Multi-agent Systems Development as a Software Engineering Enterprise

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

[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

Abstract. Multi-Agent Systems provide an ideal level of abstraction for modelling complex applications where distributed and heterogeneous entities need to cooperate to achieve a common goal, or to concur for the control of shared resources. This paper proposes a declarative framework for developing multi-agent systems. A formal approach based on Logic Programming is proposed for the specification, implementation and testing of software prototypes. Specification of the PRS agent architecture is given as an example of application of our framework.

1 Introduction

Declarative languages, such as functional and logical languages, have mainly been used in the academic world. The use of imperative paradigms for the development of industrial software is usually motivated by reasons of efficiency. However, besides being reusable, declarative knowledge is more modular and flexible than imperative knowledge. It has better semantics, makes detecting and correcting contradictory knowledge easier, and provides abstraction of the (real) world in a natural way. Moreover, in this setting the use of meta-programming techniques provides a support for the integration of different kinds of knowledge. These features make the declarative paradigm a solution that is suitable for developing and verifying prototypes of complex applications, where a set of autonomous, intelligent and distributed entities cooperate and coordinate the exchange and integration of knowledge.

Agent-oriented technology [21, 16] faces the problem of modelling such kinds of applications. It is suitable for modelling entities which communicate (*social ability*), monitor the environment and react to events which occur in it (*reactivity*), are able to take the initiative whenever the situation requires so (*proactivity*) without human beings or other agents intervening (*autonomy*). Societies of such entities are called *Multi-Agent Systems (MAS)*. They take into account the *distribution* of the involved agents and the *integration* of heterogeneous software and data. These two issues are fundamental for the success of present software

G. Gupta (Ed.): PADL'99, LNCS 1551, pp. 46–60, 1998.

systems and this is one of the reasons for the consensus that MAS have obtained in academia and industry. The combination of declarative and agent-oriented approaches has been studied over the last few years and it is very promising from both a theoretical and a practical point of view (see [9, 8, 10, 12]).

Unfortunately, at this time there is no evidence of a well-established engineering approach for building MAS-based applications. However, due to their inherent complexity, experimentation in this direction seems very important. This paper presents some features of CaseLP [11], an experimental, logic programming based, prototyping environment for MAS. In particular, we present a methodology that combines traditional software engineering approaches with considerations from the agent-oriented field. The approach exploits logic-based declarative languages for the specification, implementation and testing of the prototype. In our methodology the more formal and abstract specification of the MAS is given using the linear logic programming language $\mathcal{E}_{hhf}$ [2], which provides constructs for concurrency and state-updating. Afterwards ACLPL, an extension of the logic programming language ECLiPSe with agent-oriented constructs, is used to build a software prototype closer to a final implementation of the MAS. The declarative nature of $\mathcal{E}_{hhf}$ guides the translation into ACLPL, which, in turn, has a number of programming features making the resulting prototype more efficient and easier to integrate with other technologies.

In our framework we consider the possibility of building prototypes encompassing agents with different architectures. The designer will either choose a predefined architecture in a library that is part of the CaseLP environment or will develop a new one. In the former case, only the relevant data for a given agent will actually be provided by the user. As an example, in this paper we present the application of some steps of our methodology to the specification of the well known *Planning and Reasoning System (PRS)* agent architecture [6].

The paper is structured as follows: the next section presents our development framework, focusing on the prototype developing methodology. Section 3 describes the PRS architecture and how $\mathcal{E}_{hhf}$ and ACLPL can be used to model this architecture following (part of) the methodology described in Section 2. Section 4 compares CaseLP with other frameworks for the specification and development of MAS, and concludes the paper with some considerations on future research work.

2 A Framework for MAS Development

The development of agent-based software can be seen as a traditional software engineering enterprise [20]. First, a specification in some suitable specification formalism is given, then it is refined into an executable piece of software. To assure correctness of the refinement process, the final concrete system has to be verified with respect to the initial specification. Formal methods play an important role in the development phase of agent-based systems at a high level, and when developing complex cooperating systems.

Since there is no generally accepted taxonomy for classifying agent-based applications, it is not yet clear which approaches are appropriate for specifying the different classes of agent-based systems and which methods can be exploited to transform these specifications into a final software product [3, 5]. In the last few years many approaches based on logical languages (e.g., temporal logic [15]) have been proposed as specification formalisms for agent-based technology. Logic programming languages can contribute to such research in that they provide easy-to-define, executable specifications. We propose a framework for the realization of MAS prototypes in which both the specification and implementation are carried out using declarative logical languages.

2.1 Prototype Realization

CaseLP (Complex Application Specification Environment Based on Logic Programming) [11] is a MAS-based framework for prototyping applications involving heterogeneous and distributed entities. More precisely, the framework includes a methodology that guides the application developer to an easy and rapid definition of the prototype through the iteration of a sequence of simple steps. CaseLP provides a set of tools that are used to achieve the aim of each step in the methodology. In particular, tools for specification of the MAS, tools for describing the behaviour of agents that make up the system by means of a simple rule-based logical language, tools for the integration of legacy systems and data, and simulation tools for animating the MAS execution are provided.

A CaseLP agent can be either a *logical agent*, which shows capabilities of complex reasoning, or an *interface agent*, which provides an interface between external modules and the agents in the system. The final prototype can be built by combining existing software tools and new components. This approach furthers integration and reuse, two issues that are highly relevant for the success of new technologies. All agents share some main components that 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 incoming messages and events, an *interpreter* for accessing external software (only for interface agents).

The language for high-level MAS specification is $\mathcal{E}_{hhf}$ which has interesting capabilities for modelling concurrent and resource sensitive systems (see Section 3.1 below). The language for defining agent implementation is ACLPL, a Prolog-like language enriched with primitives for safe updates of the agent state ($assert\_state(Fact)$ and $retract\_state(Fact)$) and for communication among agents in the MAS ($send(Message, Receiver)$, $asynch\_receive(Message)$ and $sync\_receive(Message, Sender)$). $Message$ follows the syntax of KQML [13], that we have chosen as the agent communication language. The update primitives operate in such a way that if the agent fails some activities a safe state is automatically restored. The primitive *async_receive(Message)* inspects the agent's mail-box and retrieves the first message contained in the mail-box, if any. This kind of reception is not blocking. On the other hand, *sync_receive(Message, Sender)* is the blocking reception primitive. The agent waits until a message coming from the agent *Sender* enters the mailbox.

CaseLP provides appropriate primitives for loading agents into the system and for associating an appropriate interpreter to interface agents. This creates a MAS that is ready for simulation which is performed by means of a *round-robin scheduler* interleaving the activation of all the agents. The simulation produces both on-line and off-line information about the agents' state changes and about the messages that have been exchanged. The CaseLP Visualizer provides a GUI for loading, initializing and tracing the agents' execution in a graphical, user-friendly manner.

Prototyping Methodology. The realization of a MAS-based software prototype can be performed according to the following steps:

1. **Static architectural description of the prototype**. The developer decides the static structure of the MAS. This step is further broken down to: (a) determining the *classes of agents* the application needs; (b) determining a set of *requested services* and a set of *provided services* for each class; (c) determining the set of *necessary instances* for each class; (d) defining the *interconnections* between the instances of agents, matching appropriately requested and provided services.
   This phase defines the components, what they are able to do, what they require and which communication channels are needed to manage the services.
2. **Description of component interactions**. This step specifies how a service is provided/requested by means of a particular *conversation* (set of KQML messages with a specific order) between each pair of connected agents. Each conversation can be performed using different *communication models*, such as synchronous or asynchronous message passing.
3. **Architectural choice for each agent**. Each logical agent can be structured according to a particular agent architecture (e.g., only reactive or proactive agents may be needed). This step allows the developer to decide the most appropriate internal architecture for each agent in the system. For example, he/she decides a predefined architecture (such as the PRS that will be used as our guiding example in this paper) or must choose to build a specification of his/her own. Obviously, in the former case the steps involved in writing the specification will be much easier since dealing with the modelling of the internal mechanisms of the agents is not needed.
4. **High-level specification of the system**. In this step Linear Logic Programming comes into play and $\mathcal{E}_{hhf}$ is used to build an executable specification of the system. Due to its peculiar properties, $\mathcal{E}_{hhf}$ allows easy modellization of concurrency among agents, as well as updates of agents states. We can identify three different levels of modellization:
   (a) specification of interactions among agents (external concurrency), abstracting from their architecture and taking into account the interaction model specified in step 2;
   (b) specification of the (new) architectures chosen in step 3, i.e. modellization of the interactions between the internal components of the agents (internal concurrency);

(c) specification of the agents' behaviour, i.e. how they operate to provide their services.

It is important to point out that the whole process listed in steps 1 through 4 may be repeated more than once, either because the testing phase (step 5) reveals some flaws in the initial choices, or because the developer decides to refine the specification. For example, in the first stage only specification 4a listed above could be given while 4b and 4c could be defined afterwards.

5. **Testing of the system**. This phase concerns testing the system in order to verify how closely the prototype corresponds to the desired requirements. Using a logical language like $\mathcal{E}_{hhf}$ for this phase has numerous advantages and using the $\mathcal{E}_{hhf}$ interpreter makes it possible to evaluate a goal step by step, following the evolution of a particular system in detail. It is possible to verify whether a particular computation may be carried out, or what is more important, that *every* computation starting from a given configuration leads to a final state which satisfies a given property. It might also be possible to employ standard techniques to prove properties of programs in the logic programming context. This is part of authors' future work.
6. **Implementation of the prototype**. At this point of the development process we have an abstract (hopefully correct) specification of the final application. This step transforms the $\mathcal{E}_{hhf}$ specification into a prototype much closer to the final implementation (i.e., interfaces towards external software and data, message passing communication, etc.). Furthermore, performance and standardization reasons suggest using more efficient and widespread logical languages than $\mathcal{E}_{hhf}$ for an actual implementation of MAS prototypes. In this step specifications 4a, 4b and 4c have to be translated into executable ACLPL code. 4a corresponds to various implementations of message exchange. 4b is translated into suitable data structures (obtaining different architecture parts) and into a meta-program that implements the control flow of the architecture. Finally, the architecture-dependent rules defining 4c are translated into ACLPL rules. The framework also allows the user to join all the agents that form the prototype into a unique executable specification, so that the system can be further tested. Obviously, in most cases only 4c has to be translated, if existing solutions are chosen for 4a and 4b.
7. **Execution of the obtained prototype**. The last step tests the implementation choices, checking whether the system behaves as expected. Any error or misbehaviour discovered in this step may imply a revision of the choices made in the previous steps.

Practical Application of CaseLP. CaseLP has been adopted in order to develop applications in different areas. Two applications were related to transportation and logistic problems. In particular one was developed in collaboration with *FS* (the Italian railway company) to solve train scheduling problems on the La Spezia – Milano route, and another one was developed with Elsag Bailey, an international company which provides service automation, to plan transportation of goods. Another application concerned the retrieval of medical information

contained in distributed databases. In this case CaseLP was successfully adopted for a reverse engineering process. Finally, the combination of agent-oriented and constraint logic programming techniques has been faced with CaseLP to solve the transaction management problem on a distributed database.

3 The PRS Architecture

The specification and implementation of the architecture of an agent are certainly among the most difficult phases of our development methodology. We will furnish CaseLP with a library of agent architectures from which the application developer will pick the desired model. In this section steps 4b and 6 of our methodology are applied to implement the PRS architecture.

The *Procedural Reasoning System (PRS)* [6] has obtained broad consensus among researchers in the multi-agent systems field. The model of practical reasoning that underpins PRS is BDI (Beliefs, Desires and Intentions) [18] which is operationalized in PRS agents by the notion of *plans*. Each agent has a plan library, which is a set of plans and represents the agent's procedural knowledge. Each plan consists of: a *trigger*, or invocation condition, which specifies the circumstances under which the plan should be considered; a *context*, or precondition, specifying the circumstances under which the execution of the plan may commence[1]; a *maintenance condition*, which characterizes the circumstances that must remain true while the plan is executing, and a *body* defining a potentially complex course of actions which may consist of both *goals* and primitive *actions*. The agent's interpreter can be outlined by the following cycle [4]: *(1)* observe the world and the agent's internal state, and update the *event queue* consequently; *(2)* generate possible new *desires* (tasks) by finding plans whose trigger event matches an event in the event queue; *(3)* select one from this set of matching plans for execution; *(4)* push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal; *(5)* select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

On the basis of work in [4], a specification of dMARS (an implementation of PRS) using the Z specification language [19], we will show how this architecture can be modelled using $\mathcal{E}_{hhf}$ and CaseLP. First we briefly introduce some basic features of linear logic programming [2, 14].

3.1 Executable Specifications in $\mathcal{E}_{hhf}$

The linear logic language $\mathcal{E}_{hhf}$ [2] is an executable language for modelling concurrent and resource sensitive systems based on the general purpose logical specification language Forum [14]. $\mathcal{E}_{hhf}$ is a multiset-based logic combining features of extensions of logic programming languages like λProlog, e.g. goals with implication and universal quantification, with the notion of *formulas as resources* at the

[1] Note that a triggered plan can start its execution only if its context is satisfied

basis of linear logic. Below we will informally describe the operational semantics of $\mathcal{E}_{hhf}$-programs. Specifically, $\mathcal{E}_{hhf}$-programs are a collection of multi-conclusion clauses of the form:

$$A_1 \parr \ldots \parr A_n \circ\!\!- \mathit{Goal},$$

where the A_i's are atomic formulas, and the linear disjunction $A_1 \parr \ldots \parr A_n$ corresponds to the head of the clause. Furthermore, $A \circ\!\!- B$ (i.e., $B \multimap A$) is a linear implication. The main peculiarity of such clauses is that they *consume* the resources (formulas) they need in order to be applied in a resolution step.

Formally, given a multiset of atomic formulas (the state of the computation) Ω_0, a resolution step $\Omega_0 \rightarrow \Omega_1$ can be performed by applying an instance $A_1 \parr \ldots \parr A_n \circ\!\!- G$ of a clause in the program P, whenever the multiset Θ consisting of the atoms $A_1, \ldots, A_n$ is contained in Ω_0. Ω_1 is then obtained by removing Θ from Ω_0 and by adding G to the resulting multiset. In the 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 a derivation, which corresponds to a branch of the proof tree of the multiset Ω, can be used to model the evolution of a PRS agent. Ω represents the current global state, e.g., *beliefs*, *intentions* and the *event queue*, whereas P describes the possible *plans* that can be triggered at any point during a computation. $\mathcal{E}_{hhf}$ provides a way to "guard" the application of a given clause. In the following extended type of clauses

$$G_1 \,\&\, \ldots \,\&\, G_m \;\Rightarrow (A_1 \parr \ldots \parr A_n \circ\!\!- \mathit{Goal}),$$

the goal-formulas G_i's must be solved (i.e., executed in P) in order for the clause to be triggered. 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. Conditions over the current state can be tested by using goal-formulas of the form $G_1 \& G_2$. In fact, the goal $G_1 \& G_2, \Delta$ reduces to G_1, Δ and G_2, Δ. Thus, one of the two copies of the state can be consumed to verify a contextual condition. Universal quantification in a goal-formula $\forall x.G$ can be used to create a new identifier t which must be local to the derivation tree of the subgoal $G[t/x]$. Finally, the constant $\top$ succeeds in any context and the constant $\bot$ is simply removed from the current goal. Such a description can be used to observe the evolution of an agent or, by using backward analysis, to detect violations of the requirements of the specifications.

3.2 LLP Specification of a PRS Agent

In this section we explain in detail how to specify the salient aspects of the PRS architecture using $\mathcal{E}_{hhf}$.

Beliefs, Goals, Actions and Plans. *Beliefs* can be modelled by ground facts of the form *belief(Fact)*. The set of beliefs is maintained in the current state Ω. *Goals* are terms of the form *achieve(Fact)* and *query(Fact)*. An *achieve goal* commits the agent to a sequence of actions, whose execution must make the goal true. A *query goal* implies a test on the agent's beliefs to know whether the

goal is true or not. *Internal actions*, represented by the terms *assert(Fact)* and *retract(Fact)*, update the agent beliefs. *External actions* (e.g. sending messages) are denoted by generic terms. *Plans* basically consist of sequences of actions and (sub)goals. They are represented as facts in the following form

plan(Trigger, Context, Body, Maintenance, SuccActs),

where *Trigger* is a trigger linked to a particular event, *Context* is a condition that has to be true to start the plan, *Body* is a sequence of actions and (sub)goals $B_1 :: \ldots :: B_n$, *Maintenance* is a condition that has to be true while the plan is executed, *SuccActs* is a sequence of actions to be executed if the plan succeeds. Thus, the *plan library* of the agent is described as a collection of facts. On the contrary to [4] we do not take into consideration *failure actions* whose aim is generally to rollback to a safe state in case the plan fails. This can be obtained without effort in an LP setting by exploiting backtracking.

Triggers and events. Triggers are raised by events and their aim is to activate plans. We can distinguish between *external triggers*, that force an agent to adopt a new intention, and *internal triggers*, that cause an agent to add a new plan to an already existing intention. External triggers are: *addbelief(Fact)* and *delbelief(Fact)*, linked respectively to events denoting acquisition and removal of a belief, and *goal(G)* denoting acquisition of a new goal. Instead, there is only one internal trigger, i.e., *subgoal(G)*, denoting a (sub)goal call in the executing plan. Events are terms of the form *event(Trig, Id)*, where *Trig* can assume one of the forms above, and *Id* is a (possibly undefined) identifier of the plan instance causing the event.

Event queues. An event queue is associated with each PRS agent. It contains external and internal events and is represented by a term of the form

$$event\_queue([E_1 \mid \ldots \mid E_n]),$$

where $E_1 \ldots E_n$ are events. Events linked to external triggers are always inserted at the end of the event queue, whereas events linked to internal triggers are inserted at the top. Since events are taken from the top of the event queue, this policy gives priority to events generated by an attempt to achieve a (sub)goal.

Plan instances. Plan instances represent plans to execute. A new plan instance is created and inserted into an *intention stack* as soon as its trigger has been activated and its context is satisfiable. A plan instance contains an instantiated copy of the original plan from which it derives together with and the information about whether the plan is *active* or *suspended*. Differently from [4], we use shared variables and unification to inherit knowledge from the original plan. A unique identifier is associated to each plan instance. Thus, plan instances have the form

plan_inst(Id, Body, Maintenance, Active, SuccActs).

Intention stacks. An intention stack contains the currently active plan and a set of suspended plans. An internal trigger *subgoal(G)* can generate a new

plan instance that is pushed onto the same intention stack as the plan containing the (sub)goal whose execution has activated the trigger. An external trigger produces a new plan instance that is stored in a new intention stack. In our representation an intention stack is a term of the form

$$int\_stack([P_1 \mid \ldots \mid P_n]),$$

where $P_1 \ldots P_n$ are plan instances. The internal behaviour of a PRS agent can be described by means of the $\mathcal{E}_{hhf}$ rules listed below. For the sake of this example we do not explicitly give rules for *perception*, i.e., rules for simulating interactions with the external world. As in [4], the idea is to start the simulation of the agent from an initial queue of events and from an initial intention stack. During execution, perception rules can be used to build appropriate events according to information produced by some perception devices (for example environment sensors or inter-agent communication devices) and to (non deterministically) post them into the agent event queue. Other than the rules listed below, we consider a program P containing the plan library and the information to distinguish external and internal triggers and actions, of the form $external(Trigger)$, $external(Action)$, etc.

Plan triggering. Plan triggering rules handle creation of new plan instances and intention stacks, according to the raised trigger. If an external trigger has to be handled, a new intention stack is created and a new plan instance is pushed onto it. This can be formalized by the following rule:

$$plan(T, C, B, M, SA) \;\&\; external(T) \;\Rightarrow\; (event\_queue([event(T, \_)|L]) \;\circ\!\!-\; verify(C) \;\&\; (\forall Id.\; int\_stack([plan\_inst(Id, B, M, act, SA)]) \;\text{⅋}\; event\_queue(L))).$$

The conditions $plan(\ldots)$ and $external(\ldots)$ must be fulfilled by P (i.e. must be unified with facts in P), whereas the goal-formula $verify(\ldots)$ & $(\forall Id. \ldots)$ allows us to test the contextual condition over a copy of the current state (for sake of brevity $verify$ is not specified here). New identifiers for the plan instances are created by using universal quantification. Finally, note that the modification of the event queue is defined by *consuming* the current one (the head of the clause) and creating a new copy (in the body).

A similar clause formalizes what to do if the trigger is internal. The main difference is that the intention stack, whose top plan has the same identifier as the first event in the event queue, is *consumed*. After verifying the context, the intention stack is rewritten adding a new instance of the plan whose trigger matches the trigger of the topmost event in the event queue.

Plan execution. A plan is executed when the event queue of the agent is empty. An intention stack such that its top plan instance is active is non deterministically chosen and the first action of the plan instance is executed (if the maintenance condition is verified). We have developed rules for each possible plan component: external and internal actions, query goals and achieve goals. For sake of brevity we only report the most significant among them.

In case an external action has to be executed, the following rule is applied:

$$external(A) \Rightarrow (event\_queue([]) \otimes int\_stack([plan\_inst(Id, A :: B, M, act, SA)|L]) \multimap verify(M) \,\&\, (event\_queue([]) \otimes execute(A) \otimes int\_stack([plan\_inst(Id, B, M, act, SA)|L]))).$$

The call to $execute(A)$ activates the agent output devices, for example effectors on the environment, or the inter-agent communication for sending a message to another agent.

In case of internal actions the following rules are applied; if the top component of a plan is an *assert* action, the corresponding belief is added to the database (if it is not present):

$$event\_queue([]) \otimes int\_stack([plan\_inst(Id, assert(F) :: B, M, act, SA)|L]) \multimap verify(M) \,\&\, not\_believed(F) \,\&\, (belief(F) \otimes event\_queue([event(addbelief(F), Id)]) \otimes int\_stack([plan\_inst(Id, B, M, active, SA)|L])).$$

Similarly, if the top component of a plan is a *retract* action, then the corresponding belief (if present) is removed by *consuming* it.

If the plan component is an *achieve* goal, and the corresponding belief is in the database, then the agent can proceed:

$$event\_queue([]) \otimes belief(F) \otimes int\_stack([plan\_inst(Id, achieve(F) :: B, M, act, SA)|L]) \multimap verify(M) \,\&\, (event\_queue([]) \otimes belief(F) \otimes int\_stack([plan\_inst(Id, B, M, act, SA)|L])).$$

If the previous rule cannot be applied, the current plan is suspended and a new event containing a *subgoal* trigger and a reference to the current plan is created:

$$event\_queue([]) \otimes int\_stack([plan\_inst(Id, achieve(F) :: B, M, act, SA)|L]) \multimap verify(M) \,\&\, not\_believed(F) \,\&\, (event\_queue([event(subgoal(F), Id)]) \otimes int\_stack([plan\_inst(Id, B, M, susp, S)|L]).$$

If the plan component is a *query* goal and F is a belief of the agent, then the query succeeds by using a rule similar to the first *achieve* rule (the plan instance at the top of the intention stack is a *query(F)* instead of an *achieve(F)*).

Plan termination and resuming. If a plan has been completed successfully, the *SuccActs* (sequence of internal actions) has to be executed. Furthermore, awakening a previously suspended plan may be necessary. The following rules capture execution of *SuccActs*:

$$event\_queue([]) \otimes int\_stack([plan\_inst(Id, [], \_, act, assert(F) :: SA)|L]) \multimap not\_believed(F) \,\&\, (event\_queue([]) \otimes belief(F) \otimes int\_stack([plan\_inst(Id, [], \_, act, SA)|L])).$$

$$event\_queue([]) \otimes belief(F) \otimes int\_stack([plan\_inst(Id, [], \_, act, retract(F) :: SA)|L]) \multimap event\_queue([]) \otimes int\_stack([plan\_inst(Id, [], \_, act, SA)|L]).$$

When the execution of *SuccActs* has been completed, the plan instance can be popped away from the intention stack, and the (suspended) lower plan instance has to be awakened:

$$event\_queue([]) \ \otimes \ int\_stack([plan\_inst(\_,[],\_,\_,[]), plan\_inst(Id,B,M,susp,SA)|L]) \\ \circ\!\!- \ \ event\_queue([]) \ \otimes \ int\_stack([plan\_inst(Id,B,M,act,SA)|L]).$$

When an intention stack becomes empty, it is removed using the rule

$$int\_stack([]) \ \circ\!\!- \ \bot.$$

As mentioned before, such a specification can be directly executed in a logic programming language by setting an initial set of events and of intentions. The next step in the CaseLP methodology is to refine the specification in order to get closer to a real implementation.

3.3 LP Prototyping of a PRS Agent

The refinement of the $\mathcal{E}_{hhf}$ specification into a more concrete prototype aims mainly at the realization of a software product based on a more established technology and in which modules written in different languages can be integrated. Moreover, we consider the possibility of building prototypes encompassing agents with different architectures and we aim at providing the CaseLP environment with a library of meta-interpreters, each of which reproduces a particular kind of management of the agent behaviour. After being abstractly specified with $\mathcal{E}_{hhf}$, these meta-interpreters will (automatically) be translated into the more concrete ACLPL language. As a first step towards this library, we present an interpreter for the PRS architecture which is directly derived from the previously defined $\mathcal{E}_{hhf}$ specification.

An interpreter for the PRS architecture. Mapping the $\mathcal{E}_{hhf}$ specification of PRS into a CaseLP agent does not require a great effort. In fact, the $\mathcal{E}_{hhf}$ data structure previously defined finds a direct mapping into ACLPL terms. Furthermore, representation of *beliefs* and *intentions* is stored in the agent *state*, the *event queue* corresponds to the agent *mail-box*, and *plans* are memorized in the agent *behaviour* as facts.

We can note that the $\mathcal{E}_{hhf}$ clauses, used for specifying the PRS interpreter, assume the general form

$$G_1 \ \&\ldots\&\ G_m \ \Rightarrow (A_1 \otimes \ldots \otimes A_n \circ\!\!- G_{m+1} \ \&\ldots\&\ G_{m+k} \ \&\ A_{n+1} \otimes \ldots \otimes A_{n+h}),$$

where all the formulas $G_1, \ldots, G_{m+k}, A_1, \ldots, A_{n+h}$ are atomic. In order to translate a multi-conclusion guarded clause C of such a form, we can introduce an auxiliary predicate p_C, defined as

$$p_C \text{ :- } retract\_state(A_1), \ldots, retract\_state(A_n),\ G_1,\ \ldots, G_{m+k}, \\ assert\_state(A_{n+1}),\ \ldots, assert\_state(A_{n+h}).$$

where *retract_state* and *assert_state* are the state update predicates previously described. The execution of $retract\_state(A_1), \ldots, retract\_state(A_n)$ consumes the atomic formulas $A_1,\ \ldots, A_n$, the proof of $G_1,\ \ldots, G_{m+k}$ tests both the clause guard and the condition over the current state, and finally the execution of

$assert\_state(A_{n+1}),\ \ldots, assert\_state(A_{n+h})$ adds new information to the state itself.

Applying this transformation to every $\mathcal{E}_{hhf}$ clause, we can obtain a corresponding set of CaseLP clauses. Such a set can be partitioned by grouping together clauses regarding the four main activities performed by a PRS agent that are *perception*, *plan_triggering*, *plan_execution* and *action_execution*. The top-level behaviour of a simple version PRS interpreter can be given by the rule that starts the four main activities, with a priority that reflects their order in the body of the clause.

$$prs\_interpreter :- perception;\ plan\_triggering;\ plan\_execution;\ action\_execution.$$

Obviously, more sophisticated strategies can be implemented, so that the execution of the four activities may take additional information into account.

Each activity can be defined by the clauses belonging to the corresponding activity group, for example

$$plan\_triggering :- p_{C_1};\ \ldots\ ; p_{C_s}.$$

where $p_{C_1}, \ldots, p_{C_s}$ are the clauses of the plan triggering group.

We do not present the complete code for the PRS interpreter, but we only give (part of the) rules for handling internal triggers and external actions. The rule for handling an internal trigger (belonging to the definition of *plan_triggering*, together with rules for external triggers) is

$$\begin{aligned} plan\_triggering :-\ & retract\_state(mail\_box([event(T, Id1)|L]),\\ & retract\_state(int\_stack([plan\_instance(Id1, B_1, M_1, act, SA_1)|L_1])),\\ & plan(T, C, B, M, SA), internal(T),\ verify(C),\ get\_new\_id(Id),\\ & assert\_state(mail\_box([L]),\\ & assert\_state(int\_stack([plan\_instance(Id, B, M, act, SA),\\ & \quad plan\_instance(Id1, B_1, M_1, act, SA_1)|L_1])). \end{aligned}$$

In this clause the call of a p_{C_i} has been substituted by its definition and the goal $get\_new\_id(Id)$ creates a new identifier.

To execute a plan step involving an external action the interpreter uses the rule

$$\begin{aligned} & plan\_execution :-\\ & \quad retract\_state(int\_stack([plan\_instance(Id, [A|Body], Maint, active, SA)|L])),\\ & \quad retract\_state(mail\_box([])),\ external(A),\ verify(Maint),\\ & \quad assert\_state(int\_stack([plan\_instance(Id, Body, Maint, active, SA)|L])),\\ & \quad assert\_state(mail\_box([])), assert\_state(execute(A)). \end{aligned}$$

The rule for executing an external action is

$$\begin{aligned} action\_execution :-\ & retract\_state(execute(send(Message, Receiver))),\\ & send(Message, Receiver). \end{aligned}$$

where the verification of goal $send(Message, Receiver)$ has the side effect of performing the message send. Note that in CaseLP the only allowed external actions concern sending messages, so this specialized rule can be used.

The PRS interpreter is started whenever the agent is awakened by the system scheduler. This is achieved by calling the hook predicate *activate* defined as

$$activate :- prs\_interpreter.$$

After the PRS agent has executed one of the four main activities, control returns to the system scheduler so that another agent can be activated.

The above translation of the $\mathcal{E}_{hhf}$ specification into the concrete logical language ACLPL could be further refined into a more efficient one by exploiting advanced features of the ECLiPSe system.

4 Comparison and Future Work

In our paper we have described CaseLP as a tool which adopts software engineering techniques to develop multi-agent system prototypes. Logic is used both as the prototype specification and implementation language. How does CaseLP compare with other general purpose tools for MAS development and testing?

MIX [7] has been conceived as a distributed framework for the cooperation of multiple heterogeneous agents. Its basic components are the agents and the network through which they interact. A *yellow page* agent offers facilities for receiving information on the dynamic changes of the network environment.

The Open Agent Architecture (OAA) [12] provides software services through the cooperative efforts of distributed collections of autonomous agents. There are different categories of agents recognized in the framework. The *facilitator* is a server agent responsible for coordinating agent communications and cooperative problem-solving. *Application agents* are "specialists" that provide a collection of services (eventually based on legacy applications). *Meta-agents* help the facilitator agent during its multi-agent coordination phase, and the *user interface agent* provides an interface to the user.

ZEUS [17] is an advanced development tool-kit for constructing collaborative agent applications. It defines a multi-agent system design methodology, supports the methodology with an environment for capturing user specification of agents and automatically generates the executable source code for the user-defined agents. A ZEUS agent is composed of a definition layer, an organization layer and a coordination layer. ZEUS also provides predefined yellow and white pages agents.

ZEUS appears to be the most similar to CaseLP. The proposed methodology is given as a set of questions, whose answers the MAS developer uses to define the agent in terms of its role, the services it provides and the relationships with other agents. Even though it is less formalized, it has the same aim as the CaseLP methodology. A feature which characterizes all the presented approaches except for CaseLP is the presence of a predefined agent which helps discover which agent provides which services (yellow and white pages in ZEUS, yellow pages in MIX and facilitator in OAA). It would not be difficult to provide CaseLP with such an agent but, at the moment, it is up to the user to implement it, if necessary. Integration of legacy software is dealt with in MIX, OAA and ZEUS by adopting a *wrapping* approach that "agentifies" existing software. On the contrary CaseLP adopts an *interpretation* approach.

As an example of specification of MAS architecture, we have presented a possible realization of the PRS one. We think the approach we followed for

the PRS specification may have some advantages over other ones present in the literature. In [4] a specification for PRS is given using Z, a specification language based on set theory, first-order logic, and the notions of state space and transformations between states. We think that Linear Logic may be a more natural candidate for the specification of systems where the notion of state, transformations and resources are involved. The main advantage of $\mathcal{E}_{hhf}$ with respect to Z is its being directly *executable*. It also supports specification at different levels of abstraction like Z, but its higher-order extensions (Z is first-order) greatly facilitate meta-programming.

Another formal specification for PRS is presented in [15]. It is based on a particular temporal logic language, Concurrent MetateM, which supports the specification of concurrent and distributed entities. Goals, beliefs and plans are represented by means of formulas in temporal logic, whose execution is committed to a run-time execution algorithm. Verification of the specifications is rather direct even though at the moment a small, fast interpreter for Concurrent MetateM is not available. On the contrary, the language we propose is based on a linear extension of classical logic languages such as Prolog. Its execution mechanism is goal directed and is based on clause resolution and unification as is usual in Logic Programming. $\mathcal{E}_{hhf}$, like Concurrent MetateM, easily supports broadcast message-passing and can simulate meta-level activity by means of its higher-order features.

An experimental interpreter for $\mathcal{E}_{hhf}$ has been developed by the authors (see `ftp://ftp.disi.unige.it/pub/person/BozzanoM/Terzo`). The development of a more efficient interpreter for the language is part of our future work. Furthermore, we also plan to investigate the possibility of extending standard Logic Programming techniques for software verification and validation to the Linear Logic context. Among these, it would be worth considering *symbolic model checking*, and possibly techniques based on *partial evaluation* and *abstract interpretation*. As far as CaseLP is concerned, we plan to extend the system to allowing for a *real* distribution of the agents on a network, so that prototypes will be closer to final implementation. Finally, as described in [1], we intend to integrate different research experiences based on Logic Programming, including CaseLP, into a common joint project which will lead to the development of the general open framework ARPEGGIO (Agent based Rapid Prototyping Environment Good for Global Information Organization), for the specification, rapid prototyping and engineering of agent-based software. The ARPEGGIO framework will also include work on integration of multiple data sources and reasoning systems being carried out by the Department of Computer Science at the University of Maryland (USA), as well as work being done on animation of specifications at the Department of Computer Science at the University of Melbourne (Australia).

References

[1] 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.
[2] G. Delzanno. *Logic & Object–Oriented Programming in Linear Logic*. PhD thesis, Università di Pisa, Dipartimento di Informatica, 1997.
[3] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge. Formalisms for Multi-Agent Systems. *The Knowledge Engineering Review*, 12(3), 1997.
[4] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.
[5] M. Fisher, J. Mueller, M. Schroeder, G. Staniford, and G. Wagner. Methodological Foundations for Agent-Based Systems. *The Knowledge Engineering Review*, 12(3), 1997.
[6] M. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *Proc. of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, WA, 1987.
[7] C. A. Iglesias, J. C. Gonzáles, and J. R. Velasco. MIX: A General Purpose Multiagent Architecture. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
[8] R. Kowalsky 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.
[9] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
[10] S. W. Locke, L. Sterling, L. Sonenberg, and H. Kim. ARIS: A Shell for Information Agents that Exploit Web Site Structure. In *Proc. of PAAM'98*, London, UK, 1998.
[11] M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In *Proc. of PAAM'98*, London, UK, 1998.
[12] D. L. Martin, A. J. Cheyer, and D. B. Moran. Building Distributed Software Systems with the Open Agent Architecture. In *Proc. of PAAM'98*, London, UK, 1998.
[13] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
[14] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1), 1996.
[15] M. Mulder, J. Treur, and M. Fisher. Agent Modelling in METATEM and DESIRE. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.
[16] D. T. Ndumu and H. S. Nwana. Research and development challenges for agent-based systems. *IEE Proc. of Software Engineering*, 144(1), 1997.
[17] H. S. Nwana, D. T. Ndumu, and L. C. Lee. ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. In *Proc. of PAAM'98*, London, UK, 1998.
[18] A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. of ICMAS'95*, San Francisco, CA, 1995.
[19] M. Spivey. *The Z Notation (second edition)*. Prentice Hall International, 1992.
[20] M. Wooldridge. Agent-based Software Engineering. *IEE Proc. of Software Engineering*, 144(1), 1997.
[21] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 1995.