PDDLyte
A Partial Implementation of The Planning Domain Definition Language

John Martin Jr.
jdm2213@columbia.edu

February 10, 2014

Abstract

The PDDLyte language, whose name derives from the Planning Domain Definition Language (PDDL)[1], is a symbolic, specification language used to formulate and solve planning problems. Similarly to PDDL, problems are specified with an initial state, a goal description, and a domain on which to plan over. 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 will only support 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.

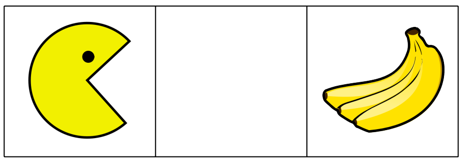
Current PDDL implementations use CLISP-based interpreters to verify the solutions. For most applications, this is where the life of PDDL ends. The PDDLyte implementation will go further and be compiled to C code, then to X86 assembly. With this design, the high-level reasoning of PDDLyte solutions will be amenable to systems-level C code interfaces.

Background: Planning

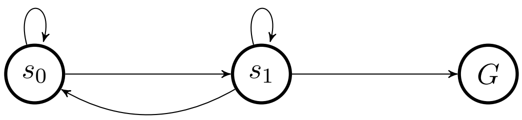
Automated planning is a branch of artificial intelligence that helps characterize intelligent behavior. Plans are explicitly deliberated in a process that chooses prearranged actions to achieve an objective. A planning problem asks if a goal description can be achieved from its initial state [4]. In classical planning, actions are assumed to be finite sets of operators that transition a system from one state to another; hence a solution to this problem is a sequence, $\{A\}$, of actions. Every plausible state of a system can be represented as vertices on a graph, and transitions that connect these states represent edges. In this framework, finding a solution is reduced to searching the state-space graph for a path that connects the initial state to the final state. Let's take a closer look at how this is done with the PDDLyte language.

Example program: Pacman

Pacman, with his ever-present, unperceptive objective to feed himself, can certainly benefit from automated planning. In this simple example, Pacman's goal, G, is to eat the bananas located in the third square; he originates in the first square, s_0. Pacman may move in any direction, provided it is between two adjacent squares, and he occupies the starting square.



With this small amount of information, the planning problem can be formulated with a triple: $P = (s_0, G, A)$. Where the set of accessible actions, A, are quickly realized as the only two available moves: move forward, or remain still. This makes planning graph simple to visualize.



When problems and their corresponding domains are specified formally in PDDLyte, graphs like these will be generated and transversed for solutions. If a solution is available – which is guaranteed to be discerned from the completeness of the search algorithm – then it will be returned as a sequence of actions, as shown in the example code.

```
(define (domain pman)
  (:predicates
   (adj ?square-1 ?square-2)
   (at ?what ?square)
  )

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (adj ?from ?to)
                       (at ?who ?from))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
  )
)

(define (problem pman_prob)
```

```
  (:domain pman)
  (:objects
   sq_11 sq_12 sq_13
   pacman
   banana
  )

  (:init
   (adj sq_11 sq_12) (adj sq_12 sq_11)
   (adj sq_12 sq_13) (adj sq_13 sq_12)

   (at banana sq_13)
   (at pacman sq_11)
  )

  (:goal (and (at pacman sq_13)
              (at banana sq_13))
  )
)

; the optimal plan consists of two moves
;---------------------------------------
;  (move pacman sq_11 sq_12)
;  (move pacman sq_12 sq_13)
```

Motivation

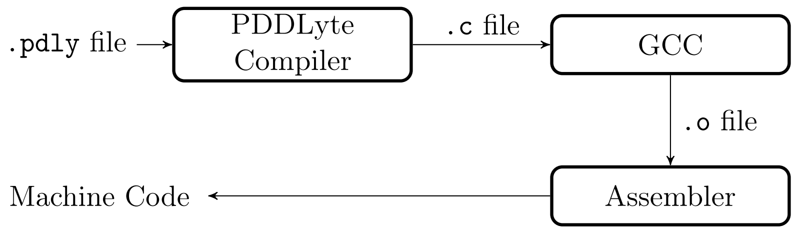
Beyond Pacman, there exists a myriad of planning problems beckoning for description. What set of actions should a person take when driving to work? What operations are required to route secure network traffic around China? Using a planning-specification language organizes the high-level reason needed to solve these problems.

PDDLyte is designed as a symbolic language to accomodate the generality of problems it is intended to describe. The developer should have the means to create and give meaning to any entity.

Overall I hope the language can serve a purpose in the planning community. Perhaps those in academia will find comfort in a familiar syntax and run PDDLyte for instructional purposes. Those in the commercial realm may initially balk at the language's extensibility, but eventually discover uses for it in simulation or operations.

PDDLyte Pipeline

A planning domain and problem will be described in a .pdly file. The compiler will first translate into C code – opening up the opportunity for interfacing with other code. From there, the code will be compiled into a machine language executable.



Lexicon

The syntax in PDDLyte derives from PDDL. The components of each language focus around the classical-planning representation of STRIPS, which itself is a restricted, state-transition system $\Sigma = (S, A, \gamma)$ over a function-free, first-order language $\mathcal{L}$. For more information on the formalism of PDDL, see [2].

Primitive types

Symbols — Extensible data objects with property lists that denote objects.
Objects — Generic datatypes identified with symbols, consisting of one or more character elements from the ASCII set. Fluent objects must be prefixed with a question mark: ?<var>.
Atoms — An atom is a predicate with a specified number of object arguments. An atom is said to be grounded when they relate to specific objects with values – not variables.

Structured types

Lists — A set of components separated with spaces and enclosed with parentheses: the component's types can be dissimilar.

Domains

Domains can be thought of as universe. What describes these universes are states, actions and means to use actions at states to make transitions. Only a single domain may be defined per file.

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

Types — Types are symbols that specify objects of the domain. This attribute is an extension of PDDL, but will be inherently supported with PDDLyte.
Actions — Actions are the operators that transition the system between states. These are represented as triples, $a = (name, precondition, effect)$, and are the basic elements of a solution [3]. In PDDL, an action's name is considered both the unique symbol and a set of parameters that define the operation. 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.

```
(: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 (?<name> - <type> ... ?<name_n> - <type_n>)
```

Preconditions are propositions that must be true for an operator to be applied. This is expressed as a logical conjunction of literals. *Literals* are positive or negative atoms.

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

Effects describe changes that occur when an action is completed at the successor state. This is expressed as a logical conjunction of literals. Furthermore, developers should be mindful to balance the preconditions with the effects; as predicate states are not automatically negated by transitions.

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

Predicates — Predicates define relationships between object variables. These can be static relations that hold from state to state, or fluent relations. Each predicate is defined with a symbolic name and one or more object name-type groups; where the object name is separated from the type with a dash:

```
(:predicates (pred ?<name> - <type> t...) ...)
```

Problems

Problems are triples defined as $P = (\Sigma, s_0, G)$. This inlcudes a domain Σ an initial state s_0, and a set of ground literals describing the goal condition G.

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

Objects — Objects refer to those used in the problem configuration. This attribute is an extension of PDDL, but it's such a common requirement for classical plans that PDDLyte will inherently support it. Each object is declared with a symbolic name and one or more object name-type groups; where the object name is separated from the type with a dash:

```
(:objects ?<name> - <type> ... ?<name_n> - <type_n>))
```

Initial State — The initial state defines the predicates that are true in the system's starting configuration. This can be any valid state within the domain and is written as a conjunction of ground literals.

```
(:init (<literal>) ...)
```

Goal Description — The goal description defines the predicates that are true in the system's final configuration. This is a conjunction of grounded literals.

```
(: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
```

Atomic Literal 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

Computationally, this accomplished Planning occurs with respect to a domain. There can be domain-specific plans focused on topics such as perception, motion, or communication, to name a few. There may also be domain-independent plans, which harness the commonality in domain-specific plans to plan for generalities.

In STRIPS-based planning, there is no explicit reference to time. Plans in this domain are assumed to have solutions which themselves are sequences of actions. This implies there is no parallelism in the solution. Planning graphs are typically exponential in size. The backbone of this language uses a polynomial approximation to a full planning graph. What we build is a structure where ever action level constatins all the actions that are applicable in the ith state level along with the constraints saying that two actions cannont both be executed at the same level. every state level contains all the literals that could result from any possible choice of actions in the previous action level along with the constraints saying which paits of literals are bot possible. it is important to note that the process of constructing the planning graph does not require choosing among action, which would entail combinatorial search. instead, it just records the impossibility of certain choices using mutex links.

If any goal literal fails to appear in the final level of the graph, then the problem is unsolvable.

- 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 $a \in A_s \leftrightarrow s \models P_s$

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] Malik Ghallab, Dana Nau, and Palo Traverso. *Automated Planning, Theory and Practice*. Morgan Kaufmann.

[3] 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.

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