Knowledge Engineering Tools for Planning in PDDL - Syntax Highlighting, Task Generation and Plan Visualization and Execution - an extensible framework

Volker Strobel

February 24, 2014

Contents

1 Abstract

1.1 TODO Insert somewhere

A introduction to the syntax of PDDL will be given, as well as Potential applications in real life environments could be ... And a way for writing proper.

1.2 Abstract

1.2.1 v_1

This thesis presents the Planvning Domain Definition Language (PDDL) and debates possibilities and limitations of its use with reference to household robots. Furthermore, a syntax highlighter for the text editor Sublime Text

and an online code editor is presesented and ways of visualizing the found plan are discussed. (more concrete!)

The purpose of this thesis is to explore the possibilities and limitations of modeling real life household domains by the Planning Domain Definition Language (PDDL). Thereto all steps for developing a new planning problem with will be given, from necessary software, over syntax to convenient PDDL constructs. That means that a widespread overview will be given, whereas the topic's main focus are household domains ("household"). A introduction to the syntax of PDDL will be given, as well as Potential applications in real life environments could be And a way for writing proper

1.2.2 v_2

1.3 Zusammenfassung

Diese Arbeit untersucht die Möglichkeiten und Grenzen der Planungssprache Planning Domain Definition Language (PDDL) im Hinblick auf die Anwendung bei Haushaltsrobotern. Dazu wird ein umfassender Einblick in die Syntax dieser Sprache gegeben und Anwendungsbereiche werden anhand praktischer Beispiele untersucht. Für die Konstruktion der Pläne wird eine Entwicklungsumgebung präsentiert, die Python, Spracherfassung und PDDL vereint. Weiterhin werden ein Plugin für den Texteditor Sublime Text (Version 2 und 3) und ein PDDL-Onlineeditor vorgestellt. Um P\"ane besser zu fassen, werden Möglichkeiten der Visualisierung mit der Robotersimulation MORSE und dem Planvisuialisierungsprogramm VISPlan angesprochen.

2 Introduction

2.1 TODO Notes

- Scope of the article (What did I miss out, what is included?)
  - Learning Algorithms
- Name Tools and provide found possibilities and limitations
- Motivation: Why is this article interesting?
- Structure of the article
- Review of relevant literature? Note: I can review the relevant literature and related work in the related chapter, if this is convenient

- Planning Architecture - 3 Tier Architecture
- ICAPS
- Alex: Do not write: I'll do this, because it does not exist yet
- Do write: I'll do this because this is interesting, because it can do xyz, etc.

2.2 Intro

The Planning Domain Definition Language (PDDL), firstly invented by Drew McDermott et al. (1998) has emerged to the standard language for modeling planning tasks (source!). Planning is about what to do and in which order for a given problem. While the scope of applications is enormous, there are several limitations and drawbacks that have to be considered. After giving a general introduction to PDDL, this thesis explores the scope of PDDL with respect to household domains. Possible applications exist in the construction of household robots, that are especially relevant for people with disabilities. A package that includes amongst other a syntax highlighter and code snippets for the text editor Sublime Text is presented and the functionality is transferred to a on-line editor with instant access.

The main focus of this thesis is about real world applications and the development of handy tools that support (and partially automatize) the planning process. On the basis of a 'kitchen domain' several possibilities and limitations are examined, while for each drawback an alternative route is provided.

Planning is one the the classic Artificial Intelligence (AI) tasks.

2.3 Intro $\mathbf{v_2}$

The Planning Domain Definition Language (PDDL), firstly invented by Drew McDermott et al. (1998) has emerged to the standard language for modeling planning tasks. Despite the process in AI planning, knowledge engineering is still a

3 PDDL and Planning Basics

3.1 TODO Notes

- Brief summary at start

- Start with a paragraph that describes the context
- Very interesing for basics of PDDL:
- `http://www.ida.liu.se/~TDDC17/info/labs/planning/writing.html`
- Konstruktionsanleitung
- Propositionale Logic -> Articifial Intelligence a Modern Approach
- To insert somewhere:
  - It should be mentioned, that almost no planner supports every part of PDDL. And, additionally, the quality of error messages is very diversified. While some simple state: error occured, other list the problem and the line.

3.2 Basics

Planning in terms of PDDL

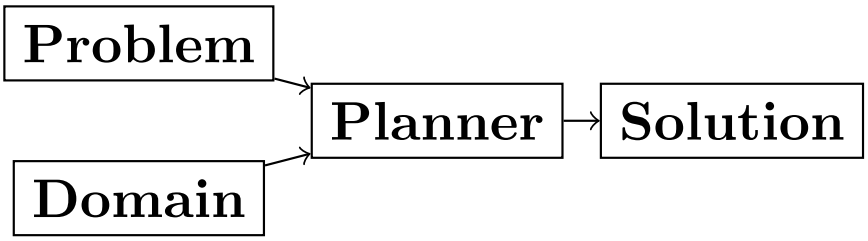


Figure 1: PDDL Planning workflow

Two files are needed for planning with PDDL: a domain file and a problem file. A planner and use the generated solution file (*plan*).

PDDL was first described in PDDL-the planning domain definition language (1998) and has been in constant development since then. This thesis makes use of *PDDL 3.1* (n.d.) if not otherwise stated.

Every planning task in PDDL is composed of two separate files:

- Domain file: description of general types, predicates, functions and actions
- Problem file: description of a concrete problem environment.

This separation allows for an intuitive process of task modeling: While general instances are described in the domain file, a specific instance of a problem is created in the problem file.

These two files shell be investigated further in the following sections.

3.3 Format of the Domain File

The description of the task files is deliberately performed without the use of the BNF notation. Fox et al. describe the BNF notation for PDDL 2.1 in Fox and Long 2003 as well as in Kovacs' (unpublished) paper Kovacs 2011.

Domain files have a strict format: All keyword arguments must appear in the order specified in the manual (an argument may be omitted) and just one PDDL definition (of a domain, problem, etc.) may appear per file. Fox and Long 2003, p. 6.

3.3.1 TODO Include simple domain -> LaTeX

3.3.2 TODO Include simple problem -> LaTeX

3.3.3 TODO Include simple plan -> not yet in LaTeX

3.3.4 Define

Every domain file start with (define (domain <domainName>) ...) where, <domainName> can be any string

3.3.5 Requirements

The requirements part is not a mandatory part of a PDDL domain file. However, as most planners only support a subset of PDDL they are useful for determining if a planner is able to act on a given problem. They are declared by the (:requirements ...) part. Some often used requirements include ...

3.3.6 Types

In order to assign to assign categories of objects, PDDL allows for type definition. Like that, parameters in actions can be typed, as well as arguments in predicates, functions [extra source!]. Later, in the problem file, objects will be assigned to types, like objects to classes in Object Orientated Programming (OOP). Adding to the (:requirement ...) part of the file guarantees, that typing can be correctly used. Strips (no types) vs ADL (types).

3.3.7 Functions

Functions are not supported by many planners (source!) and, before PDDL 3.1 they could only be modeled as

Complete that!
nil

It is notable that before PDDL 3.0 the keyword functors was used instead

3.3.8 Actions

PDDL 3.1 supports two types of actions: durative-action and the 'regular' action.

3.4 Format of the Problem File

3.5 Format of the Solution File (Plan)

3.6 Planning Process

4 Possibilities and Limitations

4.1 Kitchen Domain

In this section, a kitchen domain will be presented, whereby PDDL structures are presented that will be also useful in other domains. I will start with a rather simple domain, present possible limitations and then extend the file by more sophisticated constructs.

4.1.1 Functions

As functions have a return value, the modeling possibilities dramatically increase.

4.1.2 Numerical Expressiveness

One might assume that the distance could be modeled as follows:

```
(durative action ...
...
  :duration (= ?duration (sqrt (coord-x )))
...
```

However, PDDL does only support basic arithmetic operations (+, -, /, *).

An Euclidean distance function that uses the square root would be convenient for distance modeling and measurement. However, PDDL 3.1 supports only four arithmetic operators (+, -, /, *). These operators can be used in preconditions, effects (normal/continuous/conditional) and durations. Parkinson and Longstaff (2012) describe a workaround for this drawback. By declaring an action 'calculate-sqrt', they bypass the lack of this function and rather write their own action that makes use of the Babylonian root method.

1. Alternative #1: Only sqrt exists Assuming that a function sqrt would actually exist, the duration could be modeled as follows:

```
:duration (= ?duration
             (sqrt
              (+
               (*
                (- (pos-x (current-pos))
                   (pos-x ?goal))
                (- (pos-x (current-pos))
                   (pos-x ?goal)))
               (*
                (- (pos-y (current-pos))
                   (pos-y ?goal))
                (- (pos-y (current-pos))
                   (pos-y ?goal))))))
```

2. Alternative #2: sqrt and expt exist Assuming that a function sqrt would actually exist, the duration could be modeled as follows:

```
:duration (= ?duration
             (sqrt
              (+
               (expt
                (-
                 (pos-x (current-pos))
                 (pos-x ?goal)))
               (expt
                (-
```

```
        (pos-y (current-pos))
        (pos-y ?goal))))))
```

3. Alternative #3: Calculate distance and hard code it, e.g. (distance table kitchen) = 5.9

   - Distance Matrix
   - `http://stackoverflow.com/questions/20654918/python-how-to-speed-up-calculatio`
   - Scipy.spatial.distance (-> Clojure?)
   - Mention that the Taxicab geometry allows different ways that have an equal length

   Another alternative is to make use of an external helper and, instead of calculating every entry of the distance matrix. the distance only if needed, incorporate every possible combination of two locations. This approach has certainly a major drawback: With an increasing amount of locations, the number of combinations increases exponentially. That means, if there are 100 locations, there will be

   TODO: Calculate possibilities

   nil

   . . . . The native approach would be to iterate over the cities twice and calculate only the half of the matrix (as it is symmetric, that mean distance from A to B is the same as the distance from B to A).

4. Alternative #4: Use the manhattan distance

   Allowing the agent to move only vertically and horizontally would be that one can use the so called Taxicab geometry (or Manhattan length) as distance measurement. In the Kitchen domain, this could be modeled as follows:

```
% => Metric: reduce duration

% dKitchenware.pddl
\begin{figure}[t]
\inputminted[mathescape, linenos, numbersep=5pt, frame=lines, framesep=2mm]
            {csharp}
```

```
            {Code/dKitchenware.pddl}
\caption{The basic kitchenware domain}
\end{figure}
\section
```

5. **TODO** Human Planner Interaction
6. **TODO** World State - Plan Stepper
   - EMail + SMS connection
   - Very interesting: `http://www.tzi.de/~edelkamp/modplan/`
7. **TODO** Plan verification % Reuse existing plans (VAL!). If a plan is not reusable, find problems (e.g. step 13 and 17) and try to find solutions for that
8. **TODO** Summary List all limits and possibilities and write 5 sentences for each statement (my idea -> ask Alexandra).

5 Software Engineering Tools for AI Planning

5.1 TODO Ideas

- PDDL type hierarchy and object instantiation to UML / TikZ, store predicates (and action?) in same box as type
- Research Knowledge Engineering in Planning
- Human Computer Interaction
  - `http://hci.waznelle.com/checklist.php`
- Write Tiago (itSimple) regarding PDDL -> UML (and knowledge engineering in general
- ICKEPS (International Competition on Knowledge Engineering for Planning and Scheduling)
- Orient on "How to Design Classes"

5.2 Statement of Problem

Writing and maintaining PDDL files can be time-consuming and cumbersome. However, handy development tools can support and facilitate the task design process and reduce potential errors.

Below, methods are presented for

Section 1 The automation of the PDDL task design process. File input and output and dynamic generation (design level)

Section 2 An interactive PDDL environment: speech synthesis and recognition.

Section 3 Mathematical limitations (design level)

5.3 Implementation

5.3.1 Integrated Design

The code is written in Clojure, a LISP dialect. As PDDL has a 'LISP-like syntax', using a LISP dialect for the interface is convenient. This thesis uses Clojure[TODO: src], a relatively modern LISP dialect that runs on the Java Virtual Machine.

In order to start the document, a namespace has to be defined. The required packages are:

clojure.tools.reader.edn Safe file input. I will use this method for entirely replacing the tools in clojure.core/read

clojure.java.io Methods for file input and output (IO)

```
(ns org-ba.core
  (:gen-class)
  (require [clojure.tools.reader.edn :as edn]
           [clojure.java.io :as io]))
```

And we use a main method that allows for testing and running the script.

```
(defn -main
  "Runs the input/output scripts"
  [& args]
  (println "Running..."))

#'user/a
```

As PDDL files and 'information' will be in stored externally, a reader method is needed. Edn reader provides functionality and guarantees that no harmful commands can be read in through the reader interface.

```
(defn read-lispstyle-edn
  "Read one s-expression from a file"
  [filename]
  (with-open [rdr (java.io.PushbackReader. (clojure.java.io/reader filename))]
    (edn/read rdr)))
```

Next, a macro is provided for writing to files. It rebinds **out** to a writer (that open a file for writing). Therefore, print statements (print, prn, etc.) that normally would be send to the standard out, are redirected to the file.

```
(defmacro write->file
  "Writes body to the given file name"
  [filename & body]
  `(with-open [w# (writer ~filename)]
     (binding [*out* w#]
       ~@body))
  (println "Written to file: " ~filename))
```

Desired objects that belong to a type for a domain are sometimes provided in a plain list, like the following:

```
vw-passat
opel-corsa
chevrolet-volt
```

It would be convenient to add a type to these objects, for two reasons:

- Add a super-type to the subtypes in the list
- Add a type to a list of objects for the problem file

The following method affords that:

```
(defn read-objs
  "Read PDDL objects from a file and add type
  (e.g. 'table bed' -> (list table - furniture
                         bed - furniture))"
  [file object-type]
  (as-> (slurp file) objs
        (clojure.string/split objs #"\s")
        (map #(str % " - " object-type) objs)))
```

By the help of these methods, you can create PDDL templates, for example for a domain file:

```
(defn create-pddl
  "Creates a PDDL file from a list of objects and locations"
  [objs-file objs-type]
  (str
"(define (domain domainName)

  (:requirements
     :durative-actions
     :equality
     :negative-preconditions
     :numeric-fluents
     :object-fluents
     :typing)

  (:types\n"
   (clojure.pprint/cl-format nil "~{~&~5@T~a~}" (read-objs objs-file objs-type))
        ")

  (:constants

  )

  (:predicates

  )

  (:functions

  )

  (:durative-action actionName
     :parameters (?x - <objectType>)
     :duration (= ?duration #duration)
     :condition (at start <effects>)
     :effect (at end <effects>))
)"
))
```

PDDL widely supports 'types'. These define possible shapes for objects (similar to 'classes' in object oriented programming (OOP)). Types are defined in the ':types' section of the PDDL domain file:

```
....
(:types man woman - human
        human - agent
        robot - agent)
...
```

A meaning-full type hierarchy is the basis for clean, well-written domains. Type definitions constitute the first part in the PDDL domain design process, as they determine, on which possible objects actions can be performed.

In order to further work with the specified types, they have to be extracted from the PDDL file. For this task, a regular expression is used, that splits the types in subtypes and belonging types.

```
(defn split-up
  "Split a PDDL type list (:types obj1.1 obj1.2 - objT1 obj2 - objT2 ...)
  into strings of subtypes and associated types,
  [[subytype1 subtype 2 ... - type][subtype1 subtype2 ...][type]"
  [coll]
  (let [coll (if (= :types (first coll))
                (rest coll)
                coll)]
    ;; REVIEW: insert (\w) for trimming?
  (re-seq #"((?:\s*\w+\s*)+)-\s*(\w+)\s*"
        (clojure.string/join " " coll))))
```

The resulting list can be used for creating a hash-map, where every type from the PDDL type declaration is the hash-key and the subtypes are the values.

```
(defn types->hash-map
  "Convert splitted type list (['<expr>' '<subtype1.1> <subtype1.2> ...' '<type1>']
  to a hash-map {'<type1>': ['<subtype1.1>' '<subtype1.2>' ...], '<type2>': ...}"
  [coll]
  (reduce (fn [h-map [_ objs obj-type]]
           (let [key-obj-type (keyword obj-type)
                 existing-vals (key-obj-type h-map)]
          (assoc h-map
```

```
                key-obj-type
                (concat existing-vals
                        (clojure.string/split objs #"\s")))))
          {}
          coll))
```

Now, as these information is present in a 'native' Clojure data structure, it can be used for various purposes. A desirable purpose would be to display the type hierarchy in kind of a 'class' diagram:

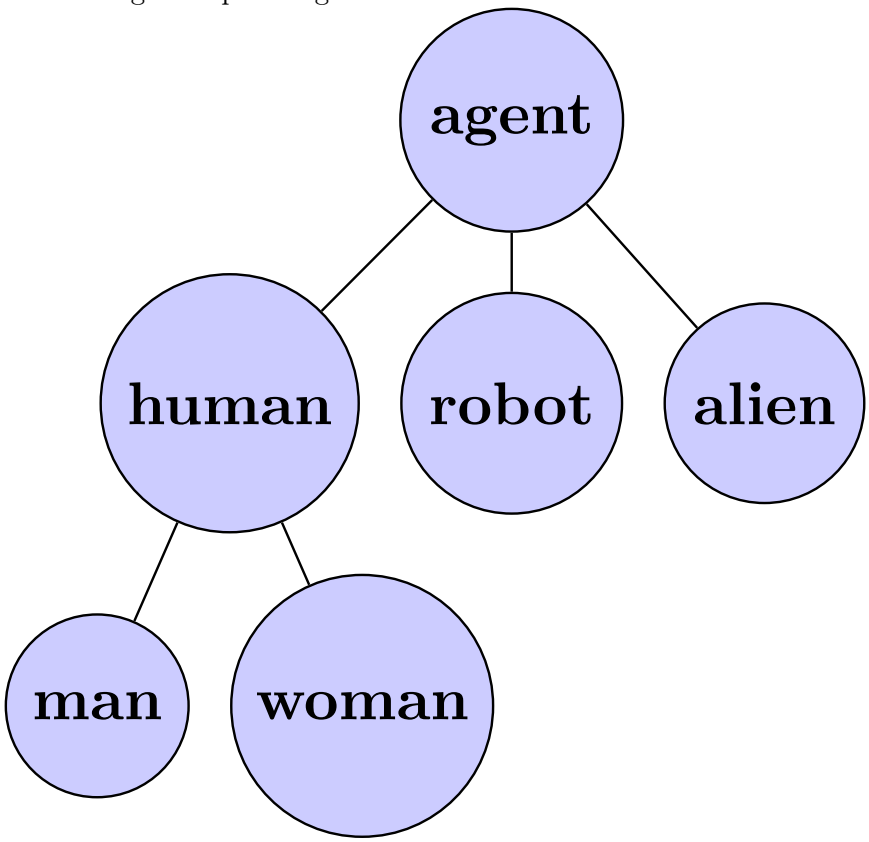
```
(defn map-entry->TikZ-seq
  "Converts a hashmap entry (:key [val1 val2 ...])
to a TikZ string (key -- { val1, val2 })"
  [entry]
(str
 (name (key entry))
        " -- "
        "{" (clojure.string/join ", " (val entry)) "}"))

#'user/map-entry->TikZ-seq
```

This method can now be used in order to create a TikZ standalone LaTeX file, that is converted to a png file by the use of lualatex.

```
(defn hash-map->TikZ-out
  "Converts complete PDDL type hash-map to TikZ file"
  [h-map]
  (str
"\\documentclass[tikz]{standalone}

\\usepackage[utf8]{inputenc}

\\usepackage{tikz}

\\usetikzlibrary{graphdrawing}
\\usetikzlibrary{graphs}
\\usegdlibrary{layered,trees}

\\begin{document}

\\begin{tikzpicture}
```

```
\\graph[layered layout, nodes={draw,circle,fill=blue!20,font=\\bfseries}]
{
  " (clojure.string/join ",\n  " (map map-entry->TikZ-seq h-map))
"
};

\\end{tikzpicture}
\\end{document}"))
```

A resulting example image would look like this:



This look can be further extended in order to create a UML class diagram for PDDL domains and problems:

5.3.2 Syntax Highlighting and Code Snippets

1. Problem Statement

Writing extensive domain and problem files is a cumbersome task: longer files can get quickly confusing. Therefore, it is convenient to have a tool that supports editing these files. Syntax highlighting describes the feature of text editors of displaying code in different colors and fonts according to the category of terms (source: Wiki). A syntax highlighting plug-in for the text and source code editors Skinner (2013a) and Skinner (2013b) is proposed and transferred to the online text editor Ace are used to implement this feature, as ST Syntax Highlighting files can easily be converted to Ace Files.

For Mac user, TextMate (TM) is very similar to ST and the syntax highlighting file can be used there, too. Besides, the general principles (e.g. regular expressions) outlined here, apply to most of other editors as well.

2. Implementation ST syntax definitions are written in property lists in the XML format.

   The syntax definition is implemented by the use of the ST plug-in *AAAPackageDev* (2014). So, the definitions can be written in YAML in converted to Plist XML later on. AAAPackageDEV provides the following features:

   > AAAPackageDev is a Sublime Text 2 and 3 plug-in that helps to create and edit syntax definitions, snippets, completions files, build systems and other Sublime Text extensions.

   By means of Oniguruma regular expressions (Kosako 2007), scopes are defined, that determine the meaning of the PDDL code block. The scope naming conventions mentioned in the *TextMate 2 Manual* are applied here. By the means of the name, the colors are assigned. Different ST themes display different colors (not all themes support all naming conventions).

   The syntax highlighting is intended for PDDL 3.1, but is downward compatible, as previous versions are subsets of later versions.

   TODO Are later versions really subsets?

   nil

   Like that, the PDDL file is parsed

TODO Is it really parsed, or are just parts highlighted?

nil

into different parts.

3. Usage and Customization By using ST as editor, language independent ST features are supported, like auto completion, code folding and column selection, described in the Sublime Text 2 Documentation.

   To enable syntax highlighting and code snippets, the files of the repository have to be placed in the ST packages folder. The first part of the PDDL.YAML-tmlanguage describes the parts of the PDDL task that should be highlighted. By removing (or commenting) include statements, the syntax highlighter is adjustable the user's need.

   By default, all scopes are included.

4. Related Work

   (a) PDDL Studio PDDL Studio is an Integrated Development Environment (IDE) for creating PDDL tasks.

   (b) PDDL Mode Announced 2005 in a mailing list entry, PDDL mode supports PDDL 2.2.

   (c) itSIMPLE

5.4 Evaluation

6 Discussion

7 Conclusion

8 Bibliography

References

[1] *AAAPackageDev*. 2014. URL: `https://github.com/SublimeText/AAAPackageDev/` (visited on 02/24/2014).

[2] Maria Fox and Derek Long. "PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains." In: *J. Artif. Intell. Res.(JAIR)* 20 (2003), pp. 61–124.

[3] K. Kosako. *Oniguruma Regular Expressions Version 5.9.1*. 2007. URL: http://www.geocities.jp/kosako3/oniguruma/doc/RE.txt (visited on 02/24/2014).

[4] DL Kovacs. "BNF definition of PDDL 3.1". In: *Unpublished manuscript from the IPC-2011 website* (2011).

[5] MacroMates Ltd. *TextMate 2 Manual*. 2013. URL: http://manual.macromates.com/en/ (visited on 02/24/2014).

[6] Simon Parkinson and Andrew P Longstaff. "Increasing the Numeric Expressiveness of the Planning Domain Definition Language". In: *Proceedings of The 30th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG2012)*. UK Planning and Scheduling Special Interest Group. 2012.

[7] *PDDL 3.1*. URL: http://ipc.informatik.uni-freiburg.de/PddlExtension.

[8] Jon Skinner. *Sublime Text 2*. http://www.sublimetext.com/2. Release 2.0.2. 2013.

[9] Jon Skinner. *Sublime Text 3*. http://www.sublimetext.com/3. Beta Build 3059. 2013.

9 Appendix