Artificial Intelligence planning with p-stable semantics

Sergio Arzola[1], Claudia Zepeda[1], Mario Rossainz[1], and Mauricio Osorio[2]

[1]Benemérita Universidad Autónoma de Puebla,
Facultad de Ciencias de la Computación
[2]Universidad de las Américas, Puebla
sinrotulos@gmail.com,czepedac@gmail.com
rossainz@cs.buap.mx,osoriomauri@gmail.com
http://sites.google.com/site/gmlogyc
(Paper received on November 28, 2010, accepted on January 28, 2011)

Abstract. Our work is intended to model and solve artificial planning problems with logic based planning, using the novel semantics called *p-stable*, which is an alternative of stable semantics. It can be applied in a variety of tasks including robotics, process planning, updates, making evacuation plans and so on.

Keywords: planning, p-stable semantics, yale shooting problem.

1 Introduction

Currently, the Artificial Intelligence has more applications as well as has been taken relevance into processes and products of industries. One of the biggest branches of Artificial Intelligence is the study of planning, because resolution of planning problems has different applications in different areas too. For example, the robotics movement planning, creating evacuation plans, etcetera.
Planning in Artificial Intelligence is decision making about the actions to be taken.
Imagine an intelligent robot. The robot is a computational mechanism that takes input through its sensors and act with the effectors, which can be motors, lights, and so on. So the sensors allow the robot to perceive its environment and to build a representation of the world has perceived before as well as its immediate surroundings. Then the robot must act according to the representation of the world it has, that cames from its perception. The robot acts through its effectors which are devices that allow the robot to change the states of the environment interacting with it as changes the states of itself, like move from a place to another, move items, and so on. At an abstract level, a robot is a mechanism that maps its observations to actions which are obtained through sensors and performed by the effectors respectively. In this context. planning is the decision making, where gives a sequence of actions by a sequence of observations [13].
There are different approaches about planning done in different areas of artificial intelligence, however our focus here is into Logic-based Planning [7]. Furthermore, our proposal is using the p-stable semantics in order to model and solve

Advances in Computer Science and Applications
Research in Computing Science 53, 2011, pp. 189-197

planning problems.
The p-stable semantics is a novel semantics that came from an alternative for stable semantics [8]. There exists evidence about the applicably of p-stable in different domains. However, the purpose of this article is about the planning domain [14] [1] In this work, we are interested in two basic parts: the first one is the modeling in language $\mathcal{A}$, where it is intended to show how we can model easily a planning problem and the second one is how to translate it in a simple way from language $\mathcal{A}$ to p-stable semantics rules. In order to see what models we get, we use as resolver the last implementation of p-stable semantics [11].
This paper is structured as follows. In section 2 we introduce the general syntax of the logic programs used in this paper. We also provide the definition of stable and p-stable semantics. In section 3 we present the logic basic planning and the p-stable approach giving a basic example of code. Finally in section 4 we present the conclusions.

2 Background

In this section we summarize some basic concepts and definitions used to understand this paper.

2.1 Logic programs

A *signature* $\mathcal{L}$ is a finite set of elements that we call *atoms*, or *propositional symbols*. The language of a propositional logic has an alphabet consisting of *proposition symbols*: $p_0, p_1, \ldots$; *connectives*: $\wedge$, $\vee$, $\leftarrow$, $\neg$; and *auxiliary symbols*: (,). Where $\wedge$, $\vee$, $\leftarrow$ are 2-place connectives and $\neg$ is a 1-place connective. Formulas are built up as usual in logic. A *literal* is either an atom a, called *positive literal*; or the negation of an atom $\neg a$, called *negative literal*. The formula $F \equiv G$ is an abbreviation for $(F \leftarrow G) \wedge (G \leftarrow F)$. A *clause* is a formula of the form $H \leftarrow B$ (also written as $B \rightarrow H$), where H and B, arbitrary formulas in principle, are known as the *head* and *body* of the clause respectively. The body of a clause could be empty, in which case the clause is known as a *fact* and can be denoted just by: $H \leftarrow$. In the case when the head of a clause is empty, the clause is called a *constraint* and is denoted by: $\leftarrow B$. A *normal* clause is a clause of the form $H \leftarrow \mathcal{B}^+ \cup \neg\mathcal{B}^-$ where H consists of one atom, $\mathcal{B}^+$ is a conjunction of atoms $b_1 \wedge b_2 \wedge \ldots \wedge b_n$, and $\neg\mathcal{B}^-$ is a conjunction of negated atoms $\neg b_{n+1} \wedge \neg b_{n+2} \wedge \ldots \wedge \neg b_m$. $\mathcal{B}^+$, and $\mathcal{B}^-$ could be empty sets of atoms. A finite set of normal clauses P is a *normal program*.

Finally, we define $RED(P, M) = \{H \leftarrow B^+, \neg(B^- \cap M) \mid H \leftarrow B^+, \neg B^- \in P\}$. For any program P, the positive part of P, denoted by $POS(P)$ is the program consisting exclusively of those rules in P that do not have negated literals.

2.2 Stable and p-stable semantics

From now on, we assume that the reader is familiar with the notion of classical minimal model [10]. We give the definitions of the stable and p-stable semantics for normal programs.

Definition 1. *[12] Let P be a normal program and let $M \subseteq \mathcal{L}_P$. Let us put $P^M = POS(RED(P, M))$, then we say that M is a* stable model *of P if M is a minimal classical model of P^M.*

Definition 2. *[12] Let P be a normal program and M be a set of atoms. We say that M is a* p-stable model of P *if: (1) M is a classical model of P (i.e. a model in classical logic), and (2) the conjunction of the atoms in M is a logical consequence in classical logic of $RED(P, M)$ (denoted as $RED(P, M) \models M$).*

Example 1. Let P be the normal program $\{b \leftarrow \neg a, \quad a \leftarrow \neg b, \quad p \leftarrow \neg a \quad p \leftarrow \neg p\}$. We can verify that $M_1 = \{a, p\}$ and $M_2 = \{b, p\}$ model the rules of P. From the definition of the RED transformation we find $RED(P, M_1) = \{b \leftarrow \neg a, \quad a \leftarrow , \quad p \leftarrow \neg a, \quad p \leftarrow \neg p\}$, and $RED(P, M_2) = \{b \leftarrow, \quad a \leftarrow \neg b, \quad p \leftarrow, \quad p \leftarrow \neg p\}$. It is clear that $RED(P, M_1) \models M_1$ and $RED(P, M_2) \models M_2$. Hence M_1 and M_2 are *p-stable models* for P. It is easy to see that M_2 is stable model of P whereas M_1 is not.

The following theorem shows the relation between the stable and p-stable semantics for normal logic programs.

Theorem 1. *[12] Let P be a normal logic program and M be a set of atoms. If M is a stable model of P then M is a p-stable model of P.*

3 Planning based on p-stable semantics

In this section we present how we model planning into the p-stable semantics.

3.1 Logic-based Planning

In a planning problem, we are interested in looking for a sequence of actions that leads from a given initial state to a given goal state. There exist different action languages that are formal models used to model planning problems, such as $\mathcal{A}, \mathcal{B}$, or $\mathcal{C}$ [9]. A planning problem specified in one of these languages has a easy encoding in declarative logic languages based on p-stable semantics. In this Section we shall present a brief overview extracted from [4] about language $\mathcal{A}$, and the encoding of planning problems based on p-stable semantics.

3.2 Language $\mathcal{A}$

The alphabet of the language $\mathcal{A}$ consists of two nonempty disjoint sets of symbols **F** and **A**. They are called the set of fluents, and the set of actions. Intuitively, a fluent expresses the property of an object in a world, and forms part of the description of states of the world. A *fluent literal* is a fluent or a fluent preceded by $\sim$. A *state* σ is a set of fluents. We say a fluent f holds in a state σ if $f \in \sigma$. We say a fluent literal $\sim f$ holds in σ if $f \notin \sigma$. Actions when successfully executed change the state of the world. Situations are representations of the history of action execution. The situation $[a_n, \ldots, a_1]$ corresponds to the history where action a_1 is executed in the initial situation, followed by a_2, and so on until a_n. There is a simple relation between situations and states. In each situation s some fluents are true and some others are false, and this 'state of the world' is the state corresponding to the situation s.

The language $\mathcal{A}$ can be divided in three sub-languages: *Domain description language*, *Observation language*, and *Query language* [9,4].

Domain description language. It is used to express the transition between states due to actions. The domain description D consists of effect propositions of the following form: a **causes** f **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ where a is an action, $f, p_1, \ldots, p_n, q_1, \ldots, q_r$ are fluents. Intuitively, the above effect proposition means that if the fluent literals $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ hold in the state corresponding to a situation s then in the state corresponding to the situation reached by executing a in s the fluent literal f must hold. The role of effect propositions is to define a transition function, Φ, from states and actions to states. The domain description part also can include *executability conditions*: **executable** a **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ where a is an action and, $p_1, \ldots, p_n, q_1, \ldots, q_r$ are fluents. Intuitively, it means that if the fluent literals $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ hold in the state σ of a situation s, then the action a is executable in s.

Observation language. A set of observations O consists of value propositions of the form: **initially** f. Given a consistent domain description D the set of observations O is used to determine the states corresponding to the initial situation, referred to as *initial states* and denoted by σ_0.

Query language. We say a consistent domain description D in the presence of a set of observations O entails a query Q of the form f **after** $a_1, \ldots, a_m$ if for all initial states σ_0 corresponding to (D, O), the fluent literal f holds in the state $[a_m, \ldots, a_1]\sigma_0$. We denote this as $D \models_O Q$.

Hence, in order to model a planning problem using language $\mathcal{A}$, we must specify a triple (D, O, G) where D is a domain description, O is a set of observations, and G is a collection of fluent literals $G = \{g_1, \ldots, g_l\}$, which we will refer to as a goal. So, we require to find a sequence of actions $a_1, \ldots, a_n$ such that for all $1 \leq i \leq l$, $D \models_O g_i$ **after** $a_1, \ldots, a_n$. We then say that $a_1, \ldots, a_n$ is a *plan* for achieves goal G with respect to (D, O).

3.3 P-stable encoding of planning problems

We have described before, how to model planning problems using language $\mathcal{A}$. In this section we present a way to encode planning problems into p-stable semantics, since this semantics is new, there is no application made for planning purposes, but we can model the language $\mathcal{A}$ into rules of this semantics as following:

Vocabulary Fluents $f_1, \ldots, f_n$ can be declared by just making a rule of each fluent as a fact:
`fluent(`f_1`), ... , fluent(`f_n`).`
Similar as above, the actions $a_1, \ldots, a_n$ can be declared by presenting each action as a fact:
`action(`a_1`), ... , action(`a_n`).`

Encoding domain description. The propositions of the form: a **causes** f **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ can be declared by the following rule:
`holds(f, `T`+1)` $\leftarrow$ `occurs(a, T), holds(`$p_1, \ldots, p_n$`, T), not_holds(`$q_1, \ldots, q_r$`, T), time (T).`
Also prepositions of the following form:
executable a **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$. can be declared by the following rule:
`occurs(a,T)` $\leftarrow$ `holds(`$p_1, \ldots, p_n$`, T), not_holds(`$q_1, \ldots, q_r$`, T), time (T).`
In order to encode the rules, we express them with holds and occurs, which means that the fluent is satisfied or the action occurs at time T respectively.

Encoding observation language. These prepositions represents the initial state of the problem. It can be declared by giving it every observation as a fact at time 0:
`holds(`f_a`,0), ..., holds(`f_m`,0).`
`not_holds(`f_n`,0), ..., not_holds(`f_z`,0).`

Encoding query language. These prepositions declare the goal, or the wished state at time N, it is represented by giving what state we want to have at time N by its fluents as follows:
`holds(`f_a`,N), ..., holds(`f_m`,N).`
`not_holds(`f_n`,N), ..., not_holds(`f_z`,N).`

In the following section we give a brief example of a planning problem modeled into language $\mathcal{A}$ and into p-stable semantics in order to clarify this.

3.4 The yale problem modeled and encoded

Here we present the yale shooting problem, that consists of the following scenario: A turkey is initially alive and a gun is initially unloaded, the goal is to kill the turkey. In order to do it, we need to load the gun first and then shoot. We are going to add, that the final state of the gun will be loaded.

We present this planning problem modeled using language $\mathcal{A}$ and encoded into p-stable semantics. Briefly we remark that an $\mathcal{A}$ model is based on a set of fluents, actions, executable conditions, an initial state and a goal.

In language $\mathcal{A}$ Here we show how to model the problem into language $\mathcal{A}$. We can represent this with two fluents, which are loaded and alive. Loaded means that the gun may be or not loaded. Alive represents the state of the turkey, that can be alive or not alive. So our set or fluents are: {`loaded, alive`}. Also we only need two actions, load the gun and shoot. Our set of actions will be: {`load, shoot`}.
The *domain description* propositions are the executable conditions and, what causes an action A. For example: the load can not shoot if it is not loaded. Representing it into language $\mathcal{A}$ will be as:

```
executable shoot if loaded.
```

The following conditions are very easy to understand:

```
executable load if not loaded.
shoot causes not alive if alive.
shoot causes not loaded if loaded.
load causes loaded if not loaded.
```

The *observation language*, which declare the initial state of the problem,is that the turkey is alive and the gun is not loaded, so our *initially* state would be: {`alive, not loaded`}.
Finally the *query language* propositions, that mean the *goal*, which are to kill the turkey and let the gun loaded. Our set will be formed by the fluents: {`not alive, loaded`}.

In p-stable semantics In this section we present its encoding based on p-stable semantics. In particular we use the new implementation for p-stable semantics [11].
Briefly, we mention that is close similar from smodels [2], which you can represent planning by describing each fluent and action into clauses.
By giving the model of the language $\mathcal{A}$ it is very easy to model it into p-stable semantics. The *fluents* and *actions* can be represented respectively as follows:

```
fluent(loaded).
fluent(alive).
action(load).
action(shoot).
```

The next rule specifies that T in time(T) is an integer, and it can be from zero to N, here we set N as number three.

```
time(0..3)
```

The following rules are the *domain description* problem, as it has modeled before in language $\mathcal{A}$, in p-stable semantics is also very easy to understand.

```
occurs(shoot, T) :- holds(loaded, T), time(T).
occurs(load, T) :- not_holds(loaded, T), time(T).
not_holds(alive, T+1) :- occurs(shoot, T), holds (alive, T), time(T).
not_holds(loaded, T+1) :- occurs(shoot, T), holds(loaded, T), time(T).
holds(loaded, T+1) :- occurs(load, T),
not_holds(loaded, T), time(T).
```

Because it is not allowed to have negative atoms at the head of a rule, we use the auxiliary form as not_holds, in order to avoid inconsistency. If a fluent F holds at time T, it can not be that F not holds at the same time T. Representing this into code:

```
holds(F, T):- not not_holds(F, T), time(T), fluent(F).
```

The following rules represent the *observation language*, that is the initial state of the problem. As it has been shown before, the turkey is alive and the gun is not loaded. In order to translate this into code, we express it into the fact that it holds or not at time 0, which it is the initial time.

```
holds(alive, 0).
not_holds(loaded, 0).
```

Finally the rules of the *query language*, indicates the goal state that we want to have at time N, where N represents the number of steps. Here we need only three steps.

```
not_holds(alive, 3).
holds(loaded, 3).
```

With these rules, we have modeled the yale shooting problem. Then we use a recent implementation of p-stable semantics [11] in order to have the p-stable models that satisfy the conditions mentioned before. These models are the plan, that is the sequence of actions that must be made in order to archive the goal. This implementation use the lparse syntaxis and it is executed as following:

```
lparse program.lp | ./PstableResolver -p 0
```

We only obtained one model, that it is the plan. Because it could not be found another plan that satisfies the rules shown before. We explain the plan into the following table:

Time	Action	State
0	load	alive, not loaded
1	shoot	alive, loaded
2	load	not alive, not loaded
3	-	not alive, loaded

It is easy to see, that in order to kill the turkey and the gun be loaded, we need to first load the gun, then shoot at the turkey and finally load the gun again.

Comparing the results obtained in this example with p-stable models and answer sets, they both obtain models in different ways according to its semantics. However, in [5] it is proved that for a normal program the p-stable models contain the answer sets, which means that p-stable semantics can bring more plans, than stable semantics does for normal programs.

4 Conclusion

Planning involves the representation of actions and world models, reasoning about the effects of actions, and so on. We show that p-stable semantics is a good way to model and solve planning problems, giving us with the p-stable models the plans that we need, in order to go from an initial state to a goal. It can be applied in a variety of tasks including robotics, process planning, autonomous agents and spacecraft mission control [3]. We have explained in this paper how to model a planning problem into language $\mathcal{A}$ and how to translate into p-stable semantics and how to encode it. For future work, we are interested in create an interface for the planning grounding like coala [6] from the potassco project , which works with answer set solving, but instead of stable semantics apply the implementation of p-stable semantics which has been shown before.

References

1. J. J. Alferes, F. Banti, and A. Brogi. A principled semantics for logic programs updates. In *Nonmonotonic Reasoning, Action, and Change (NRAC'03)*, 2003.
2. ASP_Solver. Web location of Smodels: http://www.tcs.hut.fi/software/smodels/.
3. M. Balduccini, M. Gelfond, M. Nogueira, and R. Watson. Planning with the USA-Advisor. In D. Kortenkamp, editor, *3rd NASA International workshop on Planning and Scheduling for Space*, Oct 2002.
4. C. Baral. *Knowledge Representation, reasoning and declarative problem solving with Answer Sets.* Cambridge University Press, Cambridge, 2003.

5. J. L. C. Carranza. *Fundamentos matemáticos de la semántica pstable en programación lógica*. PhD thesis, Benemérita Universidad Autónoma de Puebla, Nov 2008.
6. Coala. http://www.cs.uni-potsdam.de/ tgrote/coala/.
7. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Non-Monotonic Logic Programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181. Springer-Verlag, 1997.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, second edition, 1987.
11. A. Marin. Computing the pstable semantics. https://sites.google.com/site/computingpstablesemantic.
12. M. Osorio, J. Arrazola, and J. L. Carballido. Logical weak completions of paraconsistent logics. *Journal of Logic and Computation, doi: 10.1093/logcom/exn015*, 2008.
13. J. Rintanen. *Introduction of Automated Planning*. Albert-Ludwings-Universitat Freiburg, 2006.
14. T. C. Son and E. Pontelli. Planning with preferences using logic programming. *Theory and Practice of Logic Programming (TPLP)*, 6:559–607, 2006.