A Modal Programming Language for Representing Complex Actions

Matteo Baldoni, Laura Giordano, Alberto Martelli, and Viviana Patti
Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (ITALY)
Tel. + 39 11 74 29 111 — Fax. + 39 11 75 16 03
E-mail: {`baldoni,laura,mrt,patti`}`@di.unito.it`
URL: `http://www.di.unito.it/~argo`

Abstract

In this paper we propose a modal approach for reasoning about dynamic domains in a logic programming setting. In particular we define a language, called DyLOG, in which actions are naturally represented by modal operators. DyLOG is a language for reasoning about actions which allows to deal with ramifications and to define *procedures* to build *complex actions* from elementary ones. Procedure definitions can be easily specified in the modal language by introducing suitable axioms.

In the language the frame problem is given a *non-monotonic* solution by making use of persistency assumptions in the context of an abductive characterization. Moreover, a *goal directed proof procedure* is defined, which allows to compute a query from a given dynamic domain description.

1 Introduction

Reasoning about the effects of actions in a dynamically changing world is one of the main problems which must be faced by intelligent agents. Most of the approaches which have been developed to model actions and change allow to reason about the effects of a single action or of a sequence of primitive actions. An interesting extension consists in providing more general ways of composing actions, by defining conditional or iterative actions. This requires to develop a "programming language" for actions, where primitive actions play the role of assignments to variables in conventional programming languages, i.e. their execution causes a state change. Among the most significant achievements in this area, we mention GOLOG [22] and transaction logic [5, 6].

The effects and preconditions of actions are usually described using logic. Thus logic programming is a natural candidate to define a language for reasoning about actions, since it allows to combine reasoning capabilities with control aspects. However pure logic programming lacks an essential aspect: a way of dealing with state changes.

Starting from Gelfond and Lifschitz' seminal work [14], several proposals have been put forward for representing actions in logic programming, which provides simple and well studied nonmonotonic mechanisms. Gelfond and Lifschitz have defined a high-level action language $\mathcal{A}$ and they have given its translation into logic programming with negation as failure. Furthermore, [10] and [11] have proposed translation of (extensions of) the language $\mathcal{A}$ into abductive logic programming, while [24] has defined an extension of the language $\mathcal{A}$ to deal with concurrent actions, together with a sound and complete translation to abductive normal logic program. Other extensions of the language $\mathcal{A}$ have been proposed in the literature. For instance, the $\mathcal{AR}_0$ language of Kartha and Lifschitz [16] deals with the ramification problem, and for such language a translation is given into a formalism based on circumscription, rather than into logic programming. Furthermore, the language $\mathcal{A}_{\mathcal{K}}$, presented in [26], is an extension of $\mathcal{A}$ which deals with sensing actions.

Apart from [26] the above mentioned extensions of $\mathcal{A}$ do not cope with the problem of defining complex actions. This issue has been tackled in GOLOG [20, 22], a high level robot programming language which is based on a theory of actions in the situation calculus. The formalization of complex actions in GOLOG draws considerably from dynamic logic [17]. In particular, action operators like sequence, nondeterministic choice and iteration are provided. The definition of GOLOG is rather complex as it is based on second order logic. More precisely, second order logic is needed in defining complex actions, in particular, for iteration and for procedure definitions. While the second order formalization allows properties of GOLOG programs to be proved (through the induction principle), on the other hand it determines a gap between the definition of the language, which is quite general, and its Prolog implementation.

In this paper we present a modal action language DyLOG which allows to reason about complex actions. Its logical characterization is rather simple and very close to the procedural one, which is given by introducing a goal directed proof procedure. Since it allows complex actions, it is strongly related to GOLOG and to the $\mathcal{A}_{\mathcal{K}}$ language, which also incorporates complex actions. As a difference, rather than referring to an Algol-like paradigm for describing complex actions, DyLOG refers to a Prolog-like paradigm: complex actions are defined through (possibly recursive) definition, given by means of Prolog-like clauses.

The approach that we follow in defining our action theory is also different, as it is based on the adoption of a modal language. Starting from the similarities of GOLOG complex actions with dynamic logics, we argue that a natural definition of complex actions can be provided in a modal setting. Indeed, 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 [9, 28, 8, 30, 15], and it is motivated by the fact that modal logic allows very naturally to represent actions as state transition, through the accessibility relation of Kripke structures.

In particular, in [3] we have presented a modal logic programming language for reasoning about actions, where actions are represented by modalities. The language extends the language $\mathcal{A}$ and it allows to deal with ramifications, by means of "causal rules" among fluents, with incomplete initial states, and with nondeterministic actions. The language relies on an abductive semantics, to provide a nonmonotonic solution to the frame problem, and, when there are no ramifications, it has been proved to be equivalent to the language $\mathcal{A}$. In fact, the semantics of the language $\mathcal{A}$, which is defined in terms of a transition function among states, appears to be quite near to a canonical Kripke structure for our modal language.

The language DyLOG extends the previous language in various ways. First of all we introduce rules to specify action preconditions making use of "existential" modal operators. The main contribution of the paper concerns the extension of the language with "procedures", which allow to define complex actions. We show that this can be easily achieved in modal logics by defining a suitable set of *axioms* of the form $\langle p_1\rangle\langle p_2\rangle \ldots \langle p_n\rangle\varphi \supset \langle p_0\rangle\varphi$.

If p_0 is a procedure name, and the $p_i (i = 1, \ldots, n)$ are either procedure names, or primitive actions or test actions, the above axiom can be interpreted as a procedure definition, which can then be executed in a goal directed way, similarly to standard logic programs.

Though we do not make use of dynamic logic in our language, and, in particular, we do not introduce iteration, sequence and nondeterministic choice operators, we can model action iteration by recursive procedure definition, action sequences by action composition, and nondeterministic choice among actions by alternative clause definitions. Furthermore, we are able to give a name to complex actions by means of procedure definitions, which are not allowed in Dynamic Logic.

Summarizing, we present a language for reasoning about actions which combines the simplicity of the language $\mathcal{A}$, of which it can be regarded as an extension, with the capability to express complex actions as in GOLOG, building on the standard Kripke semantics of modal logics.

In the next section we present the language DyLOG. To make the comparison with other formalisms easier, we use a syntax close to that of language $\mathcal{A}$. In Section 3 we give a goal directed proof procedure for the language, and in Section 4 we present a logical characterization of DyLOG based on multimodal logics. We conclude by comparing our language with related work.

As it will emerge from Section 4 the adoption of a syntax close to that of language $\mathcal{A}$ is just a matter of notational convenience, and modal formulas could be directly used instead (as done for instance in [3]). Hence DyLOG can actually be regarded as a modal logic programming language.

2 The language DyLOG

The syntax of DyLOG is inspired from the language $\mathcal{A}$ and its extensions in [14, 16]. In the following we use atomic propositions for *fluent names* and we denote by F a *fluent expression*, consisting of a fluent name f or its negation $\neg f$ (note that $\neg\neg f$ is equal to f). Let $true$ be a distinguished proposition, then we denote by Fs a *fluent conjunction* defined as follows:

$$Fs \quad ::= \quad true \mid F \mid Fs_1 \wedge Fs_2$$

A *dynamic domain description* in DyLOG consist of three parts: a set of *simple action clauses*, a set of *procedures*, and a set of *observations* on the initial state.

The *simple action clauses* are rules that allow to describe direct and indirect effects of primitive actions on the world. 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 to represent actions with conditional effects. They have the form "a **causes** F **if** Fs", 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 to specify *action preconditions*, i.e. those conditions which make an action executable in a state. Precondition laws are of the form "a **possible if** Fs" 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 are of the form: "F **if** Fs" meaning that the fluent F holds if the fluent conjunction Fs holds too.

Intuitively, for describing a dynamic domain, we need a set of action and precondition laws for each primitive action together with a (possibly empty) set of causal laws.

Procedures define the behavior of *complex actions*. Complex actions are defined on the basis of 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* in DyLOG is defined as a collection of *procedure clauses* of the form

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

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). The sequence of p_i's is the body of the clause. If $n = 0$ we simply write the above procedure clause as p_0. Of course it is possible to define recursive procedures. Moreover, procedure can be *non-deterministic*: there can be more than one procedure clause for a certain procedure name p_0.

A set of *observations* describes what fluents are true in the initial state. They have the form " **initially** F" meaning that the fluent F holds in the initial state (i.e. before any action execution). For simplicity, we require the initial state to be complete, i.e. for each fluent name f either f or $\neg f$ holds in the initial state.

In the following, we use (Π, Obs) to denote a dynamic domain description, where Π is a set of simple action clauses Π_a and procedure clauses Π_p, while Obs is a set of observations on the initial state.

A *goal* in a dynamic domain description in DyLOG has the form:

$$Fs \textbf{ after } p_1, p_2, \ldots, p_n \quad (n \geq 0) \tag{1}$$

where p_i, $i = 1, \ldots, n$, is either a primitive action, or a procedure name, or a test and Fs is a fluent conjunction. Note that, if $n = 0$ we simply write the above goal as Fs.

Intuitively, the goal (1) corresponds to ask if it is possible to execute the sequence of test, primitive, and complex actions $p_1, p_2, \ldots, p_n$ in such a way that it terminates in a state where Fs is satisfied. Thus, Fs represents a *condition* on the terminating state. A goal succeeds if there is a terminating execution leading to a new state in which Fs holds. As usual in logic programming, execution of a goal 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 new one.

Though we will make use of variables in the next example, in this paper we develop the proof theory and the modal theory of our language for the propositional case only. An extension to the first order case is shortly addressed in Section 5.

Example 2.1 This example is taken from [5] and simulates the movements of a robot arm in a world of toy blocks. Possible states are represented by means of the fluents $on(x, y)$, $clear(x)$, and $wider(x, y)$ that say that block x is on top of block y, that nothing is on top of x, and that x is wider that y, respectively. We define two complex actions. The first one, $stack(n, x)$, specifies how to stack n arbitrary blocks on top of block x while the second one, $move(x, y)$, defines how to move block x on top of block y.

(1) $stack(N, X)$ **is** $(N > 0)?, move(Y, X), stack(N - 1, Y)$.
(2) $stack(0, X)$.
(3) $move(X, Y)$ **is** $pickup(X), putdown(X, Y)$.

The definition of complex action $stack(N, Y)$ is recursive. The complex action $move(X, Y)$ is defined in terms of the execution of the primitive actions $pickup(X)$ and $putdown(X, Y)$, which are ruled by the following laws:

(4) $\quad pickup(X)$ **possible if** $clear(X)$.
(5) $\quad pickup(X)$ **causes** $clear(Y)$ **if** $on(X, Y)$.
(6) $\quad putdown(X, Y)$ **possible if** $X \neq Y \wedge wider(Y, X) \wedge clear(Y)$.
(7) $\quad putdown(X, Y)$ **causes** $on(X, Y)$ **if** $true$.

The following two causal laws express some indirect effects of the primitive actions *pickup* and *putdown* by representing a fluent dependency between $on(X, Y)$ and $clear(X)$.

(8) $\quad \neg on(X, Y)$ **if** $clear(Y)$.
(9) $\quad \neg clear(Y)$ **if** $on(X, Y)$.

Suppose that we have a world with the five blocks a, b, c, d, and e such that a is wider than b, c, and d, while e is narrower than b, c, and d. Assume that all blocks are alone and, then, clear. We want to stack two blocks on a such that b remains clear. Let Π be the set of clauses (1)-(9) and let Obs be the set of observations that describes the initial situation above. Then the goal $clear(b)$ **after** $stack(2, a)$, succeeds with two solutions; the first one with answer "$pickup(c), putdown(c, a), pickup(e), putdown(e, c)$" and the second one with answer "$pickup(d), putdown(d, a), pickup(e), putdown(e, d)$". Note that the sequence "$pickup(b), putdown(b, a), pickup(e), putdown(e, b)$" is not an answer because it makes $clear(b)$ false after stacking. $\square$

To model persistency we say that if a fluent expression F holds in a state obtained after performing a sequence of primitive actions $a_1, \ldots, a_{m-1}$, and its complement $\neg F$ is not made true by the next primitive action a_m, then the fluent expression F holds after performing the actions $a_1, \ldots, a_{m-1}, a_m$. From a procedural point of view (see Section 3), our non-monotonic way of dealing with the frame problem consists in using *negation as failure* (NAF) in order to verify that the complement of the fluent F is not made true in the state $a_1, \ldots, a_{m-1}, a_m$. In the modal theory we will adopt an abductive semantics to capture persistency.

As an example, let us consider the case of executing the action $pickup(c)$ in a state in which c is on top of d and d is on top of e. Assume that before executing the action the following fluent expressions hold (among others): $clear(c)$, $\neg clear(d)$, $\neg clear(e)$, $on(c, d)$, and $on(d, e)$. Executing the action $pickup(c)$ has the immediate effect of making $clear(d)$ true, and the indirect effect of making $\neg on(c, d)$ true. All the other fluents are not affected by the action execution and, by applying persistency, they will keep their previous values. For instance, since $\neg on(d, e)$ cannot be proved after executing action $pickup(c)$, $on(d, e)$ will persist. On the other hand, the persistency of $\neg clear(d)$ and $on(c, d)$ is blocked.

As the action language in [3], DyLOG allows to deal with action ramifications. The ramification problem is concerned with the additional effects produced by an action besides its immediate effects [27, 25]. In our language ramifications can be expressed by means of causal laws which allow to formalize one-way causal relationships among fluents.

To represent causal laws correctly, we want to avoid their contrapositive use. In the logic programming setting it is quite natural to represent causal rules by making use of explicit negation [13]. When using explicit negation, a negative fluent $\neg f$ is regarded as a new positive atom. Hence, fluent expressions can be regarded as atomic formulae and causal laws can be seen as (modalized) Horn clauses, and thus they are directional implications.

3 Proof Procedure

In this section we introduce a *proof procedure* which allows to compute a query from a given dynamic domain description.

In [3] we have defined an abductive proof procedure in which alternative abductive solutions allow to model non-determinism w.r.t. the effects of primitive actions. The main focus of this paper is on the treatment of complex actions. For this reason and for sake of simplicity we will neither deal with primitive actions with non deterministic effects nor with incomplete initial state specification. To cope with such aspects an abductive proof procedure would be needed similar to the one presented in [3]. On the contrary, in this paper, we propose a proof procedure based on NAF.

The proof procedure reduces the procedure call in the query to a sequence of primitive actions and test actions, and verifies if execution of the primitive actions is possible and if the test actions are successful. To do this, it needs to reason about the execution of a sequence of primitive actions from the initial state and to compute the values of fluents at different states.

The first part of the proof procedure, denoted by " $\vdash_{ps}$ ", deals with execution of procedures (complex actions), primitive actions and test actions. To execute a procedure p we non-deterministically replace it with the body of a procedure clause for it. To execute a primitive action a, first we need to verify if that action is possible by using precondition laws. If so, then we can move to a new state, in which the action has been performed. Finally, to execute a test action $(Fs)?$, the value of Fs is checked in the current state. If Fs holds in the current state, the state action is simply eliminated, otherwise the computation fails.

During a computation, a *state* is represented by a sequence of primitive actions $a_1, a_2, \ldots, a_m$. The value of fluents at a state is not explicitly recorded but it is computed when needed in the computation. The second part of the procedure, denoted by " $\vdash_{fs}$ ", allows to determine the values of fluents in a state.

A goal of the form (1) succeeds if it is possible to execute $p_1, p_2, \ldots, p_n$ (in the order) starting from the current state, in such a way that Fs holds at the resulting state. In general, we will need to establish if a goal of the form (1) holds at a given state $a_1, \ldots, a_m$. Hence, we will write

$$a_1, \ldots, a_m \vdash_{ps} Fs \ \textbf{after} \ p_1, p_2, \ldots, p_n \text{ with answer } \sigma$$

to mean that the goal Fs **after** $p_1, p_2, \ldots, p_n$ can be proved from the dynamic domain description (Π, Obs) at the state $a_1, \ldots, a_m$ with answer σ, where σ is an action sequence $a_1, \ldots, a_m, \ldots a_{m+k}$ which represents the state resulting by executing $p_1, \ldots, p_n$ in the current state $a_1, \ldots, a_m$. In the following we denote by ε the initial state.

The first three rules define, respectively, how to execute procedure calls, test actions and primitive actions:

1) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p, p_2, \ldots, p_n$ with answer σ if there is a procedure clause (p **is** $p'_1, \ldots, p'_{n'}$) $\in \Pi$ and $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p'_1, \ldots, p'_{n'}, p_2, \ldots, p_n$ with answer σ;

2) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $(Fs')?, p_2, \ldots, p_n$ with answer σ if $a_1, \ldots, a_m \vdash_{fs} Fs'$ and $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p_2, \ldots, p_n$ with answer σ;

3) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $a, p_2, \ldots, p_n$ with answer σ if
there is a precondition law (a **possible if** Fs') $\in \Pi$ such that
$a_1, \ldots, a_m \vdash_{fs} Fs'$ and $a_1, \ldots, a_m, a \vdash_{ps} Fs$ **after** $p_2, \ldots, p_n$ with answer σ;

4) $a_1, \ldots, a_m \vdash_{ps} Fs$ with answer $\sigma = a_1, \ldots, a_m$ if $a_1, \ldots, a_m \vdash_{fs} Fs$.

Rule 4) deals with the case when there are no more actions to be executed. The sequence of primitive actions to be executed $a_1, a_2, \ldots, a_m$ has been already determined and, to check if Fs is true after $a_1, a_2, \ldots, a_m$, proof rules 5)-8) below are used.

The second part of the procedure determines the derivability of a fluent conjunction Fs at a state $a_1, a_2, \ldots, a_m$, denoted by $a_1, a_2, \ldots, a_m \vdash_{fs} Fs$, and it is defined inductively on the structure of Fs:

5) $a_1, \ldots, a_m \vdash_{fs} true$;

6) $a_1, \ldots, a_m \vdash_{fs} F$ if

   a) $m > 0$ and there exists an action law (a_m **causes** F **if** Fs) $\in \Pi$ such that $a_1, \ldots, a_{m-1} \vdash_{fs} Fs$,

   b) there exists a causal law (F **if** Fs) $\in \Pi$ such that $a_1, \ldots, a_m \vdash_{fs} Fs$,

   c) $m > 0$ and $a_1, \ldots, a_{m-1} \vdash_{fs} F$ and **not** $a_1, \ldots, a_m \vdash_{fs} \neg F$;

7) $a_1, \ldots, a_m \vdash_{fs} Fs_1 \wedge Fs_2$ if $a_1, \ldots, a_m \vdash_{fs} Fs_1$ and $a_1, \ldots, a_m \vdash_{fs} Fs_2$;

8) $\varepsilon \vdash_{fs} F$ if (**initially** F) $\in Obs$.

A fluent expression F holds at state $a_1, a_2, \ldots, a_m$ if: either F is an immediate effect of action a_m, whose preconditions hold in the previous state (rule 6(a)); or F can be derived by applying a causal law (rule 6(b)); or F holds in the previous state $a_1, a_2, \ldots, a_{m-1}$ and it persists after executing a_m (rule 6(c)). This last case allows to deal with the *frame problem*: F persists from a state $a_1, a_2, \ldots, a_{m-1}$ to the next state $a_1, a_2, \ldots, a_m$ unless a_m makes $\neg F$ true, i.e. it persists if $\neg F$ fails from $a_1, a_2, \ldots, a_m$. In rule 6(c) **not** represents *negation as failure*.

We say that a goal Fs **after** $p_1, p_2, \ldots, p_n$ *succeeds* from a dynamic domain description (Π, Obs) if it is operationally derivable from (Π, Obs) in the initial state ε by making use of the above proof rules with the *execution trace* σ as answer (i.e., $\varepsilon \vdash_{ps} Fs$ **after** $p_1, p_2, \ldots, p_n$ with answer σ).

4 The logical semantics

In this section we present a logical characterization of DyLOG in two steps. First, we introduce a multimodal logic interpretation of a dynamic domain description which describes the monotonic part of the language. Then, we provide an *abductive semantics* to account for non-monotonic behaviour of the language.

Following the proposal in [3], a dynamic domain description (Π, Obs) in DyLOG is interpreted in a multimodal logic. The basic idea is to use a finite number of modal operators to represent primitive actions, procedure names, and test actions. Moreover, we use a universal modal operator $[always]$ of type $S4$ to represent any sequence of primitive actions. Differently from [3], each dynamic domain description is interpreted in a particular multimodal logic characterized by a set of axiom schemas determined by its procedure clauses together with a theory based on the set of simple action clauses and the observation.

More precisely, given a dynamic domain description (Π, Obs), let us call $\mathcal{L}_{(\Pi,Obs)}$ the propositional modal logic on which (Π, Obs) is based. $\mathcal{L}_{(\Pi,Obs)}$ contains a finite number of modal operators $[t]$ and $\langle t\rangle$ (universal and existential modal operators, respectively) for each primitive action, procedure name, and test action t that appears in (Π, Obs). All these modalities are normal, that is, they are ruled at least by axiom $K(t) : [t](\psi \supset \varphi) \supset ([t]\psi \supset [t]\varphi)$.

Let us denote by Π_a the set of simple action clauses which are in Π and by Π_p the set of procedure clauses in Π (i.e. $\Pi = \Pi_a \cup \Pi_p$). The set Π_p is interpreted as a set of axiom schemas; more formally, the axiom system for $\mathcal{L}_{(\Pi,Obs)}$ contains an axiom schema of the form $\langle p_1\rangle\langle p_2\rangle \dots \langle p_n\rangle\varphi \supset \langle p_0\rangle\varphi$, for each procedure clause p_0 **is** $p_1, p_2, \dots, p_n$ in Π_p, where φ stands for an arbitrary formula, p_0 for a procedure name, and the p_i's for any primitive action, procedure name, or test action. We call $\mathcal{A}_{\Pi_p}$ the set of *axioms* determined by Π_p.

While the axioms determined by Π_p characterize a logic, the action clauses in Π_a and the observations in Obs define a *theory* fragment of the multimodal logic language $\mathcal{L}_{(\Pi,Obs)}$. In particular, we define the following mapping from action laws, precondition laws, causal laws, and observations which belong to Π_a and Obs to modal formulae (we call $\Sigma_{(\Pi,Obs)}$ the theory determined by Π_a and Obs):

$$\begin{array}{lll} a \textbf{ causes } F \textbf{ if } Fs & \rightsquigarrow & [always](Fs \supset [a]F) \\ a \textbf{ possible if } Fs & \rightsquigarrow & [always](Fs \supset \langle a\rangle true) \\ F \textbf{ if } Fs & \rightsquigarrow & [always](Fs \supset F) \\ \textbf{initially } F & \rightsquigarrow & F \end{array}$$

where $[always]$ is a modal operator of type $S4$ that represents any sequence of primitive actions, and it is used to denote information which holds in any state. Hence, $[always]$ occurs in front of all simple action clauses that have to hold in any state, while it does not occur in front of observations that have to hold only in the initial state. The axiom system for $\mathcal{L}_{(\Pi,Obs)}$ contains the following *interaction axiom* schema $I(always, a_i) : [always]\varphi \supset [a_i]\varphi$, one for each primitive action a_i in (Π, Obs), which rules the interaction between the modal operator $[always]$ and the modalities $[a_i]$.

The language $\mathcal{L}_{(\Pi,Obs)}$ contains test modalities. Like in dynamic logic [18], if ψ is a proposition then $\psi?$ can be used as a label for a modal operator. As usual, besides the axiom K, the modalities built by the operator "?" are axiomatized by the following: $\langle\psi?\rangle\varphi \iff \psi \wedge \varphi$. Summarizing, a domain description (Π, Obs) is interpreted in a multimodal logic $\mathcal{L}_{(\Pi,Obs)}$ and the simple action clauses in Π_a and the observations in Obs determine a theory $\Sigma_{(\Pi,Obs)}$ in such a logic.

The meaning of a formula in $\mathcal{L}_{(\Pi,Obs)}$ is given by means of a standard Kripke semantics and more details can be find in [1].

The monotonic part of the language does not account for persistency. In order to deal with the frame problem, we introduce a non-monotonic semantics for our language by making use of an abductive construction: abductive assumptions will be used to model persistency from one state to the following one, when a primitive action is performed. In particular, we will assume that a fluent expression F persists through an action unless it is inconsistent to assume so, i.e. unless $\neg F$ holds after the action.

The semantics we define is an extension of the abductive semantics proposed in [3] to deal with complex actions definitions. As a difference with [3] here we have adopted a two valued semantics, instead of a three-valued one. Moreover, here we do not allow for incomplete initial states.

In defining our abductive semantics, we adopt (in a modal setting) the style of Eshghi and Kowalski's abductive semantics for negation as failure [12]. We define a new set of atomic propositions of the form $\mathbf{M}[a_1][a_2]\ldots[a_m]F$ and we take them as being *abducibles*.[1] Their meaning is that the fluent expression F can be assumed to hold in the state obtained by executing primitive actions $a_1, a_2, \ldots, a_m$. Each abducible can be assumed to hold, provided it is consistent with the domain description (Π, Obs) and with other assumed abducibles. More precisely, in order to deal with the frame problem, we add to the axiom system of $\mathcal{L}_{(\Pi,Obs)}$ the *persistency axiom schema*

$$[a_1][a_2]\ldots[a_{m-1}]F \wedge \mathbf{M}[a_1][a_2]\ldots[a_{m-1}][a_m]F \supset [a_1][a_2]\ldots[a_{m-1}][a_m]F \tag{2}$$

where $a_1, a_2, \ldots, a_m$ $(m > 0)$ are primitive actions, and F is a fluent expression. Its meaning is that, if F holds after action sequence $a_1, a_2, \ldots, a_{m-1}$, and F can be assumed to persist after action a_m (i.e., it is consistent to assume $\mathbf{M}[a_1][a_2]\ldots[a_m]F$), then we can conclude that F holds after performing the sequence of actions $a_1, a_2, \ldots, a_m$.

Given a domain description (Π, Obs), let $\models$ be the satisfiability relation in the monotonic modal logic $\mathcal{L}_{(\Pi,Obs)}$ defined in the previous section (including axiom schema (2)). In the following, $\neg\neg p$ is regarded as being equal to p.

Definition 4.1 *A set of* abducibles Δ *is an* abductive solution *for* (Π, Obs) *if,*

a) $\forall \mathbf{M}[a_1][a_2]\ldots[a_m]F \in \Delta$, $\Sigma_{(\Pi,Obs)} \cup \Delta \not\models [a_1][a_2]\ldots[a_m]\neg F$

b) $\forall \mathbf{M}[a_1][a_2]\ldots[a_m]F \not\in \Delta$, $\Sigma_{(\Pi,Obs)} \cup \Delta \models [a_1][a_2]\ldots[a_m]\neg F$.

Condition a) is a *consistency* condition, which guarantees that each assumption cannot be assumed if its "complementary" formula holds. Condition b) is a *maximality* condition which forces an abducible to be assumed, unless its "complement" is proved. When an action is applied in a certain state, persistency of those fluents which are not modified by the direct or indirect effects of the action, is obtained by maximizing persistency assumptions.

Since p and $\neg p$ are taken as two different propositions, it might occur that both of them hold in the same state, in an abductive solution. To avoid this, we introduce a consistency condition to accept only those solutions without inconsistent states. We say that an abductive solution Δ is *acceptable* if, for every sequence of actions $a_1, a_2, \ldots, a_m$, $(m \geq 0)$, and fluent name p: $\Sigma_{(\Pi,Obs)} \cup \Delta \not\models [a_1][a_2]\ldots[a_m]p \wedge [a_1][a_2]\ldots[a_m]\neg p$.

Definition 4.2 *Given a domain description* (Π, Obs) *and a goal* Fs **after** $p_1, p_2, \ldots, p_n$, *a* solution for the goal in (Π, Obs) *is defined to be an acceptable abductive solution for* (Π, Obs) *such that* $\Sigma_{(\Pi,Obs)} \cup \Delta \models \langle p_1\rangle\langle p_2\rangle\ldots\langle p_n\rangle Fs$.

According to the definition above, a domain description may have more than one abductive solution. Indeed the semantics above also allows to model primitive actions with non-deterministic effects: executing an action may lead to alternative states, each one modeled by a different abductive solution. Though in our language primitive actions cannot be non-deterministic with respect to immediate effects, they can have non-deterministic effects through

[1] Notice that $\mathbf{M}$ has not to be regarded as a modality. Rather, $\mathbf{M}\alpha$ is the notation used to denote a new atomic proposition associated with α. This notation has been adopted in analogy to default logic, where a justification $\mathbf{M}\alpha$ intuitively means "α is consistent".

ramifications (causal laws). Since this paper is principally devoted to explore how to deal with complex actions definitions, as mentioned above, the proof procedure presented in Section 3 does not account for non-deterministic primitive actions. While we refer to [3] for an example on non-deterministic actions, we just want to observe that the case of multiple solutions occurs when there are negative dependencies in the set of causal rules (as, for instance, a **if** $\neg b \wedge c$ and b **if** $\neg a \wedge c$). These cases cannot be dealt with our proof procedure, since they cause the procedure to enter an infinite loop.

There are also cases when a set of action laws and causal rules Π may have no abductive solution. It may happen, for instance, when Π contains causal rules with negative dependencies, as the rule F **if** $\neg F$. In such a case, it might be impossible to find a set of abducibles satisfying both conditions (a) and (b). The fact that some domain description may have no abductive solution is quite similar to the problem of nonexistence of stable models in logic programs with negation as failure.

The proof procedure computes just one solution, while the abductive semantics may give no solutions or multiple solutions for a given domain description. Hence, we can say that the relation of this logical characterization with the proof procedure above is similar to the relation of stable model semantics with SLDNF. Since existence of abductive solutions is not guaranteed, what we can show is that our procedure gives sound results in the case an abductive solution exists. We can prove the following.

Theorem 4.1 (Soundness) *Let* (Π, Obs) *be a dynamic domain description and let* Fs **after** $p_1, p_2, \ldots, p_m$ *be a goal in* DyLOG. *Then, for all abductive solutions* Δ *for* (Π, Obs), *if* Fs **after** $p_1, p_2, \ldots, p_m$ *succeeds from* (Π, Obs) *then* $\Sigma_{(\Pi,Obs)} \cup \Delta \models \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_m \rangle Fs$.

The proof is omitted for sake of brevity. It can be done by induction on the rank of the derivation of the goal, and it makes use of a soundness and completeness result for the monotonic part of the proof procedure presented in Section 3 with respect to the monotonic part of the semantics.

As in the case of logic programs with negation as failure, syntactic conditions can be defined on domain descriptions which guarantee the existence of abductive solutions. For instance, it is quite natural to define a notion of *stratified* domain description, by imposing that causal laws are such that a fluent cannot depend negatively on itself. Such a restriction would allow to avoid negative loops and thus to assure existence of abductive solutions.

This restriction to stratified rule bases has also the effect to avoid multiple solutions and hence to force primitive actions to be deterministic. In this way the nondeterministic part of the language is confined at the level of procedure definitions.

Notice that the proof procedure in Section 3 does not perform any consistency check on the computed abductive solution. There are cases when a set of action laws and causal laws Π has abductive solutions, but it does not have acceptable ones. This may occur when Π contains alternative action laws for an action a, which may be applicable in the same state, and have mutually inconsistent (immediate or non immediate) effects. For instance, if there are two action laws for a as follows:

$$a \textbf{ causes } f \textbf{ if } p \qquad a \textbf{ causes } \neg f \textbf{ if } q,$$

when a is executed in a state containing both p and q, the successor state will contain both f and $\neg f$, and hence it will be inconsistent.

Another possible cause for the nonexistence of acceptable solutions comes from inconsistencies in the set of observations Obs.

Conditions can be defined on domain descriptions to guarantee the acceptability (consistency) of abductive solutions (if any). Such conditions generalize the notion of *e-consistency* [10], which has been defined for the language $\mathcal{A}$. In our context, since causal laws are present in a domain description, essentially we have to require that, for any set of action laws (for a given action) which may be applicable in the same state, the set of their immediate and indirect effects (obtainable through the causal laws) is consistent.

The requirements of stratification and e-consistency together ensure that a domain description has a unique acceptable solution (if Obs is consistent). Under these conditions we argue that completeness of the proof procedure in Section 3 can be proved. In fact, the conditions above prevent the proof procedure from entering an infinite loop.

Rather then imposing syntactic restrictions on domain descriptions, an alternative way to solve the problem of non existence of abductive solutions is that of moving to a three-valued semantics, by weakening conditions (a) and (b) above. Such a solution was adopted in [3], for the language with only primitive actions.

5 Conclusions

In this paper we have defined a logic programming language for reasoning about actions, which includes complex actions introduced by means of procedure definitions. In defining the proof theory and the model theory of our language we have only focused on the propositional part of the language. However, both the proof procedure of the language and its modal characterization can be extended to the first order case. In the first order case, variable may occur both in fluent names and in action names (a first order monotonic modal programming language of this kind was studied in [2, 1]).

The problems which arise when addressing the first order case are similar to those which occur in first order logic programs in presence of negation as failure (NAF). More precisely, the *floundering* problem has to be addressed. In our language NAF is used by the proof procedure at step 6c), the one which deals with persistency of fluents. In order to avoid floundering we have to ensure that, during the computation, all the actions in the sequence $a_1, \ldots, a_m$ are groundly instantiated, and, at each state, only ground fluent expressions can be derived. To enforce this, an *allowedness* condition (see [19]) can be put on action laws, preconditions laws and causal laws. Essentially, all the variables occurring in the head of action laws, preconditions laws and causal laws have also to occur in their bodies. Moreover, the set Obs of observations on the initial state must be ground.

The major formalisms introduced for reasoning about dynamic domains and for defining and executing complex actions are Transaction Logic and GOLOG.

Transaction Logic ($\mathcal{TR}$) [5, 6, 7] is a formalism designed to deal with a wide range of update related problems in logic programming, databases and AI. It provides a natural way to define composite transactions as named procedures, by giving them a declarative specification in a logical first order framework. In $\mathcal{TR}$ complex transactions can be constructed from elementary actions by sequential and concurrent composition. Moreover, non-deterministic transactions and constraints on transactions can be expressed, and both hypothetical and committing actions are allowed.

In our language only the sequential part of $\mathcal{TR}$ can be represented. Essentially our procedures can be defined by sequential composition, and both non-deterministic and recursive procedures are allowed. As a difference with $\mathcal{TR}$, in our language constraints on the execution of transactions cannot be defined, except on the initial and final state of the transactions. This is inherent in the semantic structure of our language, which is not based on "path structures" as the semantic theory of $\mathcal{TR}$.

A major difference with $\mathcal{TR}$ which makes our language much more similar GOLOG than to $\mathcal{TR}$ concerns elementary actions. While $\mathcal{TR}$ is parametric with respect to elementary transitions, which are defined through a "transition base" and whose behaviour is not modelled in the language, in our language effects of elementary actions have to be defined by introducing action laws and causal laws. Moreover, our domain description contains precondition laws which are used to establish if elementary actions are executable in a given state. Simple action clauses provide a very compact definition of actions. Aspects of action theory like action precondition, ramification, persistency have not to be addressed in $\mathcal{TR}$, since the "transition base" provides an extensional description of the behaviour of elementary actions (by describing the effects of actions on all possible states).

As concerns the specification of elementary actions, our language is much more similar to GOLOG [22], though, from the technical point of view, it is based on a different approach. While our language makes use of modal logic, GOLOG is based on classical logic and, more precisely, on the situation calculus. We make use of abduction to deal with persistency, while in GOLOG is given a monotonic solution of the frame problem by introducing successor state axioms. In our case, procedures are defined as axioms of our modal logic, while in GOLOG they are defined by macro expansion into formulae of the situation calculus. Moreover, in DyLOG it is very natural to express, within the goal, conditions on the final state of a procedure execution (e.g. Example 2.1).

As mentioned in the introduction, GOLOG definition is very general and it makes use of second order logic to define iteration and procedure definition. Hence there is a certain gap between the general theory on which GOLOG is based and its implementation in Prolog. In contrast, in this paper we have tried to keep the definition of the semantics of the language and of its proof procedure as close as possible.

In this work we do not deal with sensing actions, that is, actions whose effect is that of changing state of knowledge [29]. Sensing actions are foundamental to reason about conditional plans in presence of incomplete information [21, 4, 26]. In particular, in [26], the authors extend Gelfond and Lifschitz' language $\mathcal{A}$ to reason about complex plans, including conditional and iterations, in presence of sensing. In this case, plans can depend on the outcome of the sensing actions and the language allows to model a form of hypothetical reasoning on complex plans. Instead, in our language, as in [22], we can describe and execute complex plans, that are defined by means of a DyLOG logic program, though plans can only depend on information about the initial state. Nevertheless, in DyLOG we can provide names for complex plans and, moreover, we can express action preconditions and causal relationships among fluents to deal with ramifications. Extending DyLOG to reason about sensing actions is a direction for future work.

In defining our language we have not made use of dynamic logic, and, in particular, we have not introduced in our language program operators as "sequential composition", "non-deterministic choice"and "iteration". On the other hand, we have the test operator and we define our procedures as sequences of actions. The capability of defining (recursive) procedures makes a further difference with dynamic logic.

An approach to reasoning about action based on dynamic logic has been presented in [9].

There a monotonic solution to the frame problem is adopted. The language in [9] does not deal with procedure definitions but it allows to deal with concurrent actions.

The semantics of our language is defined by following an abductive approach. Hence, our work has strong connections with [10, 11, 24, 23]. All these works address the problem of reasoning about actions in a logic programming abductive setting, by defining translations of the language $\mathcal{A}$ and its extensions to abductive logic programming. These languages do not deal specifically with procedure definitions and we refer to [3] for a more detailed comparison with these approaches. We just want to mention that in [24] a language with concurrent actions is developed. Parallel composition is an operator to construct complex actions that we have not considered in our language and this is a direction in which DyLOG can be extended.

References

[1] M. Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension.* PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 1998. Available at `http://www.di.unito.it/~baldoni/`.

[2] M. Baldoni, L. Giordano, and A. Martelli. A Framework for Modal Logic Programming. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'96*, pages 52–66, Bonn, 1996. The MIT Press.

[3] 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.

[4] C. Baral and T. C. Son. Approximate Reasoning about Actions in Presence of Sensing and Incomplete Information. In J. Małuszyński, editor, *Proc. of the International Symposium on Logic Programming, ILPS'97*, pages 387–404, Cambridge, 1997. MIT Press.

[5] A. J. Bonner and M. Kifer. Transaction Logic Programming. In *Proceedings of International Conference on Logic Programming, ICLP'93*, pages 257–279, Budapest, Hungary, 1993. The MIT Press.

[6] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, 1994.

[7] A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Proc. of Joint International Conference and Symposium on Logic Programming, JICSLP '96*, pages 142–156, Bonn, 1996. The MIT Press.

[8] 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.

[9] 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.

[10] M. Denecker and D. De Schreye. Representing Incomplete Knowledge in Abduction Logic Programming. In *Proc. of 1993 International Logic Programming Symposium, ILPS '93*, Vancouver, 1993. The MIT Press.

[11] P. M. Dung. Representing Actions in Logic Programming and its Applications to Database Updates. In *Proc. of the ICLP'93*, Budapest, 1993.

[12] K. Eshghi and R. Kowalski. Abduction compared with Negation by Failure. In *Proc. of 1989 International Conference on Logic Programming, ICLP '89*, Lisbon, 1989. The MIT Press.

[13] M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In *Proc. of 1990 International Conference on Logic Programming, ICLP '90*, Jerusalem, 1990. The MIT Press.

[14] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[15] L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. of ECAI'98*, 1998. To appear.

[16] E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing actions: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[17] 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.

[18] D. Kozen and J. Tiuryn. Logics of Programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 788–840. Elsevier Science Publishers, 1990.

[19] K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7, 1989.

[20] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. A logical approach to high-level robot programming — a progress report. In B. Kuipers, editor, *Proc. of 1994 AAAI Fall Symposium: Control of the Physical World by Intelligent Systems*, 1994.

[21] H. J. Levesque. What is planning in the presence of sensing? In *Proc. of the AAAI-96*, pages 1139–1146, 1996.

[22] 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.

[23] R. Li and L. M. Pereira. Representing and Reasoning about Concurrent Actions with Abductive Logic Programs. *Annals of Mathematics and AI*, 1996. Special Issue for Gelfondfest.

[24] R. Li and L. M. Pereira. Temporal Reasoning with Abductive Logic Programming. In *Proc. of the ECAI'96*, 1996.

[25] F. Lin. Embracing Causality in specifying the Indirect Effects of Actions. In *Proc. of International Joint Conference on Artificial Intelligence, IJCAI '95*, Montréal, Canada, 1995. Morgan Kaufmann.

[26] J. Lobo, G. Mendez, and S. R. Taylor. Adding Knowledge to the Action Description Language $\dashv$. In *Proc. of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference, AAAI'97/IAAI'97*, pages 454–459, Menlo Park, 1997. AAAI Press.

[27] N. McCain and H. Turner. A Causal Theory of Ramifications and Qualifications. In *Proc. of International Joint Conference on Artificial Intelligence, IJCAI '95*, Montréal, Canada, 1995. Morgan Kaufmann.

[28] 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.

[29] R. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proc. of the AAAI-93*, pages 689–695, Washington, DC, 1993.

[30] 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.