Incremental learning of relational action rules

Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, Henry Soldano
L.I.P.N, UMR-CNRS 7030
Université Paris-Nord
Villetaneuse, France
first-name.last-name@lipn.univ-paris13.fr

***Abstract*—In the Relational Reinforcement learning framework, we propose an algorithm that learns an action model allowing to predict the resulting state of each action in any given situation. The system incrementally learns a set of first order rules: each time an example contradicting the current model (a counter-example) is encountered, the model is revised to preserve coherence and completeness, by using data-driven generalization and specialization mechanisms. The system is proved to converge by storing counter-examples only, and experiments on RRL benchmarks demonstrate its good performance *w.r.t* state of the art RRL systems.**

***Keywords*-relational reinforcement learning; inductive logic programming; online and incremental learning**

I. Introduction

Reinforcement Learning (RL) considers systems involved in a sensori-motor loop with their environment, formalized by an underlying Markov Decision Process (MDP) [1]. Usual RL techniques use propositional learning techniques. Recently, we have observed a growing interest for RL algorithms using a relational representation of states and actions. These works lead to adaptations of regular RL algorithms to relational representations. Instead of representing states by valued attributes, states are described by relations between objects (see [2] for a review). Relational representations allow better generalization capabilities, improved scaling-up and transfer of solutions since they neither rely on the number of attributes nor their order.

Another way to improve a RL system is to learn and use an action model of the environment in addition to value/reward functions [3]. Such a model permits to predict the new state after applying an action. In this paper, we focus on *incrementally* learning a deterministic *relational action model* in a noise-free context. Examples are available to the system on line and only some of these examples are stored, still ensuring the convergence of the learning process.

Our Incremental Relational Action Learning algorithm (IRALe), starts with an empty action model and incrementally revises it each time an example contradicts it (i.e., when the action model does not exactly predict the actual effects for a performed action in a given state) using data-driven learning operators. IRALe can be seen as a theory revision algorithm applied to action rule learning: starting from scratch, it learns multiple action labels, where the labels are the different possible effects – represented as conjunctions of literals – observed after executing an action in a given state. IRALe stores the examples that have raised a contradiction at some point during the model construction; as a consequence converges because the number of hypotheses to explore is bounded and that the system will never select as a revision the same hypothesis twice.

We first describe some related work in Section II. In Section III, we settle our learning framework by describing the relational first order representation of states and actions, together with the subsumption relation. We then provide an overview of the proposed incremental generalization and specialization revision mechanisms that will be further described in Section IV. In Section V, we show that our revision algorithm converges under reasonable hypotheses. Before concluding, in Section VI, the system is empirically evaluated on regular RRL benchmark environments.

II. Related work

Learning planning operators has been studied intensively, including the problem of learning the effects of actions, in both deterministic and probabilistic settings, some in the context of relational reinforcement learning (see [2] for a review). The system mostly related to ours is MARLIE [4], the first Relational RL system integrating incremental action model and policy learning. MARLIE uses TG [5] to learn relational decision trees, each used to predict whether a particular literal is true in the resulting state. The decision trees are not restructured when new examples appear that should lead to reconsider the internal nodes of the tree. Other systems [6] integrating such restructurations scale poorly.

In the planning field, several works aim at learning action models but show limitations *w.r.t.* the RL framework. Benson's work [7] relies on an external teacher; EXPO [8] starts with a given set of operators to be refined, and cannot start from scratch; OBSERVER [9] knows in advance the number of STRIPS rules to be discovered and which examples are indeed relevant to each rule; LIVE [10] doesn't scale up very well: it proceeds by successive specializations but cannot reconsider early over-specializations.

Several recent approaches learn probabilistic action models so as to take into account noise and/or non-deterministic domains (see [11] among others). In this work, the structure of the action model is learned first, before the probability distribution, both in batch mode. To our knowledge, no system is able to learn such models incrementally yet.

III. Learning problem

A. Action model and relational representation

Formally, a Markov Decision Process (MDP) is defined by a $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ t-uple. $\mathcal{S}$ is the space of possible states and $\mathcal{A}$ is the space of possible actions. $\mathcal{R}$ denotes an immediate reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. $\mathcal{T}$ is a transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. Functions may be stochastically defined but in this paper, we only address the case of deterministic environments.

Here, we focus on learning a model of the $\mathcal{T}$ function. An example x for the learning process is composed of a given state, a chosen action and the resulting new state. We respectively note them as $x.s$, $x.a$ and $x.s'$. We call an *effect* $x.e$ the description of what changed in the state after applying the action, *i.e.* $x.e = \delta(x.s, x.s')$. Rather than strictly learning a model of the transition function, we model what changes when the action applies. Let us note this model M.

States and actions are depicted using objects of the environment as well as relations between them. Relations between objects are described using predicates applied to constants. For instance, in a blocks-world, if a block a is on top of a block b, it is denoted by the literal $on(a, b)$. Let us introduce the notions of fist order logic used in this paper. Objects are denoted by constants, denoted by a lower-case character (a, b, f,...), and variables, denoted by an upper-case character (X, Y, ...), may instantiate to any object of the domain. A *term* is here a constant or a variable. Actions and relations between objects are denoted by *predicate* symbols. An *atom* is a predicate symbol applied to terms (*eg* $move(a, X)$, $on(b, Y)$). An instantiated atom is a *fact*, *i.e.* an atom without any variable. A *literal* is an atom or the negation of an atom.

B. Examples

Examples are described in a Datalog-like language (no function symbol but constants). As usual, we assume that when an agent emits an action, state literals that are not affected by the action are not described in the effect part. The examples can be noted $x.s/x.a/x.e.add, x.e.del$, with $x.s$, $x.a$, $x.e.add$ described as conjunctions of positive literals and $x.e.del$ described as a conjunction of negated literals, as usual in a STRIPS-like notation. Fig. 1 shows the example $on(a, f), on(b, f), on(c, a)/move(c, b)/on(c, b), \neg on(c, a)$.

Within the RL framework, the learning process has to be incremental, *i.e.* each time step the agent performs a new action, the system receives a new example x, which must be taken into account for potentially revising M into M', here described as a set of action rules.

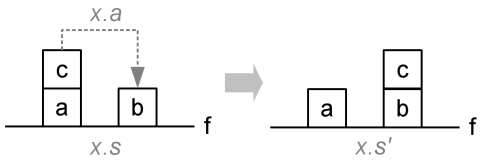

Figure 1. Example of a $move$ action in a simple blocks-world

C. Action Rules

Most works in RRL use instance based methods [12] or decision trees [5] for representing action rules. Existing instance based methods require an ad hoc distance suited to the problem. Decision trees are top-down methods which – in the incremental case — highly rely on the order of presentation of the examples, thus leading to over-specializations. This problem may be addressed by online re-structuring the tree, but this is a difficult issue.

In this paper, we propose a rule based action model. This choice is motivated by the fact that reconsidering a previous generalization or specialization choice is easier in a rule based model than in a tree structured model. This backtrack feature is critical in our incremental RL framework, where examples are drawn from actual experience, and may be presented to the system in a misleading order. Furthermore no ad hoc prior knowledge is needed as opposed to an instance-based action model.

So, the model M is composed by a rule set. Each rule r, just like an example, is composed of a precondition $r.p$, an action $r.a$ and an effect $r.e$, and is denoted as $r.p\ /r.a\ /r.e$. The precondition is represented by a conjunction of positive literals, which have to be satisfied so as to apply the rule. The action is a literal defining the performed action. The effect is composed of two literals sets: $r.e.add$ is the set of literals getting true when the rule is applied, and $r.e.del$ is the set of literals getting false when the rule is applied. According to a rule r, an action $r.a$ has no other effects but those described by $r.e$. In order to be *well formed*, a rule r must be such that i) $r.e.del \subseteq r.p$ ii) $r.e.add \cap r.p = \emptyset$ iii) $r.a \cup r.e$ must be connected[1]. Finally, all variables occurring in $r.a$ should also occur in $r.p$ and $r.e$, but $r.p$ and $r.e$ may reference objects/variables not occurring in $r.a$ (see deitic references in [11]). For instance, a well-formed rule is the following : $on(X, Z)\ /move(X, Y)\ /on(X, Y), \neg on(X, Z)$.

This formalism, we refer to as *extended deterministic* STRIPS (*EDS* for short) in the following, is more expressive than the *deterministic* STRIPS, considered for instance in [13]. For a given action and effect, it is possible to have several preconditions, each expressed as a conjunction of

[1]Any variable occurring in $r.e$ should be linked through a path of literals to a variable occurring in the action literal $r.a$.

literals. The system can learn several rules matching different preconditions to a given action-effects pair, extending the classical STRIPS formalism. It is not the case in [13], which only accepts conjunctive preconditions for an action. However, we can always express an action model in *EDS* as an action model in deterministic STRIPS, thus allowing to use any STRIPS planner, in the following way:

- For each rule $r_i = (r_i.p / r_i.a = A / r_i.e)$ for a given action A, build a STRIPS action A_i associated to the unique rule $r'_i = (r'_i.p = r_i.p / r'_i.a = A_i / r'_i.e = r_i.e)$
- For each rule r'_i add to $r'_i.a$ the variables that appear in $r'_i.p$ and $r'_i.e$ but not in $r'_i.a$.

Example 1: Consider the following *EDS* action model:
$M = \{$
$r_1 : w(X), w(Y), on(X,Z) / move(X,Y) / on(X,Y), \neg on(X,Z),$
$r_2 : w(X), b(Y), on(X,Z) / move(X,Y) / w(Y), \neg b(Y)\}$ stating that moving X on Y results either in placing X on Y, whenever X and Y are white blocks, or in only changing the color of Y to white, whenever Y is black. The resulting translation to STRIPS is as follows: $M' = \{$
$r'_1 = w(X), w(Y), on(X,Z) / move_1(X,Y,Z) / on(X,Y), \neg on(X,Z)$
$r'_2 = w(X), b(Y), on(X,Z) / move_2(X,Y,Z) / w(Y), \neg b(Y)\}$

Pragmatic actions as known by the agent before learning are denoted as p-actions, and the resulting STRIPS actions are denoted as s-actions. This means that learning in *EDS* consists in learning a set of (preconditions, p-action, effects) triples that will be further translated in a deterministic STRIPS action model[2]. As a result, learning here includes learning the "schema" of STRIPS s-actions (number of rules, exact variables involved in the action), and we argue this is difficult for an expert to accurately provide such a schema before learning takes place.

D. Subsumption

We define in the following three matching operators between rules and examples. All such matching operations rely on a *generality* relation frequently used in the Inductive Logic Programming (ILP) framework: subsumption under Object Identity (OI-subsumption) [14].

A formula G OI-subsumes a formula S iff there exists a substitution σ such that $G\sigma \subseteq S$ where σ is an injective substitution (two different variables of the domain of σ are assigned to different terms): for instance, $p(X,Y)$ does not OI-subsume $p(a,a)$ because X and Y can't be assigned to the same constant. It has been showed [14] that the set of OI-generalizations of a conjunction of literals is the set of its subsets (up to a renaming of variables) and a complete generalization operator drops literals, which is indeed a very intuitive interpretation for generalization in our context. Note that computing the least general generalization (*lgg*) of two formulas under OI may produce several *l*ggs, each corresponding to a largest common substructure between the input formulas.

[2]A translation preserving p-actions as modelized actions could be obtained by allowing conditional effects in STRIPS.

E. Coverage, contradiction, completeness, coherence

We define three matching operators.

Definition 1 (pre-matching $\overset{sa}{\sim}$): For any rule r, state s and action a, $r \overset{sa}{\sim} (s,a)$ iff there exists injective substitutions σ and θ such that i) $(r.a)\sigma = a$ ii) $(r.p)\sigma\theta \subseteq s$.
This operator permits to decide whether a given rule may apply to predict the effect of a given example.

Definition 2 (post-matching $\overset{ae}{\sim}$): For any rule r, and action a and effect e, $r \overset{ae}{\sim} (a,e)$ iff there exists an inverse substitution ρ^{-1}, and two injective substitutions σ and θ such that i) $(r.a)\rho^{-1}\sigma = a$ ii) $(r.e)\rho^{-1}\sigma\theta = e$.
This operator allows to decide whether a given rule may explain a given state modification when a given action is performed. The inverse substitution enables generalization of constants in the action and effect parts of examples/rules.

Definition 3 (covering $\approx$): For any rule r and example x, $r \approx x$ iff $r \overset{sa}{\sim} (x.s, x.a)$ and $r \overset{ae}{\sim} (x.a, x.e)$ for the same injective substitutions σ and θ.
This operator checks whether an example can be accurately predicted by the model.

Example 2: The rule $on(X,Z) \ / move(X,Y) \ / on(X,Y), \neg on(X,Z)$ pre-matches example of Fig. 1, with substitution $\sigma = \{X/c, Y/b\}$ and $\theta = \{Z/a\}$. This rule also post-matches the example with the same substitutions.

Definition 4 (contradiction $\nsim$): An example x *contradicts* a rule r ($x \nsim r$) if r pre-matches $(x.s, x.a)$ for σ and θ substitutions, and r doesn't post-match $(x.a, x.e)$ with the same substitutions.
In such a case, the rule incorrectly predicts the outcomes of the action. The model M includes a set of rules denoted by $M.R$. We say that a rule set $M.R$ or a model *covers* an example x when there exists a rule r in $M.R$ such that $r \approx x$. Similarly, we say that a rule set $M.R$ or a model *rejects* an example when no rule of $M.R$ covers it.

The above definitions can be extended for defining matching and covering between rules. The model M also includes a set of counter-examples which have been met since the beginning of the learning session, denoted by $M.X$.

Definition 5 (counter-example): x_u is a counter-example for M iff there is no rule $r \in M.R$, such that $r \approx x$.
Any uncovered example x_u may be a counter-example either for a completeness issue (no pre-matching) or for a coherence issue (pre-matching without post-matching). In both cases, the model M needs to be updated to M' in order to preserve coherence and completeness *w.r.t.* x_u and other past counter examples in $M.X$, as defined below.

Definition 6 (completeness): We say that M is *complete* iff $\forall x \in M.X$, there is at least one $r \in M.R$ such that $r \overset{sa}{\sim} (x.s, x.a)$. This means that the set of rules permits to make a prediction for any past counter-example.

Definition 7 (coherence): We say that M is *coherent* iff $\forall x \in M.X$, there is at least one $r \in M.R$ such that $r \overset{sa}{\sim} (x.s, x.a) \Rightarrow r \overset{ae}{\sim} (x.a, x.e)$. This means that for any past counter-example, if the set of rules can make a prediction, then this prediction is accurate.

IV. Incremental relational learning of an action model

There are few theory revision systems in ILP and to our knowledge, none learns action rules. The system mostly related to ours is INTHELEX [15] that also uses a lgg-based operator under OI-subsumption. INTHELEX specializes a clause by adding literals to reject a negative example, and so faces the difficult choice of selecting the best literal to add when there are several candidates. Our specialization algorithm is simpler as it specializes a rule by backtracking on previous generalizations until it finds a coherent one. As a consequence, search will take place during further generalization steps to integrate uncovered examples without introducing any contradiction.

Our algorithm, described Sec. IV, performs a revision M' of the model M whenever a newly encountered example contradicts either completeness or coherence. Each such counter-example x_u is stored in the set $M.X$, and every subsequent update will guarantee completeness and coherence *w.r.t.* $M.X$, thus preventing to ever reconsider M. This way, we can ensure convergence by storing counter-examples only.

A. Sketch of the algorithm

The system takes examples as defined in Sec. III. The examples are presented incrementally, each one possibly leading to an update of the action model. The method we propose is example-driven and bottom-up. The starting point is thus an empty rule-set; the interactions between the system and its environment produce rules by computing least general generalization (*l*gg) between examples, and between rules and examples.

x_1	$w(a), w(b), on(a,f), on(b,f)$	$move(a,b)$	$on(a,b)$ $\neg on(a,f)$
x_2	$w(b), w(a), on(b,f), on(a,f)$	$move(b,a)$	$on(b,a)$ $\neg on(b,f)$
x_3	$b(c), b(d), on(c,f), on(d,f)$	$move(c,d)$	$on(c,d)$ $\neg on(c,f)$
x_4	$w(a), b(c), on(a,f), on(c,f)$	$move(a,c)$	$b(a)$ $\neg w(a)$

Table I
Relational representation of examples of move actions in the colored block-world described in Sec. VI.

When a new counter-example x_u is identified, two kinds of modifications may occur:

- **Specialization:** x_u may contradict one or several rules of $M.R$ (such a rule necessarily pre-matches x_u). In that case, the preconditions of each such rule must be specialized in order to reject x_u, so as to preserve coherence.
- **Generalization:** If no rule $M.R$ pre-matches x_u, the algorithm (preserving completeness) then searches for a generalization of a rule of $M.R$ post-matching x_u that does not contradict any example in $M.X$ (preserving coherence). If no such generalization exists, x_u becomes a rule and is added as such to $M.R$.

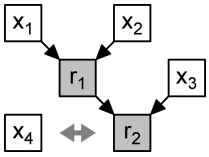

Figure 2. x_1 and x_2 are generalized into $r1$; then $r1$ and $x3$ are generalized into $r2$; x_4 leads to a backtracking specialization

In both cases, x_u is stored in $M.X$. Examples satisfied by the current hypothesis are not stored. They could later contradict a new hypothesis obtained after generalization. This is the price to pay so as to avoid memorizing all the examples (see Sec. V for a convergence analysis and discussion).

Example 3 (generalization): Referring to the examples of Tab. I, the system starts with an empty model. When x_1 is presented to the system, x_1 is learned and added as such in $M.R$. Then, example x_2 is presented, x_1 does not pre-match x_2, but x_2 post-matches x_1 (with inverse substitution $\rho^{-1} = \{b/X, a/Y\}$ and substitution $\sigma = \{X/a, Y/b\}$): the system thus computes the least general generalization of x_1 and x_2, producing rule r_1 (see Tab. II). When x_3 is presented to the system, it is not covered by the current model: r_1 post-matches x_3, but without pre-matching it. IRALe thus computes the least general generalization of r_1 and x_3, by dropping the color literals to build the resulting rule r_2.

In order to ensure convergence $M.X$ stores every past counter-example encountered. They are used to prevent repeating past over-generalizations, thus preserving the coherence of M. Let us now introduce data structures allowing to store counter-examples in rules.

Definition 8 (ancestors $r.anc$): Each rule r has a list of ancestors $r.anc$, each of them possibly having ancestors as well.

r_1	$w(X), w(Y), on(X,f), on(Y,f)$	$move(X,Y)$	$on(X,Y)$ $\neg on(X,f)$
r_2	$on(X,f), on(Y,f)$	$move(X,Y)$	$on(X,Y)$ $\neg on(X,f)$

Table II
Creation of r_1, as the *l*gg of x_1 and x_2. Replacement of r_1 by r_2, computed as the *l*gg of r_1 and x_3

rules/examples. When a *lgg* takes place, the resulting generalization keeps track of which rules/examples it comes

from. This way, each rule in $M.R$ is the top node of a memory tree as in Fig. 2. This structure is used during specialization.

Example 4 (specialization): Starting from r_2 as stated in Tab. II, if x_4 of Tab. I is presented to the system, x_4 contradicts r_2 (given the pre-matching substitution $\sigma = \{X/a, Y/c\}$), $x_4.e$ is not $on(a,c), \neg on(a,f)$. The specialization process recursively backtracks to ancestor generalizations of r_2, until it reaches one that does not contradict x_4 anymore, in our example r_1. This specialization replaces r_2 by r_1 in $M.R$, and makes x_3 uncovered.

Nodes are generalized rules, and leafs of the tree are the counter-examples in $M.X$.

Example 5 (revision): Following example Fig. 2, both x_4 and x_3 have to be integrated to the revised model r_1, coherent with x_3. Neither x_4 nor x_3 can be used to generalize r_1, and they cannot form together a general rule. The final version of the action model is therefore r_1, x_3 and x_4.

B. Detailed algorithm

Algorithm 1 is used only if the provided example x is not covered by the model M. If x contradicts at least one rule of $M.R$, a specialization algorithm (see Alg. 2) is first applied. This specialization algorithm updates the rule model so that x does not contradict a rule anymore, possibly rejecting a list of examples L_{x_u} (line 3, Alg. 1). After specialization, the obtained model is coherent, but it is still incomplete *w.r.t.* x and also possibly *w.r.t.* any formerly identified counter-examples in L_{x_u}. These have to be (re-)integrated to $M.R$ using the Algorithm 3 (line 6, Alg. 1), starting by x, and providing alternative generalization possibilities for $M.R$.

Algorithm 1 REVISION(M,x)

Require: An example x **uncovered** by a action model M
Ensure: A revised coherent/complete model M covering x
1: $L_{x_u} \leftarrow []$
2: **for all** $r \in M.R$ such that $x \nsim r$ **do**
3: $\quad L_{x_u} \leftarrow L_{x_u} + \text{SPECIALIZE}(r, x, M)$
4: **end for**
At this point, M coherence is ensured, but not completeness
5: **for all** $x_u \in [x] + L_{x_u}$ **do**
6: $\quad \text{GENERALIZE}(M, x_u)$
7: **end for**

Given an overly-general rule r and a counter-example x_u, Alg. 2 recursively looks for the hypothesis closest to r (*eg* a most general specialization of x) which is not contradicted[3] by x_u. Each time a backtracking step is performed, an over-general rule is replaced by its ancestor (more specific) rule. The associated counter-examples at the same hierarchical level are possibly not covered by M anymore. Thus they are stored in a list L_{x_u}. No formerly stored counter-example is lost at that step.

[3]Such an ancestor of r necessarily exists since the leafs of the tree are examples (*eg* Fig. 2), and we assume that the observation set is consistent.

Algorithm 2 SPECIALIZE(r,x,M):L_{x_u}

Require: An example x contradicting a rule $r \in M.R$
Ensure: A specialized model M coherent with x, a list L_{x_u} of counter-examples not covered by $M.R$ anymore
1: $M.R \leftarrow M.R \setminus \{r\}$
2: $L_{x_u} \leftarrow []$
3: **for all** $a \in r.anc$ **do**
4: $\quad$ **if** a is a leaf of M **then**
5: $\quad\quad L_{x_u} \leftarrow L_{x_u} + [a]$
6: $\quad$ **else if** $a \nsim x$ **then**
7: $\quad\quad L_{x_u} \leftarrow L_{x_u} + \text{SPECIALIZE}(a, x, M)$
8: $\quad$ **end if**
9: **end for**

The generalization algorithm (Alg. 3) has to deal with two kinds of examples. The first loop of the algorithm handles counter-examples x coming from L_{x_u} (line 3, Alg. 1), *i.e.* examples rejected after a specialization step of the model M, which are still covered by the model M; in this case, they should not be dropped (in case of potential further specialization steps), but rather be added to the ancestors' list of the most specific ones that cover them. Other counter-examples of L_{x_u}, which are not covered by M, are either generalized by means of the *lgg* algorithm, or added as such to the model, in case there is no well formed and coherent generalization of any rule of M that covers them. No counter-example is lost during the generalization algorithm, as they are either i) indexed as an ancestor of an existing rule ii) stored as an ancestor of a new general rule iii) added to M.

V. Convergence

Our revision algorithm memorizes each counter-example x_u in $M.X$ and revises the current action rule set $M.R$ so that the revised action rule set $M'.R$ is coherent and complete with respect to $M.X$, *i.e.* the whole set of counter-examples met so far. We call here an *EDS* algorithm any learning algorithm satisfying the above prerequisites, and we show that any *EDS* algorithm converges to an exact solution as far as *i)* either the hypothesis space H or the instance space $\mathcal{I} = \mathcal{S} \times \mathcal{A}$ is finite, and *ii)* there exists a solution M^* in H that is correct and complete w.r.t. the labeled instance space $\mathcal{I}_\mathcal{L}$ where each instance (s,a) is correctly labeled by the unique resulting effect e possibly made of a conjunction of literals. This means that we consider any example x as an instance/label pair $(i = (x.p, x.a)), l = x.e)$ and that the learning purpose is to find a hypothesis M such that $M.R$ always predicts the correct label of any instance i in $\mathcal{I}$, i.e. $M.R(i) = l(i)$.

Algorithm 3 GENERALIZE(M,x)

Require: An example x possibly uncovered by M, an action rule $r \in M.R$

Ensure: A generalization r' of r which covers x_u while preserving coherence. r' is inserted in $M.R$

$mod \leftarrow false$
for all most specific $r \in M.R$ such that $r \approx x$ **do**
  $r.anc \leftarrow r.anc \cup \{x\}$
  $mod \leftarrow true$
end for
if $mod = false$ **then**
  for all $r \in M.R$ **do**
    $mod \leftarrow mod \vee \mathrm{lgg(x, r, M)}$
  end for
end if
if $mod = false$ **then**
  $M.R \leftarrow M.R \cup \{x\}$
end if

Algorithm 4 lgg(x_u,r,M):mod

Require: An example x uncovered by M, an action rule $r \in M.R$ such that $r \overset{ae}{\sim} (x.a, x.e)$ given ρ^{-1}, σ and θ

Ensure: A generalization r' of r which covers x_u while preserving coherence (r' is inserted in $M.R$), a boolean mod stating if M is modified

$mod \leftarrow false$;
$p_{lgg} \leftarrow \mathrm{lgg}_{OI}((r.p)\rho^{-1}, x.s)$, given σ and θ
$r' = (p_{lgg}, (r.a)\rho^{-1}, (r.e)\rho^{-1})$
if $\mathrm{well-formed}(r')$ and there is no $x \in M.X, x \nsim r'$ **then**
  $mod \leftarrow true$
  $M.R \leftarrow M.R \cup \{r'\}$
  $r'.anc \leftarrow \{r, x\}$
end if

Proposition 1: Any *EDS* algorithm finds an exact solution M after i) less than $|\mathcal{I}|$ and ii) less than $|H|$ revisions.

Proof: i) Here we suppose $|\mathcal{I}|$ is finite and that there is an exact solution in the hypothesis space $|H|$. As counter-examples are memorized in $M.X$, the current hypothesis M is never revised into a hypothesis M' incorrectly predicting the labels of the counter-examples met so far. As there are less counter-examples than instances in $|\mathcal{I}|$ (i) is proved ii) each counter-example (i, l) prohibits all the revisions that incorrectly predict l, so as we suppose that an exact hypothesis M^* exists and all counter-examples are memorized, it follows that M^* will be found in less than $|H|$ revisions. ■

Memorizing counter-examples is the cheapest way to allow both examples and hypotheses enumeration: *a)* An algorithm that doesn't memorize counter-examples should memorize hypotheses in some way in order to enumerate them (as for instance, by maintaining as its current hypothesis the set of all the most general clauses consistent with the examples [16]) *b)* An algorithm that memorizes all examples met so far needs a memory size much larger than the number of revisions, and each revision is much more expensive. When storing counter-examples, the main complexity parameter is the number of counter-examples, *i.e.* the number of revisions.

Clearly, here counter-examples represent mistakes, and mistake bound analysis [17] is appropriate as examples are organized in episodes and depend on the learner activity, and so we cannot consider examples as drawn following some probability distribution.

VI. Empirical study

In order to compare our system IRALe with MARLIE [4] on its action model learning feature, we provide experimental results for both the *blocks-world* and the *Logistics* domains. Examples are generated in episodes: each episode starts from a random state and 30 sequential actions are then randomly performed. In order to estimate the accuracy of the model, classification error is measured as in MARLIE experiments: for a given number of learning episodes, a false positive rate FP, and a false negative rate FN are computed on 100 random test (s, a, s') triples. FP and FN represent the rate of atoms that were incorrectly predicted as true and false respectively in the resulting state when using the current action model. When performing an action in a given state, predictions are made by triggering a pre-matching rule and comparing its *add* and *del* effects to the observed effects. Among pre-matching rules, one rule with the lowest number of counter-examples is randomly selected. In a given state, several actions could be considered as illegal, meaning that in the true action model, performing this action fails and has no effect. However the learner is not aware of that. So in a given state, the learner randomly chooses among all possible instantiated actions. Each experiment consists of 10 trials and results are mean values on these trials.

A. Domains

Regular blocks world : In the blocks world, objects are either blocks (a, b ...) or the floor f. The predicates are $on(X, Y)$, $clear(X)$, $block(X)$, $move(X, Y)$. Three rules are necessary to represent the model of action $move(X, Y)$. The $move$ action is legal whenever both objects are clear. Our experiments consider 7-blocks worlds.

Colored blocks world: We now introduce color predicates as $b(X)$ (black) and $w(X)$ (white) in the regular blocks world. When $move(X, Y)$ is chosen, X is actually moved on top of Y only if X and Y have the same color. Otherwise, X is not moved but its colors shifts to the same color as Y. The 2-colors 7-blocks world is more challenging to learn as it contains 7 rules to model the action *move*, and leads IRALe to inadequate generalization choices, so enforcing

many revisions requiring its backtracking ability. Colors are uniformly distributed among the examples.

Logistics: In the (b, c, t)-Logistics setting, a state of the world describes b boxes, c cities, and t trucks. Available actions are $load(X, Y)$, $unload(X, Y)$ and $drive(X, Y)$, states are defined using the predicates $ontruck(X, Y)$, $truckin(X, Y)$ and $boxin(X, Y)$, as described in [4]. We report results regarding the hardest setting experimented in MARLIE, i.e. $(5, 5, 5)$-Logistics as well as results regarding the $(10, 10, 10)$-Logistics setting, in which the same action model has to be learned in a much larger state space.

B. Experimental results

Regular 7-blocks world and (5,5,5)-Logistics: Here, we compare our results with those of MARLIE in its action model learning experiments, as presented in [4] (Fig. 2). Figure 3 plots the classification errors FP and FN *vs* the number of learning episodes, in the 7-blocks world and in the (5,5,5)-Logistics domains. In both cases, the action model is almost always perfectly learned after less than 35 episodes, each producing 30 sequential examples. We note that *i)* FP and FN errors of IRALe are balanced, which is not the case of those obtained by MARLIE ii) MARLIE needs about 50 episodes to learn a good 7-blocks world action model (both FP and FN error $< 1\%$) where IRALe needs 10. MARLIE learns an accurate model but never reaches an exact action model in the (5,5,5)-Logistics case. To summarize, IRALe needs less examples than MARLIE to converge to an accurate action model. However, it happens that IRALe, as MARLIE, does not quickly reach the correct model. A better exploration, applying some kind of active learning, would certainly help to fine-tune the action model faster.

2-colors 7-blocks world and (10,10,10)-Logistics: The curves of the classification errors in the two domains are presented in Fig. 4. As the model to learn is more complex, IRALe needs more episodes to reach an accurate model when blocks are colored. Model size optimality is not a primary goal here, however a small number of rules indicates that an adequate level of generalization has been reached. Here, after few episodes, 3 rules are learned for the regular blocks world problem (lowest is 3) and 7 to 10 have been learned for the 2-colors problem (lowest is 7). This means that IRALe continuously restructures its action model all along the incremental learning process. Regarding the $(10, 10, 10)$-Logistics domain, the action model is learned as fast as in the $(5, 5, 5)$-Logistics domain despite the much higher number of possible states.

Learning activity and counter-example memory size: For the four domains, Fig. 5 displays the number of counter-examples stored by the learner *vs* the number of learning episodes. On the one hand, we note that the colored blocks world curve is much higher than the regular blocks world one, so confirming that the learning activity is related to the complexity of the model to learn, here estimated as the number of rules representing a same action. On the other hand, the (10,10,10)-Logistics curve is quite similar to the (5,5,5)-Logistics, again confirming that despite the increase in the state space, IRALe does not consider the learning task harder when the number of objects represented increases. As each episode represents 30 examples, the number of examples encountered before convergence is less than 600 examples, with less than 15 counter-examples stored for the three easiest problems, and less than 2400 examples, with less than 45 counter-examples stored for the hardest problem. Overall, these experiments show that storing counter-examples may help to reduce memory requirements a lot, while ensuring convergence in the deterministic case.

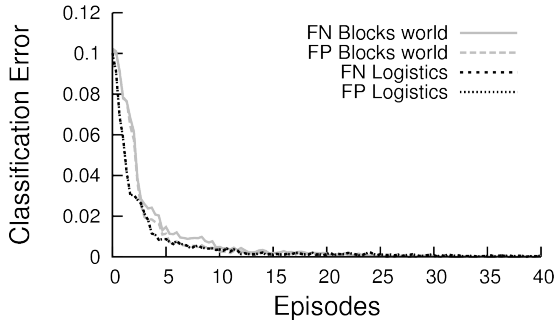

Figure 3. Classification error (*FP*, *FN*) in the 7-blocks world (in grey) and in the (5-5-5)-Logistics domains (in black).

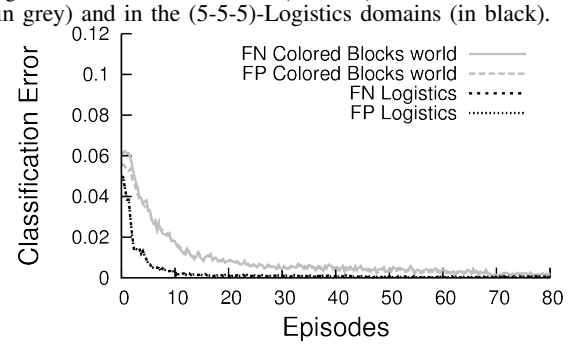

Figure 4. Classification error (*FP*, *FN*) in the 2-colors 7-blocks world (in grey) and in the (10-10-10)-Logistics (in black).

Learning a non determinist model: IRALe is designed to learn deterministic action models, and does not include any feature to handle noise or address overfitting. Nevertheless, we test here its ability to cope with some limited non-determinism in the effects of action. After [11], we consider the following noise in the 7-blocks world: whenever the action is to stack a block on top of another block, the action fails with probability p, and the block falls on the table. We then say that we apply a $p * 100$ % noise rate to this action. As a consequence, one of the three rules modeling *move* cannot be discovered by our deterministic learner, and so the complete action model is unreachable. Fig. 6 represents the global classification error of the $move$

model, mixing here false positive and false negatives, when learning with 10, 20 and 30 % noise rates. IRALe shows to be resistant to moderate levels of noise. Note that the number of counter-examples continuously increases as the model is never perfectly learned, and that the size of the model increases with the noise rate (data not shown).

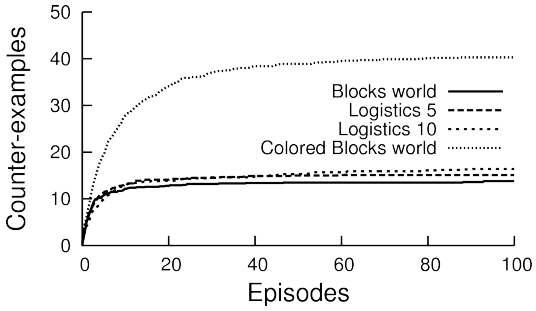

Figure 5. Counter-examples *vs* episodes in the four domains.

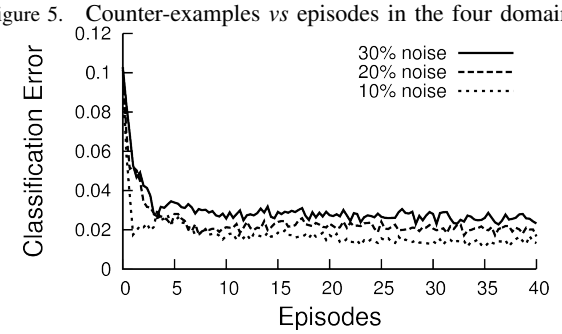

Figure 6. Classification error in the 7-blocks world domain with various noise rates.

ACKNOWLEDGMENT

We are very grateful to Tom Croonenborghs for useful discussions. This work has been partially supported by the French ANR HARRI project $JC08\_313349$.

VII. CONCLUSION

We have proposed an incremental relational revision algorithm IRALe that learns an action model as a set of rules. We have also provided a convergence proof for this algorithm, setting a loose upper bound on the number of counter-examples necessary to reach a complete and coherent action model in the deterministic case. We have also provided an empirical evaluation that shows the convergence of IRALe, with a number of counter-examples several order of magnitude smaller than the theoretical bound, suggesting that tighter bounds can be obtained when restricting the state space and the complexity of the class of models to be searched. Experiments of IRALe in the blocks world domain also show that it efficiently learns action models even when facing some limited non determinism in the action effects. We are currently including specific mechanisms to cope with noise and non determinism both in the effects of the actions and in the perceptions of the learner.

REFERENCES

[1] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] M. Van Otterlo, "The logic of adaptive behavior," Ph.D. dissertation, University of Twente, Enschede, 2008.

[3] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *ICML*, 1990, pp. 216–224.

[4] T. Croonenborghs, J. Ramon, H. Blockeel, and M. Bruynooghe, "Online learning and exploiting relational models in reinforcement learning," in *IJCAI*, 2007, pp. 726–731.

[5] K. Driessens, J. Ramon, and H. Blockeel, "Speeding up relational reinforcement learning through the use of an incremental first order decision tree algorithm," in *ECML, LNAI 2167*, 2001, pp. 97–108.

[6] W. Dabney and A. McGovern, "Utile distinctions for relational reinforcement learning," in *IJCAI*, 2007, pp. 738–743.

[7] S. Benson, "Inductive learning of reactive action models," in *ICML 1995*, 1995, pp. 47–54.

[8] Y. Gil, "Learning by experimentation: Incremental refinement of incomplete planning domains," in *ICML*, 1994, pp. 87–95.

[9] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition," in *ICML*, 1995, pp. 549–557.

[10] W. M. Shen, "Discovery as autonomous learning from the environment," *Machine Learning*, vol. 12, no. 1-3, pp. 143–165, 1993.

[11] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," *Jal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

[12] K. Driessens and J. Ramon, "Relational instance based regression for relational reinforcement learning," in *ICML*, 2003, pp. 123–130.

[13] T. J. Walsh and M. L. Littman, "Efficient learning of action schemas and web-service descriptions," in *AAAI*, 2008, pp. 714–719.

[14] F. Esposito, A. Laterza, D. Malerba, and G. Semeraro, "Locally finite, proper and complete operators for refining Datalog programs," in *Foundations of Intelligent Systems*, 1996, pp. 468–478.

[15] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli, "Multistrategy theory revision: Induction and abduction in INTHELEX," *Machine Learning*, vol. 38, no. 1-2, pp. 133–156, 2000.

[16] A. Dries and L. De Raedt, "Towards clausal discovery for stream mining," in *ILP*, 2009.

[17] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, no. 4, pp. 285–318, 1988.