Strips-like Planning in the DALI Logic Programmming Language

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

***Abstract*— In this paper we discuss how some features of the new logic programming language DALI for agents and multi-agent systems are suitable to programming agents equipped with planning capabilities. We discuss the design and implementation of an agent capable to perform STRIPS-like planning, and we propose a small but significant example. In particular, 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. We demonstrate how general and flexible is the treatment of proactivity in DALI, which is different from all the other approaches that can be found in the literature.**

I. 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. DALI programs however may contain a new 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] [11], [10]), [13].

What is new in DALI is that 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, and finally the event and the (possible) actions that have been performed are recorded as past events and past actions. Another important novel feature is that internal conclusions can be seen as 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 new approach proposed by DALI is compared to other existing logic programming languages and agent architectures such as ConGolog, 3APL, IMPACT, METATEM, BDI in [3]. However, it is useful to remark that DALI does not commit to any agent architecture. Differently from other significant approaches like, 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 in fact an "agent-oriented" general-purpose language that provides, as discussed below, a number of primitive mechanisms for supporting this paradigm, all of them within a precise logical semantics.

The declarative semantics of DALI 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 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. Procedurally, the interpreter simulates the program transformation steps, and applies an extended resolution which is correct with respect to the Least Herbrand Model of the program at each stage.

DALI is fully implemented in Sicstus Prolog [12]. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

In this paper we want to demonstrate that the features of the DALI language allow many forms of commonsense reasoning to be gracefully represented, and in particular we will consider STRIPS-like planning. We will show that it is possible to design and implement this kind of planning without implementing a meta-interpreter like is done in [8] (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, and finally arranges the postconditions.

II. 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).

I.e., 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 possible to define (again by means of directives) priorities among different internal events, and/or constraints stating for instance that a certain internal event is incompatible with another one.

A special kind of internal event is a *goal*. Similarly to any internal event, a goal P is executed whenever encountered in the (resolution-based) inference process, or can be automatically attempted according to a suitable directive. In both cases however, if multiple definitions of P are available, they are (as usual) applied one by one by backtracking, but success prevents any further attempt. As discussed later, this mechanism is defined in a declarative way on top of plain internal events, without resorting to extra-logical predicates like Prolog 'cut'.

The implementation that we propose for STRIPS-like planning is aimed at showing the power, generality and usability of DALI internal events and goals. To the best of our knowledge, no other agent language in the literature provides this kind of mechanisms, with this kind of generality.

III. DALI IN A NUTSHELL

A DALI program is syntactically very close to a traditional Horn-clause program. In particular, 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.

Let us consider an event incoming into 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 have 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*.

As 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: this is a novel feature of the DALI language, that to the best of the authors' knowledge cannot be found in any other language. Conclusion $happy$, reinterpreted as an event (postfix I standing for "internal"), determines a reaction, specified by the following *reactive rule*, where new connective $:>$ stands for *determines*:

happyI $:>$ *sing_a_songA*.

In more detail, 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. Another directive can also state in which exceptional situations $happy$ should not be interpreted as an event.

Finally, the reaction to the external event $bell\_ringsE$ can be that of go to open the door:

bell_ringsE $:>$ *go_to_open*.
go_to_open :- *dressed*, *open_the_doorA*.
go_to_open :- *not dressed*,
get_dressedA, *open_the_doorA*.
get_dressed $:<$ *grab_clothes*.

For 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. 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.

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 distinguished useful role of past conclusions is that of eliminating subsequent alternatives of a predicate definition upon success of one of them. If for instance the user designates by a directive predicate q as a conclusion to be recorded (it will be recorded with syntax qP), she can state that only one successful alternative for q must be considered (if any), by

means of the following definition:

$$q \quad :\!\!- \quad not \quad qP, \langle def_1 \rangle.$$
$$\ldots$$
$$q \quad :\!\!- \quad not \quad qP, \langle def_n \rangle.$$

This allows DALI goals to be implemented in a declarative way on top of internal events. For the user convenience, in the particular case of DALI goals syntactic sugar has been added, i.e., if q is indicated to be a goal (again by means of a directive) the plain definition of q is automatically rewritten as above.

External events and actions are used also for sending and receiving messages. 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.

The DALI interpreter is able to answer queries like the standard Prolog interpreter, but it is able to handle a disjunction of goals. In fact, from time to time it will add external and internal event as new disjuncts to the current goal, picking them from queues where they occur in the order they have been generated. An event is removed from the queue as soon as the corresponding reactive rule is applied.

IV. 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_switch :< *switch_closed*.
switch_closed :- *close_switchPA*.
close :- *closing_cond*.
closeI :> *close_switchA*.
close_switch :< *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.

need_supplyE(Prod) :> *emit_order(Prod)*.
emit_order(Prod) :- *phone_orderA(Prod)*, not *fax_orderPA(Prod)*.
emit_order(P) :- *fax_orderA(Prod)*, not *phone_orderPA(Prod)*.

This in short can be represented in a more elaboration-tolerant way by the constraint:
:- *fax_orderA(Prod), phone_orderPA(Prod)*
thus eliminating negations from the body of the action rules.

V. Evolutionary Semantics

A DALI agent is affected when events arrive, internal events and/or distinguished conclusions are proved, and actions are performed. The declarative semantics of DALI is aimed at describing how the agent is affected, without explicitly introducing a concept of state which is incompatible with a purely logic programming language. Rather, we prefer the concept of context, where modifications to the context are modeled as program transformation steps. 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.

We define the semantics of a given DALI program P starting from 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 [9] 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 some kind of bottom-up immediate-consequence operator. The declarative semantics will then correspond to the top-down procedural behavior of the interpreter.

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. 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.

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.

Then, 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 any rule, if its preconditions are true. Declaratively, whenever the preconditions of an action are true, the action atom should become true as well. Thus, any rule containing that action atom in its body can possibly be applied by the immediate-consequence operator.

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. 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. Internal events are coped with in a similar way. Finally, Σ also adds as facts past actions and conclusions, as well as facts explicitly added to the program by means of $assert$.

Given P_s and list $L = [E_n, \ldots, E_1]$ of external events, each event E_i *determines* the transition from P_{i-1} to P_i according to Σ.

The list $\mathcal{P}(P_s, L) = [P_0, \ldots, P_n]$ is called the *program evolution* of P_s with respect to L.

The sequence $\mathcal{M}(P_s, L) = [M_0, \ldots, M_n]$ where M_i is the model of $Prog_i$ is the *model evolution* of P_s with respect to L, and M_i the *instant model at step* i .

The *evolutionary semantics* $\mathcal{E}_{P_s}$ of P_s with respect to L is the couple $\langle \mathcal{P}(P_s, L), \mathcal{M}(P_s, L) \rangle$.

The DALI interpreter at each stage basically performs standard SLD-Resolution on $Prog_i$, while however it manages a disjunction of goals, each of them being a query, or the processing of an event.

VI. A complete example: STRIPS-like planning

In this section we implement and discuss an agent who is able to perform planning in a STRIPS-like fashion. Our agent's planning capabilities are really basic, e.g., we do not consider here the famous STRIPS anomaly, and we do not have any pretense of optimality.

We may assume that our agent is a child, able to perform just simple tasks, and in fact we take as an example the 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.

In order to avoid confusion, in what follows when saying *planner goal* or just *goal* we mean a goal of the planner, and when saying *DALI goal* we mean the special kind of internal event discussed in previous sections.

To start the whole thing, we suppose that some other agent, maybe our agent's mother, sends a message to intimate her 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 defining $wear\_shoes$ as a goal to be achieved. This is done by simply asserting a fact $g(wear\_shoes)$, if it is not already present. In general, a fact $g(G)$ indicates that G has been set as a goal, but has not been achieved yet.

$$put\_your\_shoes\_on\_immediatelyE :> define\_goal(wear\_shoes).$$
$$define\_goal(G) :- not(G), assert(g(G)).$$

The goal $wear\_shoes$ has the effect of putting the shoes on, and is coped with by the following rules. The first one, with head $shoes$, checks whether $wear\_shoes$ is actually a goal to be achieved, simply by looking up the fact $g(wear\_shoes)$. If so, it defines its subgoals: wearing the shoes implies wearing both the right and the left shoe, and thus it asserts (by means of procedure $define\_subgoal$) the facts $g(r\_shoe\_on)$ and $g(l\_shoe\_on)$, r standing for "right" and l standing for "left". Then, it waits for its preconditions to be verified. Since this is the "top level" goal, its preconditions coincide with its subgoals, and whenever they have been fulfilled, the overall goal succeeds.

The trick here is that $shoes$ must be declared to be a DALI goal, automatically attempted at a certain frequency. The first time it is called, the conditions $verify\_goal(wear\_shoes)$ and $define\_subgoals(wear\_shoes)$ will succeed, while the third one, i.e., $preconds(wear\_shoes)$, will fail since the subgoals have not been achieved. For a number of times, $shoes$ will be attempted again and again, where the first two conditions will keep succeeding (with no effect, though) and the third one will keep failing. Finally, when the subgoals will have been achieved, also the third condition, i.e., $preconds(wear\_shoes)$, will succeed, thus $shoes$ will succeed. Since it is an internal event, upon success the corresponding reactive rule is triggered, that in this case prints a message and "cleans up" the goal and the subgoals, by removing the corresponding facts.

$$
\begin{aligned}
&shoes \text{ :- } verify\_goal(wear\_shoes),\\
&\qquad\qquad define\_subgoals(wear\_shoes(1)),\\
&\qquad\qquad preconds(wear\_shoes).\\
&shoesI :> write(\text{'}I\ have\ the\ shoes\ on\text{'}), nl,\\
&\qquad\qquad remove\_subgoal(wear\_shoes),\\
&\qquad\qquad remove\_subgoals(wear\_shoes).
\end{aligned}
$$

$$
\begin{aligned}
&define\_subgoals(wear\_shoes) \text{ :-}\\
&\qquad\qquad define\_subgoal(r\_shoe\_on),\\
&\qquad\qquad define\_subgoal(l\_shoe\_on).\\
&preconds(wear\_shoes) \text{ :- } r\_shoe\_on, l\_shoe\_on.\\
&remove\_subgoals(wear\_shoes) \text{ :-}\\
&\qquad\qquad remove\_subgoal(r\_shoe\_on),\\
&\qquad\qquad remove\_subgoal(l\_shoe\_on).
\end{aligned}
$$

Below is the (straightforward) definition of the auxiliary procedures.

$$
verify\_goal(G) \text{ :- } g(G).
$$

$$
\begin{aligned}
&define\_subgoal(G) \text{ :- } not(G), assert(g(G)).\\
&define\_subgoal(\_).
\end{aligned}
$$

$$
\begin{aligned}
&remove\_subgoal(G) \text{ :- } clause(G, \_), retractall(g(G)).\\
&remove\_subgoal(\_).
\end{aligned}
$$

The goal r_shoe_on (and, similarly, the goal l_shoe_on) is coped with by the following rules. Here we have a more general case, since it is an "intermediate" subgoal. Moreover, we have two possible plans. Both of them involve putting a sock on, but we have the choice between the possibility of wearing a red sock, or that of wearing a blue sock. In order to ensure that the agent will be wearing similar socks at both feet, we enforce this in the preconditions of r_shoe_on.

Whenever the preconditions are fulfilled, the reactive rule will result in the execution of the action of wearing the shoe, i.e., put_r_shoeA, postfix A standing for "action." There is another internal event, namely the predicate called $right\_shoe$, whose role is that of checking if the action has been performed, by looking up in the agent's memory the fact put_r_shoePA, postfix PA standing for "past action" that will be added as soon as the action has been successfully performed. As reaction, it will clean up the subgoals, that have been achieved and are thus obsolete, an asserts to have achieved the current subgoal, namely r_shoe_on, so as the parent goal will know, and will be able to proceed.

First part of the definition: the DALI goal r_shoe has two alternatives. We remind the reader that only the first successful one will be actually applied. In the present case, if the first one succeeds, the second one will be ignored. Each of the two *plans* checks that putting the right shoe on is actually to be achieved, and then sets different subgoals and different preconditions for performing the action of wearing the shoe.

$$
\begin{aligned}
&r\_shoe \text{ :- } verify\_goal(r\_shoe\_on),\\
&\qquad\qquad define\_subgoals(r\_shoe\_on(1)),\\
&\qquad\qquad preconds(r\_shoe\_on(1)).\\
&r\_shoe \text{ :- } verify\_goal(r\_shoe\_on),\\
&\qquad\qquad define\_subgoals(r\_shoe\_on(2)),\\
&\qquad\qquad preconds(r\_shoe\_on(2)).
\end{aligned}
$$

Namely, the first plan requires to wear a red sock before putting the shoe on, while the second one requires to wear a blue sock.

$$
\begin{aligned}
&define\_subgoals(r\_shoe\_on(1)) \text{ :-}\\
&\qquad\qquad define\_subgoal(r\_sock\_on(red)).\\
&define\_subgoals(r\_shoe\_on(2)) \text{ :-}\\
&\qquad\qquad define\_subgoal(r\_sock\_on(blue)).
\end{aligned}
$$

Consequently, the two plans set different preconditions for the DALI goal r_shoe to succeed: the first plan requires that the agent is wearing red socks at both feet, while the second plan requires that the agent is wearing blue socks at both feet. This means that, for wearing any of the shoes, the agent is forced to first put on both socks, of the same color. Whatever of the two plans succeeds, r_shoe will succeed as well, and since it is an internal event, the corresponding reaction will occur, consisting in the action of putting the shoe on, that will be recorded as past action put_r_shoePA.

$$
\begin{aligned}
&preconds(r\_shoe\_on(1)) \text{ :- } r\_sock\_on(red), l\_sock\_on(red).\\
&preconds(r\_shoe\_on(2)) \text{ :- } r\_sock\_on(blue), l\_sock\_on(blue).\\
&r\_shoeI :> put\_r\_shoeA.
\end{aligned}
$$

Recording the past action will determine the success of the internal event $right\_shoe$, whose reaction asserts that the planner goal r_shoe_on has been reached, and cleans up all the subgoals previously set.

$right\_shoe$:- put_r_shoePA.
$right\_shoeI$:> $remove\_subgoals(r\_shoe\_on(\_))$,
$assert(r\_shoe\_on)$.
$remove\_subgoals(r\_shoe\_on(\_))$:-
$remove\_subgoal(r\_sock\_on(\_))$,
$remove\_subgoal(l\_sock\_on(\_))$.

The planner goal $r\_sock\_on(\_)$ (and, similarly, the goal $l\_sock\_on(\_)$) is coped with by the following definitions. Precisely, below we cope with putting a red sock on the right foot. Left foot and blue socks are coped with in a similar way. This is a special case, since it is a "leaf" planner goal. It has no subgoals, and its preconditions are facts (precisely, the single fact $have\_r\_sock(red)$ stating that the agent has the sock to put on.

$r\_sock(red)$:- $verify\_goal(r\_sock\_on(red))$,
$define\_subgoals(sock\_on\_r(red))$,
$preconds(put\_r\_sock(r))$.

$define\_subgoals(r\_sock\_on(red))$.
$preconds(put\_r\_sock(red))$:- $have\_r\_sock(red)$.
$r\_sockI(red)$:> $put\_r\_sockA(red)$.

$right\_sock(red)$:- $put\_r\_sockPA(red)$.
$right\_sockI(red)$:> $remove\_subgoal(r\_sock\_on(red))$,
$retractall(have\_r\_sock(red))$,
$assert(r\_sock\_on(red))$.

Important ingredients for the efficiency of the planner are: (i) priorities among the internal events (again to be set by means of directives), stating in this case that lower-level subgoals have higher priority; (ii) possible constraints stating which events are incompatible.

The remark to be done here is that the set of rules managing each subgoal has two parts: an object-level part, stating which preconditions to verify and which actions to perform; and a meta-level part, setting and removing subgoals, and recording which goal has been achieved. The object-level and the meta-level components are managed in a uniform way by means of the internal event mechanism.

To check that the above planner actually works 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 experimentation 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.

VII. 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. 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 as well as full planning capabilities and a real agent communication language will be integrated into the formalism.

VIII. Acknowledgments

This Research has been partially funded by MIUR 40% project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps* and by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

Many thanks to Stefano Gentile, who joined the DALI project, cooperates to the implementation of DALI, has designed the language web site, and is supporting the authors in many ways. We also gratefully acknowledge Prof. Eugenio Omodeo for useful discussions and for his support to this research.

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, Berlin, 1996. Springer-Verlag.

[8] D. Poole, A. Mackworth, R. Goebel, *Computational Intelligence*: ISBN 0-19-510270-3, Oxford University Press, New York, 1998.

[9] 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.

[10] 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.

[11] 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.

[12] SICStus home page: *http://www.sics.se/sicstus/*.

[13] V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross, *Heterogenous Active Agents*: The MIT Press, 2000.