Design and development of a visual environment for writing DyLOG programs

Claudio Schifanella, Luca Lusso, Matteo Baldoni, Cristina Baroglio
Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
Tel. +39 011 6706711 — Fax. +39 011 751603
E-mail: {schi,baldoni,baroglio}@di.unito.it, lussoluca@tiscali.it

***Abstract*— In this article we present a visual development environment for writing DyLOG programs, explaining the motivations to this work and the main design choices. We will also analyze the main components of the system and the features offered to the user. The visual environment encompasses a fully new implementation of the DyLOG language, where Java is used instead of Sicstus Prolog, and an OWL ontology that will allow the use of the language in Semantic Web applications.**

I. Introduction

Engineering multi-agent systems (MASs) is a difficult task; one of the ways for achieving the successful industrial deployment of agent technology is to produce tools that support the developer in all the steps of design and implementation. Many researchers in the Agent Oriented Software Engineering (AOSE) community are developing complete environments for MAS design. Just to mention a few examples, AgentTool [1] is a Java-based graphical development environment to help users analyze, design, and implement MASs. It is designed to support the Multiagent Systems Engineering (MaSE) methodology [2], which can be used by the system designer to graphically define a high-level system behavior. Zeus [3] is an environment developed by British Telecommunications for specifying and implementing collaborative agents. DCaseLP (Distributed CaseLP, [4], [5], [6]) integrates a set of specification and implementation languages in order to model and prototype MASs. In this scenario, the quality of the tools that the designer can use strongly influences the *choice* of a given *specification language*. The availability of a visual environment that is intuitive to use, and simplifies the design of the agents in the system, can, actually, make the difference.

In this paper we present a visual environment (VisualDyLOG) for the development of DyLOG agents. DyLOG is a logic language for programming agents, based on reasoning about actions and change in a modal framework [7], that allows the inclusion, in an agent specification, also of a set of communication protocols. In ([8]) is proposed a methodological and physical integration of DyLOG into DCaseLP in order to reason about communication protocols.

By using VisualDyLOG, the user can specify in a simple and intuitive way all the components of a DyLOG program by means of a visual interface. The adoption of such an interaction device bears many advantages w.r.t. a text editor [9] and allows the programmer to work at a more abstract level, skipping the syntactical details of the language. Moreover, it is important to notice that the learning curve of logic languages is usually quite steep: the programming environment supplied by VisualDyLOG aims also at solving this problem.

An interesting application domain for agents develeoped by means of these tools is the Web, and in particular in the *Semantic Web*. Indeed, the web is more and more often considered as a means for accessing to interactive *web services*, i.e. devices that can be retrieved, invoked, composed in an automatic way. To this aim, there is a need for languages that allow web service specification in a well-defined way, capturing what the services do, how they do it, which information they need for functioning and so on, in order to facilitate the automatic integration of heterogeneous entities. Recently some attempt to standardize the description of web services has been carried on (DAML-S [10], OWL-S [11], WSDL [12]). While the WSDL initiative is mainly carried on by the commercial world, with the aim of standardizing registration, look-up mechanisms and interoperability, OWL-S (and previously, DAML-S) is more concerned with providing greater expressiveness to service descriptions in a way that can be *reasoned* about [13], by exploiting the *action metaphor*. In particular, we can view a service as an action (atomic or complex) with preconditions and effects, that modifies the state of the world and the state of agents that work in the world. Therefore, it is possible to design agents, which apply techniques for reasoning about actions and change to web service descriptions for producing new, composite, and customized services. These researches are basically inspired by the language Golog and its extensions [14], [15], [16]. In previous work, we have studied the use of DyLOG agents in the Semantic Web, and in particular, we have described the advantages that derive from an explicit representation of the *conversation policies* followed by web services in their description (currently not allowed by OWL-S). Actually, by reasoning on the conversation policies it is possible to achieve a better personalization of the service fruition [17], and it is also possible to compose services [18]. This research line has driven us to the implementation of an OWL [19] ontology, to be used as an interchange format of DyLOG programs, with the purpose of simplifying the use and the interoperation of DyLOG agents in a Semantic Web

M. Baldoni, F. De Paoli, A. Martelli, and A. Omicini (Eds.): *WOA 2004 – Dagli Oggetti agli Agenti, Sistemi Complessi e Agenti Razionali*, Torino, Italy, November 29th – December 1st 2004. Pitagora Editrice Bologna. ISBN 88-371-1533-4. http://woa04.unito.it

context.

The paper is organized as follows. Section II is a very brief introduction to the main characteristics of the DyLOG language. Section III describes the developed editing environment while Section IV describes the developed OWL ontology and motivates the choice of the OWL language. Conclusions follow.

II. THE DyLOG LANGUAGE

Logic-based executable agent specification languages have been deeply studied in the last years [20], [21], [15]. In this section we will very briefly recall the main features of DyLOG; the interested reader can find a thorough description of this language in [22], [23].

DyLOG is a high-level logic programming language for modeling rational agents, based on a modal theory of actions and mental attitudes where *modalities* are used for representing *actions*, while *beliefs* model the agent's internal state. It accounts both for *simple* (atomic) and *complex actions*, or procedures. Atomic actions are either world actions, affecting the world, or mental actions, i.e. sensing and communicative actions producing new beliefs and then affecting the agent mental state. Atomic actions are described in terms of *precondition laws* and *action laws* that, respectively, define those conditions that must hold in the agent mental state for the action to be applicable, and the changes to the agent mental state that are caused by the action execution. Notice that besides the preconditions to a simple action execution, some of its effects might depend upon further conditions (*conditional effects*). Complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like clauses and by action operators from dynamic logic, like sequence ";", test "?" and non-deterministic choice "$\cup$". The action theory allows coping with the problem of reasoning about complex actions with incomplete knowledge and in particular to address the temporal projection and planning problem in presence of sensing and communication.

Intuitively, DyLOG allows the specification of rational agents that reason about their own behavior, choose courses of actions conditioned by their mental state and can use sensors and communication for obtaining new information. The agent behavior is given by a *domain description*, which includes a specification of the agents initial beliefs, a description of the agent behavior plus a communication kit (denoted by CKit^{ag_i}), that encodes its communicative behavior. Communication is supported both at the level of *primitive speech acts* and at the level of *interaction protocols*. With regards to communication, a mentalistic approach, also adopted by the standard FIPA-ACL [24], is taken, where communicative actions affect the internal mental state of the agent. Some authors [25] have proposed a *social approach* to agent communication [25], where communicative actions affect the "social state" of the system, rather than the internal states of the agents. Different approaches are well-suited to different scenarios. DyLOG is a language for specifying an *individual, communicating agent*, situated in a multi-agent context. In this case it is natural to have access to the agent internal state.

We introduce an example that will be used in the rest of the paper, in order to better explain concepts. More details are included in the original work [26]. Let us consider the example of a robot which is inside a room. Two air conditioning units can blow fresh air in the room and the flow of the air from a unit can be controlled by a dial. In the following we report the code of the Simple Action *turn_dial(I)* that turns the dial of the unit *I* clockwise from a position to the next one. $\mathcal{B}_{robot}$ and $\mathcal{M}_{robot}$ are written as $\mathcal{B}$ (*belief*) and $\mathcal{M}$ (dual of $\mathcal{B}$) for simplicity.

A. A DyLOG implementation

At the basis of the development of the DyLOG programming environment there is a Java reimplementation of the language and of its interpreter. The choice of this implementation language is due to the great diffusion of it, to its well-known portability, and to the huge amount of available applications and frameworks.

The first step consisted in the development of the tuDyLOG package, which implements all the DyLOG constructs, offering to the programmer a set of classes and interfaces which allow the creation and editing of programs. tuDyLOG has been built upon *tuProlog* [27], a light-weight Prolog engine written in Java, which allows the instantiation and the execution of Prolog programs by means of a minimal interface. In this way, it is possible to exploit some of the mechanisms, made available by the *tuProlog* engine, which are used both in DyLOG and in Prolog, such as *unification*. Moreover, *tuProlog* supports interactions based on TCP/IP and RMI, a useful feature to the design of multi-agent systems. The implementation of the tuDyLOG package is currently being completed by extending the *tuProlog* engine so to obtain a tuDyLOG inference engine.

The structure of the classes which implement the language constructs follow the definition of a DyLOG program. A program is an instance of the class *DomainDescription*, which contains instances of the main kinds of program components: a set of initial observations, a set of actions that define the behavior of the agent, and a set of communicative actions. Each of such categories is represented by an adequate taxonomy, that reproduces the language specifications and offers a programming interface for operations like creation, modification, and deletion.

The connecting point between tuDyLOG and *tuProlog* is the class *DyLOGStruct*, an extension of the *tuProlog Struct* class: by means of *DyLOGStruct* every DyLOG construct can be turned into a corresponding *tuProlog* structure, with the possibility of exploiting the afore mentioned mechanisms. In this case we use a different notation (prefix notation) in order to meet the internal representation of the *tuProlog Struct* class. For example the first Precondition Law mentioned above is represented in this manner:

possible(turn_dial(I),if([belief(robot,in_front_of(I)),

$\Box(\mathcal{B}in\_front\_of(I) \wedge \mathcal{B}cover\_up(I) \supset \langle turn\_dial(I)\rangle\top)$
$\Box(\mathcal{B}flow(I, low) \supset [turn\_dial(I)]\mathcal{B}flow(I, high))$
$\Box(\mathcal{M}flow(I, low) \supset [turn\_dial(I)]\mathcal{M}flow(I, high))$
$\Box(\mathcal{B}flow(I, high) \supset [turn\_dial(I)]\mathcal{B}flow(I, off))$
$\Box(\mathcal{M}flow(I, high) \supset [turn\_dial(I)]\mathcal{M}flow(I, off))$
$\Box(\mathcal{B}flow(I, off) \supset [turn\_dial(I)]\mathcal{B}flow(I, low))$
$\Box(\mathcal{M}flow(I, off) \supset [turn\_dial(I)]\mathcal{M}flow(I, low))$
$\Box(\mathcal{B}flow(I, P) \supset [turn\_dial(I)]\mathcal{B}\neg flow(I, P))$
$\Box(\mathcal{M}flow(I, P) \supset [turn\_dial(I)]\mathcal{M}\neg flow(I, P))$

Fig. 1. The DyLOG code for the simple action *turn_dial(I)*.

belief(robot,cover_up(I))]))

III. VISUAL DYLOG

In this section we will show the main characteristics and features offered by VisualDyLOG. This environment is developed in Java using the *Eclipse* platform [28] and allows the development of a DyLOG program by means of a graphical user interface.

A. The Eclipse project

Eclipse is a platform designed for building integrated development environments (IDEs) and it represents a proven, reliable, scalable technology upon which applications can be quickly designed, developed, and deployed. More specifically, its purpose is to provide the services necessary for integrating software development tools, which are implemented as Eclipse plug-ins. The main characteristic of the design of Eclipse is, actually, that –except for a small runtime kernel– everything is a plug-in or a set of related plug-ins: the effect of this choice is an increase of the software reusability and extendability. Applications are deployed and distributed as stand-alone tools using the Rich Client Platform [29], which represents the smallest subset of Eclipse plug-ins that are necessary to build a generic platform application with a user interface.

Today Eclipse (originally released by IBM) is one of the most used platforms for developing applications, it has formed an independent and open eco-system, based on royalty-free technology, and it has established a universal platform for tools integration.

B. The environment

The VisualDyLOG environment is represented in Figure 2. We can distinguish different areas, each characterized by specific functionalities. With reference to the mentioned figure, area number (1) (the *Program View*) contains a whole view of the DyLOG program; it shows all the instances of the various constructs, kind by kind, and it also allows the creation and deletion of the instances. The area number (2) (*Editor View*) shows a visual representation of an instance contained in the Program View and selected by the user. The property values of such an instance are reported in the *Properties View* (area (4)). By working in the two latter views, the user can edit the selected instance. Since instances might in some cases be quite complex, there are situations in which the Editor View might show just a portion of the selected instance. Nevertheless a miniaturized overview of the whole instance will always be available in area number (3) (the *Outline View*). Last but not least, log messages are printed in the so called *Log View* (area number (5)).

VisualDyLOG internal architecture is based on the Graphical Editor Framework (GEF), an Eclipse plug-in. GEF, by exploiting the Model-View-Controller pattern, allows the creation of a graphical representation, given an existing data model. In our application, such a model is given by the instances of the package tuDyLOG, explained in Section II-A. In particular, by means of GEF:

- the graphical representation is modified after a change in the model has occurred;
- the model is changed by modifying the graphical representation of it, exploiting the "event-action" paradigm.

These notions are sketched by Figure 3. In Figure 4, instead, the *graphical notation* used to represent the main language constructs are shown.

It has been designed so to make the use of VisualDyLOG more intuitive: similar constructs are represented by shapes with the same morphology; such shapes recall flow chart symbols, contributing to a reduction of the learning process of new users.

C. An example of use

In this section we will show how to build a Simple Action by means of VisualDyLOG; in particular, we will use as an example the *turn_dial* action, whose DyLOG definition has been introduced in Section II (see Figure 1). The first step for creating a Simple Action consists in selecting the appropriate category from the Program View, and in assigning to it name and arity (the Program View also allows the creation of a new *action law* for a specific simple action). The new action will be added to the set of available Simple Actions in the Program View itself. By means of the Editor View, instead, it is possible to specify all the characteristics of the action: a working area is associated to each precondition law and to each action law that make the just created simple action; each such area can be selected and worked upon by clicking on a tab at the bottom of the Editor View. The palette at the right of the Editor View can be used to insert beliefs and terms in the working area. In order to edit a component it is necessary to click on the corresponding graphical representation of it and, then, modify

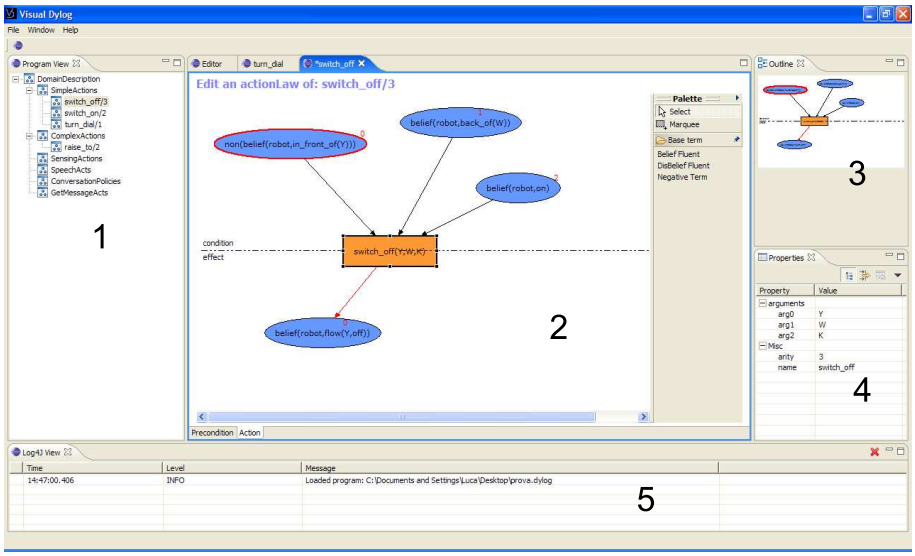

Fig. 2. A screenshot of the VisualDyLOG environment: (1) the Program View, (2) the Editor View, (3) the Outline View, (4) the Properties View, and (5) the Log View.

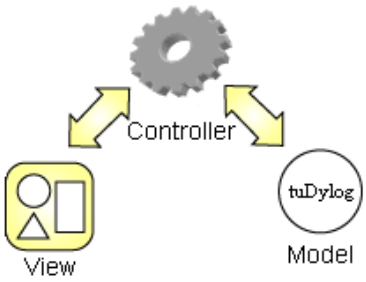

Fig. 3. GEF interaction model: the *Model*, in our case the tuDyLOG package, the *View* and the *Controller*, represented by GEF.

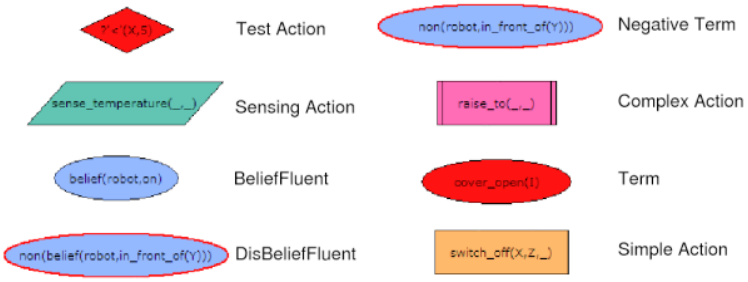

Fig. 4. The graphical notation used in VisualDyLOG

its properties by means of the Property View. In Figure 5 the working area used to create and modify the Precondition of the above mentioned action is shown. In the example *turn_dial* precondition consists of two fluents: the robot must be in front of the dial ($\mathcal{B}in\_fornt\_of(I)$) and the cover of the dial must be open ($\mathcal{B}cover\_up(I)$). In the figure, they are represented as light blue ovals. For the sake of simplicity, in the DyLOG representation of Figure 1 the agent name is omitted from the fluents. In the graphical representation, instead, it is the first argument of the fluents: since agents have a subjective view of the world, $robot$ is the agent to believe that $in\_front\_of(I)$ and $cover\_up(I)$ in order to execute the action. Notice also the prefix notation of fluents.

In Figure 6 the part of the interface devoted to the handling of one of the Action Laws is shown. The just described interaction schema is used also for the creation and editing of the other constructs of the language, such as complex actions (Figure 7), sensing actions, speech acts, and so forth.

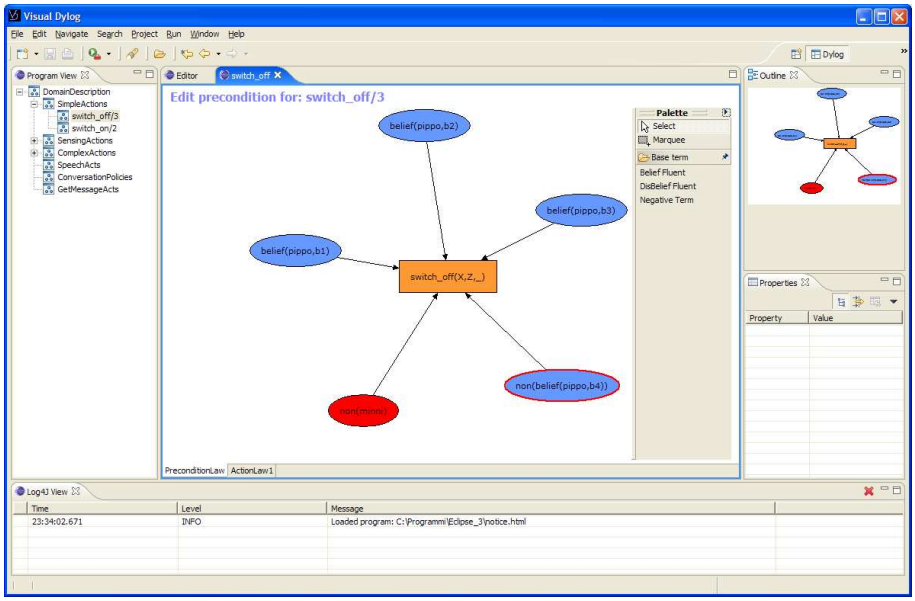

Fig. 5. Representation of a precondition law: beliefs are represented as light blue ovals, disbeliefs as blue ovals with a red border, terms as red ovals, while the action name is depicted as an orange rectangle.

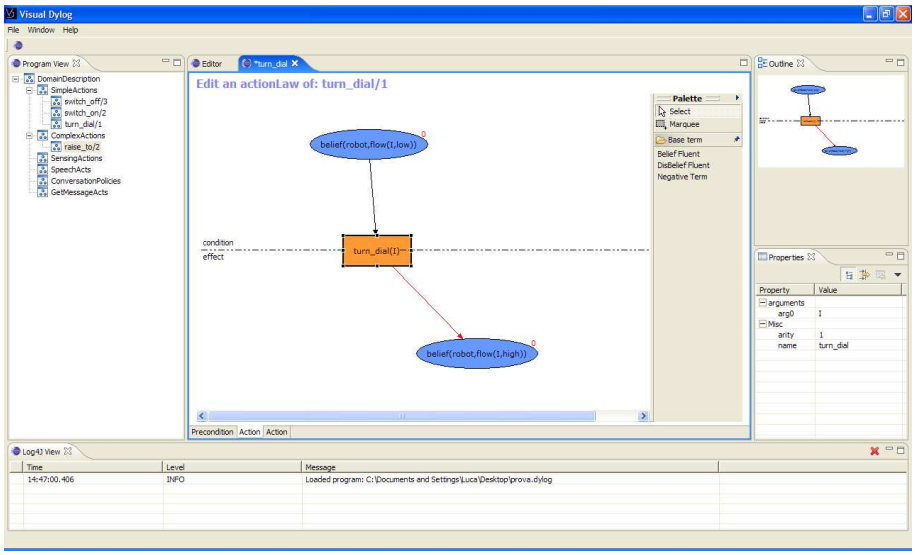

Fig. 6. Representation of an action law: the middle line divides preconditions to the effects from the action's effects themselves.

IV. An OWL ontology for DyLOG

In parallel with the work aimed at developing a programming environment for the language, we have also developed an ontology (called DyLOG Ontology) to be used for Semantic Web applications and, in particular, in the case of Semantic Web Services. We have already shown, in previous work, how the action metaphor and the mechanisms for reasoning about actions and change can fruitfully be exploited in many Semantic Web application frameworks [30], such as in educational applications and for the composition of web services. In order to allow the development of real applications over the web, there was a need of representing DyLOG programs in a way that is compatible with the infrastructure of the Semantic Web. Hence the choice of defining an OWL ontology.

OWL is a Web Ontology language [19], developed by the W3C community. The main characteristic of this language w.r.t. earlier languages, used to develop tools and ontologies for specific user communities is that it is compatible with the architecture of the World Wide Web, and with the Semantic Web in particular. In fact, OWL uses URIs (for naming) and the description framework provided by RDF, and adds the following capabilities to ontologies: the possibility of being distributed across many systems, full compatibility with Web standards for accessibility and internationalization, openess

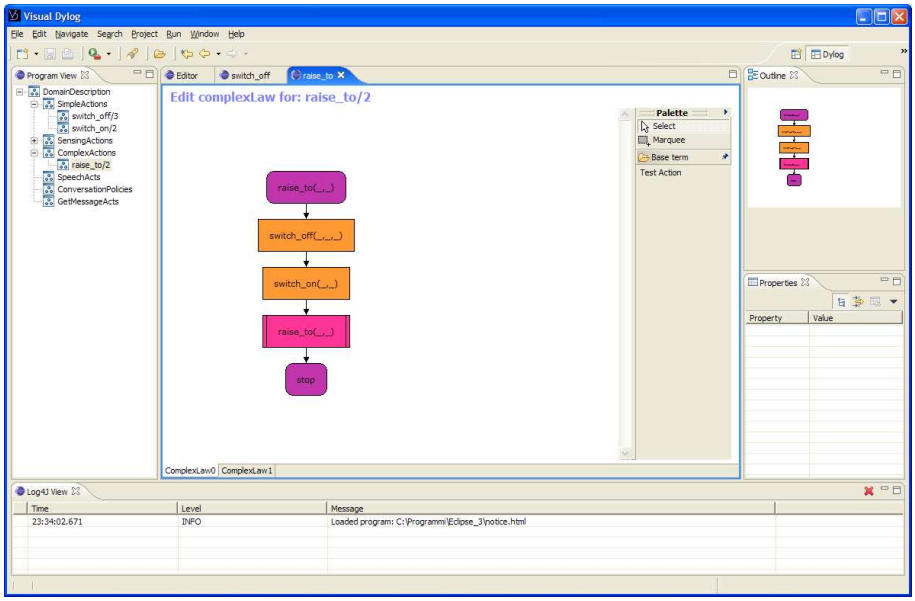

Fig. 7. Representation of a ComplexAction.

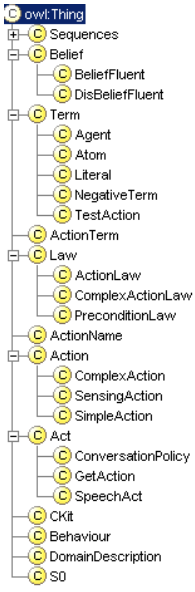

Fig. 8. The taxonomy of the DyLOG ontology

and extensiblility.

OWL builds on RDF and RDF Schema; it enriches the vocabulary so to allow the description of properties and classes. Some examples of add-ons are relations between classes (e.g. disjointness), cardinality (e.g. ”exactly one”), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

Recently, OWL has been used for defining a set of ontologies that can be considered as declarative languages and specifications for agents (more generally, web services) that are to be retrieved over the web in an intelligent way, so that they can interoperate and accomplish a common goal. A few examples are the OWL-S language [11], for web service functional description, FIPA OWL [31], an ontology for FIPA agents and ACL messages, and ConOnto [32], that allows the description of context aware systems. In the following we will describe the ontology that we have designed for describing DyLOG agents in a Semantic Web context.

A. *The* DyLOG *ontology*

As mentioned in the previous section, representing DyLOG programs by means of ontological terms allows the use of our language in the development of interoperating agents in a Semantic Web framework. Another advantage is the possibility to specify the syntactic constraints of the language directly within the ontology definition: for instance, a Simple Action must have one and only one Precondition Law; this constraint can be specified by imposing a proper restriction to the cardinality of the corresponding property. A reasoner can be used for verifying that the syntactic constraints are respected.

For representing a DyLOG program by means of the ontology it is necessary to start with an instance of class *DomainDescription*, which contains the properties for specifying the behavior, the communication policies and the initial observations (respectively *behaviour*, *ckit* and *s0*). Each such property is represented by an instance of a class that specifies all the characteristics of the corresponding DyLOG construct by means of properties and restrictions imposed to capture the syntactic constraints. In Figure 8 we report the taxonomy of the ontology, while in Figure 9 we present, as an example, the definition of the class *Simple Action* and its properties.

It is interesting to observe that, within a *Simple Action* instance, the order of the *Action Law* instances is meaningful

```
⟨owl:Class rdf:ID="SimpleAction"⟩
⟨rdfs:subClassOf rdf:resource="#Action"/⟩
⟨rdfs:subClassOf⟩ - ⟨owl:Restriction⟩
⟨owl:cardinality rdf:datatype="#int"⟩ 1 ⟨/owl:cardinality⟩
⟨owl:onProperty⟩
⟨owl:ObjectProperty rdf:about="#preconditionLaw"/⟩
⟨/owl:onProperty⟩ ⟨/owl:Restriction⟩ ⟨/rdfs:subClassOf⟩
⟨rdfs:subClassOf⟩ - ⟨owl:Restriction⟩ - ⟨owl:onProperty⟩
⟨owl:ObjectProperty rdf:about="#actionLawSeq"/⟩
⟨/owl:onProperty⟩
⟨owl:maxCardinality rdf:datatype="#int"⟩1⟨/owl:maxCardinality⟩
⟨/owl:Restriction⟩ ⟨/rdfs:subClassOf⟩ ⟨/owl:Class⟩

⟨owl:ObjectProperty rdf:ID="actionLawSeq"⟩
⟨rdfs:range rdf:resource="#ActionLawSeq"/⟩
⟨rdfs:domain⟩ - ⟨owl:Class⟩ - ⟨owl:unionOf rdf:parseType="Collection"⟩
⟨owl:Class rdf:about="#SpeechAct"/⟩
⟨owl:Class rdf:about="#SimpleAction"/⟩
⟨/owl:unionOf⟩ ⟨/owl:Class⟩ ⟨/rdfs:domain⟩ ⟨/owl:ObjectProperty⟩

⟨owl:ObjectProperty rdf:ID="preconditionLaw"⟩
⟨rdfs:range rdf:resource="#PreconditionLaw"/⟩
⟨rdfs:domain⟩ - ⟨owl:Class⟩ - ⟨owl:unionOf rdf:parseType="Collection"⟩
⟨owl:Class rdf:about="#SpeechAct"/⟩
⟨owl:Class rdf:about="#SimpleAction"/⟩
⟨/owl:unionOf⟩ ⟨/owl:Class⟩ ⟨/rdfs:domain⟩ ⟨/owl:ObjectProperty⟩

⟨owl:ObjectProperty rdf:ID="actionName"⟩
⟨rdfs:domain⟩ - ⟨owl:Class⟩ - ⟨owl:unionOf rdf:parseType="Collection"⟩
⟨owl:Class rdf:about="#Action"/⟩
⟨owl:Class rdf:about="#PreconditionLaw"/⟩
⟨/owl:unionOf⟩ ⟨/owl:Class⟩ ⟨/rdfs:domain⟩
⟨rdfs:range rdf:resource="#ActionName"/⟩ ⟨/owl:ObjectProperty⟩
```

Fig. 9. An excerpt from the OWL DyLOG ontology: definition of simple action.

because it might influence the program execution (like in prolog). Nevertheless, such an ordering cannot be represented directly in OWL. To this aim, we have defined an auxiliary structure (a linked list) that solves the problem. We have relied on this solution whenever an ordering had to be imposed over the instances of a given property.

In order to exploit the DyLOG ontology within the environment described in this article we added to tuDyLOG package functionalities to import and export a DyLOG program in Java representation to OWL and vice versa. This implementation uses libraries provided from Jena [33]: a framework, produced by HP labs, for develop Semantic Web applications.

V. Conclusions and future work

In this paper we have described VisualDyLOG, a graphical design and programming evironment for the modal logic language DyLOG. The project basically relies on two main choices. On a hand, a fully new, Java implementation of the DyLOG language has been developed, as an extension of the *tuProlog* package. The new package, named tuDyLOG actually exploits the basic mechanisms already offered by *tuProlog*, such as the methods for unification. The reason for changing implementation language (an implementation of DyLOG in Sicstus Prolog is already available) is that Java is more portable and allows us to exploit applications and frameworks that are already available, in particular, Eclipse: a well-known platform for building integrated development environments. By means of this platform it is easy to develop applications that can be deployed and distributed as stand-alone tools. The implementation of the graphical programming environment is almost complete; what still remains to do is the re-implementation of the DyLOG engine, which is on the way. Also the OWL ontology for DyLOG is ready to use and will be soon tested in a Semantic Web framework. In fact, we believe that this package will be very useful for the development of Semantic Web Services and we plan to use it in cooperation with the University of Hannover in an e-learning setting: integrating a DyLOG web service in the

Personal Reader architecture (see [34]).

REFERENCES

[1] AgentTool development system, "http://www.cis.ksu.edu/~sdeloach/ai/projects/agentTool/agentool.htm."

[2] S. A. DeLoach, *Methodologies and Software Engineering for Agent Systems*. Kluwer Academic Publisher, 2004, ch. The MaSE Methodology, to appear.

[3] ZEUS Home Page, "http://more.btexact.com/projects/agents.htm."

[4] E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio, "From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques," in *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, J. Debenham and K. Zhang, Eds. The Knowledge System Institute, 2003, pp. 578–585.

[5] I. Gungui and V. Mascardi, "Integrating tuProlog into DCaseLP to engineer heterogeneous agent systems," proceedings of CILC 2004. Available at http://www.disi.unige.it/person/MascardiV/Download/CILC04a.pdf.gz. To appear.

[6] M. Martelli and V. Mascardi, "From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents," in *Proceedings of the 8th APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'03)*, F. Buccafurri, Ed., 2003, pp. 275–286.

[7] M. Baldoni, L. Giordano, A. Martelli, and V. Patti, "Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach," in *Proc. of ICTCS'2001*, ser. LNCS, vol. 2202. Springer, 2001, pp. 405–425.

[8] M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella, "Reasoning about agents' interaction protocols inside dcaselp," in *Proc. of the International Workshop on Declarative Language and Technologies, DALT'04*, J. Leite, A. Omicini, P. Torroni, and P. Yolum, Eds., New York, July 2004, to appear.

[9] B. Shneiderman, *Designing the user interface*. Addison-Wesley, 1998.

[10] DAML-S, "http://www.daml.org/services/daml-s/0.9/," 2003, version 0.9.

[11] OWL-S, "http://www.daml.org/services/owl-s/."

[12] WSDL, "http://www.w3c.org/tr/2003/wd-wsdl12-20030303/," 2003, version 1.2.

[13] J. Bryson, D. Martin, S. McIlraith, and L. A. Stein, "Agent-based composite services in DAML-S: The behavior-oriented design of an intelligent semantic web," 2002. [Online]. Available: citeseer.nj.nec.com/bryson02agentbased.html

[14] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A Logic Programming Language for Dynamic Domains," *J. of Logic Programming*, vol. 31, pp. 59–83, 1997.

[15] G. D. Giacomo, Y. Lesperance, and H. Levesque, "Congolog, a concurrent programming language based on the situation calculus," *Artificial Intelligence*, vol. 121, pp. 109–169, 2000.

[16] S. McIlraith and T. Son, "Adapting Golog for Programmin the Semantic Web," in *5th Int. Symp. on Logical Formalization of Commonsense Reasoning*, 2001, pp. 195–202.

[17] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti, "Reasoning about interaction for personalizing web service fruition," in *Proc. of WOA 2003: Dagli oggetti agli agenti, sistemi intelligenti e computazione pervasiva*, G. Armano, F. De Paoli, A. Omicini, and E. Vargiu, Eds. Villasimius (CA), Italy: Pitagora Editrice Bologna, September 2003.

[18] ——, "Reasoning about interaction protocols for web service composition," in *Proc. of 1st Int. Workshop on Web Services and Formal Methods, WS-FM 2004*, M. Bravetti and G. Zavattaro, Eds. Elsevier Science Direct. To appear, 2004, electronic Notes in Theoretical Computer Science.

[19] OWL, "http://www.w3.org/2004/OWL/."

[20] K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V. Subrahmanian, "IMPACT: a platform for collaborating agents," *IEEE Intelligent Systems*, vol. 14, no. 2, pp. 64–72, 1999.

[21] M. Fisher, "A survey of concurrent METATEM - the language and its applications," in *Proc. of the 1st Int. Conf. on Temporal Logic (ICTL'94)*, ser. LNCS, D. M. Gabbay and H. Ohlbach, Eds., vol. 827. Springer-Verlag, 1994, pp. 480–505.

[22] M. Baldoni, L. Giordano, A. Martelli, and V. Patti, "Programming Rational Agents in a Modal Action Logic," *Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation*, 2004, to appear.

[23] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti, "Reasoning about self and others: communicating agents in a modal action logic," in *Theoretical Computer Science, 8th Italian Conference, ICTCS'2003*, ser. LNCS, C. Blundo and C. Laneve, Eds., vol. 2841. Bertinoro, Italy: Springer, October 2003, pp. 228–241.

[24] FIPA, "Fipa 97, specification part 2: Agent communication language," FIPA (Foundation for Intelligent Physical Agents), Tech. Rep., November 1997, available at: `http://www.fipa.org/`.

[25] M. P. Singh, "A social semantics for agent communication languages," in *Proc. of IJCAI-98 Workshop on Agent Communication Languages*. Berlin: Springer, 2000.

[26] M. Baldoni, L. Giordano, A. Martelli, and V. Patti, "Programming rational agents in a modal action logic," annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation. To appear.

[27] tuProlog Home Page, "http://lia.deis.unibo.it/research/tuprolog/."

[28] Eclipse platform, "http://www.eclipse.org."

[29] Eclipse Rich Client Platform, "http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/rcp/index.html."

[30] M. Baldoni, C. Baroglio, and V. Patti, "Web-based adaptive tutoring: An approach based on logic agents and reasoning about actions," *Journal of Artificial Intelligence Review*, 2004, to appear.

[31] FIPAOWL, "http://taga.umbc.edu/ontologies/fipaowl.owl."

[32] CONONTO, "http://www.site.uottawa.ca/~mkhedr/contexto.html."

[33] Jena Semantic Web Framework, "http://jena.sourceforge.net/."

[34] N. Henze and M. Herrlich, "The personal reader: a framework for enabling personalization services on the semantic web," in *Proc. of ABIS 2004*, 2004.