Reasoning About Agents' Interaction Protocols Inside DCaseLP

M. Baldoni[1], C. Baroglio[1], I. Gungui[2], A. Martelli[1],
M. Martelli[2], V. Mascardi[2], V. Patti[1], and C. Schifanella[1]

[1] Dipartimento di Informatica,
Università degli Studi di Torino, Italy
{baldoni, baroglio, mrt, patti, schi}@di.unito.it
[2] Dipartimento di Informatica e Scienze dell'Informazione,
Università degli Studi di Genova, Italy
1995s133@educ.disi.unige.it, {martelli, mascardi}@disi.unige.it

Abstract. Engineering systems of heterogeneous agents is a difficult task; one of the ways for achieving the successful industrial deployment of agent technology is the development of engineering tools that support the developer in all the steps of design and implementation. In this work we focus on the problem of supporting the design of agent interaction protocols by carrying out a methodological integration of the MAS prototyping environment DCaseLP with the agent programming language DyLOG for reasoning about action and change.

1 Introduction

Multiagent Systems (MASs) involve heterogeneous components which have different ways of representing their knowledge of the world, themselves, and other agents, and also adopt different mechanisms for reasoning. Despite heterogeneity, agents need to interact and exchange information in order to cooperate or compete not only for the control of shared resources but also to achieve their aims; this interaction may follow sophisticated communication protocols.

For these reasons and due to the complexity of agents' behavior, MASs are difficult to be correctly and efficiently engineered; even developing a working prototype may require a long time and a lot of effort. In this paper we present an ongoing research aimed at developing a "multi-language" environment for engineering systems of heterogeneous agents. This environment will allow the prototype developer to specify, verify and implement different aspects of the MAS and different agents inside the MAS, choosing the most appropriate language from a given set. In particular, the discussion will be focused on the advantages of integrating an agent programming language for reasoning about actions and change (using the language DyLOG [9, 7]) into the DCaseLP [4, 21, 29] MAS prototyping environment.

The development of a prototype system of heterogeneous agents can be carried out in different ways. The "one-size-fits-all" solution consists of developing

J. Leite et al. (Eds.): DALT 2004, LNAI 3476, pp. 112–131, 2005.

all the agents by means of the same implementation language and to execute the obtained program. If this approach is adopted, during the specification stage it would be natural to select a language that can be directly executed or easily translated into code, and to use it to specify all the agents in the MAS. The other solution is to specify each "view" of the MAS (that includes its architecture, the interaction protocols among agents, the internal architecture and functioning of each agent), with the most suitable language in order to deal with the MAS's peculiar features, and then to verify and execute the obtained specifications inside an integrated environment. Such a multi-language environment should offer the means not only to select the proper specification language for each view of the MAS but also to check the specifications exploiting formal validation and verification methods and to produce an implementation of the prototype in a semi-automatic way. The prototype implementation should be composed of heterogeneous pieces of code created by semi-automatic translations of heterogeneous specifications. Moreover, the multi-language environment should allow these pieces of code to be seamlessly integrated and capable of interacting.

The greater complexity associated with the latter solution is proportional to the advantages it gives with respect to the former. In particular, by allowing different specification languages for modeling different aspects of the MAS, *it provides the flexibility needed to describe the MAS from different points of view*. Moreover, by allowing different specification languages for the internal architecture and functioning of each agent, *it respects the differences existing among agents*, namely the way they reason and the way they represent their knowledge, other agents, and the world. Clearly, this solution also has some drawbacks in respect to the former. The coherent integration of different languages into the same environment must be carefully designed and implemented by the environment creators, who must also take care of the environment maintenance. It must be emphasized that the developer of the MAS does not have to be an expert of *all* the supported languages: he/she will use those he/she is more familiar with, and this will lead to more reliable specifications and implementations.

DCaseLP (Distributed CaseLP, [4, 21, 29]) integrates a set of specification and implementation languages in order to model and prototype MASs. It defines a methodology which covers the engineering stages, from the requirements analysis to the prototype execution, and relies on the use of UML and AUML (Agent UML, [6]) not only during the requirements analysis, but also to describe the *interaction protocols* followed by the agents. The choice of UML and AUML, initially developed for documentation purposes, to represent interaction protocols in DCaseLP is motivated by the wide support that it is obtaining from the agent research community. Even if AUML cannot be considered a standard agent modeling language yet, it has many chances to become such, as shown by the interest that both the FIPA modeling technical committee (`http://www.fipa.org/activities/modeling.html`) and the OMG Agent Platform Special Interest Group (`http://www.objs.com/agent/`) demonstrate in it. Quoting [31]: "The successful industrial deployment of agent technology requires techniques that reduce the inherent risks in any new technology and there are two ways in which this can be done: presenting a new technology as an extension of a previous, well-established one, or providing engineering tools that support industry-accepted methods of technology deployment." We can say that by choosing a UML-based language we place DCaseLP in the line of both the proposed strategies.

In DCaseLP, UML and AUML are used to describe the *public interaction protocols*, which can be animated by creating agents whose behavior adheres to the given protocols. The idea of translating UML and AUML diagrams into a formalism and check their properties by either animating or formally verifying the resulting code is shared by many researchers working in the agent-oriented software engineering field [24, 30, 35]. We followed an animation approach to check that the interaction protocols produced during the requirement specification stage are the ones necessary to describe the system requirements and, moreover, that they are correct. The "coherence check" is done by comparing the results of the execution runs with the interaction specification [4]. Despite its usefulness, this approach does not straightforwardly allow the formal proof of properties of the resulting system *a priori*: indeed, a key issue in the design and engineering of interaction protocols, that DCaseLP does not currently address. One possible extension in the line of [25] is the integration of *formal methods* to perform validation tests, i.e., to check the coherence of the AUML description with the specifications derived from the analysis. To this aim, it is possible to rely on works that give to AUML sequence diagrams a semantics based on Petri Nets [22, 23, 12]. Validation tests, however, are just one side of the problem. In fact, another kind of a priori verification that is very important for the MAS designer is to check properties of *specific* implementations, obtained on the basis of the public protocol description.

One step in this direction is to exploit the characteristic of DCaseLP of being a multi-language development environment and to integrate a language, DyLOG [9, 7], which, being based on computational logic, can be exploited both as an implementation language and for verifying properties. DyLOG is a logic-based agent language that includes a fully integrated "communication kit", that allows the implementation of *interaction protocols* as *agent conversation policies* based on speech acts, and it supports reasoning about interaction properties. In the language reasoning about the conversations, defined by a protocol implementation, basically means to check if there is a conversation after whose execution a given set of properties holds. This characteristic can for instance be exploited to determine which protocol, from a set of available ones, satisfies a goal of interest, and also to compose many protocols for accomplishing complex tasks. In this perspective, DyLOG is particularly interesting because there is a *conformance relation* between DyLOG implementations of interaction protocols and AUML sequence diagrams: in fact it is possible to prove in a formal way if every conversation generated by a DyLOG program is correct w.r.t. a specification expressed by AUML diagrams [8]. After proving desired properties of the interaction protocols, the developer can animate them thanks to the facilities offered by DCaseLP, discussed in Section 2.

So far, the integration of DyLOG into DCaseLP is a *methodological integration*: it extends the set of languages supported by DCaseLP during the MAS engineering process and augments the verification capabilities of DCaseLP, without requiring any real integration of the DyLOG working interpreter into DCaseLP (see Section 4). Nevertheless, DyLOG can also be used to directly specify agents and execute them inside the DCaseLP environment, in order to exploit the distribution, concurrency, monitoring and debugging facilities that DCaseLP offers.

2 The DCaseLP Environment

DCaseLP is a prototyping environment where agents specified and implemented in a given set of languages can be seamlessly integrated. It provides an agent-oriented software engineering methodology to guide the developer during the analysis of the MAS requirements, its design, and the development of a working MAS prototype. The methodology is sketched in Figure 1. Solid arrows represent the information flow from one stage to the next one. Dotted arrows represent the iterative refinement of previous choices. The first release of DCaseLP did not realize all the stages of the methodology. In particular, as we have pointed in last section, the stage of properties verification was not addressed. The integration of DyLOG into DCaseLP discussed in Section 4 will allow us to address also the verification phase. The tools and languages supported by the first release of

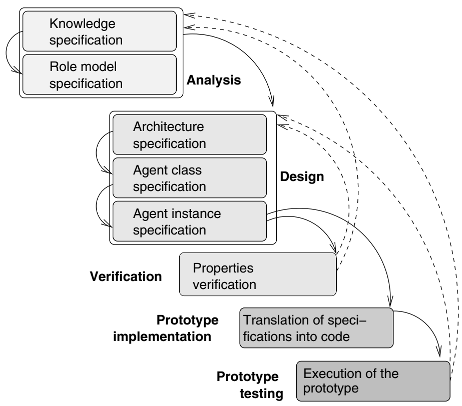

Fig. 1. DCaseLP's methodology

DCaseLP, discussed in [29, 4], included UML and AUML for the specification of the general structure of the MAS, and Jess [27] and Java for the implementation of the agents.

DCaseLP adopts an existing multi-view, use-case driven and UML-based method [5] in the phase of requirements analysis. Once the requirements of the application have been clearly identified, the developer can use UML and/or AUML to describe the interaction protocols followed by the agents, the general MAS architecture and the agent types and instances. Moreover, the developer can automatically translate the UML/AUML diagrams, describing the agents in the MAS, into Jess rule-based code. In the following we will assume that AUML is used during the requirements analysis stage, although the translation from AUML into Jess is not fully automated (while the translation from pure UML into Jess is).

The Jess code obtained from the translation of AUML diagrams must be manually completed by the developer with the behavioral knowledge which was not explicitly provided at the specification level. The developer does not need to have a deep insight into rule-based languages in order to complete the Jess code, since he/she is guided by comments included in the automatically generated code. The agents obtained by means of the manual completion of the Jess code are integrated into the JADE (Java Agent Development Framework, [26]) middle-ware. JADE complies with the FIPA specifications [16] and provides a set of graphical tools that support the execution, debugging and deployment phases. The agents can be distributed across several machines and can run concurrently. By integrating Jess into JADE, we were able to easily monitor and debug the execution of Jess agents thanks to the monitoring facilities that JADE provides.

A recent extension of DCaseLP, discussed in [21], has been the integration of tuProlog [36]. The choice of tuProlog was due to two of its features:

1. it is implemented in Java, which makes its integration into JADE easier, and
2. it is very light, which ensures a certain level of efficiency to the prototype.

By extending DCaseLP with tuProlog we have obtained the possibility to execute agents, whose behavior is completely described by a Prolog-like theory, in the JADE platform. For this purpose, we have developed a library of predicates that allow agents specified in tuProlog to access the communication primitives provided by JADE: asynchronous send, asynchronous receive, and blocking receive (with and without timeout). These predicates are mapped onto the corresponding JADE primitives. Two predicates for converting strings into terms and vice-versa are also provided, in order to allow agents to send strings as the content of their messages, and to reason over them as if they were Prolog terms.

A developer who wants to define tuProlog agents and integrate them into JADE can do it without even knowing the details of JADE's functioning. An agent whose behavior is written in tuProlog is, in fact, loaded in JADE as an ordinary agent written in Java. The developer just needs to know how to start JADE.

3 Interaction Protocols in DyLOG

Logic-based executable agent specification languages have been deeply investigated in the last years [3, 17, 13, 9, 28]. In this section we will briefly recall the main features of DyLOG, by focussing on how the communicative behavior of an agent can be specified and on the form of reasoning supported.

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. We refer to a mentalistic approach, which is also adopted by the standard FIPA-ACL [16], where communicative actions affect the internal mental state of the agent. More recently, some authors have proposed a *social approach* to agent communication [34], where communicative actions affect the "social state" of the system, rather than the internal states of the agents. The social state records the social facts, like the *permissions* and the *commitments* of the agents, which are created and modified along the interaction. The dissatisfaction to the mentalistic approach is mostly due to the difficulty of verifying that an agent acts according to a commonly agreed semantics, because it is not possible to have access to the agents' private mental state [37], a problem known as *semantics verification*. The growing interest into the social approach is motivated by the fact that it overcomes this problem by exploiting a set of established commitments between the agents, that are stored as part of the MAS social state. In this framework it is possible to formally prove the correctness of public interaction protocols with respect to the specifications outcoming from the analysis phases; such proof can be obtained, for instance, by means of model checking techniques [32, 37, 19, 10] (but not only, e.g., [11]).

When one passes from the public protocol specification to its *implementation* in some language (e.g. Java, DyLOG), a program is obtained which, by definition, relies on the information contained in the internal "state" of the agent for deciding which action to execute [20]. In this perspective, the use of a declarative language is helpful because it allows the proof of properties of the *specific implementation* in a straightforward way. In particular, the use of a language that explicitly represents and uses the agent internal state is useful for proving to which extent certain properties depend on the agent mental state or on the semantics of the speech acts. For instance, in our work we perform hypothetical reasoning about the effects of conversations on the agent mental state, in order to find conversation plans which are proved to respect the implemented protocols, achieving at the same time some desired goal, and we can prove the conformance of an implemented protocol w.r.t. its specification in AUML.

3.1 DyLOG in Brief

Intuitively, DyLOG [9, 7] 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 fresh knowledge. The agent behavior is described by a *domain description*, which includes, besides a specification of the agent initial beliefs, a description of the agent behavior plus a *communication kit* (denoted by CKit^{ag_i}), that encodes its *communicative behavior*. 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. 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.

Communication is supported both at the level of *primitive speech acts* and at the level of *interaction protocols*. Thus, the communication kit of an agent ag_i is defined as a triple $(\Pi_{\mathcal{C}}, \Pi_{\mathcal{CP}}, \Pi_{\mathcal{S}get})$: $\Pi_{\mathcal{C}}$ is a set of laws defining precondition and effects of the agent speech acts; $\Pi_{\mathcal{CP}}$ is a set of procedure axioms, specifying a set of interaction protocols, and can be intended as a library of *conversation policies*, that the agent follows when interacting with others; $\Pi_{\mathcal{S}get}$ is a set of sensing axioms for acquiring information by messages reception.

Speech acts are represented as atomic actions with preconditions and effect on ag_i's mental state, of form $\mathsf{speech\_act}(ag_i, ag_j, l)$, where ag_i (sender) and ag_j (receiver) are agents and l (a fluent) is the object of the communication. Effects and preconditions are modeled by a set of effect and precondition laws. We use the modality $\Box$ to denote such laws, i.e., formulas that hold *always*, after every (possibly empty) arbitrary action sequence.

A DyLOG agent has a twofold representation of each a speech act: one holds when it is the sender, the other when it is the receiver. As an example, let us define the semantics of the *inform* speech act within the DyLOG framework:

a) $\Box(\mathcal{B}^{Self} l \wedge \mathcal{B}^{Self} \mathcal{U}^{Other} l \supset \langle \mathsf{inform}(Self, Other, l) \rangle \top)$
b) $\Box([\mathsf{inform}(Self, Other, l)] \mathcal{M}^{Self} \mathcal{B}^{Other} l)$
c) $\Box(\mathcal{B}^{Self} \mathcal{B}^{Other} authority(Self, l) \supset [\mathsf{inform}(Self, Other, l)] \mathcal{B}^{Self} \mathcal{B}^{Other} l)$
d) $\Box(\top \supset \langle \mathsf{inform}(Other, Self, l) \rangle \top)$
e) $\Box([\mathsf{inform}(Other, Self, l)] \mathcal{B}^{Self} \mathcal{B}^{Other} l)$
f) $\Box(\mathcal{B}^{Self} authority(Other, l) \supset [\mathsf{inform}(Other, Self, l)] \mathcal{B}^{Self} l)$

In general, for each action a and agent ag_i, $[a^{ag_i}]$ is a universal modality ($\langle a^{ag_i} \rangle$ is its dual). $[a^{ag_i}]\alpha$ means that α holds after every execution of action a by agent ag_i, while $\langle a^{ag_i} \rangle \alpha$ means that there is a possible execution of a (by ag_i) after which α holds. Therefore clause (a) states *executability preconditions* for the action $\mathsf{inform}(Self, Other, l)$: it specifies the mental conditions that make the action executable in a state. Intuitively, it states that $Self$ can execute an inform act only if it believes l (we use the modal operator $\mathcal{B}^{ag_i}$ to model the beliefs of agent ag_i) and it believes that the receiver ($Other$) does not know l. It also considers possible that the receiver will adopt its belief (the modal operator $\mathcal{M}^{ag_i}$ is defined as the dual of $\mathcal{B}^{ag_i}$, intuitively $\mathcal{M}^{ag_i}\varphi$ means the ag_i considers φ possible), clause (b), although it cannot be certain about it -autonomy assumption-. If agent $Self$ believes to be considered a trusted *authority* about l by the receiver, it is also confident that $Other$ will adopt its belief, clause (c). Since executability preconditions can be tested only on the $Self$ mental state, when $Self$ is the receiver, the action of informing is considered to be *always* executable (d). When $Self$ is the receiver, the effect of an inform act is that

$Self$ will believe that l is believed by the sender (*Other*), clause (e), but $Self$ will adopt l as an own belief only if it thinks that *Other* is a trusted authority, clause (f).

DyLOG supports also the representation of *interaction protocols* by means of procedures, that build on individual speech acts and specify communication patterns guiding the agent communicative behavior during a protocol-oriented dialogue. Formally, protocols are expressed by means of a collection of procedure axioms of the action logic of the form $\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi$, where p_0 is the procedure name the p_i's can be i's speech acts, special sensing actions for modeling message reception, test actions (actions of the form $Fs?$, where Fs is conjunction of belief formulas) or procedure names [1]. Each agent has a subjective perception of the communication with other agents; for this reason, given a protocol specification, we have as many procedural representations as the possible roles in the conversation (see example in the next section).

Message reception is modeled as a special kind of sensing action, what we call *get message actions*. Indeed, from the point of view of an individual agent receiving a message can be interpreted as a query for an external input, whose outcome cannot be predicted before the actual execution, thus it seems natural to model it as a special case of sensing. The *get message actions* are defined by means of inclusion axioms, that specify a finite set of (alternative) speech acts expected by the interlocutor.

DyLOG allows reasoning about agents' communicative behavior, by supporting techniques for proving existential properties of the kind "given a protocol and a set of desiderata, is there a specific conversation, respecting the protocol, that also satisfies the desired conditions?". Formally, given a DyLOG domain description Π_{ag_i} containing a CKit^{ag_i} with the specifications of the interaction protocols and of the relevant speech acts, a *planning* activity can be triggered by *existential queries* of the form $\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_m \rangle Fs$, where each p_k $(k = 1, \ldots, m)$ may be a primitive speech act or an interaction protocol, executed by our agent, or a get message action (in which our agent plays the role of the receiver). Checking if the query succeeds corresponds to answering to the question "is there an execution of $p_1, \ldots, p_m$ leading to a state where the conjunction of belief formulas Fs holds for agent ag_i?". Such an execution is a plan to bring about Fs. The procedure definition constrains the search space.

Actions in the plan can be speech acts performed or received by ag_i, the latter can be read as the *assumption* that certain messages will be received from the interlocutor. The ability of making assumptions about which message (among those foreseen by the protocol) will be received is necessary in order to actually build the plan. Depending on the task that one has to execute, it may alternatively be necessary to take into account all of the possible alternatives that lead to the goal or just to find one of them. In the former case, the extracted plan will be *conditional*, because for each get_message it will generally contain

[1] For sake of brevity, sometimes we will write these axioms as $\langle p_0 \rangle \varphi \subset \langle p_1; p_2; \ldots; p_n \rangle \varphi$.

many branches. Each path in the resulting tree is a linear plan that brings about *Fs*. In the latter case, instead, the plan is linear.

4 Integrating DyLOG into DCaseLP to Reason About Communicating Agents

Let us now illustrate, by means of examples, the advantages of adding to the current interaction design tools of DCaseLP the possibility of converting AUML sequence diagrams into a DyLOG program. In the first DCaseLP release, AUML interaction protocols could be only translated into Jess code, which could not be formally verified but just executed. The use of DyLOG bears some advantages: on the one hand it is possible to automatically verify that a DyLOG implementation is *conformant* to the AUML specification (see below), moreover, it is also possible to *verify properties* of the so obtained DyLOG program. Property proof can be carried out using the existing DyLOG interpreter, implemented in Sicstus [1].

Besides the methodological integration, DyLOG can be also integrated in a *physical way*. Recently we have begun a new implementation in Java of the language, based on *tuProlog* [36]. A visual editor based on *Eclipse* is also being implemented; the editor will allow the designer to write DyLOG programs in a graphical and intuitive way, the designer will also have the possibility of exporting them in OWL [33] for realizing Semantic Web applications like the one described hereafter. Once the physical integration will be completed, it will be possible to animate complete DyLOG agents into DCaseLP. This will mean that agents specified in Jess, Java, DyLOG, will be able to interact with each other inside a single prototype whose execution will be monitored using JADE.

In the rest of this section, however, we deal with the *methodological integration*. Let us suppose, for instance, to be developing a set of interaction protocols for a restaurant and a cinema that, for promotional reasons, will cooperate in this way: a customer that makes a reservation at the restaurant will get a free ticket for a movie shown by the cinema. By restaurant and cinema we here mean two generic service providers and not a specific restaurant and a specific cinema. In this scenario the same customer will interact with both providers. The developer must be sure that the customer, by interacting with the composition (by sequentialization) of the two protocols, will obtain what desired. Figure 2 shows an example of AUML protocols, for the two services; (i) and (ii) are followed by the cinema, (iii) by the restaurant. This level of representation does not allow any proof of properties because is lacking of a formal semantics. Supposing that the designed diagrams are correct, the protocols are to be implemented. It is desirable that the correctness of the implementation w.r.t. the AUML specification can be verified. If the protocols are implemented in DyLOG, this can actually be done. In [8] we, actually, show that, given an AUML protocol specification and a DyLOG implementation, it is possible to prove if the latter will never produce conversations that are not foreseen by the protocol. This problem is known as *conformance verification*. Briefly, with reference to Figure 3, this can be done by turning the problem into a problem of verification of the inclusion of

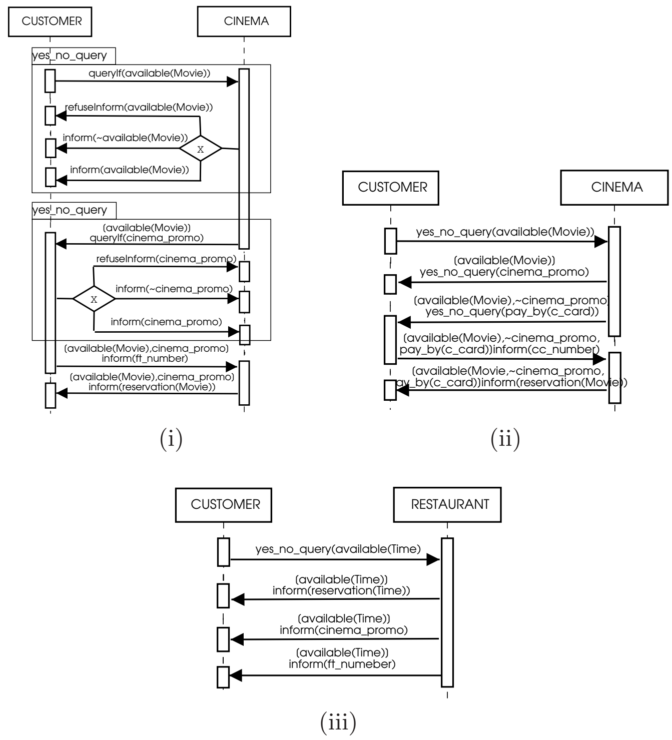

Fig. 2. AUML sequence diagrams representing the interactions between customer and provider: (i) and (ii) are followed by the cinema service, (iii) is followed by the restaurant. Formulas in square brackets represent preconditions to speech act execution

the language of all the sequences generated by the implementation $L(G_{p_{DyLOG}})$ in the language of all the sequence generated by the AUML sequence diagram $L(G_{p_{AUML}})$. In particular, we have studied the dependence of conformance on the agent private mental state and on the semantics of speech acts, proposing three degrees of conformance, at different levels of abstraction. The strongest of the three, *protocol conformance*, is proved to be decidable and tractable, and if it holds also the other degrees (which depend at some extent on the the agent mental state) hold.

Let us describe one possible implementation of the two protocols in a DyLOG program. Each implemented protocol will have two complementary views (customer and provider) but for the sake of brevity, we report only the view of the customer. It is easy to see how the structure of the procedure clauses corresponds to the sequence of AUML operators in the sequence diagrams. The subscripts next to the protocol names are a writing convention for representing the role that the agent plays; so, for instance, Q stands for *querier*, and C for *customer*. The customer view of the restaurant protocol is the following:

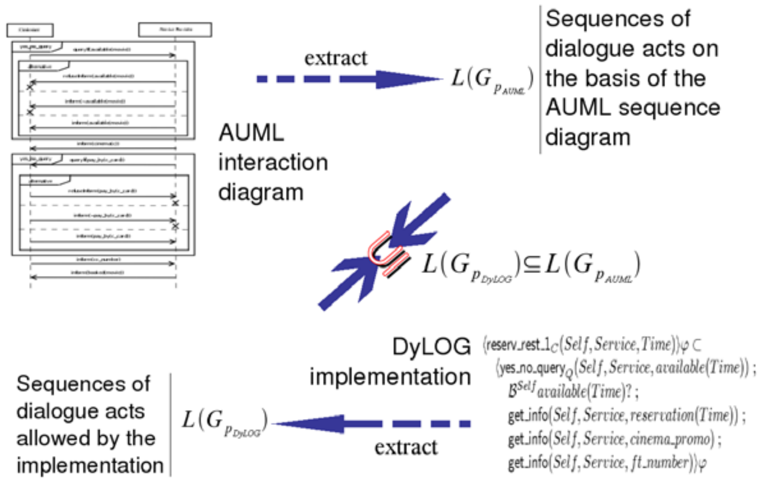

Fig. 3. Conformance verification of a DyLOG implementation w.r.t. an AUML protocol: the problem is interpreted as the verification of language inclusion

(a) $\langle \textsf{reserv\_rest}_C(Self, Service, Time)\rangle\varphi \subset$
$\langle \textsf{yes\_no\_query}_Q(Self, Service, available(Time))$;
$\mathcal{B}^{Self} available(Time)?$;
$\textsf{get\_info}(Self, Service, reservation(Time))$;
$\textsf{get\_info}(Self, Service, cinema\_promo)$;
$\textsf{get\_info}(Self, Service, ft\_number)\rangle\varphi$

(b) $[\textsf{get\_info}(Self, Service, Fluent)]\varphi \subset [\textsf{inform}(Service, Self, Fluent)]\varphi$

Procedure (a) is the protocol procedure: the customer asks if a table is available at a certain time, if so, the restaurant informs it that a reservation has been taken and that it gained a promotional free ticket for a cinema ($cinema\_promo$), whose code number (ft_number) is returned. Clause (b) shows how get_info can be implemented as an inform act executed by the service and having as recipient the customer. The question mark amounts to check the value of a fluent in the current state; the semicolon is the sequencing operator of two actions. The cinema protocol, instead, is:

(c) $\langle \textsf{reserv\_cinema}_C(Self, Service, Movie)\rangle\varphi \subset$
$\langle \textsf{yes\_no\_query}_Q(Self, Service, available(Movie))$;
$\mathcal{B}^{Self} available(Movie)?$;
$\textsf{yes\_no\_query}_I(Self, Service, cinema\_promo)$;
$\neg\mathcal{B}^{Self} cinema\_promo?$;
$\textsf{yes\_no\_query}_I(Self, Service, pay\_by(c\_card))$;
$\mathcal{B}^{Self} pay\_by(c\_card)?$;
$\textsf{inform}(Self, Service, cc\_number)$;
$\textsf{get\_info}(Self, Service, reservation(Movie))\rangle\varphi$

(d) $\langle \mathsf{reserv\_cinema}_C(Self, Service, Movie)\rangle\varphi \subset$
$\langle \mathsf{yes\_no\_query}_Q(Self, Service, available(Movie))$;
$\mathcal{B}^{Self} available(Movie)?$;
$\mathsf{yes\_no\_query}_I(Self, Service, cinema\_promo)$;
$\mathcal{B}^{Self} cinema\_promo?$;
$\mathsf{inform}(Self, Service, ft\_number)$;
$\mathsf{get\_info}(Self, Service, reservation(Movie))\rangle\varphi$

Supposing that the desired movie is available, the cinema alternatively accepts credit card payments (c) or promotional tickets (d). *We can verify if the two implementations can be composed with the desired effect*, by using the reasoning mechanisms embedded in the language and answering to the query:

$$\langle \mathsf{reserv\_rest}_C(customer, restaurant, dinner) ; \mathsf{reserv\_cinema}_C(customer, cinema, movie)\rangle (\mathcal{B}^{customer} cinema\_promo \wedge \mathcal{B}^{customer} reservation(dinner) \wedge \mathcal{B}^{customer} reservation(movie) \wedge \mathcal{B}^{customer}\mathcal{B}^{cinema} ft\_number)$$

This query amounts to determine if it is possible to compose the interaction so to reserve a table for dinner ($\mathcal{B}^{customer} reservation(dinner)$) and to book a ticket for the movie *movie* ($\mathcal{B}^{customer} reservation(movie)$), exploiting a promotion ($\mathcal{B}^{customer} cinema\_promo$). The obtained free ticket is to be spent ($\mathcal{B}^{customer}$ $\mathcal{B}^{cinema}$ ft_number), i.e., *customer* believes that after the conversation the chosen cinema will know the number of the ticket given by the selected restaurant. If the customer has neither a reservation for dinner nor one for the cinema or a free ticket, the query succeeds, returning the following linear plan:

$\mathsf{queryIf}(customer, restaurant, available(dinner))$;
$\boxed{\mathsf{inform}(restaurant, customer, available(dinner))\ ;}$
$\mathsf{inform}(restaurant, customer, reservation(dinner))$;
$\mathsf{inform}(restaurant, customer, cinema\_promo)$;
$\mathsf{inform}(restaurant, customer, ft\_number)$;
$\mathsf{queryIf}(customer, cinema, available(movie))$;
$\boxed{\mathsf{inform}(cinema, customer, available(movie))\ ;}$
$\mathsf{queryIf}(cinema, customer, cinema\_promo)$;
$\mathsf{inform}(customer, cinema, cinema\_promo)$;
$\mathsf{inform}(customer, cinema, ft\_number)$;
$\mathsf{inform}(cinema, customer, reservation(movie))$

This means that there is first a conversation between *customer* and *restaurant* and, then, a conversation between *customer* and *cinema*, that are instances of the respective conversation protocols, after which the desired condition holds. The linear plan, will, actually lead to the desired goal given that some *assumptions* about the provider's answers hold. In the above plan, assumptions have been outlined with a box. For instance, an assumption for reserving a seat at a cinema is that there is a free seat, a fact that can be known only at execution time. Assumptions occur when the interlocutor can respond in different ways depending on its internal state. It is not possible to know in this phase which the answer will be, but since the set of the possible answers is given by the protocol, it is possible to identify the subset that leads to the goal. In the example they are answers foreseen by a yes_no_query protocol (see Figure 2 (i) and [7]). Returning such assumptions to the designer is also very important to understand the correctness of the implementation also with respect to the chosen speech act ontology.

Using DyLOG as an implementation language is useful also for other purposes. For instance, if a library of protocol implementations is available, a designer might will to search for one that fits the requirements of some new project. Let us suppose, for instance, that the developer must design a protocol for a restaurant where a reservation can be made, not necessarily using a credit card. The developer will, then, search the library of available protocol implementations, looking for one that satisfies this request. Given that *search_service* is a procedure for searching in a library for a given category of protocol, a protocol fits the request if there is at least one conversation generated by it after which $\neg\mathcal{B}^{service} cc\_number$; such a conversation can be found by answering to the existential query:

$$\langle search\_service(restaurant, Protocol)\ ;\ Protocol(customer, service, time)\rangle$$
$$(\mathcal{B}^{customer}\neg\mathcal{B}^{service} cc\_number \wedge \mathcal{B}^{customer} reservation(time))$$

which means: find a protocol with at least one execution after which the customer is sure that the provider does not know his/her credit card number and a reservation has been taken.

5 Generating and Executing Jess Agents That Adhere to the AUML Protocols

From the AUML sequence diagrams represented in Figure 4, and by defining two more AUML diagrams that provide information on the classes and instances of agents that will be involved in the MAS ("class diagram" and "agent diagram", see [4, 29]) we can automatically generate the Jess code for the given agent classes. Here, by "agent class" we mean a group of agents that share the same role (in the *restaurant + cinema* example the roles are customer, cinema and restaurant) and the same internal structure (in the *restaurant + cinema* example agents are conceptualized using mental attitudes, thus we can assume that their internal structure is based on a BDI-style architecture). The code for the program that characterizes each class must be completed by adding the conditions under which a message can be sent. In the diagrams in Figure 4, these conditions appear just above the message which labels each arrow, thus the developer can easily add them to the Jess code. Once the code is completed, the developer must define the initial state of the agent instances. The information about the initial state cannot be found in the diagrams in Figure 4, since these diagrams describe general patterns of interaction between roles, rather than between instances of agents, and they abstract from the details that characterize the agents' state.

As an example, the Jess rule shown in Table 1 is taken from the program of the agents that play the Cinema role. It manages the situation in which the Cinema agent has received a queryIf(available(Movie)) message from an agent playing the role of Customer, and that there are seats available for Movie. In this case an inform(available(Movie)) message is to be sent to the Customer agent[2]. The bold font indicates the part of code added by the developer. The added code, **(seats ?movie ?s)** and **(> ?s 0)**, allows to retrieve the seats available for movie, and to verify that they are more than zero.

Table 1. Jess rule for the Cinema agent class

```
(defrule E_2_1_1
   (state E_1 ?cid)
   (seats ?movie ?s)
   (¿ ?s 0)
 =>
   (assert (state E_2_1_1 ?cid))
   (retract-string
      (str-cat "(state E_1 " ?cid " )"))
   (send (assert (ACLMessage
      (communicative-act inform)
      (role-sender Cinema) (role-receiver Customer)
      (conversation-id ?cid) (content (available ?movie))))) )))
```

The developer will be interested in configuring simulation runs which differ from the initial state of the agents involved, and check that, whatever the initial state may be, the interaction protocols are always followed and the properties verified using DyLOG are always satisfied. For each simulation run, once the initial state of the agents has been defined, the Java classes for interfacing Jess and JADE can be automatically created and the resulting JADE prototype can be executed.

The agent's state determines the protocol diagram branch that will be followed in a simulation run. As an example, let us suppose that the customer agent Customer_1 sends a queryIf(available(the_lord_of_the_rings)) request to the cinema agent Cinema_1. If the current state of Cinema_1 includes the information (seats ?the_lord_of_the_rings 2), the client request can be accepted and the number of available seats for the "The Lord of the Rings" movie is updated consequently. Cinema_1 will then ask to Customer_1 if it adheres to the promotional offer of a free ticket. Since Customer_1 adheres to the offer, it will issue an inform(cinema_promo) message followed by the number of its free ticket. The interaction ends when Cinema_1 confirms the reservation by sending an inform(reservation(the_lord_of_the_rings)) message to Customer_1.

[2] The syntax of messages used in both Figure 4 and this paragraph is Prolog-like, while Jess uses a Lisp-like syntax with variables preceeded by a question mark. Messages can be easily converted from the Prolog-like syntax to the Lisp-like one, and vice-versa.

Let us also suppose that, besides Customer_1, in the MAS there are two more customer agents, namely Customer_2, which does not adhere to the promotional offer, and Customer_3, which adheres to the promotional offer. Both of them want to buy a ticket for the "The Lord of the Rings" movie. Customer_2 asks if there are available seats to Cinema_1 and gets the information that there is one. Cinema_1 considers this seat as reserved, and thus, when Customer_3 asks for available seats, it answers that there are no more left: the ones initially possessed by Cinema_1 have already been issued to Customer_1 and Customer_2.

The performatives of messages exchanged between Cinema_1 and Customer_1 can be seen in Figure 4 which shows the output of the JADE Sniffer agent. Figure 5 shows the details of the message that Cinema_1 sends to Customer_3 to inform it that there are no seats left.

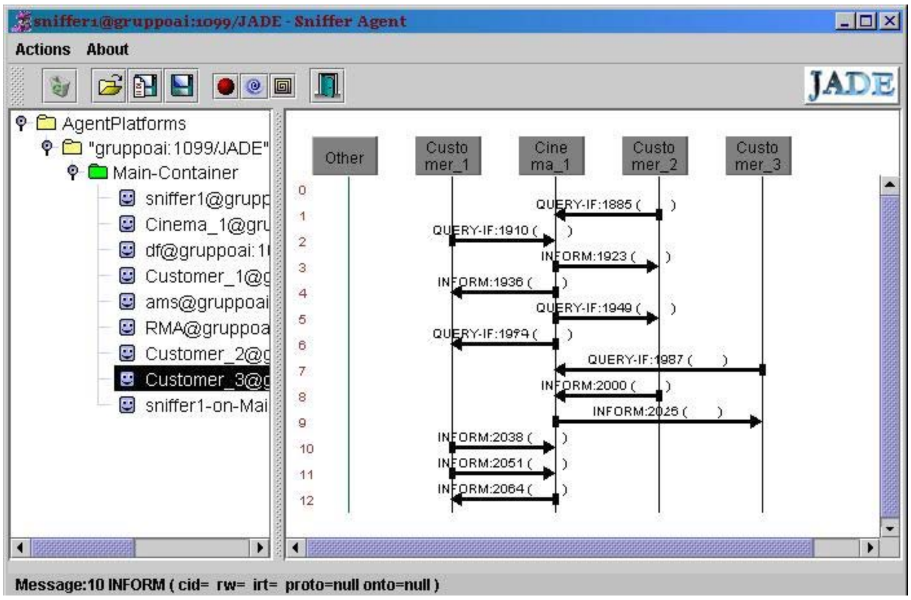

Fig. 4. Interactions between three customers interested in the "The Lord of the Rings" movie, and a cinema

By running the prototype a sufficient number of times starting from as many different agents' initial states as possible, all the possible evolutions of the MAS should be observed. If the software engineer who captured the requirements of the system using the AUML diagrams of Figure 4 forgot to describe some interaction patterns or described them incorrectly, and the verification carried out by means of DyLOG did not allow to discover these deficiencies, the prototype execution may help the developer in completing (resp. correcting) the missing (resp. incorrect) interaction patterns.

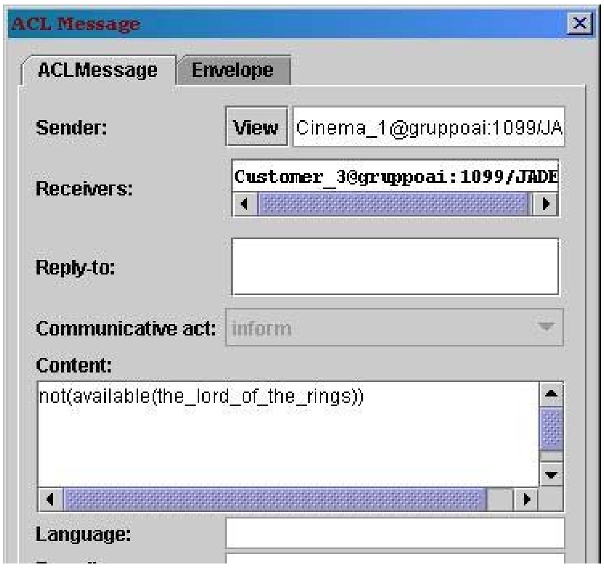

Fig. 5. Details of the last message sent by Cinema_1 to Customer_3

The possibility to both verify some properties of a set of AUML diagrams by means of their translation into DyLOG, and *animate* the diagrams by creating a simulation of a MAS, helps the MAS engineer in the task of developing a *real MAS* that is correct w.r.t. the initial requirements. Once the simulation of the MAS works properly, the real MAS can be obtained by substituting the agents developed using Jess, with agents that show the same behavior but are developed using Java[3]. A prototype of the MAS that includes only Java agents, is very close to a final implementation. Java agents can easily act as interfaces towards existing services, databases or the Web thus allowing the integration of legacy software and data and the interaction with Web services.

The integration of DyLOG inside DCaseLP, although just methodological, is a step forward towards achieving the goal of making DCaseLP a truly *multi lingual* environment, where agents that are *heterogeneous* in both the language they are specified/implemented and in their internal architecture[4] are used in the different stages of the engineering process.

[3] The substitution should be carried out in such a way that the internal and social behavior of the Java agents is exactly the same as the one of the Jess agents. For the moment, techniques and tools for proving the correctness of the substitution are not provided with the DCaseLP environment: the MAS developer must ensure this correctness by him-/herself.

[4] The agents of the *restaurant + cinema* example have a BDI-like architecture, but simpler reactive or proactive agents could be specified/implemented as well using Jess and Java.

6 Conclusions and Related Work

AOSE does not yet supply solid and complete environments for the seamless integration of *heterogeneous* specification and implementation languages, nonetheless, some interesting results have already been achieved with the development of prototypical environments for engineering *heterogeneous agents*. Just to cite some of them, the AgentTool development system [2] 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 [14], which can be used by the system designer to graphically define a high-level system behavior. The system designer defines the types of agents in the system as well as the possible communications that may take place between them. This system-level specification is then refined for each type of agent in the system. To refine the specification of an agent, the designer either selects or creates an agent's architecture and then provides detailed behavioral specification for each component in such architecture. Zeus [38] is an environment developed by British Telecommunications for specifying and implementing collaborative agents, following a clear methodology and using the software tools provided by the environment. The approach of Zeus to the development of a MAS consists of analysis, design, realization and runtime support. The first two stages of the methodology are described in detail in the documentation, but only the last two stages are supported by software tools. The description of other prototyping environments can be found starting from the UMBC Web Site (`http://agents.umbc.edu`) and following the path `Applications and Software, Software, Academic, Platforms`. The reader can refer to [15] for a comparison between some of them, including the predecessor of DCaseLP (CaseLP).

In respect to the existing MAS prototyping environments, DCaseLP stresses the aspect of *multi-language support* to cope with the heterogeneity of both the views of the MAS and the agents. This aspect is usually not considered in depth, and this is the reason why we opted to work with DCaseLP rather than with other existing environments. In particular, in this paper we have focused on the *methodological* integration of the agent logic-based implementation language DyLOG into the MAS prototyping environment DCaseLP, with the main aim of exploiting the formal methods supported by DyLOG in order to reason about agent protocol-driven interactions.

A methodology for integrating DyLOG into DCaseLP has been proposed that is based on the semi-automatic generation of a DyLOG implementation from an AUML sequence diagram, in a similar way as it has been done for the AUML → Jess translation [4]. Such an integration allows to support the MAS developer in many ways. In fact, by means of this integration we add to DCaseLP the ability of verifying properties of the implemented protocols during the *design phase* of the MAS; this feature is not offered by DCaseLP (without DyLOG) since protocols can only be translated into Jess code and executed. The ability of reasoning about possible interactions is very useful in many practical tasks. In this paper we have shown a couple of examples of use: *selection* of already developed protocols from a library and verification of *compositional properties*.

In recent work, part of the authors have used formal methods for proving other kinds of properties of the interaction protocols implemented in DyLOG. In particular, we have faced the *conformance* problem, which amounts to determine if a given protocol implementation respects a protocol specification (in our case the specification language is AUML). In [8] we have, in fact, proposed three definitions of conformance, characterized by different levels of abstraction from the agent private mental state, we have shown that by interpreting the conformance test as a problem of language inclusion, protocol conformance (the strongest of the three) is actually decidable and tractable (see Figure 3).

In the future, we mean to study the application of other techniques derived from the area of logic-based protocol verification [18] where the problem of proving universal properties of interaction protocols (i.e., properties that hold after every possible execution of the protocol) is faced. Such techniques could be exploited to perform the *validation stage* [25] in order to check the coherence of the AUML description with the specifications derived from the analysis. This is usually done by defining a model of the protocol (AUML) and expressing the specification by a temporal logic formula; thus model checking techniques test if the model satisfies the temporal logic formula.

Acknowledgement

This research is partially supported by MIUR Cofin 2003 "Logic-based development and verification of multi-agent systems (MASSiVE)" national project.

References

1. Advanced logic in computing environment. Available at `http://www.di.unito.it/~alice/`.
2. AgentTool development system. http://www.cis.ksu.edu/ sdeloach/ai/projects/agentTool/agentool.htm.
3. K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V.S. Subrahmanian. IMPACT: a platform for collaborating agents. *IEEE Intelligent Systems*, 14(2):64–72, 1999.
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 J. Debenham and K. Zhang, editors, *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, pages 578–585. The Knowledge System Institute, 2003.
5. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proceedings of SEKE 2002*. ACM Press, 2002.
6. AUML Home Page. `http://www.auml.org`.
7. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about self and others: communicating agents in a modal action logic. In C. Blundo and C. Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS'2003*, volume 2841 of *LNCS*, pages 228–241, Bertinoro, Italy, October 2003. Springer.

8. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying protocol conformance for logic-based communicating agents. In J. Leite and P. Torroni, editors, *Pre-Proc. of 5th Int. Workshop on Computational Logic in Multi-Agent Systems, CLIMA V*, pages 82–97, Lisbon, Portugal, September 2004.
9. 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*, 41(2-4):207–257, 2004.
10. J. Bentahar, B. Moulin, J. J. Ch. Meyer, and B. Chaib-Draa. A computational model for conversation policies for agent communication. In J. Leite and P. Torroni, editors, *Pre-Proc. of 5th Int. Workshop on Computational Logic in Multi-Agent Systems, CLIMA V*, pages 66–81, Lisbon, Portugal, September 2004.
11. A. Bracciali, P. Mancarella, K. Stathis, and F. Toni. On modelling declaratively multiagent systems. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, LNCS 3476, Springer-Verlag (2005). In this volume.
12. L. Cabac and D. Moldt. Formal semantics for AUML agent interaction protocol diagrams. In *Proceedings of Agent-Oriented Software Engineering (AOSE), 2004*.
13. G. De Giacomo, Y. Lespérance, and H. J. Levesque. CONGOLOG, a concurrent programming language based on situation calculus. *Artificial Intelligence*, 121:109–169, 2000.
14. S. A. DeLoach. *Methodologies and Software Engineering for Agent Systems*, chapter The MaSE Methodology. Kluwer Academic Publisher, 2004. To appear.
15. T. Eiter and V. Mascardi. Comparing Environments for Developing Software Agents. *AI Communications*, 15(4):169–197, 2002.
16. FIPA Specifications. `http://www.fipa.org`.
17. M. Fisher. A survey of concurrent METATEM - the language and its applications. In D. M. Gabbay and H.J. Ohlbach, editors, *Proc. of the 1st Int. Conf. on Temporal Logic (ICTL'94)*, LNCS 827, pages 480–505. Springer-Verlag, 1994.
18. L. Giordano, A. Martelli, and C. Schwind. Specifying and Verifying Systems of Communicating Agents in a Temporal Action Logic. In A. Cappelli and F. Turini, editors, *Proc. of the 8th Conf. of AI*IA*, LNAI 2829, Springer-Verlag, 2003.
19. L. Giordano, A. Martelli, and C. Schwind. Verifying communicating agents by model checking in a temporal action logic. In J. Alferes and J. Leite, editors, *9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, LNAI 3229, pages 57–69, Lisbon, Portugal, Sept. 2004. Springer-Verlag.
20. F. Guerin and J. Pitt. Verification and Compliance Testing. In M.P. Huget, editor, *Communication in Multiagent Systems*, LNAI 2650, pages 98–112. Springer-Verlag, 2003.
21. 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`.
22. G. Gutnik and G. Kaminka. A scalable Petri Net representation of interaction protocols for overhearing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 3, pages 1246–1247, 2004.
23. G. Gutnik and G.A. Kaminka. A comprehensive Petri Net representation for multi-agent conversations. Technical Report 2004/1, Bar-Ilan University, 2004.
24. M-P. Huget. Model checking agent UML protocol diagrams. Technical Report ULCS–02–012, CS Department, University of Liverpool, UK, 2002.

25. M.P. Huget and J.L. Koning. Interaction Protocol Engineering. In M.P. Huget, editor, *Communication in Multiagent Systems*, LNAI 2650, pages 179–193. Springer, 2003.
26. JADE Home Page. `http://jade.cselt.it/`.
27. Jess Home Page. `http://herzberg.ca.sandia.gov/jess/`.
28. J. Leite, A. Omicini, P. Torroni, and P. Yolum, editors. *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, LNCS 3476, Springer-Verlag (2005). This volume.
29. M. Martelli and V. Mascardi. From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents. In F. Buccafurri, editor, *Proceedings of the 8th APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'03)*, pages 275–286, 2003.
30. H. Mazouzi, A. El Fallah Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 517–526. ACM Press, 2002.
31. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence*. 2000.
32. L. R. Pokorny and C. R. Ramakrishnan. Modeling and verification of distributed autonomous agents using logic programming. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, LNCS 3476, Springer-Verlag (2005). In this volume.
33. C. Schifanella, L. Lusso, M. Baldoni, and C. Baroglio. Design and development of a visual environment for writing dylog, 2004. Submitted.
34. M. P. Singh. A social semantics for agent communication languages. In *Proc. of IJCAI-98 Workshop on Agent Communication Languages*, Berlin, 2000. Springer.
35. F. Stolzenburg and T. Arai. From the specification of multiagent systems by statecharts to their formal analysis by model checking: Towards safety-critical applications. In M. Schillo, M. Klusch, J. Müller, and H. Tianfield, editors, *Proceedings of the First German Conference on Multiagent System Technologies*, LNAI 2831, pages 131–143. Springer-Verlag, 2003.
36. tuProlog Home Page. `http://lia.deis.unibo.it/research/tuprolog/`.
37. C. Walton. Model checking agent dialogues. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, LNCS 3476, Springer-Verlag (2005). In this volume.
38. ZEUS Home Page. `http://more.btexact.com/projects/agents.htm`.