Modeling Agents in a Logic Action Language

Matteo Baldoni[1], Laura Giordano[2], Alberto Martelli[1] and Viviana Patti[1]

[1]Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (ITALY)
E-mail: {baldoni, mrt, patti}@di.unito.it

[2]Dipartimento di Scienze e Tecnologie Avanzate — DSTA
Università del Piemonte Orientale "A. Avogadro"
Corso Borsalino 54 - 15100 Alessandria, ITALY.
e-mail: laura@di.unito.it

1 Introduction

In this paper we describe how a theory for reasoning about actions can be used for specifying agents and for executing agent specifications. In fact, reasoning about the effects of actions in a dynamically changing world is one of the main problems which must be faced by intelligent agents. Furthermore, the internal dynamics of the agent itself can be regarded as resulting from the execution of actions on the mental state. The agent mental state contains all its basic attitudes, as its belief, desires and intentions in the BDI model [13].

The action theory we adopt is a modal action theory, which has been developed in [8, 9]. It provides a non-monotonic solution to the frame problem by making use of abductive assumptions and it deals with the ramification problem by introducing a "causality" operator.

The adoption of Dynamic Logic or a modal logic to deal with the problem of reasoning about actions and change is common to many proposals, as for instance [6, 12, 5, 14, 8], and it is motivated by the fact that modal logic allows a very natural representation of actions as state transition, through the accessibility relation of Kripke structures. Since the intentional notions (or attitudes), which are used to describe agents, are usually represented as modalities, our modal action theory is also well suited to incorporate such attitudes.

In the following we will recall the definition of the action language, referring to the proposals in [8, 9, 1], and of its extension to deal with complex actions, as proposed in [2, 4]. The formalization of complex actions draws considerably from dynamic logic [11]. As a difference, rather than referring to an Algol-like paradigm for describing complex actions as in Golog [10], it refers to a Prolog-like paradigm: instead of using the iteration operator, complex actions are defined through (possibly recursive) definition, given by means of Prolog-like clauses. In this context, the nondeterministic choice among actions is allowed by alternative clause definitions.

This action theory has a computational counterpart in a logic programming language [1] which in [2, 4] has been extended to deal with complex actions and with knowledge producing actions. The logical characterization of the action language is rather simple and very close to the procedural one. In [1, 2, 4] we have addressed the problem of reasoning about possible courses of actions and we have introduced a goal directed proof procedure to solve the temporal projection problem. This paper will focus on an implementation of the language, called **Dylog**, and will describe its use for specifying agent behaviours. Dylog can be used as an ordinary programming language for executing procedures which model the behaviour of an agent, but also to reason about them, by extracting from them linear or conditional plans.

2 The action language

2.1 Primitive actions

In the action language each primitive action $a \in A$ is represented by a modality $[a]$. The meaning of the formulas $[a]\alpha$ is that α holds after any execution of action a. The meaning of the formula $\langle a\rangle\alpha$ is that there is a possible execution of action a after which α holds. We also introduce a modality $\Box$, which is used to denote those formulas that hold in all states, that is, after any action sequence. The intended meaning of a formula $\Box\alpha$ is that α holds after any sequence of actions.

A state consists of a set of *fluents*. The *simple action laws* are rules that allow to describe direct and indirect effects of primitive actions on a state. In particular, simple action clauses consist of *action laws*, *precondition laws*, and *causal laws*.

Action laws define direct effects of primitive actions on a fluent and allow actions with conditional effects to be represented. They have the form

$$\Box(Fs \rightarrow [a]F)$$

where a is a primitive action name, F is a fluent, and Fs is a fluent conjunction, meaning that action a has effect on F, when executed in a state where the *fluent preconditions* Fs hold.

Precondition laws allow *action preconditions*, i.e. those conditions which make an action executable in a state, to be specified. Precondition laws have form

$$\Box(Fs \rightarrow \langle a\rangle true)$$

meaning that when the fluent conjunction Fs holds in a state, execution of the action a is possible in that state.

Causal laws are used to express causal dependencies among fluents and, then, to describe *indirect* effects of primitive actions. They have the form:

$$\Box(Fs \rightarrow F)$$

meaning that the fluent F holds if the fluent conjunction Fs holds too[1].

In Dylog we have adopted a more readable notation: action laws have the form "a **causes** F **if** Fs" precondition laws have the form "a **possible_if** Fs" and causal laws have the form "F **if** Fs".

Let us consider a simple example of a robot which moves in an environment consisting of two rooms, 1 and 2, each one connected to a corridor. Inside each room there is a light which can be on or off. The robot can be in a room or at one among three positions in the corridor: at(1), in front of room 1, at(2), in front of room 2, or at(3). The fluents are:

$$in\_room(1), in\_room(2), at(1), at(2), at(3), light\_on(1), light\_on(2).$$

The first five fluents are mutually exclusive. This can be expressed by a set of causal rules like:

$\neg in\_room(1)$ **if** $at(1)$
$\neg at(2)$ **if** $at(1)$

and so on.

Some of the actions of the robot are the following:

up : go up one step in the corridor
up **possible_if** $at(I), I < 3$
up **causes** $at(J)$ **if** $at(I), J$ **is** $I + 1$[2]

getin(i) : get in room i
$getin(I)$ **possible_if** $at(I)$
$getin(I)$ **causes** $in\_room(I)$

[1] In a logic programming context we represent causality by the directionality of implication. A more general solution, which makes use of modality "causes", has been provided in [9].

[2] Dylog can call any Prolog goal.

toggle(i): toggles the switch in room i
$toggle(I)$ **possible_if** $in\_room(I)$
$toggle(I)$ **causes** $light\_on(I)$ **if** $\neg light\_on(I)$
$toggle(I)$ **causes** $\neg light\_on(I)$ **if** $light\_on(I)$

An action can be executed in a state s if the preconditions of the action hold in s. The execution of the action modifies the state according to the action and causal laws. Furthermore we assume that the value of a fluent persists from one state to the next one, if the action does not cause the value of the fluent to change.

In general, the execution of an action will have an effect on the environment, such as for instance moving a robot or sending a message. This can be specified in Dylog by associating with each primitive action some Prolog code which implements the effects of the action on the world (in our case it could be the call to a robot simulator). Therefore, when an agent executes an action it must commit to it, and it is *not allowed to backtrack* by retracting the effects of the action. This is the main difference with respect to the use of the language for reasoning about actions, where the agent can do hypothetical reasoning on possible sequences of actions by exploring different alternatives.

2.2 Procedures

Procedures define the behavior of *complex actions*. Complex actions are defined on the basis of other complex actions, primitive actions, and *test* actions. Test actions are needed for testing if some fluent holds in the current state and for expressing conditional complex actions. They are written as "*Fs*?", where *Fs* is a fluent conjunction. A *procedure* is defined as a collection of *procedure clauses* of the form

$$p_0 \ \textbf{is} \ p_1, \ldots, p_n \ \ (n \geq 0)$$

where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either a primitive action, or a test action, or a procedure name (i.e. a procedure call)[3]. Procedures can be recursive and can be executed in a goal directed way, similarly to standard logic programs.

In our modal action theory, procedure clauses have to be regarded as axiom schemas of the logic. More precisely, each procedure clause p_0 **is** $p_1, \ldots, p_n$, can be regarded as the following axiom schema[4]

$$\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi \supset \langle p_0 \rangle \varphi.$$

Its meaning is that if in a state there is a possible execution of p_1, followed by an execution of p_2, and so on up to p_n, then in that state there is a possible execution of p_0.

Procedures can be used to describe the behaviour of an agent. In particular we assume the behaviour of a rational agent to be driven by a set of goals. For each goal, the agent has a set of procedures (sometimes called plans) for achieving the given goal.

The behaviour of the robot of our example can be described by giving a collection of procedures for achieving various goals. For simplicity we define only one procedure, $achieve(G)$, whose parameter G is the goal to be achieved. Goals are fluents, and the structure of the procedure is based on that of the primitive actions. For instance, if goal G is caused by action a whose preconditions are p, then to achieve G we try to achieve p and then execute a.

$achieve(G)$ **is** $G?$
$achieve(at(I))$ **is** $at(J)?, J > I?, down, achieve(at(I))$
$achieve(at(I))$ **is** $at(J)?, J < I?, up, achieve(at(I))$
$achieve(at(I))$ **is** $in\_room(J)?, getout(J), achieve(at(I))$
$achieve(in\_room(I))$ **is** $achieve(at(I)), getin(I)$
$achieve(\neg light\_on(I))$ **is** $light\_on(I)?, achieve(in\_room(I)), toggle(I)$

[3] Actually in Dylog p_i can also be a Prolog goal.

[4] These axioms have the form of rewriting rules as in grammar logics. In [3] decidability results for subclasses of grammar logics are provided, and right regular grammar logics are proved to be decidable.

As pointed out before, whenever the interpreter executes an action, it commits to that action, and cannot backtrack. Thus procedures are deterministic, or, at most, they can implement a kind of don't care non determinism.

For instance, if the initial state is:

$$\neg at(1), \neg at(2), at(3), \neg in\_room(1), \neg in\_room(2), light\_on(1), light\_on(2)$$

the call to $achieve(\neg light\_on(1)))$ will execute the sequence of actions: $down, down, getin(1), toggle(1)$.

The above formulation of the behaviour of the agent has many similarities with agent programming languages based on the BDI paradigm such as dMARS [7]. As in dMARS, plans are triggered by goals and are expressed as sequences of primitive actions, tests or goals. The main difference is in the control of the interpreter. In our case we rely on a standard sequential Prolog interpreter and we assume to have only one goal at a time to achieve, whereas in dMARS it is possible to interleave the execution of several plans[5].

3 Planning

Up to now we have assumed that Dylog procedures which specify the behaviour of an agent are executed by an interpreter as in usual programming languages. However a rational agent must also be able to cope with complex or unexpected situations, by reasoning about the effects of a procedure before executing it.

In general, a *planning problem* amounts to determine, given an initial state and a goal Fs, if there is a sequence of actions that, when executed in the initial state, leads to a state in which Fs holds. In our context, in which complex actions are represented as procedures, we can consider a specific instance of the planning problem in which we want to know if there is a possible execution of a procedure p leading to a state in which some condition Fs holds. In such a case the execution sequence is not an arbitrary sequence of atomic actions but it is an execution of p. This can be formulated by the query

$$\langle p \rangle Fs$$

which asks for a terminating execution of p (i.e. a finite action sequence) leading to a state in which Fs holds.

The execution of the above query returns as a side-effect an answer which is an *execution trace* "$a_1, a_2, \ldots, a_m$", i.e. a primitive action sequence from the initial state to the final one, which represents a *linear plan*.

To achieve this, Dylog provides a metapredicate *plan(p, Fs, as)*, where p is a procedure, Fs a goal and as a sequence of primitive actions. The procedure p can be nondeterministic, and *plan* will extract from it a sequence as of primitive actions, a *plan*, corresponding to a possible execution of the procedure, leading to a state in which Fs holds, starting from the current state. Procedure *plan* works by executing p in the same way as the interpreter of the language, with a main differences: primitive actions are executed without any effect on the external environment, and, as a consequence, they are backtrackable.

Instead of simply obtaining any plan, procedure *plan* could extract a *best plan*, according to some optimality criterion. In our case, *plan* uses iterative deepening to find the plan with the minimal number of actions, but it might be modified to make use of some domain dependent heuristics.

Let us assume that in our example the robot may have the goal of switching off all lights.

$achieve(all\_lights\_off)$ **is** $achieve(\neg light\_on(1)), achieve(\neg light\_on(2))$
$achieve(all\_lights\_off)$ **is** $achieve(\neg light\_on(2)), achieve(\neg light\_on(1))$

[5] However this limitation can be overcome by providing an explicit representation of goals as literals, in the same way as we suggest later for belief literals.

i.e. lights can be switched off in any order.

Let us assume that the robot is given the composite goal

$$? - achieve(all\_lights\_off), achieve(at(1))$$

meaning "turn off all lights and then go to 1". By applying the planning procedure to this goal, starting from the initial state given above, Dylog will return the sequence of primitive actions *down*, $getin(2)$, $toggle(2)$, $getout(2)$, *down*, $getin(1)$, $toggle(1)$, $getout(1)$, which can subsequently be executed by the robot. On the other hand, if the query $achieve(all\_lights\_off), achieve(at(1))$ is directly executed, the robot will commit to the first rule for achieving all_lights_off and will execute a longer sequence of actions, by going first to room 1 and then to room 2.

In general, planning before acting is required when the agent has to achieve a conjunction of goals which may interact.

A planning problem in Dylog is restricted to a specific procedure. However, we can model the usual planning problem by defining a general procedure:

p **is** $a_1; p.$
p **is** $a_2; p.$
...
p **is** $a_n; p.$
p **is** $(true)?.$

whose execution traces are all possible sequences of primitive actions.

4 Sensing

In general we cannot assume that the value of each fluent in a state is known to an agent, and we want to be able of representing the fact that some fluents are unknown and to reason about the execution of actions on incomplete states. If we want to represent explicitly the unknown value of some fluents, we need to introduce an epistemic level in our representation language. Hence in [4] we introduce an epistemic operator $\mathcal{B}$, to represent the beliefs an agent has on the world: $\mathcal{B}f$ will mean that the fluent f is known to be true, $\mathcal{B}\neg f$ will mean that the fluent f is known to be false. Fluent f is undefined in the case both $\neg\mathcal{B}f$ and $\neg\mathcal{B}\neg f$ hold. In the following, we will write $u(F)$ for $\neg\mathcal{B}f \wedge \neg\mathcal{B}\neg f$.

In Dylog we do not use explicitly the epistemic operator $\mathcal{B}$, but we extend our notation with the test $u(F)?$. Thus each fluent can have one of the three values: true, false or unknown.

An agent will be able to know the value of F by executing a *sensing* action. In our example, we introduce a sensing action $sense\_light(I)$, for knowing whether the light in room I is on or off. This action can be executed if the robot is in room I:

$sense\_light(I)$ **possible_if** $in\_room(I)$
$sense\_light(I)$ **senses** $light\_in(I)$

We can now extend the procedure *achieve* of the agent by introducing a new kind of goal, $KW(F)$, where KW stands for *knows whether*, which is satisfied in a state where the value of F is known.

$achieve(KW(light\_on(I)))$ **is** $light\_on(I)?$
$achieve(KW(light\_on(I)))$ **is** $\neg light\_on(I)?$
$achieve(KW(light\_on(I)))$ **is** $u(light\_on(I))?, achieve(in\_room(I)), sense\_light(I)$

Furthermore, the following rule will be added:

$achieve(\neg light\_on(I))$ **is** $u(light\_on(I))?, achieve(KW(light\_on(I))), achieve(\neg light\_on(I))$

By applying Dylog's planning predicate *plan* to a procedure containing sensing actions, we obtain a *conditional plan* whose branches correspond to the different outcomes of sensing actions.

For instance, if the robot does not know the value of fluent $light\_on(1)$ in the initial state, the execution of *plan* with the previous query $achieve(all\_lights\_off), achieve(at(1))$ will return the plan

$down$, $getin(2)$, $toggle(2)$, $getout(2)$, $down$, $getin(1)$, $sense\_light(1)$, ($light\_on(1)?$, $toggle(1)$, $getout(1)$ **or** $\neg light\_on(1)?$, $getout(1)$)

5 Final remarks

Dylog can be seen both as a specification language and as an agent programming language. Its interpreter is a straightforward implementation of the proof procedure, and thus Dylog appears as a promising candidate to reduce the gap between theory and practical use of agent programming languages.

In this paper we have shown how to program an agent in a goal directed way. Other formulations are of course possible. For instance we might benefit from the modal approach by introducing explicit modalities for belief, goal and intentions, in the style of BDI theories. On the other hand we might formulate the behaviour of a "reactive" agent as follows:

p **is** $cond_1?, a_1, p$
p **is** $cond_2?, a_2, p$
...

Presently Dylog is used for building web applications which require planning capabilities. Extensions of the language with communicative actions are under way.

Dylog has been implemented in Sicstus Prolog, and is available at `http://www.di.unito.it/~alice`.

References

[1] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Proc. of the 2nd International Workshop on Non-Monotonic Extensions of Logic Programming, NMELP'96*, volume 1216 of *LNAI*, pages 132–150. Springer-Verlag, 1997.

[2] M. Baldoni, L. Giordano, A. Martelli and V. Patti. Modal programming language for representing complex actions. In *Proc. DYNAMICS'98: Transactions and Change in Logic Databases. Technical Report MPI-9808*, pages 1–15, 1998.

[3] M. Baldoni, L. Giordano, and A. Martelli. A Tableau Calculus for Multimodal Logics and Some (Un)Decidability Results. In H. de Swart ed., *Proc. TABLEAUX'98*, LNAI 1397, pages 44–59, 1998.

[4] M. Baldoni, L. Giordano, A. Martelli and V. Patti. Reasoning about Complex Actions with Incomplete Knowledge: a Modal Approach. Technical Report 53/2000, Dipartimento di Informatica, University of Torino, 2000.

[5] M. Castilho, O. Gasquet, and A. Herzig. Modal tableaux for reasoning about actions and plans. In S. Steel, editor, *Proc. of European Conference on Planning (ECP'97)*, LNAI, pages 119–130. Springer-Verlag, 1997.

[6] G. De Giacomo and M. Lenzerini. PDL-based framework for reasoning about actions. In *Topics of Artificial Intelligence, AI*IA '95*, volume 992 of *LNAI*, pages 103–114. Springer-Verlag, 1995.

[7] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS, in *Proc. ATAL'97*, LNAI 1365, pages 155–176, 1997.

[8] L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. of ECAI'98.* pp. 537–541, 1998.

[9] L. Giordano, A. Martelli, and C. Schwind. Ramification and causality in a modal action logic. *Journal of Logic and Computation*, Vol. 10, No. 5, pp. 625–662, 2000.

[10] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31, 1997.

[11] D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel Publishing Company, 1984.

[12] H. Prendinger and G. Schurz. Reasoning about action and change. a dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996.

[13] A.S. Rao, M.P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. KR-91*, pages 473–484, Morgan Kaufmann, 1991.

[14] C. B. Schwind. A logic based framework for action theories. In J. Ginzburg, Z. Khasidashvili, C. Vogel, J.-J. Lévy, and E. Vallduví, editors, *Language, Logic and Computation*, pages 275–291, Stanford, USA, 1997. CSLI publication.