PDDLyte
A Partial Implementation of The Planning Domain Definition Language

John Martin Jr.
jdm2213@columbia.edu

February 9, 2014

Description

The PDDLyte language, whose name derives from the Planning Domain Definition Language (PDDL)[1], is a specification language used to formulate and solve planning problems. Similarly to PDDL, problems are specified with an initial state, a set of actions, and a set of goal states. From there, PPDLyte uses causal reasoning to deduce solutions, provided they exist, as a sequence of actions that map the initial state to the goal state(s). The PDDLyte language is limited in comparison to its predecessor, in that it only supports classical planning problems for a single agent: finite, fully-observable, deterministic, static environment descriptions. Another distinguishing characteristic of PDDLyte is in the way it's compiled.

Current PDDL implementations use a LISP interpreter to verify the solutions. For most applications, this is where the life of PDDL ends. The PDDLyte implementation will go further and be compiled from C code to X86 assembly. With this design, the high-level reasoning of PDDLyte solutions are able to interface with systems-level C code.

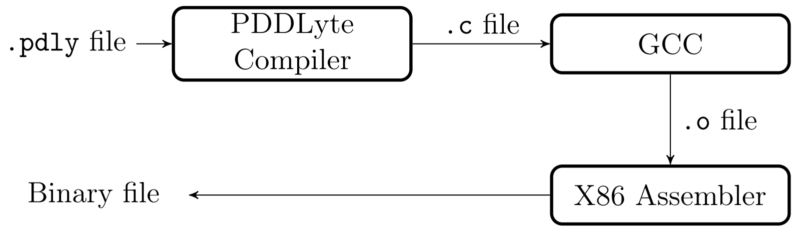
Background

Background for languages similar to pddl.

Motivation

Motivation for creating PDDLyte.

PDDLyte Pipeline



Lexicon

PDDLyte will compute boolean statements from conjunctions of atomic objects. In addition to these datatypes, the PDDLyte language will support integer arithmetic for various quantifications.
PDDLyte is a strongly-typed language. It inherits nearly all its syntax from PDDL and only a subset of the datatypes. This is intended to provide the minimum set, required to solve classical-planning problems.

Primitive types

Boolean — True or false values.
Strings — One or more character elements from the ASCII set comprise a string.
Integers — Any element of the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Structured types

Lists — A set of components separated with spaces and enclosed with parentheses: the component's types can be dissimilar.
Atoms — An atom has a string literal identifier and set of associated properties. Properties are a list of atoms. An atom has no properties when it's instantiated. Properties associate in the initial condition and as result of actions.

Domains

Domains are the global problem descriptor. Only a single domain may be defined per file. A set of action schemas serve as a definition of a planning domain. The initial state and goal define the specific problem. [3] All domain preconditions and action parameters must be prefixed with a question mark to distinguish them from the problem objects.

```
(define (domain <name>)
          (<types_def>)
         (<actions_def>)
     (<predicates_def>))
```

Types — PDDLyte allows for any object to be defined as a list of atoms.

Actions — Actions are the operators that transition the system between states. These are the basic elements which a solution is built[2]. There are pre-conditions and effects associated with each action. The pre-conditions must be satisfied for the transition to take place. Furthermore, the effects must be valid according to an active problem description. If no pre-conditions are specified, then an action is always valid. The action definition must know the type of parameter its preconditions are composed of. Actions are defined with a schema.

```
(:action  <name>
          :parameters     (<param_def>)
          :precondition  (<precond_def>)
          :effect        (<effect_def>))
```

Parameters are a list of atoms used in the action's precondition and effect conjunctions.

```
(:parameters <atom_list>)
```

Preconditions are propositions that must be true for an operator to be applied. This is expressed as a logical conjunction of literals.

```
(:precondition <literal>)
```

Effects describe changes that occur when an action is completed. Predicate states are not affected in anyway by Effects. When an action is complete, the effect is simply set. It is the responsibility of the user to negate the preconditions of an action upon completion. This conjunction of proposition will add an edge to the planning graph. Their negation will remove an edge from the planning graph. This is expressed as a conjunction of literals. Effects will always take place at time $t + 1$

```
(:effect <literal>)
```

Predicates — Predicates consist of a list of declarations that define a property of an object. These can be either true or false.

```
(:predicates <atom_list>)
```

Problems

Problems include an initial state and one or more goal descriptions. All problems are associated with a domain, which provide the context for the solution to be in.

```
(define (problem <name>)
              (<domain>)
          (<objects_def>)
       (<init_state_def>)
   (<goal_descrip_def>))
```

Objects — specify a list of atoms used to define the problem space.

```
(:objects <atom_list>)
```

Initial State — define the predicates that are true in the system's starting configuration. This can be any valid state within the domain. Unless predicates are specified as true in this statement, they are set false. This is a conjunction of ground atoms

```
(:init <atom_list>)
```

Goal Description — The goal description defines the predicate(s) that are true in the system's final configuration(s). This is a conjunction of literals which may contain variables.

```
(:goal <literal>)
```

Operators

Operators will be specified using prefix notation; where the operator is placed to the left of its arguments. This convention is adopted from PDDL, which inherits its syntax from LISP to simplify parsing.

Comments — Comments begin with a semicolon (;) and terminate at the next new line. Furthermore, they do not nest and may not be composed within comments.

```
; commentary ends when the line breaks
```

Arithmetic operators

A minimal level of integer arithmetic will be supported in PDDLyte.

Assignment — Explicit assignments are only permitted with integer values:

```
(set <atom> <int>)
```

Modulus — The modulus of two integers can be computed:

```
(mod <int> <int>)
```

Addition — Integer addition is supported for one or more numbers:

```
(+ <int> ... <int_n>)
```

Subtraction — Integer subtraction is supported for one or more numbers:

```
(- <int> ... <int_n>)
```

Conditional Operators

Equal — Comparing the value of one or more integers is accomplished with the = predicate. The result evaluates to true if every argument is equal to the others. Otherwise, the result is nil.

```
(= <int> ... <int_n>)
```

Less than — The `<` predicate is used to compare if the arguments are in monotonically increasing order; if so, the result is true. Otherwise, the result is nil.

```
(< <int> ... <int\_n>)
```

Greater than — The `>` predicate is used to compare if the arguments are in monotonically decreasing order; if so, the result is true. Otherwise, the result is nil.

```
(> <int> ... <int\_n>)
```

Less than or equal to — The `<=` predicate is used to compare if the arguments are in monotonically non-decreasing order; if so, the result is true. Otherwise, the result is nil.

```
(<= <int> ... <int\_n>)
```

Greater than or equal to — The `>=` predicate is used to compare if the arguments are in monotonically non-increasing order; if so, the result is true. Otherwise, the result is nil.

```
(>= <int> ... <int\_n>)
```

Boolean Operators

Conjunction — Logical conjunctions are formed with the `and` predicate:

```
(and <literal>)
```

Disjunction — Logical disjunctions are formed with the `or` predicate:

```
(or <literal>)
```

Negation — The value of a logical conjunction of literals is inverted with the `not` predicate. The function returns true if its argument is nil, otherwise false.

```
(not <literal>)
```

Keywords

Keyword	Description
`define`	instantiates a domain or problem specification
`domain`	domain specification
`problem`	problem specification
`:types`	specifies a list of objects
`:action`	specifies an action
`:precondition`	specifies an action's preconditions
`:effect`	specifies an action's effects
`:parameters`	specifies an action's parameters
`:predicates`	specifies the domain predicates
`:objects`	specifies a problem's objects
`:init`	specifies a problem's initial state
`:goal`	specifies a problem's goal description

GraphPlan Algorithm

Underlying the PPDLyte language is a graph-based planner that generates solutions to the program's problem. The algorithm which computes the solution is the GraphPlan [?] algorithm will be used to solve problems.

- Phase one — plan graph expansion: creates graph encoding pair-wise consistency and reachability of actions and propositions from initial state. Graph includes, as a subset, all plans that are complete and consistent.
- Phase two — Solution extraction: graph is treated as kind of a constraint satisfaction problem. Selects whether or not to perform each action at each time point by assigning CSP variables and testing consistency.

the solver works under the premise that an action can be executed in a given state, provided the preconditions are satisfied. That is $a \in A_s \leftrightarrow s \models P_s$

Testing

Testing procedures stressed two factors: function and performance. Testin compares an input-output pair to a specification. Unit testing validates every piece of the program independent from the rest of the code. Overall program.

Example program

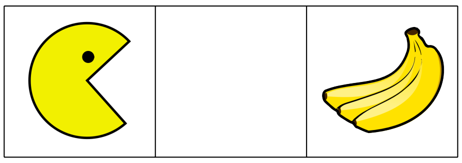
A program in PPDLyte consists of a domain and problem statement. Executing the program solves the problem and generates output.

Monkeys and Bananas

A monkey is in a room tormented by a bundle of bananas hanging from the ceiling, just out of its reach. Inside the room, there is also a chair and a stick. Both items are proportioned just right to allow the monkey enough reach to retrieve the bananas – provided the monkey has the sense to use the items properly. Once the chair is positioned under the bananas, the monkey can step up and use the stick to wackk the bundle free.

Pacman

This program is simple to conceptualize, and therefore should be a good first step in generating plans. Pacman starts in one side of the world, with the goal of eating the bananas in another square. The solution is to simply move right twice.



Example PDDLyte code specifies the problem.

Bibliography

[1] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl — the planning domain definition language. Technical report, AIPS Planning Competition Committee.

[2] Nils J. Nilsson and Richard E. Fikes. Strips: A new approach to the application of theorm proving to problem solving. Technical report, Artificial Intelligence Group, Stanford Research Institute.

[3] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall.