Defining and Maintaining Agent's Experience in Logical Agents

Stefania Costantini[1]

Università di L'Aquila, Italy stefania.costantini@univaq.it

Abstract. In this paper, we extend our previous approach to memory in the DALI language from facts to (sets of) rules, and we extend their management by introducing operators for reasoning about the context and agent is involved in, and about modules that should be associated to that context in the working memory. We exploit and extend our past work where we introduced meta-axioms for run-time self-checking and self-reconfiguration and the possibility of employing sub-modules for various forms of reasoning.

1 Introduction

Agents are by definition software entities which interact with an environment, and thus are subject to modify themselves and evolve according to both external and internal stimuli, the latter due to the proactive and deliberative capabilities of the agent themselves.

A well-known and particularly difficult problem in AI is the so-called "brittleness" problem: automated systems tend to "break" when confronted with even slight deviations from the situations specifically anticipated by their designers. Brittleness and inflexibility are often attributed to rule-based systems (see, e.g., [1]) due to their supposed over-commitment to particular courses of action.

In past work, we have defined DALI (cf. [2, 3]), a logic programming agent-oriented language. In the development of DALI, we have come to understand the important role of memory in an agent's behavior, from reactivity to self-checking. We have designed for DALI a flexible management of memories [4], limited however to storing, retrieving and managing past events, that can be external or internal events, and actions performed in the past. The set of agent's memories is maintained not by means of rules as usually intended (which is, in the opinion of [1], a source of brittleness) but by means of special constraints to be dynamically checked with a certain (customizable) frequency. This approach has taken its basic inspiration from the long-termed work on memory presented in [5, 6, 7].

In order to enlarge the set of perceptions they can recognize, elaborate on and react to, and in order to expand their range of expertise, agents need to learn (either via "deep" learning or by other trusted agents [8, 9, 10]). Also, they should recognize the situation they are in at the moment (the present "context") and put different "competencies" at work. In a rule-based approach, these "competencies" can be seen as sets of rules, that can themselves evolve in time. In this paper, we extend our previous approach to memory from facts to (sets of) rules, and we extend their management by

introducing operators for reasoning about the past. In this direction we exploit our past work [9, 11] where we introduced: (i) meta-axioms for run-time self-checking and self-reconfiguration; (ii) introduced the possibility of employing sub-modules for reasoning about how to possibly react to an event and (naively) about possibility and necessity.

The paper is structured as follows: in Section 2 we discuss some issue concerning memory management in agents. In Section 3 we introduce items of our past works that we will use as building blocks in the proposed approach, that we illustrate in Section 4. Finally, in Section 5 we conclude. We assume that the reader is somewhat familiar with logic programming, and in particular with prolog-like logic programming and with answer set programming (ASP). The reader may however refer to [12] for the former and to [13] for the latter, and to the references therein.

2 Motivation: Memory

Assume an agent that is capable of remembering the received external stimuli, the reasoning process adopted and the performed actions. Through "memory", the agent is potentially able to learn from experiences and ground what it knows through these experiences [6]. The interaction between the agent and the environment can play an important role in constructing its "memory" and may affect its future behavior. Most methods to design agent memorization mechanisms have been inspired by models of human memory [14, 15] developed in cognitive science.

In 1968, Atkinson and Shiffrin [14] proposed a model of human memory which posited two distinct memory stores: short-term memory and long-term memory. This model has been suggested and further enhanced by Gero and Liew for constructive memory [5, 7] whose implementation has been presented in [5]. Memory construction [in this model] occurs whenever an agent uses past experiences in the current environment in a situated manner. In a constructive memory system, any information about the current design environment, the internal state of the agent and the interactions between the agent and the environment is used as cues in its memory construction process.

Baddeley and Hitch introduced in 1974 the notion of working memory [16], a workspace for reflective and reactive processes where explicit reasoning occurs. Items of information within the working memory are combined with the stored knowledge and experiences, manipulated, interpreted and recombined to develop new knowledge, assist learning, form goals, and support interaction with the external environment.

In fact, memory, experience, and knowledge are strongly related. Correlation between these elements can be obtained via neural networks as in [17], via mathematical models as in [18] or via logical deduction. In [7] the reader can find a nice summary with many references of cognitive studies and their applications to software agents, mainly by Gero et al. on design agents and by Laird et al. on the SOAR cognitive architecture [19, 20]. These applications are based upon a combination of a connectionistic component and some kind of software component, consisting of production rules in the case of SOAR.

In [7] it is remarked that, while the term 'memory' in computational systems often just refers to a place that holds data and information (that may be called "memories"), memory in an agent is a a reasoning process: in particular, it is the process of learn-

ing or reinforcing a concept. In agents, such memories must be associated with both the previous memories (called "experiences" when used in the current situation), and the current need for a memory (in terms of environment stimuli). Important features of the constructive memory proposed in [7] are constructive learning and experiential grounding mechanisms [6]. Constructive learning has an effect that brings changes in the structure of the memory system. Experiential grounding is concerned with the provision of meanings to the experiences processed by an artificial agent. It is similar to historical grounding, which considers the consequence of the utility of an experience in determining its meaning. In SOAR [19, 20], beyond short-term memory (graph structure) and long-term memory (production rules) there is an "episodic memory" which contains temporally ordered "snapshots" of working memory and a "semantic memory" which contains symbolic structures representing general facts, and provides the ability to store and retrieve declarative facts about the world.

To the best of our knowledge, general-purpose agent-oriented logic languages do not have coped with the problem of memory (except for specific implementations of particular applications where some "ad hoc" kind of treatment is provided). In current logic-based agent languages, the various memory components are in fact "blurred" in a general-purpose knowledge base. The only exception we are aware of is our approach of [4], summarized below, which however concerns only facts, though providing memory management by means of special constraints to be dynamically checked on need or at a certain frequency. This avoids the problem mentioned in [1] that in rule-based systems "every item which is added to memory via a rule must be maintained by other rules . . . " thus resulting, in their opinion, in brittleness and inflexibility.

In this paper, we build upon previous work in order to devise techniques for enriching DALI, but in principle any logic-based agent language, with advanced memory treatment. We do not complain in principle with the integration of connectionist and symbolic approaches. However, in this paper we deal with what can be possibly done at the symbolic level.

In the illustration of the approach we will adopt a syntax which is reminiscent of the DALI language. However, we invite the reader to consider this syntax as being by no means mandatory: it is basically aimed at illustrating on the one hand the conceptual elements of the approach and, on the other hand, how it can be put at work. A suitable variant of the syntax can be developed when applying the approach in some other practical setting. The approach is not fully implemented, but all its building blocks have been either implemented in the DALI interpreter or at least emulated by pieces of DALI software.

3 Background

3.1 Defining agent experience

Some of the authors of this paper have proposed in [2, 3] a method of correlating agent experience and knowledge by using a particular construct, the internal events, that has been introduced in the DALI language (though it can be in principle adopted elsewhere). We have defined the "static" agent memory in a very simple way as composed of the original knowledge base augmented with *past events* that record the agent's activities.

Past events can play a role in reaching internal conclusions. These conclusions, which are proactively pursued, take the role of "dynamic" memory that supports decision-making and actions: in fact, the agent can inspect its own state and its view of the present state of the world, so as to identify the better behavioral strategy in that moment. More specifically, *past events*, in our approach, record external events that have happened, internal events that have been raised and actions that have been performed by the agent. Each past event is time stamped to also record when the event has happened. Past events have at least two relevant roles: describe the agent experience; keep track of the state of the world and of its changes, possibly due to the agent intervention. With time, on the one hand past events can be overridden by more recent ones of the same kind (take for example temperature measurement: the last one is the "current" one) and, on the other hand, can also be overridden also by more recent ones of different kinds, which are somehow related.

In [4], we have extended and refined the concepts that we had introduced in the above-mentioned previous work. In particular, we introduced a set P of current "valid" past events that describe the state of the world as perceived by the agent. We also introduced a set PNV where we store all previous ones. Thus, the history H referred to in the definition of the evolutionary semantics that we introduced in [21] is the tuple $\langle P, PNV \rangle$. In practice, H is dynamically augmented with new events that happen. Let $\mathcal{Y} = (E \cup I \cup A)$ be the set of all the events (both external and internal) that may happen and the actions that the agent may perform. Each event or action in $X \in \mathcal{Y}$ may occur none or several times in the agent's life. Each occurrence is therefore indicated as $X : T_i$ where T_i is a time stamp indicating when this specific occurrence has happened (where the time stamp can be omitted if irrelevant). Each $X \in \mathcal{Y}$ is a ground term, with the customary prolog-like syntax. If one is interested in identifying which kind of event is X, a postfix (that can be omitted if irrelevant) can provide this indication. I.e., let X_E be an external event, X_A an action and X_I an internal event. As soon as X is perceived by the agent, it is recorded in P in the form ${X_P}^Y : T_i$ where P is a postfix that syntactically indicates past events and Y is a label indicating what is X, i.e., if it belongs to E, I or A. By abuse of notation for the sake of conciseness we will often omit label Y if the specific kind of event is irrelevant.

Clearly, as new "versions" of an event arrive, they should somehow "override" the old versions that have to be transferred into PNV: for instance, P will contain the most recent measure of the outside temperature, while previous measurements will be recorded in PNV. Note that past events in PNV may still have a relevant role for the entity decision process. In fact, an agent could be interested for instance in knowing how often an action has been performed or a particular stimulus has been received by the environment, or the first and last occurrences, etc. In the previous example, measurements recorded in PNV might for instance be used for computing the average temperature in a certain period of time. Clearly, PNV will have a limited size and thus eldest or less relevant events will have to be canceled. We do not cope with this issue here, while we cope with the issue of how to keep P up-to-date. Consider for example to have an agent that opens or respectively closes some kind of access. The action of opening the access can be performed only if the access is closed, and vice versa for closing.

Assume that this very simple agent believes that no external interference may occur, and thus the access is considered (by the agent) to be closed if the agent remembers to have closed it, and vice versa it is considered to be open if the agent remembers to have opened it. These "memories", in our approaches, are past events in P. Therefore, the agent will have previously closed the door (and thus it considers itself enabled to open it) if a past event such as $close_P^A : t_1$ is in P. After performing the action $open_A : t_2$, not only the past event $open_P^A : t_2$ must be inserted into P, but for avoiding possible mistakes the previous past event $close_P^A : t_1$ should be removed from P and transferred into PNV.

Past Constraints define which past events must be eliminated and under which conditions. These constraints should be automatically checked and their outcome actually applied in order to keep the agent memory consistent with the external world. Formally, we define a *Past Constraint* as follows (where we overlook the label Y indicating the kind of past event).

Definition 1 (Past Constraint). *A Past Constraint has syntax:*

$$X_{kP} : T_k, ..., X_{mP} : T_m \trianglelefteq X_{sP} : T_s, ..., X_{zP} : T_z, \{C_1, ..., C_n\}$$

where $X_{kP} : T_k, ..., X_{mP} : T_m$ are the past events which are no longer valid whenever past events $X_{sP} : T_s, ..., X_{zP} : T_z$ become known and conditions $C_1, ..., C_n$ are all true, i.e., as we will say, whenever the constraint holds*.*

For the previous example, we would have the following past constraint.

$$close_P^A : t_1 \trianglelefteq open_P^A : t_2, t_1 < t_2$$

A relevant role of past constraints is to allow one to specify which versions of analogous past events one intends to keep simultaneously in P. Assume for instance to have past events of the form $temperature_P(C, V) : T$ being C the place where the temperature is measured and V its value. One may reasonably want to move one such past event, e.g., $temperature_P(bruxelles, 22) : t1$, into PNV whenever another measurement for the same place arrives. This can be specified by means of the following past constraint:

$$temperature_P(P_1, V_1) : t1 \trianglelefteq temperature_P(P_1, V_2) : t2, t_1 < t_2$$

We define $H \star \{X_1, \ldots, X_n\}$ as the operation of adding the set of events and actions $\{X_1, \ldots, X_n\}$ to the the history of an agent. Basically, this operation corresponds to adding the new upcoming events to P and transferring past events from P to PNV according to the past constraints.

Definition 2. *Let PC be a set of past constraints and S a set of past events. By $F = PC(S)$ we indicate the result of the application of the past constraints in PC, that is F includes the left-hand side of all the constraints in PC which hold given the past events in S.*

Definition 3. *Given a history $H = \langle P, PNV \rangle$, a set of past constraints PC and a set of events $\{X_1, \ldots, X_n\}$, the result of $H \star \{X_1, \ldots, X_n\}$ is an updated history $H' = \langle P', PNV' \rangle$ where: (i) $P' = (S \cup \{X_1, \ldots, X_n\}) \setminus F$ with $F = PC(P \cup \{X_1, \ldots, X_n\})$; (ii) $PNV' = PNV \cup F$.*

Sets P and PNV and past constraints have been fully implemented within the DALI multi-agent system [22], and used in a number of applications. For instance, in the real-world application described in [23], which concerns agents that act as guides to tourists during their visit to a museum or an archeological area, sets P and PNV have been used to record the user activity and preferences in order to personalize her/his route.

Overall, P represents the set of the most recent events perceived and PNV the previous experiences that have been recorded. Thus, P constitutes a kind of short-term memory with includes the current state-of-affairs of the outside environment, to the extent to which the agent has been able to perceive it. PNV instead is a kind of long-term memory.

3.2 ASP Modules

In [24], we have proposed kinds of ASP (Answer Set Programming) modules to be invoked by a logical agents. In particular, one kind is defined so as to allow forms of reasoning to be expressed on possibility and necessity analogous to those of modal logic. In this approach, the "possible worlds" that we consider refer to an ASP program Π and are its answer sets. Therefore, given atom A, we say that A is possible if it belongs to some answer set, and that A is necessary if it belongs to the intersection of all the answer sets.

Precisely, given answer set program Π with answer sets enumerated as $M_1, \ldots, M_k$, and an atom A, the *possibility* expression $P(w_i, A)$ is deemed to hold (w.r.t. Π) whenever $A \in M_{w_i}$, $w_i \in \{1, \ldots, k\}$. The possibility operator $P(A)$ is deemed to hold whenever $\exists M \in \{M_1, \ldots, M_k\}$ such that $A \in M$. Given answer set program Π with answer sets $M_1, \ldots, M_k$, and an atom A, the *necessity* expression $N(A)$ is deemed to hold (w.r.t. Π) whenever $A \in (M_1 \cap \ldots \cap M_k)$. Possibility and necessity can possibly be evaluated within a context, i.e., if $E(Args)$ is either a possibility or a necessity expression, the corresponding *contextual* expression has the form $E(Args) : Context$ where $Context$ is a set of ground facts and rules. $E(Args) : Context$ is deemed to hold whenever $E(Args)$ holds w.r.t. $\Pi \cup Context$, where, with some abuse of notation, we mean that each atom in $Context$ is added to Π as a new fact. The answer set module T where to evaluate an operator can possibly be explicitly specified, in the form: $E(T, Args) : Context$.

In this approach, one is able for instance to define meta-axioms, like, e.g., the following, which states that a proposition is plausible w.r.t. theory T if, say, it is possible in at least two different worlds, given context C:

$$plausible(T, Q, C) \leftarrow P(T, I, Q) : C, P(T, J, Q) : C, I \neq J.$$

Another kind of ASP module that we have proposed is reactive ASP modules, defined so as to be suitable for specifying the reaction to external stimuli, where, in an invocation, the inputs include the external stimuli and the outputs include a set of actions to be executed in response to the stimuli according to the assumptions. In our view in fact, reactive ASP modules should be used to describe knowledge and beliefs concerning how an agent would cope with some events in a given situation. The answer

sets of a reactive module are meant to represent the possible courses of action that the agent might undertake whenever these events actually occur, given the present context.

Operationally, invocation of ASP modules can explicitly occur in an agent program, where the precise way to invoke a module will depend upon the agent language at hand. In DALI for instance, the simple reactive rules of the language can be used to directly resort to a reactive module whenever relevant events occur together (where DALI provides a way of specifying what it does mean to happen together for a given set of events, e.g., in the same day, same second, etc.). At invocation, reactive ASP modules can be (optionally) fed with input, representing (items of) the current agent state. The elements (facts and rules) of the input will be added to the module code.

3.3 A-ILTL

In [25] we have proposed an extension to the well-known LTL Linear Temporal Logic [26, 27, 28] called A-ILTL, for "Agent-Interval LTL", which is tailored to the agent's world in view of run-time verification.

Based on this new logic, we are able to enrich agent programs by means of A-ILTL rules. These rules are defined upon a logic-programming-like set of formulas where all variables are implicitly universally quantified. They use operators over intervals that are reminiscent of LTL operators.

During the agent life, each A-ILTL rule is attempted at a certain frequency and with certain priorities (possibly customizable by means of directives). If the current state of affairs satisfies every A-ILTL rule, then no action is required. Otherwise, some kind of repair action has to be undertaken with respect to the violated A-ILTL rule. Below we report the definition of such a rule.

Definition 4. *An A-ILTL rule with a repair is a rule the form:*
$OP(M, N; K)\varphi :: \chi \div \psi$*, where:*

- $OP(M, N; K)\varphi :: \chi$ *is a contextual A-ILTL rule;*
- ψ *is called the* repair action *of the rule.*

In particular, OP is a temporal operator which is required to hold in the interval between time instants M and N on formula φ evaluated in the context χ (where the context is aimed at providing values for variables occurring in φ). The rule is supposed to be checked at run-time at frequency K (if K is omitted, at a default frequency). If a violation is detected, ψ is executed which is supposed to be a procedure that determines suitable self-modifications so as to cope with the unwanted situation and/or improve future agent's behavior. The repair action is specified via an atom that is executed as an ordinary goal.

The A-ILTL rule in the following example monitors the achievement of goals, and specifies that, in case of violation (some goal, though not achieved, has been dropped), the present level of commitment of the agent to its objectives has to be increased. This can be specified as:

$$NEVER_{m,n}\,(not\ achieved(G), dropped(G)\,) :: \\ (goal(G), deadline(G, T), NOW(T1), T1 \leq T) \div inc\_comt(T1)$$

Suppose that at a certain time t the monitoring condition $NEVER\,(not\ achieved(g), dropped(g))$ is violated for some goal g. Upon detection of the violation, the system will attempt the repair action consisting in executing the goal $?{-}inc\_comt(t)$. Its execution will allow the system to perform the specified run-time re-arrangement of the program that attempts to cope with the unwanted situation.

3.4 Learning by rule exchange

In [8, 10], we have introduced an approach to learning centered on the possibility of acquiring sets of rules from other agents, namely "learning by being told". In the approach, agents do not blindly incorporate the new knowledge. Rather, they evaluate how useful the new knowledge is, and on this basis decide whether to keep or discard it. The approach is based upon maintaining a meta-level description of the acquired (sets of) rules, that associates to the acquired knowledge a specific objective and (possibly) a set of conditions, including a time limit. The purpose is that the new rules should help the agent reach that objective and fulfill the conditions within the given time limit. After a while, the agent will evaluate by means of meta-reasoning performed at a meta-control layer whether (or to what extent) this has been achieved. If the evaluation is unsatisfactory, then the new knowledge will be discarded or possibly deactivated for future re-trial in a modified context. There is a clear similarity between our approach and reinforcement learning, where here the action that is to be evaluated is the use of the new knowledge.

4 Advanced Memory Management in Logical Agents

In the overall framework that we have outlined up to now (based on DALI but easily adaptable to other logic languages for agents), and embracing the view of [7] where memory is a reasoning process, we can set the following scenario. The semantic memory (say S) coincides with the agent's initial knowledge base. The short-term memory is constituted by set P. The set P plus the A-ILTL axioms constitute the working memory. The long-term memory is constituted by set PNV.

The first new element that we intend to introduce is a notion concerning the *context* an agent is presently involved in. For instance, if you consider an agent that is able to play a number of games (say, e.g., chess, poker, and others, and even gamble on the stock exchange as a particular kind of game), the context includes the game the agent is presently playing (if any). The context need not be a single one. For instance, the agent can be at leisure or at work, at home or somewhere else, and either in a normal or in an emergency situation, and so on.

We assume that, like in DALI, the various agent activities may in principle proceed in parallel (where in the DALI interpreter they are actually interleaved, also according to priorities). Thus for instance, playing a game and answering the phone may occur in combination.

We may simply assume to represent context by means of a set of (meta-)facts concerning a distinguished predicate $context$ (again, we expect the reader not to stick on the

syntax we adopt here). The version $contextN$ of this predicate will define the present context (by adapting the DALI notion of present events). This version will become, when no longer actual, a time-stamped past event with predicate $contextP$. A sample context definition can be the following:

$$context(at\_leisure, at\_work, play\_chess, normal, emergency, \ldots).$$
$$contextN(play\_chess).$$
$$contextN(at\_leisure).$$
$$contextN(normal).$$

where the first fact lists all possible contexts, and the following ones the contexts that are active at the moment. More generally, it is required that the agent initial program contains a fact, that we call *context declaration*, of the form:

$$context(PC_1, \ldots, PC_r).$$

where we call DC the set $PC_1, \ldots, PC_r$, and we call the PC_is, which are either constants or ground terms, *contexts*. This declaration states which are the contexts the agent may possibly find itself in, or may choose to set itself in. Facts $contextN$ may be initially stated, but this is not strictly required as the agent may want to set a context later. We let CN be the set of contexts occurring in facts $contextN(C)$ which are present in the knowledge base.

In our setting, in order to either set or change context, the agent may invoke a distinguished action (postfix A):

$$change\_contextA(M_1, \ldots, M_k; C_1, \ldots, C_n).$$

where: (a) each M_j is an atom and each C_i can be either an atom or a disjunction of atoms in the form:

$$(C_{i_1} \mid \ldots \mid C_{i_s}) \ : \ Prefs$$

b) all the M_js, C_i and C_{i_r}s occur in DC.

The part : $Prefs$ is optional, and expresses in the notation of [29, 30] preferences about which of the C_{i_j}s is preferred under which conditions. The intended meaning is that the agent wishes to switch to contexts $M_1, \ldots, M_k, C_1, \ldots, C_n$ where: (i) the M_js are mandatory, i.e., after the context switch all of them must be present contexts; (ii) the C_is are wished for, i.e., they will become present contexts if possible; (iii) for each C_i which is a disjunction, any of the C_{i_j}s can be selected, either indifferently or according to preferences, if stated. For instance,

$$change\_contextA(at\_leisure; (play\_chess \mid play\_checkers \ : \ less\_difficult)).$$

means that the agent intends to switch to a context where it is at leisure and wishfully plays either chess or checkers, preferring the one which is less difficult. Binary predicate $less\_difficult(X, Y)$ must be defined in the agent's knowledge base where whenever $less\_difficult(g_1, g_2)$ holds, $g_1, g_2 \in \{play\_chess, play\_checkers\}$, the former is best preferred. This notation extends to sets of elements and to any binary predicate which establishes an ordering over these elements.

A distinguished ASP module is supposed to be defined in order to manage context shift. This module is similar to a "reactive ASP module" as defined in [24]. An ASP module is needed (that may have no, one or several answer sets) as there can be either complementarity or incompatibilities among contexts: for instance, one cannot be both at leisure and at work, does not play games if in emergency, and, say, one gambles on the stock exchange only when working. The definition of this module is therefore an important part of the definition of an agent. A possible (naive) definition of this module related to our example can be for instance the following (where we remind the reader that rules starting with :- are *constraints* and state that their conditions cannot simultaneously hold, and that an *even cycle* like a:- b, b:- a generates indifferently either a or b.

```
at_leisure:- not at_work.
at_work:- not at_leasure.
normal:- not emergency.
emergency:- not normal.
:- emergency, game.
game:- play_chess.
game:- play_checkers.
play_check:- not play_checkers.
play_checkers:- not play_chess.
:- at_leisure,at_work.
:- normal,emergency.
```

Such a module, that we can call *context-switch module*, may have none, one or more answer set. We say that a context switch is *enabled* if the context-switch module admits answer sets. If so, we will say that an answer set M of the context-switch module *entails* $change\_contextA(M_1, \ldots, M_k; C_1, \ldots, C_n)$ if for every M_j, $j \leq k$, $M_j \in M$, and that M *enables* each C_i which is an atom if $C_i \in M$, and each C_i which is a disjunction, if at least one element C_{i_j} of the disjunction is in M.

In our prototype implementation, the interpreter treats the action of context change $CC = change\_contextA(M_1, \ldots, M_k; C_1, \ldots, C_n)$ as follows, where as said CN is the set of facts of the form $contextN(C)$ included in the knowledge base.

- The context-switch ASP module is invoked, with the M_js as input. I.e., the M_js will be added to the module as new facts.
- If the resulting module has no answer set, then the action fails and a failure past event will be generated, that can possibly be suitably managed by a DALI internal event.

– If the resulting module admits answer sets, than it entails CC by construction (as the M_js are added to the module). One answer set $\hat{M}$ is selected in the following way.
  - The best preferred answer sets $N_1, \ldots, N_k$ are chosen according to the preferences expressed in the CC elements[1].
  - Among the N_vs, the answer set $\hat{M}$ is chosen that maximizes the intersection with CN. I.e., as few changes as possible are performed (other strategies are of course possible). If more than one of the N_vs fulfils the requirements, one of them is nondeterministically chosen.
– All facts of the form $contextN(C)$ presently included in the knowledge base are removed, and corresponding facts $contextP(C) : T$ are added to PNV, where T is the present time.
– For each $C \in \hat{M} \cap DC$, a new fact of the form $contextN(C)$ is added to the knowledge base.

In the proposed setting, each context is associated to one or more modules (intended as set of rules) aimed at managing the situation the context is about. These modules can be part of the agent program, can have been acquired from other agents (see [8, 10]) or even somehow synthesized by the agent itself via some form of learning. However, following [10], we assume that such modules are kept in a meta-level format enriching each module with additional information. Let for instance a possible form be the following, where let $\ulcorner M \urcorner$ be any meta-level representation of the module M (that we take here to be a set of DALI rules).

$$mod(context(c), name(n), source(a), time(t_1), goal(g), timeout(t_2), \\ feature(f), eval(v), active(b), lastused(t_3), \ulcorner M \urcorner)$$

The elements of the description have the following meaning:

– $context(c)$ indicates which context the module is aimed at managing. c is a constant, might in principle be a list. $name(n)$ specifies the module name.
– $source(a)$ indicates which agent the module has been acquired from. a is the agent's name, can be *self* if the module is part of the agent program. $time(t_1)$ is the time of acquisition, can be 0 if the module is part of the agent program from the beginning.
– $goal(g)$ is the goal that the module is aimed at reaching. E.g., for a module with context *chess* the goal can be, e.g., *win-game* or *teach-to-play*, where $timeout(t_2)$ states the deadline for reaching the goal. The goal may be empty or can be in principle a list, the timeout may be empty. $feature(f)$, if specified (f might be a list), may express some refinement with respect to the context and the goal. For instance, for any game where the goal is to win, the feature might be the level of expertise at which the module is supposed to enable the agent to play.

[1] Fundamental techniques for combining preferences (seen as generic binary relations) can be found for instance in [31]. Regarding combination of preferences in Logic Programming, criteria are also given, for instance, in [32, 33, 34, 35, 29].

- $eval(v)$ is some kind of rating associated to the module, that should be related to "how good" the module has been in reaching the goal in past usages. The evaluation can be inherited by sender agent in case of acquired knowledge, and/or can be updated by the agent itself.
- $active(b)$ states whether the module is presently in use or not, if not $lastused(t_3)$ states the last time of module usage.

The set of module descriptions will include none, one or more module(s) for each possible context. In our view, these descriptions are part of the long-term memory. We also enhance the definition of the working memory, that in our setting at this point will be composed not only by the set P plus the A-ILTL axioms, but also by the context declarations, and by one module for each present context. This module will be loaded whenever a context enters into play.

Then, when updating contexts, two more actions have to be performed:

- Eliminate from working memory modules concerning context which are removed, and update the corresponding descriptions in the fields $active$, which is set to false, $lastused$, that is set to the time of removal, $eval$ that can be updated according to agent's satisfaction (we do not treat this aspect here).
- Load into the working memory one module for each new context. In the extreme case where no module is available, a request might be issued to sibling agents (this aspect has been discussed at some length in [8, 10]).

It remains to be seen what to do if there is more than one module corresponding to one context. The different modules may correspond to different goals or to the same goal with different features. E.g., for the context *poker* the goals might be either *win-money* or *minimize-loss* and the features might be for instance either *low-risk* or *high-risk*, referring to the style and attitude of the player.

To this aim, we can improve the $change\_contextA$ format. Precisely, each C_i (or each C_{i_j} for elements of disjunctions can be of the form:

$$C_i(Goal, Feature).$$

e.g., in the above example, *poker(win-money,high-risk)*. The specification of goal and feature is to be intended as optional, but can also be enriched to:

$$C_i(Goal, Feature\_pref).$$

where $Feature\_pref$ expresses preferences about features, e.g., in the above example, *poker(win-money, low-risk* $>$ *high-risk* $pref\_when$ *short-money)*, stating (in the style of [29]) that in the case *short-money* is entailed by the present knowledge base, *low-risk* should be preferred as a feature to *high-risk*, in case both modules are available (otherwise, the choice is indifferent). Another possible choice is whether one might select (given goal and features) the most recent or the best evaluated module.

What would it happen in case the agent loads a module in the view of a goal, but the goal is not reached within the given deadline? Clearly, the evaluation of the module

should be affected accordingly. However, in case alternative modules are available, the agent might wish to remain in the context where it is, but exchange the module "on the fly". This can be obtained by means of a suitable A-ILTL axiom. For simplicity, let us assume that whenever loading a context C and a related module, an object-level fact is created of the following form:

$$module(C, N, Goal, Timeout, Feature).$$

where N is the module name and the other fields (possibly empty) represent goal, time-out and features as extracted from the module definition. The above fact is to be removed on removal of the module. Below is an A-ILTL axiom that, whenever checked at run-time (at a customizable frequency), if it finds that a module has failed its objective then it replaces it with another one (if available).

$$NEVER\,(module(C, N, Goal, Timeout, Feature),\\ not\ achieved(Goal), expired(Timeout)) \div\\ replace\_module(C, N, Goal, Timeout, Feature)$$

Or, in the case an agent wishes to update the feature, e.g., passing from beginner to expert in some game, let us assume it asserts a fact $new\_feature(C, Goal, F)$. The following A-ILTL axiom would perform a module exchange (if a suitable module is available) whenever checked:

$$NEVER\,(module(C, N, Goal, Timeout, Feature),\\ new\_feature(C, Goal, F), F \neq Feature) \div\\ update\_module(C, Goal, F)$$

5 Concluding Remarks

In this paper, we have presented a context-sensitive approach to managing memory and memories in logical agents. The approach was born in the context of the DALI language, but can be easily adapted to other logic-programming based agent-oriented languages. To the best of our knowledge the proposed approach, though in its early stage, is a novelty in the logical agents realm. It drew inspiration from related work in Artificial Intelligence, yet it introduces original aspects.

A full implementation is still lacking, but all the proposed features have been simulated and experimented in DALI. Much remains to done for refining and extending the approach, and for completing the implementation.

References

[1] research group, S.: SOAR: A comparison with rule-based systems (2010) URL: http://sitemaker.umich.edu/soar/home.

[2] Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
[3] Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
[4] Costantini, S., Dell'Acqua, P., Pereira, L.M., Tocchio, A.: Ensuring agent properties under arbitrary sequences of incoming events. In: Proc. of 17th RCRA Intl. Worksh. on Experimental evaluation of algorithms for solving problems with combinatorial explosion. (2010)
[5] Liew, P., Gero, J.S.: An implementation model of constructive memory for a situated design agent. In: Proc. of the Intl. Conf. on Agents in Design. (2002) 257–276
[6] Liew, P., Gero, J.S.: Constructive memory for situated design agents. AIEDAM, Artificial Intelligence for Engineering Design, Analysis and Manufacturing
[7] Gero, J.S., Peng, W.: Understanding behaviors of a constructive memory agent: A markov chain analysis. Knowledge-Based Systems
[8] Costantini, S., Tocchio, A.: Learning by knowledge exchange in logical agents. In: From Objects to Agents: Intelligent Systems and Pervasive Computing, Proc. of WOA'05. (2005) ISBN 88-371-1590-3.
[9] Costantini, S., Dell'Acqua, P., Pereira, L.M.: A multi-layer framework for evolving and learning agents. In: Proc. of the AAAI-08 Workshop on Metareasoning: Thinking about Thinking, Stanford University, AAAI Press (2008)
[10] Costantini, S., Dell'Acqua, P., Pereira, L.M.: Conditional learning of rules and plans by knowledge exchange in logical agents. In: Proc. of RuleML 2011 at IJCAI. (2011)
[11] Costantini, S., Dell'Acqua, P., Pereira, L.M., Tsintza, P.: Runtime verification of agent properties. In: Proc. of the Int. Conf. on Applications of Declarative Programming and Knowledge Management (INAP09). (2009)
[12] Apt, K.R., Bol, R.: Logic programming and negation: A survey. The Journal of Logic Programming **19-20** (1994) 9–71
[13] Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, Chapter 7. Elsevier (2007)
[14] Pearson, D., Logie, R.H.: Effect of stimulus modality and working memory load on menthal synthesis performance. Imagination, Cognition and Personality **23**(2-3) (2004) 183–191
[15] Logie, R.H.: Visuo-Spatial Working Memory. Psychology Press, Essays in Cognitive Psychology (1994)
[16] Pearson, D., Logie, R.H.: Working memory and mental synthesis (2000)
[17] Liew, P.S., Gero, J.S.: Constructive memory for situated design agents. AI EDAM: Artificial Intelligence for Engineering Design, Analysis, and Manufactoring **18**(2) (2004) 163–198
[18] Lerman, K., A. Galstyan, A.: Agent memory and adaptation in multi-agent systems. In: AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2003) 797–803
[19] Laird, J., Newell, A., Rosenbloom, P.: SOAR: An architecture for general intelligence. Artificial Intelligence **33**(1) (1987) 1–64
[20] Laird, J.E.: Extending the SOAR cognitive architecture. In: Proc. of the First Artificial General Intelligence Conf. (2008) 224–235
[21] Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. LNAI 3904. Springer-Verlag (2006) 106–123
[22] Costantini, S., D'Alessandro, S., Lanti, D., Tocchio, A.: Dali web site, download of the interpreter (2010) http://www.di.univaq.it/stefcost/Sito-Web-DALI/WEB-DALI/index.php,

With the contribution of many undergraduate and graduate students of Computer Science, L'Aquila. For beta-test versions of the interpreter (latest advancements) please ask the authors.

[23] Costantini, S., Mostarda, L., Tocchio, A., Tsintza, P.: Dalica agents applied to a cultural heritage scenario. IEEE Intelligent Systems, Special Issue on Ambient Intelligence **23**(8) (2008)

[24] Costantini, S.: Answer set modules for logical agents. In Gottlob, G., ed.: Datalog 2.0. LNCS. Springer (2011) Forthcoming.

[25] Costantini, S., Dell'Acqua, P., Pereira, L.M.: A multi-layer framework for evolving and learning agents. In M. T. Cox, A.R., ed.: Proceedings of Metareasoning: Thinking about thinking workshop at AAAI 2008, Chicago, USA. (2008)

[26] Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. Acta Informatica **20** (1983) 207–226

[27] Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, vol. B. MIT Press (1990)

[28] Lichtenstein, O., Pnueli, A., Zuch, L.: The glory of the past. In: Proc. Conf. on Logics of Programs. LNCS 193, Springer Verlag (1985)

[29] Costantini, S., Formisano, A.: Modeling preferences and conditional preferences on resource consumption and production in ASP. Journal of of Algorithms in Cognition, Informatics and Logic **64**(1) (2009)

[30] Costantini, S., Formisano, A., Petturiti, D.: Extending and implementing RASP. Fundamenta Informaticae **105**(1-2) (2010) 1–33

[31] Andréka, H., Ryan, M., Schobbens, P.Y.: Operators and laws for combining preference relations. Journal of Logic and Computation **12**(1) (2002) 13–53

[32] Balduccini, M., Mellarkod, V.S.: CR-Prolog$_2$ with ordered disjunction. In De Vos, M., Provetti, A., eds.: Proceedings of the ASP'03 Workshop. Volume 78 of CEUR Workshop Proceedings. (2003)

[33] Brewka, G., Niemelä, I., Syrjänen, T.: Logic programs with ordered disjunction. Computational Intelligence **20**(2) (2004) 335–357

[34] Brewka, G.: Complex preferences for answer set optimization. In: Proc. of KR'04. (2004)

[35] Son, T., Pontelli, E.: Planning with preferences using logic programming. Theory and Practice of Logic Programming **6**(5) (2006) 559–607