APPLYING LOGIC PROGRAMMING TO THE SPECIFICATION OF COMPLEX APPLICATIONS

Maurizio Martelli, Viviana Mascardi, Floriano Zini

DISI - Università di Genova
via Dodecaneso, 35 - 16146 Genova (Italy)
tel: +39 10 3536727 - fax: +39 10 3536099
e-mail: {martelli, mascardi, zini}@disi.unige.it

ABSTRACT

In this paper we show how multi-theory logic programming techniques can be profitably applied to the realization of multi-agent systems which solve complex problems. Our goal is to use logic programming as a tool to give an executable specification of a multi-agent system, obtained from a high-level informal description of the interactions among the different system components. We have implemented a planner for goods transportation as a demonstration of such an approach.

KEYWORDS

Multi-agent systems, logical theories, meta-programming, specification.

INTRODUCTION

The agent paradigm has become, in the last few years, an important developing methodology for complex applications, involving problems like coordination, communication, sharing and integration of knowledge. The advantage of this technology is the ability of modelling heterogeneous programming environments, and its drawback is the lack of standard methods to build the desired system, defining and putting together the various parts, using techniques of software engineering that permit a formal specification of the system's behaviour.

Our final goal is to realize a specification tool for multi-agent systems, using a logical approach. We see two main advantages in the methodology we are developing: (1) Logic programs (without heavy use of extra-logical features and/or optimization techniques) can be considered executable specifications; they are ideal for fast and cheap prototyping, especially when we have to face complex problems: testing a distributed multi-agent system can be a very hard matter, so the verification of implementation choices on a simple prototype helps very much in the system design. (2) Building a multi-agent system modelling a complex real-world problem requires to integrate different kinds of existing software; we think such an integration can be made simpler using meta-programming techniques and easy to carry out in a logic programming paradigm.

The paper organization is the following: we first present our model of multi-agent system; the following section illustrates the methodology to obtain an executable specification in conformity with our models; the next section shows how a real-world problem (planning goods transportation) has been faced using our approach; related and future work concludes the paper.

AGENTS AS LOGICAL THEORIES

A main component of our system is a programming language which manages agents, designed along the lines of KBMS / [MCH+90]. Such a language (we named it ACLPL - i.e., Agent Constraint Logic Programming Language) is implemented in the constraint logic programming language ECLiPSe
[ACD+95] and extends standard (constraint) logic programming providing an environment in which the global knowledge is partitioned into theories. A theory is a knowledge-base (facts and rules) which is logically separated from other knowledge-bases; every theory has the primitives load, prove and update. An agent is a theory enriched with the functionalities send and receive which allow it to exchange messages; it is written in ACLPL, and its code may contain primitives concerning communication and updating. At the moment ACLPL is ECLiPSe with some extensions to deal with theories and agents. The behaviour of an agent consists essentially in executing different operations depending on the messages it receives and its internal state: receiving a message leads the agent to change its state and dispatch messages to other agents.

Communication. In our model each agent may have a communication channel towards any other agent, according to the chosen topology of communication; we assume that a message can reach the receiver within a maximum fixed time. In the channel there isn’t any order between messages, so a message sent before another may be received after the latter. Messages are written in a language which is a subset of KQML [MLP95], a language widely used in the agents community, and which allows to make the content of a message independent from the message itself: we’ll see below this is a useful feature to develop applications involving integration of multi-language environments. The communication primitives an agent can use are send and receive; exchanging of messages is asynchronous: the send primitive is not blocking, whereas the receive blocks the execution of an agent until a message is in its input channel. The system handles send and receive in a way that ensures the atomicity of managing a message; even if many sends are spread all over the definition of an agent’s behaviour related to receiving a particular message, the system records them and performs them all as last action, avoiding interruptions while the agent is running.

Updating. Receiving a message usually updates the agent’s knowledge-base, adding or removing information, and leading the agent into a new state; so a primitive update, with the list of updates as its argument, is provided in our language. The system handles a request of update by forwarding a private message to the agent itself. Since the request of update is translated into a send of a message, it respects the policy of atomicity. Obviously the private messages have the highest priority and they are handled as soon as possible; however it is clear that an update made within an agent, cannot affect the agent state until the message execution is over. This may appear a useless constraint, but it ensures the cleanliness of the system semantics and the consistency between the status of the computation and the contents of the knowledge-base.

Multi-language integration. A main feature of our approach is the possibility to integrate multi-language environments in a common logical framework. In our model we distinguish two sorts of agents, logical agents and interface agents. The former is able to perform complex reasoning, send and receive messages and update its knowledge base whereas the latter may be seen as the one “responsible” of a particular domain (for example a database, a planner or a CIL system), accessible via a set of primitives defining the domain interface. The two classes of agents aren’t disjoint: an agent could be both a logical agent and an interface agent. Figure 1 shows a possible situation: a logical agent coordinates two interface agents. As we said above, choosing KQML as communication language among agents gives us the complete independence between the content of a message and the message itself. When an interface agent receives a message, an interpreter manages its content and accesses the agent’s domain. The content of a message represents a call to the agent’s domain; by defining different interpreters it is possible to manage different domain languages in a natural way. In our model there is a unique communication language (KQML), known by every agent in the system, and possibly different languages to access the various domain. In a logical agent the communication and the domain language are integrated in a unique language whereas an interface agent could be seen as a logical agent that can use only the communication primitives to exchange informations, without the capability to perform complex reasoning.

System simulation. To simulate the behaviour of a multi-agent system in a mono-processor environment as the one in which our ECLiPSe works, we realized a Round-Robin scheduler which activates in turn each agent, allowing it to inspect its input channel and manage the messages it received. Even the scheduler is a theory: in such a way it is possible to have a homogeneous logical
Figure 1: A multi-agent system: a logical agent interacts with two interface agents.

interpretation of the whole system (scheduler and agents). To start a simulation, after loading the system and all agents, it is necessary to initialize the communication channels of some agents, putting in them some messages which fire the communication. The primitive load is provided to load the code of an agent into the ECLiPSe knowledge-base. Meta-programming techniques are used to add to the clauses of this code the information that they belong to a specific agent, and they must be considered separate from all the other clauses present in the ECLiPSe knowledge-base. A primitive initialize is provided to put some initial messages in some channels. The scheduling process halts when every channel is empty: if no message has to be managed, no message will be consequently sent, and the exchange stops. The prove primitive, which proves a goal using only the knowledge of a particular agent, is called by the scheduler and the goal it proves is the predicate activate that must be defined in the code of each agent to handle the receiving of messages.

THE EXECUTABLE SPECIFICATION

The approach we use to obtain and test an executable specification is quite simple, and it can be summarized in the following steps:

1. Identification of the set of agents the application needs. In this step the specification developer decides the static structure of the system he wants to realize; he chooses the kind of agents, as well as the interconnection topology among them.

2. High-level informal description of the interactions among 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. Implementation of each agent by means of a different logical theory. The developer specifies and implements the behaviour of the agents.

4. Execution of the obtained system. In this step the implementation choices are tested, checking if the system does what it has been built to do.

Steps 1 and 2 are still quite informal, in fact we haven’t still realized any tool for performing them, while step 3 has been developed in a more formal way. The behaviour of an agent is specified with a function:

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

The message $M_i$ 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

$$\text{behaviour}(A, S, M) = (S_1, \{M_1, \ldots, M_n\})$$
The messages are predicates with their arguments. It is trivial to translate them into ACLPL facts and rules. The executions on test cases of the resulting system (step 4) can of course give many feedbacks to the previous steps; it can be used to tune the implementation choices previously done.

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 companies of delivery are organized, and which transporters are present on the territory; he would like to take advantage of a reliable service, which ensures him that the goods will reach destination respecting the constraints of minimizing time or cost. To solve this problem we followed the steps outlined in the previous section.

Step 1: identification of the set of agents. We identified four kinds of agents: client, agency, distributor and transporter. The first, which has to send goods, can communicate only with the agency, that communicates with clients and with distributors for handling the clients’ requests. The third is involved in the delivery of goods: it exchanges messages with the agency, other distributors, and transporters. Finally, the last, which carries out the physical transport, 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: choosing the communication protocol. The messages recognized by all the agents identified in step 1 are the performatives ask and reply (KQML-like); it is possible to specify if the ask needs a reply or not. The contents of these performatives are written in ACLPL, i.e., they are requests to the logical agents that don’t need any special interpreter.

Step 3: building the agents. A system behaviour according to the previous steps can be obtained by different implementations of agents. In this step we fix the particular implementation defined by an instance of the general function described previously; it was easy and natural to translate it into ACLPL. An example of definition of the function behaviour is given in figure 2 and from this definition we can define:

\[
\text{behaviour} (\text{Transporter}, \text{State}, \text{Message}) = (\text{New\_state}, \text{Set\_of\_messages\_to\_send})
\]

The strategy we chose to solve the problem is inform-reserve, with a stage of information retrieval, and a stage of actual reservation. The client is not explicitly coded: it is simulated by the system user, that puts his request in the channel of agency, using the primitive initialize. The agency receives a query from the client, forwards it to one of the distributors present in the departure place, and when this distributor answers, it sends a message containing the results of the mission to the client.

A distributor covers a part of the territory in which places are connected by tracks; working over these tracks, there are some transporters which can provide information and services for the transportation. In some places, there are different distributors operating on it. The distributor receives a request from the 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. In this case it doesn’t answer to its applicant until all the distributors it involved have answered and then 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 it got from the transporters. If the distributor had been contacted by the agency, and not by another distributor, when it obtains all the information needed it must decide which is the best way to deliver the goods 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. Concurrency of missions of many clients is supported and since the strategy is inform-reserve, while getting information no
if
  Message =
  ask(  
    content(  
      information about(  
        from(Place1), to(Place2),  
        how many(Number),  
        mission id(Identifier, Nmission)),  
      language(eclipse),  
      ontology(planning_transport),  
      reply(yes),  
      sender(Sender), receiver(Transporter))  
  and State = state of the agent Transporter  
  and the agent Transporter calculates that List time cost is the list of departure times  
  and costs, relative to transports from Place1 to Place2, in which more than Number  
  places are available
then
  New state = State;
  Set of messages to send =  
  { reply(  
    content(  
      information about(  
        from(Place1), to(Place2),  
        informations(List time cost),  
        mission id(Identifier, Nmission)),  
      language(eclipse),  
      ontology(planning_transport),  
      sender(Transporter), receiver(Sender) )  
  )

Figure 2: Example of definition of the function behaviour.

kind of booking is made. Obviously, it can happen that places available at the moment of the  
retrieval of information, are no more available in stage of reservation; In this case the distributor  
sends messages to delete the reservations already made, and tries to reserve a sub-optimum way (it  
records all ways found, so it hasn’t to ask information again). If reservations fails on all the roads  
found, or if no road was found, the distributor informs the agency that the mission failed and then  
the agency informs the client about the failure. 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.

To give an idea of the complexity of the problem, it is sufficient to say that the distributors  
must recognize and handle 13 different instances of ask and reply, and that in an example with five  
concurrent missions, more than 100 messages have been exchanged.

Step 4: executing the specification. As seen before, the executable specification is made up  
by agents implemented as communicating logical theories, loaded and handled by our system. The  
time passing by, is simulated by setting to zero a global clock, common to all the agents in the  
system, when the simulation starts, and adding one unit to it, whenever the Round Robin schedu-  
er completes a round. Messages have associated to them a random time-stamp ranged between the  
current time and the current time more ten, which indicates when the message has to be managed.

The lines of code of our software agents are about 500 for the agency, about 700 for the trans-  
porters and about 7500 for the distributors. The scheduler for the execution of the specification  
required about 1000 lines of code. For what the developing time of the project is concerned, we  
spent about two man-months to carry out the formal specification (steps 1–3) and less than one  
man-month to translate it into ACLPL (step 3). Another man-month was then used to test and  
correct the first, static specification (step 4).
RELATED AND FUTURE WORK

Attempts to use logic languages for programming multi-agent systems have been done by various authors (see for example [BEL95, LLL+95]); in these papers the emphasis is especially on semantic aspects of the language. Our approach is different: the goal we have put at the top of our interests is proving that logic programming can be used for the specification of real-world complex applications (even involving the integration of heterogeneous data and software modules) modelled by means of multi-agent systems. An interesting example of integration methodology is presented in Hermes [SAB+95], a system for integrating informations from diverse sources. The Hermes technology permits a user to submit the system, in a transparent way, queries involving accesses to heterogeneous data sources, without worrying about the organization of each domain. In our approach this could be modelled using only interface agents towards the different domains and a unique logical agent that receives the query from the user and submit it to the appropriate set of interface agents. Of course, to reason about heterogeneous information a semantic integration among the data is needed. With this integration and having, as we have, the possibility to define more logical agents, we'll be able to model more complex problems scenarios in which many active components (logical agents) interact with each other, as well as with passive components (interface agents), the "doors" towards different data/software environments. We use a CLP language as an implementation tool; this choice can be useful in the integration of different CLP environments [LMM96] in a common framework, a promising application field of our approach.

The specification of a planner for goods transportation is a first demonstration of the potentialities of our approach. To obtain a complete specification system a lot of work has still to be done. Developing techniques and tools for the automatization of the system building process is indispensable for the realization of a system easy to use. From a logical point of view the study of the semantics of the languages used for the specification is certainly interesting; this can supply formal methods to study the system's properties.

REFERENCES


