(1) D.I.S.I. - Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
{bozzano,martelli,mascardi,zini}@disi.unige.it

(2) Max-Planck Institut für Informatik
Im Stadtwald, Gebaude 46.1, D-66123 Saarbrücken
delzanno@mpi-sb.mpg.de

Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics

Marco Bozzano[1] Giorgio Delzanno[2] Maurizio Martelli[1]
Viviana Mascardi[1] Floriano Zini[1]

Abstract

The paper presents an ongoing research project that uses Logic Programming, Linear Logic Programming, and their related techniques for executable specifications and rapid prototyping of Multi-Agent Systems. The MAS paradigm is an extremely rich one and we believe that Logic Programming will play a very effective role in this area, both as a tool for developing real applications and as a semantically well founded language for basing program analysis and proof of properties on.

1 Introduction

During the last few years *Multi-Agent Systems (MAS)* [WJ95, NN97] have certainly been one of the most debated approaches to software development. The main reason for this great interest is the intriguing way a software system is viewed in the MAS setting i.e. as a set of autonomous, intelligent and interacting entities that either cooperate to achieve a common goal or that compete to satisfy personal interests. MAS allow a *cognitive* vision of the system and provide the ability to abstract from details thus making them an ideal tool for *Software Engineering* [Pre94]. Furthermore, MAS technology is strictly related to two fundamental aspects of modern software products i.e. *distribution* of computational entities and resources and *integration* of legacy software and data.

Given the high complexity of a MAS, it would be desirable to have a clear methodology supporting the whole development process, from the first informal requirement specification to the final implementation. In order to be applicable, this refinement process should preserve the correctness of the intermediate specifications w.r.t. the original requirements. An automatic tool supporting *formal* specification methods and providing *executable* specifications might greatly help verify correctness constraints when passing through the different refinement steps. It should allow the user to write, execute and

test specifications of agent behaviour, agent architecture, communication protocols and interaction between agents. It should also support integration with existing software products and the automatic translation of the agent specification into existing programming languages.

To achieve this goal, an approach based on *Logic Programming (LP)* would be rather suitable for several reasons. First of all, LP is a powerful paradigm for rapid prototyping based on a very high-level syntax. As a consequence, the use of an LP-based formalism can provide a noteworthy reduction in software development time, a clear description of software functionalities, easy code maintenance and re-usability. Furthermore, the large amount of theoretical effort spent studying verification, analysis, and transformation of logic programs could be applied to develop automatic tools that validate the produced specification.

In this paper we present a framework that includes two different ways of applying LP-based technologies to the development of MAS. This is divided into two phases, supported by different specification and programming languages, both of which are based on the LP paradigm. In the preliminary phase, we use a high-level LP language to specify an application, to simulate the effect of a concurrent execution of the agents, and to check it for correctness. This phase is useful for the construction and validation of the conceptual structure of the system. In a subsequent phase, a "traditional" LP language, extended with communication primitives, is used to produce a more concrete software prototype, to be simulated and tested in a real distributed environment. The first phase concerns the study of a particular communication protocol between agents and the preliminary verification of the agents' behaviour in that context. The next phase provides an effective implementation of the communication protocol and of the agents' code, and faces the problem of the external software integration.

The paper is organized as follows. In the next section we present the main concepts underlying the MAS paradigm and we explain the choice of the LP languages that we use to develop MAS. Section 3 presents a general agent architecture, subsuming some existing models. Some ideas on how an LP-based specification language for this architecture could be structured are described in Section 4. Section 5 concerns a methodology for the specification of a MAS prototype, from informal requirements to implementation. It is applied on a simple toy example. Finally, Section 6 presents conclusions and future work.

2 Why and which LP languages for MAS development?

Logic Programming, which in the 1980s had been identified as the best technology to implement knowledge intensive applications on highly parallel computer architectures, is today relegated to a secondary role in industrial software development. Languages like `C`, `C++` and `Java` are undoubtedly more

known and used than `Prolog`. Imperative languages are adopted for reasons of efficiency while object-orientation is today perceived as the possible new unifying paradigm for computing. However, efficiency is not always a real issue and extensions and integrations of LP with other paradigms fill the gaps of the first LP proposals. Moreover, some peculiarities of LP make it competitive or even better than imperative or object-oriented paradigms for facing particular kinds of applications. In particular, as remarked in [KS98, Wag97], LP is closely connected to the design and development of MAS. We will analyze which features make LP an ideal tool for specifying and prototyping MAS-based applications after having introduced the concepts of "agent" and "MAS".

2.1 A conceptual framework for agents and Multi-Agent Systems

The clearer and the more operationally usable the term "agent" is, the more generally accepted the definition of what an agent and a MAS are will be. At the moment there is no completely satisfactory definition of agent. However, the *weak definition of agency* [WJ95] is the most widely accepted among researchers. According to this definition an (artificial) agent is a computer system (both hardware or software) characterized by:

- *autonomy*: agents work largely independent of human intervention;
- *social ability*: agents communicate with each other, usually by means of some *agent communication language*;
- *reactivity*: agents perceive the external world, including other agents, and react to the incoming information accordingly;
- *pro-activeness*: agents are able to show goal-directed behaviour by taking the initiative to achieve their goals.

A "Multi-Agent System" is a collection of interacting agents which cooperate and coordinate with each other to achieve personal or common goals.

Agents usually need to symbolically represent the state of the world in which they are situated and thus are usually provided with an internal state to contain this piece of information. Beliefs and goals can be first order objects, explicitly represented in the state. In this case, according to [KS96], we mean *rational* agents. Otherwise, what the agents aim at is implicitly coded into the behaviour they are given.

From a practical applications point of view, the main types of agents [NN97] can be roughly classified as:

- *collaborative agents*: emphasize autonomy and cooperation; they are typically static and large and may have to negotiate to achieve acceptable agreements;

- *personal assistance agents*: support and provide pro-active assistance to users struggling with complex application programs;
- *mobile agents*: are software processes that can roam wide-area networks such as the world-wide web, interacting with foreign hosts, acting on behalf of their owner and returning 'home' having achieved their goals;
- *information agents*: are pro-active, dynamic, adaptive and cooperative information managers that manipulate or collate information from many distributed resources.

Even though some authors do define non-autonomous agents (see [LD95]), we believe that "computational" autonomy is a fundamental property for agents. It allows us to have *awake computational entities* in which a programmable *task control*, definable at the meta-level, can manipulate different kinds of behaviour. These kinds of behaviour are determined by the agent's features besides autonomy and social ability, thus resulting in various agent classes[1].

Reactive agents are characterized by behaviour determined by rules expressing what has to be done when external input is received. This input can be either a message from another agent or a signal coming from the *environment* and intercepted by the agent's sensors. Reactive agents do not have a symbolic representation of beliefs or goals and thus do not perform reasoning about them. This does not mean that a reactive agent does not have an internal state, but only that goal-oriented "intelligent" behaviour is an emergent property (see [Bro91]).

Pro-active agents exhibit active behaviour to achieve their goals. The difference with respect to reactive agents is that the actions the agent carries out are not directly driven by an external event (message or signal), but the agent can independently decide which action to perform. Also for this agent's class goals remain hard-wired into the behaviour of the agents and are not explicitly represented in their internal state.

On the other hand *rational* agents have an explicit knowledge base, encompassing beliefs and goals. Goal-oriented "intelligent" behaviour is explicitly "coded" into the agents (a typical example is [RG91]). An agent can usually exploit many different *plans* to achieve the goals that have been ascribed to it. A plan is chosen on the basis of the current beliefs and goals of the agent and can be dynamically modified if the beliefs and/or the goals change.

2.2 Logic Programming and Multi-Agent Systems.

A language for specification and programming of agents must be able to express the ideas underlying the concept of agent and to allow an easy modelling of them. The following observations naturally lead us to the choice of LP paradigms:

[1] The classification is not standard, but it is an attempt by the authors to identify some interesting classes of agents.

- *MAS execution*: the evolution of a MAS consists of non deterministic succession of events; from an abstract point of view an LP language is a non deterministic language in which computation occurs via a search process.

- *Meta-reasoning capabilities*: agents need to dynamically modify their behaviour so as to adapt it to changes in the environment. Thus, the possibility given by LP of viewing programs as data is very important in this setting. This feature is useful also for integrating external heterogeneous software; this is a fundamental aspect in MAS applications.

- *Rationality and reactiveness of agents*: the *declarative* and the *operational* interpretation of logic programs are strictly related to the main characteristics of agents, i.e., *rationality* and *reactiveness*. In fact, we can think of a *pure* logic program as the specification of the rational component of an agent and we can use the operational view of logic programs (e.g. left-to-right execution, use of non-logical predicates) to model the reactive behaviour of an agent. The adoption of LP for combining reactivity and rationality is carefully described in [KS96].

The above observations represent a good starting point to consider LP as a theoretical and practical foundation for the designing and development of MAS. However, as already remarked, traditional LP languages do not fulfill all requirements arising during the MAS development. In particular, though useful to specify a single agent (e.g. [KS98]), they do not provide facilities for modelling collections of distributed and communicating agents. In this paper we propose two extension in this sense:

- from a theoretical point of view, we will consider more powerful specification languages, namely linear logic programming languages;

- from a practical point of view we will propose new extensions of LP-based systems with features specific to the development of MAS.

Linear Logic Programming for MAS specification. Given the complexity of MAS, a good specification language should help specify many different operational aspects in a uniform and natural way. Extensions of Logic Programming based on Linear Logic seem particularly well-suited for this task. *Linear Logic* [Gir87] enriches the operational interpretation of classical logic in that formulas can be treated as resources. This idea has been incorporated in recent extensions of Logic Programming, the so-called *Linear Logic Programming (LLP) paradigm* [Mil95]. It has been successfully applied to formalize important programming aspects such as data management [HM94, BDM97], object-orientation [AP90, DM95, BDLM96], state-based computations [Chi95], and aspects of concurrency [Mil93, MMP96].

These features make LLP a suitable framework for specifying distributed systems and agent systems in particular. The notion of state in LLP has a

natural correspondence with the notion of state and beliefs of an agent. The possibility of using resources during a computation is a natural means to support dynamic changes in the behaviour of an agent. Besides being very powerful specification languages, linear logic-based frameworks can also be used as programming languages as shown in [AP90, HM94, HPW96, Del97].

For our purposes, we will adopt the language $\mathcal{E}_{hhf}$ proposed in [Del97]. It is based on a particular subset of Forum [Mil96], a presentation of higher-order linear logic in terms of goal-driven proofs. $\mathcal{E}_{hhf}$ extends previous proposals like [AP90, HM94] and is defined in a *higher-order* setting, thus facilitating the development of applications based on meta-programming. In Sections 4 and 5 we briefly discuss the role of $\mathcal{E}_{hhf}$ in the specification methodology of our framework.

Logic Programming for MAS implementation. Even though it is executable, the LLP specification is too high level to produce a final agent-based software product. In fact, in writing an $\mathcal{E}_{hhf}$ MAS specification, some important issues must be neglected. For example, an interface between an $\mathcal{E}_{hhf}$ specification and existing software cannot be provided since this issue is abstracted away at the specification level and the integration of external modules and data is not supported by the language. Moreover, performance reasons suggest using a more efficient language than $\mathcal{E}_{hhf}$ for the actual implementation of a MAS prototype.

We are addressing these issues by means of CaseLP (*Complex Application Specification Environment based on Logic Programming* [MMZ97, MMZ98]), a prototyping tool for agent-based software realized in the Constraint Logic Programming language ECLiPSe [ACD+95]. Our tool provides an agent-oriented extension of ECLiPSe that is used to build a more concrete implementation of the MAS. Our implementation language has a number of programming features making the resulting prototype more efficient and easier to integrate with other technologies. CaseLP is described in Section 3, where it is analyzed as a simplified realization of a more general and flexible architecture.

3 A general Multi-Agent System architecture

Agent architectures should be flexible enough to support the amalgamation of the different kinds of agents we have listed in Section 2 so as to give origin to *hybrid* agents with different degrees of reactivity, pro-activeness and rationality. Based on well-known agent architectures we propose a framework where reactive, pro-active and rational components are integrated, as shown in Figure 1. Dotted arrows represent the atomic actions that the various components can perform, while thin, continuous arrows represent the input that the components use to perform their actions. The environment, which is sensed through sensors and modified through effectors (the thick arrows at the top

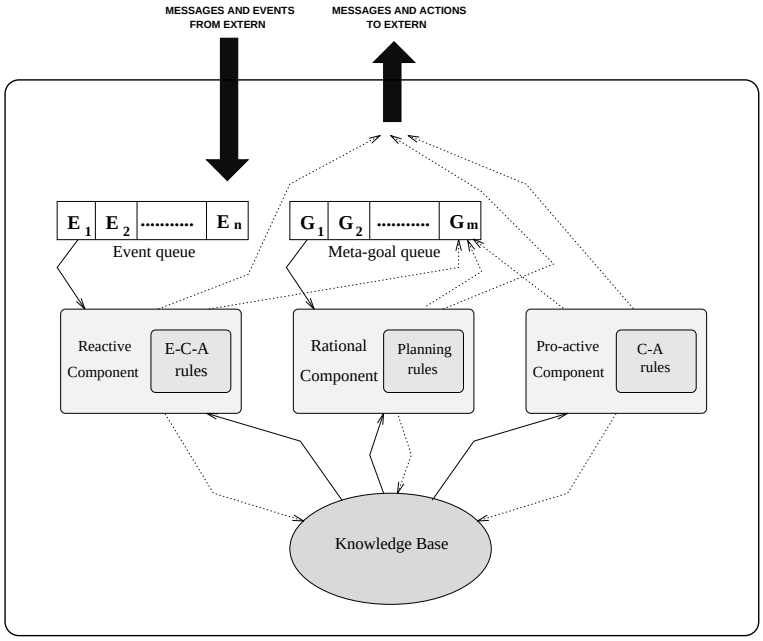

Figure 1: The general architecture of an agent.

of the figure) is outside of the rectangle containing the agent's components. Interaction with other agents occurs via asynchronous point-to-point message passing. More precisely, the main components of an agent are:

The event queue. Events can be either "communication" or "perception" events. The former consist in the reception of messages coming from another agent, while the latter consist in the perception of the environment through sensors.

The meta-goal queue. An agent can respond to external stimuli or to state changes in a simple and immediate manner, but it can also respond in a more complex way by identifying a meta-goal which requires some sophisticated plan to be achieved. The meta-goals generated by the reactive and pro-active components are put into a meta-goal queue and are handled by the rational component, which can add meta-goals to the meta-goal queue as well. This happens, for example, when the rational component interrupts its execution and records a meta-goal describing what it has to do later.

The knowledge base. It reflects the beliefs of the agent at a given moment. It must be expressed in a language (such as first order logic ground

facts) that the rational component is able to perform high level reasoning on. In the following we will refer to it as "state" or "knowledge base" indifferently.

The reactive component. It bases its behaviour on *event-condition-action* rules. The reaction cycle of this component is:

1. pick one event from the event queue and remove it;
2. check the current state;
3. according to the current event and the current state
   - update the state and/or
   - perform actions on the environment and/or
   - send messages to other agents and/or
   - put one or more meta-goals into the meta-goal queue.

The action of putting a meta-goal into the meta-goal queue is performed when the reaction to the event needs to be a complex task which requires some sort of reasoning. It represents a link between the reactive and the rational components.

The pro-active component. This component acts similarly to the reactive one, but does not take the event queue into account. Its behaviour is defined by a set of *condition-action* rules. The pro-action cycle is the same as the reactive component one, except for picking one event from the event queue and considering it during condition evaluation.

The rational component. This component applies some form of reasoning to achieve its current meta-goal. Its execution cycle is:

1. pick one meta-goal from the event queue and remove it;
2. check if the meta-goal still needs to be executed;
3. if it is, construct one or more plans to satisfy the meta-goal;
4. select a plan from the possible alternatives;
5. execute a certain number of atomic actions of the plan.

After the meta-goal has been chosen (step 1), a control is needed to verify if it must still be executed (step 2). In the agent's current situation, which is determined by its knowledge base content, performing the meta-goal may have become useless. After the agent's rational component has executed a certain number of atomic actions of the plan (step 5), the meta-level *task control* can decide to interrupt it. A meta-goal, representing the state of the plan execution, is then posted in the meta-goal queue to resume later on.

The execution of these three components is a question of the *task control* operating at the meta-level and regulating their interactions. The policy might vary from a very constrained sequential execution of the three activation cycles to a completely asynchronous one. In any case strategies to ensure the coherence of the state must be provided, together with some time-constrained evaluation method of the condition in event-condition-action and condition-action rules. This evaluation must not take too long, otherwise the advantage of the immediate action execution given by event-condition-action and condition-action rules is lost.

3.1 Four particular cases of the general architecture

The general architecture presented above should be expressive enough for most of the software agent applications and it is the final target of our ongoing research. Below some existing proposals related to this general abstract model are presented.

Schroeder and Wagner's Vivid Agents [SW].
Vivid agents are software controlled systems whose state comprises the mental components of knowledge, perceptions, tasks and intentions and whose behaviour is represented by means of action and reaction rules. The main functionalities of vivid agents regard handling perception and communication events via a perception system, updating and reasoning on a knowledge system, and representing and performing reactions and actions in order to react to events and to generate and execute plans.

It is easy to see that most of the features of the general agent architecture are present in Schroeder and Wagner's vivid agent. The pro-active behaviour is closely connected to the rational one and when there is no event in the event queue, the agent goes on executing the actions of the current plan. The vivid agents' reactive and rational components act concurrently by interleaving the plan execution with the reaction to incoming events. The two components work over two distinct copies of the knowledge base. Strategies are adopted to ensure the coherence of concurrent actions and the coherence between copies of the knowledge base.

Kowalski and Sadri's Unified Agent Architecture [KS96].
Kowalski and Sadri try to combine the rational and reactive behaviour of an agent by giving a complete proof reduction procedure based on the observation that in many cases it is possible to replace a goal G by an equivalent set of condition-action rules R. Moreover, they face the problem of controlling the reasoning process so that it works correctly with bounded resources. The resulting execution cycle is the following:

1. observe any input coming from the environment at time T;
2. record all input;

3. resume the proof procedure of the current goal statement by first propagating the input[2];

4. continue applying the proof procedure for a total of n inference steps;

5. select an atomic action respecting time constraints;

6. execute any such action and record the results.

This architecture and the more generic one described previously share the same aim of allowing an agent to be both reactive and rational. While in the general architecture the rational and reactive components are clearly separated, here they are collapsed into a single entity, whose behaviour is determined by the proof procedure and the resource bounding. The more the agent is reactive, the less it is rational, and vice versa. This probably means less flexibility of use, but easier implementation of the architecture.

Wooldridge's Computational Multi-Agent System [Woo92].
In this approach the behaviour of an agent is described by the following standard cycle:

1. interpret any message received;

2. update beliefs according to previous action and message interpretation;

3. derive deductive closure of belief set;

4. derive set of possible messages, choose one and send it;

5. derive set of possible actions, choose one and apply it.

Wooldridge defines two execution models for multi-agent systems: in the synchronous one all the agents in the system begin and complete an execution cycle together; in the more realistic asynchronous model, where execution is interleaved, at most one agent is allowed to act at any fixed point of time.

Wooldridge's architecture is a simplification of the general one. As in CaseLP described below, events are always communicative ones, and agents are reactive ones. They receive a message and react to it, without having an explicit representation of their goals and without adopting plans to achieve them.

[2] The *propagation of input* replaces the current goal statement with a simpler one, taking into account the observed input and the integrity constraints characterizing the agent's behaviour.

Martelli, Mascardi and Zini's CaseLP [MMZ97, MMZ98].
CaseLP (*Complex Application Specification Environment based on Logic Programming*) is a prototyping and simulation environment for agent-based software applications developed at the Computer and Information Science Department of the University of Genova (Italy).

CaseLP agents communicate via point-to-point message passing, with messages written in KQML [MLF95]. There are two types of agents in the model: *logical agents*, which show capabilities of complex reasoning, and *interface agents* which only provide an interface between external modules[3] and the agents in the system. 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;
- a *mail-box* for public messages and
- an *interpreter*.

The interpreter is a peculiarity of interface agents. It translates the requests for services that are provided by external modules into the appropriate procedure call and translates the results back into a syntax comprehensible to all the agents in the system.

At the moment the 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 in the following cycle:

1. pick one message from the mail-box and remove it;
2. select one rule whose head unifies with the current message;
3. prove the body of the rule.

The last step is carried out by means of the ECLiPSe interpreter. The atomic actions of updating the state and sending messages are carried out in the CaseLP setting by the *assert_state*, *retract_state* and *send* predicates which operate in a safe way.

Also CaseLP agents, like Wooldridge's, are reactive, still adopting goal reduction to find out what to do. The event queue is represented by the mail-box and contains only communicative events. Even though CaseLP is currently a simplification of the general architecture described above, it has the advantage of explicitly taking into account the integration of external software carried out by the interface agents.

[3] *External modules* are usually legacy passive service providers to be integrated into the MAS.

CaseLP is an ongoing project that we plan to extend and improve on the basis of new and more demanding applications. However, the present implementation has already been successfully applied to some real-world case studies. Two of them were related to transportation and logistic problems [MMZ98]. One was developed in collaboration with *FS* (the Italian railways) and the other one was developed with Elsag Bailey, an international company which provides service automation. CaseLP was successfully adopted for a reverse engineering process in an application concerning the retrieval of medical information in distributed databases [Per98]. Finally, the combination of agent-oriented and constraint logic programming techniques has been used to solve the transaction management problem on a distributed database [MM].

4 A Multi-Agent System specification language

Logical languages have often been adopted to specify agents and Multi-Agent Systems [LLL+95, Rao96, MTF97]. In this section we propose the adoption of Linear Logic Programming as a high-level specification language, and list the reasons that make LLP a very suitable paradigm for these kinds of applications.

Agents combining reactive, pro-active and rational behaviour, such as the ones previously outlined, can be described in an LP setting by a tuple containing:

- the current state;
- the event and meta-goal queues;
- the event-condition-action rules driving the behaviour of the reactive component;
- the condition-action rules related to the pro-active component;
- the high level rules defining how the rational component constructs and executes plans.

We call this tuple a *configuration* of an agent. If we assume that the various rules defining the reactive, pro-active and rational behaviour do not change over time, they can be left out of the agent's configuration. The configuration of a MAS can simply be defined as the set of all the agents' configurations.

When an agent performs an action in a certain MAS configuration, the whole MAS reaches another configuration. If, for example, the action is sending a message, the receiver of the message will change its configuration since its event queue changes, thus leading to a MAS configuration change. A configuration change is called *transition*.

The execution of a MAS can be described as the sequence of the various configurations reached by the MAS. We can take all the configurations into

consideration, but we can also concentrate on configurations that are reached after a complete execution cycle by a certain agent. This would involve many intermediate transitions, one for every atomic action performed by the agent. We could even consider only the configurations which are reached when every agent in the system has completed at least one cycle of execution. The granularity of the MAS execution changes according to what we are interested in observing.

4.1 The role of Linear Logic Programming

Linear Logic Programming allows us to characterize the form of computation previously outlined at a high level of abstraction. In fact, as briefly discussed in the introduction, LLP provides us with connectives to express *concurrency* and *synchronization* primitives. When combined with some representation of the agents, these primitives may be useful for simulating and testing a given MAS specification. In order to give an idea of the approach based on LLP, and in particular on the $\mathcal{E}_{hhf}$ fragment, it is necessary to outline what a linear logic program looks like.

The key point is to extend the syntax of *clauses* as defined in standard Logic Programming so as to provide *multi-conclusion* clauses. More precisely, $\mathcal{E}_{hhf}$-programs are collection of clauses of the form:

$$Cond \;\Rightarrow\; A_1 \parr \ldots \parr A_n \multimap Goal,$$

where the linear disjunction $A_1 \parr \ldots \parr A_n$ corresponds to the head of the clause (A_i's are atomic formulas), $Cond$ is a goal representing the guard of the clause, and $Goal$ is a goal representing the body of the clause. For the sake of the reader, we have limited our considerations to conditions defined by Horn programs.

The main peculiarity of such clauses is that the resources (formulas) they need in order to be applied are consumed right after their execution. In a sense multi-conclusion clauses resemble conditional multiset rewrite rules. Formally, given a program P and a multiset of atomic formulas Ω_0, a resolution step $\Omega_0 \rightarrow \Omega_1$ can be performed by applying a ground instance $C \;\Rightarrow\; A_1 \parr \ldots \parr A_n \multimap G$ of a clause in the program P, provided:

- the multiset Θ consisting of the atoms $A_1, \ldots, A_n$ is contained in Ω_0;
- the condition C is satisfied in P;
- Ω_1 is obtained by removing Θ from Ω_0 and by adding G to the resulting multiset.

In the $\mathcal{E}_{hhf}$-interpreter, instantiation is replaced by unification. At this point, since G may be a complex formula, the search rules (i.e., the logical rules of the connectives occurring in G) must be exhaustively applied in order to proceed.

Such derivations can be used to model the evolution of a collection of agents. For instance, let $Ag_1, \ldots, Ag_n$ be atomic formulas describing a collection of agents. The clause

$$Cond \Rightarrow (Ag_1 \parr \ldots \parr Ag_n \circ\!\!- Ag'_1 \parr \ldots \parr Ag'_n),$$

describes the evolution of the state of the agents (e.g. Ag_i evolves in Ag'_i) provided the condition $Cond$ is satisfied. New components can be added to the current state by using goal-formulas of the form $G_1 \parr G_2$. In fact, the goal $G_1 \parr G_2, \Delta$ simply reduces to G_1, G_2, Δ. This description, which is potentially non-terminating, can be used to observe the evolution of the simulated agent system, or, by using backward analysis, to detect potential violations of the specification requirements.

4.2 A syntax for the general architecture

For the sake of clarity, introducing an abstract, high-level and readable syntax to describe the behaviour of agents and to specify their different components (the reactive part, the pro-active one, and the rational one) will suffice.

As a matter of fact, the syntax we present has a direct mapping onto linear logic formulas, therefore the translation from the high-level language into linear logic clauses could be provided automatically through a compilation process. The only delicate point concerns the implementation of some kind of mechanism that guarantees the right interactions among the different components of an agent and among the agents in the system. At the moment, however, we have not committed ourselves to a particular model of task control, thus this mechanism has not been specified and the syntax we present below must simply be considered as an example[4].

We start the specification of the agents' components from the reactive part. This is specified through simple *event-condition-action* rules, which can be written as follows:

`on event`	event
`check`	st_query
`update`	st_update_list
`perform`	action_list
`try`	meta-goal_list

The meaning of each line is in agreement with the description in Section 3. In addition, sub-languages must be provided to describe events, state queries and updates, actions, and meta-goals. We do not deal with these issues here, but in Section 5.1 we do give an example. The syntax for the pro-active part

[4] The code of the examples given in the paper is available by anonymous ftp at the address `ftp://ftp.disi.unige.it/pub/person/BozzanoM/Terzo`.

is similar to the previous one except for the first line, which is not present since the pro-active component does not perceive external events. Lastly, the syntax for the rational component might look like

```
on goal     goal
check       st_query
generate    plan
```

A simple case-study. We consider a syntax for the case study which is explained in more detail in Section 5.1. The case study involves agents which show a *purely reactive* behaviour. In fact they simply react to incoming events by updating their state and generating new events depending on the current state. Events only include sending and receiving messages, and the communication protocol is based on asynchronous message passing: each agent may be thought of as owning a mailbox for incoming messages. In this case, we can specialize the syntax previously described for specifying the agents' behaviour.

```
on receiving   message
check          st_query
update         st_update_list
send           message_list
```

In the syntax above, *message* is an incoming message, *st_query* is a query on the current state, *st_update_list* is the list of state updates, and *message_list* is the list of new messages sent by the agent.

Each message can be an "ask" (a service request sent to another agent) or a "reply" (a response sent for a given request). Accordingly, we will represent a message with a term like

$$\begin{array}{l} type((\text{content}(C)) \\ \qquad (\text{sender}(S)), \\ \qquad (\text{receiver}(\text{R}))) \end{array}$$

where *type* may be *ask* or *reply*. We will also assume that every agent has a simple state consisting of ground facts, and a simple language for state modification based on the primitives $assert(Fact)$ and $retract(Fact)$.

5 Towards a specification methodology

In this section we analyze the different phases which make up the specification methodology of our framework, trying to outline the different contributions given to the development process by each phase. Our approach can be compared with the classical development cycle for software prototypes given in [Pre94].

1. **Identification of the set of agents and their interconnecting structure**. In this step the specification developer decides the static structure of the system and identifies the kind of agents the application requires. He/she also chooses the interconnection topology, i.e. which communication channels will be needed among them. This phase is quite informal, allowing different choices in the number and kind of agents required.

2. **Choice of the communication protocol among each pair of communicating agents**. This step consists in choosing the communication protocol between each pair of connected agents. As for the previous step, there is room for different choices, depending on what kind of information the agents need to exchange and on the synchronization mechanism.

3. **Specification of the behaviour of each agent in the system**. This step consists in specifying the behaviour of the agents, namely what each agent is able to do and how it performs its tasks. This is where Linear Logic Programming comes into play. This is the first phase that achieves some degree of formalization, by building an executable specification written in $\mathcal{E}_{hhf}$. It is important to notice that the whole process including steps 1 through 3 may be repeated more than once, either because the testing phase (step 4) reveals some flaws in the initial choices, or because the developer wishes to refine the specification by using a greater degree of granularity. The concept of *granularity* of a specification is fundamental. The developer often needs to study the execution of a Multi-Agent System at different levels of abstraction, progressively refining the specification as he/she is convinced of the design correctness. The $\mathcal{E}_{hhf}$ specification language seems to be quite suitable for this kind of design.

4. **Testing of the system**. This phase concerns testing the system in order to verify how much the prototype corresponds to the desired requirements. This may lead to changing, improving or refining the design. Using a logical language like $\mathcal{E}_{hhf}$ in this phase has great advantages since:

   - It is possible to evaluate a goal step by step, following the evolution of a particular system in detail. Various abstraction levels are possible, for instance observing only the messages exchanged between the agents, and/or observing the behaviour of a single agent, and so on.
   - Through *backtracking* it is possible to follow all the different evolutions of a given system, depending for example on the order of arrival of the various messages. It is therefore possible to simu-

late a distributed environment, where the order in which messages arrive may not correspond to the order in which they were sent.

- It is possible to verify whether a particular computation may be carried out, or, more importantly, that *every* computation starting from a given configuration leads to a final state satisfying a given property, independently of the order of arrival of the messages and the order of execution by the agents. To this aim it suffices to run the desired computation together with a goal negating the desired property of the final state, and then to check whether the global goal fails.
- It may be possible to employ standard techniques for proving program properties that have been developed in the logic programming context. Extending these techniques to the linear logic setting is part of our future work (see Section 6).

5. **Implementation of the prototype**. In this step each agent specification is firstly translated into executable code, then the MAS is built, creating a unique executable specification embedding all the defined agents. This step and the following one can be dealt with using CaseLP as a prototyping environment. CaseLP provides facilities for automatically translating an agent specification written in an extended logic language into an executable piece of code. It also allows the user to load these agents into a unique Multi-Agent System for further execution.

6. **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 made in the first 3 steps. CaseLP allows us to initialize the mail-boxes of some agents with initial messages and then starting the MAS execution. In this phase CaseLP uses a round-robin scheduler which recursively activates each agent in the MAS. The activated agent inspects its mail-box looking for new messages and manages them according to the rules defining its reactive behaviour. When an agent has managed its messages, the scheduler passes to the following one. The scheduler activity stops when all the mail-boxes of the agents are empty. It is possible to monitor the execution of the system thanks to on-line and off-line text visualization of the exchanged messages. The CaseLP Visualizer [Ped98] provides the user with an interface which allows him/her to initialize the system, integrate external software, start and monitor the execution in a user-friendly graphic fashion.

5.1 An example: student data retrieval

We present a very simple example in which agents, based on the CaseLP architecture, are described by the syntax presented at the end of Section 4.

This will serve as an illustration of the specification methodology previously presented.

The problem. Suppose a user wants some information about students and marks of some courses at the University of Genova. The possible queries include the best, worst and average marks of each course, and the names of the students who got the marks. An external database contains this information. Three C procedures, *min*, *max* and *avg*, used to evaluate the minimum, maximum and average element of an integer array, could be linked to produce the final system.

This problem can be faced by developing a Multi-Agent System according to the given methodology. The third and fourth steps are described rather carefully, while the other ones are treated quite briefly. More details can be found in [MMZ98].

Step 1: identification of the set of agents and their interconnecting structure. An application of this type could be simulated using four agents: *user*, a logical agent which asks for information about the courses; *course information provider (cip)*, a logical agent which receives the user request and executes it; *mathematical function provider (mfp)*, an interface agent which is interfaced with the C procedures `min`, `max` and `avg`, and *course data provider (cdp)*, an interface agent which is interfaced with the database of University courses.

User is capable of sending requests to and receiving answers from *cip* which is the "core" of the system. It exchanges messages with both interface agents that are only able to communicate only with *cip*. Figure 2 depicts the structure of the MAS.

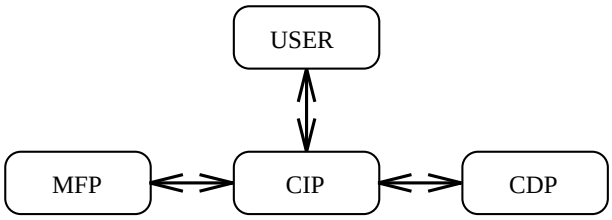

Figure 2: Agents in the "Student data retrieval" example.

Step 2: choice of the communication protocol among each pair of communicating agents. Communication takes place via the protocol described at the end of Section 4. Each pair of agents communicate using asynchronous message passing, where a message can have *ask* or *reply* type.

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 behaves on the basis of the type of request. If, for instance, *user* wants to know the best, worst, or average marks of a course, *cip* asks *cdp* to get the list of marks for 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 then sent back to *user*.

This kind of behaviour by the *cip* agent is illustrated by the clauses in Figure 3, written according to the high-level syntax presented in Section 4. The first clause applies when *cip* receives a message requesting the best mark

```
on receiving
   ask(content(best_mark(Course)), sender(S), receiver(cip))
check
   req_id(Id), Id1 is Id + 1
update
   retract(req_id(Id)), assert(req_id(Id1)),
   assert(associated(Id1, best_mark(Course),S))
send
   ask(content(marks(Course, Id1)), sender(cip), receiver(cdp)).

on receiving
   reply(content(marks(Mark_List,Course, Id)), sender(cdp), receiver(cip))
check
   associated(Id, best_mark(Course),S))
update

send
   ask(content(max(Mark_List,Id)), sender(cip), receiver(mfp)).

on receiving
   reply(content(max(Max,Id) sender(mfp), receiver(cip))
check
   associated(Id, best_mark(Course),S))
update
   retract(associated(Id, best_mark(Course),S))
send
   reply(content(best_mark(Course, Max)), sender(cip), receiver(S)).
```

Figure 3: Code for *cip* using the abstract syntax for agents.

of a given course. This request is managed by sending a message to *cdp* asking for the list of marks for that particular course. In order to keep track

of pending requests, the *cip* agent associates every request with a unique identifier and stores this piece of information in its internal state. The second clause applies when the corresponding reply from *cdp* arrives. By consulting its internal state, *cip* realizes that the request was to calculate the maximum mark of the list it received from *cdp*, therefore it contacts *mfp* to carry out this task. When the corresponding answer from *mfp* finally arrives, *cip* consults its internal state and forwards the result to S (third clause). The remaining clauses for *cip* and for the other agents in the system can be written similarly.

Remark. To get an idea of how a program written in this high-level syntax can be mapped into a linear logic program, we present the translation of the first clause of Figure 3. It must be recalled that a linear logic program is basically a collection of conditional multiset rewrite rules. The multi-conclusion clause for the considered rule is defined as follows:

Id1 is Id + 1 ⇒
on_receiving(ask(content(best_mark(Course)),sender(S), receiver(cip))) ⊗
ag(cip) ⊗ req_id(Id) ⟜
ag(cip) ⊗ req_id(Id1) ⊗ associated(Id1,best_mark(Course),S)) ⊗
send(ask(content(marks(Course,Id1)),sender(cip),receiver(cdp)).

The effect of such a clause is to *rewrite* the components of the current global state in agreement with the specification associated with the considered event.

Note that agents and events are represented by atomic formulas. The condition defined in the `check` part of the rule is handled in a special way. More precisely, the part of the condition which does not involve the global state (e.g. Id1 is Id+1) becomes a condition in the corresponding multi-conclusion rule, whereas, the part that involves the global state (e.g. req_id(Id)) becomes part of the head of the clause. In this way, req_id(Id) is automatically removed from the current state and substituted by req_id(Id1). The new information associated(..) and the new goal send(..) are asserted by simply including them in the body of the rule. The event `on_receiving` is generated as soon as an agent removes a message from its mailbox. We can specify this behaviour as follows:

receive(Msg, Ag) ⊗ mailbox([Msg|T], Ag) ⊗ ag(Ag) ⟜
ag(Ag) ⊗ on_receiving(Msg) ⊗ mailbox(T, Ag).

Again, note that the content of the mailbox is modified by rewriting the old list of messages into the new one.

The system is completely specified once the agents in the system and the initial state of each agent are specified. For instance, in the case of the *cip* agent, we have to specify what the initial value of *req_id* is. The prototype can then be tested starting from a particular configuration, i.e., a particular initialization of the agents' mail-boxes. Note that more than one agent can share the same behaviour. We should assume, for example, that more than

one *cdp* agent is available to simulate a replicated database. In this case the behaviour is defined just once.

Another feature of this specification framework that we have previously insisted on is the possibility to refine the specification using different levels of abstraction. For instance, in this particular example it would be possible to firstly define the behaviour of the *cdp* agent in a very simple manner without implementing the actual mechanism which accomplishes a particular task. The set of clauses for this purpose would look like

```
on receiving
   ask(content(marks(course,Id)), sender(S), receiver(cdp))
send
   reply(content(marks(course, [28,30,...])), sender(cdp), receiver(S)).
```

for each *course* under consideration. Once the system has been tested and the interactions among the agents has been proved correct, the specification might be refined by describing the exact manner in which *cdp* accesses the database and finds an answer to a query.

Step 4: Testing the system. It is possible to test the system and verify its correctness with respect to the given requirements by executing the specification written in the previous step. To this aim, the $\mathcal{E}_{hhf}$ interpreter can be used to execute the code for this example. The user can set up an initial configuration, made up of some agents and some initialization messages, and follow how one computation proceeds. He/she can impose that the end of the computation correspond, for instance, to the situation in which all messages have been processed by the agents. He/she can then observe the final configuration, i.e. the agents and their corresponding states. Variable bindings, as usual in LP, can return values as well. For instance, it is possible to prove a goal like the following ("||" means concurrent execution):

```
ag cdp || ag cip || ag user || ag mfp ||
req_id cip 0 ||
send(ask(
    content (best_mark(data_base),Best),
    sender(user),
    receiver(cip)),cip)
```

The output of the simulation is the final configuration

```
ag cdp || ag cip || ag user || ag mfp ||
req_id cip 1
```

together with the variable binding *Best = 30.*

The execution of a goal may be observed at various levels of granularity. The interpreter supports both a *trace* level, which allows us to observe low level details of the computation, and a *debug* level, which allows us to observe the interactions among the agents.

Another possibility is to exploit backtracking in order to follow *all* possible computations starting from a given configuration. By doing so a user can verify how the order in which the messages are exchanged affects the computation (this is crucial in distributed simulation). It is possible to verify whether a given property is satisfied *independently* of all possible orders of message exchanging and of all possible solutions for variable bindings. This is done by negating the property to be satisfied and proving that the corresponding goal necessarily fails. For instance, the failure of the following goal

```
(ag cdp || ag cip || ag user || ag mfp ||
req_id cip 0 ||
send(ask(
    content (best_mark(Course,Best),
    sender(user),
    receiver(cip)),cip)),
Best < 28.
```

proves that for *each* course the best mark is greater or equal to 28.

Step 5: Implementation of the prototype. In this step we use CaseLP to describe the agent behaviour by means of logical rules. Figure 4 shows some fragments of the *cip* agent code. The correspondence between this code and the first clause of Figure 3 is quite easy to see. `Activation` defines what the

```
Activation
   activate :- receive_all.

Initial state
    request_identifier(0).

Behaviour
   ask(content(best_mark(Course)), sender(S), receiver(cip)) :-
        req_id(Id), retract_state(req_id(Id)), Id1 is Id + 1,
        assert_state(req_id(Id1)),
        assert_state(associated(Id1, best_mark(Course), S)),
        send(ask( content(marks(Course, Id1)), sender(cip), receiver(cdp)), cdp)
```

Figure 4: Code for *cip* in CaseLP.

agent does when it is activated by the system scheduler i.e. it gets all the messages in its public mail-box. `Initial state` defines the initial state of the agent. `Behaviour` comprises the logic rule describing how to reply to a "best_mark" query.

In CaseLP implementation, *cdp* and *mfp* have been realized as *interface agents*. They share simple behaviour and are interfaced respectively to an ECLiPSe database and a C module via two different interpreters.

Step 6: Execution of the obtained prototype. CaseLP provides a tool (CaseLP Visualizer) that allows the user to load agents and external modules into the simulation environment and to visualize the execution of the MAS by means of a simple GUI. In our example the four agents, the ECLiPSe database and the C module are first loaded.

To start the simulation the mail-boxes of some agents are to be initialized, putting some messages into them. The CaseLP Visualizer provides an appropriate window for this aim, as illustrated in Figure 5.

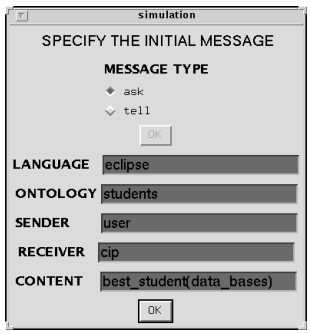

Figure 5: CaseLP Visualizer: initialization window.

While the execution is running, windows for each loaded agent appear on the screen. Information about state changes and exchanged messages is visualized for each agent. Figure 6 presents a snapshot of the MAS execution. After the execution has ended, it is possible to see a more detailed visualization of the occurred events, as illustrated in Figure 7. Exchanged messages and state updates are shown for each agent. Clicking on an event, it is possible to see more details, as shown in Figure 8. The figure represents an answer received by the agent *user*. Both the on-line and off-line visualization modalities provided by the CaseLP Visualizer are useful to monitor and verify the behaviour of the prototype, in order to check whether it behaves correctly.

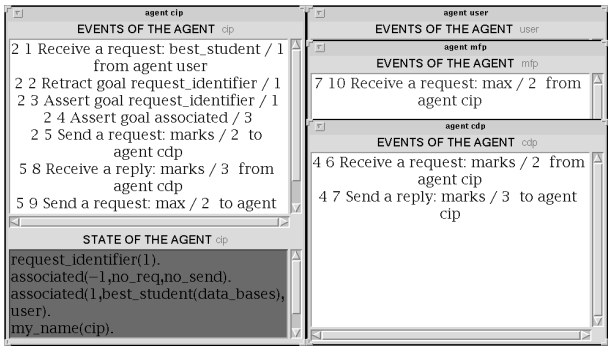

Figure 6: CaseLP Visualizer: on-line visualization of execution.

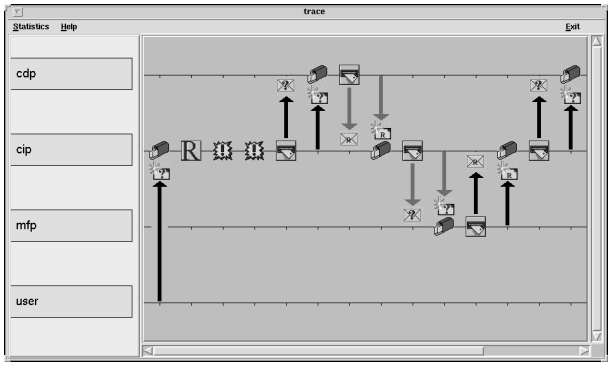

Figure 7: CaseLP Visualizer: off-line tracing of execution.

6 Conclusions and future work

In the paper *Agent Based Software Engineering* [Woo97], M. Wooldridge considers

> "the problem of building a Multi-Agent System as a software engineering enterprise [involving three main issues]: how agents might be specified; how these specifications might be refined or otherwise transformed into efficient implementations; and how implemented agents and Multi-Agent Systems might subsequently be verified."

The aim of our paper was to suggest a potential answer to these three questions, assuming that the target of the implementation is a MAS prototype

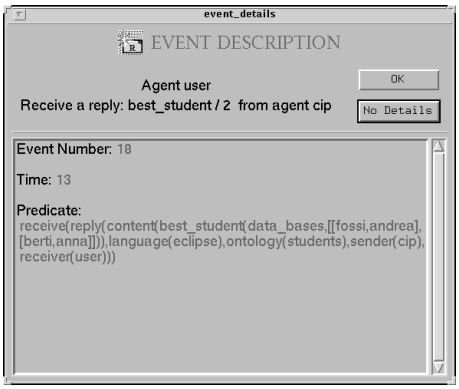

Figure 8: CaseLP Visualizer: details of an event.

instead of a final software product. In a prototype we do not need great efficiency, therefore it can be realized using more formal but less efficient technology. The role of a declarative language like Logic Programming is important. We can keep the first specification closer to implementation and this can be very useful in automatizing the whole prototyping process.

In fact, we use two logical languages to specify a Multi-Agent System at different refinement levels. The first language is Linear Logic Programming, suitable for an initial stage of the specification process. It can be easily used to describe the behaviour of systems where agents need to synchronize and run concurrently. An important aspect that is neglected in the LLP specification is the actual integration of external software. The behaviour of external modules is in fact simulated by LLP agents. We can directly execute the LLP abstract specification by means of the $\mathcal{E}_{hhf}$ interpreter and in this phase we can prove that the MAS is correct with respect to some original requirements expressed as a set of linear formulas.

The LLP specification is subsequently transformed into a more efficient LP program. Thanks to the meta-programming techniques that are easy to use in a logic programming paradigm, LP provides a tool for integrating existing external software modules in quite an easy, natural way. LP seems to be appropriate for a second phase of the specification and prototyping process, when the developer has an idea of the general mechanisms regulating the system under development and he/she wants to actually integrate some external modules. The LP specification is executed by means of the CaseLP prototyping environment. CaseLP allows us to follow what happens to the agents as if they were really distributed communicating entities. It gives the feeling of how MAS execution could go on by allowing us to change the initial conditions (the initial messages put into the mail-boxes before the simulation

starts), and providing a certain nondeterminism by randomly delaying the messages sent by the agents. Moreover, it allows real integration of existing software, thus permitting the building of a system which is partly simulated and partly already implemented.

In this paper we have stressed the methodological aspects of building a MAS, rather than the analysis of an existing system. The ideal system, in fact, should support the MAS developer in all the steps pointed out in Section 5 and should allow the specification of reactive, pro-active and rational agents. Such a system would provide facilities for debugging the specification, for integrating external software modules and agents written in different languages, for supporting different communication protocols, and for checking properties at different levels. We could obtain this kind of system only by taking into account the different approaches we have outlined throughout this paper, and integrating them into a more general, multi-purpose system.

A proposal comes from the ARPEGGIO project, outlined in [DKM+]. ARPEGGIO *(Agent based Rapid Prototyping Environment Good for Global Information Organization)* aims to become a general open framework for the specification, rapid prototyping, and engineering of agent-based software. This framework will include contributions from the Department of Computer Science at the University of Maryland (USA) for aspects concerning the integration of multiple data sources and reasoning systems, from the Department of Computer Science at the University of Melbourne (Australia) for the work on animation of specifications, and from the Department of Computer and Information Science at the University of Genova (Italy) for the work on CaseLP and $\mathcal{E}_{hhf}$.

Lastly, an improvement in CaseLP and $\mathcal{E}_{hhf}$ functionalities is surely desirable. We could start by working on the following issues:

LLP specification. The efficiency and user-friendliness of the $\mathcal{E}_{hhf}$ interpreter for linear logic formulas could be improved. Furthermore, a tool for the automatic translation of a high level syntax, like the one of Section 4.2, into linear formulas and possibly into CaseLP clauses would be desirable.

Testing. It would be interesting to analyze how to extend standard LP testing techniques (symbolic model checking, partial evaluation, abstract interpretation) to support an automatic property verification of programs, and how to integrate these techniques with more traditional testing methods.

CaseLP implementation. We need to extend the range of languages/tools to which CaseLP can be interfaced. Currently we can integrate C, Tcl/Tk, the ECLiPSe Data and Knowledge Base and obviously ECLiPSe modules, but we would like to support other programming languages, thus providing a really *multi-language* specification tool.

System execution. In CaseLP we only simulate the distribution of the agents. We need a scheduler, and all the MAS execution runs over a single processor. We are studying how to actually distribute CaseLP agents over different machines, and are evaluating the use of CORBA [OHE97] for this purpose.

References

[ACD+95] A. Abderrahamane, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. *ECLiPSe 3.5 User Manual.* European Computer Research Center, Munich, December 1995.

[AP90] J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. In D.H. Warren and P.Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 495–510. The MIT Press, Cambridge, MA, 1990.

[BDLM96] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. A Linear Logic Calculus of Objects. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 67–81. The MIT Press, 1996.

[BDM97] M. Bozzano, G. Delzanno, and M. Martelli. A Linear Logic Specification of Chimera. In *Proceedings of DYNAMICS'97, a satellite workshop of ILPS '97*, 1997.

[Bro91] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Chi95] J. Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[Del97] G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic.* PhD thesis, Università of Pisa, Dipartimento di Informatica, March 1997.

[DKM+] P. Dart, E. Kazmierckaz, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications. Submitted to FASE'99.

[DM95] G. Delzanno and M. Martelli. Objects in Forum. In *Proceedings of the International Logic Programming Symposium*, pages 115–129. The MIT Press, 1995.

[Gir87] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[HM94] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[HPW96] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, Munich, Germany, July 1996.

[Iee97] *IEE Proceedings of Software Engineering*, 144(1), February 1997.

[KS96] R. Kowalski and F. Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In *Proc. of International Workshop on Logic in Databases*, San Miniato, Italy, 1996. Springer-Verlag.

[KS98] R. A. Kowalski and F. Sadri. From Logic Programming to Multi-agent Systems. Submitted to publication, 1998.

[LD95] M. Luck and M. D'Inverno. A Formal Framework for Agency and Autonomy. In *Proc. of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 254–260, San Francisco, CA, June 1995.

[LLL+95] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II*, pages 331–346. Springer-Verlag, 1995. LNAI 1037.

[Mil93] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P.Mello, editors, *Proceedings of the 1992 Workshop on Extension to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, Berlin, 1993.

[Mil95] D. Miller. Survey of Linear Logic Programming. *Computational Logic: The Newsletter of the European Network of Excellence in Computational Logic*, 2(2):63–67, 1995.

[Mil96] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[MLF95] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II*, pages 347–360. Springer-Verlag, 1995. LNAI 1037.

[MM] V. Mascardi and E. Merelli. Agent-Oriented and Constraint Technologies for Distributed Transaction Management. Submitted to IIA'99.

[MMP96] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *ENTCS*, 3, 1996.

[MMZ97] M. Martelli, V. Mascardi, and F. Zini. Applying logic programming to the specification of complex applications. *Mathematical Modelling and Scientific Computing*, 8, 1997. Proc. of *11th International Conference on Mathematical and Computer Modelling and Scientific Computing*.

[MMZ98] M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In H. S. Nwana and D. T. Ndumu, editors, *Proc. of The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM98)*, pages 331–354, London, UK, March 1998.

[MTF97] M. Mulder, J. Treur, and M. Fisher. Agent Modelling in METATEM and DESIRE. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.

[NN97] D. T. Ndumu and H. S. Nwana. Research and development challanges for agent–based systems. In *IEE Proceedings of Software Engineering* [Iee97], pages 2–10.

[OHE97] R. Orfaly, D. Harkey, and J. Edwards. *Instant CORBA*. John Wiley and Sons, 1997.

[Ped98] M. De Pedrini. CaseLP Visualizer: un tool di visualizzazione per sistemi multi agente logici. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[Per98] G. Persano. Gestione Distribuita di Informazioni Mediche Mediante Tecniche Multi-Agente. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[Pre94] R. S. Pressman. *Software Engineering. A Practitioner's Approach*. McGraw–Hill International, UK, 3rd edition, 1994. European Edition. Adapted by D. Ince.

[Rao96] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer-Verlag, 1996. LNAI 1038.

[RG91] A. Rao and R. Georgeff. Modeling Rational Agents within a BDI–Architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[SW] M. Schroeder and G. Wagner. Vivid Agents: Theory, Architecture, and Applications. Submitted to the Journal of Logic and Computation.

[Wag97] G. Wagner. Artificial Agents and Logic Programming. In *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi–Agents*, pages 69–87, Leuven, Belgium, July 1997.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Woo92] M. Wooldridge. *The Logical Model of Computational Multi–Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992.

[Woo97] M. Wooldridge. Agent–based software engineering. In *IEE Proceedings of Software Engineering* [Iee97], pages 26–37.