Getting Started With Automated Planning

Practical Tips and Resources

Dominik Schreiber
Last updated 2019-01-09

1 Introduction to PDDL

The Planning Domain Description Language (PDDL) is a standardized language for expressing planning problems. It is used as the official format for the International Planning Competitions (IPC)[1] and has lots of enhancements to be useable for other planning types than classical planning as well.
A PDDL specification of a planning problem consists of two files: the domain file and the problem file. In the domain file, we describe the invariant rules of our world model, like object types, conditions of the world state, and possible actions to perform. The problem file is based on the domain file and describes one concrete problem, specifying the objects which are part of the problem, an initial state, and the goals to fulfil.

1.1 Building blocks for modeling a problem

The general syntax of a PDDL file can be easily retrieved by taking any classical planning PDDL files and modifying them based on your specific needs. For this reason, we will present the PDDL features as a number of "building blocks" which are needed to model your own planning problems.

1.1.1 Types

Everything that we model in PDDL is based on a set of *objects*, each of which belongs to certain types. Creating problems without any typing is possible in PDDL, but for most scenarios it is a much cleaner approach to define types for the objects of the world we model. Types can be defined as follows:

```
(:types location moving-object - object
        box person - moving-object)
```

In the first line, we define two different types `location` and `moving-object`, both of a common supertype `object`. In the second line, we define two subtypes for the type `moving-object`. This allows for simple polymorphism: an object of type `box` will now also have the types `object` and `moving-object`.

1.1.2 Constants and Variables

There are generally two different ways to refer to objects in our world: As a constant or as a variable. When we use a constant, then we know exactly which specific object we are referring to. For instance, `yard` and `house` could be constants of type `location`, and `alfred` could be a constant of type `person`. All relevant objects of the modeled world are introduced as constants.
In contrast, when we want to argue about *any* applicable object in our world of some type, we define and use variables, like the expression `(?l1 ?l2 - location)` to refer to two arbitrary objects of type `location`. Note that variables are always written with a '`?`' as a prefix, while constants are written without it. This provides a simple way to distinguish between the two.

[1] `http://icaps-conference.org/index.php/Main/Competitions`

1.1.3 Predicates

A predicate is an atomic statement which is used to express certain conditions in the logic of a planning problem.

```
(at ?o - moving-object ?l - location)
```

This is a binary predicate which takes one `moving-object` and one `location` as arguments. For each applicable pair of constants, in every possible world state, it is either true of false. (Of course, the predicate may be irrelevant for some combinations of constants – clever procedures inside a planner will dispose of these instantiations, or not even create them.)
For each predicate `p`, the expression `not(p)` naturally refers to its negation. There is also a special predicate `(= ?x ?y)` to denote equality; where it is supported, it will evaluate to `true` if and only if `?x` and `?y` have the identical value of some constant `c`.

1.1.4 Actions

This is how an action may look like:

```
(:action move                                          ; Name of the action
    :parameters (?p - person ?l1 ?l2 - location)       ; parameters
    :precondition (and                                 ; preconditions
        (at ?p ?l1)
    )
    :effect (and                                       ; effects
        (not (at ?p ?l1))
        (at ?p ?l2)
    )
)
```

Actually, these lines just define an action *template*, what we also call an *operator*. When setting two constants of type `location` as the parameters and propagating these values to the remaining body of the operator, we get an actual *action*, like `move(alfred house yard)` with the precondition `(at alfred house)` and the effects {`not(at alfred house), (at alfred yard)`}. The basic idea is that the "user" of a planning environment only writes the basic template for the actions in his problem model, and the planner does all the work of creating actions. This process of creating actions out of operators and atoms out of predicates is called *grounding* or *instantiation*.

1.1.5 Conditional effects

Conditional effects are a slightly more advanced PDDL feature. They are added to an action definition just like a normal effect. They will not have any impact on the applicability of an action in any given state. But when the action *is* applied to a state, then the conditions of the conditional effect will be checked, and if they hold, the action has *additional* effects on the resulting state.

```
(when (at ?p2 ?l2) (and (chatting ?p ?p2) (chatting ?p2 ?p)))
```

This could be an additional, conditional effect for above action. When there is already another person at position `?l2`, then the two people will be chatting with each other. Multiple predicates are legal both for the conditions and the consequences of an additional effect, provided that they are enclosed by an `(and )`.

1.1.6 Universal quantifications

Sometimes, an action should have some kind of "global" effect; it should have an impact on all objects of some kind which are known in the problem model. Likewise, we sometimes require some global properties to hold in order to execute an action or to satisfy a planning goal. For these scenarios, we can use quantifications. Just like the mathematical notation, e.g. $\forall x \in \mathbf{R} : P(x)$, we use a set of typed variables to quantify over all constants of such a type.

```
(forall (?p - person ?p2 - person) (not (chatting ?p ?p2))
```

This universal quantification expresses that no pair of people may be chatting with each another.
Universal quantifications can be part of action preconditions and effects as well as the goal definition.
In the PDDL standard, there are also existential quantifications ($\exists$) which are a bit more complex to realize from the viewpoint of a planner. They are described in section 1.1.8.

1.1.7 Action costs

When aiming for an optimal plan, it is practical or even necessary to attribute some distinct cost to each action. In its most basic form, this can be realized by defining a function `total-cost` which is initialized to zero in the initial state and increased by a constant amount when executing an action, provided that its effects feature an expression like this:

```
(increase (total-cost) 3)
```

The exact syntax of how to define such an action cost function and how to set it as the metric to be minimized is not discussed here, but can be easily seen in PDDL files dealing with such measures.

1.1.8 Disjunctive conditions (ADL features)

So far, we restricted the preconditions (and prerequisites of conditional effects) of the problem to one big conjunction (AND) of predicates. Universal quantifications meet this restriction as well, given that we instantiate all predicates in the body of the quantification by assigning each possible combination of constants to the quantified variables.
For even more advanced problem modeling, the conditions can be generalized to *any logical expression* of function-free first-order logic. When supported, complex conditions can be assembled:

```
(and
  (or (at ?p kitchen) (at ?p garden))
  (not (and (at ?p kitchen) (at ?p garden)))
  (imply (at ?p kitchen) (cooking ?p))
)
```

In this example, the top-level logical connective is an AND of three conditions which all have to hold. The first condition expresses that person `?p` is at the kitchen `or` at the garden – or, technically, at both.
This is refined by the second condition, saying that `?p` must not be at both of these places (logical NAND). Note that the `not` expression now encloses a non-atomic expression, which is not possible when only admitting conjunctive conditions.
The third expression is an implication: It says, `?p` being at the kitchen implies that `?p` is cooking. So this condition becomes true if `?p` is not at the kitchen or if `?p` is cooking.
In addition, the non-conjunctive evaluation style of conditions allows for existential quantifications:

```
(exists (?p1 ?p2 - people) (chatting ?p1 ?p2))
```

In analogy to universal quantifications, the previous example expresses that there is *some* pair of people (i.e. at least one) which is chatting. While universal quantifications can be resolved to a big AND expression, existential quantifications can be resolved to a big OR expression.
The advanced constructs just described belong to a feature set called ADL (Action Description Language), which is obviously not supported by all planners. If making use of ADL features, the semantics of a planning model become more complex. This can have an impact on the run times of a grounding and planning system. On the other hand, more complex logical scenarios can be modeled, which sometimes also leads to a more compact and thus more efficient domain description.
Keep in mind that disjunctive conditions can only be used in preconditions, goals, and in the prerequisites of conditional effects. Effects of an action still need to be strictly conjunctive – anything different would lead to non-deterministic action effects.

1.1.9 Derived predicates

Derived predicates are the mechanism of PDDL to account for *domain axioms*; properties of our world state which can be derived from the state itself. They are defined directly after the `(:predicates)` block like this:

```
(:derived (at-house ?p) (or (at-kitchen ?p) (at-garden ?p) (at-livingroom ?p)))
```

Here, a derived predicate `(at-house ?p)` is defined to be derived as true whenever the subsequent condition is met. Following the closed world assumption, `(at-house ?p)` is assumed to be false otherwise. The derived predicate can then be used just like any normal predicate inside preconditions and goals. Evidently, it cannot be part of an effect (except in a conditional effect's prerequisite), because a derived predicate changes not explicitly, but implicitly according to its definition.
As in the given example, derived predicates can help to create convenient and intuitive conditions without the need of re-defining them in action effects every time their logical backbone changes. Derived predicates can contain other derived predicates, or even themselves recursively, in their definition; but the extent to which this is supported varies greatly among planners.

1.1.10 Numeric planning

In classical planning, world states are restricted to be a set of Boolean variables. We lift this restriction in *numeric planning* and also allow for integer- and real-valued variables in our states.
Modeling numeric planning problems in PDDL boils down to the following aspects:

1. Define a number of *numeric fluents*, or *functions* which complement the set of predicates, but map to a numeric value instead of a Boolean value.
2. Enhance the domain logic with numeric fluents by adding *numeric preconditions* (i.e. comparisons between numeric values) and *numeric effects* (i.e. updating values of fluents) to actions.
3. Set the value of each numeric fluent in the initial state (just like defining all initial atoms).

Say we want to introduce a planning domain for filling a rucksack. We define functions as follows:

```
(:functions (weight ?o - object) - number
            (rucksack-weight) - number
            (strength ?p - person) - number)
```

The function `weight` maps a `rucksack` object to its current weight. `(rucksack-weight)` has no arguments, so it is just a plain numeric variable indicating the current weight of the rucksack to fill. The function `strength` maps a `person` to the maximum weight he or she can carry.
Now we can define an action `(put-in-rucksack ?o - object)` as follows:

```
(:action (put-in-rucksack)
  (:parameters ?o - object)
  (:precondition (and
    (not (in-rucksack ?o))
    (exists (?p - person) (>= (strength ?p) (+ (rucksack-weight) (weight ?o))))
  ))
  (:effect (and
   (in-rucksack ?o)
   (increase (rucksack-weight) (weight ?o))
  ))
)
```

A couple of things are interesting here: Firstly, we have a numeric condition inside one of the preconditions. It has three parts: The comparator `>=`, and then two numeric expressions. The second numeric expression is the sum of two other expressions (beware the prefix notation of all operators and comparators in this setting!). The complete condition says that some person must exist who has enough strength to lift the combined weight of the rucksack and the object to add.
Secondly, there is a numeric effect: The fluent `(rucksack-weight)` is increased by the value of `(weight ?o)`.
Valid comparators for numeric conditions are `=`, `>=`, `<=`, `<`, and `>`. For numeric operators, the four basic operations (`+ - * /`) are supported. In addition to `increase`, there are also the keywords `assign` (assignment of a value), `decrease` (subtraction), `scaleUp` and `scaleDown` (multiplication and division).
The numeric fluents are still missing their initial values. We define these just like atoms in the initial state (`(:init )` block):

```
(= (rucksack-weight) 0) (= (strength linda) 30) (= (strength ben) 40)
(= (strength carlo) 20) (= weight cellphone 1) (= (weight cement-block) 100) ...
```

Only constant assignments are supported here. Also keep in mind that we are still doing planning under certainty, so all fluent values must be known exactly.
According to PDDL standard, the goal definition may contain numeric conditions, too, although it does not seem common to make use of this.

Generally, making use of numeric PDDL features should be considered carefully. The set of planners which support them is limited, and the problem becomes significantly less manageable than before (in particular, state space becomes infinite). Nonetheless, many real-world problems require numeric variables in order to be modeled accurately, and at the very least it is nice to know that PDDL provides a way to create such models.

1.2 Editing and troubleshooting

Designing a PDDL file is not unlike writing a website or a program: you will make mistakes, and you will need to find them when something does not work. For PDDL files, we recommend trying the following options:

1.2.1 editor.planning.domains

The online PDDL editor editor.planning.domains[2] features not only PDDL syntax highlighting and auto completion, it also includes a number of plugins which may help debugging PDDL files. In

[2] `http://editor.planning.domains`

particular, the plugins Solver (to run an external planner on your files) and Torchlight (to analyze a planning problem) are interesting for debugging and testing.

1.2.2 Running a planner

Depending on the planner, any error output will be more or less comprehensive, or in bad cases even non-existent. On the flipside, trying your PDDL files on different planners with success and with reasonable results increases the probability that the syntax is correct and your model makes sense.
We present the Aquaplanning framework in the next section. Usually, its parsing error output should be reasonably comprehensive such that you might find the error just by following the given line/column numbers and the error text. Still, you should note that input errors sometimes emerge at positions which differ from their true origin, just like in debugging program code.
One last thing to keep in mind: Most planners only support some subset of PDDL, so a planner might complain about a feature it does not know even when your files are completely fine. That being said, the PDDL features introduced before should generally work on competitive planners.

2 Software

In the following, some software of interest is presented and discussed.

2.1 Aquaplanning

The framework *Aquaplanning*[3] has been specifically developed for the *Automated Planning and Scheduling* course. We felt that there is no allrounder PDDL library which both supports some advanced features of the PDDL language and is still somewhat "educative" and easily extensible regarding the source code. Ideally, high quality submissions by students can be merged into the framework, adding features as we go.

You find general information (also on downloading and installing) on the Aquaplanning Github page linked below. There are only a few dependencies, and the Maven build system resolves these automatically.

When diving into the code, the `Main` class in the base package is a good starting point (obviously). The generic three-step approach of running a planner is seen there:

1. Parse the problem, receiving some object representation of the lifted problem.
2. Ground the problem, resulting in a representation without any variables.
3. Do the actual planning.

It should not be necessary to look into the details of parsing PDDL files (in `aquaplanning.parsing`), except if you want to know some highly specific stuff about how the files are being interpreted. The PDDL features which Aquaplanning supports are described on the Github page / in the Readme. The supported subset of PDDL should suffice to already model quite complex planning problems.
The model of lifted planning problems is located in `aquaplanning.model.lifted`. Similarly, objects for representing ground problems are located in `aquaplanning.model.ground`. The most important thing to know is that you can access all information on some planning problem by querying the corresponding instance of `PlanningProblem` (in lifted form) and/or of `GroundPlanningProblem` (in ground form).

[3] `https://github.com/domschrei/aquaplanning`

The grounder to be used is called `RelaxedPlanningGraphGrounder`. Just tell it to ground a problem, and a ground problem will be returned. Quantifications will be resolved into flat lists of atoms, and equality predicates are treated as normal predicates for which `(= c c)` will be added to the initial state for each constant `c`. Conditional effects will still be in their explicit conditional form after grounding (in contrast to being compiled out of the problem by introducing additional actions). In case of planning with ADL features, the tree-like structure of logical expressions is preserved just as specified within the problem domain.
Planning with a ground planning problem is generally easier than dealing with all the parameters, constants and object types of a lifted problem. On the other hand, some aspects of planning like the computation of some heuristic can be more convenient when dealing with the lifted representation of the problem, because more information is available there and the representation is less rigid. When implementing something, ideally try to make a choice early on with which representation you will work, as switching from one to the other may be somewhat tedious.

2.2 Efficient off-the-shelf planners

To test, debug, and finally to actually solve your own planning problems, you may want to use a state-of-the-art planning system with good performance. Nonetheless, using the planner should be easy and self-explanatory. In the following, we recommend some planners which are (more or less) good choices in that regard.

2.2.1 Madagascar

The Madagascar planning system is based on translating planning problems into propositional logic and then resolving it using a SAT Solver. It is quite fast on many domains because it attempts to execute multiple actions at the same step in a potential plan.
The plug-and-play executables of Madagascar can be downloaded from here[4]. It is probably enough to just use the executable called `M` (which is one of the three versions `M`, `Mp`, and `MpC`). Start it like this:

```
./M path/to/domain.pddl path/to/problem.pddl -Q
```

The `-Q` flag causes Madagascar to output the plan in sequential form (i.e. the found actions are de-parallelized into a total order).

2.2.2 Metric-FF

Metric-FF is a heuristic state space planning system. It features a couple of interesting heuristics leading to good performances. Metric-FF can be downloaded from here[5] and built with a simple `make` call (requiring `flex` and `bison`). We recommend to launch it like this:

```
./ff -p path/to/ -o domain.pddl -f problem.pddl -s 0
```

With `-s 0`, a search configuration which does *not* depend on goal metrics is used. When you have action costs defined in your problem, you can also discard this argument.

2.2.3 Fast Downward

A bit more complicated to use, the planning suite Fast Downward has loads of features, options and arguments to pick from. Extensive documentation on how to aquire, install, and use the planner is

[4] `https://research.ics.aalto.fi/software/sat/madagascar/`
[5] `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`

available here[6]. For example, one easy way to launch a planning procedure after building it is the following:

```
python fast-downward.py --alias seq-sat-lama-2011 path/to/domain.pddl
                        path/to/problem.pddl
```

Keep in mind that with many configurations, the planner will keep calculating after it already found a solution in order to optimize the plan. Be careful when using this planner suite that your arguments make sense – it is probably best to stick with the recommended configurations[7] if you are unsure.

[6] http://www.fast-downward.org/

[7] http://www.fast-downward.org/PlannerUsage