About declarative semantics of logic-based agent languages

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. In this paper we cope with providing an approach to declarative semantics of logic-based agent-oriented languages, taking then as a case-study the language DALI which has been previously defined by the authors. This "evolutionary semantics" does not resort to a concept of state: rather, it models reception of events as program transformation steps, that produce a "program evolution" and a corresponding "semantic evolution". Communication among agents and multi-agent systems is also taken into account. The aim is that of modeling agent's evolution according to either external (environmental) or internal changes in a logical way, thus allowing in principle the adoption of formal verification methods. We also intend to create a common ground for relating and comparing different approaches/languages.

1 Introduction

The original perspective on agents in Computational Logic focused on agent's reasoning process, thus identifying "intelligence" with rationality, while neglecting the interactions with the environment. The identification of intelligence with rationality has been heavily criticized, even adopting the opposite point of view, i.e., that intelligent behavior should result solely from the ability of an agent to react appropriately to changes in its environment.

A novel view of logical agents, able to be both rational and reactive, i.e., capable of timely response to external events, that has been introduced by Kowalski and Sadri in [15] [16]. A meta-logic program defines the "observe-think-act" cycle of an agent. Integrity constraints are used to generate actions in response to updates from the environment. After that, both the notion of agency and its interpretation in computational logic has evolved, and many interesting approaches and languages have been proposed in the last two decades [23].

A foundational approach in artificial intelligence and cognitive science is BDI, introduced in [4] that stands for "Belief, Desire, Intentions": and agent's *beliefs* correspond to information the agent has about the world, which may be incomplete and incorrect; an agent's *desires* intuitively correspond to its objectives, or to the tasks allocated to it; as an agent will not, in general, be able to achieve all its desires, the desires upon which the agent commits are *intentions* that the agent will try to achieve. The

theoretical foundations of the BDI model have been investigated (the reader may refer to [19]). The original formal definition was in terms of modal logics. However, there have been reformulations of the BDI approach that have led to the logic programming language AgentSpeak(L) and to the programming language *3APL* [11].

Among agent-oriented languages based on logic programming are the following. Go! [5] is a multi-paradigm programming language with a strong logic programming aspect. Go! has strong typing, and higher-order functional aspects. Its imperative subset includes action procedure definitions and rich program structuring mechanisms. Go! agents can have internal concurrency, can deductively query their state components, can communicate with other Go! agents using application specific symbolic messages. DALI [6] [7] is syntactically very similar to traditional logic programming languages such as prolog. The reactive and proactive behavior of a DALI agent is triggered by several kinds of events: external events, internal, present and past events. Events are coped with by means of a new kind of rules, *reactive rules*. DALI provides a filter cn communication for both incoming and out-coming message. In fact, a message is accepted (or otherwise discarded) only if it passes the check of the communication filter. This filter is expressed by means of meta-rules specifying two distinguished predicates.

Interesting agent architectures are logic-based, like KGP [3], which builds on the original work of Kowalski and Sadri, where various forms of reasoning can be gracefully specified. IMPACT [2] provides interoperability by accommodating into a computational logic shell various others kinds of agents.

The semantics of the above-mentioned languages and approaches has been defined in various ways. All of them have suitable operational models that account for agent's behavior. Many of them enjoy, in the tradition of logic, a logic declarative semantics. It is however not so easy to find a common ground for relating and comparing the different approaches. Aim of this paper is to introduce an approach to declarative semantics of logical agent-oriented languages that considers evolution of agents, without introducing explicitly a concept of state. Rather, changes either external (i.e., reception of exogenous events) or internal (i.e., courses of actions undertaken based on internal conditions) are considered as making a change in the agent program, which is a logical theory, and in its semantics (however defined). For such a change to be represented, we understand this change as the application of a program-transformation function. Thus, agent evolution is seen as program evolution, and semantic evolution. This novel approach will perhaps not encompass all the existing ones, but, in our opinion, it can constitute a starting point for establishing a common viewpoint.

In Section 2 we review the features that intelligent logical agents in our opinion possess. In Section 3 we introduce the approach. In Sections 5 and 6, as a case-study, we show in detail the approach with respect to the DALI language, that for the sake of clarity we shortly review in Section 4. Finally, we conclude in Section 7.

2 Features of Evolving Logical Agents

A great deal can be said about features that agents in general and logical agents in particular should possess (for a review the reader may refer for instance to [20], for a

discussion to [14]). It is widely recognized however that agents, whatever the language and the approach on which they are based, should exhibit the following features.

- *Autonomy:* agents should be able to operate without the direct intervention of humans.
- *Reactivity:* agents should perceive their environment (which may be the physical world, a user, a collection of agents, the internet,etc.) and respond in a timely fashion to changes that occur in it.
- *Pro-activeness:* agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where it is appropriate.
- *Social ability:* i. e., agents should be able to interact, when they deem it appropriate, with other agents or with humans in order to complete their own problem solving and to help others with their activities.

In order to exhibit these properties, logical agents usually rely upon some form of rationality, which might be based on some or all the following internal features:

- *Perceptive abilities*, i.e. the possibility of be aware (to same extent) of what is going on (what *events* happen) in the environment.
- *Inferential abilities*, which may include many different forms of either formal or commonsense ways of drawing consequences from what is known.
- *Introspective abilities* which imply some kind of control over their actions and internal state, for being able to affect their own functioning according to priorities, desires (or goals), time or resource bounds, etc. This point may include a concept of time.
- *Meta-level control* for establishing goals and selecting intentions.
- *Learning abilities* which at the lowest level include the memory of what has happened so as to be able influence the future course of actions depending on the past, and then may include more or less sophisticated forms of belief revision or classification.
- *Communication control* in the sense that it should be possible to decide which social interactions are appropriate and which not, both in terms of generating appropriate requests and of judging incoming requests.

These abilities must be operationally combined in the context of either a cycle, or a multi-threaded execution, or an interleaving, so as to be able to timely respond to the environment changes based on reasoning, planning, learning, etc. Effective operational models have been proposed, based either on versions of the observe-think-act cycle introduced by [15], or on operational semantics of various kinds.

For logical agents, it is in our opinion important to give a declarative account of the agent behavior. This would have at least two advantages:

(i) understanding the agent behavior more easily than in an operational model;
(ii) being able to verify properties of both agents and multi-agent systems.

The difficulty is that agents evolve, according to both changes in the environment and to their internal state. Traditionally, logic includes neither the notion of state nor that of evolution. It includes the notion of a theory with some kind of model(s). In this paper we propose a semantic approach that accounts for evolution, though not introducing state explicitly.

3 Declarative Semantics of Evolving Agents

The evolutionary semantics that we propose is aimed at declaratively modeling the changes inside an agent which are determined both by changes in the environment and by the agent's own self-modifications. The key idea is to understand these changes as the result of the application of program-transformation functions. In this approach, a program-transformation function is applied upon reception of either an external or an internal event, the latter having a possibly different meaning in different formalisms.

As a typical situation, perception of an external event will have an effect on the program which represent the agent: for instance, the event will be stored as a new fact in the program . This transforms the program into a new program, that will procedurally behave differently than before, e.g., by possibly reacting to the event. Or, the internal event corresponding to the decision of the agent to undertake an activity triggers a more complex program transformation, resulting in version of the program where the corresponding *intention* is somewhat "loaded" so as to become executable.

Then, in general one will have an initial program P_0 which, according to these program-transformation steps (each one transforming P_i into P_{i+1}), gives rise to a Program Evolution Sequence $PE = [P_0, ..., P_n]$. The program evolution sequence will have a corresponding Semantic Evolution Sequence $[M_0, ..., M_n]$ where M_i is the semantic account of P_i.

The different languages and different formalisms will influence the following key points:

1. When a transition from P_i to P_{i+1} takes place, i.e. which are the external and internal factors that determine a change in the agent.
2. Which kind of transformations are performed.
3. Which semantic approach is adopted, i.e., how M_i is obtained from P_i. M_i might be for instance a model, or an initial algebra, or a set of Answer Sets if the given language is based on Answer Set Programming (that comes from the stable model semantics of [13]. In general, given a semantics $\mathcal{S}$ we will have $M_i = \mathcal{S}(P_i)$

A particular internal event that may determine a transition can be the decision of the agent to revise its knowledge, for instance by verifying constraints, removing "old" facts, or performing any kind of belief revision. Also belief revision in fact can be seen in our approach as a step of program transformation that in this case results in the updated theory.

We also believe it useful to perform an *Initialization step*, where the program P_{Ag} written by the programmer is transformed into a corresponding program P_0 by means

of some sort of knowledge compilation. This initialization step can be understood as a rewriting of the program in an intermediate language and/or as the loading of a "virtual machine" that supports language features. This stage can on one extreme do nothing, on the other extreme it can perform complex transformations by producing "code" that implements language features in the underlying logical formalism. P_0 can be simply a program (logical theory) or can have additional information associated to it.

In Multi-agent systems (MAS), where new pieces of knowledge (beliefs, but possibly also rules or sets of rules) can be received by other agents, an agent might have to decide whether to accept or reject the new knowledge, possibly after having checked its correctness/usefulness. This can imply a further knowledge compilation step, to be performed:

(ii) Upon reception of new knowledge.
(ii) In consequence to the decision to accept/reject the new knowledge.

To summarize, we will have in principle at least the following program-transformation functions: Γ_{init} for initialization, Γ_{events} for managing event reception, Γ_{revise} for belief revision, Γ_{learn} for incorporating new knowledge. Then, given an agent program P_{Ag} we will have:

$$P_0 = \Gamma_{init}(P_{Ag})$$

and

$$P_{i+i} = \Gamma_{op}(P_{Ag})$$

with $op \in \{learn, revise, events\}$.

The evolutionary semantics of an agent represents the history of an agent without introducing a concept of a "state".

Definition 1 (Evolutionary semantics). *Let P_{Ag} be an agent program. The evolutionary semantics $\varepsilon_{P_{Ag}}$ of P_{Ag} is the couple*
$\langle PE, ME \rangle$.

In order to illustrate the approach on a case-study, in the rest of the paper we will discuss the evolutionary semantics of the DALI language. With respect to previous discussions [7] that considered external events only, we here account for all kinds of events and also consider the DALI communication architecture.

4 DALI in a Nutshell

DALI [7] [8] [20] is an active agent-oriented Logic Programming language designed in the line of [14] for executable specification of logical agents. The Horn-clause language is a subset of DALI. The reactive and proactive behavior of the DALI agent is triggered by several kinds of events: external events, internal, present and past events. All the events and actions are time-stamped, so as to record when they occurred.

A DALI program with its interpreter give rise to an agent since, when activated, it stays "awake" and monitors the arrival of events from the external world. These events are treated similarly to user's queries in the sense that they trigger an inference activity that in this case can be considered a reaction. At a certain frequency and/or upon certain conditions, the interpreter tries on its own initiative to prove certain distinguished goals. If they succeed, their success is interpreted as the reception of an event, thus triggering further inference. This is the mechanism of "internal events" is an absolute novelty of DALI (other languages such as 3-APL language have internal events, but with a different meaning).

Internal events make DALI agents proactive, and make them exhibit a behavior that depends on the logic programs, but also on the history of the interaction of the agent with its external environment. In fact, all the external and internal events and the actions that the agent performs are recorded as "past events", and having past events in the conditions of distinguished goals will influence the internal event to happen or not. We will now shortly present the language in a more formal way.

An external event is a stimulus perceived by the agent from the environment. We define the set of external events perceived by the agent from time t_1 to time t_n as a set $E = \{e_1 : t_1, ..., e_n : t_n\}$ where $E \subseteq S$, and S is the set of the external stimuli that the agent can possibly perceive.

A single external event e_i is an atom indicated with a particular postfix in order to be distinguished from other DALI language events.

Definition 2 (External Event). *An external event is syntactically indicated by postfix E and it is defined as:* $ExtEvent ::= << Atom_E >> | seq << Atom_E >>$
withe the usual definition of atoms and terms.

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.

Definition 3 (Reactive rule). *A reactive rule has the form:*

$ExtEvent_E :> Body$ *or*

$ExtEvent_{1E}, ..., ExtEvent_{nE} :> Body$

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. Before the event is reacted to, the agent has the possibility of reasoning about it. Then, each external event $Atom_E$ has a counterpart called "present event" that may occur in the body of rules with suffix *N*. In particular, the present event $Atom_N$ is true as far as the external event $Atom_E$ is still in EV.

Internal events make a DALI agent proactive independently of the environment, of the user and of the other agents, and also allow the agent to manipulate and revise its knowledge.

Definition 4 (Internal Event). *An internal event is syntactically indicated by postfix I:*

$InternalEvent ::=<< Atom_I >>$

The internal event mechanism implies the definition 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:
$IntEvent : -Conditions$
$IntEvent_I :> Body$

Internal events are automatically attempted with a default frequency customizable by means of directives in the initialization file. A DALI agent is able to build a plan in order to reach an objective, by using internal events of a particular kind, called *planning goals*.

After reaction to either an external or an internal event, the agent remembers to have reacted by converting the external event into a *past event*, postfix P (time-stamped).

Actions are the agent's way of affecting the environment, possibly in reaction to either an external or internal event. An action in DALI can be also a message sent by an agent to another one.

Definition 5 (Action). *An action is syntactically indicated by postfix A:*

$Action ::=<< Atom_A >> | message_A << Atom, Atom >>$

Actions occur in the body of rules.

In DALI, actions 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 following syntax:

Definition 6 (Action rule). *An action rule has the form:*
$Action :< Preconditions$
where $Preconditions ::= seq << Object >>$ *and*
$Object ::=<< PastEvent_P >> | << Atom >> | << Belief >> |...$

Similarly to external and internal events, actions are recorded as past actions.

Procedurally, DALI is based on an Extended Resolution Procedure that interleaves different activities, and can be tuned by the user via directives. The operational semantics of DALI is based on Dialogue Games Theory [9] [20]: the DALI Interpreter is modeled as a set of cooperating players.

4.1 DALI Communication Architecture

DALI supports inter-agent interaction and cooperation by providing a flexible communication architecture. The architecture [9] [10] provides: a FIPA-compliant (i.e., a standard [12]) communication protocol; a filter on communication, i.e. a set of rules that decide whether or not to receive or send a message; a meta-reasoning layer, where meta-rules can be specified to help the agent to understand message contents.

The DALI communication filter is defined by means of meta-level rules defining the distinguished predicates *tell* and *told*. Actually, the FIPA/DALI communication protocol itself is implemented by means a piece of DALI code consisting of default *tell/told*

rules. The same agent program, if equipped with a different filter, results in a different agent with different communication behavior.

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 any condition. 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.

Symmetrically, the messages that an agent means to send are subjected to a check via *tell* rules. 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 if the goal *tell(Receiver,Sender,primitive(Content))* succeeds.

5 Declarative Semantics of basic DALI programs

In this section we describe the declarative semantics of the basic DALI language according to the proposed approach. In the next section we will take care of the communication architecture.

Let us first consider the initialization step. As a design choice, a DALI program is transformed into a corresponding Horn-clause program. In this way, we keep all the useful properties of the Horn-clause language and we are still able to exploit all the technical machinery related to it. Then, the semantics $\mathcal{S}$ that we adopt is either the least Herbrand model [17] (for definite programs) or the well-founded model [22] (for programs with negation-as-failure). We give on the one hand the intuition that underlies the transformation for the various language elements, and on the other hand a formal description of Γ_{init}^{DALI}.

Reactive rules can be declaratively modeled by making some considerations on Horn clause language. Consider the plain Horn-clause language, and the following program:

$p.$
$p : -q.$
$q.$

Its least Herbrand model is $\{q, p\}$, like in the following slightly modified version:

$p.$
$p : -p, q.$
$q.$

Since *p* is true by means of a unit clause, the second rule for *p* does not change the meaning of the program, since it differs from the previous version only in that there is *p*

itself in the body. This is exactly the "trick" that we will use for our reactive Horn-clause programs.

Precisely, 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 the reactive rule can be applied by the immediate-consequence operator only if $p(Args)_E$ is available as a fact.

Definition 7 (Transformation of external events rules Γ_{er}). *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 transform the reactive rule corresponding to each internal event.

Definition 8 (Transformation of internal events rules Γ_{ir}). *For internal events, $q(Args)_I$ is allowed to be applied only if the subgoal $q(Args)$ has been proved. For this aim, we transform each reactive rule for internal events:*

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

into the standard rule:

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

Now, we have to declaratively model actions, with or without an action rule. The point is, an action atom should become true (given its preconditions, if any) whenever the action is actually performed in some rule. Consider another simple program written in the plain Horn-clause language:

$p.$
$p : -b, a.$
$b.$

Its least Herbrand model is $\{p, b\}$, since both *p* and *b* are given as facts. If we modify the program as follows:

$p.$
$p : -b, a.$
$b.$
$a : -p, b.$

its least model is $\{p, b, a\}$. Assuming that *p* is an event atom and *a* is an action atom with no defining clause, this modification ensures that the action atom *a* becomes true whenever the action is actually performed.

Similarly, let us assume that *a* has no defining clause, like in the program:

p.
$p : -b, a$.
b.
$a : -c$.
c.

Its least Herbrand model is $\{p, b, a\}$, since *p, b* and *c* are given as facts. We modify the program as follows:

p.
$p : -b, a$.
b.
$a : -c, p, b$.

Its least model is still $\{p, b, a\}$, but, interpreting *a* as an action atom, we state that *a* can be derived only if the corresponding action is actually performed in the rule defining *p*.

More formally, an action A is performed by a DALI agent whenever A is executed as a subgoal in a rule of the form

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

where the A_is 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 that the rule can be applied by the immediate-consequence operator. To model this behavior we introduce the following:

Definition 9 (Transformation of action rules Γ_{ar}). *For every action atom A, with action rule:*

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

we modify this rule into:

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

If A has no defining rule, we instead add the clause: $A : -D_1, \ldots, D_h$.

Definition 10. *The program-transformation initialization function Γ_{init}^{DALI} is defined as $\Gamma_{ir} \bigcup \Gamma_{ir} \bigcup \Gamma_{ir}$. Given a DALI logic program P_{Ag}, let $P_s = \Gamma_{init}^{DALI}(P_{Ag})$ be the horn-clause logic program obtained by applying the above transformations.*

P_s is the basis for the evolutionary semantics, that describes how the agent is affected by actual arrival of events. In fact, we need now to specify the agent evolution according to the events that happen. According to the proposed approach, the program P_s is actually affected by the events by means of subsequent syntactic transformations. When the agent receives an external event, we ideally stop the evolution and calculate the least Herbrand model, thus generating a 'snapshot' of the agent change process. Then, the evolution restarts and the process goes on until the reception of an incoming event. The declarative semantics of agent program P_{Ag} 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 and internal events happen, the reactive behavior of the agent is put at work.

In order to obtain the evolutionary declarative semantics of P, as a first step we explicitly associate to P_s the list of the external events that we assume to have arrived up to a certain point, in the order in which they are supposed to have been received and the list of internal events that have became true. In this context, we make the simplifying assumption that all internal events are attempted at the same default frequency. Precisely, we assume that they are attempted whenever a new external event arrives. We also assume that past events are never removed.

Let $EXTH$ be the set of all ground instances of atoms corresponding to external events which occur in the DALI logic program. Similarly, let $INTH$ the set of ground instances of internal events. Finally, let $ACTH$ be the set of ground instances of action atoms. Given the above sets, at each step, say n, we can consider the subsets: $EXTH_n$ of the external events perceived up to that step; $INTH_n$ of the internal events that have succeeded up to that step; and $ACTH_n$ of the actions performed up to that step. We consider these sets as lists, that reflect the order in which external events/internal events/actions respectively happened/were proved/were performed.

We let $P_0 = \langle P_s, [\,], [\,] \rangle$, to indicate that initially no event has happened. Later on, the program P_s will be modified by the perceived external events, the triggered internal events and finally by the performed actions. Namely, let $P_n = \langle Prog_n, EXTH_n, INTH_n \rangle$ be the result of n steps, where $Prog_n$ is the current program that has been obtained from P_s step by step by means of the program-transformation *transition function* Γ^{DALI}_{events}. We also let $\mathcal{S}(P_n)$ be the least Herbrand model of $Prog_n$.

In particular, Γ^{DALI}_{events} specifies that, at the $n-th$ step, the current external event E_n, the triggered internal events and the performed actions are added to the program as facts. E_n is also added as a present event. The immediate-consequence operator will consequently be able both to apply the reactive rule related to E_n, and the rules in whose bodies the corresponding present event occurs. Previous step external/present event E_{n-1} is removed and is added as a past event. Concerning internal events, all the internal events/goals that have been proved up to this step are added as facts, thus enabling the corresponding reactive rules.

The program-transformation transition function Γ^{DALI}_{events} is related to an external event, an internal event and an action all together. In case any of them should be missing, the corresponding part of the definition would just not apply.

Definition 11 (Transition function). *Let E_{jE} be an external event, I^k_{lI} an internal event and A^m_{nA} an action. The program-transformation transition function Γ^{DALI}_{events} is defined as follows:*

$\Gamma^{DALI}_{events}(P_{n-1}, E_{nE}, I^k_{nI}) = \langle \Gamma^{DALI}_{events\,P}(P_{n-1}, E_{nE}, I^k_{nI}), [E_{nE}|EXTH_{n-1}], [I^k_{nI}|INTH_{n-1}] \rangle$
where:

$\Gamma^{DALI}_{events}(P_0, E^k_1, I^k_1) = \Gamma^{DALI}_{events}(\langle P_s, [], [] \rangle, E_1, I^k_1) = P_s \cup E_{1E} \cup E_{1N} \cup I^k_1.$
$\Gamma^{DALI}_{events}(\langle Prog_{n-1}, [E_{n-1E}|T_e], [I^k_{n-1I}|T_i] \rangle, E_n, I^k_n) =$
$\{\{Prog_{n-1} \cup E_{nE} \cup E_{nN} \cup E_{n-1P} \cup I^k_{nI} \cup I^r_{n-1P} \cup A^s_{n-1P}\}\}$
We do that $\forall\ I^k_n \in INTH_n$, $\forall\ I^r_{n-1} \in INTH_{n-1}$ *and* $\forall\ A^s_{n-1} \in ACTH_n$.

Definition 12 (Program evolution). *Let P_s be a DALI program, $EXTH_n = [E_n, \ldots, E_1]$ be the list of external events and $INTH_n = [I_n, \ldots, I_1]$ the list of internal events. Let forall i, $Pi = \Gamma^{DALI}_{events}(P_{i-1}, E_i, I^k_i)$. The list $[P_0, \ldots, P_n]$ that we denote by $PE(Ps, EXTH_n, INTH_n)$ (or for short PE if we assume the arguments as given) is the* program evolution *of P_s with respect to $EXTH_n$ and $INTH_n$.*

Notice that $P_i = \langle Prog_i, [E_i, \ldots, E_1], [I^k_i, \ldots, I^l_1] \rangle$, where $Prog_i$ is the program as it has been transformed after the i-th application of Γ^{DALI}_{events}. If the program evolution is understood from when the agent is activated, then we will let $P_0 = \langle P_s, [\,], [\,] \rangle$. Otherwise, the program evolution can be restarted from a P_k which has been obtained from a previous evolution.

Definition 13 (Model Evolution). *Let P_s be a DALI program, $EXTH$ and $INTH$ be the lists of events, and $PE = [P_0, \ldots, P_n]$ be the program evolution of P_s with respect to $EXTH$ and $INTH$. Let M_i be the least Herbrand model of $Prog_i$. Then, the list $[M_0, \ldots, M_n]$ that we denote by $ME(P_s, EXTH_n, INTH_n)$ (or for short PE if we assume the arguments as given) is the* model evolution *of P_s with respect to PE, and M_i is the instant model at step i.*

Definition 14 (Evolutionary semantics). *Let P_s be a DALI program, $EXTH$ and $INTH$ be the lists of events. The evolutionary semantics ε_{Ps} of P_s with respect to $EXTH$ and $INTH$ is the couple*
$\langle PE, ME \rangle$.

6 Semantics of Communication

In this section we extend the declarative semantics of DALI so as to encompass the communication part. For doing so, we have to modify the initialization stage, i.e., we have to extend the program-transformation initialization function Γ^{DALI}_{init}. To this purpose, we build upon some of the author's previous work on meta-logic. In [1], a logical framework for called RCL (Reflective Computational Logic) is introduced, based on the concept of "Reflection Principle". Reflection principles are understood in RCL as logical schemata intended to capture the basic properties of a domain. The purpose of introducing reflection principles is to make it easier to build a complex theory, by allowing a basic theory to be enhanced by a compact meta-theory.

These schemata need however to be given a role in the theory, both semantically (thus obtaining a declarative semantics for the resulting theory) and procedurally (making them usable in deduction). To this aim, they are interpreted as procedures, more precisely as functions that transform rules into (sets of) rules. These new Horn clauses are called "reflection axioms". Then, the model-theoretic and fixed point semantics of the given program plus a reflection principle coincides with the corresponding semantics of the program obtained from the given one, by adding the reflection axioms.

Definition 15. *Let C be a definite clause. A* reflection principle $\mathcal{R}$ *is a mapping from rules to (finite) sets of rules. The rules in $\mathcal{R}(C)$ are called reflection axioms.*

Definition 16. *Let $\mathcal{R}$ be a reflection principle. Let $\mathcal{R}(P)$ be the set of reflection axioms obtained by applying $\mathcal{R}$ on all the clauses of P. Let $P' = P \cup \mathcal{R}(P)$ be the resulting program. Let $\Gamma_{\mathcal{R}}$ be a function that performs the transformation from P to P'.*

Reflection principles thus allow extensions to be made to a logic language like for instance the Horn-clause language leaving the underlying logic unchanged. Several reflection principles can be associated to a program. A potential drawback is that the resulting program $(P \cup \mathcal{R}(P))$ may have, in general, a large number of rules, which is allowed in principle but difficult to manage in practice. To avoid this problem, one can apply reflection principles in the inference process only as necessary, which means by computing the reflection axioms *on the fly* as needed. In [1] an extended resolution procedure with this behavior is defined.

Reflection principles can be expressed as axiom schemata in the form

$$new\_rules \Leftarrow given\_rule$$

The left-hand-side of $\Leftarrow$ denotes a (set of) rule(s) (possibly facts) which is produced by applying the given reflection principle to the program at hand. The right-hand-side is the starting rule (the one that actually occurs in the given program), plus possibly some conditions for the application of the correspondence.

For coping with the DALI communication architecture, it is then sufficient to augment Γ_{init}^{DALI} so as to apply suitable reflection principles. In particular, we add the following three, that for the sake of brevity we denote together by $\mathcal{R}_{Dcomm}$.

The first reflection principle takes a $told$ rule occurring in the DALI logic program, and, assuming a generic incoming message $message\_received(Ag, primitive(Content, Sender))$, generates the actual filter rule where the constraints are instantiated with the message elements $primitive$, $Content$, $Sender$. The second reflection principle generates an external event from every successful application of $told$. Notice that the second reflection principle acts on the actual $told$ rules generated by the first one. The last reflection principle transforms a successful application of a $tell$ rule into a message to be sent.

$$\begin{array}{l} told(Ag, primitive(Content)) : - \\ \qquad constraint_1, \ldots, constraint_n, \\ \qquad message\_received(Ag, primitive(Content, Sender)). \\ \Leftarrow told(Ag, primitive(Content)) : - constraint_1, \ldots, constraint_n. \end{array}$$

$$\begin{array}{l} primitive(Content)_E : - told(Ag, primitive(Content)). \\ \Leftarrow told(Ag, primitive(Content)) : - \\ \qquad\qquad constraint_1, \ldots, constraint_n, \\ \qquad\qquad message\_received(Ag, primitive(Content, Sender)). \end{array}$$

$$\begin{array}{l} message\_to\_be\_sent(To, Comm\_primitive(Content)) : - \\ \qquad tell(To, Comm\_primitive(Content)), constraint_1, \ldots, constraint_n \\ \Leftarrow tell(To, Comm\_primitive(Content)) : - constraint_1, \ldots, constraint_n. \end{array}$$

Let $\Gamma_{\mathcal{R}_{Dcomm}}$ be the function that augments a program P by applying $\mathcal{R}_{Dcomm}$. The new initialization stage is then performed by a program-transformation function

$\Gamma_{init}^{DALI_{comm}}$ that, beyond coping with event and action rules, also applies reflection principles related to communication.

Definition 17. *We define* $\Gamma_{init}^{DALI_{comm}}$ *as* $\Gamma_{init}^{DALI} \bigcup \Gamma_{\mathcal{R}_{Dcomm}}$.

6.1 Generalization

Reflection principles allow many kinds of language extensions to be modeled in uniform way. For instance, another DALI feature that can be modeled is the attempt of understanding message contents. I.e., there can be the case where a $Content$ passes the $told$ filter, but cannot be added as an external event because it is not understandable by the agent, as it does not occur in the head of any reactive rule. In this case, $Content$ is automatically submitted to a procedure $meta$, which is predefined but user-extensible, which tries (possibly for instance by using ontologies) to translate $Content$ into an equivalent though understandable form.

Actually, also the program transformations performed by Γ_{init}^{DALI} for reactive and action rules can be represented by means of reflection principles.

One may find it awkward the notion of the semantics of the given program being defined to be the semantics of the program after the transformations. Again by resorting to RCL, it is possible to clean up this notion, by introducing the concept of a *reflective model*.

Definition 18. *Let* I *be an interpretation of a program* P*. Then,* I reflectively satisfies P *(with respect to a (set of) reflection principle(s)* $\mathcal{R}$ *if and only if* I *satisfies* $P' = \Gamma_{\mathcal{R}}(P) = P \cup \mathcal{R}(P)$.

Definition 19. *Let* I *be an interpretation of a logic program* P*. Then,* I *is a* reflective model *of* P *if and only if* I *reflectively satisfies* P.

The model intersection property still holds, so there exists a least reflective Herbrand model of P. Then, in the evolutionary semantics we may let $\mathcal{S}$ be the least reflective Herbrand model of $Prog_n$.

7 Concluding Remarks

In this paper we have presented an approach to giving a declarative semantics to logical agents that evolve according to both their perceptions and their internal way of "reasoning". This semantics is evolutionary, as it models each step of this evolution as the generation of an updated agent program with a correspondingly updated semantics. The proposed approach may constitute a ground for comparing different languages/approaches, according to: (i) which factors trigger the transition form one version of the agent program to the next one; (ii) which kind of transformation is performed, and which changes this implies in the agent behavior. Then, such a semantic view of logical agents can make verification techniques such as model-checking easier to apply in this field.

In logic-programming-based languages such as DALI, a procedural semantics can be defined, that corresponds to the declarative one [20] and can then be linked to the operational model [9]. Another advantage of the approach for logic-programming-based languages is that all the analysis, debugging and optimization techniques related this kind of languages (such as methods for program analysis and optimization, abstract interpretation, partial evaluation, debugging, etc.) remain applicable.

Several aspects of the agent behavior remain however not described by the proposed semantics, e.g., how often to check for incoming messages, how often to perform belief-revision, etc. These aspects do not affect the logical semantics of the agent, but affect in a relevant way its run-time behavior, according to Kowalski's famous principle *Program = Logic + Control*. In practice, this kind of "tuning" can be done via directives associated to the agent program. Directives can be even specified in a separate module, which is to be added to the agent program when the agent is initialized. Then, on the one hand directives can be modified without even knowing the agent program. On the other hand, the same agent program with different directives results in a different agent (e.g., apparently more quick, more lazy, eager to remember of ready to forget things, etc.). Directives can be coped with in the operational semantics of the language [9]. It seems more difficult to account for them in the declarative semantics, but this is however a subject of future research.

References

1. J. Barklund, S. Costantini, P. Dell'Acqua e G. A. Lanzarone, *Reflection Principles in Computational Logic*, J. of Logic and Computation, Vol. 10, N. 6, December 2000, Oxford University Press, UK.
2. P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, R. Ross and V.S. Subrahmanian, *Heterogeneous Agent Systems*, The MIT Press, 2000.
3. A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni, *The KGP Model of Agency for Global Computing: Computational Model and Prototype Implementation*. In Global Computing 3267, (2004).
4. M. E. Bratman, D. J. Israel and M. E. Pollack, *Plans and resource-bounded practical reasoning*," Computational Intelligence, vol. 4, pp. 349-355, 1988.
5. Clark, K. L. and McCabe G. *Go! A multi-paradigm programming language for implementing multi- threaded agents*, Annals of Mathematics and Artificial Intelligence 41, ISSN: 1012-2443, pp. 171 - 206, 2004.
6. 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), Available on-line, URL
   http://www.di.unipi.it/
   brogi/ResearchActivity/COCL99/proceedings/index.html.
7. 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, LNAI 2424, Springer-Verlag, 2002.
8. S. Costantini, A. Tocchio. *The DALI Logic Programming Agent-Oriented Language*. In J. J. Alferese and J. Leite(eds.), Logics in Artificial Intelligence, Proceedings of the 9th European Conference, Jelia 2004, Lisbon, September 2004. LNAI 3229, Springer-Verlag, Germany, 2004.

9. S. Costantini, A. Tocchio and A. Verticchio *A Game-Theoretic Operational Semantics for the DALI Communication Architecture*, In Proc. of WOA04, Turin, Italy, ISBN 88-371-1533-4, December 2004.
10. S. Costantini, A. Tocchio and A. Verticchio. *Communication and Trust in the DALI Logic Programming Agent-Oriented Language*, In M. Cadoli, M. Milano and A. Omicini (eds), Italian J. of Artificial Intelligence, March 2005.
11. M. d'Inverno K. Hindriks and M. Luck, *A formal architecture for the 3APL agent programming language*, In First Intern. Conf. on B and Z Users, Springer-Verlag 1878, 2000, pp.168-187.
12. FIPA, *Communicative Act Library Specification*, Technical Report XC00037H, Foundation for Intelligent Physical Agents, 10 August 2001.
13. M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming*. In Proceedings of the Fifth Joint International Conference and Symposium. The MIT Press, Cambridge, MA, 1988, 1070–1080.
14. 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.
15. Kowalski, R. A. and Sadri, F. *Towards a unified agent architecture that combines rationality with reactivity*, In Proc. International Workshop on Logic in Databases, San Miniato (PI), Italy, LNCS 1154, Springer Verlag, Berlin, 1996.
16. R. A. Kowalski, and F. Sadri. *An Agent Architecture that Unifies Rationality with Reactivity* , Department of Computing, Imperial College, 1997.
17. J. W. Lloyd. Foundations of Logic Programming, 1987
18. A. S. Rao, *AgentSpeak(L): BDI agents speak out in a logical computable language*, In W. Van de Velde and J. W. Perram, (eds.), Agents Breaking Away: Proc. of the Seventh Europ. Work. on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038, Springer-Verlag, Heidelberg, Germany, pp. 42-55, 1996.
19. A. S. Rao and M. Georgeff, *BDI Agents: from theory to practice*, In Proc. of the First Int. Conf. on Multi-Agent Systems (ICMAS-95), San Francisco, CA, pp. 312-319, 1995.
20. A. Tocchio. *Multi-Agent sistems in computational logic*, Ph.D. Thesis, Dipartimento di Informatica, Universitá degli Studi di L'Aquila, 2005.
21. 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.
22. A. Van Gelder, K. A. Ross and J. Schlipf. *The well-founded semantics for general logic programs*, J. of the ACM *38(3)*, 620-650, 1990.
23. M. Wooldridge and N. R. Jennings, *Intelligent agents: Theory and practice* , *Knowl. Eng. Rev.*, Vol. 10., N. 2, pp. 115-152, 1995.