Action Learning with Reactive Answer Set Programming: Preliminary Report

Michal Čertický
Department of Applied Informatics
Comenius University Bratislava, Slovakia
certicky@fmph.uniba.sk

***Abstract*—Action learning is a process of automatic induction of knowledge about domain dynamics. The idea to use Answer Set Programming (ASP) for the purposes of action learning has already been published in [2]. However, in reaction to latest introduction of *Reactive ASP* and implementation of effective tools [6], we propose a slightly different approach, and show how using the *Reactive ASP* together with more *compact knowledge encoding* can provide significant advantages. The technique proposed in this paper allows for real-time induction of action models in larger domains, and can be easily modified to deal with sensoric noise and non-determinism. On the other hand, it lacks the ability to learn the conditional effects.**

***Keywords*-ASP; learning; actions; induction;.**

I. Introduction

A. Action Learning and Reasoning about Actions

Knowledge about domain dynamics, describing how certain actions affect the world, is essential for planning and intelligent goal-oriented behaviour of both living and artificial agents. In artificial systems, this knowledge is referred to as **action model**. It is an expression of all the **actions** that can be executed in a given domain, with all their **preconditions** and **effects** in some kind of representation language.

Action learning, as a type of reasoning about actions, is a process of automatic generation and/or modification of action models based on sensoric observations. Autonomous and automated systems may benefit from action learning, since it allows them to adapt to unpredicted changes in environment's behaviour (caused for example by addition of new unknown agents, by hardware malfunction, etc.).

Action models in general are used specifically for the purposes of planning. As a motivating example, imagine an autonomous *Mars Rover* robot (similar to one described in [4]). Such robot acts in an unknown environment, but plans its actions based on a static hard-wired knowledge about about their effects. Now imagine, that one of its wheels gets damaged, which would change the effects of the *"moveForward"* action. If such robot was capable of action learning, it could update his action model accordingly, and continue acting successfully towards its goals.

B. Background and Methods

Recent action learning methods take various approaches and employ the wide variety of AI tools. We should mention the action learning technique based on heuristic greedy search, introduced in [16], perceptron algorithm-based method which can be found in [11], two solutions based on the reduction of action learning into different classes of satisfiability problems, available in [1] and [15], learning with inductive logic programs described in [12], or learning module written in Answer Set Programming (ASP), described by M. Balduccini in [2].

The method we propose in this paper is closest to Balduccini's learning module, in that it also uses ASP for induction and representation of action models. There are however several differences, that make it usable under different conditions. We will address these differences in detail in Section IV.

C. Reactive ASP

Answer Set Programming (ASP [7], [3]) has lately become a popular declarative problem solving paradigm, with growing number of applications [6], among others also in the field of reasoning about actions. Semantics of ASP enables us to elegantly deal with incompleteness of knowledge, and makes the representation of action's properties easy, due to nonmonotonic character of negation as failure operator [10]. With ASP, our knowledge is represented by so-called extended logic programs - sets of rules of the following form:

$$c \leftarrow a_1 \dots a_n, not\ b_1 \dots not\ b_m.$$

where a_i, b_j and c are literals, i.e., either first-order logic atoms, or atoms preceded by explicit negation sign "$\neg$". The "not" symbol denotes negation by failure. Part of the rule before "$\leftarrow$" sign is called *head* and part after it is *body*. Rule with an empty body (n = m = 0) is called a *fact* and rule without a head is an integrity *constraint*. Every logic program has a corresponding (possibly empty) finite set of **answer sets**.

Since we are dealing with dynamic systems, we can take advantage of so-called incremental logic programs. An incremental logic program is a triple $(B, P[t], Q[t])$ of logic programs, with a single parameter t ranging over natural numbers [5]. While B only contains **static** knowledge, parametrized $P[t]$ constitutes a **cumulative** part of our knowledge ($Q[t]$ is so-called **volatile** part, but we don't use it in our solution). In our method, t will always identify the most recent time step, and $P[t]$ will describe how the newest observation affects our previous beliefs.

In [6], Gebser et al. go further, and augment the concept of incremental logic programming with asynchronous information, refining the statically available knowledge. They also introduce their **reactive ASP solver** *oClingo* which we use in our solution.

II. Learning with Reactive ASP

We will now describe our learner, which is basically a short incremental logic program $(B, P[t], \emptyset)$. It has a large (but gradually decreasing) number of **answer sets**, each **corresponding to** a single **possible action model**.

At time step 1, the online ASP solver *oClingo* [6] computes the first answer set and stores it in memory (along with all the partial computations it has done so far). At every successive time step, we confront this answer set with new observations.

Note 1: The *oClingo* is a server application, which listens on a specified port and waits for new knowledge. This knowledge is sent to *oClingo* by a *controller* application, and in our case always represents the latest action and fluent observations.

A. Encoding the Action Model

First of all, we needed to choose the **encoding** of action models into a logic programs that is sufficiently expressive, but at the same time remains **as compact as possible**.

The main idea behind our encoding lies in the fact that for a given fluent F, every possible action A can either:

1) **cause** a fluent F to hold (we encode this by a fact `causes(A,F).`),
2) **cause** a **complementary** fluent $\neg F$ to hold (`causes(A,¬F).`),
3) or **keep the value** of that fluent literal (`keeps(A,F).` resp. `keeps(A,¬F).`).

In addition to that, we want our action models to contain the information about action's executability. In that respect, every action A can either:

1) **have** a fluent F as its **precondition** (encoded by a fact `pre(A,F).`),
2) or **not have** it as its **precondition** (`¬pre(A,F).`).

B. Generating Answer Sets

Answer sets corresponding to all the possible action models are generated by **static** part of our program (logic program B). It consists of the set of rules depicted in figure 1.

In the first part, we have three **choice rules**, that **generate answer sets** where A either causes F, causes $\neg F$, or keeps F. Next two **constraints filter out** the answer sets corresponding to **impossible models** - where A causes F and $\neg F$ at the same time, or where A both keeps and changes the value of F. Last two rules merely express the equivalence between two possible notations of A keeping F.

```
% Effect generator and axioms:
causes(A,F) ← not causes(A,¬F), not keeps(A,F).
causes(A,¬F) ← not causes(A,F), not keeps(A,F).
keeps(A,F) ← not causes(A,F), not causes(A,¬F).
← causes(A,F), causes(A,¬F).
← causes(A,F), keeps(A,F).
keeps(A,F) ← keeps(A,¬F).
keeps(A,¬F) ← keeps(A,F).

% Precondition generator and axioms:
pre(A,F) ← not ¬pre(A,F).
¬pre(A,F) ← not pre(A,F).
← pre(A,F), ¬pre(A,F).
← pre(A,F), pre(A,¬F).
```

Figure 1. Static (time-independent) part of our learner logic program.

Second part is very similar. Here we have two choice rules generating answer sets where A either has, or doesn't have a precondition F. Constraints here eliminate those answer sets, where A has and doesn't have precondition F at the same time, or where it has both F and $\neg F$ as its preconditions.

Note 2: Keep in mind, that the logic programs in this paper are simplified to improve the readability and save some space. You can download the exact ready-to-use ASP solver compatible encodings from [9].

C. Answer Set Elimination

Now, we need to process new knowledge received at successive time steps[1]. We use a time-aware, **cumulative** program $P[t]$ for that.

```
#external obs/2.
#external exe/2.
← obs(F,t), exe(A,t), causes(A,¬F).
← obs(¬F,t), obs(F,t−1), exe(A,t), keeps(A,F).
← exe(A,t), obs(¬F,t−1), pre(A,F).
```

Figure 2. Cumulative (time-aware) part of our learner logic program.

First two statements merely instruct *oClingo* that it should accept two kinds of binary atoms from the *controller* application[2]: fluent observations `obs` and executed actions `exe`.

Remaining **three constraints take care of actual learning**, by **disallowing answer sets** that are **in conflict with** the latest **observation**. First constraint says, that if F was observed after executing A, then A cannot cause $\neg F$. Second constraint tells us, that if F changed value after executing A, then A does not keep the value of F. And the last one means, that if the action A was executed when $\neg F$ held, F cannot be a precondition of A.

[1]Note, that we allow at most one action to be executed in every time step.

[2]Telling *oClingo* what to accept is not necessary, if we use new `--import=all` parameter. This option was implemented only after we designed our logic programs.

```
#step 9.
exe(move(b1,b2,table),9).
obs(on(b1,table),9).
obs(¬on(b1,b2),9).
obs(on(b2,table),9).
obs(¬on(b2,b1),9).
#endstep.
```

Figure 3. Observation example from Blocks World domain[14] sent to *oClingo* at time step number 9. It describes the configuration of two blocks $b1$ and $b2$ on the table after we moved $b1$ from $b2$ to the $table$.

At every time step, these constraints are added to our knowledge with parameter t substituted by a current time step number. Also, we need to add the latest observation. These observations are sets of `obs` and `exe` atoms. See the example of observation that is sent to *oClingo* in figure 3.

We say, that a constraint "fires" in an answer set, if its body holds there. In that case, this answer set becomes illegal, and is thrown away. Now recall, that in time step 1, *oClingo* generated the first possible answer set and stored it in memory. An observation like the one above can cause some of our constraints to fire in it and eliminate it. For example, if our answer set contained the `causes(`$move(b1, b2, table)$`,`$on(b1, b2)$`)` atom, the first constraint would fire.

If that happens, *oClingo* simply **computes** the **new answer set**, that is not in conflict with any of our previously added constraints. This is how we update our knowledge, so that we always have an action model consistent with previous observations at our disposal. Note, that each observation potentially reduces the number of possible answer sets of our logic program $(B, P[t], \emptyset)$, thus making our knowledge more precise. After a sufficient number of observations, $(B, P[t], \emptyset)$ will have only one possible answer set remaining, which will represent the correct action model.

III. Dealing with Noise and Non-determinism

The problem may arise, in the presence of sensoric noise or non-deterministic effects, since the noisy observations could eliminate the correct action model. This could eventually leave us with an empty set of possible models. However, we propose a workaround, that can overcome this issue.

The problem with non-determinism is, that we cannot afford to eliminate the action model after the first negative example. We need to have some kind of error tolerance. For that reason, we should modify the cumulative part of our program $P[t]$, so that our constraints fire only after a certain number of negative examples:

Here we can see, that our observations don't directly appear in the bodies of constraints. Instead, they are capable of increasing the negative example count (which is kept in the variable `C` of `negExCauses`, `negExKeeps` and `negExPre` predicates). Constraints then fire, when the number of negative examples is higher than 5.

```
negExCauses(A,F,C+1) ←
        negExCauses(A,F,C), obs(¬F,t), exe(A,t).

negExKeeps(A,F,C+1) ←
        negExKeeps(A,F,C), obs(¬F,t),
        obs(F,t−1), exe(A,t).

negExPre(A,F,C+1) ←
        negExPre(A,F,C), exe(A,t), obs(¬F,t−1).

← causes(A,F), negExCauses(A,F,C), C > 5.
← keeps(A,F), negExKeeps(A,F,C), C > 5.
← pre(A,F), negExPre(A,F,C), C > 5.
```

Figure 4. Modified $P[t]$ should be able to deal with sensoric noise, by introducing error tolerance.

Note 3: The *error tolerance threshold* is a numeric constant 5 here to keep things simple, but we can easily imagine better, dynamically computed threshold values (for example based on the percentage of the negative examples in the training set, etc.).

IV. Comparison to Alternative Method

Let us now take a closer look at the similarities and differences between our method and Balduccini's *learning module* approach described in [2].

	Inc. kn.	Prec.	Cond. eff.	Noise	Real-time
Balduccini	yes	yes	yes	?	no
Our method	yes	yes	no	yes	yes

A. Incomplete Knowledge

From the viewpoint of domain compatibility, both methods share the ability to deal with incompleteness of knowledge. The absence of complete observations may slow down the learning process[3], but cannot lead us to induce incorrect action models.

B. Action's Preconditions

Our method learns not only the effects, but also preconditions of actions. Similarly, Balduccini's learning module supports the induction of preconditions, in a form of so-called *impossibility conditions*.

C. Conditional Effects

Balduccini's learning module allows for direct induction of *conditional effects*, which is a greatest advantage over our method. Having the conditional effects allows for more elegant representation of resulting action models. (Note however, that they are not necessary, and can be expressed by a larger number of actions with right preconditions.) On

[3]By *slowing down* we understand, that we might require more time steps to induce precise action models. Incompleteness of observations will *not* hinder the computation times at individual time steps.

the other hand, we must keep in mind that learning them is in general harder and more time-consuming problem.

D. Encoding of Action Models

An action model is in the case of Balduccini's learning module encoded by a set of atoms of the following types: $d\_law(L)$, $s\_law(L), head(L, F), action(L, A)$, and $prec(L, F)$, with L substituted by a name (unique constant identifier) of a C-language [8] law, A by an action instance, and F by a fluent literal. Notice, that this way, we directly encode the syntactic form of individual C-language laws into logic programs. See figure 5 for a simplistic example of this encoding.

Following C-language law:

caused on(b1,table) after move(b1,b2,table), on(b1,b2), free(b1), free(table).

Is in Balducccini's learning module translated into:

```
d_law(dynamicLaw25).
head(dynamicLaw25,on(b1,table)).
action(dynamicLaw25,move(b1,b2,table)).
prec(dynamicLaw25,on(b1,b2)).
prec(dynamicLaw25,free(b1)).
prec(dynamicLaw25,free(table)).
```

Figure 5. Example of C-language $\rightarrow$ LP translation used in Balduccini's learning module.

Every time the observation is added, the whole history is confronted with this set. If the observation is not explained [2] by it, we add more atoms to it (either creating new laws, or adding effect conditions to existing ones).

In our case, the action model encoding is much more compact[4]. We don't translate an action model from any given planning language, which allows us to omit the d_law and s_law predicates and L parameter. Instead we have chosen an abstract, semantics-based, direct encoding of a domain dynamics, where every effect or precondition is represented by a single atom. Notice also, that the size of our action model doesn't increase over time.

E. Extending the Techniques

The bottom line here is, that our representation structures are simpler, for the price of lower expressiveness. The semantic-based encoding makes it fairly easy to extend learning by an ability to deal with noise. It is probable, that Balduccini's learning module could also be similarly extended, but it would be far less straightforward process (it consists of 14 rules, describing a syntactic structure of action model, rather than focusing directly on semantics).

[4]Note, that the main reason we can afford more compact encoding is the fact, that we don't use conditional effects.

F. Performance and Online Computation

Our method can be considered **semi-online**, in a sense that we only consider the most recent observation as relevant input for our computation. This is possible because of the use of **Reactive ASP**: At each time step, the solver keeps everything that it has computed at previous steps in memory, and only adds new observation. If the current model is disproved, some revisions might be needed, but significant part of the computation has already been performed and results are stored in memory. This, together with relatively compact encodings allows us to learn action models in **real-time**.

G. Experiments and Conclusion

In [2], we can find an experimental comparison of Balduccini's learning module and Otero and Varela's *Iaction* learning system [12]. Their experiment was conducted with 5 narratives of 4 blocks and 6 actions. *Iaction* system found a solution in 36 seconds on Pentium 4, 2.4GHz, while the results of Balduccini's module was fairly comparable with 14 seconds on somewhat faster computer (Pentium 4, 3.2Ghz).

To demonstrate the speedup resulting from using a compact encoding and Reactive ASP, we have decided to try significantly larger problem instance. Our domain was also a Blocks World with 4 blocks + table (`b1,b2,b3,b4,table`), but we had **32** possible **actions** (valid instances of `pickup(Block,From)` and `puton(Block,To)`) and our training set consisted of **150 observations** of randomly chosen legal actions.

Processing the training set with full observations took 17.9 seconds, while similar set with partial observations took 14.98 seconds. The experiment was performed with *oClingo* solver, version 3.0.92b, under 64bit Linux system with Intel(R) Core(TM) i5 3.33GHz CPU. The input files that we used can be downloaded from [9], together with detailed instructions.

We conclude, that the decrease in the size of encoding, together with preservation of results from previous time steps (using reactive ASP approach), enables us to learn action models considerably faster. This seems to result from the fact, that the complexity of answer set computation algorithms is exponential in the number of input atoms [13] (thus reducing the input for solver even a little can speed up the computation significantly), and that an important part of computation can be recycled from previous time steps.

In the near future, we are planning to provide an in-depth comparison of our method to a wide variety of action learning approaches, followed by a specification of the most apropriate practical applications in the area of autonomous and automated systems.

Acknowledgment

We acknowledge the support from VEGA project of Slovak Ministry of Education and Slovak Academy of Sciences no. 1/0688/10.

References

[1] E. Amir and A. Chang. *Learning partially observable deterministic action models*. Journal of Artificial Intelligence Research, Volume 33 Issue 1, pp. 349-402. 2008.

[2] M. Balduccini. *Learning Action Descriptions with A-Prolog: Action Language C*. In Proceedings of Logical Formalizations of Commonsense Reasoning, 2007 AAAI Spring Symposium. 2007.

[3] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press. 2003.

[4] T. Estlin et al. *Increased Mars rover autonomy using AI planning, scheduling and execution*. Proceedings 2007 IEEE International Conference on Robotics and Automation, pp. 4911-4918. 2007.

[5] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. *Engineering an Incremental ASP Solver*. In Proceedings of the 24th International Conference on Logic Programming (ICLP'08), pp. 190-205. 2008.

[6] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. *Reactive Answer Set Programming*. In Proceedings of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2011), pp. 54-66. 2011.

[7] M. Gelfond and V. Lifschitz. *Classical Negation in Logic Programs and Disjunctive Databases*. New Generation Computing, pp. 365-387. 1991.

[8] E. Giunchiglia and V. Lifschitz. *An Action Language based on Causal Explanation: Preliminary Report*. In proceedings of 15th National Conference of Artificial Intelligence (AAAI'98), pp. 623-630. 1998.

[9] www.dai.fmph.uniba.sk/upload/9/9d/Oclingo-learning.zip. Last accessed: 13 February 2012.

[10] V. Lifschitz. *Answer Set Programming and Plan Generation*. Artificial Intelligence, vol. 138, pp. 39-54. 2002.

[11] K. Mourao, R. P. A. Petrick, and M. Steedman. *Learning action effects in partially observable domains*. In Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 15-22. 2010.

[12] R. Otero and M. Varela. *Iaction, a System for Learning Action Descriptions for Planning*. In Proceedings of 16th International Conference on Inductive Logic Programming, ILP 06. 2006.

[13] P. Simons, I. Niemela, and T. Soininen. *Extending and Implementing the Stable Model Semantics*. Artificial Intelligence, 138(1-2), pp. 181-234. 2002.

[14] J. Slaney and S. Thiebaux. *Blocks World revisited*. Artificial Intelligence 125, pp. 119-153. 2001.

[15] Q. Yang, K. Wu, and Y. Jiang. *Learning action models from plan examples using weighted MAX-SAT*. Artificial Intelligence, Volume 171, pp. 107-143. 2007.

[16] L. S. Zettlemoyer, H. M. Pasula, and L. P. Kaelbling. *Learning probabilistic relational planning rules*. MIT TR. 2003.