A Logic Programming Framework for Component-Based Software Prototyping

Maurizio Martelli, Viviana Mascardi, Floriano Zini

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

Abstract

The paper presents CaseLP, a logic-based prototyping environment for specifying and verifying complex distributed applications. CaseLP provides a set of languages for modeling intelligent and interacting components (*agents*) at different levels of abstraction. It also furnishes tools for integrating legacy software into a prototype.

The possibility of integrating, into the same executable prototype, agents which are only specified as well as already developed components can prove extremely useful in the engineering process of complex applications. In fact, the reusability of existing components can be verified before the application has been implemented and the developer can be more confident on the correctness of the new components specification, if it has been executed and tested by means of an interaction with the existing components.

Besides the aspects of integration and reuse, CaseLP also faces another fundamental issue of nowadays applications, namely distribution. The components which constitute the prototype are *logically* distributed. The features of the network (latency and reliability of the communication channels between agents) can be set by the prototype developer, thus allowing a realistic simulation of a *physically* distributed application.

1 Software Development Challenges

Despite thirty years of research and experience and many successful results, software systems are still difficult to engineer to guarantee correctness and reliability. This is particularly true for distributed systems, where a set of entities have to cooperate and coordinate for exchanging information coming from diverse existing sources. Hence, integration and reusing of different kinds of information and software tools is an urgent necessity that new software products have to face.

The agent-oriented paradigm [28, 24] is an emerging technology which allows a high level model of applications in which many autonomous, intelligent and interacting entities (a Multi-Agent System or MAS) cooperate to achieve a common goal or compete to satisfy personal needs. We assume a loose "declarative" definition that an agent is an autonomous, social, reactive and proactive piece of software that provides some services and is able to communicate with other agents using a common agent communication language.

Our interest in intelligent agents is driven by the firm belief that they can be profitably adopted in building prototypes where existing components coexist with components which are only specified. A prototype of this kind should prove extremely useful for two reasons.

- The problem of testing the behavior of already developed software when used for a purpose different from its original one, is fundamental in order to decide if the software can be easily reused or not. If a prototype for a new application is developed and the existing software is plugged into it, it is easier to understand how it behaves in this application and to individuate if and which modifications are necessary.

- The new components of the application under development need to be specified and tested, before being implemented into the final implementation language. The possibility of testing their specification (embedding it in an environment which is partly implemented) should allow the developer to reach a greater confidence in the correctness of the specification.

Moreover, the agent-oriented technology provides the support for communication and interaction among the prototype components (both implemented and specified) by means of a high level communication language. If mechanisms are provided to simulate the features of communication channels between the logically distributed agents, very realistic simulations can be performed, which can give important feedbacks for the application implementation.

In this paper we describe CaseLP, an environment which allows software engineers to prototype specifications for facilitating the development of correct heterogeneous software. The base for CaseLP is Logic Programming (LP) [16], a high level programming paradigm which blurs the distinction between specifications and code. Logic Programming has proven very suitable in the definition of our environment, mainly because it can provide the right mix between formalism and experimentation. It is a high level language, amenable for formal reasoning. A logic program is an executable specification and this encourages rapid prototyping. A logical language can naturally be the target language for the animation of many not executable specification languages [27]. On the side of integration of heterogeneous systems, Logic Programming can profitably address semantic integration that is the process of specifying methods to resolve conflicts, pool information together and define new operations based on operations belonging to different domains [17]. Further, meta-programming features of Logic Programming can be flexibly exploited to define agents with different architecture and control [14, 5], in such a way that many kinds of agents, suitable for different tasks, can be encompassed into an application.

The paper is structured as follows. Next section describes the main features of CaseLP. Section 3 describes the tools and languages provided by our environment, which can be used in the MAS modeling phase. Section 4 faces the problems of legacy software integration, distributed agents integration and semantic integration. Finally, Section 5 concludes the paper.

2 The Rationale Behind CaseLP

The development of CaseLP started as an applied research on rapid prototyping of MAS by means of logic programming [18].

The work was carried on with the aim of overcoming the rigid solutions of the first prototyping environment, experimenting different languages to implement it, exploiting their features in order to gain in efficiency, adding new functionalities to the tool and developing a well-defined methodology to guide the prototype developer. Various directions were followed:

- **Definition of an agent model.** A great effort was devoted to make CaseLP agents as flexible as possible, avoiding to commit to a specific agent architecture or type. From a *descriptive* point of view, a CaseLP agent is characterized by:

  - a *kind*;
  - an *architecture*;
  - an *interpreter*;
  - 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[1]. *Architecture* refers to the internal architecture of the agent. An architecture specifies 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. Finally, *services* define functionalities that the agent provides or requires for accomplishing its purpose. 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. Services typically depend on the domain of application of the prototype.

Abstracting from the architecture adopted, from a *computational* point of view a CaseLP agent is formed by three components:

- a *state*;
- a *behavior*;
- 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.

- **Development method.** A six-steps method has been developed to guide the prototype developer from the informal description of the application to the execution of the working prototype. [19] describes the most recent version of the method.

- **Application modeling.** CaseLP embeds a set of languages which can be used to model the application at different abstraction levels. Section 3 describes these specification languages and their purpose.

- **Integration.** As far as software integration is concerned, CaseLP uses *interface* agents to access external modules. Also the problems of *distributed agents integration* and *semantic integration* are considered in our environment, as described in Section 4.

- **Simulation and execution.** CaseLP has been extended with a set of utilities to define the system parameters (the communication channels latency and reliability, the granularity of simulation time, ...), to trace its execution on and off-line, and to collect statistics on it. A graphical interface [25] supports these activities in a more user-friendly fashion. [19] describes the simulation utilities in detail.

- **CaseLP implementation.** CaseLP was initially implemented in ECLiPSe [1] and a version running under SICStus Prolog [13] was then developed. Both ECLiPSe and SICStus Prolog allow to partition the knowledge base into *modules*. This proved useful to avoid the explicit implementation of a multi-theory system, thus gaining in efficiency and modularity. A centralized round-robin scheduler activates in turn each agent (module) loaded into the system. The currently active agent performs all the actions defined by its cycle predicate. These actions can involve primitives for getting and processing incoming messages, or can depend on the internal state of the agent. The activation of an agent may cause the agent itself to send messages and/or to update its state, according to the agent architecture and behavior. When the agent terminates the execution of actions in its cycle, the scheduler can activate the next agent. To make the development of prototypes running both under ECLiPSe and under SICStus Prolog easier, agents can be implemented in Prolag (*PROLog for AGents*). It

[1] Facilitator and manager agents are still to be fully integrated in CaseLP.

AgentClassDef ::= **agentclass** AgentClassName {ClassDef}
ClassDef ::= **kind**: Kind; **architecture**: Architecture;
    interpreters: InterpretersList; RequiredAndProvidedServices
Kind ::= logical | interface | facilitator | manager | ...

Architecture ::= reactive | proactive | prs | ...

RequiredAndProvidedServices ::= **requires** ServicesList; **provides** ServicesList;
    import ServicesList; **export** ServicesList;

AgentInstancesDef ::= **agentInstances** {AgentInstDefList }
AgentInstDef ::= NumberOfInstances AgentName AgentClassName

LinksDef ::= **link** {LinkList}
Link ::= OneToOneLink | OneToManyLink | ManyToOneLink
OneToOneLink ::= AgentNameAndService ← AgentNameAndService
OneToManyLink ::= AgentNameAndServiceList ← AgentNameAndService
ManyToOneLink ::= AgentNameAndService ← AgentNameAndServiceList
AgentNameAndService ::= AgentName.Service | AgentName*.Service

Figure 1: MAS-adl: a language for MAS architectural description.

is an extended platform-independent prolog language we have defined and for which we have implemented compilers into ECLiPSe and SICStus Prolog.

- **Case studies.** CaseLP has been adopted for the development of prototypes in very different areas. Among the modeled applications, two of them 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 [8]. The latter involved the planning of goods transportation [20], 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 [26]. 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 [21].

3 Modeling Complex Applications

CaseLP supports the prototype modeling phase by providing a set of languages and tools among which the developer can choose the more suitable or familiar ones. At present the languages and tools embedded in CaseLP are the following five. The addition of new ones is part of our future work.

3.1 MAS-adl: a simple architectural description language

MAS-adl [8] is a simple, customized *architectural description language* [15] 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.

The syntax for MAS-adl is sketched in Figure 1. AgentClassDef defines the classes of agents used in the MAS under construction. AgentInstancesDef defines the instances of classes previously defined i.e., the agents that constitute the MAS. LinksDef defines the directed links between instances of agents, from a service provided by an agent to a service required by another agent. AgentClassName and AgentName are strings of characters. InterpretersList is a list of interpreters, each for any external domain to which the agent is interfaced. Service is one of the services that have been individuated for the application.

3.2 Tools for conversation model

The conversation model of the MAS is based on a subset of KQML [22]. 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. $c_1 sc_{m_i} c_2$ denotes that c_2 must start after message m_i in c_1 has been handled by the receiving agent. $c_1 sc?_{m_i} c_2$ 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.

3.3 $\mathcal{E}_{hhf}$

The linear logic language $\mathcal{E}_{hhf}$ [10] is an executable language for modeling concurrent and resource sensitive systems based on the general purpose specification logical language Forum [23]. $\mathcal{E}_{hhf}$ 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 basis of linear logic. A $\mathcal{E}_{hhf}$-program is a collection of multi-conclusion clauses of the form:

$$A_1 \parr \ldots \parr A_n \multimapinv Goal,$$

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, $A \multimapinv B$ (i.e., $B \multimap A$) 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

$$G_1 \,\&\, \ldots \,\&\, G_m \;\Rightarrow (A_1 \parr \ldots \parr A_n \multimapinv Goal),$$

the goal-formulas G_i's must be solved in order for the clause to be triggered.

$\mathcal{E}_{hhf}$ can be used to specify the overall MAS behavior, the engine for a specific architecture and the single agent behavior. At the MAS level, a $\mathcal{E}_{hhf}$ specification can prove useful to check properties of the communication among agents (for instance *deadlock detection*). At the architecture level, a $\mathcal{E}_{hhf}$ specification includes details on agent data structures, and rules to model the architecture engine. At the agent behavior level, specific architecture dependent rules are defined. At any level, a $\mathcal{E}_{hhf}$ 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 [3] a $\mathcal{E}_{hhf}$-based framework for MAS is defined. It is used to model MAS in which agents with heterogeneous architectures can coexist.

Hints for a semi-automatic translation from $\mathcal{E}_{hhf}$ into Prolag can be found in [6] and [19].

3.4 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 $\mathcal{E}_{hhf}$ 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)* [12], fully described in [6].

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)

Figure 2: ACLPL syntax.

3.5 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. Figure 2 presents the ACLPL syntax. 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 as an alternative to $\mathcal{E}_{hhf}$ to model the agent behavior. Notice that the syntax of ACLPL could be extended in such a way behavioral specification of more complex agents can be performed.

A description of how to translate an ACLPL agent into a Prolag piece of code can be found in [19].

4 Integration and Distribution

Integration and distribution are two strictly related issues: the legacy software to be integrated into the prototype does not necessarily reside on the same computer where CaseLP runs. Moreover, CaseLP agents themselves could be distributed over a network, and they should be integrated into a working prototype in order to carry out the application simulation.

To make CaseLP a component-based prototyping system, we have faced both the problems of *legacy software integration* and *distributed agents integration*. We have also taken under consideration *semantic integration*, introducing *ontologies* as a part of our environment.

4.1 Legacy software integration

The problem of integrating legacy software has been faced in CaseLP following an *interpretation* approach [11]. For any external module which has to be integrated into the prototype, an *interface* agent is defined. This agent translates the messages coming from the other agents into the right procedure call (or method invocation, or database query, according to the module to integrate). The results have to be translated back into an answer the requesting agent can understand. From a conceptual point of view this approach does not present any difficulty: the interface agents know how to request the services exported from the module, and how to "speak" with the other agents. Some problems can arise at a lower level: while it is clear, for example, how to access a C function from an ECLiPSe module (ECLiPSe provides a built-in interface towards C), how should we access a database, or a Java method, if ECLiPSe does not provide an interface towards them? When possible, the easiest way to proceed is to find a "bridge language" for which both the CaseLP implementation language and the software to integrate provide an interface. This approach has been successfully experimented for integrating an Oracle database within an ECLiPSe prototype. In this case the "bridge language" was C. Other external modules can be integrated by adopting this method. Note that once an interface agent has been defined, it can be reused in different prototypes: we aim at developing a *library* of interface agents for many external modules.

Figure 3 shows how this approach has been followed in a real situation. We are currently working on the development of a prototype for spatial reasoning. We have to integrate an ECLiPSe module which manages combinatorial maps by means of simplification operations [4]. We also need to integrate the C++ library CGAL to allow the visualization of the maps and their interactive manipulation. The maps we deal with are stored in an Oracle database. We are developing interface agents for all of these components. A logical coordinator agent will process the user requests and will decide the proper sequence of actions to perform in order to satisfy the query. The actions should involve for example the retrieval of a map from the database, its processing by means of the spatial reasoner and the visualization of the resulting map by means of the CGAL graphical interface. All these actions require an access to external modules via their interface agents.

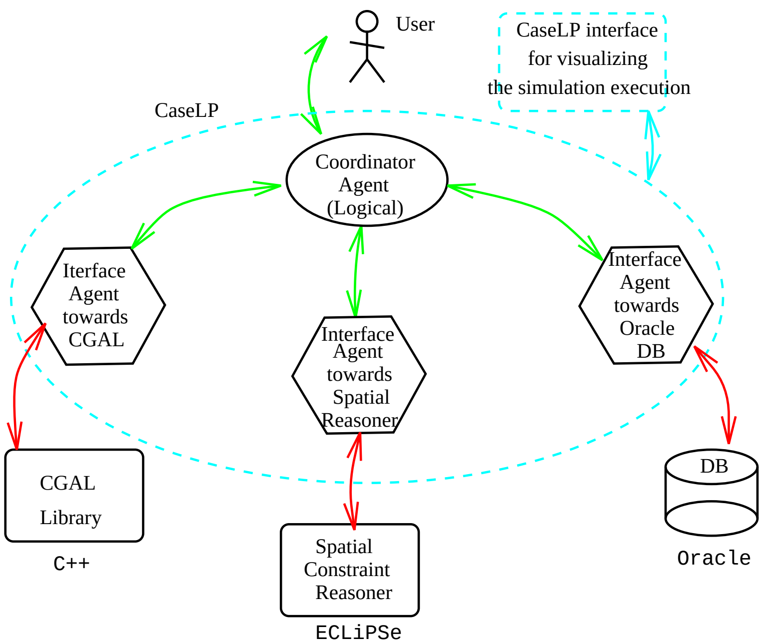

Figure 3: A MAS prototype for spatial reasoning.

The problem of integration of heterogeneous information sources can be faced in very different ways and the interpretation approach is among the simplest ones. We are part of a joint project, ARPEGGIO (*Agent based Rapid Prototyping Environment Good for Global Information Organization*) [9], together with the Department of Computer Science at the University of Maryland, USA and the Department of Computer Science & Software Engineering at the University of Melbourne, Australia; the American partners of the project have been working for years on the *mediation approach to software integration*, developing the mediated system HERMES [17, 2] and the *Interactive Maryland Platform for Agents Collaborating Together*, IMPACT which builds upon it. The purpose of ARPEGGIO is to combine different research experiences based on Logic Programming in a general open framework for the specification, rapid prototyping and engineering of agent-based software, and *integration* is one of the key issues both for the aspect of *external modules integration* and for the aspect of *high level specification languages integration.*

Our contribution to the project is mainly related to the aspects of modelization of the application and integration of different specification languages into the executable prototype. Nevertheless, the aspect of external software integration is central in our research.

4.2 Semantic integration

CaseLP approach to semantic integration is based on the *ontology* concept.
Ontologies are *content* theories about objects, their properties, and relationships among them that are possible in a specific domain of knowledge [7]. In knowledge-based systems, an ontology is that part of the system which specifies what things exist and what is true about them. In the setting of knowledge-based agents and multi-agent systems, an ontology is a description of the concepts and relationships that can exist for an agent or a community of agents. Roughly speaking, ontologies specify the vocabulary used to talk about both the agents, their characteristics and functionalities, and the domain in which agents operate. It is an "official" set of attributes and relationships related to the agents and their domain of application.

In a MAS context, an ontology is designed for the purpose of enabling knowledge sharing and reuse. A ontological commitment is an agreement between agents to use a vocabulary in a way that is consistent (but not complete) with respect to the theory specified by an ontology.

From the point of view of a single agent in a MAS, three different sources of knowledge may be necessary:

- domain knowledge;
- agent knowledge;
- computational knowledge.

Agents manipulate information related to some particular application domain. Domain knowledge can be incorporated into a domain ontology. A proper structuring of domain dependent knowledge could profitably be shared among different agents and/or reused for building multi-agent systems that operates on similar domains, either from the point of view of enclosed objects, or the point of view of structuring of relationships among objects.

On the other hand, agents need to know characteristics and properties of other agents with which they interact. Agent knowledge is expressed in an agent ontology, which describes features that do not depend on the domain to which a multi-agent system is applied. An agent ontology can be used to realize several MAS that operate on different application domains.

Finally, computational knowledge is formed by agent state and agent behavioral rules and plans, and it is exploited by agents to execute their tasks.

Domain and agent level ontologies are joined in CaseLP to allow analysis of MAS architectural description and single agent behavior. Agents exploit domain level ontology during computation as knowledge repositories. In [29], the extension of CaseLP with ontologies is described. A domain ontology is given for an application that finds the "best" sporting city in Italy. The prototype MAS compares city results from most popular Italian sports and, according to a heuristic analysis, the best city is returned.

4.3 Distributed CaseLP

At present the CaseLP core is a scheduler which in turn activates the agents in the system. We simulate a concurrent execution of the agents by means of a sequential activation of them. 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.

If we want to improve the efficiency of CaseLP as a simulation tool, we need to operate at two different levels: we need to change the simulation from time-driven to *event-driven*, and to *distribute* it. The first modification proves useful when the events are quite rare and nothing happens in many time ticks: in the current setting of our environment, agents are activated even if they have no messages to process and no state-dependent actions to perform. As far as distribution is concerned, the absence of a global clock makes it necessary to adopt synchronization mechanisms in order to let the agents coherently simulate the time evolution. If a *conservative* approach is followed, deadlock avoidance or detection/recovery mechanisms must be provided, while an *optimistic* approach needs rollback mechanisms in order to ensure the respect of causality between events. In any case, the algorithms for dealing with distribution are quite sophisticated, even if they are well-established and tested.

To cope with the distribution of simulation, CaseLP agents should be equipped with an additional data structure for *internal events* which is not necessary in a centralized, time-driven simulator. Any agent should be simulated by an active ECLiPSe process. Communication among these processes should happen by means of sockets; we have already tested the feasibility of this approach. A local simulation engine should be added to any agent to implement its life cycle. It 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). The agents should be implemented taking into account that only the occurrence of an event (external or internal) can permit the agent to evolve. We should evaluate how this constraint affects the implementation of pro-active and rational agents.

5 Conclusions

CaseLP environment can be considered as an *agent-based approach to software prototyping*. The potential of such an approach has been demonstrated by the adoption of CaseLP for developing applications in very different areas.

The future directions of our research will deal with the improvement of CaseLP as a simulation tool, thus distributing the simulation over a computer network, and developing *interface agents* for new external components. The implementation of a graphical interface for the interactive description of the MAS as a graph of communicating agents is under study, as well as the improvement of the existing CaseLP Visualizer for the on and off-line tracing of the MAS execution. The set of specification languages supported by our environment can be extended, and the semi-automatic compilation process from the existing ones into Prolag needs to be completely automatized. Finally, as far as semantic integration is concerned, we are planning to realize an *ontology manager* for automatic creation and updating of ontologies.

References

[1] 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, 1995.

[2] S. Adali and V.S. Subrahmanian. Intelligent Caching in Hybrid Knowledge Bases. In N. Mars, editor, *Proc. 1995 Intl. Conf. on Very Large Knowledge Bases*, pages 247–256, Twente, The Netherlands, May 1995. IOS Press.

[3] A. Aretti. Semantica di sistemi multi-agente in logica lineare. Master's thesis, DISI – Università di Genova, Italy, 1999.

[4] M. Bertolotto. *Geometric Modeling of Spatial Entities at Multiple Levels of Resolution.* PhD thesis, DISI – Università di Genova, Italy, 1998.

[5] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In *The Logic Programming Paradigm. A 25-Year Perspective.* Springer-Verlag, 1999.

[6] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In G. Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages*, San Antonio, Texas, January 1999. Springer Verlag. LNCS 1551.

[7] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. What Are Ontologies, and Why Do We Need Them. *IEEE Intelligent Systems*, 14(1):20–26, Jan/Feb 1999.

[8] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. An Agent Based Prototype for Freight Trains Traffic Management. In *Proc. of FMERail'99 Workshop (satellite event of FM'99)*, Toulouse, France, September 1999.

[9] 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. In *Proc. of STEP'99*, Pittsburgh, PA, September 1999.

[10] G. Delzanno. *Logic & Object–Oriented Programming in Linear Logic.* PhD thesis, Università di Pisa, Dipartimento di Informatica, 1997.

[11] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, 1994.

[12] M. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *Proc. of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, WA, 1987.

[13] The Programming System Group. *SICStus v3 User's Manual.* Swedish Institute of Computer Science, Kista, Sweden, 1995.

[14] 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.

[15] J. Kramer. Distributed Software Engineering. In B. Fadini, editor, *Proc. of the 16th International Conference on Software Engineering*, pages 253–266, Sorrento, Italy, 1994. IEEE Computer Society Press.

[16] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[17] J. Lu, A. Nerode, and V.S. Subrahmanian. Hybrid Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, October 1996.

[18] M. Martelli, V. Mascardi, and F. Zini. Applying Logic Programming to the Specification of Complex Applications. In M. Falaschi, M. Navarro, and A. Policriti, editors, *Proc. of Appia–Gulp–Prode 1997: Joint Conference on Declarative Programming*, pages 491–499, Grado, Italy, June 1997.

[19] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, September 1999.

[20] V. Mascardi. Un sistema multi-agente logico. Master's thesis, DISI – Università di Genova, Italy, 1996.

[21] V. Mascardi and E. Merelli. Agent-Oriented and Constraint Technologies for Distributed Transaction Management. In *Proc. of IIA and SOCO*, Genova, Italy, June 1999. ICSC Academic Press.

[22] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.

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

[24] D. T. Ndumu and H. S. Nwana. Research and development challenges for agent-based systems. *IEE Proc. of Software Engineering*, 144(1), 1997.

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

[26] G. Persano. Gestione distribuita di informazioni mediche mediante tecniche multi-agente. Master's thesis, DISI – Università di Genova, Italy, 1998.

[27] L. Sterling, P. Ciancarini, and T. Turnidge. On the Animation of "not Executable" Specifications by Prolog. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.

[28] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 1995.

[29] F. Zini and L. Sterling. Designing Ontologies for Agents. In *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, September 1999.