Towards Multi–Agent Software Prototyping

Maurizio Martelli, Viviana Mascardi, Floriano Zini
D.I.S.I. - Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
email: {martelli, mascardi, zini}@disi.unige.it

Abstract

Integration and reusing of different kinds of information and software tools is a pressing necessity that more and more complex applications have to cope with. This fact and the distributed nature of many applications made it very appealing to use multi–agent technology. However, agent–based software still lacks well founded development methodologies, thus rapid prototyping and executable specifications could be very important for the realization of these applications.

We present CaseLP, a specification framework for agent–based complex applications founded on Logic Programming. Many of the desirable 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 case studies that is, real–world applications that have been specified using CaseLP.

1 Introduction

The number of different software applications that we meet in our everyday life is dizzily increasing, but nowadays two aspects seem to be fundamental for new software products: distribution and integration. The first aspect concerns the more and more distributed nature of software, due both to its increasing complexity (that requires a computational power easier to find in a distributed programming environment) and to the many physical sites where an application may access the resources it needs. The second issue concerns instead the integration of existing software (legacy software) into new applications; due to (obvious) economical reasons the reusing of existing software for solving more complex tasks has a great relevance for the success of new technologies. In this paper we focus on Complex Distributed Software Applications (CDSAs). Such applications can be defined as software products that operate in (logically) distributed environments, integrate existing software with new components and coordinate the interactions among them for executing a global task.

The realization of reliable CDSAs is certainly a fundamental goal for software designers. The final product should suit, as much as it is possible, the initial requirements given by the customer and for this intent we need instruments that can be used to assure that the final product is what we wanted at the beginning. Even for simpler applications this request cannot be satisfied by the major part of traditional Software Engineering techniques [Pre94] and also applying Formal Methods [Hol97] to the specification of complex
systems is not easy. If we cannot follow well established techniques for realizing a CDSA as an end product we can, more realistically, think about a well founded realization of a prototype of the final product. In fact, a software prototype is generally a simplified and/or not very efficient version of an end product, that is developed for testing and (if necessary) modifying the initial requirements without much effort and cost. In a prototype we do not need a great efficiency, therefore we can realize it using more formal but less efficient technologies.

Multi–Agents Systems (MAS) [WJ95] provide abstraction capabilities that can be very useful for CDSAs prototyping. Firstly, they allow a cognitive vision of the system that has to be specified. Secondly, they provide an high level tool for developing applications in which many autonomous components interact to coordinate themselves and to share and integrate knowledge. (Notice that the behaviour of some components cannot generally be specified in a deterministic manner, but it can vary depending on incoming information, their internal state, the changing of their goals as the result of interaction with other components and/or perception of the external world. In other words, the components behaviour can be reactive and/or pro–active.) Finally, MAS facilitate a compositional approach to the prototype development, favouring the logical decomposition of a task and the subdivision of the resulting sub–tasks among the different agents composing the system.

Even if many agent–based applications have been realized, most of them has been developed using a problem–dependent approach, resulting into many incomparable models of MAS, difficult to verify using formal established approaches. Therefore, a major challenge for research on MAS is the establishment of standard, general purpose techniques that can be applied to the definition of agent–based software. This issue is twofold: on the one hand we need comprehensive methods for all phases in the definition of agent–based applications, from requirements to the final implementation; on the other hand the definition of a generic agent model could be very useful for realizing real problem–independent MAS architectures. In this way, an agent can be seen as a skeleton (defining its static structure), that can be filled by behavioural rules (according to the task it has to accomplish) and communication rules (according to the communication protocol the agent is undergone). Such approach can enhance reusing of both the static and dynamic structures. In fact, most of the agents of a MAS could share the same skeleton, and behavioural and communicative patterns could be reused for different agents.

We aim at defining a specification framework for CDSAs prototypes modelled as MAS [MMZ97]. Our base is essentially Logic Programming (LP) [Llo87]. LP can be considered (as it is well known) a powerful and executable specification language and executable specifications can be seen as prototypes. However, CDSA prototypes can assume notable dimensions and directly specifying a CDSA using LP is not proposable: it lacks sufficient expressive power for defining the specifications we need. Nevertheless, LP itself can be an easy way to implement MAS [KS96] thus, combining a high level agent–based modelling with a logic–based implementation, we can obtain a powerful tool for the definition of prototypes. On the other hand, an important issue concerns the approach used for developing an application based on MAS: until now, almost all the applications have been developed in a bottom–up manner, resulting into many problem–dependent incomparable models. In the last few years, issues concerning the production of distributed software has been addressed by Distributed Software Engineering (DSE) [Kra94]; we deem that using technologies rising from this field can be extremely useful for enhancing standardization.
of the production of agent–based software.

Our final aim is to define an environment for CDSAs prototyping which addresses the following issues:

1. Providing a methodology for the development of complex software applications realized as MAS.

2. Providing a homogeneous way to specify the prototype, abstracting from the implementation of single components.

3. Defining a general agent architecture, to allow standardization and reusing in the development of applications based on MAS.

4. Investigating how heterogeneous software modules and data (indispensable in real–world application) can be integrated into a prototype.

5. Providing a set of specification languages for the definition of the agent behaviours as well as a methodology to translate/integrate them into a unique executable specification.

6. Providing automatic code generation tools for obtaining an executable prototype from its specification.

7. Providing testing tools for proving the specified prototypes.

In this paper we address some of the issues above presenting CaseLP (*Complex Application Specification Environment based on Logic Programming*), a prototyping environment for CDSAs. We are implementing our system using the CLP language ECLiPSe [ACD+95]. As we said above, we chose a logic programming paradigm because logic programs are executable specifications. Further, it easily allows to program agents that maintain a symbolic description of the world and to perform high–level reasoning over this description. Finally, 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 using a logic programming paradigm. As it is well known, standard logic programs lack adequate modellization of state changes, usually dealt with using extra–logic primitives as `assert` and `retract` that do not have a well–defined declarative semantics. However we believe it is possible to explore the semantics of a system involving dynamic agents by means of *Linear Logic* [Del97].

CaseLP is an ongoing project and we will describe a first real implementation of it; the system 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 paper is organized as follows. 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; in Section 3 the prototype developing methodology is described, whereas Section 4 concerns the specification of the single agents and ACLPL (*Agent Constraint Logic Programming Language*), the language we have defined for programming the agents; Section 5 describes how the agents can be inserted in a context of a MAS and how its execution is performed. 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 Agent architecture

A generally accepted idea about MAS is their capacity of abstracting from details and using this abstraction to model, explain, and understand complex real-life situations. **CaseLP** provides an agent architecture useful for describing in a simple and high-level manner the heterogeneous and distributed entities involved in a particular application; it also furnishes a built-in control mechanism (goal-directed theorem proving) that allows the diverse components to engage in dialog (using an Agent Communication Language [GK94]) and to negotiate transfer of information. Moreover, it gives us the potential to specify entities including representations of other entities, which is a crucial feature for agents that must cooperate and coordinate themselves.

In our model agents communicate via point-to-point message passing, with messages written in KQML [MLF95]. KQML specifies both the messages format and the protocol to handle them. Since the language we use to implement our agents is based on logic programming, we slightly changed the syntax of the language with respect to the standardization [LF97]: in our model the messages 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 the first prototype of our system, 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 an ECLiPSe term of the form $F(A_1, \ldots, A_n)$. The following differentiation of the kinds of agents allows to recover the simplification of the KQML model.

We consider 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 perform high-level reasoning: the **hybrid agents**. A common feature of these three kinds of agents is *awakeness* that is, once they have been activated, they remain always ready to answer any external request. 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 external components are not agents; they provide some functionalities possibly complex, but they are not awake, do not perceive the environment in which they are working, and do not have 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 and Knowledge Base and obviously ECLiPSe modules.

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
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 do not include the interpreter. Figure 1 shows an example of MAS in which a logical agent interacts with two interface agents; dotted lines represent KQML communication.

Figure 1: A MAS: a logical agent interacts with two interface agents.

At the moment our system allows us to define awake reactive agents: every agent is activated at the beginning of the prototype execution and remains active until the end of the simulation. The behaviour of the agents consists essentially in reacting to an incoming message, but this behaviour varies according to the kind of agent.

The behaviour of an interface agent is simple: any other agent in the system does not 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 particular message. Further, the simple tasks performed by interface agents do not require that they have a state. An example of interface agent could be an agent which is able to evaluate the mean of ten numbers when receiving a message ask with content \texttt{eval\_mean(A0,...,A9)}. 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 \texttt{mean} with ten arguments, or a Pascal module exporting the function \texttt{evaluate\_mean} with only one argument which is an array of ten elements, or an ECLiPSe module which exports the predicate \texttt{mean\_value} with one argument which is a list of values. The interpreter will translate \texttt{eval\_mean(A0,...,A9)} 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 \texttt{assert\_state} and \texttt{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 state of the agent. 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 a preference on how to handle the side–effecting update procedures.

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 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 \texttt{ask} and \texttt{reply} receives a message of type \texttt{achieve}.

2. The agent recognize the performative, but not the content; for example, an agent which handles only messages with performative \texttt{ask} and content \texttt{information\_about(X)} receives a message of type \texttt{ask} with content \texttt{evaluate\_expression(sin(pi))}.

3. The agent recognizes both the performative and the content; this does not 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 does not 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 \texttt{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.

With respect to the desirable characteristics of an agent architecture, presented in Section 1, our agents are still a little primitive. Communication actions are embedded into the rules defining the behaviour of the agents, task control is fixed (an agent cannot choose anything but reactively answer to incoming messages). However, we believe this architecture is a good starting point: it has suited very well the application we have realized and it is easily extendible. For instance, we can enhance task control without much effort, defining a meta–interpreter embedding different strategies of behaviour or communication.

3 Specification methodology

The approach we use to obtain and test an executable specification mimics the classical development cycle for software prototypes [Pre94] and can be summarized in the following steps:

1. Identification of the set of agents and their interconnecting structure. In this first step the specification developer decides the static structure of the system and chooses the kind of agents, as well as the interconnection topology among them.

2. Choice of the communication protocol among each pair of communicating agents. This step consists on choosing the communication protocol between each pair of connected agents that is the set of KQML performatives they use during the communication.

3. Specification of the behaviour of each agent in the system. The developer specifies the behaviour of the agents: what they are able to do and how they perform it. In a system there can be many instances of a same kind of agent; if this is the case, in this step the behaviour can be defined only once for each kind and then reused for all instances. Further, in this step (possibly) different specification languages can be used, choosing the most appropriate for describing the intended behaviour of the agent.

4. Implementation of the prototype. In this step each agent specification is first translated into executable code. Then the MAS is built, making a unique executable specification embedding all defined agents.
5. **Execution of the obtained prototype.** The last step tests the implementation choices, checking if the system behaves as expected. Any specification error or misbehaviour discovered in this step may imply a revision of the choices performed in the first 3 steps.

At the moment we do not use any completely automatic tool for making an executable prototype but, as we will see in the following sections, the development process we have defined is rigorous enough for allowing a not difficult automation.

**Example: student data retrieval**

This example is very simple. We have chosen it as a running example to show the main features of our system.

**The problem.** Suppose a user wants to have some information about students and votes of some courses at the University of Genova. The interested data are the best, worst and average votes and students of every course. We have a Data Base which contains the information about the students of the various courses, and their votes. Three C procedures, \( \text{min} \), \( \text{max} \) and \( \text{avg} \), are able to evaluate the minimum number in an array, the maximum one and the average of the array values.

**The solution.** We illustrate how our development methodology can be applied to this example.

**Step 1: identification of the set of agents and their interconnecting structure.** An application of this type should be simulated using four agents: **user**, a logical agent which asks information about the courses and handles them for its purposes; **courses information provider (cip)**, a logical agent which receives the user request, and executes a plan to satisfy it; **mathematical function provider (mfp)**, an interface agent which is interfaced with the C procedures \( \text{min} \), \( \text{max} \) and \( \text{avg} \), and **courses data provider (cdp)**, an interface agent which is interfaced with the data base of University courses. The static structure of the prototype is shown in figure 2. **User** is simply capable of sending requests to and receiving answers from **cip** that is the “core” of the system: it exchanges messages with both the interface agents that can communicate only with it.

---

![Figure 2: Student data retrieval: agents and their interconnections.](image-url)
Step 2: choice of the communication protocol among each pair of communicating agents. The communication happens using a very small set of performatives: each pair of agents communicate using ask and reply as illustrated in figure 3. (to be continued in Section 4)

Figure 3: Student data retrieval: communication protocol among agents.

4 Specifying and programming an agent

This section concerns the specification of the behaviour of an agent and the translation of this specification into ACLPL code (steps 3 and partially 4 of the development process). To give a specification of the reactive behaviour of a logical agent we have defined a transition function

\[ \text{la}_\text{behaviour} : \text{Agent} \times \text{State} \times \text{Message} \rightarrow \text{State} \times \mathcal{P}(\text{Message}) \]

where \( \text{Agent} \) is the set of all the defined agents, \( \text{State} \) is the set of all the possible states and \( \text{Message} \) is the set of all the definable messages. 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

\[ \text{la}_\text{behaviour}(A, S, M) = (S_1, \{M_1, \ldots, M_n\}) \]

The behaviour of an interface agent can instead be defined by a simpler function that does not take into account the state, that is

\[ \text{ia}_\text{behaviour} : \text{Agent} \times \text{Message} \rightarrow \text{Message} \]

When an interface agent receives a message \( M \) the only thing it does is to pass the message to its associated interpreter, collect the answer and forward it to the sender of \( M \). The message \( M \), received by agent \( A \), leads the receiver to send the messages \( M_1 \), iff

\[ \text{ia}_\text{behaviour}(A, M) = M_1 \]

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 will explain below), but we are planning to automatize this task.

Consider now the static specification expressed by the function \( \text{la}_\text{behaviour} \). As far as the communication is concerned, 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) are not 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 will 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(Fact)**, to modify the state of the agent by adding Fact to it, and
- **retract_state(Fact)**, to delete the first fact in the state that unifies with Fact.

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 updates, sending of messages and unknown requests. 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 illustrated in figure 4. The action to

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

  < Initial state of the agent >
  < Behaviour of the agent >
```

Figure 4: Structure of the program defining an agent.

...
ask(
    content(X),
    ontology(O),
    sender(S),
    receiver(R)
) :-
    interpret(X, Answer),
    Answer != none,
    send(
        reply(
            content(Answer),
            ontology(O),
            sender(R),
            receiver(S) ),
        S).

Figure 5: Behaviour of an interface agent.

is the result elaborated by the external module and translated into the standard syntax by the interpreter; this result may be none if some problem occurs. Notice that 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 figure 6, where the general form of rules for handling messages is also shown. 

Figure 6: Behaviour of a logical agent and general form of rules.

may be ask, reply, tell, 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 assert_state, retract_state and send will appear between other user-defined primitives.

Example (cont.)

Step 3: specification of the behaviour of each agent in the system. The behaviour of the agents can be explained in natural language as follows. User simulates an external user, asking questions to cip and receiving answers from it. Cip receives a request from the user and, according to the type of request, behaves differently. If user wants to know the names of the best, or worst, or average students of a course, cip asks cdp to know the list of votes of that course; when the answer arrives, it asks mfp to evaluate the maximum, minimum or average value of the list; when mfp gives back the result, it asks cdp again to know who are the students of the course who got the vote
evaluated by mfp, and when it receives the list of names, it sends it to user. If user only wants to know the best, or worst, or average vote given in a certain course, the behaviour of cip is simpler. It asks cdp to know the list of votes of that course; when the answer arrives, it asks mfp to evaluate the maximum, minimum or average value of the list; the result provided by mfp is sent to user. Here, we do not give the functional specification of the whole system, but only one example. The functional notation

\[
\text{ia} \_ \text{behaviour}(\text{cip}, S, \text{ask}(\text{content}(\text{best student}(\text{Course})), \ldots)) = (S', \{\text{ask}(\text{content}(\text{votes}(\text{Course})), \ldots, \text{receiver}(\text{cdp}))\})
\]

is an “instance” of the informal specification given by the sentence “if user wants to know the names of the best students of a course, cip asks cdp to know the list of votes of that course”. Some parts of the functional specification have been left incomplete for sake of visual clearness.

**Step 4: implementation of the prototype (first part).** Figure 7 shows some fragments of the code of the agent cip. **Activation** defines what the agent does when it is activated: it gets all the messages in its public mailbox. **Initial state** defines the initial state of the agent. **Behaviour** comprises the logic rule obtained from the functional specification given above. This rule defines the behaviour of cip when it receives a message requiring information about the best students of the course 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 cdp a message in which it asks information about the votes of the Course students.

---

**Figure 7:** Code of cip.
The code of interface agents is very short, and it is the same for both cdp and mfp. We do not 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, figure 8 shows about half code of the interpreter associated to the cdp agent. Query5 is a procedure exported by the ECLiPSe Data Base, loaded in the module

```prolog
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)).
```

Figure 8: Interpreter associated to cdp.

db_functions, which we have built for this application. The interpreter associated with mfp is instead shown in figure 9. 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.

```prolog
:- 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
```

Figure 9: Interpreter associated to mfp.

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.

Int : name of the file containing the source code of the interpreter associated with the agent; if it has no interpreter associated, the value of this argument has to be set to none.

5 Building and executing the specification of the MAS

(to be continued in Section 5)
Flag\_ignore\_unknown\_message: the two values it can assume are ignore\_unknown\_message and tell\_unknown\_message. 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 did not 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 handling a message will be done.

Flag\_send\_only\_if\_success: in a similar way as for the preceding flag, Flag\_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 sends met proving 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. Its first argument 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.

Every message has a random time stamp associated, which represents the moment in which the message arrives into the mail-box. 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 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 send with the argument high\_priority, the time stamp associated with it is not a number but the atom high\_priority: the message is handled before any other one in the mail-box. If many messages in the mail-box have high priority, they are handled respecting the order in which they arrived.

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 screen at run time.

The goal of starting the simulation requires the use of a scheduler: we use a round-robin scheduler which recursively calls a predicate

\[ activate\_one\_agent(Actual\_num, Num, Flag\_empty\_mail\_boxes). \]

Actual\_num is the integer identifier associated with the agent, Num is the total number of agents present in the system and Flag\_empty\_mail\_boxes is used to verify the termination condition: if for a complete round of the scheduler, all the mail-boxes are empty, the simulation stops. activate\_one\_agent(Actual\_num, Num, Flag\_empty\_mail\_boxes) checks
if there are messages in the mail-box of the \texttt{Actual\_num} agent and, if there are any, calls the predicate \texttt{activate} defined by that agent.

The body of this predicate contains the \texttt{receive\_one} or \texttt{receive\_all} primitives. They inspect the mail-box of the agent looking for messages whose time stamp is less or equal to the system time: \texttt{receive\_one} will handle the first of them while the primitive \texttt{receive\_all} will manage all the messages with the right time stamp.

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

After the simulation has been completed, the user can analyze the messages exchanged using the primitive \texttt{see\_output}. 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.)

\textbf{Step 4: implementation of the prototype (part 2).} By means of \texttt{CaseLP} primitives it is possible to load the agent into the simulation environment. In this phase, shown in figure 10, also the external modules are loaded. The file \texttt{db.pl} contains the code that

\begin{verbatim}
load('cip.pl', cip, none,  
tell\_unknown\_message, update\_success, send\_however),
load('user.pl', user, none,  
tell\_unknown\_message, update\_however, send\_success),
load('interfac.pl', cdp, 'int\_db.pl',  
tell\_unknown\_message, update\_however, send\_success),
load('interfac.pl', mfp, 'int\_math.pl',  
tell\_unknown\_message, update\_however, send\_success),
compile('db.pl').
\end{verbatim}

\textbf{Figure 10: Building the prototype.}

defines the external component accessing the students data base.

\textbf{Step 5: execution of the obtained prototype.} Figure 11 concerns execution of the prototype for the \texttt{student data retrieval} problem. To start the simulation, the \texttt{initialize} primitive must be used, for example with the content shown in the upper part of the figure. While the execution is running, messages like the one shown in the middle of figure 11 appear on the screen. After the execution has stopped, it is possible to see the complete messages exchanged using \texttt{see\_output}; it is possible for example to ask for all messages received by \texttt{user}; a sample answer is shown in the lower part of the figure. Both the visualization modalities are very useful to monitor the behaviour of the
6 Two real–case applications

The two real–case applications we present show some common features: they involve distributed entities which do not have a common knowledge, each 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. The problems that have been addressed are presented below.

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.

The problem. We faced the problem of planning goods transportation: a client which could be a private citizen, a small company or a large industry, requests to transport goods from a place to another. The client does not know how the companies of delivery (Posts, Federal Express, DHL, ...) are organized and which transporters (railway companies, airline companies, ...) are present on the territory: the request specifies only how many goods units must be transported, when they must leave, and whether saving money or time. No information about how realizing the transport is given to the planner, that must provide the client with a solution respecting the time/money constraints she/he set. The
client will be provided with the path (if any) that goods will follow to reach the final destination.

**The solution.** To solve this problem we applied the development methodology described in Section 3.

1. **Identification of the set of agents and their interconnecting structure.**
   We identified four kinds of agents for this application: *client, agency, distributor* and *transporter*. We suppose to have one or more instances of *client, distributor* and *transporter* agents and only one instance of *agency* agents; this instance will be denoted by *agency* in the following. Clients can communicate only with *agency*. *Agency* communicates with clients and interacts with distributors. Distributors are involved in the delivery of goods; they exchange messages with *agency*, other distributors, and transporters. Finally, transporters carry out the physical transport and can only communicate with distributors. All these agents are *logical*, the application we developed did not need any kind of software integration.

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*.

3. **Specification of the behaviour of each agent in the system.** Clients are simulated by the system user that puts their requests in the mail–box of *agency*, using the primitive *initialize*.

   *Agency* receives a query from a client, forwards it to one of the distributors present in the departure place and when this distributor answers, it sends a message containing the result 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. 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 room availability on means of transport, and involves the neighbour distributors if the place to reach is not in its zone. If the agent has to involve other distributors, it does not answer to its applicant until all the distributors it involved have answered. Successively, it provides the applicant with information it got from both transporters and distributors it consulted. Instead, if it can get to the final place, it answers the applicant providing it with all information it got from the transporters. If the distributor had been contacted by *agency* and not by another distributor, when it obtains all the needed information, 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 *agency* that the mission was successful. Obviously, it can happen that means of transport available at the moment of information retrieval 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 fail on all the paths found, or if no path was found, the distributor informs agency that the mission failed. Transporters have a wide knowledge about all the details of time, costs and available room on the different tracks, but they do not perform complex reasoning; in fact, they only give information to the distributors and make reservations for them.

4. Implementation of the prototype. We first gave a static specification of the agents using an instance of the general function la Behaviour presented in Section 4 and then we translated this functional static specification into ACLPL programs. The translation was easy and natural, and we took only few person–weeks to do it. The lines of the code we obtained after this (hand–made) translation are about 500 for agency, about 700 for the transporters and about 7500 for the distributors.

In this application we appreciated the possibility of reusing the rules defining the behaviour of the various agents; for example, as far as the distributor agents are concerned, only a few lines of their code defines their initial state and are thus specific for every agent. The greatest part of the code defines the agent behaviour which is the same for every instance of distributor. We wrote this code once, and used it for every distributor in our application.

5. Execution of the obtained prototype. 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; After this stage the MAS prototype was reliable and behaved in the expected way.

6.2 La Spezia Inter–modal System

This application has been realized by Robert Ziu for his graduation thesis [Ziu97] at the Computer and Information Science Department (DISI) of the University of Genova, in collaboration with FS, the Italian railway company.

The problem. This 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 station of La Spezia with the scheduled arrival of trains, and providing the person in charge for La Spezia station with a decisional support for the management of the wagons on arrival.

The solution. The steps followed to identify and develop the MAS specification are the five ones individuated in our methodology.

1. Identification of the set of agents and their interconnecting structure. There are four kinds of agents: generic terminal, GIOC (Goods Inter–compartment Operative Coordinator), La Spezia production assistant, and section agent. In this application there are many instances of generic terminal and section agent, whereas
there is only one instance of GIOC and of La Spezia production assistant. Generic terminals exchange messages with the section agent of their section and with GIOC; GIOC exchanges messages with all the other agents in the system; La Spezia production assistant exchanges messages with the section agent of Genova and with GIOC; section agents communicate with the terminals of their section and with GIOC. Also in this application the agents are logical ones.

2. Choice of the communication protocol among each pair of communicating agents. The KQML performatives used in this application are ask, reply, inform and the default tell.

3. Specification of the behaviour of each agent in the system. The behaviour shown by the agents is the following: the main problems of a generic terminal concern to obtain the license for a loading, and to know whether it is possible to dispatch a train over the line La Spezia–Milan or not. The request of the license is sent to GIOC, while the dispatching is handled in collaboration with the appropriate section agent. When a section agent receives the request from a terminal, it checks the situation evaluating the congestion index (number of trains per Km). If accepting the new train does not 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, 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 La Spezia production assistant which checks if the station can receive the train; the parameters considered are the number of trains present at La Spezia, the number of trains arriving from the terminals, the number of existing tracks and the average flow of trains moving from the station to the port. When La Spezia production assistant answers to GIOC, it forwards this answer to the terminal which had requested the license.

4. Implementation of the prototype. Also in this application the static specification of the agents was given in terms of the function la behaviour and then translated into ACLPL.

5. Execution of the obtained prototype. 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. An application-dependent graphical interface has been provided using ProTcXl, which enriches ECLiPSe with Tcl/Tk functionalities.

7 Related and future work

This section 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 CaseLP 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 Georgeff [KG96] 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 [Mü1996], 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 [IGV95] (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.

A challenge for the future is building MAS using techniques rising from Software Engineering. On this way goes [Woo1997], that places the bases for an Agent–based Software Engineering. In that paper the construction of a MAS is seen as a software engineering enterprise: issues concerning specification, implementation and verification of MAS are presented, focusing on a number of case studies. In the same journal [Iee1997] two other interesting papers are presented: [Lau1997] presents two case studies (kiosks for telephone sales and support and information access for service technicians), to which agent modellization has been applied to integrate legacy systems into information–dependent applications; instead, [BHM1997] presents a study on how agent–based technology can be applied to traffic and transportation problems.

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+1995] that present CONGOLOG, a concurrent multi–agent programming language based on a logical theory of action, and AgentSpeak(L) [Rao1996] 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 cannot. 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 work: in particular the steps which characterize the agent 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 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 do not introduce inconsistence 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 ours, 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.

CaseLP can be classified as a system for exploiting logic programming in the realization of applications, with a particular attention to their development and to integration of legacy software. On the first side, issues concerning the production of distributed software has been addressed in the last few years by Distributed Software Engineering (DSE) [Kra94]; as we said in Section 1, we deem that using technologies rising from this field can be extremely useful for enhancing standardization of the production of agent-based software. On the other side, we are studying how CaseLP can be coupled with existing integration systems (see for example the IMPACT project at the University of Maryland [AKO+97]). Many other extension to CaseLP are possible. Extending the agent architecture presented in Section 2 and making available multiple specification languages for defining the behaviour of the agents is necessary for enhancing CaseLP in flexibility. Another aspect of the system we want to improve is the simulation: we are realizing statistical and visualization tools for better monitoring prototype execution as well as a more user friendly graphical interface. 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


