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

Volker Strobel

March 13, 2014

Contents

Abstract

Automated planning and scheduling is a key component of artificial intelligent (AI) behavior. A planning language that is widely used for AI task specifications is the Planning Domain Definition Language (PDDL). The aim of this project was to develop tools that simplify the extensive knowledge engineering effort required by PDDL and thus facilitate the construction of planning domains. For this purpose, an extensible interface between the programming language Clojure and PDDL was developed, that supports knowledge engineers in developing PDDL projects. A tool based on this interface visualizes the type hierarchy in PDDL domains, allowing knowledge engineers to keep track of the domain structure and others that may have to work with the domains to understand them at a glance. As further implementation of this interface, a tool that calculates the distances of objects within a problem file to each other was created to show the extent of this tool and to present a way of bypassing PDDL's limited modeling capacity. Lastly, a plug-in for the code editor Sublime Text was implemented to aid developers with efficiently creating new PDDL file, navigating within the code and with finding and fixing mistakes. In order to test the quality of the syntax highlighter and the type diagram generator, a user study was conducted with inexperienced PDDL users that were asked to develop domains with and without the tools and to subsequently evaluate the tools in regard to their usefulness and usability. Both the time on task, number of errors and a post questionnaire were analyzed and it was found that… All tools developed within the scope of this thesis are unique in some regard and they can all be altered very easily to work with other editors and planning languages. An auxiliary tool to transform PDDL code into Clojure code that was needed for the latter two tools, could be relevant for future works in the field, as there are many additional use cases such as…

`https://www.ece.cmu.edu/~koopman/essays/abstract.html`

Chapter 1

Introduction

Assume you are a monkey in a cage. There are some bananas dangling from the ceiling and a bunch of boxes lying around on the ground. You are hungry and those bananas look delicious, but you just cannot reach them. To solve the problem at hand it is important that you figure out that the boxes can be stacked on top of each other and that you can then climb on top of the boxes and reach the bananas, i.e. you need to come up with a sequence of actions that take you from your initial state to your goal state. This famous experiment is described by Wolfgang Köhler in his book "The mentality of apes." and became known (in a similar form) in Artificial Intelligence (AI) research as the *monkey and banana problem*. Importantly, with this experiment, Köhler demonstrated that the monkeys did not acquire the solution by trial and error, but rather that they actually understood the environment enough to devise a plan only by contemplating the problem in the context of their world. As can be seen from the scenario given above, planning is a crucial component of problem-solving. However, while monkeys and humans are able to create and continuously update their mental models of the world thanks to sensors, AI implementations are yet to fully master this skill (SOURCE! WHAT IS STILL MISSING?). Thus, the most common and extensive approach to planning in AI to this day is by means of knowledge engineering (KE). In KE, a human expert that is familiar with the underlying syntax integrates world information into a computer system Feigenbaum and McCorduck (1983). In automated planning, this is usually done using a planning language applied in an editor. A standard Both the world and the problem are modeled with the planning language and are then fed to the planning software as inputs. The software produces the solution to the problem in the form of a plan, that means a sequence of action, leading from the initial state to the goal state as output. Naturally, the process of creating these modeled worlds, from here on after called domains, and problems is error-prone and time consuming. While it cannot be denied that the planning systems are improving steadily as computer processing times decrease and algorithms are altered to work even better, the human factor in the knowledge engineering approach cannot be ignored (SOURCE). Performances of planners are largely dependent on the input they receive and the manner in which it is written (SOURCE). Therefore, focusing on the usability of planning languages and hence facilitating the knowledge engineering process is worthwhile. Although recent PDDL extensions increased the expressiveness of PDDL and thus allowed for real-world applications (SOURCE!!!), they also demand a higher level of knowledge and attention on the part of the knowledge engineer. Particularly during the first International Competition on Knowledge Engineering for Planning and Scheduling in 2005, advances were made in shifting the modelling process from a text-based to a graphical programming environment. Even though such tools seem more user-friendly at first, they also demonstrated considerable drawbacks such as limited functionality, expenditure of time and editing difficulty (SOURCE). Obviously, the usefulness of such tools depends on the demands of the knowledge engineer. Yet, the question arises, whether it is not better to develop tools that facilitate text-based programming to such a degree that it is as easy to use as graphical interfaces while still allowing the user full functionality and the necessary insight into the code to edit it (Classification of Concrete Textual Syntax Mapping Approaches - Nice Paper). As can be seen from the two problems described above, tools that ensure greater usability of planning language editors and thus help in producing standardized, high-quality domains and problems that not only planners but also other knowledge engineers can easily work with are greatly needed. The main focus of this thesis is on the development of such handy tools that support (and partially automate) the planning process. At first, already existing planning tool are reviewed in order to put this thesis in context. The body of this thesis consists of three parts. The first part introduces the basics of planning and the PDDL syntax. This is followed by the second part, which presents a extensible interface between the programming language Clojure Hickey (2008) and PDDL. Based on this interface, a plug-in for the text and code editor Sublime Text (ST) was implemented that consists of a type diagram generator and a distance calculator. Furthermore, the development, application and customization of a sophisticated syntax highlighter for ST will be presented. The third part is devoted to the evaluation of the type diagram generator and the syntax highlighter in terms of their usefulness and usability. As means to this end, a small user study was conducted with subjects that had no prior PDDL experience. The results and their implications will be discussed before an outlook for future research and developments in the field concludes this thesis (perhaps mention results and outlook here).

1.1 Finding of the research topic

During the research for this thesis, it turned out, that the tools for writing and expanding extensive PDDL descriptions in a reasonable time are limited, while tools for checking plans (Howey, Long, and Fox (2004) + second topic, Glinsk and Barták (2011)) and applying PDDL descriptions (broad range of planner)s, are far more matured. While the original research interest was concentrated on possibilities and limitations of artificial intelligence planning using PDDL, a focus shift was performed, recognizing, that the main PDDL limitation is still the *basic* modeling process, meaning that efficient modeling of useful domains and problems *by hand* is hardly possible by the existing tools (that's too hard!). Anymore, PDDL's general representation ability is already limited through the missing support of mathematical operations besides basic arithmetics. On this account, a possibility for *extending* PDDL was searched and found in Clojure, using the relatedness of both languages embellished by PDDL's LISP-derived notation. In the course of the development of this PDDL/Clojure interface between great potential was seen for facilitating the PDDL design process and thereby push the acceptance and usage of PDDL in real world models. The customizability and extensibility of the ST editor as well as the broad variety of build-in editing features, constituted a convenient basis for the design of a development environment for PDDL. A large variety of language-independent plug-ins exist and is constantly developed, like package managers, git connection . This project focuses the A key concept for the development was the ease of application, so that new users should be able to effectively use the majority of functions intuitively within a short time.

Chapter 2

Related Work

2.1 PDDL Studio

PDDL Studio (Plch et al. 2012) is an Integrated Development Environment (IDE) for creating PDDL tasks (domains and problems) with supporting editing features that are based on a PDDL parser, like syntax highlighting, code collapsing and code completion. It provides a sophisticated on the fly error detection, that splits errors in syntax errors, semantic errors and errors found during parsing. The code completion feature allows completion for standard PDDL constructs and dynamic list completions, that were used in the current project (TODO: technical terms!).

Tasks are organized into project, that means a project is composed of a domain and associated problem(s).

An interface allows the integration of command line planners in order to run and compare different planning software.

Furthermore it provides a XML Export/Import feature (i.e. XML->PDDL, PDDL->XML) and further common editor features like a line counter, bracket matcher and a auto-save feature.

that means syntax and semantic checking, syntax highlighting, code completion and project management. While colors for highlighted code can be customized, the background color of the tool is always white.

2.2 PDDL Mode for Emacs

PDDL-mode is a major Emacs mode for browsing and editing PDDL 2.2 files. It supports syntax highlighting by basic pattern matching, regardless of the current semantic, automatic indentation and completions. Code snippets for the insertion of domains, problems and actions are provided. A declaration menu shows all actions and problems in the current PDDL file.

2.3 itSIMPLE

The itSIMPLE project is a tool that supports the knowledge engineer in designing a PDDL domain by the use of UML diagrams. This approach is reversed to the approach mentioned in this paper: while itSIMPLE generates PDDL from UML, this paper generated UML from PDDL.

itSIMPLE is able to import and separate elements of domain and problems specified in a file. The itSIMPLE PDDL editor supports syntax highlighting by keywords and variables.

It also provides templates for PDDL constructs, like predicates, actions, goals, initial definitions.

itSIMPLE focuses on the initial states of the design process. The design environment is

2.4 GIPO

2.5 ModPlan

Also see VEGA plan visulazation on the MODplan page

- Very interesting: `http://www.tzi.de/~edelkamp/modplan/`

2.6 VISPLAN

2.7 Conclusion

As it can be seen, there is need for an up-to-date, customizable, text editor with PDDL support, that supports the current standard PDDL 3.1.

Chapter 3

Planning Basics and PDDL

Introduction to planing: `http://books.google.de/books?id=eCj3cKC_`
`3ikC&printsec=frontcover&dq=automated+planning&hl=en&sa=X&ei=3wgNU5fQIcHx4gSTsoDABA&re`
`esc=y#v=onepage&q=automated%20planning&f=false`

AI planning describes . . .

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.

PDDL planning task specifications are composed of two separate text files:

- Domain file: description of general types, predicates, functions and actions -> uninstanciated problem independent
- Problem file: description of a concrete problem environment -> instance specic

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

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

3.1 Domain File

```
(define (domain name)

  (:requirements :requirement1
```

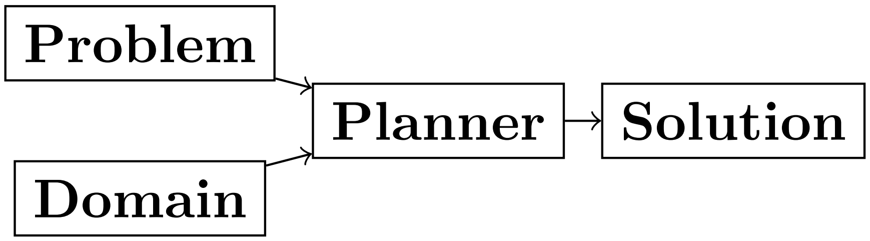


Figure 3.1: PDDL Planning workflow

```
                  :requirement2...)
  (:types subsubtype1 subsubtype2 - subtype1
          subtype1 - type1
          subtype2 - type2...
          type1 type2 ...  - object)

  (:predicates (predicateName1 ?var1 - typeOfVar1)
               (predicateName2 ?var2 - typeOfVar2 ?var3 - typeOfVar3)
               ...)

  (:action actionName1
    :parameters (...)
    :precondition (...)
    :effect (...))

  (:action actionName2
    :parameters (...)
    :precondition (...)
    :effect (...))

...)
```

The domain file contains the frame for planning tasks and determines, which types, predicates and actions are possible

Domain files have a strict format: All keyword arguments must appear in the order specified in the manual (an argument may be omitted, according to 1998, only the strict part requires this order) and just one PDDL definition (of a domain, problem, etc.) may appear per file (same here). Fox and Long 2003, p. 6.

3.1.1 Define

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

3.1.2 Requirements

The requirements part is not a mandatory part of a PDDL domain file. However, PDDL supports different "levels of expressivity", that means subsets of PDDL features McDermott et al. (1998, p. 1). As most planners only support a subset of PDDL the requirements part is 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 `:strips` For a list of current requirement flags and their meaning, see ... 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.

3.1.3 Types

If order to be able to use types in a domain file, the requirement :typing should be declared (TODO: is :adl enough?).

In order to assign categories to objects, PDDL allows for type definitions. 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).

3.1.4 Actions

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

3.1.5 Functions

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

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

3.2 Problem File

Problems are designed with respect to a domain. Domains usually have multiple problems p01.pddl, p02.pddl, ... Problems declare the initial world state and the goal state to be reached. They instantiate types, in they way that they create objects

3.3 Planning

A planning solution is a sequence of actions that lead from the initial state to the goal state. PDDL itself does not declare any uniform plan layout.

The input to the planning software is a domain and a belonging problem, the output is usually a totally or partially ordered plan. are software tools that Due to the yearly ICAPS, there is a broad range of available planners. This thesis uses the planner SGPLAN$_6$ Hsu and Wah (2008), a 'extensive' (in the sense of its supporting features) planner for both temporal and non-temporal planning problems.

An overview of different planners is given at `http://ipc.informatik.uni-freiburg.de/Planners`.

Chapter 4

Software Engineering Tools for AI Planning

4.1 Statement of Problem

Writing and maintaining PDDL files can be time-consuming and cumbersome Li et al. (2012). So, the following development tools shell support and facilitate the PDDL task design process and reduce potential errors.

Below, methods are presented for

Syntax Highlighting and Code Snippets Environment for Editing PDDL files

Class Diagram Generator The automation of the PDDL task design process. File input and output and dynamic generation (design level)

Human Planner Interaction An interactive PDDL environment: speech synthesis and recognition.

Domain Generator Mathematical limitations (design level)

4.2 Syntax Highlighting and Code Snippets

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 on-line 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. So, a Pygments extension was written, that allows for syntax highlighting in LaTeXdocuments.

4.2.1 Implementation and Customization

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

For the ease of creation, the PDDL syntax highlighter is 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* (2014) is a ST plugin, that helps to create, amongst others, ST packages, syntax definitions and 'snippets' (re-usable code).

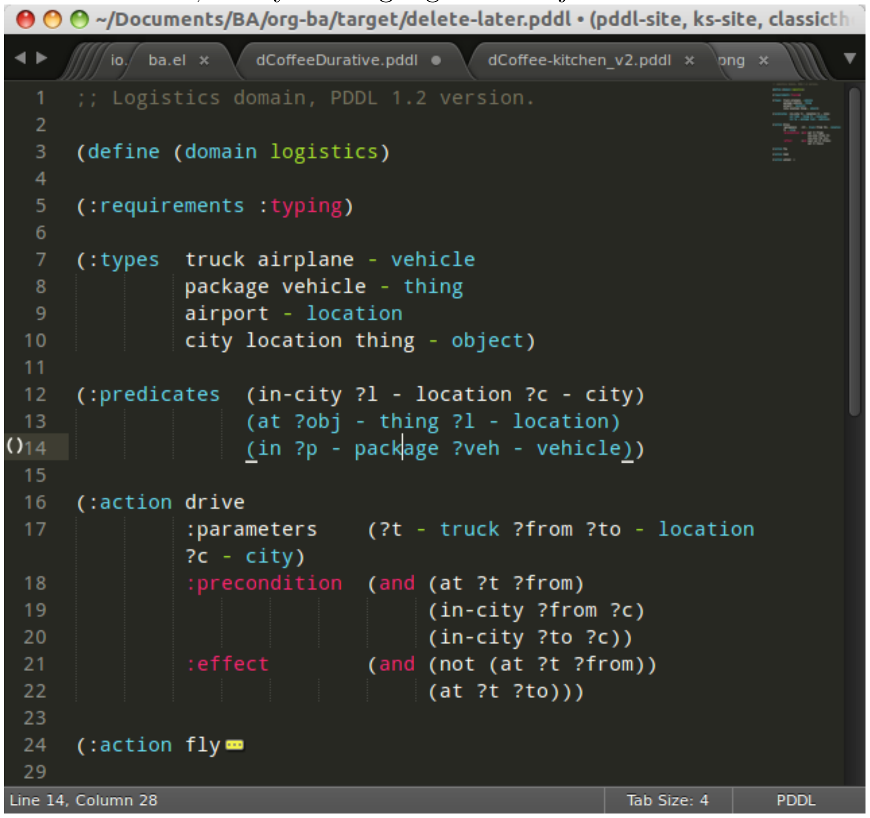
By means of Oniguruma regular expressions (Kosako 2007), scopes are defined, that determine the meaning of the PDDL code block. ST themes highlight different parts of the code through by the use of scopes. Scopes are defined by the use of regular expressions (regexes) in a tmLanguage file. The scope 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 backward compatible to previous version. It's based on the Backus-Naur Form (BNF) descriptions, formulated in Kovacs (2011), Fox and Long (2003), and McDermott et al. (1998).

The pattern matching heuristic that is implemented by the use of regular expressions is used for assigning scopes to the parts of the file. As a result of PDDL's LISP-derived syntax, PDDL uses the s-expression format for representing information (SOURCE!). So, the semantic of a larger PDDL part (sexpr) can be recognized by a opening parenthesis, followed by PDDL keyword and finally matched closing parentheses (potentially containing further sexpr). These scopes allow for a fragmentation of the PDDL files, so that constructs are only highlighted, if they appear in the right section.

The YAML-tmlanguage file is organized into repositories, so that expressions can be re-used in different scopes. This organization also allows for a customization of the syntax highlighter. The default

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.


~/Documents/BA/org-ba/target/delete-later.pddl • (pddl-site, ks-site, classicth
io. ba.el ×
dCoffeeDurative.pddl ●
dCoffee-kitchen_v2.pddl ×
ong ×

```
;; Logistics domain, PDDL 1.2 version.

(define (domain logistics)

(:requirements :typing)

(:types  truck airplane - vehicle
         package vehicle - thing
         airport - location
         city location thing - object)

(:predicates  (in-city ?l - location ?c - city)
              (at ?obj - thing ?l - location)
              (in ?p - package ?veh - vehicle))

(:action drive
         :parameters    (?t - truck ?from ?to - location
         ?c - city)
         :precondition  (and (at ?t ?from)
                             (in-city ?from ?c)
                             (in-city ?to ?c))
         :effect        (and (not (at ?t ?from))
                             (at ?t ?to)))

(:action fly
```

4.2.2 Usage and Customization

To enable syntax highlighting and code snippets in ST, the files of this repository have to be placed in the ST packages folder (`http://www.sublimetext.com/docs/3/packages.html`). Following, the features can be activated by changing ST's syntax to PDDL (`View->Syntax->PDDL`).

By using ST as editor, language independent ST features are supported, like auto completion of words already used in this file, code folding and column selection, described in the Sublime Text 2 Documentation.

The PDDL.YAML-tmlanguage file is split in two parts:

By default, all scopes are included.

4.2.3 Evaluation

4.3 Clojure Interface

PDDL, as planning language modeling capabilities are limited, a interface with a programming is handy a can reduce dramatically the modeling time. In IPC, task generators are used write extensive domain and problem files.

As PDDL's syntax is inspired by LISP (Fox and Long 2003, p. 64), using a LISP dialect for the interface seems reasonable. This thesis uses Clojure (Hickey 2008), a modern LISP dialect that runs on the Java Virtual Machine.

In this section, I will not only show a method for generating PDDL constructs, but also for reading in PDDL files are handling the input.

4.3.1 Basics

Through the higher-order filter method in Clojure, parts of PDDL files can be easily extracted. Like that, one can extract parts of the file and handle the constructs in a Clojure intern way.

As an example, the type handling will be represented here, but the basic approach is similar for all PDDL constructs.

The here developed tools should be platform independent with a development focus in UNIX/Linux systems, as most planners (source!) run on Linux.

4.3.2 Functions

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

4.3.3 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
```

TODO:

TODO Human Planner Interaction

4.4 Type Diagram Generator

Graphical notations, have some advantages compared to textual notations, as they simplify the communication between developers and help to quickly grasp the connection of related system units.

But for all that one disadvantage has to be accepted:

The UML was invented in order to standardize modeling in software engineering (SE). The UML consists of several part notations, the here presented tool uses the 'class diagram' notation, as PDDL types and classes in OOP have strong resemblance (see Tiago 2006, p 535).

Types play a major role in the PDDL design process: they are involved, besides their definition, in the constants, predicates and actions part. So, a fine grasp of their hierarchy, as well as their involved predicates becomes handy and assists the KE in the planning process. Types strongly resemble classes in object oriented programming As mentioned in chapter (...), the type definitions follow a specific syntax. For example `truck car - vehicle` would indicate, that both `truck` and `car` are subtypes of the super-type vehicle.

Subtypes and corresponding super-types can be extracted using regular expressions. #"(....)" matches every kind of that form and a Clojure-friendly representation in form of a hash-map can be created.

PDDL side ——————————————— Clojure side

'(:types ... ... — ...) {... [... ... ...], ...}

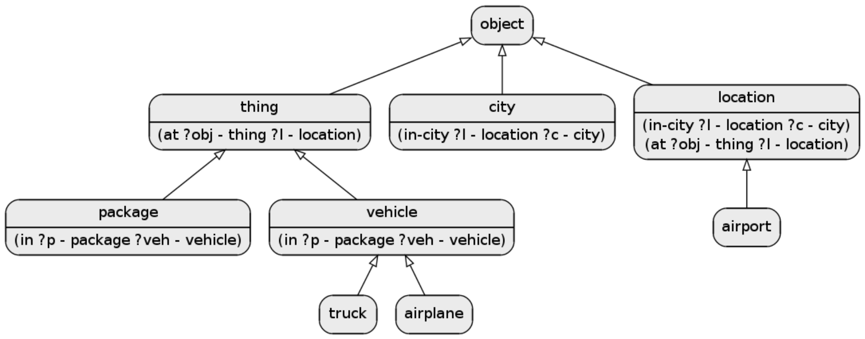

object
thing
(at ?obj - thing ?l - location)
city
(in-city ?l - location ?c - city)
location
(in-city ?l - location ?c - city)
(at ?obj - thing ?l - location)
package
(in ?p - package ?veh - vehicle)
vehicle
(in ?p - package ?veh - vehicle)
airport
truck
airplane

Figure 4.1: Part of a PDDL domain and the corresponding, generated UML diagram

Chapter 5

Analysis

5.1 Participants

Ten non-paid students (six female) took part in the experiment. All had knowledge about LISP syntax, but neither one had faced PDDL prior to this study.

5.2 Material

The usability of the Syntax Highlighter (see 4.2) and the Type Diagram Generator (see `Type Diagram Generator`) were tested.

5.3 Design

The participants had to

5.4 Procedure

Chapter 6

Conclusion and Outlook

The tools presented in this thesis have been designed to support knowledge engineers in planning tasks. They can support engineers in the early planning design process, as well as in the maintenance of existing domains and problems. The communication between engineers can be facilitated and

The plug-in for the editor ST could be further extended to provide features of common integrated developing environments (IDE). A build script for providing input to a planner for auto-matching domain and matching problem(s) (or problem and matching domain) in ST could be convenient. Detecting of semantic errors besides syntactic errors (as in PDDL Studio) could be the next step to detecting errors fast and accurate. Possible semantic errors could be undeclared variables or predicates in a domain specification.

6.1 Outlook

Besides ICKEPS, as mentioned in the introduction, also the yearly workshop Knowledge Engineering for Planning and Scheduling (KEPS) will promote the research in planning and scheduling technology. Potentially, the main effort of for implementing models in planning will be shifted from the manual KE to the automated knowledge acquisition (KA). Perception systems, Nevertheless, a engineer who double-checks the generated tasks will be irreplaceable.

Bibliography

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

[2] Edward A Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley Reading, 1983.

[3] 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.

[4] Radoslav Glinsk and Roman Barták. "VisPlan–Interactive Visualisation and Verification of Plans". In: *KEPS 2011* (2011), p. 134.

[5] Rich Hickey. "The clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM. 2008.

[6] Richard Howey, Derek Long, and Maria Fox. "VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL". In: *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*. IEEE. 2004, pp. 294–301.

[7] Chih-Wei Hsu and Benjamin W Wah. "The sgplan planning system in ipc-6". In: *Proceedings of IPC*. 2008.

[8] Wolfgang Köhler. "The mentality of apes." In: (1924).

[9] 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).

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

[11] Yi Li et al. "Translating pddl into csp#-the pat approach". In: *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*. IEEE. 2012, pp. 240–249.

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

[13] Drew McDermott et al. "PDDL-the planning domain definition language". In: (1998).

[14] 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.

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

[16] Tomas Plch et al. "Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio". In: *and Exhibits* (2012), p. 15.

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

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

Chapter 7

Appendix

```
(ns org-ba.core
  (:gen-class :main true)
  (:require [clojure.tools.reader.edn :as edn]
            [clojure.java.io :as io]
            [clojure.pprint :as pprint]
            [dorothy.core :as doro]
            [rhizome.viz :as rhi]
            [clojure.math.numeric-tower :as math]
            [quil.core :as quil]
            [clojure.java.shell :as shell]
            [me.raynes.conch :as conch]
            [me.raynes.conch.low-level :as conch-sh]
            [fipp.printer :as p]
            [fipp.edn :refer (pprint) :rename {pprint fipp}]
            [me.raynes.fs :as fs])
  (:import [javax.swing JPanel JButton JFrame JLabel]
           [java.awt.image BufferedImage BufferedImageOp]
           [java.io File]))

(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)))

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

(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)))



(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"
   (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>))
)"
   ))

(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]
  ;; Remove ':types' if it is present.
  (let [coll (if (= :types (first coll))
               (rest coll)
               coll)]
    ;; Capturing group 1 is type1.1 type1.2.
    ;; Capturing group 1 is type1.
    (re-seq #"((?:(?:\b[a-zA-Z](?:\w|-|_)+)\s+)+)-\s+(\b[a-zA-Z](?:\w|-|_)+)"
            (clojure.string/join " " coll))))


(defn types->hash-map-helper
  "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))

(defn types->hash-map
  "Splits types and converts them into a hash-map"
  [pddl-types]
  (types->hash-map-helper (split-up pddl-types)))

(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)) "}"))

(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}"))

(defn types-map-entry->dot-language
  "Converts one hash-map entry
to the dot language"
  [entry]
  (str
   "\"" (name (key entry)) "\""
   " -> "
   "{" (clojure.string/join " " (map #(str "\"" % "\"")  (val entry))) "}"))


(defn types-hash-map->dot-language
  "Converts a PDDL types hash-map
to the dot language notation"
  [pddl-types-map]
  (clojure.string/join "\n" (map types-map-entry->dot-language pddl-types-map)))

;;; Read PDDL predicates and generate UML 'type' diagram
(defn get-types-in-predicate
  "Takes a PDDL predicate,
  e.g. '(at ?x - location ?y - object)
  and returns the involved types, e.g.
  '(location object)"
  [pddl-pred]
  (remove
   (fn [s]
     (let [first-char (first (name s))]
       (or (= \- first-char)
           (= \? first-char)))) (rest pddl-pred)))

(defn pddl-pred->hash-map-long
  "Takes a PDDL predicate, e.g.
  '(at ?x - location ?y - object) and returns a
  hash-map, that assigns the involved types
  to this predicate, e.g.
  {location [(at ?x - location ?y - object)],
   object [(at ?x - location ?y - object)]}"
  [pddl-pred]
  (reduce (fn [h-map pddl-type]
            (assoc h-map
              pddl-type
              (list pddl-pred)))
          {}
          (get-types-in-predicate pddl-pred)))


(pddl-pred->hash-map-long '(at ?x - location ?y - object))

;;; TODO: Create short version wiht prolog predicate style
;;; e.g. at/2
(defn all-pddl-preds->hash-map-long
  "Takes a list of PDDL predicates and
  returns a hash-map of types and the
  assigned predicate"
  [pddl-preds]
  (let [pddl-preds (if (= :predicates (first pddl-preds))
                     (rest pddl-preds)
                     pddl-preds)]
    (apply merge-with concat
           (map pddl-pred->hash-map-long pddl-preds))))

(defn hash-map->dot
  "Converts a hash-map to
  dot language for creating
  UML diagrams"
  [h-map]
  (map (fn [map-entry]
         (str (key map-entry)
              "[label = \"{"
              (key map-entry)
              "|"
              (clojure.string/join "\\l"  (val map-entry))
              "}\"]\n"))
       h-map))

(defn hash-map->dot-with-style
  "Adds dot template to
hash-map>dot"
  [h-map]
  (str
   "digraph hierarchy {
node[shape=record,style=filled,fillcolor=gray92]
edge[dir=back, arrowtail=empty]
\n"
   (clojure.string/join (hash-map->dot h-map))
   "}"))


(defn PDDL->dot-with-style
  "Adds dot template to
hash-map>dot"
  [preds types]
  (str
   "digraph hierarchy {
node[shape=record,style=filled,fillcolor=gray92]
edge[dir=back, arrowtail=empty]
\n"

   (clojure.string/join (hash-map->dot (all-pddl-preds->hash-map-long preds)))
   (types-hash-map->dot-language (types->hash-map types))

   "}"))

;;; Example for Predicate:
(def predicates
  '(:predicates (at ?x - location ?y - object)
                (have ?x - object)
                (hot ?x - object)
                (on ?f - furniture ?o - object)))
```

```

;;; Example invocation:
(hash-map->dot-with-style (all-pddl-preds->hash-map-long predicates))


(defn get-PDDL-construct
  "Takes a PDDL keyword and a PDDL domain/problem
file and returns all parts of the file that
belong to the PDDL keyword."
  [pddl-keyword pddl-file]
  (filter #(and (seq? %)
                (= (keyword pddl-keyword)
                   (first %)))
          (read-lispstyle-edn pddl-file)))


                                        ; TODO: Throw error if length != 1
(defn get-PDDL-predicates
  "Get all predicates in a PDDL file"
  [pddl-file]
  (first (get-PDDL-construct 'predicates pddl-file)))

(defn get-PDDL-init
  "Get all predicates in a PDDL file"
  [pddl-file]
  (first (get-PDDL-construct 'init pddl-file)))


                                        ; TODO: Throw error if length != 1
(defn get-PDDL-types
  "Get all types in a PDDL file"
  [pddl-file]
  (first (get-PDDL-construct 'types pddl-file)))

(defn PDDL->dot
  "Takes a complete PDDL file
and generates a UML type diagram"
  [pddl-file]
  (PDDL->dot-with-style (get-PDDL-predicates pddl-file)
                        (get-PDDL-types pddl-file)))

(defn PDDL->dot-commandline-input
  "Assumes that the PDDL input is
a string and 'reads' this string"
  [pddl-file]
  (print "The type is " (type pddl-file))
  (PDDL->dot (edn/read-string pddl-file)))


(defn PDDL->dot-file-input
  "Reads PDDL file"
  [pddl-file-name]
  (PDDL->dot pddl-file-name))

;;;; math helper functions

```

```clojure
(defn sqr
  "Square of a number"
  [x]
  (* x x))

(defn round-places [number decimals]
  "Round to decimal places"
  (let [factor (math/expt 10 decimals)]
    (double (/ (math/round (* factor number)) factor))))

(defn euclidean-squared-distance
  "Computes the Euclidean squared distance between two sequences"
  [a b]
  (reduce + (map (comp sqr -) a b)))

(defn euclidean-distance
  "Computes the Euclidean distance between two sequences"
  [a b]
  (math/sqrt (euclidean-squared-distance a b)))

;;;; End math helper functions

(defn calc-distance-good
  "Calculates the distance and writes
the calculated distances to a string
IS VERY GOOD !!!"
  [locations]
  (for [[ _ loc1 & xyz-1] locations
        [ _ loc2 & xyz-2] locations]
    ;; Euclidean distance rounded to 4 decimal places.
    (list 'distance loc1 loc2 (round-places (euclidean-distance xyz-1 xyz-2) 4))))

(defn get-specified-predicates-in-pddl-file
  "Extracts all locations in the predicates part
(by the specified name) in a PDDL file"
  [pddl-file predicate-name]
  (filter #(and (seq? %)
                (= predicate-name (first %)))
          (get-PDDL-predicates pddl-file)))

(defn get-specified-inits-in-pddl-file
  "Extracts all locations in the init part
(by the specified name) in a PDDL problem"
  [pddl-file predicate-name]
  (filter #(and (seq? %)
                (= predicate-name (first %)))
          (get-PDDL-init pddl-file)))

(defn calc-distance
  "Calculate distances of PDDL objects"
  [locations]
  (for [[ _ loc1 & xyz-1] locations
        [ _ loc2 & xyz-2] locations]
    ;; Euclidean distance rounded to 4 decimal places.
    `(~'distance ~loc1 ~loc2
                 ~(euclidean-distance xyz-1 xyz-2))))
```

```

; LOOK UP: extended equality: 'hello = :hello

(defn add-part-to-PDDL
  "Takes a PDDL domain or problem
and add the specified part to the
specified position"
  [pddl-file position part]

  (map #(if (and (seq? %)
                 (= (keyword position) (first %)))
          (concat % part)
          %)
       (read-lispstyle-edn pddl-file)))

(defn find-new-file-name
  "Take a filename and determines, the new number
that has to be added to create a new file. E.g.
file1.img file2.img file3.img means that, file4.img
has to be created"
  [filename extension]
  (loop [n 0]
    (if-not (io/.exists (io/as-file
                         (str filename n extension)))
      (str filename n extension)
      (recur (inc n)))))


;;; Copied from https://www.refheap.com/9034
(defn exit-on-close [sketch]
  "Guarantees that Clojure script will be
exited after the JFrame is closed"
  (let [frame (-> sketch .getParent .getParent .getParent .getParent)]
    (.setDefaultCloseOperation frame javax.swing.JFrame/EXIT_ON_CLOSE)))


(defn extract-locations-from-file
  "Read a Blender LISP file and write object positions to out-file"
  [file-in file-out]
  (let [map-destructorer-local (fn [[_addgv _furniture object
                                     [_make-instance _object-detail
                                      _pose [_tfmps
                                             _type-name
                                             _type-num
                                             [_vector-3d x y z & more]
                                             & _more1]
                                      & _more2]]] (list "location" (name object) x y z))]
    (with-open [rdr (java.io.PushbackReader. (io/reader file-in))]
      (println
      (doall
          (map map-destructorer-local
               (filter #(and (seq? %) (= 'addgv (first %)))
                       (take-while #(not= % :end)
                                   (repeatedly  #(edn/read {:eof :end} rdr))))))))))


```

```clojure
;; Main method
;; TODO: Command line options
(defn -main
  "Runs the input/output scripts"
  [& args]

  (cond
   ;; Create a new PDDL project
   (= "new" (first args))
   (let [project-name (second args)]
     (fs/mkdir project-name)
     (fs/mkdir (str project-name "/dot"))
     (fs/mkdir (str project-name "/diagrams"))
     (fs/mkdir (str project-name "/domains"))
     (fs/mkdir (str project-name "/problems"))
     (fs/create (io/file (str project-name "/domain.pddl")))
     (fs/create (io/file (str project-name "/p01.pddl"))))

   ;; -l flag for adding locations in PDDL file
   (= (second args) "-l")
   (let [content (add-part-to-PDDL (first args)
                                   'init
                                   (calc-distance-good
                                    (get-specified-inits-in-pddl-file (first args)
                                                                      'location)))
         new-filename (clojure.string/replace-first (first args)
                                                    #"(.+).pddl"
                                                    "$1-locations.pddl")] ; TODO: location as arg

     (write->file new-filename (pprint/pprint content)))


   ;; Write dot graph to file.
   :else
   (let [input-domain (first args)
         new-dot-filename (find-new-file-name "dot/dot-diagram" ".dot")
         new-png-filename (find-new-file-name "diagrams/png-diagram" ".png")
         input-domain-filename (fs/name input-domain)
         domain-version (find-new-file-name
                         (str "domains/" input-domain-filename) (fs/extension input-domain))]

     ;; Save input domain version in folder domains.
     (fs/copy+ input-domain domain-version)

     ;; Create folders for dot files and png diagrams
     (fs/mkdir "dot")
     (fs/mkdir "diagrams")

     ;; Create dot language file in dot folder.
     (doall
      (write->file new-dot-filename
                   (print (PDDL->dot-file-input input-domain))))

     ;; Create a png file from dot
     (fs/exec "dot" "-Tpng" "-o" new-png-filename new-dot-filename)

     ;; Settings for displaying the generated diagram.
     (def img (ref nil))

     (defn setup []
       (quil/background 0)
       (dosync (ref-set img (quil/load-image new-png-filename))))

     (def img-size
       (with-open [r (java.io.FileInputStream. new-png-filename)]
         (let [image (javax.imageio.ImageIO/read r)
               img-width (.getWidth image)
               img-height (.getHeight image)]
           [img-width img-height])))

     (defn draw []
       (quil/image @img 0 0))

     ;; Display png file in JFrame.
     (exit-on-close
      (quil/sketch
       :title (str "PDDL Type Diagram - " input-domain-filename)
       :setup setup
       :draw draw
       :size (vec img-size)))))
```