Designing Ontologies for Agents

Floriano Zini[1] Leon Sterling[2]

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

(2) Department of Computer Science & Software Engineering
University of Melbourne, Parkville 3052, Australia
leon@cs.mu.oz.au

Abstract

This paper discusses an approach to adding explicit ontologies in multi-agent systems based on logic programming. Ontologies are content theories about knowledge domains, developed to clarify knowledge structure and enhancing knowledge reuse and standardization. Ontologies allow explicit organization of knowledge in agent-based applications, and unambiguous description of characteristics and properties of agents. We consider in detail the use of explicit ontologies in CaseLP, a declarative logical framework for prototyping agent-based applications. Our running example comes from the domain of sport results, to which CaseLP has been applied. Concepts such as sport, competition, competitors are included in the ontology, as well as relationships that relate these concepts. We introduce an agent level ontology to formalize attributes and functionalities of CaseLP agents, for example their kind, architecture and services, either at the domain level or at the agent level. Domain and agent level ontologies are exploited in CaseLP to perform semantic checks of agent architectural descriptions, to check agent behavioural rules used by an agent to provide its services, and as a knowledge repository to be exploited during agent execution.

Keywords: Ontologies, Multi-Agent Systems.

1 Ontologies and their use in multi-agent systems

Ontologies are *content* theories about objects, their properties, and relationships among them that are possible in a specific domain of knowledge [3]. In knowledge-based systems, an ontology is that part of the system which specifies what things exist and what is true about them. Ontological analysis clarifies the structure of knowledge. An ontology related to a domain forms the heart of any system of knowledge representation for that domain. Without ontologies, there cannot be a vocabulary for representing knowledge. Ontologies enable knowledge sharing, since

they capture the intrinsic conceptual structure of the domain using a suitable knowledge representation language. This language can be shared with others that have similar needs for knowledge representation in that domain, thereby eliminating the need for replicating the knowledge-analysis process.

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.

A multi-agent system (MAS) enables an application developer to decompose a particularly complex task into simpler and easier-to-implement subtasks. Each agent in a MAS has a specific task (or set of tasks) according to its *role*. There are two organizations that should be considered in the context of multi-agent systems. The allocation of tasks to agents determines the *task-based* organization of MAS development. The way knowledge is embedded in agents is the essence of *knowledge-based* organization.

Explicit ontologies make the knowledge-based organization in a MAS clear. An ontology is designed for the purpose of enabling knowledge sharing and reuse. An 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 be profitably shared among different agents and/or reused for building multi-agent systems that operate 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. Furthermore, the most debated issue in the agent community is still a clear definition of what an agent and a multi-agent system are [4]. A suitable agent ontology could certainly help in the definition of a set of general and unambiguous features to characterize different classes of agents.

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

Multi-agent systems have a lot of potential to facilitate our interaction with the Internet. Information finding agents can cooperate to return knowledge to the user. In [11], the activities of the Intelligent Agent Lab at the University of Melbourne are described applying to a range of activities, including finding sports scores, a prototypical information finding task on the Internet. In this paper we extend finding sports scores to a MAS. A domain ontology is given for an application that finds the "best" sporting city in Italy[1]. Our prototype MAS compares city results from most popular Italian sports and, according to a heuristic analysis, the best city is returned. Moreover, we present an agent level ontology for agents that can be defined in CaseLP, a declarative framework for rapid prototyping of agent-based software applications. Domain and agent level ontologies are joined in CaseLP to allow analysis of MAS architectural description and single agent behaviour. Agents exploit domain level ontology during computation as knowledge repositories.

Our paper is structured as follows. The next section presents the ontology we have defined for the domain of sport results. Section 3 sketches the main characteristics of CaseLP and introduces the ontology we have defined for CaseLP agents. Section 4 gives some examples on how ontologies can be used in CaseLP. Finally, Section 5 briefly discusses benefits and drawbacks of our work and compares it to other proposed approaches to build ontologies.

2 A domain ontology for sport results

The domain of sports results is interesting as it highlights the contrast between the uniformity and diversity of information on the Web. Our prototype MAS copes with this diversity. It is capable of answering the question:

> "Which is the most successful sporting city in Italy?"

The answer is found from searching a number of sport information sources. The starting point for this application [12] is *SportsFinder* [7], an information agent to extract sports results from the Web. The sports considered include cycling, soccer, basketball and volleyball. The competitors considered include Italian cyclists in the first 50 positions of "UCI road rankings", "Serie A" and "Serie B" soccer teams, and "A1" and "A2" basketball and volleyball teams. Competitions for cycling are the 10 world cup races and the two main stage races, that is "Tour de France" and "Giro d'Italia". For soccer, we consider "Serie A" and "Serie B" championships. "A1" and "A2" championships are considered both for basketball and volleyball.

A major issue is how to measure the "success" of a particular city, in such a way it can be compared with other cities. For doing this, each Italian city taken into account is given a *score* that represents its success. The values for scores are influenced by various factors, including

- relative importance sports are given in Italy;
- relative importance competitions of a same sport are given;

[1] It should be straightforward to find the best sporting city in Australia or the U.S.

- how different competitions of the same importance contribute to the city score.

The score for a city is calculated using a heuristic function that considers the factors above. Partial scores are calculated by *aggregator* agents, one for each sport, that interact with a $result\_finder$ agent to collect results related to the sport. A $best\_finder$ agent receives partial scores from each aggregator and calculates the final score of every city. Cities whose score is maximum are returned to the user, as well as the maximum score.

Relevant objects for this application domain are sports, competitors, competitions and cities. They are incorporated in the domain ontology we have defined.

The natural way of modelling an ontology for an agent-based system based on logic programming is as a 5-tuple

$$< name, terms, predicates, kb, constraints >$$

where *name* identifies the ontology, *terms* is a set of constants and function symbols, *predicates* is a set of predicate symbols to be applied to terms, the knowledge base *kb* is a set of definite Horn clauses, and finally *constraints* is a set of constraints that *kb* has to satisfy.

For our running example, the name of the ontology is *sports*. The terms are a union of the constants denoting sports, competitors, competitions, competition classes, and competition types. There are no function symbols in *sports*. The predicate symbols are $sport/1$, $competitor/1$, $type/1$, $class/1$, $competition\_of/2$, $competition\_type/2$, $competition\_class/2$, $participant/2$, and $plays/2$.

Part of the knowledge base of the ontology *sports* is presented below. The knowledge base contains facts for every sport. They are

$$sport(cycling),$$
$$sport(soccer),$$
$$sport(basketball),$$
$$sport(volleyball).$$

Each sport is associated to a set of competitions that, in turn, are associated to a competition type and a competition class. Competition type distinguish competitions organized as a series of matches between pair of competitors (soccer, basketball, volleyball) from competitions whose result is a ladder containing all the competitors (cycling). Competition class refers to the importance of a competition. For example, "Serie A" is more important than "Serie B" and "Tour de France" is more important than whatever world cup race.

Part of the corresponding section of knowledge base is

$$competition\_of(serie\_a, soccer),$$
$$competition\_of(serie\_b, soccer),$$
$$competition\_type(serie\_a, match),$$
$$competition\_type(tour\_de\_france, ladder),$$
$$competition\_class(serie\_a, 1),$$
$$competition\_class(serie\_b, 2).$$

Arguments of *competition_of* are a competition and a sport, whereas arguments of *competition_type* and *competition_class* are respectively a competition and a type and a competition and a class.

Each competition has participants. We define a predicate *participant* that holds for a competitor and a competition. For example,

$$participant(marco\_pantani, giro\_italia),$$
$$participant(torino, serie\_a)$$

are part of the knowledge base. Participant of a competition are competitors, so predicate *competitor* is defined as

$$competititor(X) \leftarrow participant(X, Y) \wedge competition\_of(Y, Z).$$

Furthermore, each competitor plays a sport, then

$$plays(X, Y) \leftarrow participant(X, Z) \wedge competition\_of(Z, Y)$$

is included in the knowledge base.

We also need an ontology for cities consisting of the names of the cities and the names of the competitors as constants, and two predicate symbols $city/1$ and $is\_from/2$ that hold respectively on terms that represent cities and for pairs of competitors and cities. The knowledge base for the city ontology is expressed with facts as

$$city(torino),$$
$$city(bologna),$$
$$is\_from(torino, torino),$$
$$is\_from(marco\_pantani, bologna).$$

The ontology for the multi-agent system is the union of the sports and city ontologies. Note that the participant names are assumed to be the same. The semantics of our union operation is analogous to composing logic programs as described by Brogi [2] and applied in LogicWeb [6].

The set of constraints is the union of the constraints belonging to both the sport ontology and the city ontology. Furthermore, new constraints can be defined. For example,

$$plays(X, Y) \rightarrow competitor(X) \wedge sport(Y),$$
$$is\_from(X, Y) \rightarrow competitor(X) \wedge city(Y)$$

are two constraints. The first constraint states that relationships *plays* holds on a competitor and a sport, the second one that is_from holds between a competitor and a city. Constraints can be exploited to check knowledge base consistency whenever the ontology is updated.

3 Adding ontologies to CaseLP

CaseLP is a framework for rapid prototyping of agent-based software applications[2]. It is built upon the logic programming language ECLiPSe[1]. CaseLP provides both

[2]The state-of-the-art of CaseLP is described in detail in [8]. This paper presents some extensions that have still to be fully integrated in the current release.

an iterative method for specification, implementation, execution and testing of MAS-based prototypes, and a set of languages and tools to facilitate agent development. The method allows an application developer to build a prototype following a sequence of steps, refining the prototype against the client's desired requirements.

In CaseLP there are four kinds of agents. *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, that can be used to find agents that provide some services. *Manager agents* create and delete other agents in an application.

A CaseLP agent is characterized by a *kind*, an *architecture*, an eventual *interpreter*, a set of *provided services*, a set of *required services*, a set of *exported services* and a set of *imported services* [8]. The agent level ontology contains domain independent knowledge. Values for attributes defining *kind, architecture and interpreter* do not depend on the particular domain of application for which the prototype is being constructed. Instead, they depend specifically on the fact that agents are CaseLP agents. Values for *kind* define the type of an agent. This value constrains the possible further attributes that the agent has and the values that attributes can assume. For example, *interpreter* is only defined for interface agents, that also have necessarily a reactive architecture.

The name of the ontology we have defined is *agent*. First, we need terms for expressing the available kinds of agents and a predicate that says these terms are proper values for that agent kind. The natural choice is defining four constants

$$logical, interface, facilitator, manager$$

and a predicate

$$agent\_kind/1.$$

Then, the knowledge base of ontology *agent* will contain four facts of the form

$$\begin{array}{l} agent\_kind(logical), \\ agent\_kind(interface), \\ agent\_kind(facilitator), \\ agent\_kind(manager). \end{array}$$

Similarly, we need to express the available agent architectures. We have the terms

$$reactive, proactive, hybrid(reactive, proactive), prs$$

and the predicate

$$agent\_architecture/1.$$

The corresponding part of knowledge base is

$$\begin{array}{l} agent\_architecture(reactive), \\ agent\_architecture(proactive), \\ agent\_architecture(hybrid(reactive, proactive)), \\ agent\_architecture(prs). \end{array}$$

To understand the agent architectures in CaseLP we briefly discuss the rule language of CaseLP. More details can be found in [8]. ACLPL is a rule based declarative language that is used to define initial state and behaviour of single instances of agent. By reactive, we mean an agent whose behaviour is given by *event-condition-actions* ACLPL rules. By proactive, we mean instead agents behaving according to *condition-actions* ACLPL 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 set of beliefs that form the agent state or the sending of messages to other agents in the system. A hybrid architecture is obtained mixing proactive and reactive rules in the behaviour of an agent. Finally, *prs* refers to an agent whose architecture is a *procedural reasoning system* [5], an implementation of the BDI model for agents [10]. Each of the above terms refer to an architecture for which an engine has been realized. An engine is actually a meta-interpreter that implements the particular architecture *task-control*. New terms will be enclosed into the ontology as soon as other architectures are available in CaseLP.

An interface agent is given an interpreter to access an external legacy module. This part of the ontology is defined as

$$interpreter(c\_int),$$
$$interpreter(eclipse\_int),$$
$$interpreter(eclipsedb\_int).$$

Predicate $interpreter/1$ holds for terms that represent possible interpreters. Currently we have interpreters for C modules, ECLiPSe modules, ECLiPSe database modules.

In addition to the previously mentioned features, an agent provides a set of basic services. These services are implicitly provided by certain CaseLP agents, but have to be made explicit by the agents that require them. Agent basic services can include:

- creation or deletion of an agent;
- provision of information about agent kind, architecture, services;
- link of a service provided by an agent with a service required by another agent;
- search for an agent that provides some services.

The ontology *agent* includes vocabulary to identify basic services. Predicate

$$agent\_service/1$$

holds for all the terms that represent basic services provided by agents, and the

corresponding part of the knowledge base is defined as

$$agent\_service(createag),$$
$$agent\_service(destroyag),$$
$$agent\_service(agent\_type),$$
$$agent\_service(architecture\_type),$$
$$agent\_service(provserv),$$
$$agent\_service(reqserv),$$
$$agent\_service(expservices),$$
$$agent\_service(impservices),$$
$$agent\_service(link\_services),$$
$$agent\_service(who\_provides\_service).$$

Each service needs a specific *conversation* between the agent that provides the service and the one that requires it. A conversation is actually composed of messages that are exchanged between pairs of agents.

CaseLP allows the specification of conversations, as well as (sub)conversations that can be started in the middle of another conversation. The *agent communication language* is a subset of KQML[9]. Terms for possible *contents* of KQML messages are defined in the ontology *agent*, as well as predicates $agent\_msg/1$ and $associated\_to/2$, that respectively identify a term as a message content and associate message contents to services. Advantage is taken of variables in logic programming to allow incomplete messages where answers can be returned. For example, a term for expressing the message content related to service *agent_type* is

$$agent\_type(Type).$$

This term is the content of a KQML message that may be sent by an agent a to another agent b to ask b's kind. Variable $Type$ will be instantiated by b before replying to a. Content of the knowledge base related to $agent\_type(Type)$ is

$$agent\_msg(agent\_type(Type)),$$
$$associated\_to(agent\_type(Type), agent\_type).$$

To give another example, service *createag* is associated to the message content

$$createag(Agentname, Kind, Arch, Int, Provserv, Reqserv, Expserv, Impserv)$$

and service *destroyag* to the message content

$$destroyag(Agentname).$$

These messages commit the *manager* agent that receives them respectively to create a new agent *Agentname* with characteristics provided by other parameters, and to delete the agent named *Agentname*. Finally, service *who_provides_service* is associated to the message content

$$provide(Service, Agentname).$$

It is received by a *facilitator* agent, that instantiates *Agentname* before replying to the agent that submitted the request.

Not all kinds of agents provide the same set of services. Thus, relationships to relate services to particular kinds of agents are needed in the ontology. For example,

$$service\_of(createag, manager)$$

states that creation of a new agent is a service provided by manager agents. Similarly,

$$service\_of(who\_provides\_service, facilitator)$$

states that the service *who_provides_service* is given by facilitator agents.

The predicate $architecture\_of/2$ defines a relationship between kinds of agents and their allowed internal architectures. For example, it does not make sense that an interface agent can have a *prs* architecture. This because of the task that interface agents have to accomplish. They have just to translate requests from other agent in the MAS into queries to be submitted to the external module to which they are linked. A simple reactive architecture certainly suits this task more appropriately. On the other hand, logical agents can be built using all available internal architectures. Their task is coordinating and controlling activities of the MAS. According to the complexity of their task, the suitable architecture can be chosen. Similar considerations can be done for facilitator and manager agents. Facilitators reactively respond to questions about service providers so they can be thought as purely reactive agents. Managers can either react to creation request performing initialization of a new agent or, according to their internal state and plans, autonomously create new instances. Their architecture should be as flexible as possible.

The remarks above are captured in the ontology knowledge base as

$$\begin{aligned}
&architecture_of(logical, reactive),\\
&architecture_of(logical, proactive),\\
&architecture_of(logical, hybrid(reactive, proactive)),\\
&architecture_of(logical, prs),\\
&architecture_of(interface, reactive),\\
&architecture_of(facilitator, reactive),\\
&architecture_of(manager, reactive),\\
&architecture_of(manager, proactive),\\
&architecture_of(manager, hybrid(reactive, proactive)),\\
&architecture_of(manager, prs).
\end{aligned}$$

So far, we have presented the domain independent part of the ontology for CaseLP agents. Besides the agent level services already presented, agents have at their disposal domain level services that are related to the application domain of the MAS. In the following, we define domain level services and message contents for the domain of sports results, as well as constraints on the ontology. We introduce the predicate

$$domain\_service/1$$

to express that some terms represent services at the domain level. Predicate *domain_service* represents a "link" between the agent level and the domain on which an agent operates. Furthermore, we need a predicate

$$domain\_msg/1$$

to express that some terms are domain level message contents, as well as a predicate

$$domain\_associated\_to/2,$$

to associate domain level message contents to domain level services.

Part of the ontology knowledge base for domain services for the sports application is

$$domain\_service(last\_result),$$
$$domain\_service(last\_adder\_position),$$
$$domain\_service(city\_sport\_score),$$
$$domain\_service(best\_cities),$$
$$domain\_service(best\_city\_score).$$

Service *last_result* is provided by *result_finder* agents and is of use for *aggregator* agents to retrieve the last result obtained by a competitor in a competition. Service *last_adder_position* is provided as well by *result_finder* agents and is of use for *aggregator* agents to retrieve the last position of a competitor in a competition ladder. Aggregators combine these two results to calculate the competitor score in a competition. These scores are in turn combined to obtain the city score for a particular sport. The city score for a particular sport is transmitted from *aggregator* agents to the *best_finder* agent via service *city_sport_score*. Services *best_cities* and *best_city_score* are exported by the *best_finder* agent to provide answers to external users.

For each of the above (not exported) services, we include some terms for messages contents in the ontology *sports*. For example, the content message for *last_result* and its association to the service are expressed as

$$domain\_msg(last\_result(Sport, Competition, Competitor, Result)),$$
$$domain\_associated\_to(last\_result(Sport, Competition, Competitor, Result),$$
$$\quad last\_result)$$

Similarly, content message for *city_sport_score* and its association to the service are expressed as

$$domain\_msg(score(City, Sport, Score)),$$
$$domain\_associated\_to(score(City, Sport, Score), city\_sport\_score).$$

The last component of an ontology is a set of constraints on the knowledge base. For example, relationship *associated_to* holds only between an *agent_msg* and a *service*. This is captured by the constraint

$$associated\_to(X, Y) \rightarrow agent\_msg(X) \wedge service(Y).$$

Analogously, relationship *service_of* holds only between a *service* and a *kind*. The corresponding constraint is

$$service\_of(X, Y) \rightarrow service(X) \wedge kind(Y).$$

Similar constraints can be set to state other properties that the knowledge base has to satisfy.

As far as domain level is concerned, constraint

$$domain\_associated\_to(X, Y) \rightarrow \\ domain\_msg(X) \wedge domain\_service(Y).$$

states that relationships $domain\_associated\_to(X, Y)$ holds on domain message contents and domain services.

In the next section, we give some ideas about how ontologies we have defined can be used in CaseLP.

4 Exploiting ontologies in LP agents

The ontology *agent* can be exploited to perform a semantic check on an architectural description of a CaseLP MAS. In CaseLP, the domain-independent architectural description for an agent class is

```
agentclass <ClassName> {
    kind: <Kind>;
    architecture: <ArchType>;
    interpreter: <IntName>;
}
```

In order to realize a MAS prototype, various classes of agents are defined. From class descriptions, deriving a theory T_{ad} is straightforward. T_{ad} is a clausal representation of the architectural description of a MAS. For each class definition, it contains clauses of the form

$$agent\_class(ClassName) \leftarrow \\ agent\_kind(Kind) \wedge \\ agent\_architecture(ArchType) \wedge \\ architecture\_of(Kind, ArchType) \wedge \\ interpreter(IntName).$$

We can check semantic consistency of this theory proving that

$$agent_{kb} \models T_{ad}$$

where $agent_{kb}$ is the knowledge base of ontology *agent*. For example, if we define class $result\_finder$ in which `kind` assumes the value *logical*, this definition does not pass semantic check, because of logical agents have not an associated interpreter. The same happens if we define a $facilitator$ agent with a *proactive* architecture, or if the service *createag* is linked to a non-manager agent.

We can use the domain dependent part of the ontology *agent* to check agent behavioural rules. Every reactive *event-condition-action* rule that is defined in the behaviour, has to be related to a service provided by the agent. For example, the event part of the rule used by a $result\_finder$ agent to provide service *last_result* is the following:

$$on\ message \\ ask(content(last\_result(Sport, Competition, Competitor, Result)), \\ sender(Aggr))$$

Content of the message in the head of the rule has to be a content that is associated to the mentioned service.

As far as domain-dependent ontologies are concerned, predicates that are defined in the domain ontology can be exploited in agents' behaviour as *conditions* to be satisfied to proceed computation. For example *aggregator* agents use predicates *plays* and *participant* in ontology *sports* to retrieve, respectively, all the players of the sport under their responsibility, and the competition in which these players participate. The agent *best_finder* uses the predicate *is_from* to select all players that live in a city.

5 Discussion

In this paper we have sketched how ontologies can be defined in the setting of CaseLP, a framework based on logic programming. Logic programming has undoubtedly many good characteristics that make it a suitable paradigm for implementing ontologies. For example, LP languages naturally provide declarative representation and organization of symbolic knowledge. LP incorporates inference and reasoning capabilities that can be exploited either to derive information based on ontological knowledge, or to verify whether or not ontology constraints are satisfied. In CaseLP agents are finally implemented as logical modules. Ontologies are implemented as logical modules as well, in such a way as they can be easily included in agent code. We plan to realize an *ontology manager* for automatic creation and updating of ontologies.

This paper is the first step towards the integration of ontologies into the CaseLP framework. Tools for implementing the various kinds of analysis outlined in Section 4 are under study. In particular, semantic check of a MAS architectural description can be executed after it has been defined in step 1 of CaseLP prototyping method [8]. Check on behavioural rules of agents is performed using domain dependent part of the ontology *agent*. This check can be executed by the ACLPL compiler, that can properly exploit the agent level ontology. This ontology reflects the current definition of CaseLP agents. Even if we retain that this definition is accurate enough, it could be changed in the future to include new useful agent features. In this case, the ontology will be properly updated.

Our approach to the definition of ontology is certainly a bottom-up approach. For the definition of ontology *agent* we started from the particular model represented by CaseLP agents and to define ontology *sports* our start has been a specific domain of application. This is certainly a drawback, since we started from scratch to create an organization of concepts that could be borrowed from some *general* ontologies. Research on *general* ontologies (see [3] for a survey) has led to *task-independent* ontologies, for example CYC, Wordnet, Um-Thing and Sowa's. Currently an ontology cannot cover all possible potential uses, and choosing the most appropriate general ontology for a particular application domain is problematic at best. General ontologies are huge tree structures of concepts. If we cannot identify a suitable subtree for our application domain, we are obliged to include a very large set of concepts that may not be of interest.

The choice of keeping agent and domain ontologies separate has been driven by the necessity of facilitating ontology management and improving ontology comprehension and standardization. Features captured by the ontology *agent* enhance the comprehension of what a CaseLP agent is. Moreover, it can be a starting point for the definition of a more general ontology that represents and organizes characteristics of a broader class of agents and MAS. As far as domain ontologies are concerned, this separation keeps them as independent as possible from their use in a MAS. Knowledge in a domain ontology can be defined without regard to how it will be manipulated by CaseLP agents, which certainly enhances reuse and standardization. Agent and domain levels have to be linked, and part of the agent ontology must include terms that will be manipulated at domain level. Predicates *domain_service*, *domain_msg* and *domain_associated_to* have been included in the agent ontology for this purpose.

Acknowledgments

The first author would like to thank the Department of Computer Science & Software Engineering at the University of Melbourne for hosting his visit during which this research was performed.

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 Centre, Munich, 1995.

[2] A. Brogi. *Program Construction in Computational Logic.* PhD thesis, Università di Genova-Pisa-Udine, 1993.

[3] 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.

[4] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III.* Springer-Verlag, 1996. Lecture Notes in Artificial Intelligence 1193.

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

[6] S.W. Loke. *Adding Logic Programming Behaviour to the World Wide Web.* PhD thesis, Department of Computer Science & Software Engineering, The University of Melbourne, Parkville, 3052, Australia, 1998.

[7] H. Lu, L. Sterling, and A. Wyatt. SportsFinder: An Information Agent to Extract Sports Results from the World Wide Web. In *Proc. of PAAM'99*, London, UK, 1999.

[8] 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.

[9] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II*. Springer Verlag, 1995. Lecture Notes in Artificial Intelligence 1037.

[10] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In W. Swartout C. Rich and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[11] L. Sterling. On finding needles in WWW haystacks. In *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence*, Perth, Australia, 1997. Springer-Verlag. Lecture Notes in Artificial Intelligence 1342.

[12] L. Sterling and F. Zini. Finding the Best: a Multi-Agent System Approach. Technical report, Department of Computer Science & software Engineering, University of Melbourne, Parkville, 3052, Australia, 1999. Draft.