Modeling and Encoding Automated Planning problems with the p-stable semantics

Sergio Arzola and Claudia Zepeda

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
sinrotulos@gmail.com,czepedac@gmail.com

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. Also we present a method to encode a general planning problem and then we present an example, which is the blocks world problem. This can be applied in a variety of tasks including robotics, process planning, updates, making evacuation plans and so on.
Keywords: planning, p-stable semantics, blocks world problem.

1 Introduction

Automated planning is a branch of artificial intelligence and it has been an area of research for over three decades. The automated planning is a key ability for intelligent systems, increasing their autonomy and flexibility through the construction of plans or sequences of actions in order to achieve their goals. This sequence of actions can be executed by intelligent agents, autonomous robots, and so on [15]. Planning techniques have been applied in a variety of tasks including robotics, process planning, autonomous agents, creating evacuation plans, etc.
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 [14].
Planning involves all the characteristics described before such as the representation of actions and world models, reasoning about the effects of actions, and

techniques for efficiently searching the space of possible plans. Therefore there are different approaches about planning done in different areas of artificial intelligence, however our focus here is into Logic-based Planning [8]. Furthermore, our proposal is using the p-stable semantics in order to model and solve planning problems.
The p-stable semantics is a novel semantics that came from an alternative for stable semantics [9]. There exists evidence about the applicably of p-stable in different domains [18] [1]. Furthermore, in [16] is presented a small example of how p-stable semantics can represent and solve planning problems through its last implementation [12].
In this work, we are interested in two basic parts: the first one is to present the action language $\mathcal{A}$, where it is intended to show how we can model easily a planning problem and the second one is to present a different method, which is more complete, of how we can model a planning problem with p-stable semantics rules based in the language $\mathcal{A}$. We will illustrate this by presenting as example the world blocks problem. In order to see what models we get, we use as resolver the last implementation of p-stable semantics [12].
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 with action language $\mathcal{A}$ and the p-stable approach, and then we present the world blocks problem represented in both, language $\mathcal{A}$ and p-stable rules. 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 [11]. We give the definitions of the stable and p-stable semantics for normal programs.

Definition 1. *[13] 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. *[13] 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. *[13] 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 $\mathcal{C}$ [10]. 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 [5] 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* [10,5].

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 more complete method to encode planning problems with background knowledge into p-stable semantics, since this semantics is new, there is no application made for planning purposes, but we can translate the language $\mathcal{A}$ model into p-stable rules as follows:

Background knowledge The background knowledge is declared as facts, but if it has elements from others, they are set as the body of the rule:
`background(`b_1`), ... , background(`b_i`).`
`compoundbackground(`B_j`)` $\leftarrow$ `background(`B_j`).`

Vocabulary Fluents $f_1, \ldots, f_n$ can be defined in two forms: if they do not have elements of the background knowledge they are delcared as facts, otherwise they are defined in terms of their background knowledge, where the the fluent be at the head and the background knowledge at the body respectively:
`fluent(`f_1`), ... , fluent(`f_i`).`
`fluent(`F_j, F_k`)` $\leftarrow$ `background(`F_j`) , background(`F_k`).`
`fluent(`F_n`)` $\leftarrow$ `background(`F_n`).`
Similar as above, the actions $a_1, \ldots, a_n$ can be declared in two ways: as facts if they do not have elements of the background knowledge, or as rules if they do:
`action(`a_1`), ... , action(`a_i`).`
`action(`A_j, A_k`)` $\leftarrow$ `background(`A_j`) , background(`A_k`).`
`action(`A_n`)` $\leftarrow$ `background(`A_n`).`
Also we need to set the time. It can be done, by defining a constant and then a fact called time where goes from 0 to the constant defined:

```
const length=t.
time(0..length).
```

Encoding domain description. The propositions of the form:
executable a **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$. can be declared by the following rule:

```
executable(A,T) ← holds(p_1,...,p_n, T), not holds(q_1,...,q_r, T),
        background(A), time (T), T<length.
```

Propositions of the following form:
a **causes** f **if** $p_1, \ldots, p_n, \sim q_1, \ldots, \sim q_r$ can be declared just as:

```
causes(action(A),fluent(F))← background(A), background(F).
```

In order to encode the rules, we express them with holds, which means that the fluent is satisfied at time T.
Because the holds rule need that executable rule and causes rule be truth, and both have the same conditions, then there is no necessary to add the conditions to the causes rule.
We use four auxiliary rules. The first two are to set what fluent must be set as true:
`literal(G) ← fluent(G).`
`literal(neg(G)) ← fluent(G).`
The second two help us to set the opposite of the truth value of a fluent:
`contrary(F, neg(F)) ← fluent(F).`
`contrary(neg(F), F) ← fluent(F).`
We define three holds rules. The first one refers to the initial state, which shows what fluents are satisfied at the beginning:
`holds(F,0) ← literal(F),initially(F).`
The second is for setting the fluents that are truth in the next time, which is $T + 1$, by some conditions of the previous time, which is T:
`holds(F, T+1) ← literal(F), time(T), T < length, action(A), executable(A,T), occurs(A,T), causes(A,F).`
The third is for setting the opposite of the fluent value for the next time:
`holds(F, T+1) ← literal(F), literal(G), contrary(F,G), time(T), T < length, holds(F,T), not holds(G, T+1).`
We also need to add the rule occurs and not_occurs, which means that the action A occurs or not at time T respectively. To define the rule occurs we use the auxiliary possible rule, that shows which actions are possible to execute at time T and if there is no evidence of achieving the goal:
`possible(A,T) ← action(A),time(T),executable(A,T), not goal(T).`
With the possible rule we define the rules occurs and not_occurs:
`occurs(A,T) ← action(A),time(T),possible(A,T), not not_occurs(A,T).`
`not_occurs(A,T) ← occurs(AA,T),action(A),action(AA),time(T),A!=AA.`

Encoding observation language. These prepositions represents the initial state of the problem. It can be declared by initially facts of what fluents are satisfied at the beginning:
`initially(`f_a`), ..., initially(`f_m`).`

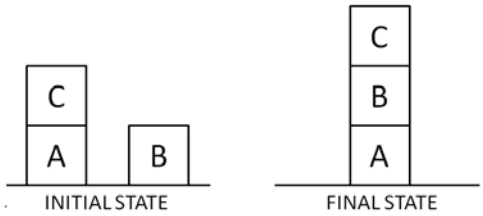
Encoding query language. These prepositions declare the goal, or the wished state at time N. It is represented as finally facts:
`finally(`f_a`), ..., finally(`f_m`).`
We use two auxiliary rules which help us to determine whether if the goal is reached or not. `not_goal(T) ← time(T), literal(X), finally(X), not holds(X,T).`
`goal(T) ← time(T), not not_goal(T).`
For last, the purpose of solving a planning problem is to find a plan in a given time. So we include two rules that indicates this to the program. The first indicates that exists a plan if the goal is reached according to the length of time.

`exists_plan` ← `goal(length).`
The second is a restriction which states that we do not want that a plan do no exist:
← `not exists_plan.`

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 worlds block problem modeled and encoded

Here we present the worlds block problem, that consists of the following scenario: There is a set of cubes (blocks) sitting on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved [17]. We are going to present the Sussman anomaly instance [2]:



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 $\mathcal{A}$.

First we are going to represent our *background knowledge*. As we can see is composed by the three blocks: {`block(a),block(b),block(c)`}.
Our set of *fluents* are: {`on(X,Y),ontable(X), clear(X),holding(X),handempty`}
The first fluent indicates the state of block X is on block Y. The second fluent indicates the state of block X is on the table. The third fluent shows that block X is clear, which means that there is no block above it. The fourth fluent indicates that block is holding by the hand. The last fluent indicates that the hand is empty.
We define four *actions*, which are: {`pick_up, put_down, stack, unstack`} These actions are the operations allowed to do. Pick up and put down refers to set or remove a block from the table. Stack and unstack refers to set or remove a block from another.

The *domain description* propositions are the executable conditions and, what causes an action A. We define a executable condition for each action:

```
executable pick_up(X) if clear(X), ontable(X), handempty.
executable put_down(X) if holding(X).
executable stack(X,Y) if holding(X), clear(Y).
executable unstack(X,Y) if clear(X), handempty, on(X,Y).
```

As well we define what causes each action:

```
pick_up(X) causes not ontable(X), not clear(X), holding(X), not handempty.
put_down(X) causes ontable(X), clear(X), not holding(X), handempty.
stack(X,Y) causes not holding(X), not clear(Y), clear(X), handempty, on(X,Y).
unstack(X,Y) causes holding(X), clear(Y), not clear(X), not handempty, not
on(X,Y).
```

The *observation language*, which declare the initial state of the problem is:

```
handempty, clear(c), clear(b), ontable(a), ontable(b), on(c,a).
```

Finally the *query language* propositions, that mean the *goal*, which is the configuration of the blocks stacked in decreasing order:

```
handempty, clear(c), on(c,b), on(b,a), ontable(a).
```

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 [12].
Briefly, we mention that is close similar from smodels [3], which you can represent planning by describing each fluent and action into clauses.
There are more instances of the blocks world problem in [12]. By giving the model of the language $\mathcal{A}$ and by the auxiliary rules we set above, it is very easy to model a planning problem into p-stable semantics.

First we define the time and our background knowledge:

```
const length=6.
time(1..length).
block(a).
block(b).
block(c).
```

Then the *fluents* and *actions* can be represented respectively as follows:

```
fluent(on(X,Y)) ← block(X), block(Y).
fluent(ontable(X)) ← block(X).
fluent(clear(X)) ← block(X).
fluent(holding(X)) ← block(X).
fluent(handempty).
action(pick_up(X)) ← block(X).
action(put_down(X)) ← block(X).
action(stack(X,Y)) ← block(X), block(Y).
```

```
action(unstack(X,Y)) ← block(X), block(Y).
```

Then we add the auxiliary rules defined before:

```
not_goal(T)← time(T), literal(X), finally(X), not holds(X,T).
goal(T) ← time(T), not not_goal(T).
exists_plan ← goal(length).
← not exists_plan.
```

The following rules are the *domain description* problem, as it has modeled before in language $\mathcal{A}$, so there are very easy to understand.

```
executable(pick_up(X), T) ← block(X), time(T), T < length,
        holds(clear(X), T), holds(ontable(X), T), holds(handempty, T).
executable(put_down(X), T) ← block(X), time(T), T < length,
        holds(holding(X),T).
executable(stack(X,Y),T) ← block(Y), block(X), time(T), T < length,
        holds(holding(X),T), holds(clear(Y), T).
executable(unstack(X,Y),T) ← block(Y), block(X), time(T), T < length,
       holds(clear(X), T), holds(on(X,Y), T), holds(handempty, T).
```

Because it is not allowed to have causes rules where an action has various effects, then it is required to define each effect in one rule.

```
causes(pick_up(X), neg(ontable(X))) ← block(X).
causes(pick_up(X), neg(clear(X))) ← block(X).
causes(pick_up(X), holding(X)) ← block(X).
causes(pick_up(X), neg(handempty)) ← block(X).
causes(put_down(X), ontable(X)) ← block(X).
causes(put_down(X), clear(X)) ← block(X).
causes(put_down(X), neg(holding(X))) ← block(X).
causes(put_down(X), handempty) ← block(X).
causes(stack(X,Y), neg(holding(X))) ← block(X), block(Y).
causes(stack(X,Y), neg(clear(Y))) ← block(X), block(Y).
causes(stack(X,Y), clear(X)) ← block(X), block(Y).
causes(stack(X,Y), handempty) ← block(X), block(Y).
causes(stack(X,Y), on(X,Y)) ← block(X), block(Y).
causes(unstack(X,Y), holding(X)) ← block(X), block(Y).
causes(unstack(X,Y), clear(Y)) ← block(X), block(Y).
causes(unstack(X,Y), neg(clear(X))) ← block(X), block(Y).
causes(unstack(X,Y), neg(handempty)) ← block(X), block(Y).
causes(unstack(X,Y), neg(on(X,Y))) ← block(X), block(Y).
```

We add the auxiliary rules mentioned earlier.

```
literal(G) ← fluent(G).
literal(neg(G)) ← fluent(G).
contrary(F, neg(F)) ← fluent(F).
contrary(neg(F), F) ← fluent(F).
holds(F, 1) ← literal(F), initially(F).
```

```
holds(F, T+1) ← literal(F), time(T), T < length, action(A),
        executable(A,T),occurs(A,T), causes(A,F).
holds(F, T+1) ← literal(F), literal(G), time(T), T < length,
        contrary(F,G), holds(F,T), not holds(G, T+1).
possible(A,T) ← action(A), time(T), executable(A,T), not goal(T).
occurs(A,T) ← action(A), time(T), possible(A,T), not not_occurs(A,T).
not_occurs(A,T) ← action(A), action(AA), time(T), occurs(AA,T), A!=AA.
```

The following rules represent the *observation language*, that is the initial state of the problem. As we have shown before, we represent them by initially facts:

```
initially(handempty).
initially(clear(c)).
initially(clear(b)).
initially(ontable(a)).
initially(ontable(b)).
initially(on(c,a)).
```

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. This has been defined by the const length.

```
finally(handempty). finally(clear(c)). finally(on(c,b)). finally(on(b,a)).
finally(ontable(a)).
```

With these rules, we have modeled the blocks world problem. Then we use a recent implementation of p-stable semantics [12] 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, because it could not be found another plan that satisfies the rules shown before in 6 steps. We explain the plan into the following table:

Time	Action	State
0	unstack(c,a)	clear(a), clear(c), ontable(a), ontable(b), on(c,a), handempty
1	put_down(c)	clear(a), clear(b), ontable(a), ontable(b), holding(c)
2	pick_up(b)	clear(a), clear(b), clear(c), ontable(a), ontable(b), ontable(c), handempty
3	stack(b,a)	clear(a), clear(c), ontable(a), ontable(c), holding(b)
4	pick_up(c)	clear(b), clear(c), ontable(c), ontable(a), on(b,a), handempty
5	stack(c,b)	clear(b), ontable(a), on(b,a), holding(c)
6	-	clear(c), ontable(a), on(c,b), on(b,a), handempty

It is easy to verify that the plan is correct.

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 [6] 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 [4]. We have explained in this paper how to model a planning problem into language $\mathcal{A}$ and a more complete method of 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 [7] from the potassco project , which works with answer set solving, but instead of stable semantics apply the newest implementation of p-stable semantics [12].

5 Funding

This work was supported by the CONACyT [CB-2008-01 No.101581].

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 DLVk: http://www.dbai.tuwien.ac.at/proj/dlv/k/.
3. ASP_Solver. Web location of Smodels: http://www.tcs.hut.fi/software/smodels/.
4. 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.
5. C. Baral. *Knowledge Representation, reasoning and declarative problem solving with Answer Sets*. Cambridge University Press, Cambridge, 2003.
6. J. L. C. Carranza. *Fundamentos matemáticos de la semántica pstable en programación lógica*. PhD thesis, Benemérita Universidad Autónoma de Puebla, Nov 2008.
7. Coala. http://www.cs.uni-potsdam.de/ tgrote/coala/.
8. 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.

9. 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.
10. M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, second edition, 1987.
12. A. Marin. Computing the pstable semantics. https://sites.google.com/site/computingpstablesemantic.
13. 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.
14. J. Rintanen. *Introduction of Automated Planning*. Albert-Ludwings-Universitat Freiburg, 2006.
15. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pretince Hall, 2009. 1152 pages. ISBN: 0136042597.
16. C. Z. Sergio Arzola and M. Osorio. Artificial intelligence planning with p-stable semantics. *ENC 2011*, 2011.
17. J. Slaney and S. Thibaux. Blocks world revisited. *Artificial Intelligence 125*, pages 119–153, 2001.
18. T. C. Son and E. Pontelli. Planning with preferences using logic programming. *Theory and Practice of Logic Programming (TPLP)*, 6:559–607, 2006.