A Logic Programming Language for Multi-Agent Systems*

Stefania Costantini Arianna Tocchio

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

Abstract. This paper presents a new logic programming language for modelling Agents and Multi-Agent systems in computational logic. The basic objective of the specification of this new language has been the identification and the formalization of what we consider to be the basic patterns for reactivity, proactivity, internal "thinking", and "memory". The formalization models these concepts by introducing different kinds of events, with a suitable treatment. We introduce a novel approach to the language semantics, called the evolutionary semantics.

1 Introduction

This paper presents a new logic programming language for modelling Agents and Multi-Agent systems in computational logic. Traditional logic programming has proved over time to be a good knowledge representation language for rational agents. Logical agents may possibly interact with an external environment by means of a suitably defined observe–think–act cycle. Significant attempts have been made in the last decade to integrate rationality with reactivity and proactivity in logic programming (see for instance [3], [20], [19], [13], [14] and [9] for a discussion). In [10] we have called logic programming augmented with reactive and proactive features "Active Logic Programming".

DALI is an Active Logic Programming language, designed for executable specification of logical agents, without committing to any specific agent architecture. DALI allows the programmer to define one or more agents, interacting among themselves, with an external environment, or with a user.

The basic objective of the specification of this new language has been the identification and the formalization of what we consider to be the basic patterns for reactivity, proactivity, internal "thinking", and "memory". The formalization

* Research funded by MIUR 40% project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps*. Many thanks to Stefano Gentile, who has joined the DALI project, has cooperated to the implementation of DALI, has designed the language web site, and has helped and supported the authors in many ways.

models these concepts by introducing different kinds of events, with a suitable treatment. We introduce in particular the classes of external, present, past and internal events (Sections 3– 5). Events are represented by special atoms, and are managed by special rules. A limited treatment of time is provided: events are time-stamped, so as internal events are considered at specified intervals and past events are kept or "forgotten" according to suitable conditions.

An important aim in the definition of DALI has been that of introducing in a declarative fashion all the essential features, keeping the language as close as possible to the syntax and semantics of the plain Horn–clause language. DALI inference engine is based on an extended resolution (Section 6).

We have devised a novel approach to the language semantics, called the *evolutionary semantics* (Section 7) In this approach, the semantics of a given DALI program P is defined in terms of a modified program P_s, where reactive and proactive rules are reinterpreted in terms of standard Horn Clause rules. The agent reception of an event is formalized as a program transformation step. The evolutionary semantics consists of a sequence of logic programs, resulting from this subsequent transformations, together with the sequence of the Least Herbrand Model of these programs. Therefore, this makes it possible to reason about the "state"of an agent, without introducing explicitly such a notion, and to reason about the conclusions reached and the actions performed at a certain stage.

The semantic approach we propose in this paper is orthogonal, rather than competing, to the approach of *Updating Logic Programs* (ULP for short) [2] [12] [1]. The common underlying assumption is that of representing state evolution as program evolution. In ULP, program evolution is explicit (and can even be controlled by means of the special-purpose meta-language LUPS) and has the objective of incorporating knowledge changes. In DALI, program evolution is implicit (the program code does not actually change) and is determined by the events. Then, the two approaches do not collide, and might be profitably combined. This is also true for the EPI language [15], which is an extension of LUPS that takes into account external events. There are interesting relationships and similarities between the EPI semantics and the DALI evolutionary semantics, that might lead in the future to a useful integration.

The DALI language was first presented in [10]. Since then, the treatment of events has been extended and refined, the declarative semantics has been properly defined (after a first sketch given in the aforementioned paper), and the language has been implemented. What is completely new in this paper is: (i) the treatment of time, and of the relations among different classes of events; (ii) the approach to belief revision through internal events; (iii) the evolutionary semantics.

A prototype implementation of the DALI language has been developed in Sicstus Prolog by the authors of this paper at the University of L'Aquila. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

2 Syntactic Features of DALI

A DALI program is syntactically very close to a traditional Horn-clause program. In particular, a Prolog program is a special case of a DALI program. Specific syntactic features have been introduced to cope with the agent-oriented capabilities of the language.

DALI agents cope with events, that are represented as special atoms, called *event atoms*. In order to emphasize event atoms, the corresponding predicates are indicated by a particular postfix.

Let us consider an event incoming into the agent from its "external world", like for instance $alarm\_clock\_rings$. From the agent's perspective, this event can be seen in different ways.

Initially, the agent has perceived the event, but she still have not reacted to it. She can however reason about the event (for instance thinking that it is a nuisance to be waked up so early). In this situation, the event is called *present event,* and is written $alarm\_clock\_ringsN$, postfix N standing for "now".

Later on, the agent decides to react to this event (for instance in order to switch the alarm off) At this stage, the event is called *external event,* and written $alarm\_clock\_ringsE$, postfix E standing for "external".

After reaction, the agent is able to remember the event. An event that has happened in the past will be called *past event,* and written $alarm\_clock\_ringsP$, postfix P standing for "past".

A special feature of DALI is the concept of *internal event.* An internal event is a conclusion reached by the agent, to which the agent may want to react, in the sense of triggering further inference. For instance, $food\_is\_finished$ may be such a conclusion, since the agent may want to react and go to buy other food.

Whenever a conclusion is intended to trigger a proactive behavior, it will be called *internal event,* and written $food\_is\_finishedI$, postfix I standing for "internal". The agent is able to remember internal events as well, and then they will become past events.

Some atoms denote *actions* that the agent performs in order to achieve an effect on its environment. To point out actions, each *action atom* like for instance buy_food will be written buy_foodA, postfix A standing for "action". If the agent wants to remember to have previously performed this action, it will be kept in the form buy_foodPA, postfix PA standing for "past action".

All rules are in the form of Horn clauses, but *reactive rules* that have an event in their head are syntactically emphasized. If the head of a rule is an event, then the body of the rule represents the agent's reaction to the event: i.e., the event "determines" the reaction. In order to make it visually evident, the connective ":-" is replaced by ":>", where ":>" reads "determines".

Notice that the conceptual distinction between the different kinds of events and the introduction of reactive rules are fundamental features of the DALI language. Instead, the above-mentioned syntactic notation is syntactic sugar, and therefore it just suits the particular taste of the authors of the paper.

3 External and Present Events, and Actions: Reactivity

The rule below is an example of a *reactive rule* in DALI, modelling a waiter in a cafeteria shop, when a customer approaches.

customer_entersE :> *say_good_morningA*, *offer_helpA*.

Predicate *customer_entersE* denotes an *external event,* i.e. something that happens in the "external world" in which the agent is situated.

To make it recognizable that rules with an external event in the conclusion are reactive rules, the token ":-" has been replaced by ":>". In the declarative semantics however, as discussed in Section 7 this rule is treated as a plain Horn clause. The subgoals in the body are in this case actions (discussed below), recognizable by the postfix A. The body of a reactive rule can consist of any mixture of actions and other subgoals, since a reaction to an event can involve rational activity. Formally, we associate to agent Ag a set of distinct predicates

$$PE_{Ag} = \{pE_1, \ldots, pE_s\}, \quad s \geq 0$$

representing external events, to which the agent is able to respond. An atom of the form $pE_j(Args)$ is called an *event atom.* The set of all the event atoms, which is a subset of the Herbrand Universe, is called E_{Ag}.

Events are recorded in a set $EV \subseteq E_{Ag}$. As soon as an event happens, it is inserted into EV, that represents the "input channel" of the agent. In the declarative semantics, EV is represented by a list, indicating the order in which the agent *consumes* the events. An event is consumed whenever the corresponding reactive rule is activated, in the usual resolution style: when the event atom is unified with the head of a reactive rule with mgu θ, $Body\theta$ is added to the current goal, and the event atom is removed from EV. For each event atom $pE_j(Args)$, Ag may possibly provide **only one** rule of the form:

$$pE_j(Fargs) :> R_{j,1}, \ldots, R_{j,q}. \qquad q \geq 1$$

where $pE_j(Args)$ and $pE_j(FArgs)$ are unifiable.

In the implementation, events are time-stamped, and the order in which they are "consumed "corresponds to the arrival order. The time-stamp can be useful for introducing into the language some (limited) possibility of reasoning about time. It is for instance possible to write the following rule:

customer_entersE: *T* :> *lunchtime(T)*, *offer_appetizersA*.

It is also possible to have a conjunction of events in the head of a reactive rule, like in the following example.

rainE, windE :> *close_windowA*.

In order to trigger the reactive rule, all the events in the head must happen within a certain amount of time. The length of the interval can be set by a directive, and is checked on the time stamps.

An important feature of DALI is that each event atom in EV is also available to the agent as a *present event* (indicated with postfix N, for "Now") and can occur as a subgoal in the body of rules.

The introduction of present events establishes a distinction between *reasoning* about events and *reacting* to events. In the following example, the bell at the door is ringing, and the reaction is that of going to open the door. Before the reaction however, the event is perceived as a present event, allowing the agent to draw the conclusion that a visitor has arrived.

$visitor\_arrived$:- $bell\_ringsN$.

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

Notice that the action subgoals previously seen in the above examples do not occur in the head of any rule. In DALI, these *action atoms* represent actions without preconditions, and always succeed. If however the action cannot properly affect the environment, the interpreter might generate a "failure event", to be managed by a suitable rule. For actions with preconditions, action atoms are defined by *action rules*. In the example below, the agent emits an order for a product P of which she needs a supply. The order can be done either by phone or by fax, in the latter case if a fax machine is available.

$need\_supplyE(P)$:> $emit\_oder(P)$.

$emit\_oder(P)$:- $phone\_orderA$.

$emit\_oder(P)$:- fax_orderA.

fax_orderA :- $fax\_machine\_available$.

Action subgoals always succeed also for actions with preconditions, since the implementation automatically adds a rule of the form: $ActionA$:- $emit\_error\_message$ so as to cope with potential failures.

External events and actions are used also for expressing communication acts. An external event can be a message from another agent, and, symmetrically, an action can consist in sending a message. In fact, current techniques for developing the semantics of Agent Communication Languages trace their origin in speech act theory (see [27] and the references therein), and in the interpretation of speech acts as rational actions [6] [7].

We do not commit to any particular agent communication language, but we attach to each event atom the indication of the agent that has originated the event For events like $rainsE$ there will be the default indication *environment*. Then, an event atom can be more precisely seen as a triple:

$Sender : Event\_Atom : Timestamp$

The $Sender$ and $Timestamp$ fields can be omitted whenever not needed.

4 Past Events: Memory

A DALI agent keeps track of the events that have happened in the past, and of the actions that she has performed. As soon as an event is removed from EV (and then the corresponding reactive rule is triggered), and whenever an action subgoal succeeds (and then the action is performed), the corresponding atom is recorded in the agent database. Past events are indicated by the postfix PE, and past actions by the postfix PA.

Past events are recorded in the form: $Sender : Event\_Atom : Timestamp$ and past actions in the form: $Action\_Atom : Timestamp$. The following rule for instance says that Susan is arriving, since we know her to have left home.

is_arriving(susan) :- *left_homePE(susan).*

The following example illustrates how to exploit past actions. In particular, the action of opening (resp. closing) a window can be performed only if the window is closed (resp. open). The window is closed if the agent remembers to have closed it previously. The window is open if the agent remembers to have opened it previously.

sunny_weatherE :> *open_the_windowA.*
rainy_weatherE :> *close_the_windowA.*
open_the_windowA :- *window_is_closed.*
window_is_closed :- *close_the_windowPA.*
close_the_windowA :- *window_is_open.*
window_is_open :- *open_the_windowPA.*

It is important that an agent is able to remember, but it is also important to be able to forget. In fact, an agent cannot keep track of *every* event and action for an unlimited period of time. Moreover, sometimes subsequent events/actions can make former ones no more valid. In the previous example, the agent will remember to have opened the window. However, as soon as she closes the window this record becomes no longer valid and should be removed: the agent in this case is interested to remember only the last action of the sequence. In the implementation, past events and actions are kept for a certain default amount of time, that can be modified by the user through a suitable directive. Also, the user can express the conditions exemplified below:

keep shop_openPE until 19:30.

The information that the shop is open expires at closing time, and at that time it will be removed. Alternatively, one can specify the terminating condition. As soon as the condition is fulfilled (i.e. the corresponding subgoal is proved) the event is removed.

keep shop_openPE until shop_closed.
keep open_the_windowPA until close_the_windowA.

In particular cases, an event should never be dropped from the knowledge base, as specified below:

keep born(daniele)PE : 27/Aug/1993 forever.

5 Internal Events: Proactivity

A DALI agent is now able to cope with external events, and to reason about the past and the present. We want now to equip the agent with a sort of "consciousness", so as to determine an independent proactive behavior. The language does not commit to a fixed infrastructure: rather, it provides a mechanism that a user can program according to the application at hand.

The mechanism is that of the *internal events*. To the best of our knowledge, this mechanism is an absolute novelty in the context of agent languages. Any subgoal occurring in the head of a rule can play the role of an internal event, provided that it also occurs in the head of a reactive rule, with postfix I. Consider the following example, where the agent goes to buy food as soon as it is finished. The conclusion $finished(Food)$ is interpreted as an event. The internal event is treated exactly like the external ones, i.e. can trigger a reactive rule.

$$\begin{aligned} \textit{finished(Food)} &\ \texttt{:-}\ \textit{eaten(Food).} \\ \textit{finishedI(Food)} &\ \texttt{:>}\ \textit{go\_to\_buyA(Food,Where).} \\ \textit{go\_to\_buyA(Food,bakery)} &\ \texttt{:-}\ \textit{bread\_or\_biscuit(Food).} \\ \textit{go\_to\_buyA(Food,grocery\_shop)} &\ \texttt{:-}\ \textit{dairy(Food).} \end{aligned}$$

Goals corresponding to internal events are automatically attempted from time to time, so as to trigger the reaction as soon as they succeed. The implementation provides a default timing, and also gives the possibility of explicitly stating the timing by means of directives.
The format of this directive is:

$$try\ \ Goal\ \ [Time\_Interval]\ \ [Frequency]\ \ [until\, Condition]$$

Below in an example of a goal that should be tried often.

$$\begin{aligned} \textit{too\_high(Temperature)} &\ \texttt{:-}\ \textit{threshold(Th),}\ \textit{Temperature} > \textit{Th.} \\ \textit{too\_high(Temperature)I} &\ \texttt{:>}\ \textit{start\_emergencyA, alert\_operatorA.} \end{aligned}$$

Like external events, internal events are recorded as past events, and are kept according to user indication. Internal events can be also used for triggering belief revision activities, so as to take past events into consideration, and decide whether to take them, cancel them, or incorporate them as knowledge of the agent. A declarative approach to belief revision based on meta-knowledge about the information, that in our opinion might be well integrated in DALI is that of [4], [5]. However, presently we do not commit to any specific belief revision strategy. We are planning to provide a distinguished predicate *incorporate* that might be either explicitly invoked, or, optionally, treated as an internal event. The user should be in charge of providing a suitable definition for this predicate, according to her own favorite approach to belief revision. When attempted, *incorporate* might return lists of facts or rules to be either added or removed by the DALI interpreter.

6 Procedural Semantics

Procedural semantics of DALI consists of an extension to SLD–resolution. DALI resolution is described in detail in [10]. Its basic features are summarized below.

We assume to associate the following sets to the goal which is being processed by a DALI interpreter: (i) the set $EV \subseteq E_{Ag}$ of the external events that are available to the agent (stimuli to which the agent can possibly respond); (ii) the set $IV \subseteq I_{Ag}$ of subgoals corresponding to internal events which have been

proved up to now (internal stimuli to which the agent can possibly respond); (iii) the set $PV \subseteq EP_{Ag}$ of past events (both internal and external); (iv) the set EVT of goals corresponding to internal events, to be tried from time to time.

A goal in DALI is a *disjunction* $G^1; G^2; \ldots; G^n$ of *component goals*. Every G^k is a goal as usually defined in the Horn–clause language, i.e. a conjunction. The meaning is that the computation fails only if all disjuncts fail.

The procedural behavior of a DALI agent consists of the interleaving of the following steps. (i) Trying to answer a user's query like in plain Horn–clause language. (ii) Responding to either external or internal events. This means, the interpreter picks up either an external event from EV or an internal event form IV, and adds this event G^{ev} as a new query, i.e. as a new disjunct in the present goal. Thus, goal $G^1; G^2; \ldots; G^n$ becomes $G^1; G^2; \ldots; G^n; G^{ev}$, and G^{ev} is inserted into PV. (iii) Trying to prove a goal corresponding to an internal event. the interpreter picks up an atom from EVT, and adds this atom G^{evt} as a new query, i.e. as a new disjunct in the present goal. Thus, goal $G^1; G^2; \ldots; G^n$ becomes $G^1; G^2; \ldots; G^n; G^{evt}$.

The interleaving among these activities is specified in the basic cycle of the interpreter. As mentioned before, the user can influence the behavior of the interpreter by means of suitable directives included in an initialization file.

7 Evolutionary Semantics

We define the semantics of a given DALI program P starting from the standard declarative semantics (Least Herbrand Model [1]) of a modified program P_s, obtained from P by means of syntactic transformations that specify how the different classes of events are coped with. P_s is the basis for the evolutionary semantics, that describes how the agent is affected by actual arrival of events.

For coping with external events, we have to specify that a reactive rule is allowed to be applied only if the corresponding event has happened. We assume that, as soon as an event has happened, it is recorded as a unit clause (this assumption will be formally assessed later). Then, we reach our aim by adding, for each event atom $p(Args)E$, the event atom itself in the body of its own reactive rule. The meaning is that this rule can be applied by the immediate-consequence operator only if $p(Args)E$ is available as a fact. Precisely, we transform each reactive rule for external events:

$$p(Args)E \ :> \ R_1, \ldots, R_q.$$

into the standard rule:

$$p(Args)E \ :- \ p(Args)E, R_1, \ldots, R_q.$$

Similarly, we have to specify that the reactive rule corresponding to an internal event $q(Args)I$ is allowed to be applied only if the subgoal $q(Args)$ has been proved. To this aim, we transform each reactive rule for internal events:

[1] The Least Herbrand Model is obtained as usual for the Horn-Clause language, i.e., by means of the immediate-consequence operator, iterated bottom-up.

$q(Args)I \ :> \ R_1, \ldots, R_q.$

into the standard rule:

$q(Args)I \ :- \ q(Args), R_1, \ldots, R_q.$

Now, we have to declaratively model actions, without or with an action rule. Procedurally, an action A is performed by the agent as soon as A is executed as a subgoal in a rule of the form

$B \ :- \ D_1, \ldots, D_h, A_1, \ldots, A_k. \quad h \geq 1, k \geq 1$

where the A_i's are actions and $A \in \{A_1, \ldots, A_k\}$. Declaratively, whenever the conditions $D_1, \ldots, D_h$ of the above rule are true, the action atoms should become true as well (given their preconditions, if any), so as the rule can be applied by the immediate-consequence operator. To this aim, for every action atom A, with action rule

$A \ :- \ C_1, \ldots, C_s. \quad s \geq 1$

we modify this rule into:

$A \ :- \ D_1, \ldots, D_h, C_1, \ldots, C_s.$

If A has no defining clause, we instead add clause:

$A \ :- \ D_1, \ldots, D_h.$

We need to specify the agent *evolution* according to the events that happen. We propose here an approach where the program P_s is actually affected by the events, by means of subsequent syntactic transformations. The declarative semantics of agent program P at a certain stage then coincides with the declarative semantics of the version of P_s at that stage.

Initially, many of the rules of P_s are not applicable, since no external and present events are available, and no past events are recorded. Later on, as soon as external events arrive, the reactive behavior of the agent is put at work, according to the order in which the events are received.

In order to obtain the *evolutionary* declarative semantics of P, as a first step we explicitly associate to P_s the list of the events that we assume to have arrived up to a certain point, in the order in which they are supposed to have been received. We let $P_0 = \langle P_s, [] \rangle$ to indicate that initially no event has happened.

Later on, we have $P_n = \langle Prog_n, Event\_list_n \rangle$, where $Event\_list_n$ is the list of the n events that have happened, and $Prog_n$ is the current program, that has been obtained from P_s step by step by means of a *transition function* Σ. In particular, Σ specifies that, at the n-th step, the current external event E_n (the first one in the event list) is added to the program as a fact. E_n is also added as a present event. Instead, the previous event E_{n-1} is removed as an external and present event, and is added as a past event.

Precisely, the *program snapshot* at step n is $P_n = \Sigma(P_{n-1}, E_n)$ where

Definition 1. *The* transition function Σ *is defined as follows.*

$\Sigma(P_{n-1}, E_n) = \langle \Sigma_P(P_{n-1}, E_n), [E_n | Event\_list_{n-1}] \rangle$

where

$\Sigma_P(P_0, E_1) = \Sigma_P(\langle P_s, []\rangle, E_1) = P_s \cup E_1 \cup E_{1N}$
$\Sigma_P(\langle Prog_{n-1}, [E_{n-1}|T]\rangle, E_n) =$
$\{\{Prog_{n-1} \cup E_n \cup E_{nN} \cup E_{n-1PE}\} \setminus E_{n-1N}\} \setminus E_{n-1}$

It is easy to extend Σ_P so as to remove the past events that have expired according to the conditions specified in *keep* directives.

Definition 2. *Let P_s be a DALI program, and $L = [E_n, \ldots, E_1]$ be a list of events. Let $P_0 = \langle P_s, []\rangle$ and $P_i = \Sigma(P_{i-1}, E_i)$ (we say that event E_i* determines *the transition from P_{i-1} to P_i). The list $\mathcal{P}(P_s, L) = [P_0, \ldots, P_n]$ is the* program evolution *of P_s with respect to L.*

Notice that $P_i = \langle Prog_i, [E_i, \ldots, E_1]\rangle$, where $Prog_i$ is the program as it has been transformed after the ith application of Σ.

Definition 3. *Let P_s be a DALI program, L be a list of events, and PL be the* program evolution *of P_s with respect to L. Let M_i be the Least Herbrand Model of $Prog_i$. Then, the sequence $\mathcal{M}(P_s, L) = [M_0, \ldots, M_n]$ is the* model evolution *of P_s with respect to L, and M_i the* instant model at step i .

The evolutionary semantics of an agent represents the history of the events received by the agents, and of the effect they have produced on it, without introducing a concept of a "state".

Definition 4. *Let P_s be a DALI program, L be a list of events. The* evolutionary semantics $\mathcal{E}_{P_s}$ *of P_s with respect to L is the couple $\langle \mathcal{P}(P_s, L), \mathcal{M}(P_s, L)\rangle$.*

It is easy to see that, given event list $[E_n, \ldots, E_1]$, DALI resolution simulates standard SLD-Resolution on $Prog_n$.

Theorem 1. *Let P_s be a DALI program, $L = [E_n, \ldots, E_1]$ be a list of events and P_n be the program snapshot at step n. DALI resolution is correct and complete with respect of P_n.*

The evolutionary semantics allows standard model checking techniques to be employed for verifying several interesting properties. By reasoning on the program evolution, it is possible for instance to know whether a certain event will be at a certain stage in the memory of the agent. By reasoning on the model evolution, it will be possible for instance to know whether a certain action A has been performed at stage k, which means that A belongs to M_k.

The evolutionary semantics can be extended to DALI multi-agent programs, by considering the evolutionary semantics of all agents involved. A requirement is that the sequence of incoming events for each agent must be somehow specified.

8 Related Work and Concluding Remarks

The main objective in the design of DALI has been that of understanding whether modelling agents in logic programming was possible, and to which extent. We wanted this new language to be as simple and easy to understand as

the Horn clause language: therefore, both syntax and semantics are very close to the Horn clause language, and so is the procedural semantics. Experiments in practical applications will tell to which extent such a simple language can be satisfactory. Clearly, in order to develop real–world examples, DALI can be equipped with an agent communication language, and with primitives for coordination and cooperation among agents. Moreover, past experience of the authors in the field of meta-logic programming has suggested to use a meta-programming approach for coping with important aspects such as the ontology problem, and management of incomplete information. For lack of space we cannot further comment on these aspects here: a preliminary discussion can be found in [9]. However, the examples that the reader will find on the DALI web page show that nice multi-agent applications can be obtained with the simple features of DALI.

Presently, DALI does not includes specific features for planning. Our view is that the planning features should constitute a separate module that an agent can invoke on specific goal(s), so as to obtain possible plans and choose among them. In this direction, we mean to integrate an Answer Set Solver ([21] and references therein) into the implementation.

DALI is meant to be a logic general-purpose programming language like Prolog, aimed at programming agents. DALI does not commit to any specific agent architecture, and also, as mentioned above, does not commit to any planning formalism. Then, DALI does not directly compare (at least for the moment) with approaches that combine logic and imperative features, and that are mainly aimed at planning. Two important such approaches are ConGolog [11], which is a multi-agent Prolog-like language with imperative features based on situation calculus, and 3APL [18], [17], which is rule-based, planning-oriented, and has no concept of event. Also, a comparison with very extensive approaches for Multi-Agent-Systems like IMPACT [26] is premature, since IMPACT is not just a language, but proposes a complex agent architecture.

A purely logic language for agents is METATEM [16] [22], where different agents are logic programs which are executed asinchronously, and communicate via message-passing. METATEM has a concept of time, and what happened in the past determines what the agent will do in the future. Differently from DALI, METATEM agents are purely reactive, and there are no different classes of events.

In the BDI (Belief-Desire-Intention) approach [25] [24], agents are systems that are situated in a changing environment, receive perceptual input, and take actions to affect their environment, based on their internal mental state. Implementations of BDI agents are being used successfully in real application domains. An experiment that we want to make is to use DALI as an implementation language for the BDI approach. This experiment follows the example of AgentSpeak(L) [23], a purely reactive logic language with external events and actions, meant to (indirectly) model BDI features in a simple way. The internal state of an AgentSpeak agent constitutes its beliefs, the goals its desires, and plans for achieving goals its intentions. External events are interpreted as goals,

which are pursued in some order (according to a selection function) by means of plans (selected by another special function). A plan can include new goals that, when encountered during the execution of the plan, become the highest in priority, i.e. the first ones that will be attempted. These goals are called internal events. However, apart from the name, DALI internal events are aimed at triggering proactive behavior, while AgentSpeak internal events are aimed at refining plans.

In conclusion, we claim that some of the features of DALI are really novel in the field: in particular, the different classes of events (especially the internal and present events), their interaction, the interleaving of different activities, and the use of past events and past actions.

References

1. J. J. Alferes, P. DellAcqua, E. Lamma, J. A. Leite, L. M. Pereira, and F. Riguzzi. A logic based approach to multi-agent systems. *ALP Newsletter*, 14(3), August 2001.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Logic Programming*, 45(1):43–70, September/October 2000.
3. J. Barklund, K. Boberg, P. Dell'Acqua, and M. Veanes. Meta-programming with theory systems. In K.R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, pages 195–224. The MIT Press, Cambridge, Mass., 1995.
4. G. Brewka. Declarative representation of revision strategies. In C. Baral and M. Truszczynski, editors, *NMR'2000, Proc. Of the 8th Intl. Workshop on Non-Monotonic Reasoning*, 2000.
5. G. Brewka and T Eiter. Prioritizing default logic. In *Festschrift 60th Anniversary of W. Bibel*. Kluwer Academic Publishers, 2000.
6. P. R. Cohen and H. J. Levesque. Rational interaction as the basis for communication. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 221–256. MIT Press, 1990.
7. P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In V. Lesser, editor, *Proc. 1st Intl. Conf. on Multi-agent Systems*, AAAI Press, pages 65–72. MIT Press, 1995.
8. S. Costantini. Meta-reasoning: a survey. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future: Special volume in honour of Bob Kowalski (in print)*. Springer-Verlag, Berlin. invited paper.
9. S. Costantini. Meta-reasoning: a survey. In *Computational Logic: From Logic Programming into the Future –Special volume in honour of Bob Kowalski(to appear)*. Springer-Verlag. invited paper.
10. 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, Paris, France, September 1999. http://www.di.unipi.it/ brogi/ ResearchActivity/COCL99/ proceedings/index.html.

11. G. De Giacomo, Y. Lespérance, and Levesque. H. J. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, (121):109–169, 2000.
12. P. Dell'Acqua and L. M. Pereira. Updating agents. In *Procs. of the ICLP99 Workshop on Multi-Agent Systems in Logic (MASL99)*, Las Cruces, New Mexico, 1999.
13. P. Dell'Acqua, F. Sadri, and F. Toni. Combining introspection and communication with rationality and reactivity in agents. In J. Dix, F.L. Del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence*, LNCS 1489, Berlin, 1998. Springer-Verlag.
14. P. Dell'Acqua, F. Sadri, and F. Toni. Communicating agents. In *Proc. International Workshop on Multi-Agent Systems in Logic Programming, in conjunction with ICLP'99*, Las Cruces, New Mexico, 1999.
15. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proc. 17th Intl. Joint Conf. on Artificial Intelligence, IJCAI 2001*, Seattle, Washington, USA, 2001. Morgan Kaufmann. ISBN 1-55860-777-3.
16. M. Fisher. A survey of concurrent METATEM – the language and its applications. In *Proceedings of First International Conference on Temporal Logic (ICTL)*, LNCS 827, Berlin, 1994. Springer Verlag.
17. K. Hindriks, F. de Boer, W. van der Hoek, and J. J. Meyer. A formal architecture for the 3apl programming language. In *Proceedings of the First International Conference of B and Z Users*, Berlin, 2000. Springer Verlag.
18. K. V. Hindriks, F. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
19. R. A. Kowalski and F. Sadri. From logic programming to multi-agent systems. In *Annals of Mathematics and Artificial Intelligence*. (to appear).
20. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic in Databases*, LNCS 1154, Berlin, 1996. Springer-Verlag.
21. V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Proc. of ICLP '99 Conference*, pages 23–37, Cambridge, Ma, 1999. MIT Press. Invited talk.
22. M. Mulder, J. Treur, and M. Fisher. Agent modelling in concurrent METATEM and DESIRE. In *Intelligent Agents IV*, LNAI, Berlin, 1998. Springer Verlag.
23. A. S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In W. Van De Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, LNAI, pages 42–55, Berlin, 1996. Springer Verlag.
24. A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.
25. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.
26. V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents*. MIT-Press, 2000. 580 pages.
27. M. Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–32, 2000.