write on a file temp_file all the definitions which characterize it as a module with name AN. The procedures which this module exports are activate_AN, message_AN, and other ones. The procedures it imports from the main module caselp are the ones for handling the state updates, and for managing public and private communication.

2. Every clause in the file FN is read; the occurrence of the string activate in FN is substituted by activate_AN in temp_file, also the occurrences of receive_one, receive_all, assert_state(Clause), retract_state(Clause) and send(Message, Receiver) are replaced with the appropriate system primitives imported from caselp.

3. The heads of the clauses handling messages become arguments of a predicate message_AN. For example the clause
   ask(Arguments) :- Body.
   becomes
   message_AN(ask(Arguments)) :- Body.

4. A new clause
   message_AN(Message) :- Body_unknown_content.
   is written on the file temp_file to define the behaviour of the agent in case the Message content is unknown.

5. A new clause
   message_AN(.)
   is written on temp_file after all the other clauses defining the message_AN predicate, to establish the behaviour of the agent in the case that it receives a message with unknown performative: the goal message_AN succeeds, avoiding a failure which would jeopardize the system’s running, even if the management of the message is considered failed.

6. The code which defines the behaviour of the agent pertinent to assert_state, retract_state and send is written on temp_file, according to the last two arguments of load.

7. The file temp_file is closed and then compiled to effectively insert the agent AN into the system.

The core of the system relies on this complex translation performed by load.

After all the agents have been loaded into the system, the user calls the primitive initialize to initialize the mail–boxes of some agents and to start the simulation. The first goal is simple to achieve: for every
message $Msg$ to put into the mailbox of $AN$, the fact $\text{mailbox}(AN, \text{List})$ is retracted, $Msg$ is added to $\text{List}$, obtaining $\text{New\ List}$, and $\text{mailbox}(AN, \text{New\ List})$ is asserted. Every message has a random time stamp associated, which represents the moment in which the message arrives into the mailbox. The system has a global clock: only those messages whose time stamp is less or equal to the global time can be received and managed. The random time stamp is associated with a message when it is sent by an agent or when it is created by the user with the $\text{initialize}$ primitive; it is evaluated by adding a random number between 1 and 10 to the actual global time. It is used to simulate possible random delays in the exchange of messages due to the transport mean and to other factors. If the message has been sent using the version of the performative $\text{send}$ with the argument $\text{high\ priority}$, the time stamp associated with it isn’t a number but the atom $\text{high\ priority}$: the message is handled before any other one in the mailbox. If many messages in the mailbox have high priority, they are handled respecting the order in which they arrived.

The goal of starting the simulation requires the use of a scheduler: we use a round-robin scheduler which recursively calls a predicate $\text{activate\ one\ agent}(\text{Actual}\ num, \text{Num}, \text{Flag\ empty\ mail\ boxes})$.

$\text{Actual}\ num$ is the integer identifier associated with the agent (the first agent loaded into the system has identifier 1, the second 2, and so on), $\text{Num}$ is the total number of agents present in the system (it is used to pass from the agent $\text{Num}$ to the agent 1) and $\text{Flag\ empty\ mail\ boxes}$ is used to verify the termination condition: if for a complete round of the scheduler, all the mailboxes are empty, the simulation stops. The $\text{activate\ one\ agent}(\text{Actual}\ num, \text{Num}, \text{Flag\ empty\ mail\ boxes})$ predicate, checks if there are messages in the mailbox of the $\text{Actual}\ num$ agent, if there are any, finds the name $AN$ of the agent, and calls the predicate $\text{activate\ AN}$ exported by the module $AN$ created while loading the agent. The body of this predicate contains the $\text{system\ receive\ one}$ or $\text{system\ receive\ all}$ primitives. They inspect the mailbox of the agent looking for messages whose time-stamp is less or equal to the system time. If the messages found in order of increasing time-stamp (the messages with high priority come first) are $\text{Mess}_1$, ..., $\text{Mess}_n$, $\text{system\ receive\ one}$ will handle the first of them by calling $\text{message\ AN}(\text{Mess}_1)$; this message is deleted from the mailbox, and the other ones will be managed at the successive scheduler round. The primitive $\text{system\ receive\ all}$, indeed, will manage all the messages together, by calling $\text{message\ AN}(\text{Mess}_1)$, ..., $\text{message\ AN}(\text{Mess}_n)$. The handled messages are deleted from the mailbox. Notice that after a message has been handled, the agent inspects its private mailbox to perform those updates required by the message, before managing any other one.

After an agent has managed its messages, the scheduler passes to the following agent. The scheduler activity stops when all the mailboxes of the agents are empty: if no agent has messages to handle, no agent will do anything, and the simulation runs out.

The lines of the Eclipse code of our system are about 1900 and we developed it in about a person-month; we had already developed a previous prototype which didn’t use Eclipse modules but programs realized with meta-programming techniques, along the lines of KBMS1 [MCH*90]; it helped us very much in pointing out the intrinsic problems and the desirable features of a tool for the executable specification of complex applications.

Example (cont.)

Building a module from an agent. Let’s see what happens when for example the predicate

\[
\text{load(‘agents/interfac.pl’, interface\ db, ‘interpr/inter\ db.pl’,}
\text{tell\ unknown\ message, update\ however, send\ success),}
\]

is called. A new file ‘temp.file’ is created and then compiled. A fragment of its code follows:

Agent as a module
\[
\text{:- module(interface\ db).}
\text{:- export activate\ interface\ db/0, activate\ private\ messages\ interface\ db/0,}
\text{activate\ send\ messages\ interface\ db/0, message\ interface\ db/1.}
\text{:- begin\ module(interface\ db).}
\text{:- use\ module(caselp).}
\text{activate\ interface\ db :- system\ receive\ all(interface\ db).}
\]
message_interface_db(  
  ask(  
      content(X, ...) ) :-  
  system_assert_current_goal(interface_db),  
  interpret(X, Answer),  
  !(Answer = none),  
  system_send(  
    reply(  
      content(Answer),  
      ........),  
    ........),

Predicates to define the agent behaviour w.r.t. the updates  
activate_private_messages_interface_db :-  
system_private_messages([Private_messages, interface_db]),  
system_succeeding_goal(N, interface_db),  
satisfy_update(Private_messages, N).

Associated interpreter  
Code of the associated interpreter as it is.

6 Two real-case applications

The two real-case applications we present show some common features:

- they involve distributed entities which don’t have a common knowledge;
- there isn’t any global design in the architecture of the entities, which are autonomous and with specific capabilities;
- the entities must cooperate to exchange information for developing a plan which is used to achieve a goal.

Because of these features, it seemed very appropriate to model the applications as Multi-Agent Systems. Let’s see, more specifically, which problems have been addressed.

6.1 Planning goods transportation

The application example we present has been realized in cooperation with Elsag–Bailey, an Italian company that operates on international scale providing services automation.

We propose a model for the problem of planning goods transportation: a client has to transport goods from a place to another, specifying the hour in which the goods must leave, and choosing whether to save money or time. He doesn’t know how the great companies of delivery (Posts, Federal Express, DHL, ...) are organized and which transporters are present on the territory.

Concerning the methodology for realizing an application we use at the moment an informal 4-steps development process; in the following we show how this methodology can be applied to planning goods transportation.

Step 1: identification of the set of agents the application needs and their interconnecting structure. We identified four kinds of agents for this application: client, agency, distributor and transporter. The first, which has to send goods, can communicate only with agency. This one communicates with clients and distributors, to solve the client’s problem. The third is involved in the delivery of goods; it exchanges messages with agency, other distributors, and transporters. Finally, the last carries out the physical transport and can only communicate with distributors. All these agents are logical agents, the application we developed didn’t need any kind of software integration.

Step 2: choice of the communication protocol among each pair of communicating agents. The messages recognized by all the agents identified in step 1 are the performatives ask, reply and the default tell.

Step 3: specification and programming of the agents. In this step we had to define the behaviour of the agents. We first gave a static specification of the agents using an instance of the general function behaviour presented in Section 3 and then we translated this functional static specification into ACLPL programs. The translation was easy and natural, and we took few person-weeks to do it. Thanks to the
execution of the system we found some mistakes in the agent specification, and we could correct them. The stage of correction and improvement of the system took about one person-month.

The behaviour of the agents is the following. The client is simulated by the system user, that puts his request in the mail-box of agency, using the primitive initialize.

Agency receives a query from the client, forwards it to one of the distributors present in the departure place and when this distributor answers, telling if the delivery was successful or not and giving informations about the found path, it sends a message containing those results to the client.

A distributor covers a part of the territory in which places are connected by virtual tracks (air, train, road, . . . ); working over those tracks, there are some transporters which can provide information and services for the transportation. Some places are within the domain of more distributors; in this places an exchange of goods may happen. Think for example of the posts of European countries: they are the distributors while the national airline and railway companies are the transporter. In the French city of Lille would happen the exchanges between the Posts of France and Belgium, in the Italian city of Ventimiglia the exchanges between Italian and French Posts would take place, and so on. The distributor receives a request from agency or another distributor, checks if it can reach the arrival place by itself, consults the transporters to get the details of departure times, costs and place availability, and involves the neighbour distributors if the place to reach is not in its zone. For example, the French Post distributor which has to transport a parcel from Paris to Bruxelles, can’t reach Bruxelles by itself; it has to contact the neighbour distributors with whom it shares a common place where the parcel may be swapped, asking to them if they would be able to bring the parcel to Bruxelles. If the agent has to involve other distributors, it doesn’t answer to its applicant until all the distributors it involved have answered. Then it provides to the applicant all the information it got both from the transporters it consulted and from the distributors. Instead, if it can get to the final place, it answers to the applicant providing all information he got from the transporters.

If the distributor had been contacted by agency, and not by another distributor, when it obtains all the information needed, it must choose which is the best way and reserve the tracks on it, sending messages to the transporters present in its zone and to the distributors eventually involved. Only when it receives all the answers about the successful reservation, it answers to agency that the mission was successful. Going on with the European Posts example, if the French distributor had been contacted by agency to bring the parcel from Paris to Bruxelles, after all the neighbour distributors have answered making the information available to individuate various possible itineraries, for example Paris–Lille–Bruxelles, Paris–Marseilles–Ventimiglia–. . . –Bruxelles, and so on, the distributor chooses the one that minimizes time or cost, and tries to reserve it. If for example the best way is Paris–Lille–Bruxelles, the French distributor will reserve the track Paris–Lille, sending a message to the right transporter, for example the French railway company, and will ask to the Belgium distributor to reserve the track Lille–Bruxelles. Obviously, it can happen that places available at the moment of the retrieval of information, are no more available in stage of reservation: concurrency of missions of many clients is supported and since the strategy is inform–reserve, while getting information no kind of booking is made. In this case the distributor sends messages to delete the reservations already made, and tries to reserve a sub-optimum way. If reservations fails on all the roads found, or if no road was found, the distributor informs agency that the mission failed.

The final agents in the MAS, the transporters, have a wide knowledge about all the details of time and costs of the different tracks, but they don’t perform complex reasoning; in fact, they give only information to the distributors and reserve places for them.

**Step 4: executing the specification.** About this step much has been said before; to give an idea of the complexity of the problem, it is sufficient to say that in an example with five concurrent missions, more than 100 messages have been exchanged.

### 6.2 La Spezia Inter–modal System

This application has been realized by Robert Ziu for his graduation thesis [Ziu97] at the Computer Science Department (DISI) of the University of Genova, in collaboration with FS, the Italian railway company. The problem is more specific than the one described before; it concerns the specialized traffic of containers involving the Italian port of La Spezia, the terminals in the north of Italy, and the railway line La Spezia–Milan; the aim of the thesis was to develop a model of this area with the purpose of handling the dispatching of trains, evaluating the performances of the railway service, providing the terminals and the goods stations of La Spezia with the scheduled arrival of trains, and providing the person in charge for La Spezia goods stations with a decisional support for the management of the wagons on arrival. The steps followed to identify and develop the Multi-Agent System specification are the four ones individuated in our methodology:
Step 1: identification of the set of agents. The agents working in the system are of four kinds: generic terminal, GIOC (Goods Inter-compartment Operative Coordinator), La Spezia production assistant, and section agent. The first agent exchanges messages with the agent of its section and GIOC which exchanges messages with all the other agents in the system. La Spezia production assistant exchanges messages with the agent of the section of Genova and with the GIOC; the last agent communicates with the terminals of its section and with GIOC. Also in this application the agents are logical ones.

Step 2: choosing the communication protocol. The KQML performatives used in this application are ask, reply inform and the default tell. There are also three performatives application-dependent: send modifications, reply confirmation and when arrive the train.

Step 3: specification and programming of the agents. The behaviour shown by the agents is the following: the main problems of the generic terminal concern obtaining the license for a loading, and knowing whether it is possible to dispatch a train over the line La Spezia–Milan or not. The request of the license is sent to the GIOC, while the dispatching is handled in collaboration with the section agent. When the section agent receives the request from the terminal, it checks the situation evaluating the congestion index (number of trains per Km). If accepting the new train doesn’t bring the index over the critical level, the dispatch is accepted, otherwise GIOC is involved to solve the problem. GIOC has a global vision of the situation, and so it can evaluate the effective impact of the dispatching of a new train, accepting or refusing it according to this global knowledge. As far as the license to loading is concerned, GIOC checks the scheduled traffic of the section to which the terminal belongs, for the day in which the train has to be sent. It evaluates the congestion index for that day, and refuses the license if the critical level is exceeded. Otherwise, it forwards the request to the La Spezia production assistant which checks the goods station availability to receive the train; the parameters considered by the agent are the number of trains present at La Spezia, the number of trains arriving from the terminals, the number of existing tracks at La Spezia goods station and the average flow of trains moving from the goods station to the port. When La Spezia production assistant answers to the GIOC, it forwards this answer to the terminal which had requested the license. A particular kind of agent is the section agent of Genova: it can exchange messages directly with the production assistant of La Spezia, to have informations about trains.

Step 4: executing the specification. This step has been realized with little modifications to the scheduling policy of CaseLP and it allowed to simulate this dynamic application in which trains move from one place to another while agents are exchanging messages. A graphic interface application-dependent has been provided, using the ProTeXL interface, which enriches Prolog with functionalities of the X11 library Xlib and Tcl/Tk. We are now working to make this graphic interface independent from the application.

7 Related and future work

This sections concerns both the existing techniques for the design of MAS and the applications of logic programming in realizing agents and MAS; further, some considerations about future extensions of our CaseLP system are presented.

As we said in Section 1, methodologies and systems for designing MAS and developing agent–oriented application are still quite rare, even if more and more researchers have begun to work in this field in the last period. About design methodologies the research work is really in its infancy. We can cite a paper by Kinny and Georff [KG96] in ATAL '96 regarding a methodology for the design and specification of large–scale commercial and industrial software systems based on MAS (modelled extending object oriented techniques) and a book by Müller [Mü96], perhaps the first monograph about theory and practice in designing MAS. Regarding systems for the specification of agent–based applications, one of the first systems that have been developed is DESIRE [BDKJT96], a declarative compositional modelling framework to model MAS: it is based on the concept of compositional architecture, in which a complex process is designed as a set of interacting task-based hierarchically structured components, i.e., agents; the conceptual design of a system considers both static and dynamic aspects, formalized using temporal logics. Other research works regard the realization of environments for the construction of MAS based on specific architectures: we recall MIX (a multi-agent architecture conceived as a general purpose distributed framework for the cooperation of multiple heterogeneous agents) and [BF95] that presents a shell for building Multi-Agent Systems that provides reusable languages and services for agent construction.
We can roughly classify applications of logic programming for realizing MAS in two categories: researches investigating the use of logic languages for studying theoretical models for MAS, in which the properties of the systems can be proven using theorem proving, and works addressed to exploit logic programming in the realization of applications. Belonging to the first group there are the works by Beyssade et al. [BEL95], dealing with a logic-based model of MAS, in which agents are higher-order logic processes, Lesperance et al. [LLL+95] that present CONGOLOG, a concurrent multi-agent programming language based on a logical theory of action, and Rao’s AgentSpeak(L) [Rao96] which uses a restricted first-order language with events and actions to model the internal mental state of an agent and its behaviour.

It may be interesting to compare CaseLP with CONGOLOG. The main difference lies in the presence of a global state in CONGOLOG, which is not present in CaseLP, where every agent has its own state. It is however possible to extend CaseLP with a global knowledge base which can be used as a blackboard for the agents: from an implementation point of view it is trivial to do this, and it would allow the agents to share a common view of the world state. Another main difference is that the simulation in CONGOLOG is carried out proving that with certain precondition axioms, a program which defines the behaviour of the agents, and an initial state, it is possible to reach a final state. The obtained final state encapsulates all the information about the intermediate states and the actions performed in them by the agents. CaseLP approach is more focused on the behaviour of agents as independent communicating programs.

As far as AgentSpeak(L) is concerned, it is a specification language consisting of a set of base beliefs and a set of plans; even if these components resemble respectively CaseLP facts and rules, there are some differences: in AgentSpeak(L), the head of a plan consists in a triggering event and in a context defining when the event may activate the plan, while the body is a set of actions or goals. In CaseLP (and in logic programs in general) the head of a rule is a goal like the ones in the body. AgentSpeak(L) is more general than CaseLP because the event that causes the execution of a plan is whatever triggering event, while up to now in CaseLP it is only the reception of a message that activates an agent. Moreover AgentSpeak(L) agents can be interrupted while executing a plan (they can be viewed as multi-threaded interruptible logic programming clauses), while CaseLP agents can’t. It is very interesting in Rao’s paper the proof theory of the language, given as a labeled transition system: it leads to a one-to-one correspondence between the proof rules and the operational semantics of the language.

About exploiting logic programming to realize applications based on MAS, we can cite Schroeder et al. [SdAMP96], which present a formalism for specifying and implementing diagnoses agents based on extended logic programming, and MAGIQUE [BM95], a hybrid architecture for MAS that permits an integration between the blackboard model and autonomous agents and that utilizes an object oriented technology. There are many similarities between Schroeder et al. approach and our’s: in particular the steps which characterize the agent’s behaviour are the same: read the current message, update the state (which in Schroeder et al. approach is defined in terms of mental attitude, unlike in CaseLP) and execute some actions; the main difference between the two approaches lies in how the actions to perform are found: in the first case a meta–predicate demo(inconsistent, Actions) finds out all those actions which don’t introduce inconsistency in the model of the world when performed, while in our model the actions are all the ones defined by the succeeding rule. Another similarity, which shows the common aim of Schroeder et al. work and our’s, lies in the output of a simulation: both the environments help their users in understanding the behaviour of a Multi-Agent System by performing a simulation and showing the messages exchanged between the agents during the computation.

Between these two categories, CaseLP can be classified as belonging to the works which want to exploit logic programming in the realization of applications, with a particular attention to those in which integration of heterogeneous existing software and data is needed.

An interesting example of integration methodology is presented in HERMES [SAAB+95], a system for integrating informations from diverse sources; the possibility to couple HERMES and CaseLP is under study. Many other extension to CaseLP are possible: as explained in Section 2 we are studying how to integrate external communicating agents in our system (to permit a greater flexibility of utilization) and how to make available multiple specification languages for defining the behaviour of the agents. Finally, the use of logic programming–based agents for Internet applications is a new exciting area of research [TDBH96]: another interesting issue will be to investigate the potentialities of CaseLP in this direction.
References


