Context-based Commmonsense Reasoning in the DALI Logic Programmming Language[⋆]

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 will discuss the context management features of the new logic programming language DALI, aimed at defining agents and multi-agent systems. In particular, a DALI agent, which is capable of reactive and proactive behaviour, builds step-by-step her context. Context update is modelled by the novel concept of "evolutionary semantics", where each context manipulation is interpreted as a program transformation step. We show that this kind of context-based agent language is well-suited for representing many significant commonsense reasoning examples.

1 Introduction

The new logic programming language DALI [Co99], [CT02], [CGT02] has been designed for modelling 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 modelling reactive agents (see among the main approaches [KS96], [DST99], [Fi94] [Ra91], [Ra96]), [SPDEK00].

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 behaviour, and, finally, she is able to remember the conclusion, and what she did in

[⋆] Research 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 has joined the DALI project, has cooperated to the implementation of DALI, has designed the language web site, and has helped and supported the authors in many ways.

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 *evolutionary semantics* of the language consists of a sequence of logic programs, resulting from subsequent transformations, together with the sequence of the Least Herbrand Models of these programs. This makes it possible to model an evolving agent incorporating an evolving context. In this way, it is possible to reason about the conclusions reached and the actions performed by the agent at a certain stage, or, better, in a certain context.

In this paper we want to demonstrate that the features of the DALI language allow many forms of commonsense reasoning to be gracefully represented.

A prototype implementation of the DALI language has been developed by the authors of this paper at the University of L’Aquila. The implementation, together with a set of examples, is available at the URL [CGT02].

2 Context-dependent Reasoning in everyday situations

A DALI program is syntactically very close to a traditional Horn-clause program. In particular, a Prolog program is a special case of a DALI program. Specific syntactic features have been introduced to 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: goal $happy$ has been indicated to the interpreter as an internal event by means of a suitable directive. Then, from time to time the agent wonders whether she is happy, by trying the goal (the frequency can also be set in the directive). If the goal $happy$ succeeds, it is interpreted as an event, thus triggering the corresponding reaction. I.e., internal events are events that do not come from the environment. Rather, they are goals defined in some other part of the program.

For coping with unexpected unpleasant situations that might unfortunately happen to ruin a good day, one can add a directive of the form:

$keep$ *happyI* $unless$ $\langle$ *terminating_condition* $\rangle$.

stating in which situations $happy$ should not become an internal event. $\langle terminating\_condition \rangle$ is any predicate, that must be explicitly defined in the program, and is attempted upon success of $happy$. This formulation is elaboration-tolerant, since it separates the general definition of happiness, from what (depending on the evolution of the context) might "prevent" happiness.

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

bell_ringsE :> *open_the_doorA*.

After reaction, the agent is able to remember the event, thus enriching her reasoning context. An event (either external or internal) that has happened in the past will be called *past event*, and written $bell\_ringsP$, $happyP$, postfix P standing for "past". 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.

3 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 soon as an event (either internal or external) is reacted to, and whenever an action subgoal succeeds (and then the action is performed), the corresponding atom is recorded in the agent database. By means of directives, it is also possible to indicate other kinds of conclusions that should be remembered. 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. In particular, the action of opening (resp. closing) a door can be performed only if the door is closed (resp. open). The window is closed if the agent remembers to have closed it previously. The window is open if the agent remembers to have opened it previously.

open_the_doorA :- *door_is_closed*.

door_is_closed :- *close_the_doorPA*.

close_the_doorA :- *door_is_open*.

door_is_open :- *open_the_doorPA*.

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

rainE, windE :> *close_windowA*.

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

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 door. However, as soon as she closes the door 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_the_doorPA* $until$ *close_the_doorA*.

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 P of which she needs a supply. The order can be done either by phone or by fax, in the latter case if a fax machine is available.

need_supplyE(P) :> *emit_oder(P)*.
emit_order(P) :- *phone_order$A$*.
emit_order(P) :- *fax_order$A$*.
fax_order$A$:- *fax_machine_available*.

If we want to express that the order can be done either by phone or by fax, but not both, 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(P) :> *emit_order(P)*.
emit_order(P) :- *phone_order$A$*, not *fax_orderPA*.
emit_order(P) :- *fax_order$A$*, not *phone_orderPA*.

4 Evolutionary Semantics

The declarative semantics of DALI is aimed at describing how an agent is affected by actual arrival of events, 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 modelled as program transformation steps. For a full definition of the semantics the reader may refer to [CT02]. 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 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 [PP90] for a discussion). In the following, for short we will just say "Model". It is important to notice that P_s is aimed at modelling the declarative semantics, which is computed by some kind of immediate-consequence operator, and not represent the procedural behaviour 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 (this assumption will be formally assessed later). Then, we reach our aim by adding, for each event atom $p(Args)E$, the event atom itself in the body of its own reactive rule. The meaning is that this rule can be applied by the immediate-consequence operator only if $p(Args)E$ is available as a fact. Precisely, we transform each reactive rule for external events:

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

into the standard rule:

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

Similarly, we have to specify that the reactive rule corresponding to an internal event $q(Args)I$ is allowed to be applied only if the subgoal $q(Args)$ has been proved.

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

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

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

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

we modify this rule into:

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

If A has no defining clause, we add clause:

$A\ \text{:-}\ D_1, \ldots, D_h.$

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

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

Then, given P_s and list $L = [E_n, \ldots, E_1]$ of 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.

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. Then, 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 .

Finally, 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.

5 A complete example: barman and customer

Below we show the DALI code for two agents: *Barman*, who is the shopkeeper of a cafeteria, and *Gino*, who is a customer coming in to drink a beer.

The barman waits for events of the form $C : requestE(P)$ where C is the name of the customer agent, and P is the product he would like to get. For instance, we may have $C = Gino$ and $P = beer$. The barman examines the request, and if Pr is available at a cost A, he asks the customer for payment (in this cafeteria you pay in advance!). Otherwise, he tells the customer that there Pr is not available. The action $messageA(C, M)$ consists in sending message M to agent C.

```
                 Barman
      C:requestE(Pr) :> examine_request(C,Pr).
examine_request(C,Pr) :-  not finished(Pr), cost(Pr,A),
                          messageA(C,ok(Pr)), messageA(C,please_pay(Pr,A)).
examine_request(C,Pr) :-  finished(Pr), messageA(C,no(Pr)).
```

The barman concludes that Pr is finished if the quantity left in store is zero. This conclusion is an internal event, and thus (via the next rule) triggers a reaction, that consists in ordering a supply of the product, but *only if the order has not been issued already*: in fact, in the body of the rule there is a check that there is not in the memory of the agent past action $order\_productPA(Pr, Q1)$.

If the payment arrives (event $paidE(C, Pr, A1)$), then the barman makes some checks. First, if he *remembers* that Pr is finished (in fact, $finishedP$ is a past event), he tells again the customer that Pr is finished, and that he should take the money back. Otherwise, if the customer has paid an amount $A1$ which is different from the cost A, he will be required again to pay. Finally, if everything is ok, Pr will actually be served to the customer. Then, $serve(C, Pr)$ is interpreted as an internal event, and will cause the available quantity of Pr to be updated.

```
Barman (continued)
       finished(Pr)  :-  quantity(Pr,0).
      finishedI(Pr)  :>  not order_productPA(Pr,Q1), order_productA(Pr,Q).
  C:paymentE(Pr,A)  :>  check_payment(C,Pr,A).
check_payment(C,Pr,A1)  :-  finishedP(Pr), messageA(C,no(Pr)), messageA(refund(A1)).
 check_payment(C,Pr,A)  :-  cost(Pr,A), A =/= A1, messageA(C,please_pay(Pr,A)).
 check_payment(C,Pr,A)  :-  serveA(C,Pr).
          serveI(C,Pr)  :>  update_quantity(Pr).
```

We have now to explain one more reason why it is useful to use internal events also form a procedural point of view. In fact, one may wonder why not write a rule such as:

```
check_payment(C,Pr,A) :- serveA(C,Pr), update_quantity(Pr).
```

Consider however that the Barman might receive several concurrent requests by several customers. Therefore, these requests are to be "contextualized", i.e., they have to be considered in a sequence, keeping in mind the information about the available quantity of each product. Procedurally, a purely reactive rule would produce concurrent attempts to update the same quantity. The use of internal events prevents any problem of "dirty update": in fact, the internal events to be reacted to are put in a FIFO queue. Then, the different updates to the quantity of Pr are performed one at a time, and cannot interfere with each other.

Moreover, the mechanism of internal events is more elaboration-tolerant since it separates the phase where the agent becomes aware of something, and the phase where the agent decides what to do in consequence. Rules for updating the quantity are straightforward, and those for making the order have been reported in a previous example.

The code for the customer agent might look for instance like the following. Agent Gino is thirsty whenever he has played tennis. Then, as a reaction ($thirsty$ is an internal event) he asks the barman for a beer. If he is told by the Barman that the beer is finished, as a reaction he asks for a coke. He pays when requested by the external event $please\_payE(beer, amount)$ coming from Barman. The rule for payment is general, and can be used for either beer or coke. Notice that Gino is disappointed whenever what he asked for is not available. This conclusion is drawn from the present event $finishedN(Pr)$ coming from Barman.

```
                     Gino
                  thirsty  :-  play_tennisPA.
                  thirstyI :>  messageA(Barman,request(beer)).
Barman:please_payE(Pr,A)   :>  messageA(Barman,paymentE(Pr,A)).
   Barman:finishedE(beer)  :>  messageA(Barman,request(coke)).
             disappointed  :-  Barman:finishedN(Pr).
```

6 Concluding Remarks

We have presented some examples of context-based commonsense reasoning in the formalism of DALI logical agents. Their ability to behave in a "sensible" way comes from the fact that DALI agents are not just reactive, but 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.

References

[Co99] 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.

[CGT02] S. Costantini, S. Gentile, A. Tocchio. DALI home page: http://gentile.dm.univaq.it/dali/dali.htm.

[CT02] S. Costantini, 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

[DST99] 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.

[Fi94] M. Fisher. A survey of concurrent METATEM – the language and its applications. In *Proc. of First International Conf. on Temporal Logic (ICTL)*, LNCS 827, Berlin, 1994. Springer Verlag.

[KS96] 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.

[PP90] Przymusinska, H., and Przymusinski, T. C., *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.

[Ra96] 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, pages 42–55, Berlin, 1996. Springer Verlag.

[Ra91] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[SPDEK00] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents.* MIT-Press, 2000.