Specification and Simulation of Multi-Agent Systems in CaseLP

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

Abstract

Nowadays software applications are characterized by a great complexity. It arises from the need of reusing existing components and properly integrating them. The distribution of the involved entities and their heterogeneity makes it very useful the adoption of the agent-oriented technology. The paper presents the state-of-the-art of CaseLP, an experimental logic-based prototyping environment for multi-agent systems. CaseLP provides a prototyping method and a set of tools and languages which support the prototype realization. At the system specification level, an architectural description language can be adopted to describe the prototype in terms of agents classes, instances, their provided and required services and their communication links. At the agent specification level, a rule-based, not executable language can be used to easily define reactive and proactive agents. An executable, linear logic language can define more sophisticated agents and the system in which they operate. At the implementation level, new primitives are defined to extend the target prolog-like language. Finally, simulation tools are integrated within CaseLP to visualize the prototype execution and to collect statistics on it.

Keywords: logic programming, multi-agent systems, prototyping.

1 Introduction

Nowadays applications are strongly characterized by their complexity. They are usually composed by heterogeneous and distributed entities which must cooperate and coordinate in an “intelligent” way to exchange and share knowledge.

The agent-oriented paradigm [22, 16] is an emerging technology that faces the problem of modeling such kinds of applications. It is suitable for modeling entities (agents) 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 arises (proactivity) without the intervention of human beings or other agents (autonomy). Societies of agents are called Multi-Agent Systems (MAS). They take
into account the distribution of the involved agents and the integration of heterogeneous software and data, fundamental aspects for the success of today’s software systems.

Unfortunately, at this time there is no evidence of a well-established engineering approach for building MAS-based applications. Due to the inherent complexity of those applications the prototyping approach seems more promising than the classical waterfall model thanks to its greater flexibility.

The paper presents the state-of-the-art of CaseLP (Complex Application Specification Environment Based on Logic Programming), an experimental logic programming based prototyping environment for MAS. The development of CaseLP started two years ago as an applied research on MAS rapid prototyping by means of logic programming [12]. The applications modeled during this period of time helped in understanding how to extend the framework. New features have been progressively added to cope with the emerging necessities.

CaseLP has been profitably used to develop prototypes in very different areas. Two applications were related to transport and logistic problems. The former has been developed in collaboration with FS (Italian railways) to solve problems of freight train scheduling along the railway between La Spezia and Milano [4]. The latter involved the planning of goods transportation, and has been realized in co-operation with Elsag s.p.a., an international company which provides service automation. Another application concerned the retrieval of medical information contained in distributed databases [18]. In this case CaseLP has been successfully adopted for a reverse engineering process. Finally the combination of agent-oriented and constraint logic programming techniques have been applied to the management of distributed database transactions [13].

The paper is structured as follows. Section 2 introduces CaseLP agents. Section 3 describes the six-steps method that is used to realize a prototype. Sections 4, 5 and 6 focus on languages and tools that can be used in different stages of the prototype development. Specifically, Section 4 presents tool and languages used in the modeling phase of the prototype. Section 5 describes instruments used in the implementation phase. Section 6 is related to the execution phase. Section 7 concludes the paper outlining the related and future work.

2 Agents in CaseLP

A CaseLP agent can be represented from two different perspectives. From a descriptive point of view, a CaseLP agent is characterized by a kind, an architecture, an interpreter and a set of services.

Kind identifies an agent as logical, interface, facilitator or manager. Logical agents provide control and coordination among MAS components using their reasoning capabilities. Interface agents provide an interface between external software modules and the agents in the MAS. Facilitator agents supply other agents with a yellow-pages service. Manager agents create and delete other agents in an application. Architecture refers to the internal architecture of the agent. An architecture speci-

1Facilitator and manager agents are still to be fully integrated in CaseLP.
fies both the data structures that form the agent’s internal components and the flow of control that masters components activity. Interpreter depends on the type of external module to which an interface agent is linked. The interpretation approach to software integration in a MAS [7] is adopted in CaseLP. External modules can be written in many programming languages. Finally, services define functionalities that the agent provides or requires for accomplishing its purpose. Services typically depend on the domain of application of the prototype. Some provided services can be exported to external users or other MAS. On the other hand, some services can be imported from outside the MAS. Thus, exported and imported services represent an interface between the MAS as a whole and the external world. They are the functionalities that the MAS provides or requires to external users, both humans or other MASs. Exported and imported services are a subset of the provided and required services set. If the CaseLP prototype is used as a decision support tool, at least one exported service must be provided in order to interact with the human user. Otherwise, if the prototype main purpose is simulation, the set of exported and imported services can be empty.

Abstracting from the architecture adopted, from a computational point of view a CaseLP agent is formed by a state, a behavior and an engine. State, behavior, and engine are architecture-dependent components. State includes data structures to represent the current situation of computation. Behavior represents knowledge used by the agent to accomplish its task, i.e. providing its services. According to the agent’s architecture, behavior can range form simple rules to complex plans. Engine encodes the task control that governs the flow of control of agent’s activities.

Interacting agents with the outlined features can be described in the CaseLP framework at different levels of abstraction, using the different languages and tools that the environment provides. A method guides the user in the various prototyping stages, from the informal description of the application to the development of a working prototype. In any step the user can adopt the more suitable languages and tools among the provided ones.

3 CaseLP prototyping method

The realization of a MAS-based software prototype can be performed according to the following steps. Figure 1 gives a graphical representation of the method.

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) for each class, determining the kind of the agents belonging to the class, their architecture, their eventual interpreters and their set of services; (c) determining the set of necessary instances for each class; (d) defining the interconnections between the instances of agents, linking appropriately requested and provided services. This phase defines the agents that form the prototype, and the communication channels among them. Notice that architecture can refer either to an architec-
ture already available in a library or to a “new” architecture that has still to be defined.

2. **Description of communication between agents.** Each provided or requested service needs a specific *conversation* between the agent that provides the service and the one that requires it. This step allows to specify the sequence of exchanged messages. Some conversations can start some other (sub)conversations. This is properly captured in this step. The *conversation model* of the MAS includes the set of all the conversations and the sub-conversation relationship among them.

3. **High-level specification of the system and the agents.** At this stage a high-level specification language is used to model the prototype. Three different levels of modeling can be identified:

   (a) specification of interactions among agents in the MAS, abstracting from their architecture and taking into account the interaction model specified in step 2;

   (b) specification of the “new” architectures chosen in step 1, i.e. modeling of the interactions between the internal components of an agent;

   (c) specification of the agents’ behavior.

4. **Verification and testing of the system.** This step verifies and/or performs a preliminary validation of the abstract specification defined in step 3.
Validation is possible if the specification language used in the previous step is *executable*. Verification is possible if proof mechanisms are available for the used specification language. If the specification is given using a logical language, goal-directed execution is a proof procedure. In this case, given a prototype specification and an initial configuration, it is possible to follow the evolution of a particular MAS 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.

5. **Implementation of the prototype.** This step transforms the abstract specification of the application into the final prototype. An extended concrete logic-programming language is used for prototype implementation. Interfaces towards external software and data are provided using appropriate *interpreters*. Moreover, message passing communication, and other lower-level details which were abstracted in the previous steps, are now implemented.

6. **Execution of the obtained prototype.** Prototype execution is used to *validate* the prototype against the client’s desiderata. Execution provides the user and the developer with information about the events that occur in the MAS, and allows them to check if the prototype behaves as expected and if it encompasses all the desired features. Any error or misbehavior discovered in this step may imply a revision of the choices made in the previous steps.

We can distinguish three different phases in the method adopted by CaseLP. Steps 1 through 4 form the *modeling phase* of the prototype. Step 5 represents the *implementation phase* and step 6 represents the *execution phase*. The next three sections focus on languages and tools that CaseLP provides to face each of the phases listed above.

4 **Prototype specification**

This section describes the languages and tools which can be used during prototype modeling.

4.1 **MAS-adl: a simple architectural description language**

In the first step of the method MAS-adl [4] is adopted. It is a simple, customized *architectural description language* [11] for MAS and it is used to describe the various classes of agents involved in the MAS and the interconnections between their instances. For any class, the architectural description gives information about the *kind*, the *architecture*, the *interpreters* and the *services* of agents in that class. Here is a template for the definition of an agent class using MAS-adl.
agentclass <ClassName> {
    kind: <Kind>;
    architecture: <ArchType>;
    interpreter: <IntName>;
    provides: <ProvServices>;
    requires: <ReqServices>;
    exports: <ExpServices>;
    imports: <ImpServices>;
}

Multiple instances of a class can be defined. Each service provided by an agent instance is linked to one or more services required by another instance. Conversely, several provided agent services can be linked to a single agent instance. The communication structure among agents can be set up by specification of links.

4.2 Tools for conversation model

The second step of the method describes the conversation model of the MAS. The agent communication language is a subset of KQML [14]. The conversation model is defined by choosing, for each service, the sequence of messages (conversation), as well as their performative and the content of each message. Some conversations may start in the middle of other conversations. For example, imagine agent a, requested for a service by agent b, that has to require an accessory service to agent c, in order to reply to agent b. A relation \( sc \) defines which conversations eventually start during other conversations. Let \( c_1 = \{m_1, \ldots, m_k\} \) be a conversation composed by messages \( m_1, \ldots, m_k \) and let \( c_2 \) be another conversation. \( c_1 sc_m, c_2 \) denotes that \( c_2 \) must start after message \( m_i \) in \( c_1 \) has been handled by the receiving agent. \( c_1 sc?_m, c_2 \) denotes that \( c_2 \) eventually starts after message \( m_i \) has been handled by the receiving agent. In the latter case, the decision about starting \( c_2 \) is up to the receiving agent.

4.3 \( \mathcal{E}_{hhf} \)

The linear logic language \( \mathcal{E}_{hhf} \) [6] is an executable language for modeling concurrent and resource sensitive systems based on the general purpose specification logical language Forum [15]. \( \mathcal{E}_{hhf} \) is a multiset-based logic combining features of extensions of logic programming languages like \( \lambda \)Prolog, e.g. goals with implication and universal quantification, with the notion of formulas as resources at the basis of linear logic. A \( \mathcal{E}_{hhf} \)-program is a collection of multi-conclusion clauses of the form:

\[
A_1 \not\subset \cdots \not\subset \not\subset A_n \leftarrow \text{Goal},
\]

where the \( A_i \)'s are atomic formulas, and the linear disjunction \( A_1 \not\subset \cdots \not\subset \not\subset A_n \) corresponds to the head of the clause. Furthermore, \( A \leftarrow B \) (i.e., \( B \sim A \)) is a linear implication. Clauses of this kind consume the resources (formulas) they need in order to be applied in a resolution step. \( \mathcal{E}_{hhf} \) provides a way to “guard” the application of a given clause. In the following extended type of clauses

\[
G_1 \& \ldots \& G_m \Rightarrow (A_1 \not\subset \cdots \not\subset \not\subset A_n \leftarrow \text{Goal}),
\]

the goal-formulas \( G_i \)'s must be solved in order for the clause to be triggered.
Referring to step 3 of the CaseLP development method, $E_{hhf}$ can be used to specify all the three levels of modeling. At the MAS level, a $E_{hhf}$ specification can prove useful to check properties of the communication among agents (for instance deadlock detection). At the architecture level, a $E_{hhf}$ specification includes details on agent data structures, and rules to model the architecture engine. At the behavior level, specific architecture dependent rules are defined. At any level, a $E_{hhf}$ 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. In [1] a $E_{hhf}$-based framework for MAS is defined. It is used to model MAS in which agents with heterogeneous architectures can co-exist. Note that one of CaseLP aims is to provide a set of specification languages together with their associated compilers into the prototype implementation language. The MAS developer will pick up and use the most suitable or familiar ones among them. At present $E_{hhf}$ is a mandatory choice if a high-level specification of the system is needed. The adoption of other high-level specification languages in CaseLP, as alternatives to $E_{hhf}$, is part of our future work.

4.4 The library of architecture specifications

The specification of an agent architecture is certainly one of the most difficult phases of the development method. To help the developer in this phase, CaseLP furnishes a library of already-tested architectures specified in $E_{hhf}$ from which he/she will pick up the desired model. Currently, the library contains four specifications: specification of reactive, proactive and reactive-proactive architectures, and specification of the Procedural Reasoning System (PRS) [8], fully described in [3].

4.5 ACLPL

ACLPL is a rule-based language that is used to program the behavior of reactive, proactive and reactive-proactive agents. The behavior of a reactive agents is expressed by means of event-condition-actions ACLPL rules. Proactive agents behave accordingly to condition-actions rules. In order to execute actions, an agent has to be (eventually) triggered by the perception of a particular event, currently implemented as a message sent by an agent, and condition has to be satisfied by the agent state. Actions are either updates of the agent state or the sending of messages to other agents in the system. Behavior for reactive-proactive agent is obtained mixing proactive and reactive rules. The syntax of ACLPL is sketched below.

Behavior ::= behavior RulesList endbehavior
RulesList ::= Rule | Rule RulesList
Rule ::= ReactiveRule | ProactiveRule
ReactiveRule ::= on message Msg check Condition do ActionsList
ProactiveRule ::= check Condition do ActionsList
Msg ::= Performative { content: Content; sender: Sender; receiver: Receiver; }
Condition ::= StateCondition and AuxiliaryCondition
StateCondition ::= true | Goal
AuxiliaryCondition ::= true | Goal
Goal ::= A | A and Goal | A or Goal
Action ::= assert_state(Fact) | retract_state(Fact) | send(Receiver, Msg)

An agent is given an initial state that is a (possibly empty) set of atomic ground formulas. A reactive rule is fired by a message taken from the agent’s mail-box, and if the condition is satisfied, corresponding actions are executed. Condition is actually formed by two distinct conditions. The former is about the agent state, and expresses what the state must contain in order to execute a sequence of actions. The latter is an auxiliary condition, and is actually a set of calls to auxiliary procedures. If these calls succeed, the auxiliary condition is satisfied and the sequence of actions is then executed. An auxiliary procedure is either a prolog-like clause defined in the agent code, or a built-in procedure. If the prototype contains only reactive, proactive or reactive-proactive agents, ACLPL can be used in step 3 of the development method as alternative to $\mathcal{E}_{bf}$. Use of ACLPL is limited to the behavior specification level. Notice that the syntax of ACLPL could be extended in such a way behavioral specification of more complex agents can be performed.

5 Prototype implementation

Step 5 of the method deals with implementation of a “concrete” version of the prototype. The target language for implementation is Prolog (Prolog for Agents), an extended platform-independent prolog language.

5.1 Extending Prolog to develop agents

Prolog is defined as standard Prolog extended with communication capabilities and with safe state updates. Two predicates allow an agent to update its state.

- `assert_state(Fact)` asserts Fact to the agent’s state. Its implementation is safe since, when the predicate is backtracked, the previously asserted fact is removed from the state.

- `retract_state(Fact)` removes the first fact in the agent’s state that unifies with Fact. If it is backtracked, the fact is re-asserted.

Three predicates allow an agent to inspect its mail-box and read messages from it.

- `sync_receive(Message)` blocks the calling agent until a message unifying with Message enters the mail-box.

- `async_receive_one(Msg_input, Msg_output)` searches the calling agent’s mail-box for a message unifying with Msg_input. If such a message is present, Msg_output is unified with it, otherwise Msg_output unifies with the atom no_message.

---

2The CaseLP environment, initially developed in ECLiPSe, has been successively implemented also in SICStus Prolog. Prolog is assumed as the language for prototype implementation; automatic compilers have been developed from Prolog to ECLiPSe and SICStus Prolog.
• async_receive_all(List_of_messages) fetches all the messages in the the mail-box of the calling agent. These messages are unified with List_of_messages.

One predicate allows an agent to send a message.

• send(Receiver, Message) sends Message from the calling agent to the agent Receiver. The send predicate is implemented using the safe update predicates assert_state and retract_state. These predicates are actually called to execute updates to the agents’ mail-boxes.

5.2 Implementing a working prototype

To implement a working prototype, the proper Prolog code must be supplied for any agent in the system. This code must

1. implement all the computational features characterizing an agent: its initial state, its behavior and its engine; and

2. respect the specifications of the agent and of the system given in steps 1 through 3 of the method.

For interface agents, the Prolog code must also define the interpreter predicates.

The ability of automatically translating the agents’ specifications into Prolog code and of reusing already implemented structures, should avoid the burden of a hand-made prototype implementation developed by scratch. The research in this direction is quite recent and far from resulting into a completely automatic translation process, but some preliminary, promising results have already been obtained.

In [3] an engine for the PRS architecture [8] have been specified using \( \mathcal{E}_{hhf} \). An approach to the translation of \( \mathcal{E}_{hhf} \) rules into Prolog code has also been given. The \( \mathcal{E}_{hhf} \) clauses used for specifying the PRS engine assume the general form

\[
G_1 \& \ldots \& G_m \Rightarrow (A_1 \& \ldots \& A_n \leftarrow G_{m+1} \& \ldots \& G_{m+k} \& A_{n+1} \& \ldots \& A_{n+h}),
\]

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

\[
p_C := \text{retract_state}(A_1), \ldots, \text{retract_state}(A_n), \ G_1, \ldots, G_{m+k}, \\
\text{assert_state}(A_{n+1}), \ldots, \text{assert_state}(A_{n+h}).
\]

where retract_state and assert_state are the predicates for state update described in Subsection 5.1. 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. Applying this transformation to every \( \mathcal{E}_{hhf} \) clause, a corresponding set of Prolog clauses is obtained. 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 engine of a Prolag agent is defined by a predicate cycle. The cycle defining the PRS engine results into:

\[
\text{cycle} :\text{-} \text{perception; plan\_triggering; plan\_execution; action\_execution.}
\]

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

\[
\text{plan\_triggering} :\text{-} p_{C_1}; \ldots ; p_{C_s}.
\]

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

Another issue that have been considered is the compilation from ACLPL into Prolag. Due to their rule-based nature, ACLPL specifications can be easily translated into a form that can be manipulated by a reactive-proactive engine. A working compiler has been implemented, which takes as input an ACLPL program and generates a code of this form.

\[
\text{reactive\_rule}(\text{Message, Condition, Actions}).
\]

\[
\ldots
\]

\[
\text{proactive\_rule}(\text{Condition, Actions}).
\]

\[
\ldots
\]

\[
p_1(\ldots) :\text{-} \ldots 
\]

\[
\ldots
\]

\[
p_m(\ldots) :\text{-} \ldots 
\]

\text{reactive\_rule} is a reactive rule, \text{proactive\_rule} is a proactive rule and \( p_1, \ldots, p_m \) are Prolag predicates.

The cycle for a reactive-proactive agent may have the form

\[
\text{cycle} :\text{-}
\]

\[
\text{async\_receive\_one}(\text{Msg\_input, Msg\_output}), \\
(\text{Msg\_output == no\_message ->} \\
\text{Actions1} = [\ldots]; \\
\text{select\_reactive\_rule}(\text{Msg\_output, Actions1}) \\
),
\text{select\_proactive\_rule}(\text{Actions2}),
\text{choose}(\text{Actions1, Actions2, Actions}),
\text{execute}(\text{Actions}).
\]

The predicate cycle defines a simple engine for the agent architecture. First, the mail-box is inspected and a message is picked up from it. Then, a fired reactive rule is selected from the agent behavior and the set of associated actions is returned. Selection among proactive rules is also performed, and a set of proactive actions is returned. Finally, a set of actions is chosen and executed. Obviously, more sophisticated strategies for rules selection and actions choice can be taken into a more complete engine. The integration in CaseLP of the compiler for ACLPL and the engines for reactive, proactive and reactive-proactive architecture has still to be completed.
6 Prototype execution

After all the agents have been implemented as Prolag pieces of code, they can be compiled into the target language and then loaded into the memory, to make their code available for running the simulation. The compile_agent primitive has been implemented to take more than one file as its input: this furthers the reuse of the same Prolag file for agents which differ only in small portions of their state or behavior. The output of this primitive is a file containing a SICStus Prolog or an ECLiPSe module.

Once all modules have been generated, the system developer must choose if he/she wants to load the agents, initialize their mail-boxes, start the simulation and follow its execution by means of a graphical interface, or issuing commands within the SICStus Prolog or ECLiPSe shell.

6.1 The CaseLP Visualizer: a graphical simulation interface

The CaseLP Visualizer [17] provides documentation about events that happen at the agent level during MAS execution. According to the developer needs, the code of some agents can be automatically instrumented after it has been loaded. Instrumentation adds probes to agents code; events related to state changes and/or exchanged messages can be recorded and collected for on-line and/or off-line visualization. Windows for initializing some agents mail-boxes and for setting the simulation length and the on-line visualization granularity\(^3\) are provided. During the simulation, views related to instrumented agents are shown. At the end of the simulation a more complete trace of all the instrumented events can be visualized. Instrumentation is completely independent from execution. This will not influence a possible future change of the execution support.

Figure 2 shows the off-line trace of execution of the prototype for train traffic management [4].

6.2 Shell commands for the simulation execution

If the developer prefers to type in commands within a shell, he/she can use a set of primitives to execute the simulation.

To load an agent from a file, load(File_name, Agent_name) can be used. initialize(List_of_initial_messages) is used to initialize the agents’ mail-boxes. global_parameters(Iter, Min_time_unit, Out, Verbose) sets the global simulation parameters. Iter is the number of iterations the simulator will perform, expressed in the unit Min_time_unit. Out is the name of the file where all exchanged messages will be collected, and Verbose is a flag setting whether the messages have to be visualized on the monitor or not, during the simulation run.

net_parameters(From_ag, To_ag, Failure_rate, Min_latency, Max_latency, Unit) allows to define, for any couple of interacting agents, the failure rate and

\(^3\)The simulation must stop to visualize the events occurred. The periodicity of these interruptions (after any occurred event, after 50 occurred events, \ldots) is decided by the developer.
the minimum and maximum latency of the communication channel. _Failure_rate_ sets how many messages, on 100,000,000, are lost due to communication problems. _Min_latency_ and _Max_latency_ are the minimum and maximum values, expressed in _Unit_, of the delay over the communication channel. If, for a couple of communicating agents, _net_parameters_ is not defined, the channel between them is assumed to be without delay and completely reliable.

_start_simulation_ starts the execution of the prototype built during the loading stage, respecting the parameters described above. The time-driven simulation starts, with a round-robin scheduler which activates in turn all the agents in the system, allowing them to call their _cycle_ predicate.

Finally, after the simulation execution has stopped, the primitive _get_statistics_ can be used to collect information on it. It allows setting the start time, the end time and the length of intervals in which the simulation period is divided, as well as the “weights” (representing number of bytes, importance, ...) assigned to the monitored messages.

The file where statistics are collected contains, for any simulation interval, the number of messages received in that interval and the sum of their weights. The total number of exchanged message, the maximum and average number of messages per interval, the maximum and average weight per interval, and the weight standard deviation are also provided.

This information proves useful to evaluate the communication channels occupation, which is of fundamental importance in those applications where the communication bandwidth is limited.

### 7 Related and future work

The tools and the method constituting _CaseLP_ represent an _agent-based approach to software prototyping_. The potential of such an approach has been demonstrated...
by the adoption of a MAS-based prototyping technology for developing applications in very different areas. The choice of Logic Programming as the base for our prototyping environment has proven successful, as discussed in [2]. The modularity and expressiveness of LP are extremely useful to describe in a clear and concise fashion the complex behavior of agents. It also proves useful to model the non-deterministic MAS execution, the meta-reasoning capabilities of agents and to support the integration of external software.

There are several environments for MAS specification and implementation that are based on logic languages. Logic Programming has been adopted by Kowalsky and Sadri [10] for developing an architecture that unifies rationality with reactivity. All the interesting objects that form the architecture are represented using a logical formalism and the agent task control is performed by a meta–interpreter that executes a perception, reaction and proof procedure cycle. Wagner [21] takes an approach similar to the previous one. He defines Vivid Agents and Vivid Reagents that is, respectively, rational ad reactive agents whose behavior is represented by action and reaction rules. The exploitation of logic programming to realize applications based on MAS can be found in Schroeder et al. [20], which present a formalism for specifying and implementing diagnostic agents based on extended logic programming.

Other not-executable logic formalisms have been used to define agent languages: ConGolog [9] is a concurrent multi–agent programming language based on a logical theory of action, while AgentSpeak(L) [19] uses a restricted first–order language with events and actions to model the internal mental state of an agent and its behavior.

Our future work will mainly aim at improving different aspects of our environment and in evaluating its applicability in new areas:

**Distribution of the simulation.** From a simulation point of view, CaseLP is a time-driven, centralized simulator, with a global time known from all the agents in the system. To improve the efficiency of CaseLP it is necessary to change the simulation from time-driven to event-driven, and to distribute it.

To cope with the distribution of simulation, CaseLP agents should be equipped with additional data structures. Any agent should be simulated by an active process. A local simulation engine should be added to any agent to implement its life cycle, which would mainly consist in the inspection of the communication channels and the internal events list and the management, if possible, of the received event(s).

**Integration of specification languages and legacy software.** As already observed, the set of available specification languages needs to be extended, and tools for animating the not executable ones should be provided. Also the set of languages and tools which can be interfaced by means of an interface agent should be augmented. These extensions will be faced within the ARPEGGIO project [5]. The purpose of ARPEGGIO (Agent based Rapid Prototyping Environment Good for Global Information Organization) is the integration of different research experiences based on Logic Programming, including CaseLP, into a common joint project. It will lead to the development of a general open framework for the specification, rapid prototyping and engineering of agent-based software.
Development of new applications in hot areas. One of the wider field of application of MAS technology is the Web. We are extending CaseLP with ontologies [23] in order to semantically integrate data coming from different domains, which is a typical situation when exploring and collecting data from the Web. However, we have not yet developed a Web model and our research in this direction is still at the beginning.

Acknowledgements

Part of this research has been realized while Floriano Zini was visiting the Department of Computer Science & Software Engineering at the University of Melbourne.

References


