Communication Architecture in the DALI Logic Programming Agent-Oriented Language*

Stefania Costantini Arianna Tocchio Alessia Verticchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost,tocchio}@di.univaq.it

Abstract. In this paper we describe the communication architecture of the DALI Logic Programming Agent-Oriented language. We have implemented the relevant FIPA compliant primitives, plus others which we believe to be suitable in a logic setting. We have designed a meta-level where: on the one hand the user can specify, via two distinguished primitives tell/told, constraints on communication and/or a communication protocol; on the other hand, meta-rules can be defined for filtering and/or understanding messages via applying ontologies and forms of commonsense and case-based reasoning. These forms of meta-reasoning are automatically applied when needed by a form of reflection.

1 Introduction

Interaction is an important aspect of Multi-agent systems: agents exchange messages, assertions, queries. This, depending on the context and on the application, can be either in order to improve their knowledge, or to reach their goals, or to organize useful cooperation and coordination strategies. In open systems the agents, though possibly based upon different technologies, must speak a common language so as to be able to interact. Agent Communication Languages (ACL), such as the widely-adopted FIPA ACL, provide standardized catalogues of performatives (communication acts), designed in order to ensure interoperability among agent systems [12].

However, beyond standard forms of communication, the agents should be capable of filtering and understanding message contents. A well-understood topic is that of interpreting the content by means of ontologies, which are essentially dictionaries and descriptions that allow different terminologies to be coped with. In a logic language, the use of ontologies can be usefully integrated with forms of commonsense and case-based reasoning, that improve the "understanding" capabilities of an agent. A more subtle point is that it would be useful for an agent to have the possibility to enforce constraints on communication. This implies being able to accept or refuse or rate a message, based on various conditions like for instance the degree of trust in the sender. This also implies to be able to follow a communication protocol in "conversations". Since the degree of trust, the protocol, the ontology, and other factors, can vary with the context,

* We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

or can be learned from previous experience, in a logic language agent should and might be able to perform meta-reasoning on communication, so as to interact flexibly with the "external world."

This paper presents the communication architecture of the DALI language. DALI is an Agent-Oriented Logic Programming language designed for executable specification of logical agents, that allows one to define one or more agents interacting among themselves, with other software entities and with an external environment. A main design objective for DALI has been that of introducing in a declarative fashion all the essential features, while keeping the language as close as possible to the syntax and semantics of the traditional Horn-clause language. In practice, most Prolog programs can be understood as DALI programs. Special atoms and rules have been introduced for representing: external events, to which the agent is able to respond (reactivity); actions (reactivity and proactivity); internal events (previous conclusions which can trigger further activity); past and present events (to be aware of what has happened), goals (that the agent can reach). Then, on the line of the arguments proposed in [10], DALI is an enhanced logic language with fully logical semantics [5], that integrates rationality and reactivity, where an agent is able of both backwards and forward reasoning, and has the capability to enforce "maintenance goals" that preserve her internal state, and "achievement goal" that pursue more specific objectives. An extended resolution is provided, so that the DALI interpreter is able to answer queries like in the plain Horn-clause language, but is also able to cope with the different kinds of events.

We have introduced in DALI a communication architecture that specifies in a flexible way the rules of interaction among agents, according to the above-mentioned criteria. The various aspects are modeled in a declarative way, are adaptable to the user and application needs, and can be easily composed. Basically, DALI agents communicate via FIPA ACL, augmented with some primitives which are suitable for a logic language. As a first layer of the architecture, we have introduced a check level that filters the messages. This layer verifies that the message respects the communication protocol of the agent, as well as some domain-independent coherence properties. Several other properties to be checked can be however additionally specified, by expanding the definition of the distinguished predicates *tell/told*. If the message does not pass the check, it is deleted and does not produce any effect. As a second layer, meta-level reasoning is exploited so as to try to understand messages coming from other software entities by using ontologies, and forms of commonsense reasoning.

In summary, when a DALI agent receives a message by another agent, the message is submitted to a check level that controls if it respects the communication protocol and the conditions expressed by the check rules. If the message gets over this control, the agent invokes meta-level reasoning in order to understand its content. The meta-reasoning process uses the agent's ontology and other properties of the terms occurring in the message.

It is important to notice that the definition of the enhanced DALI/FIPA ACL is imported by the agent's code as a library, so as in principle DALI agents may adopt different communication protocols. Also, checks and constraints on communication can be modified without affecting (or without even knowing) the agent's code. The layers of message check and understanding have a predefined default part. However, as

mentioned before they can be extended, improved and adapted to the specific user or application needs by adding rules to the definition of some distinguished predicated.

In this paper we will not be concerned with formal aspects. Rather, we mean to illustrate the communication architecture, and to demonstrate its usefulness mainly by means of significant examples.

The paper is organized as follows. We start by shortly describing the main features of DALI in Section 2. In Section 3 we discuss the new DALI/FIPA protocol and in Section 4 we introduce DALI communication filter. Then, in Sections 5 and 6 we summarize the meta-reasoning layer and how the new architecture works. In Section 7 we show an example of communication between DALI agents, and then conclude in Section 8 with some final remarks.

2 The DALI language

DALI [4] [5] is an Active Logic Programming language designed for executable specification of logical agents. A DALI agent is a logic program that contains a particular kind of rules, reactive rules, aimed at interacting with an external environment. The environment is perceived in the form of external events, that can be exogenous events, observations, or messages by other agents. In response, a DALI agent can perform actions, send messages, invoke goals. The reactive and proactive behavior of the DALI agent is triggered by several kinds of events: external events, internal, present and past events. It is important to notice that all the events and actions are timestamped, so as to record when they occurred. The new syntactic entities, i.e., predicates related to events and proactivity, are indicated with special postfixes (which are coped with by a preprocessor) so as to be immediately recognized while looking at a program.

2.1 External Events

The external events are syntactically indicated by the postfix *E*. When an event comes into the agent from its "external world", the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The special token $:>$, used instead of $:-$, indicates that reactive rules performs forward reasoning. E. g., the body of the reactive rule below specifies the reaction to the external event *bell_ringsE* that is in the head. In this case the agent performs an action, postfix *A*, that consists in opening the door.

$bell\_ringsE :> open\_the\_doorA.$

The agent remembers to have reacted by converting the external event into a *past event* (time-stamped).

Operationally, if an incoming external event is recognized, i.e., corresponds to the head of a reactive rule, it is added into a list called EV and consumed according to the arrival order, unless priorities are specified. Priorities are listed in a separate file of directives, where (as we will see) the user can "tune" the agent's behaviour under several respect. The advantage introducing a separate initialization file is that for modifying the directives there is no need to modify (or even to understand) the code.

2.2 Internal Events

The internal events define a kind of "individuality" of a DALI agent, making her proactive independently of the environment, of the user and of the other agents, and allowing her to manipulate and revise her knowledge [3]. An internal event is syntactically indicated by the postfix *I*, and its description is composed of two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen.

Internal events are automatically attempted with a default frequency customizable by means of directives in the initialization file. The user's directives can tune several parameters: at which frequency the agent must attempt the internal events; how many times an agent must react to the internal event (forever, once, twice,...) and when (forever, when triggering conditions occur, ...); how long the event must be attempted (until some time, until some terminating conditions, forever).

For instance, consider a situation where an agent prepares a soup that must cook on the fire for K minutes. The predicates with postfix *P* are past events, i.e., events or actions that happened before, and have been recorded. Then, the first rule says that the soup is ready if the agent previously turned on the fire, and K minutes have elapsed since when she put the pan on the stove. The goal *soup_ready* will be attempted from time to time, and will finally succeed when the cooking time will have elapsed. At that point, the agent has to react to this (by second rule) thus removing the pan and switching off the fire, which are two actions (postfix *A*).

$$\begin{aligned} soup\_ready :- \ & turn\_on\_the\_fireP, put\_pan\_on\_the\_stoveP : T, \\ & cooking\_time(K), time\_elapsed(T, K). \\ soup\_readyI :> \ & take\_off\_pan\_from\_stoveA, turn\_off\_the\_fireA. \end{aligned}$$

A suitable directive for this internal event can for instance state that it should be attempted every 60 seconds, starting from when *put_the_pan_on_the_stove* and *turn_on_the_fire* have become past events.

Similarly to external events, internal events which are true by first rule are inserted in a set IV in order to be reacted to (by their second rule). The interpreter, interleaving the different activities, extracts from this set the internal events and triggers the reaction (again according to priorities). A particular kind of internal event is the *goal*, postfix *G*, that stop being attempted as soon as it succeeds for the first time.

2.3 Present Events

When an agent perceives an event from the "external world", it doesn't necessarily react to it immediately: she has the possibility of reasoning about the event, before (or instead of) triggering a reaction. Reasoning also allows a proactive behavior. In this situation, the event is called present event and is indicated by the suffix *N*.

2.4 Actions

Actions are the agent's way of affecting her environment, possibly in reaction to an external or internal event. In DALI, actions (indicated with postfix *A*) may have or not

preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. An action rule is just a plain rule, but in order to emphasize that it is related to an action, we have introduced the new token $:<$, thus adopting the syntax $action :< preconditions$. Similarly to external and internal events, actions are recorded as past actions.

2.5 Past events

Past events represent the agent's "memory", that makes her capable to perform its future activities while having experience of previous events, and of its own previous conclusions. As we have seen in the examples, past event are indicated by the postfix *P*. For instance, *alarm_clock_ ringsP* is an event to which the agent has reacted and which remains in the agent's memory. Each past event has a timestamp T indicating when the recorded event has happened. Memory of course is not unlimited, neither conceptually nor practically: it is possible to set, for each event, for how long it has to be kept in memory, or until which expiring condition. In the implementation, past events are kept for a certain default amount of time, that can be modified by the user through a suitable directive in the initialization file. Implicitly, if a second version of the same past event arrives, with a more recent timestamp, the older event is overridden, unless a directive indicates to keep a number of versions.

3 DALI/FIPA Agent Communication Language

An agent communication language (ACL) is a set of primitives and rules that guide the interaction among several agents [7]. There are a number of standardized languages that the agents can use for communication. The most widely acknowledged is the FIPA ACL, which for the sake of interoperability we have adopted (with few extensions) for DALI. The specification of FIPA messages has the following structure:

- **receiver:** name of the agent that receives the message;
- **language:** the language in which the message is expressed;
- **ontology:** the vocabulary of the words in the message, or, more generally, the description of conceptual relationships between terms and sentences of the same domain, which are expressed in a different terminology;
- **sender:** name of the agent that sends the message;
- **content:** the content of the message, which is the main part, discussed in detail below.

In DALI, a message which has to be sent has the format:

$primitive(content)$

where *primitive* is what is called a *communication performative*, i.e., the specification of the intended meaning of the message, which is then further specified by *content*. For instance, *propose(content)* is a performative aimed at asking another agent to do something, where what should be done is specified by *content*. The DALI interpreter

automatically adds the missing FIPA parameters, thus creating the structure which is actually sent, i.e.:

$$message(\ receiver\_address, receiver\_name, sender\_address, sender\_name,\\ language, ontology, primitive(content, sender))$$

Symmetrically, from a message which is received the interpreter extracts the part *primitive(Content,Sender)*, that is what the receiver agent has to consider. In most cases, the receiver will record the item *primitive(Content,Sender)* as a past event. Please notice that *content* may be a conjunction, as the FIPA performatives have different arities.

In the rest of this section, we illustrate the main performatives we adopt. In brackets we indicate if the primitive is FIPA or if it is peculiar of DALI. In most cases, the receiver will record the item *primitive(Content,Sender)* as a past event.

send_message (DALI)

$$send\_message(external\_event, sender\_agent)$$

The act of sending a message to a DALI agent that the receiver will perceive as the communication that the given external event has happened.

propose (FIPA)

$$propose(action, [precondition_1, ..., precondition_n], sender\_agent).$$

The act of asking another agent to perform a certain action, given certain preconditions. A DALI agent accepts a proposal if the preconditions are all true, else she rejects the proposal.

accept_proposal (FIPA)

$$accept\_proposal(\ action\_accepted, [condition_1, ..., condition_n], sender\_agent) or\\ accept\_proposal(\ action\_accepted, [condition_1, ..., condition_n], in\_response\_to(),\\ sender\_agent)$$

The action of accepting a previously received proposal to perform an action where the the conditions of the agreement are enclosed.

reject_proposal (FIPA)

$$reject\_proposal(\ action\_rejected, [reason_1, ..., reason_n], sender\_agent) or\\ reject\_proposal(\ action\_rejected, [reason_1, ..., reason_n], in\_response\_to(),\\ sender\_agent)$$

The action of rejecting a proposal to perform some action during a negotiation, listing the reasons for rejection.

failure (FIPA)

$$failure(action\_failed, motivation, sender\_agent)$$

The action of telling another agent that an action was attempted but the attempt failed, enclosing the motivation of the failure.

cancel (FIPA)

$$cancel(action\_to\_cancel, sender\_agent).$$

The action of canceling some previously requested action which had a temporal extent (i.e. cannot be instantaneous).

execute_proc (DALI)

$execute\_proc(call\_procedure, sender\_agent).$

The act of invoking a procedure inside a DALI program.

query_ref (FIPA)

$query\_ref(property, N, sender\_agent)$

The action of asking another agent for the object referred to by an expression containing free variables. *Property* is the string on which the matching is attempted, and N is the requested number of matches for the object to be identified.

inform (FIPA)

$inform(something, sender\_agent)\ or$
$inform(primitive, values/motivation, sender\_agent)$

The sender informs the receiver that a certain "something" is happened.

is_a_fact (DALI)

$is\_a\_fact(proposition, sender\_agent)$

The act of asking if the proposition indicated in the primitive is true.

refuse (FIPA)

$refuse(action\_refused, motivation, sender\_agent)$

The action of refusing to perform a given action, and of explaining the reason for the refusal. For example,a DALI agent refuses to do an action if the preconditions of the corresponding active rules in the DALI logic program are false. The refusal is recorded as a past event.

confirm (FIPA)

$confirm(proposition, sender\_agent)$

The sender informs the receiver that a given proposition is true, where the receiver is supposed to be uncertain about the proposition. The proposition is asserted as a past event.

disconfirm (FIPA)

$disconfirm(proposition, sender\_agent)$

The sender informs the receiver that a given proposition is false, where the receiver is instead supposed to believe that the proposition is true. Then, this proposition is deleted from past events.

4 DALI communication filter

In real applications, the interaction between agents raises the problem of security. If an agent is not sufficiently self-defending, she can suffer from damages to either her knowledge base or her rules. It may happen in fact that an agent sends to another one a message with a wrong content, intentionally or not, thus potentially bringing a serious damage. How to recognize a correct message? And a wrong message?

The solution adopted in DALI is aimed at providing a tool for coping, as far as possible, with these problems. When a message is received, it is examined by a check layer composed of a structure which is adaptable to the context and modifiable by the user. This filter checks the content of the message, and verifies if the conditions for the reception are verified. If the conditions are false, this security level eliminates the supposedly wrong message. We have constrained the reception of messages by restricting the range of allowed utterances to the FIPA/DALI primitives, according to additional conditions defined by the user, or, in perspective, learned by the agent herself. For instance, filtering conditions can be based upon reliability of the sender agent. The DALI filter is specified by means of meta-level rules defining the distinguished predicates *tell* and *told*. These meta-rules are contained in a separate file, and can be changed without affecting or even knowing the DALI code. Then, communication in DALI is elaboration-tolerant with respect to both the protocol, and the filter.

4.1 Filter for the incoming messages

Whenever a message is received, with content part *primitive(Content,Sender)* the DALI interpreter automatically looks for a corresponding *told* rule, which is of the form:

$told(Sender, primitive(Content)) : -constraint_1, \ldots, constraint_n.$

where $constraint\_i$ can be everything expressible either in Prolog or in DALI. If such a rule is found, the interpreter attempts to prove told(Sender, primitive(Content)). If this goal succeeds, then the message is accepted, and primitive(Content)) is added to the set of the external events incoming into the receiver agent. Otherwise, the message is discarded. Semantically, this can be understood as implicit reflection up to the filter layer, followed by a reflection down to whatever activity the agent was doing, with or without accepting the message. For a detailed and general semantic account of this kind of reflection, the reader may refer to [1].

Below we propose a number of examples of filtering rules. Notice that each agent can have her own set of filtering rules. Since she takes these rules from a separate file, her filtering criteria can vary (by importing a different file) according to the context she is presently involved into.

The following rule accepts a *send_message* primitive if the receiver agent remembers (presumably from past experience) that the sender is reliable, and believes that the content is worth knowing.

$told(\ Sender\_agent, send\_message(External\_event)) : -$
$\quad not(unreliableP(Sender\_agent)), interesting(External\_event).$

Similarly, the request of either executing a procedure or asserting a fact is taken into consideration if the agent who is asking us is reliable, and in the former case if we actually have the code of that procedure, in the latter case if we are interested in the new fact.

$told(\ Sender\_agent, execute\_proc(Procedure)) : -$
$\quad not(unreliableP(Sender\_agent)), know(Procedure).$
$told(\ Sender\_agent, is\_a\_fact(Proposition)) : -$
$\quad not(unreliableP(Sender\_agent)), interesting(Proposition).$

A *query_ref* is acceptable, according to the constraint below, if the Sender agent is reliable and friendly.

$$told(\ Sender\_agent, query\_ref(Proposition, 3)) : -$$
$$not(unreliableP(Sender\_agent)), friendly(Sender\_agent).$$

The agent accepts a *confirm* primitive if the Sender is reliable and the proposition is consistent with her knowledge base. The proposition is recorded as a past event and kept, according to the directive specified in this rule, for 200 seconds. Vice versa, the agent disconfirms a proposition that she knows if the Sender is reliable.

$$told(\ Sender\_agent, confirm(Proposition), 200) : -$$
$$not(unreliableP(Sender\_agent)),$$
$$consistent\_with\_knowledge\_base(Proposition).$$
$$told(\ Sender\_agent, disconfirm(Proposition)) : -$$
$$not(unreliableP(Sender\_agent)), in\_knowledge\_base(Proposition).$$

The proposal to do an action is acceptable if the agent is specialized for the action and the Sender is reliable.

$$told(\ Sender\_agent, propose(Action, Preconditions)) : -$$
$$not(unreliableP(Sender\_agent)), specialized\_for(Action).$$

The following constraint checks if the communication protocol is respected. An agent in fact can receive an *accept_proposal* only in response to *propose*. The agent remembers as a past event (for 200 seconds) that she has accepted the proposal to perform an action. This information can be used by an internal event for further inferences.

$$told(\ Sender\_agent, accept\_proposal(Action, Conditions),$$
$$in\_response\_to(Message), 200) : -$$
$$not(unreliableP(Sender\_agent)),$$
$$functor(Message, F, \_), F = propose.$$

We have a similar approach for the other FIPA/DALI primitives *reject_proposal*, *failure*, *refuse* and *inform*.

As the previous examples may have suggested, this model allows one to integrate into the filtering rules the concept the degree of trust. Trust derives from the credibility of the beliefs and of their sources, from the sources' number, convergence, and reliability. All those parameters are easily expressible in the *told* rules. Finally, we emphasize how this communication filter can express constraints not only for a generic communication primitive but also for different contents of the same primitive. For example, we can write:

$$told(Sender\_agent, confirm(love\_me(julie)), forever) : -$$
$$not(unreliableP(Sender\_agent)), ...$$

When a message is deleted, the DALI interpreter displays the primitive that has not been accepted and the reason on the operator console. The console is a special window which is used to activate/stop agents. The user can optionally keep it open in order either to monitor the agents behaviour, or to participate to the conversation by sending messages to the agents.

4.2 Filter layer for outcoming messages

Symmetrically to *told* rules, the messages that an agent sends are subjected to a check via *tell* rules. There is, however, an important difference: the user can choose which messages must be checked and which not. The choice is made by setting some parameters in the initialization file. The syntax of a *tell* rule is:

$tell(Receiver, Sender, primitive(Content)) : -constraint_1, \ldots, constraint_n$

For every message that is being sent, the interpreter automatically checks whether an applicable *tell* rule exists. If so, the message is actually sent only upon success of the goal $tell(Receiver, Sender, primitive(Content))$.

Below we show as an example two of the default rules coping with the DALI/FIPA primitives. The first *tell* rule authorizes the agent to send the message with the primitive *inform* if the receiver is active in the environment and is presumably interested to the information: via rules like this one we can considerably reduce useless exchange of messages. According to the second rule, the agent sends a *refuse* only if the requested primitive is *is_a_fact* or *query_ref*.

$tell(\ Agent\_To, Agent\_From, inform(Proposition)) : -$
$\quad active\_in\_the\_world(Agent\_To),$
$\quad specialized(Agent\_To, Specialization),$
$\quad related\_to(Specialization, Proposition).$
$tell(\ Agent\_To, Agent\_From, refuse(Something, Motivation)) : -$
$\quad arg(1, Something, Primitive),$
$\quad functor(Primitive, F), (F = is\_a\_fact; F = query\_ref).$

5 Meta-reasoning layer

In heterogeneous Multi-agent Systems, in general not all the components speak the same language, and not all of them use the same words to express a concept. The agent that doesn't understand a proposition can either accept the defeat and ignore the message, or try to apply a reasoning process in order to interpret the message contents. The latter solution can be more easily put at work by taking advantage of meta-reasoning capabilities of a logic language. In fact, the use of *ontologies*, which are dictionaries of equivalent terms, can be integrated with several kinds of commonsense reasoning. The ontology of a DALI agent is in a file .txt containing equivalent terms and other properties useful in the meta-reasoning process. E.g., agent *bob*'s ontology can be the following, where *symmetric* is a property of relations, which is asserted to hold of predicate *friend*), and allows him to conclude both *friend(bob,lucy)* and *friend(lucy,bob)* even if originally he could derive only one. The name of the agent enclosed to each item of the ontology allows a group of agents to use the same ontology file, though sharing the contents only partially.

$ontology(bob, rain, water\_falling\_from\_sky).$
$ontology(bob, friend, amico).$
$\ldots\ symmetric(friend).$

Each DALI agent is provided (again in a separate file) with a distinguished procedure called *meta*, to support the meta-reasoning process. This procedure by default includes a number of rules for coping with domain-independent standard situations. The user can add other rules, thus possibly specifying domain-dependent commonsense reasoning strategies for interpreting messages, or implementing a learning strategy to be applied when all else fails. Below we report some of the default meta-reasoning rules that apply the equivalences listed in the ontology, and possibly also exploit symmetry (for binary predicates only):

$$
\begin{array}{l}
meta(\ Initial\_term, Final\_term, Agent\_Sender) :- \\
\quad clause(agent(Agent\_Receiver), \_), \\
\quad functor(Initial\_term, Functor, Arity), Arity = 0, \\
\quad ((ontology(Agent\_Sender, Functor, Equivalent\_term); \\
\quad ontology(Agent\_Sender, Equivalent\_term, Functor)); \\
\quad (ontology(Agent\_Receiver, Functor, Equivalent\_term); \\
\quad ontology(Agent\_Receiver, Equivalent\_term, Functor))), \\
\quad Final\_term = Equivalent\_term. \\
\ldots \\
meta(\ Initial\_term, Final\_term, Agent\_Sender) :- \\
\quad functor(Initial\_term, Functor, Arity), Arity = 2, \\
\quad symmetric(Functor), Initial\_term = ..List, \\
\quad delete(List, Functor, Result\_list), \\
\quad reverse(Result\_list, Reversed\_list), \\
\quad append([Functor], Reversed\_list, Final\_list), \\
\quad Final\_term = ..Final\_list.
\end{array}
$$

The procedure meta is automatically invoked, again via reflection, by the interpreter. It is necessary to avoid unwanted variable bindings while meta-reasoning about messages. In DALI, message contents are always reified, i.e., transformed into a ground term, before they are sent, and thus they are received in reified form. Then, the interpreter includes facilities for "reification", or "naming", and "de-reification", or "unnaming" of language expressions (also this issue is discussed at length in [1]).

6 DALI Communication Architecture

In this section we summarize the overall DALI communication architecture, that puts together the functionalities of filter, meta-reasoning and the protocol layers. The architecture consists of three levels: the first level implements the protocol and the filter, i.e., the first two layers of the communication structure; the second level includes the meta-reasoning layer; the third level consists of the DALI interpreter, which is able to activate the agents. Each agent is defined by a .txt file, containing the agent code written in DALI. When an agent receives an external event through the primitive *send_message*, the DALI interpreter calls the filter layer by invoking its internal rule:

$$
\begin{array}{l}
receive(\ send\_message(External\_event, Agent\_Sender)) :- \\
\quad told(Agent\_Sender, send\_message(External\_event)), ...
\end{array}
$$

If the message overcomes the security check, then the interpreter automatically invokes the meta-level in order to understand the external event. The meta reasoning pro-

cess initially has to "un-name" the message content, so as to verify if (an instance of) *External_event* belong to the set of external events known by the agent without applying the meta procedure. If it is the case, then the agent reacts directly, else the interpreter takes advantage of meta reasoning capabilities for finally finding a reactive rule of the logic program applicable to the context.

Similarly, the meta reasoning level is called for the primitives: *propose*, *execute_proc*, *query_ref* and *is_a_fact*. The procedure *meta* in fact contains a special rule for each different communication act. E. g., the code for the primitive *propose* is reported below: if the action proposed belong to the actions occurring in the logic program of agent, then the interpreter sends back to the proposer agent a message *accept_proposal*, else a *reject_proposal*.

$$
\begin{array}{ll}
receive(propose(& Action, Conditions, Sender\_Agent)) : - \\
 & told(Sender\_Agent, propose(Action, Conditions)), \ldots, \\
 & call\_meta\_propose(Action, Conditions, Sender\_Agent). \\
call\_meta\_propose(& Action, Conditions, Sender\_Agent) : - \\
 & once(call\_propose(Action, Conditions, Sender\_Agent)). \\
call\_propose(& Action, Conditions, Sender\_Agent) : - \\
 & denaming(Action, New\_action), exists\_action(New\_action), \\
 & execute\_propose(New\_action, Conditions, Sender\_Agent). \\
call\_propose(& Action, Conditions, Sender\_Agent) : - \\
 & meta(Action, New\_term, \_), denaming(New\_term, New\_action), \\
 & exists\_action(New\_action), \\
 & execute\_propose(New\_action, Conditions, Sender\_Agent).
\end{array}
$$

In the first rule of *call_propose* the interpreter, after the de-naming of the term, verifies whether the agent knows the action, without applying either the ontology or other properties. In the second rule, called if the first one fails, the interpreter changes the name of the action by applying the meta reasoning and, after the de-naming, checks again if (an instance of) the required action exists among those feasible by the agent. The link with the part of the DALI interpreter that handles events, goals and actions is represented by the procedure *execute_propose*. Via this procedure the action, if recognized, is put into the queue of actions that are waiting to be executed.

7 Example: an Italian client in an English pub

We propose an example of interaction between DALI agents employing the communication architecture that we have discussed. We consider four agents: (i) **agent waiter:** he is a pub (or cafeteria) waiter that receives orders and serves drinks to clients; (ii) **agent *gino*:** an italian client, who walks into the cafeteria and orders a beer incorrectly; (iii) **agent *wife*:** wife of another client, **bob**, who is a drunkard and never finds his way home; (iv) **agent *friend*:** friend of the italian client *gino*.

The italian client *gino* walks into the pub and orders a beer, mixing Italian and English languages and mispelling the word *beer*. He sends to the waiter the message:

$$send\_message(voglio(gino, ber), gino).$$

The waiter speaks a bit of italian, i.e., he applies the item of his ontology: $ontology(\_, voglio, request)$, and understands that *voglio* is equivalent to *request*. But he still doesn't understand *ber*, and thus informs *gino* about this problem.

$$
\begin{array}{ll}
not\_know(C,P) :- & requestP(C,P), not(clause(product(P,\_),\_)).\\
not\_know(C,P) :- & requestP(C,\_,P), not(clause(product(P,\_),\_)).\\
not\_knowI(C,P) :> & clause(agent(A),\_)\ ,\\
 & messageA(C, inform(not\_know(P), A)).
\end{array}
$$

gino, not speaking English very well, asks friend for how to formulate his request correctly:

$$
\begin{array}{ll}
ask\_to\_friend(F) :- & informP(not\_know(F), waiter).\\
ask\_to\_friendI(F) :> & clause(agent(Agent),\_),\\
 & messageA(friend, send\_message(how\_tell(F, Agent), Agent)).
\end{array}
$$

The agent *friend*, aware of the poor English of *gino*, by using the ontology he has learned during their acquaintance (that thus contains the fact ***ontology(gino,ber,beer)***), informs the waiter that *ber* is equivalent to *beer*.

$$
\begin{array}{ll}
how\_tellE(F, Agent) :> & clause(agent(Ag),\_),\\
 & clause(ontology(Ag, F, P1),\_),\\
 & messageA(waiter, inform(how\_tell(Agent, F, P1), Ag)).
\end{array}
$$

The waiter, when receives the information about the term *ber* adds it to his ontology and serves the beer (if available).

$$
\begin{array}{ll}
request(Agent, F, P1) :- & informP(how\_tell(Agent, F, P1),\_).\\
requestI(Agent, F, P1) :- & assert(ontology(Agent, F, P1)).\\
serve\_drink(C, F) :- & requestP(C,\_, F), available(F).\\
serve\_drinkI(C, F) :> & serveA(C, F).
\end{array}
$$

The agent *wife*, if the husband *bob* isn't back home by 11 p.m., tries to go to the pub in order to find out if *bob* is there. She asks the waiter (by using the primitive is_a_fact) if *bob* is in the pub. The waiter responds by the primitive *inform*, according to the DALI/FIPA protocol.

$$
\begin{array}{ll}
not\_at\_home(Husband, Today) :- & missing\_husbandP(Husband),\\
 & datime(T), arg(3, T, Today),\\
 & arg(4, T, Hour), Hour >= 23.\\
not\_at\_homeI(Husband, Today) :> & clause(agent(Agent),\_), go\_to\_pubA,\\
 & messageA(waiter,\\
 & is\_a\_fact(in\_pub(M, Today), Agent)).
\end{array}
$$

Then, by exploiting the content of the primitive *inform* sent back by the waiter (who records all clients that enter and exit the pub), knows that the husband is at the pub. She reacts by screaming, taking her husband home and telling to the waiter that he mustn't serve alcoholic drinks to her husband (including wine).

$$
\begin{array}{ll}
husband\_in\_pub :- & informP(agree(in\_pub(gino, 9)), values(yes), waiter),\\
 & messageA(waiter, inform(not\_serve(gino, wine), wife)),\\
 & messageA(waiter, confirm(alcoholic(wine), wife)).
\end{array}
$$

The *inform* and *confirm* primitives sent to the waiter are used by the told rule of waiter's check layer:

$told(\ Ag, send\_message(request(Ag, P))) : -$
$\quad not(informP(not\_serve(Ag, P), wife)), alcoholicP(P).$

When the drunkard husband will ask for an alcoholic drink in the future, the message will be eliminated.

8 Conclusions

This paper has discussed how communication has been designed and implemented in the DALI Logic Programming Agent-Oriented language. We have shown the kinds of interaction and the abstract roles that the DALI/FIPA protocol supports and the functionalities of the check and meta-reasoning layers. We have noticed that the proposed architecture takes profit of features which are proper of logic languages, such as meta-reasoning and logical reflection.

There are other FIPA-compliant agent frameworks. A future aim of our experiments in fact is that of ensuring interoperability between DALI and these other approaches. A relevant one is JADE, a FIPA-compliant framework fully developed in Java. Each agent platform, written in Java and importing the JADE libraries, can be split on several hosts. Each agent is implemented as a Java thread, and Java events are used for effective and light-weight communication between agents on the same host. A number of FIPA-compliant DFs (Directory Facilitators) agents can be started at run time in order to implement multi-domain applications. About security, JADE provides proper mechanisms to authenticate and verify the rights assigned to agents. [2]

The 3APL platform is the first platform that has supported easy and direct implementation and execution of cognitive agents. The platform can be distributed across different machines connected in a network. Moreover, the 3APL platform is FIPA compliant in the sense that agents running on this platform can in principle communicate with agents that run on a different FIPA compliant platforms such as JADE. [11]

However, the DALI project demonstrates that a logic programming language with a logic semantics [5] is able to exhibit, also for communication, features that are as powerful (and even more flexible) as those of approaches that are either not fully logical, or semantically more complex. The DALI implementation cannot currently compete in efficiency with others like JADE, which have been developed in the industry, and on which a lot of effort has been spent by several companies and universities. Nevertheless, DALI is competitive from the point of view of the ease and flexibility of use, for every kind of application, but especially where context-sensitivity, adaptability and intelligence are needed. We have equipped DALI with an interface with Java, that has allowed us to develop applications (namely in component management and reconfiguration in distributed systems [3]) where the Java part is able to interact at a low level with legacy systems, and the DALI part implements intelligent reasoning and sophisticated interaction. The declarative semantics of DALI has been defined in [5]. The operational semantics is being defined in a Ph.D. Thesis. The behaviour of DALI interpreter has been modeled and checked by using the Murϕ model checker [9]. A future aim of this research is that of developing and experimenting cooperative models for DALI logical agents, also based on game theory. As a first step, we are studying formal models for

making DALI agents adaptive with respect to the level of trust that they assign to the other agents.

References

1. J. Barklund, S. Costantini, P. Dell'Acqua e G. A. Lanzarone, *Reflection Principles in Computational Logic*, Journal of Logic and Computation, Vol. 10, N. 6, December 2000, Oxford University Press, UK.
2. F. Bellifemine, A. Poggi, G. Rimassa, *JADE: A FIPA-compliant agent framework*, In: *Proc. of the 4th International Conference and Exhibition on The Pratical Application of Intelligent Agents and Multiagents*, held in London,UK, December 1999.
3. M. Castaldi, S. Costantini, S. Gentile, A. Tocchio, *A Logic-Based Infrastructure for Reconfiguring Applications*, In: J. A. Leite, A. Omicini. L. Sterling, P. Torroni (eds.), *Declarative Agent Languages and Technologies, Proc. of the 1st International Workshop, DALT 2003* (held in Melbourne, Victoria, July 2003), Available also on-line, URL http://centria.di.fct.unl.pt/ jleite/dalt03/papers/dalt2003proceedings.pdf.
4. S. Costantini. Towards active logic programming. In A. Brogi and P. Hill, editors, *Proc. of 2nd International Workshop on component-based Software Development in Computational Logic (COCL'99)*, PLI'99, (held in Paris, France, September 1999), Available on-line, URL http://www.di.unipi.it/ brogi/ResearchActivity/COCL99/proceedings/index.html.
5. S. Costantini and A. Tocchio, *A Logic Programming Language for Multi-agent Systems*, In S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), *Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*, (held in Cosenza, Italy, September 2002), LNAI 2424, Springer-Verlag, Berlin, 2002.
6. U. Endriss, N. Maudet, F. Sadri, F. Toni, *Logic-based Agent Communication Protocols*, In: *Lecture Notes in Computer Science, Advances in Agent Communication, Proc. of the International Workshop on Agent Communication Languages ACL03*, (held in Melbourne, Australia, July 14 2003), LNCS 2922, Springer-Verlag, Berlin, 2004.
7. M.-P. Huget, J.-L. Koning, *Interaction Protocol Engineering*, In: *Lecture Notes in Computer Science, Communication in Multiagent Systems, Agent Communication Languages and Conversation Policies*, LNCS 2650, Springer-Verlag, Berlin, 2003.
8. M. N. Huhns, L. M. Stephens, *Multiagent System and Societies of Agents*, In: *Multiagent Systems: A modern Approach to Distributed Artificial Intelligence*, 1999.
9. B. Intrigila, I. Melatti, A. Tocchio, *Model-checking DALI with Murϕ*, Tech. Rep., Univ. of L'Aquila, 2004.
10. R. A. Kowalski, *How to be Artificially Intelligent - the Logical Way*, Draft, revised February 2004, Available on line, URL http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html.
11. E.C. Van der Hoeve, M. Dastani, F. Dignum, J.-J. Meyer, *3APL Platform*, In: *Proc. of the The 15th Belgian-Dutch Conference on Artificial Intelligence(BNAIC2003)*, held in Nijmegen, The Netherlands, 2003.
12. J. M. Serrano, S. Ossowski. *An Organisational Approach to the Design of Interaction Protocols*, In: *Lecture Notes in Computer Science, Communications in Multiagent Systems: Agent Communication Languages and Conversation Policies*, LNCS 2650, Springer-Verlag, Berlin, 2003.