Planning Experiments
in the DALI Logic Programming Language*

Stefania Costantini and 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 We discuss some features of the new logic programming language DALI for agents and multi-agent systems. In particular, we aim at illustrating the treatment of proactivity, which is based on the mechanism of the *internal events*. This mechanism is general and flexible, and it is different from all the other approaches that can be found in the literature. In this paper, as a case-study we discuss the design and implementation of an agent capable to perform simple forms of planning. In particular, we demonstrate how it is possible in DALI to perform STRIPS-like planning without implementing a meta-interpreter. In fact a DALI agent, which is capable of complex proactive behavior, can build step-by-step her plan by proactively checking for goals and possible actions.

1 Introduction

The new logic programming language DALI [1], [3], [2] has been designed for modeling Agents and Multi-Agent systems in computational logic. Syntactically, DALI is close to the Horn clause language and to Prolog. In fact, DALI can be seen as a "Prolog for agents" in the sense that it is a general-purpose language, without prior commitment to a specific agent architecture. Rather, DALI provides a number of mechanisms that enhance the basic Horn-clause language to support the "agent-oriented" paradigm. Then, DALI programs are agents which are reactive, proactive, capable to communicate and to make actions. The definition of DALI is meant to contribute to the understanding of what the agent-oriented paradigm may mean in computational logic. In fact, in the context of a purely logic semantics and of a resolution-based interpreter, some new features have been introduced: namely, events can be considered under different perspectives, and there is a careful treatment of proactivity and memory.

DALI programs may contain a special 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 from other agents. In response, a DALI agent can either perform actions or send messages. This

is pretty usual in agent formalisms aimed at modeling reactive agents (see among the main approaches [7], [4], [5] [15], [14], [17]).

There are however in DALI some aspects that can hardly be found in the above-mentioned approaches. First, the same external event can be considered under different points of view: the event is first perceived, and the agent may reason about this perception; then a reaction can take place; finally, the event and the (possible) actions that have been performed are recorded as past events and past actions. The language has advanced proactive features, discussed at length in this paper.

The new approach proposed by DALI is compared to other existing logic programming languages and agent architectures such as ConGolog, 3APL, IMPACT, METATEM, AgentSpeak in [3]. It is useful to remark that DALI does not commit to any agent architecture. Differently from other significant approaches, e.g., DESIRE [6], DALI agents do not have pre-defined submodules. Thus, different possible functionalities (problem-solving, cooperation, negotiation, etc.) and their interactions must be implemented specifically for the particular application. DALI is not directly related to the BDI approach, although its proactive mechanisms allow BDI agents to be easily implemented.

The declarative semantics of DALI, briefly summarized in Section 4, is an *evolutionary semantics,* where the meaning 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 Clauses. The agent receiving an event/making an action is formalized as a program transformation step. The evolutionary semantics consists of a sequence of logic programs, resulting from these subsequent transformations, together with the sequence of the models 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. Procedurally, the interpreter simulates the program transformation steps, and applies an extended resolution which is correct with respect to the model of the program at each stage.

The proactive capabilities of DALI agents, on which we concentrate in this paper, are based on considering (some distinguished) internal conclusions as events, called "internal events": this means, a DALI agent can "think" about some topic, the conclusions she takes can determine a behavior, and, finally, she is able to remember the conclusion, and what she did in reaction. Whatever the agent remembers is kept or "forgotten" according to suitable conditions (that can be set by directives). Then, a DALI agent is not a purely reactive agent based on condition-action rules: rather, it is a reactive, proactive and rational agent that performs inference within an evolving context.

The DALI language has not been designed for planning: Answer Set Programming [11] [10] (based on the Answer Set Semantics of [8] [9]), is in our opinion the best logic programming paradigm for planning. In fact, DALI agents can invoke an Answer Set Solver [Systems] for building plans. Nevertheless, an agent must be able to act in a goal-oriented way, to solve simple planning problems (regardless to optimality) and to perform tasks. To this aim, we have introduced a subclass of internal events, namely the class of "goals", that once invoked are attempted until they succeed, and then expire.

In this paper we want to demonstrate that the features of DALI allow many forms of commonsense reasoning to be gracefully represented. In particular, to show the usefulness of the "goal" mechanism, we will consider as a case-study the implementation of STRIPS-like planning. We will show that it is possible to design and implement this kind of planning without defining a meta-interpreter like is done in [12] (Ch. 8, section on Planning as Resolution). Rather, each feasible action is managed by the agent's proactive behavior: the agent checks whether there is a goal requiring that action, sets up the possible subgoals, waits for the preconditions to be verified, performs the actions (or records the actions to be done if the plan is to be executed later), and finally arranges the postconditions.

DALI is fully implemented in Sicstus Prolog [16]. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

The paper is organized as follows. In Section 2 the principles and concepts of DALI proactive features are outlined; in Section 3 the language syntax, main constructs and their use are illustrated; in Section 4 the evolutionary semantics is briefly recalled; finally, in Section 5 we present the case-study.

2 Proactivity in DALI

The basic mechanism for providing proactivity in DALI is that of the *internal events*. Namely, the mechanism is the following: an atom A is indicated to the interpreter as an internal event by means of a suitable directive. If A succeeds, it is interpreted as an event, thus determining the corresponding reaction. By means of another directive, it is possible to tell the interpreter that A should be attempted from time to time: the directive also specifies the frequency for attempting A, and the terminating condition (when this condition becomes true, A will be not attempted any more).

Internal events are events that do not come from the environment. Rather, they are predicates defined in the program, that allow the agent to introspect about the state of her own knowledge, and to undertake a behavior in consequence. This mechanism has many uses, and also provides a mean for gracefully integrating object-level and meta-level reasoning. It is also possible to define priorities among different internal events, and/or constraints stating for instance that a certain internal event is incompatible with another one. Internal events start to be attempted when the agent is activated, or upon a certain condition, and keep being attempted (at the specified frequency) until the terminating condition occurs. The syntax of a directive concerning an internal event p is the following:

    try p [since *SCond*] [frequency f] [until *TCond*].

It states that: p should be attempted at a frequency f; the attempts should start whenever the initiating condition *SCond* becomes true; the attempts should stop as soon as the terminating condition *TCond* becomes true. All fields are optional. If all of them are omitted, then p is attempted at a default frequency, as long as the agent is active.

Whenever p succeeds, it is interpreted as an event to which the agent may react, by means of a *reactive rule*:

$$pI \;:> \; R_1, \ldots, R_n.$$

The postfix I added to p in the head of the reactive rule stands for "internal", and the new connective $:>$ stands for *determines*. The rule reads: "if the internal event pI has happened, pI will determine a reaction that will consist in attempting $R_1, \ldots, R_n$". The reaction may involve making actions, or simply reasoning on the event.

Below is a simple example of an internal event to be attempted more or less every five minutes:

> *ready(cake)* :- *in_the_oven(cake), color(cake,golden), smell(cake,good).*
> *ready(cake)I* :> *take_from_oven(cake), switch_off_the_oven, eat(cake).*

A special kind of internal event is a *goal*. Differently from the other internal events, goals start being attempted either when encountered during the inference process, or when invoked by an external event. Each goal G will be automatically attempted until it succeeds, and then expires. Moreover, if multiple definitions of G are available, they are (as usual) applied one by one by backtracking, but success of one alternative prevents any further attempt. The examples that we propose in the ongoing for STRIPS-like (or, also, BDI-AgentSpeak like) planning are aimed at showing the power, generality and usability of DALI internal events and goals.

3 DALI in a nutshell

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

3.1 Events

Let us consider an event arriving to the agent from its "external world", like for instance $bell\_ringsE$ (postfix E standing for "external"). From the agent's perspective, this event can be seen in different ways.

Initially, the agent has perceived the event, but she still has not reacted to it. The event is now seen as a present event $bell\_ringsN$ (postfix N standing for "now"). She can at this point reason about the event: for instance, she concludes that a visitor has arrived, and from this she realizes to be happy.

> *visitor_arrived* :- *bell_ringsN.*
>   *happy* :- *visitor_arrived.*

Then, the reaction to the external event $bell\_ringsE$ can be to go and open the door. This is specified by the following reactive rule.

$$bell\_ringsE \; :> \; go\_to\_open.$$

About going to open the door, there are two possibilities: one is that the agent is dressed already, and thus performs the action of opening the door directly (*open_the_doorA*, postfix A standing for "action"). The other one is that the agent is not dressed, and thus she has to get dressed in the first place. The action *get_dressedA* has a defining rule, emphasized by the special token $:<$, that specifies the preconditions for the action to be performed.

$$go\_to\_open \;\; :- \;\; dressed, open\_the\_doorA.$$
$$go\_to\_open \;\; :- \;\; not \;\; dressed,$$
$$get\_dressedA, open\_the\_doorA.$$
$$get\_dressed \;\; :< \;\; grab\_clothes.$$

External events and actions are used also for sending and receiving messages. Actually, DALI is equipped with a FIPA-compliant Agent Communication Language. Since however the precise format of messages does not matter here, we simply assume an external event/message to be a triple $Sender : Event\_Atom : Timestamp$. The $Sender$ and $Timestamp$ fields can be omitted whenever not needed.

Going back to the agent of the above example, we may imagine that, if she is happy, she feels like singing a song, which is an action (postfix A). This is obtained by means of the mechanism of internal events.

Conclusion $happy$, reinterpreted as an event (postfix I standing for "internal"), determines a reaction, specified by the following *reactive rule*:

$$happyI \; :> \; sing\_a\_songA.$$

As outlined above, the mechanism is the following: conclusion $happy$ has been indicated to the interpreter as an internal event by means of a directive. Then, from time to time the agent wonders whether she is happy, by attempting to prove $happy$. If $happy$ succeeds, it is interpreted as an event, thus triggering the corresponding reaction, until a terminating condition occurs. Another directive can also state in which exceptional situations $happy$ should not be interpreted as an event.

3.2   Past Events

The agent remembers events and actions, thus enriching her reasoning context. An event (either external or internal) that has happened in the past will be called *past event,*

and written $bell\_ringsP$, $happyP$, etc., postfix P standing for "past". Similarly for an action that has been performed. It is also possible to indicate to the interpreter plain conclusions that should be recorded as *past conclusions* (which, from a declarative point of view, are just lemmas).

A practically useful role of past conclusions is that of eliminating subsequent alternatives of a predicate definition upon success of one of them. Assume that the user has designated predicate q as a conclusion to be recorded (it will be recorded with syntax qP). Then, she can state that only one successful alternative for q must be considered (if any), by means of the following definition:

$$q \ :\text{-} \ not \ qP, \langle def_1 \rangle.$$
$$\ldots$$
$$q \ :\text{-} \ not \ qP, \langle def_n \rangle.$$

In this way we have implemented goals on top of internal events.

3.3 Goals

As mentioned above, a *goal* (postfix G) is a special kind of internal event. Goals start being attempted either when encountered during the inference process, or when invoked by an external event. Each goal, say $goalG$, will be automatically attempted until it succeeds, and then expires. If multiple definitions of $goalG$ are available, they are (as usual) applied one by one by backtracking, but success of an alternative prevents any further attempt.

When $goalG$ becomes true, a reaction is triggered, by means of a reactive rule:

$$goalI \ :> \ R_1, \ldots, R_k$$

After reaction, the goal is recorded as a past event $goalP$, so as the agent is aware that it has been achieved. This may in turn allow other internal events or goals to succeed, and so on. Then, a DALI agent is in constant evolution.

Goals can be used in a planning or problem-solving mechanism, for instance by employing the following schema:

$$goalG \ :\text{-} \ condition_{11}, \ldots, condition_{1k}(1)$$
$$subgoal_{11G}, \ldots, subgoal_{1nG}(2)$$
$$subgoal_{11P}, \ldots, subgoal_{1nP}(3)$$

where:

 part (1) verifies the preconditions of the goal;

part (2) represents the invocation of the subgoals;

part (3) verifies that previously invoked subgoals have been achieved (they have become past conclusions);

The reason why $goalG$ must be attempted repeatedly is that, presumably, in the first place either some of the preconditions will not hold, or some of the subgoals will not succeed. The reason why part (3) is needed is that a (sub)goal is assumed to have succeeded whenever the corresponding reaction has taken place.

If the goal is part of a problem-solving activity, or if it is part of a task, then the reaction may consist in directly making actions. In planning, the reaction may consist in updating the plan (by adding to it the actions that will have to be performed whenever the plan will be executed).

3.4   Coordinating Actions based on Context

A DALI agent builds her own context, where she keeps track of the events that have happened in the past, and of the actions that she has performed. As discussed above, whenever an event (either internal or external) is reacted to, whenever an action subgoal succeeds (and then the action is performed), and whenever a distinguished conclusion is reached, this is recorded in the agent's knowledge base.

Past events and past conclusions are indicated by the postfix P, and past actions by the postfix PA. The following rule for instance says that Susan is arriving, since we know her to have left home.

> *is_arriving(susan)* :- *left_homeP(susan)*.

The following example illustrates how to exploit past actions. We consider an agent who opens and closes a switch upon a condition. For the sake of simplicity we assume that no exogenous events influence the switch. The action of opening (resp. closing) the switch can be performed only if the switch is closed (resp. open). The agent knows that the switch is closed if she remembers to have closed it previously. The agent knows that the switch is open if she remembers to have opened it. Predicates *open* and *close* are internal events, that periodically check the opening/closing condition, and, whenever true, perform the action (if feasible). previously.

> *open* :- *opening_cond*.
> *openI* :> *open_switchA*.
> *open_switchA* :< *switch_closed*.
> *switch_closed* :- *close_switchPA*.
> *close* :- *closing_cond*.
> *closeI* :> *close_switchA*.
> *close_switchA* :< *switch_open*.
> *switch_open* :- *open_switchPA*.

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

$$keep\ open\_switchPA\ until\ close\_switchA.$$

As soon as the $until$ condition (that can also be $forever$) is fulfilled, i.e., the corresponding subgoal has been proved, the past event/action is removed. 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. Past events, past conclusions and past actions, which constitute the "memory" of the agent, are an important part of the (evolving) context of an agent. The other components are the queue of the present events, and the queue of the internal events. Memories make the agent aware of what has happened, and allow her to make predictions about the future.

The following example illustrates the use of actions with preconditions. The agent emits an order for a product $Prod$ 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. We want to express that the order can be done either by phone or by fax, but not both, and we do that by exploiting past actions, and say that an action cannot take place if the other one has already been performed. Here, not is understood as default negation.

$$
\begin{aligned}
need\_supplyE(Prod)\ &:>\ emit\_order(Prod).\\
emit\_order(Prod)\ &:\text{-}\ phone\_orderA(Prod),\\
&\quad\ not\,fax\_orderPA(Prod).\\
emit\_order(P)\ &:\text{-}\ fax\_orderA(Prod),\\
&\quad\ not\,phone\_orderPA(Prod).
\end{aligned}
$$

This can be reformulated in a more elaboration-tolerant way by the constraints:
:- $fax\_orderA(Prod), phone\_orderPA(Prod)$
:- $fax\_orderPA(Prod), phone\_orderA(Prod)$
thus eliminating negations from the body of the action rules.

4   Semantics

The DALI interpreter can answer user queries like the standard Prolog interpreter, but in general it manages a disjunction of goals. In fact, from time to time external and internal

event will be added (as new disjuncts) to the current goal. The interpreter extracts the events from queues where they occur in the order in which they have been generated.

All the features of DALI that we have previously discussed are modeled in a declarative way. For a full definition of the semantics the reader may refer to [3]. We summarize the approach here, in order to make the reader understand how the examples actually work.

Some language features do not affect at all the logical nature of the language. In fact, attempting the goal corresponding to an internal event just means trying to prove something. Also, storing a past event just means storing a lemma.

Reaction and actions are modeled by suitably modifying the program. This means, inference is performed not in the given program, but in a modified version where language features are reformulated in terms of plain Horn clauses.

Reception of an event is modeled as a program transformation step. I.e., each event that arrives determines a new version of the program to be generated, and then we have a sequence of programs, starting from the initial one. In this way, we do not introduce a concept of state which is incompatible with a purely logic programming language. Rather, we prefer the concept of program (and model) evolution.

More precisely, we define the declarative semantics of a given DALI program P in terms of the declarative semantics of a modified program P_s, obtained from P by means of syntactic transformations that specify how the different classes of events/conclusions/actions are coped with. For the declarative semantics of P_s we take the Well-founded Model, that coincides with the the Least Herbrand Model if there is no negation in the program (see [13] for a discussion). In the following, for short we will just say "Model". It is important to notice that P_s is aimed at modeling the declarative semantics, which is computed by a bottom-up immediate-consequence operator. The declarative semantics will then correspond to the top-down procedural behavior of the interpreter.

We assume that events which have happened are recorded as facts. We have to formalize the fact that a reactive rule is allowed to be applied only if the corresponding event has happened. 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:

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

In a similar way we 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.

Then, we have to declaratively model actions, without or with an action rule. An action is performed as soon as its preconditions are true *and* it is invoked in the body of a rule, such as:

$$B \ :< \ D_1, \ldots, D_h, aA_1, \ldots, aA_k. \quad h \geq 1, k \geq 1$$

where the aA_i's are actions and the D_j's are not actions. Then, for every action atom aA, with action rule

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

we modify this rule into:

$$aA \;\; :- \;\; D_1, \ldots, D_h, C_1, \ldots, C_s.$$

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

$$aA \;\; :- \;\; D_1, \ldots, D_h.$$

We repeat this for every rule in which aA is invoked.

In order to obtain the *evolutionary* declarative semantics of P, 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. 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.

Formally we have:

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

where

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

It is possible to extend Σ_P so as to deal with internal events, add as facts past actions and conclusions, and remove the past events that have expired.

Definition 1. *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 2. *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 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". It is easy to see that, given event list $[E_n, \ldots, E_1]$, DALI resolution simulates standard SLD-Resolution on $Prog_n$.

Definition 3. *Let P_s be a DALI program, L be a list of events. The* evolutionary seman-
tics *$\mathcal{E}_{P_s}$ of P_s with respect to L is the couple $\langle \mathcal{P}(P_s, L), \mathcal{M}(P_s, L) \rangle$.*

5   A sample application: STRIPS-like planning

In this section we show that the DALI language allows one to define an agent that is
able to perform planning (or problem-solving) in a STRIPS-like fashion without imple-
menting a metainterpreter.

For the sake of simplicity, the planning capabilities that we consider are really basic,
e.g., we do not consider here the famous STRIPS anomaly, and we do not have any
pretense of optimality. However, we will show that the goal mechanism of DALI is
general enough so that failure of one subgoal does not necessarily affect the others, and
that the agent can still be able to reach feasible objectives.

We consider the sample task of putting on socks and shoes. Of course, the agent
should put her shoes on her socks, and she should put both socks and both shoes on.

We suppose that some other agent sends a message to ask our agent to wear the
shoes. This message is an external event, which is the head of a reactive rule: the
body of the rule specifies the reaction, which in this case consists in invoking the goal
put_your_shoesG.

$$goE \; :> \; put\_your\_shoesG.$$

This goal will be attempted repeatedly, until it will be achieved.

It is important to recall the mechanism of DALI goals:

– For a goal g to be achieved, the predicate gG must become true, by means of a rule
  $gG :\text{-} Conds$, where $Conds$ specify preconditions and subgoals.
– For a goal g to be achieved, as soon as gG becomes true the reactive rule
  $gG :> PostAndActions$ is activated, that performs the actions and/or sets the post-
  conditions related to the goal.
– as soon as a goal gG is achieved (or, in short, we say that gG *succeeds*, even though
  this involves the above two steps), it is recorded as a past event, in the form gP.

This explains the structure of the rule below:

$$put\_your\_shoesG \; :\text{-} \; put\_right\_shoeG, put\_left\_shoeG,$$
$$put\_right\_shoeP, put\_left\_shoeP.$$

In particular, it is required that the agent puts both the right and left shoe on. These
conditions occur first as subgoals, that will be attempted, and then as past events, in
order to check whether they actually succeeded. This means, *put_your_shoesG* will be-
come true only if its subgoals will have been achieved. In practice, after the invocation
of the subgoals, the overall goal is suspended until the subgoals become past events.

In the meantime, the subgoals *put_right_shoeG* and *put_left_shoeG* will be attempted.

put_right_shoeG :- *have_right_shoe*, *put_right_sockG*, *put_right_sockP*.

This rule verifies a precondition, i.e., that of having the shoe to put on. Then it attempts the subgoal *put_right_sockG* and waits for its success, i.e., waits for the subgoal to become a past event. The rule for the subgoal is:

put_right_sockG :- *have_right_sock*.

This rule doesnt invoke subgoals, but it just checks the precondition, i.e., to have the right sock. Upon success, the corresponding reactive rule is triggered:

put_right_sockI :> *right_sock_on*.

Now we have two possibilities: in a problem-solving activity, we will have the rule:

right_sock_on :- *wear_right_sockA*.

that actually executes the action of wearing the sock. In a planning activity, we will have the rule:

right_sock_on :- *update_plan(wear_right_sock)*.

that adds to the plan that is being built the step of wearing the sock. In any case, the goal *put_right_sockG* has been achieved, and will be now recorded as past event.

At this point, *put_right_shoeG* becomes true, thus triggering the reactive rule:

put_right_shoeI :> *right_shoe_on*.

After having made (or recorded) the action of wearing the shoe, *put_right_shoeP* becomes true.

Analogously, the agent will eventually record the past event *put_left_shoeP*. Since the subgoals of wearing the right and the left shoe are unrelated, no order is enforced on their execution. Only, the overall goal becomes true whenever both of them have been achieved.

At this point, the reactive rule related to the overall goal will be applied:

put_your_shoesI :> message(tell_shoes_onA).

which means that the goal has succeeded, and in particular the agent declares to have the shoes on.

The planning mechanism that we have outlined activates a descendant process that invokes the subgoals, and an ascending process that executes (or records) the corresponding actions.

This methodology allows an agent to construct plans dynamically. In fact, a change of the context, i.e., new information received from the outside, can determine success of subgoals that could not succeed before.

Another interesting aspect is related to multiple definitions of a goal. It is useful to recall that the first successful one eliminates all the other alternatives.

In the example below, by means of multiple definitions one can avoid that the possible failure of partial objectives blocks the achievement of the others.

The planner outlined below is structured so as to allow the agent to achieve as much as possible. We propose the example of an agent that has to go to the supermarket in order to buy milk and bananas, and to the hardware shop in order to buy a drill.

> *buy_allG :- supermarket_open, hardware_shop_open,*
> *buy_bananaG, buy_milkG,*
> *buy_drillG, go_to_homeG,*
> *pay_bananaP, pay_milkP, pay_drillP, at_homeP.*

> *buy_allG :- supermarket_open, not hardware_shop_open,*
> *buy_bananaG, buy_milkG, go_to_homeG,*
> *pay_bananaP, pay_milkP, at_homeP.*

> *buy_allG :- not supermarket_open, hardware_shop_open,*
> *buy_drillG, go_homeG,*
> *pay_drillP, at_homeP.*

> *buy_allI :> whateverA.*

To check that the above planners actually work fine, the reader is invited to refer to the DALI web site, URL http://gentile.dm.univaq.it/dali/dali.htm.

A future direction of this experimental activity is that of writing a meta-planner with general meta-definitions for root, intermediate and leaf planner goals. This meta-planner would accept the list of planner goals, and for each of them the list of plans, preconditions and actions.

6   Concluding Remarks

We have presented how to implement a naive version of STRIPS-like planning in DALI, mainly by using the mechanism of internal events and goals. However, the ability of DALI agents to behave in a "sensible" way comes from the fact that DALI agents have several classes of events, that are coped with and recorded in suitable ways, so as to form a context in which the agent performs her reasoning. A simple form of knowledge update and "belief revision" is provided by the conditional storing of past events and actions. In the future, more sophisticated belief revision strategies will be integrated into the formalism.

7   Acknowledgments

References

1. S. Costantini, *Towards active logic programming*, In A. Brogi and P. Hill, (eds.), *Proc. of 2nd International Works. 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.
2. S. Costantini, S. Gentile and A. Tocchio, *DALI home page:* http://gentile.dm.univaq.it/dali/dali.htm.
3. 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*, Cosenza, Italy, September 2002, LNAI 2424: Springer-Verlag, Berlin, 2002
4. P. Dell'Acqua, F. Sadri, and F. Toni, *Communicating agents*, In *Proc. International Works. on Multi-Agent Systems in Logic Progr., in conjunction with ICLP'99*, Las Cruces, New Mexico, 1999.
5. M. Fisher, *A survey of concurrent METATEM – the language and its applications*, In *Proc. of First International Conf. on Temporal Logic (ICTL)*, LNCS 827, Springer Verlag, Berlin, 1994.
6. C. M. Jonker, R. A. Lam and J. Treur, *A Reusable Multi-Agent Architecture for Active Intelligent Websites. Journal of Applied Intelligence*, vol. 15, 2001, pp. 7-24.
7. R. A. Kowalski and F. Sadri, *Towards a unified agent architecture that combines rationality with reactivity*, In *Proc. International Works. on Logic in Databases*, LNCS 1154,Springer-Verlag, Berlin, 1996.
8. M. Gelfond and V. Lifschitz, *The Stable Model Semantics for Logic Programming,* In: R. Kowalski and K. Bowen (eds.) Logic Programming: Proc. of 5th International Conference and Symposium, The MIT Press, 1988: 1070–1080.

9. M. Gelfond and V. Lifschitz, *Classical Negation in Logic Programming and Disjunctive Databases,* New Generation Computing 9, 1991: 365–385.
10. V. Lifschitz, *Answer Set Planning,* in: D. De Schreye (ed.) Proc. of the 1999 International Conference on Logic Programming (invited talk), The MIT Press, 1999: 23–37.
11. W. Marek and M. Truszczyński, *Stable Models and an Alternative Logic Programming Paradigm,* In: The Logic Programming Paradigm: a 25-Year Perspective, Springer-Verlag, Berlin, 1999: 375–398.
12. D. Poole, A. Mackworth, R. Goebel, *Computational Intelligence*: ISBN 0-19-510270-3, Oxford University Press, New York, 1998.
13. H. Przymusinska and T. C. Przymusinski, *Semantic Issues in Deductive Databases and Logic Programs*, R.B. Banerji (ed.) Formal Techniques in Artificial Intelligence, a Sourcebook: Elsevier Sc. Publ. B.V. (North Holland), 1990.
14. 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: Proc. of the Seventh European Works. on Modelling Autonomous Agents in a Multi-Agent World*, LNAI: Springer Verlag, Berlin, 1996.
15. A. S. Rao and M. P. Georgeff, *Modeling rational agents within a BDI-architecture*, In R. Fikes and E. Sandewall (eds.), Proc. of Knowledge Representation and Reasoning (KR&R-91): Morgan Kaufmann Publishers: San Mateo, CA, April 1991.
16. SICStus home page: *http://www.sics.se/sicstus/*.
[Systems] Web location of the most known ASP solvers:
aspps: *http://www.cs.uky.edu/ai/aspps/*
CCalc: *http://www.cs.utexas.edu/users/tag/cc/*
Cmodels: *http://www.cs.utexas.edu/users/tag/cmodels.html*
DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
NoMoRe: *http://www.cs.uni-potsdam.de/ linke/nomore/*
SMODELS: *http://www.tcs.hut.fi/Software/smodels/*
17. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross, *Heterogenous Active Agents*: The MIT Press, 2000.