

MyPDDL - A Modular Knowledge
Engineering System for the Planning Domain
Definition Language

Volker Strobel

April 2, 2014

Contents

1	Introduction	1
2	Planning Basics and Introduction to PDDL	3
2.1	Analysis	5
2.2	Domain File	6
2.2.1	Define	6
2.2.2	Requirements	6
2.2.3	Types	7
2.2.4	Predicates	8
2.2.5	Actions	9
2.3	Problem File	9
2.3.1	Define (define (problem NAME) ...)	9
2.3.2	Domain (:domain NAME)	10
2.3.3	Objects (:objects ...)	10
2.3.4	Init (:init ...)	10
2.3.5	Goal (:goal ...)	11
2.4	Planning	11
3	Related Work	12
3.1	PDDL Studio	12
3.2	ItSIMPLE	13
3.3	PDDL-Mode for Emacs	14
3.4	Conclusion & Summary	15
4	Software Engineering Tools for AI Planning	16
4.1	Statement of Problem	16
4.2	General Interface between PDDL and Clojure (/myPDDL-i/f)	17
4.3	Create PDDL Projects (myPDDL-new) p	18
4.4	Syntax Highlighting	19
4.4.1	Implementation and Customization	19
4.4.2	Usage and Customization	20

4.5	Code Snippets (<i>myPDDL-snp</i>)	23
4.6	Distance Calculation for PDDL Locations (<i>myPDDL-loc</i>)	24
4.7	Type Diagram Generator (<i>myPDDL-gen</i>)	25
4.7.1	Limitations	27
4.8	Syntax Highlighting and Code Snippets (<i>myPDDL-sub</i>)	27
5	Analysis	29
5.1	Design Goals	29
5.2	Empirical Study	29
5.3	User Study	30
5.3.1	Participants	30
5.3.2	Material	30
5.3.3	Design	31
5.3.4	Procedure	31
5.3.5	Results	32
6	General Discussion	33
7	Outlook and Conclusion	34
7.1	Future Work	34
7.2	Outlook	35
7.3	Conclusion	35
8	Appendix	39

Abstract

TODO: some context

Writing and maintaining planning problems, specified in the widely used *Planning Domain Definition Language* (PDDL), can be difficult, time-consuming and error-prone. This thesis will present myPDDL, a toolkit that helps knowledge engineers to develop, visualize and manipulate (PDDL) planning tasks. For these purposes, new, structured PDDL projects can be created, consisting of a domain template and associated problem files. A tool visualizes the type hierarchy in PDDL domains, allowing knowledge engineers to understand the representation structure at a glance and to keep track of developments by a revision control. Distances between objects specified by predicates in a problem file can be calculated with an auxiliary tool, presenting a way of bypassing PDDL's limited modeling capacity. All these tools are based on an interface that provides a general way for reading and writing PDDL specifications by the use of the programming language Clojure. These tools are made accessible in the text editor Sublime Text, where two additional features, a syntax-highlighter and the possibility of using code snippets for common PDDL constructs, were implemented. A small user study, conducted with eight inexperienced PDDL users, shows some initial evidence that the syntax highlighting feature and the type diagram generator could support knowledge engineers in the design and analysis process. So, the users detected $..%$ more errors using the syntax highlighter in the same time as non-users and the average task completion time for questions on a hierarchical domain could be reduced by $..%$.

Chapter 1

Introduction

Have you ever struggled to devise a sequence of actions in order to complete the Rubics' Cube? Automated planning could support you. Being a key aspect of Artificial Intelligence (AI), AP is concerned with the devise of a plan of actions to achieve a desired goal. It can be both a tool for creating automated systems as well as a means for supporting and understanding human behavior. However, the usefulness of planning largely depends on the formalization of the underlying problem task (source). The Planning Domain Definition Language (PDDL) (source) is a formal language and the quasi standard for the description of planning tasks (source). To lay the foundations to this thesis, a basic introduction to planning and PDDL will be given in Chapter 2. Creating these specifications can be error-prone and time consuming (source) ¹. Therefore, focusing on the usability of tools for planning languages and hence facilitating the Knowledge Engineering (KE) process is worthwhile. The aim of KE in automated planning is thereby to turn the process of writing planning tasks from "an art into an engineering discipline" (source). This requires the analysis of the underlying problem itself and the systematic use of appropriate tools specialized for developing planning tasks.

In the scope of this thesis, the modular toolkit myPDDL (/m/odeling /e/fficiently PDDL) was developed in order to tackle frequent needs of knowledge engineers, like project management, efficient development, debugging and error-detection, team collaboration features and visualization, automation and inspecting, maintenance features (source) and pushing the acceptance and usage of PDDL in real world models (source). Existing tools will be critically reviewed and compared to myPDDL in Chapter 3.

MyPDDL should support knowledge engineers in the entire design cycle of

¹http://icaps14.icaps-conference.org/workshops_tutorials/keps.html

specifying planning tasks. Therefore, it allows for the creation of new PDDL projects filled with templates. A syntax highlighting feature that can allow for a faster error-detection and help with understanding parts of PDDL files at a glance. As PDDL code consists of recurrent blocks, code snippets allow the user to insert and complete syntactic code templates in a convenient way. Understanding the textual representation of deep type hierarchies in domain files can be confusing, so a further myPDDL tool allows for the visualization of them. PDDL's limited modeling possibilities were bypassed by developing a PDDL/Clojure interface, that supports the conversion of PDDL code into Clojure code and vice versa and therefore allows, amongst others, for a partial automation of modeling tasks. A tool based on this interface calculates distances between objects specified in a problem file. The modules, which are integrated in the customizable and extensible Sublime Text editor will be described in detail in Chapter 4. Since a main focus of the toolkit is the ease of application, a user test was conducted (Chapter 5) with eight subjects that had no prior experience with planning languages. The results indicate that both error-detection and the understanding of a given domain can be facilitated by myPDDL. The implications, as well as general ideas, will be discussed in Chapter 6. Finally, an outlook for future research and developments in the field concludes this thesis (Chapter 7).

Chapter 2

Planning Basics and Introduction to PDDL

Very good for While humans... <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.362.4331&rep=rep1&type=pdf#page=7>

The human brain is an astonishing structure that allows us to get along in a highly complex world and plan more or less rational reasons for our past or planned actions. While computer systems are yet to fully master these skills, the study of artificial intelligence (AI) tries to narrow this gap. For this purpose, constructs are needed that can represent the information about the world and the problem. The discipline that is engaged with the integration of world information into a computer system by a human expert is called knowledge engineering (KE) Feigenbaum and McCorduck [2]. In automated planning, this is usually done using a planning language. Planning is then the decision making process that finally leads to a sequence of actions for solving the specified problem.

The planning domain definition language (PDDL) is a formal language and the quasi standard for the description of planning tasks. PDDL was first described in PDDL-the planning domain definition language (1998) and has been in constant development since then and is currently used in version 3.1.

Consider the following (fictional) world that should be integrated into a computer system using pddl:

If a hacker is hungry, he has to eat some pizza in order to exploit vulnerable software.

In this description, we can identify several constructs, that should somehow be integrated into the computer. There are:

Types of entities: The world consists of hackers, software and pizza.

Logical states: Hackers can be hungry or not, software can be vulnerable or not.

Actions: Hackers can exploit (that means hack into) software and they can eat pizza.

This world description could be specified in PDDL, using a domain file. The domain file provides general, abstract constructs and conditions.

Next, consider the following problem particular to this domain:

Gary is a *hungry* hacker who should somehow exploit the vulnerable software *MagicFailureApp*. Some pepperoni pizza is laying around.

Again, several constructs can be identified:

Objects The hacker Gary, the pepperoni pizza, the software

Initial state Gary is hungry and the software 'MagicFailureApp' is vulnerable

Goal state MagicFailureApp is exploited.

Assume, that Gary wants to have help of an automated planning system, in order to plan the sequence of required actions (*Who has to eat pizza?* and *What to hack?* and which one of both has to be done first?), leading from the initial state to the goal state. These specifications have to be formalized, this time in a problem file. Finally, Gary will be able to feed the domain and problem file into a planner and generate the sequence of required actions.

Summing up, PDDL planning tasks are composed of two separate, corresponding files:

Domain file General, problem-independent description of types, predicates (logical states) and actions.

Problem file Specification of a concrete problem within a particular domain, expressed by the initial state and the goal state. The values are assigned to the templates provided by the domain file.

This separation allows for a powerful process of task modeling: While general instances are described in the domain file, specific instances of problems are created in the problem files. So, one abstract modeling of a *world* can be used for solving many problem instances.

Figure ?? visualizes the workflow for planning in PDDL.

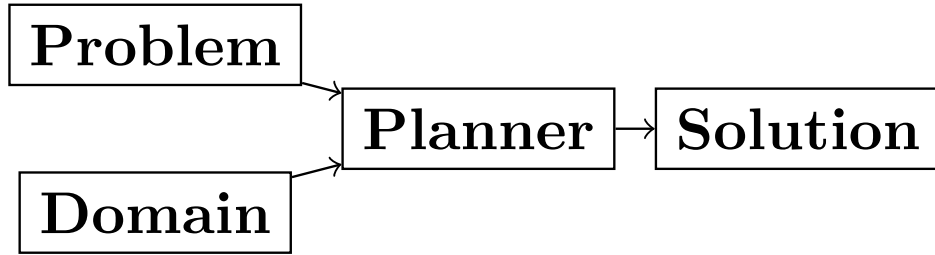


Figure 2.1: PDDL Planning workflow

TODO: Add predicates and actions to domain, init and goal to problem and sequence of actions to plan The PDDL workflow. domain.pddl and problem.pddl represent typical planning specification files, with the standard file extension *.pddl*

PDDL is manifold and not all parts are mandatory components of task specifications. More complete descriptions as well as a formulations in Backus-Naur form (BNF) can be found in Fox and Long [3] for PDDL 2.2 and Kovacs [7] for PDDL 3.1. The rest of this section will give general design guidelines and an introduction to PDDL, to provide a basis for the rest of this thesis. To this end, the syntax of common constructs of domain and problem files shall be investigated further in this section, in a step-by-step approach, where both files are described by the above described example *Gary's Hacker World*.

TODO:Functions, Metrics

2.1 Analysis

In order to integrate information into a computer system, first and foremost, the problem has to be understood. How to Design Classes (HtDC), describes a incremental process for modeling specification in object oriented programming (OOP). The general principles will be transferred to PDDL, so that in the style of the *design principles* of object oriented programming (HtDC) a stepwise, iterative modeling process can be identified:

Analysis: Every task specification begins by an analysis of the informal world and the problem statement. In this design step, one determines relevant types, adequate examples and identifies both the initial and the goal state. One keeps track of the analysis using any kind of list.

Type diagram: Based on the preceding analysis, the relationship of the identified types is represented, using a diagram. This can be done by

pen and paper or by means of a graph editor.

Domain definition: In this step, the (graphical) diagrams are translated into PDDL. Furthermore predicates and actions are declared.

Problem definition: After completing the domain definition, objects can be instantiated in the problem file. By means of the predicates, declared in the domain file, the initial and goal logical values are defined.

Two further steps can be identified:

Planning Provide domain and problem definition to a planner.

Plan analysis Inspect the resulting plan and optionally restart at a earlier design step.

(Plan analysis can be supported by VAL or Visplan A further convenient method is the use of ITSIMPLE, so that the hierarchy can be translated to PDDL.)

2.2 Domain File

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

2.2.1 Define

Every domain file starts with `(define (domain DNAME) ...)` where, DNAME specifies the name of the domain. It must be a string that starts with a character, and then contains further characters(**a-z**), numbers (**0-9**), hyphens (**-**) or underscores (**_**). A semicolon (**;**) declares the rest of the line as comment. PDDL's syntax is case insensitive.

```
; Gary's Hacker world - A realistic example
(define (domain garys-hacker-world)
```

2.2.2 Requirements

- PDDL: Levels of expressivity (level 1 .. 4)
- Formal description of PDDL tasks

PDDL is composed of subsets of PDDL features McDermott et al. [10, p. 1]. As most planners only support particular elements of PDDL, the requirements part is useful for a planner to determine if it is able to act on a given problem. Often used requirements are:

:strips The most elementary subset of PDDL that supports approximately the specifications declared in the STRIPS specification of 1971. If no requirements are declared :strips is assumed.

:typing Enables the typification of variables (see 'Types' below), so that variables have to be of particular type.

:equality Specifies equality ($\backslash=$) as built-in predicate, so that

Besides **:typing**, Gary's hacker world will use a further requirement:

:negative-preconditions Allows for the specification of negative preconditions in actions, so that a action can only be executed if a predicate is *not* true at the beginning of the action.

Complete lists of requirements and their meaning can be found in Fox and Long [3] for PDDL 2.1 and Kovacs [7] for PDDL 3.1.

```
(:requirements :typing
               :negative-preconditions)
```

Listing 1: The requirements part of Gary's Hacker World

2.2.3 Types

In the real-world, there will be often individual objects of the same kind or *type*. There may be lots of different pizza types in existence, sharing common properties. Each pizza was made from a similar set of ingredients and may contain similar components.

The **(:types ...)** part, PDDL allows for declaring types and thereby structuring the domain. Relations can be expressed by a type hierarchy, whereby any type can be a subtype of yet another type. Typed lists are used to assign types to variable lists. 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).

Types are declared by a list of strings, followed by a hyphen (-), followed by the higher-level type. Every PDDL domain includes the built-in types `object` and `number`, whereby every defined type is subtype of `object`.

```
(:types hacker non-hacker - person
      white-hat gray-hat black-hat - hacker
      application system tool - software
      driver os - system
      pizza person software - object)
```

Listing 2: The type hierarchy for Gary’s Hacker World, consisting of different types of persons, hackers, software. The elements on the left-hand side (for example `driver os`) are declared subtypes of the right-hand side (`system`). The hierarchy is expressed by using a type both on the left-hand side (for example `hacker`) and on the right-hand. *Software* can be *application* software, *system* software or programming *tools*. System software can be further divided into drivers and operating systems.

2.2.4 Predicates

Predicates are templates for logical facts and describe the properties of objects. They can be either true or false. The `:(predicates ...)` part declares predicate names and number of arguments, together with the corresponding type. The general syntax is `(p ?v1 - t1 ?v2 - t2 ...)`, whereby `?` followed by a name (`v1`, `v2`), declares a variable, and the expression `(t1, t2)` after the minus sign (-) determines the type of this variable. Thereby, the type has to be declared in the typing section first. The number of variables determines the arity of a predicate, ranging from zero arguments (0-ary predicate) to any positive integer (n-ary predicate). Type assignments for variables that have the same type and are declared side by side can be grouped, so that `(p ?v1 ?v2 - t)` is similar to `(p ?v1 - t ?v2 - t)`.

```
(:predicates (has ?s - software ?p - person)
             (hungry ?p - person)
             (vulnerable ?s - software)
             (exploited ?s - software))
```

Listing 3: This section declares four predicates: `has` (2-ary), `hungry`, `vulnerable` and `exploited` (1-ary).

2.2.5 Actions

PDDL is an action-centered language. Actions are the operators in PDDL and are able to change the truth value of predicates (and therefore properties of objects), so that problems can be solved (if there exists a solution). Actions usually consist of three parts

:parameters A (typed) argument list that determines which variables can be used in the precondition and effect part.

:precondition Describes the applicability of an action. The logical expression that is expressed in this part has to be **true**, before an action can be applied.

:effect The effect describes the post-condition of an action, that means it assigns new truth values to the mentioned predicates.

```
;; Eat a delicious pizza
(:action eat-pizza
 :parameters (?pi - pizza ?p - person)
 :precondition (hungry ?p)
 :effect (not (hungry ?p)))

;; Exploit vulnerable software of a victim
(:action exploit
 :parameters (?h - hacker ?s - software ?p - person)
 :precondition (and (has ?s ?p)
                   (vulnerable ?s)
                   (not (hungry ?h)))
 :effect (exploited ?s)))
```

2.3 Problem File

Planning problems are described by the pairing of domain and problem files. Problem files declare the initial world state and the goal state to be reached on the basis of the logical values of the instantiated predicates. Furthermore, they create (instantiate) concrete objects.

2.3.1 Define (define (problem NAME) ...)

Analog to the domain definition, problem files are initiated with **(define (problem PNAME) ...)**, whereby, PNAME declares the name of the problem.

```
(define (problem garys-huge-problem)
```

Listing 4: Initiating the problem file with the name garys-huge-problem

2.3.2 Domain (:domain NAME)

Problems are designed with respect to a domain, which has to be declared here. That means that DNAME in (:domain DNAME) and DNAME (define (domain DNAME) ...) in the particular domain file have to be equal.

```
(:domain garys-hacker-world)
```

Listing 5: The domain file "garys-hacker-world" is the corresponding domain file to the problem garys-huge-problem

2.3.3 Objects (:objects ...)

Whereby types are only templates, they can be created (instantiated) in the =(object...) part. That means that types are assigned to concrete objects.

```
(:objects big-pepperoni-pizza - pizza
          gary - white-hat
          gisela - non-hacker
          magicfailureapp - application)
```

Listing 6: This part creates concrete objects from the type templates. So, = magicfailureapp - application= means that the object magicfailureapp is of the type application.

2.3.4 Init (:init ...)

A problem consists of two situations, the current one, which is called the initial situation and the desired one which is called the goal state.

This part describes the initial state of the world by a list of instantiated predicates that are declared as true. All other predicates are assumed to be false (closed-world assumption).

```
(:init (hungry gary)
      (vulnerable mysterious-tex-mex-mix)
      (has magicfailureapp gisela))
```

Listing 7: The initial situation in Gary’s Hacker World, where the hacker Gary is hungry, the application magicfailureapp is vulnerable and belongs to the non-hacker Gisela.

2.3.5 Goal (:goal ...)

The goal state refers to the situation one likes to obtain. It is described by the logical fact that is desirable and should be reached by the execution of the plan. In PDDL, several goals would be combined with `=(and ...)`, whereby all non-specified predicates are also nonrelevant, that means that they can be either true or false in the goal state.

```
(:goal (exploited magicfailureapp))
```

Listing 8: In the end, the software magicfailureapp should be exploited.

2.4 Planning

The effort of the formalization of the planning task will be finally rewarded by the automatic generation of the plan.

The input to the planning software is a domain file and a corresponding problem file, the output is a sequence of actions (the *plan*), leading from the initial state to the goal state.

Due to the yearly ICAPS, there is a broad range of available planners. This thesis uses the planner SGPlan₆ Hsu and Wah [5], 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>.

Additionally, the quality of error messages is very diversified. While some simple state: error occurred, other list the problem and the line.

Chapter 3

Related Work

3.1 PDDL Studio

PDDL Studio [12], is an application for creating and managing PDDL projects. A project is regarded as a collection of PDDL files. Its IDE is inspired by Microsoft Visual Studio and imperative programming paradigms. Its core function is the PDDL project management, consisting of managing PDDL projects and creating, adding, so that corresponding as well as inspecting, analyzing and modifying the underlying domain and problem files. Besides general editing features like line counting, bracket matching and auto-save, it supports PDDL specific editing features including syntax highlighting, code folding (collapse code blocks to see only a single visible line) and context aware code completions, all based on a PDDL to XML parser. This parser can also be used to convert PDDL to XML files and vice versa for domain and problem file editing. Also based on this parser is a included, sophisticated on the fly error detection, recognizing both syntax errors (missing keywords, parentheses, etc.) and semantic errors (wrong type of predicate parameters, misspelled predicates, etc.). As semantic errors can be of a *interfile nature*, meaning that there is a mismatch between domain and problem file, PDDL Studio can detect such errors. TODO: Explain further. The code completion feature allows for the selection of completion suggestions for a for standard PDDL constructs and dynamic list completions, that were used in the current project (TODO: technical terms!). An interface allows the integration of command line planners in order to run and compare different planning software. 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. In its most recent version (of 15.6.2012), PDDL Studio's parser supports PDDL 1.2, the

official language of the first and second IPC in 1998 and 2000 respectively. Since then, PDDL has largely evolved, the most recent and most powerful version is PDDL 3.1, supporting amongst others durative actions. PDDL Studio does not support the insertion of larger code skeletons (called *snippets* in this thesis). The customization features (without editing the C source code) are limited to the choice of font style and color of highlighted PDDL expressions. PDDL Studio is written as standalone program, meaning that there are no PDDL independent no extensions .

3.2 ItSIMPLE

The ITSIMPLE project is a graphical interface that allows for designing planning models in an object-oriented approach, using Unied Modeling Language (UML) diagrams. UML was invented in order to standardize modeling in software engineering (SE). It 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). ITSIMPLE proposes UML.P (UML in a Planning Approach), a UML variant that specifies a structure for Class (domain specification), Object (problem specification) and StateChart Diagrams (dynamic behavior of actions).

ITSIMPLE's main focus is to support knowledge engineers in the initial stages of the design phase by providing an opportunity for the transition of the informality of real world requirements to domain models as formal specifications. The assertive statement is to provide a tool for a “{”disciplined process of elicitation, organization and analysis of requirements}. Petri Nets can be generated from the UML model and be used to validate the planning domain's static and dynamic bevhavior. Finally, a PDDL representation can be generated from the UML diagram, if required, edited, and finally used as input to a variety of planning systems. The generated plan can be inspected using the in-build plan analysis, consisting of a plan visualization and plan simulation (TODO: write some more info). ITSIMPLE's mdoeling workflow is unidirectional, as changes in the PDDL domain do not affect the UML model and UML models have to be modeled manually, meaning that they cannot by generated using PDDL.

Starting in version 4.0 (currently in beta status as of writing of this thesis) ITSIMPLE expanded its features to allow the creation of PDDL projects from scratch (i.e. without UML to PDDL translation process). Thus far, the PDDL editing features are basic (see YouTube video). A minimal syntax highlighting feature recognizes PDDL keywords and variables. Furthermore, ITSIMPLE provides templates for PDDL constructs (similar to the code snippets

presented in this thesis), consisting of requirement specifications, predicates, actions, goals and initial definitions.

ITSIMPLE's original and main design approach is reversed to the process presented in this paper. While ITSIMPLE generates PDDL models from UML specifications, myPDDL generates type diagrams from PDDL. So, while ITSIMPLE focuses on the initial design phase, the tools presented here are made for later stages.

Agent, environment, problems of semantic assumptions, disadvantages, advantages, associations (many arrows could be distracting),

myPDDL allows for a representation of a arbitrary, n-ary predicates, without making assumptions about semantics. On the one hand this allows for a visualization of any PDDL domain (and n-ary predicates), while on the other hand (that means, if semantic assumptions are made correctly, U

ITSIMPLE's modeling process is focused on a graphical design process and the newly added PDDL editing features are basic, consisting of highlighted keywords and variables. The templates primarily insert PDDL keywords, without showing the required syntax (e.g. `(:predicates ...)` instead of `(:predicates (predicate-name ?x - object)`). ITSIMPLE is not customizable (without editing the Java source code). It is not possible to define custom key shortcuts for commands. General editor features, like to displaying line numbers, matching brackets or code folding are not (yet) supported.

myPDDL shell support both, the initial design process of creating domains (by code snippets and the Clojure interface) and the later step of checking validity of existing domains and problems by the type generator (and possibly extending them).

3.3 PDDL-Mode for Emacs

PDDL-mode (announced 2005 in a mailing list) is a major Emacs mode for browsing and editing PDDL 2.2 files. It provides syntax highlighting by basic pattern matching of `{pddl}` keywords, variables and comments, regardless of the current context. Furthermore, it provides automatic indentation and completions and bracket matching. Code snippets for the insertion of domains, problems and actions are provided. A declaration entry in the Emacs menu bar shows all actions in the current PDDL file and allows for jumping to the definition.

Being an Emacs mode, PDDL-mode is highly and easily customizable. Text editor features, like auto-completion, can be extended independently of this mode, by installing further Emacs modes.

myPDDL uses Sublime Text, an editor, that is an extensible and customiz-

able editor as well. The syntax highlighting feature of *myPDDL* supports all PDDL versions, up to the most recent version 3.1. in contrast to *PDDL-mode*, *myPDDL-h*'s syntax highlighting feature is context-dependent and more extensive, as it can recognize almost any PDDL construct and highlight it according to its semantic.

By syntax highlighting, both tools can support code navigation, however, *PDDL-mode* does not allow for an fast and evident error detection.

3.4 Conclusion & Summary

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. *myPDDL* integrates and expands features described in this section, while keeping an focus on application, efficiency and customization opportunities.

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. [8]. To this end, a collection of extensible development tools (*myPDDL*) shell support and facilitate the PDDL task design process and reduce potential modeling errors. Main goals are a fast and reliable (or good) design process that should support the collaboration between knowledge engineers and thereby promote the use of PDDL in real-world applications.

myPDDL is a extensible, modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL domains and problems. Based on a general interface between PDDL and Clojure, allowing for file input (reading PDDL) and output (PDDL domain and problem generation), the following integral parts of *myPDDL* will be presented in the following sections:

myPDDL-new Create a new PDDL project folder with domain and problem skeletons

myPDDL-gen A PDDL type diagram generator for analyzing the structure of type and object hierarchies.

myPDDL-loc Automated distance calculation for PDDL locations

myPDDL-syn A syntax highlighting feature that colorizes PDDL constructs by its context

myPDDL-snp Code snippets (templates), which can be inserted in PDDL files.

myPDDL-sub The integration of *myPDDL-syn*, *-snip* and *-gen* into a environment to be used in Sublime Text

myPDDL is focused on customizability and extensibility, ranging from the choice of key bindings and themes to the adaptability of the code snippets to the point of adding a new module based on the general interface.

TODO: Mindmap for modular hierarchy.

4.2 General Interface between PDDL and Clojure (/myPDDL-i/f)

Being a planning language, PDDL's modeling capabilities are limited. For this reason, a interface with a programming seems reasonable and can partly automate the modeling process as well as reduce the modeling time (see e.g. distance calculator). Furthermore, In IPC, task generators are used to write extensive domain and problem files. As PDDL is used to create more and more complex domains (SOURCE1, SOURCE2, SOURCE3, ...).

In this section, a general approach for generating PDDL constructs, but also for reading in domain and problem files, handling, using and modifying the input, and generating PDDL files as output, will be presented.

While it seems to be reasonable to further extend PDDL's modeling capability to at planning time instead of modeling time, a modeling support tool as preprocessor is appropriate in any case (<http://orff.uc3m.es/bitstream/handle/10016/14914/proceedings-WS-IPC2012.pdf?sequence=1#page=47>)

As PDDL's syntax is inspired by LISP [3, p. 64], using a LISP dialect for the interface seems reasonable, as file input and output methods can use s-expressions instead of regular expressions. This way, PDDL expressions can be extracted from a task specification and written back in a similar manner, and parts of PDDL files can be accessed in a convenient way. This thesis uses Clojure [4], a modern LISP dialect that runs on the Java Virtual Machine.

The interface is built on two methods:

read-construct(keyword,file) Allows for the extraction of a PDDL construct, specified by its name.

add-construct(file,position,part) Provides a means for adding PDDL constructs to a specified position, indicated by a keyword.

Once a part is extracted and represented in Clojure, the processing possibilities are manifold. An implementation using the **read-construct** method

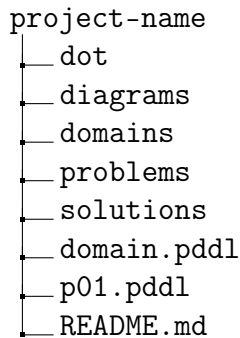


Figure 4.1: The project folder structure created by myPDDL-new. project-name is chosen by the user.

is myPDDL-gen. The combination of these two methods allows for the manipulation of existing PDDL files, as well as the creation of new files, as shown by myPDDL-loc. Possible further applications could consist of domain and problem generators, ...

4.3 Create PDDL Projects (myPDDL-new) p

Prior to each implementation of a PDDL task specification stands the creation of at least one domain and a belonging problem file. In order to facilitate the creation of these files and to keep track of their development, *myPDDL-new* creates a structured PDDL project folder, given a project name (Figure 4.4).

In this project folder, the domain file `domain.pddl` and the problem file `p01.pddl` (in folder `problems`) are filled with basic PDDL skeletons (TODO: remove this sentence or add functionality or even better: specify a template, which can be added!).

The `domains`, `dot` and `diagrams` folders are created for the use with *myPDDL-gen*, which will save its generated output to these folders and thereby allows for a basic version control system (see section 123).

As domain files usually have multiple problem files, the `problems` folder is designed for the collection of all associated problem files. \, that most knowledge engineers do not write any documentation related to the specified planning task (source: Shah, Chrupa), `README.md` is a Markdown file, which is, amongst others, intended for information about the author(s) of the project, contact information, informal domain and problem specifications, TODOs and licensing information.

The functionality of *myPDDL-new* is available through a command line interface, which allow for an integration of ST (and every other tool that

holds an interface for command line). New PDDL projects can be generated by invoking the following command:

```
$ java -jar path/to/my\textsc{pddl}.jar new NAME
```

This approach should support an structured and organized design process. The choice of a folder structure (instead of a project file) has the advantage of being readable and customizable by every editor. So the need for team work (source, shah KE) is tackled by using a structures project folder where and changes can be seen in the view of the diagram. + Git connection This directory organization is intended to contain a single or just a few domain files in one project, stored in the project root directory, while problem files are stores in the subfolder problems.

4.4 Syntax Highlighting

* Statement of Problem

Writing extensive domain and problem files is a cumbersome and time-consuming task Zhuo et al. [15]. Additionally, longer files can get quickly confusing. Therefore, it is convenient to have a tool that supports editing these files. Syntax highlighting, a common feature of text and code editors, describes the feature of displaying code in different styles (colors, fonts) according to the category of terms. In order to facilitate editing PDDL files, a syntax highlighting plug-in for the text and source code editor Sublime Text [13, 14] is proposed.

The process of writing PDDL files usually involves extending them and making continual amendments to them. SH provides code in a more readable way and can help to find and fix code errors quickly (see evaluation).

4.4.1 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 implemented by the use of the ST plug-in *AAAPackageDev* [1]. So, the definitions can be written in YAML in converted to Plist XML later on. *AAAPackageDev* [1] 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 [6], scopes are defined, that determine the meaning of the PDDL code block. ST themes highlight different parts of the code by the use of scopes. Scopes are defined by the use of regular expressions (regexes) in a tm-Language file. The scope naming conventions

mentioned in the *TextMate 2 Manual* are applied here. By the means of the name, the colors are assigned according to the current used ST theme. That means that colors are not assigned per se, but dependently on the current scheme. Through that, experienced users can use their default theme and all can easily change the colors by changing the scheme. 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 [7], Fox and Long [3], and McDermott et al. [10].

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 to the user's need.

```
file:///home/pold/Documents/BA/org-ba/thesis/img/coffee_errors_
no.pngp
```

4.4.2 Usage and Customization

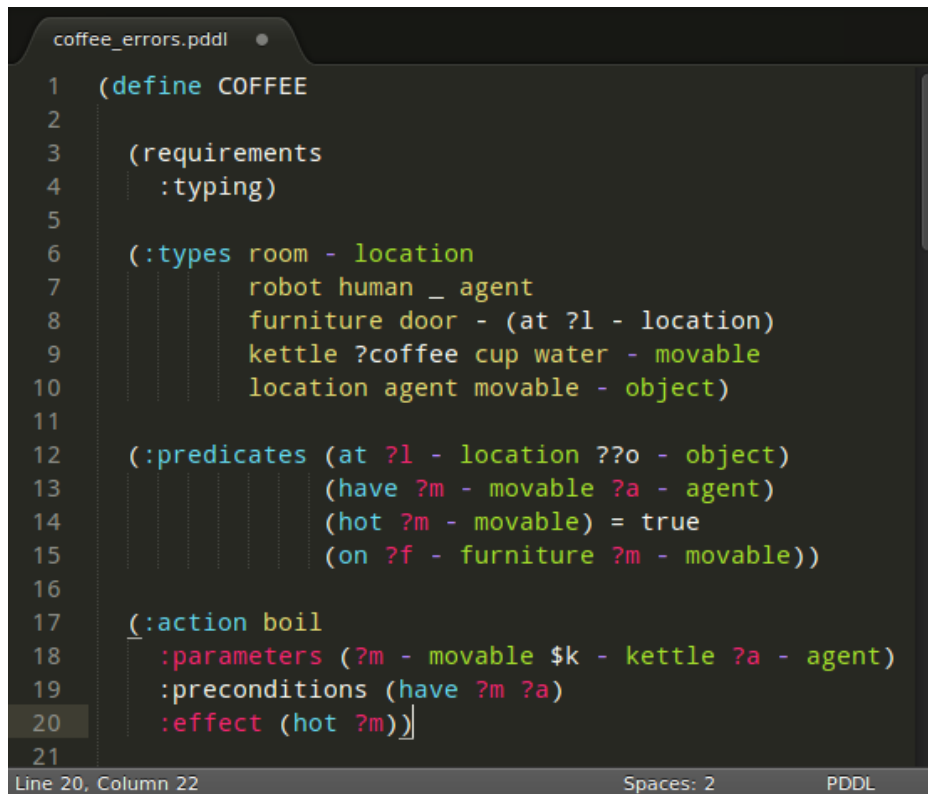
MYPDDL can be installed via Package Control or by placing the files of this repository (...) have to be placed in the ST packages folder (<http://www.sublimetext.com/docs/3/packages.html>). Following, the features can be activated by changing ST's syntax to PDDL (View->Syntax->\textsc{pddl}).

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.

1. Workflow Gary creates a new PDDL project using the command line, to this end he types



```
1 (define COFFEE
2
3   (requirements
4     :typing)
5
6   (:types room - location
7     robot human _ agent
8     furniture door - (at ?l - location)
9     kettle ?coffee cup water - movable
10    location agent movable - object)
11
12   (:predicates (at ?l - location ??o - object)
13     (have ?m - movable ?a - agent)
14     (hot ?m - movable) = true
15     (on ?f - furniture ?m - movable))
16
17   (:action boil
18     :parameters (?m - movable $k - kettle ?a - agent)
19     :preconditions (have ?m ?a)
20     :effect (hot ?m))
21
```

Line 20, Column 22 Spaces: 2 PDDL

Figure 4.2: Coffee domain with and without syntax highlighting

```
$ java -jar pddl.jar new hacker-world
```

changes into that directory

```
$ cd bulb-world
```

and renames the file domain.pddl to



```
$ mv domain.pddl garys-hacker-world.pddl
```

To get an overview over the world structure, Gary doodles a quick type diagram with the freely available graph editor and layout program yEd (yFiles software, Tübingen, Germany) that represents the world and its structure. Of course, he could also do this by pen and paper or using any other graph editor.

```
[./garysketch.svg]
```

He then opens this domain file in the Sublime Text 2 editor

```
$ sublime gary-hacker-world.pddl
```

and starts to model his world. To this end, he uses the code snippets `domain` for creating the domain skeleton, navigates inside the domain file with , creates new type definitions with the snippets `t2` and `t3`. After completing his first draft, he presses , for saving his file and displaying the PDDL type diagram and sees the following diagram:

[../hacker-world/diagrams/png-diagram3.png]

He recognizes, that he forgot to model that system software can be sub-divided into drivers and operating systems. Therefore he closes the diagram and adds the missing type declaration. He continues to write the PDDL domain and adds the required predicates with `p1` and `p2`, for example he types

The syntax highlighter shows Gary, if he uses incorrect PDDL syntax or if he forgets to close a parenthesis, as then parts don't get highlighted.

A final check show that everything is as expected:

[../hacker-world/diagrams/png-diagram3.png]

Gary knows, that the type diagram generator uses the Clojure interface. So, adding `#_` just before the predicates s-expression (that means `#_(:predicates ...)`) excludes the predicates from the type diagram, as this is the Clojure notation for commenting out s-expressions (and more convenient than commenting every single line). However, the `#_` construct is *not* correct PDDL, so Gary generates the diagram without the predicates, checks and sees that everything is fine, removes the `#_`, saves and closes the file.

The final version in the ST editor now looks like this: [./domain2.pdf]

In the command line, he now opens the PDDL problem file `p01.pddl`

```
$ sublime p01.pddl
```

and adds the problem skeleton by typing `problem` and pressing 

.

The relevant output lines of the output file are

The planner `SGPlan5` can be invoked by

```
$ ./sgplan -o garys-hacker-world.pddl \  
           -f p01.pddl \  
           -out plans/solution0.soln
```

where `-o` specifies the domain file, `-f` the problem file and `-out` the output file. The extension `.soln` for `solution0.soln` is used to show that solution files are not specified by PDDL per se, however, [3, p. 91] specifies plan syntax as a sequence of timed actions.

TODO: Possibly change planner to one that does not use time stamps.

0.001: (EAT-PIZZA BIG-PEPPERONI-PIZZA GARY) [1]

1.002: (EXPLOIT GARY MYSTERIOUS-TEX-MEX-MIX GISELA) [1]

Gary now definitely knows, that he first has to eat the pepperoni pizza, before he can exploit Gisela's application *MysteriousTexMexMix*.

The numbers to the left of the actions (0.001, 1.002) and to the right (both [1]) specify the start time and the duration of the actions, respectively. They are dispensable in this case, as only the sequence of actions is relevant.

The generated files (`dot-diagram[0-2].dot`, `png-diagram[0-2].png`, `garys-hacker-world[0-2].pddl`) are the revision control versions, generated each time the Clojure script is invoked (by pressing F8).

It can probably be seen, that this rather short description of the world and in problem results in rather extensive PDDL files.

4.5 Code Snippets (*myPDDL-snp*)

While writing and extending pddl files, knowledge engineers are supposed to use the same constructs many times. To facilitate and fasten the implementation of standard constructs, my-PDDL-snp provides code snippets. These snippets are templates for often used pddl constructs, like domain and problem definitions, predicates and actions. They can be inserted by typing a trigger keyword. The inserted content contains fields with placeholders, that can be accessed and filled in consecutively. PDDL constructs with a specified arity can be inserted by adding the arity number to the trigger keyword.



Figure 4.3: Example for the use of snippets. `=p2=` creates a binary predicate template that can be filled in.

And gets (has ?s - software ?p - person) and action for the action definition.

Every snippet is stored in a separate file, located in the PDDL/ folder. New snippets can be added and existing snippets can be customized there.

```
(:action actionName
  :parameters (?x - <objectType>)
  :precondition (<conditions>)
  :effect (<effects>))
```

4.6 Distance Calculation for PDDL Locations (myPDDL-loc)

While one might assume that , However, PDDL does only support basic arithmetic operations (+, -, /, *). A planning problem In temporal domains, it could be desirable to One might assume that the Euclidean distance could be modeled using `sqrt`

myPDDL-loc uses the PDDL-Clojure interface and reads a problem file and extracts all locations, defined in the `:init` part. In Clojure, the Euclidean distances between all locations are calculated and then written back to an extended problem file.

The calculator works on any dimension, so that locations can be specified both two dimensionally and three dimensionally (or n-dimensionally).

```
...
(:init (location home-gary 7 3)
      (location home-gisela 10 5))
...
```

Listing 9: Before

```
(:init
 (location home-gary 7 3)
 (location home-gisela 10 5)
 (distance home-gary home-gary 0.0)
 (distance home-gary home-gisela 3.6056)
 (distance home-gisela home-gary 3.6056)
 (distance home-gisela home-gisela 0.0))
```

Listing 10: After

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 [11] 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.

Another alternative is to make use of an external helper and, instead of calculating every entry of the distance matrix, calculate 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 4950 distance entries in the problem file.

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).

Inspect problem file and calculate distances while planning calculating.

4.7 Type Diagram Generator (*myPDDL-gen*)

As stated by the adage "A picture is worth a thousand words" graphical representations can have some advantages compared to textual representations. In computer science, they should simplify the communication between developers and help to quickly grasp the connection of related system units (source!). graphical representations are not always superior to textual representations (see introduction for a short discussion on this topic), both text and graphics can complement each other and facilitate the understanding of complex problems. To support this theory, a user test was performed, showing that ...)

The extended expressive power provided by ADL includes the ability to express a type hierarchy in the domain and a object hierarchy in the problem file.

Assuming that `:typing` or `:adl` is declared, object types play a major role in the PDDL design process: they constrain the types of arguments to predicates and determine the types of actions. So, a fine grasp of their hierarchy, as well as their involved predicates becomes handy and assists knowledge engineers in the planning process. Furthermore, in order to understand, use and extend available domains, a crucial part is the grasping of types, their hierarchy, and the predicates they that make use of them. 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` (TODO: possibly move to basics part).

myPDDL-gen uses `get-\textsc{pddl}-part(file,types)`, declared in *myPDDL-i/f* for extracting the textual type hierarchy declared in a PDDL file. These extracted types get separated in are then separated in subtypes and supertypes, using regular expressions (regex).

PDDL side	Clojure side
(:types ... — ...)	

1. Visualization

The visualization is generated using `dot` from the GraphViz package, a collection of programs for drawing graphs. `dot` is a scriptable, graphing tool, that is able to generate hierarchical drawing of directed graphs in a variety of output format (png, pdf, ...), from specific text files, written in the DOT language.

From this representation, the description of a directed graph (**digraph**) in the dot language is created and saved in the folder `dot/`. This file is then passed to the command line program `dot` and a PNG graphic is created in the folder `diagrams/` and immediately opened and displayed in a window. In addition, a copy of the domain file is stored in the folder `domains/`. Every time *myPDDL-gen* is invoked, these steps are executed and the saved file names are extended by a ascending revision number. This way, one cannot only identify associated pddl, dot and png files, but also use this feature for basic revision control. The structure and revision number of a previous version can be identified by the png type diagram and then, one can revert to a previous revision, stored in the `domains/` folder. All folders are created if necessary.

Figure xyz displays a type diagram generated from the **Gary's Hacker World** domain. In the diagram, types are represented with boxes, whereby every box consists of two parts:

- The header displays the name of the type.
- The lower part displays all predicates that use the corresponding type at least once in their arguments. The predicates are written in the same way, as they appear in the PDDL code.

Generalization relationships ("is a", for example "a driver *is a* type of software") are expressed by arrows from the specialization (the subtype,

```

garys-hacker-world.pddl
├── dot
│   ├── dot-diagram0.dot
│   └── dot-diagram1.dot
├── diagrams
│   ├── png-diagram0.png
│   └── png-diagram1.png
└── domains
    ├── garys-hacker-world0.pddl
    └── garys-hacker-world1.pddl

```

Figure 4.4: Folder structure after two invocations of `textitmyPDDL-gen`.

here: driver) to the generalization (the super type - here: software), where the arrow head aims at the super type. This relationship expresses, that every subtype is also an instance of the illustrated super type.

4.7.1 Limitations

myPDDL-gen does not display predicates without argument (nullary or 0-ary predicates), like `(is-rainy)`, as they have no assigned type. Furthermore, it does not support predicates defined by `(either ...)` and types that have to super type.

4.8 Syntax Highlighting and Code Snippets (myPDDL-sub)

While *snp* and *syn* are devised explicitly for ST and therefore integrated from the outset, the other tools (*new*, *gen*, *loc*) can be used independently of ST utilizing the command line interface and any PDDL file. To provide a central interface for using *myPDDL*, *-sub* integrates *new*, *gen* and *loc*, aiming at a user-friendly execution and use of the system.

The three tools can be invoked using the ST command palette (`ctrl` + `⬆` + `P`), and then choosing one of the PDDL menu entries:

PDDL: Create Project for myPDDL-new *PDDL: Create Project* requires the user to specify a project name in the then displayed input panel.

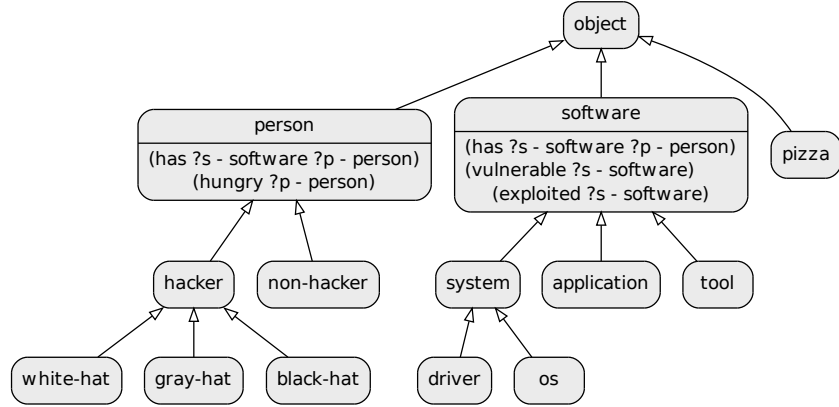


Figure 4.5: The type diagram that was generated from `garys-hacker-world.pddl` using `myPDDL-gen`.

PDDL: Calculate Distances for `myPDDL-loc` Saves and

PDDL: Display Diagram for `myPDDL-dia`

Extending a available editor. Furthermore, ST was used as it provides a framework for general code editing. Features include code folding,

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 \LaTeX documents.

Chapter 5

Analysis

5.1 Design Goals

	PDDL Studio	itSIMPLE	P
latest supported version	PDDL 1.2	PDDL 3.1	P
syntax highlighting	Yes	Yes	Y
syntactic error detection	Yes	No	N
semantic error detection	Yes	No	N
code completion	Yes	No	Y
code snippets	No	Yes, but rather basic and not customizable	Y
code folding	Yes	No	Y
project management	Yes	Yes	N
visualization feature	No	Yes	N
planner integration	Basic	Yes	N
automatic indentation	No	No	Y
customization features	No	No	ex

5.2 Empirical Study

A key challenge of creating a sophisticated syntax highlighter without the availability of a lexical parser, is the use of regular expressions for creating a preferably complete PDDL identification. While this is not possible by the expressiveness of regexes, this syntax highlighter tries to come as close as possible.

The consistency and capability to highlight every PDDL construct in a color according to its meaning, were checked by 320 (syntax error-free)

PDDL files, consisting of 87 domain and 230 problem files (list of files). In that, no inconsistencies nor non-highlighted words could be found.

While syntax highlighting can improve the time and ability to get along in code files, it is mainly intended to distinct language structures and syntax errors.

5.3 User Study

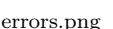
5.3.1 Participants

Eight non-paid students (two female, $\text{Mean}_{\text{age}}=23$, $\text{SD}_{\text{age}}=2$) took part in the experiment. All had knowledge about at least one LISP dialect, and therefore about program code written as parenthesized lists, but nobody had faced PDDL or any other planning language prior to this study. Furthermore, nobody has used Sublime Text before that test.

5.3.2 Material

The usability of myPDDL-syn (Syntax Highlighter, see 4.4) and myPDDL-gen (Type Diagram Generator, see **Type Diagram Generator**) were tested. For this purpose, two domains (*Planet Splisus*, *Store*) with fantasy type names were created. Participants were asked to answer five questions that required to understand the PDDL type hierarchy. Subjects were asked to work on questions, while time on task (per question) was measured without subjects' knowledge, by asking the S to say out loud the regarding answer.

Furthermore, two deliberately incorrect domain files were provided to the S, each containing 17 errors in total (consisting of X semantic errors and Y syntax errors). Participants were asked to detect as many errors as possible in six minutes and immediately correct found errors in the code (as this could change the syntax highlighting of other code parts) and write down the line and a description or the correction of the error on a sheet of paper for an easy identification in the analysis of test results.

TODO: Include 

5.3.3 Design

S	Order			
A	<i>Planet Splisus</i>	<i>Logistics</i>	Store	Coffee
B	Store	Coffee	<i>Planet Splisus</i>	<i>Logistics</i>
C	Planet Splisus	Logistics	<i>Store</i>	<i>Coffee</i>
D	<i>Store</i>	<i>Coffee</i>	Planet Splisus	Logistics
E	<i>Logistics</i>	<i>Planet Splisus</i>	Coffee	Store
F	Coffee	Store	<i>Logistics</i>	<i>Planet Splisus</i>
G	Logistics	Planet Splisus	<i>Coffee</i>	<i>Store</i>
H	<i>Coffee</i>	<i>Store</i>	Logistics	Planet Splisus

Italic: Tools part

5.3.4 Procedure

At the earliest, 24 hours ahead testing date, participants received a link ¹ to a 30-minute video tutorial and were asked to watch this video before the test, if possible. This tutorial comprised a general introduction to planning and a more specific introduction to PDDL's domain syntax. In the video, participants were also asked to fulfill tasks regarding PDDL and check their answers with the provided solutions in the video.

At testing date, participants were asked to sign a consent form and to take a seat in front of a Laptop with a 13" display and a connected monitor with a 17" display. If they did not already watch the PDDL tutorial the participants first were asked to watch the tutorial then. After that, any open questions regarding PDDL and the general testing procedure were clarified.

All participants were provided with a one page summary of PDDL domain syntax (*cheat sheet*) that they could always refer to. Furthermore, they were allowed to take any hand-written notes that they took during the video tutorial. (and to rewatch the video tutorial at any time).

Participants were then tested, according to a assigned order of tasks.

The participants did not and that there will be a *tools* part. Immediately before the tools part, a three minute video introduction to the functionality of the syntax highlighter (myPDDL-syn) and the usage of

¹<http://www.youtube.com/playlist?list=PL3CZzLUZuiIMWEfJxy-G60xYVzUrvjwuV>

(myPDDL-gen) was given. Directly after his, participants were asked to work on the tools parts. so that they faced the tools were not confronted with the tools before the actual test.

5.3.5 Results

Task	Time	Points
1		
2		
3		
4		
5		
Sum		

Table 5.1: Planet Splisus **Aggregated processing time of tasks with correct answers**

The questionnaire used The mean System Usability Scale (SUS) score was XX, arguing for a high usability.

Chapter 6

General Discussion

As seen in the conducted study, missing actions in the type diagram can confuse. So, it is possibly helpful to exclude predicates in the diagram and only display the plain type hierarchy (as all participants were faster) before actions have not been added. Nevertheless, it is worth noting that only PDDL novices were tested, after watching a introduction video, without ever writing a domain by scratch.

Very likely, a learning effect will occur, so that tasks are more easily to fulfill if they are done for the second time.

Chapter 7

Outlook and Conclusion

7.1 Future Work

The PDDL/Clojure interface provides a basis for dynamic and interactive planning scenarios. So, time-dependent knowledge could be modeled by adding facts (learning) to and retracting facts (forget) from facts a PDDL file.

Besides that, interactive planning

Furthermore the interface can be used to extract knowledge, specified in another formal language and create PDDL files from them.

Knowledge engineers can customize and extend Sublime Text as Although, myPDDL is concentrated on Sublime Text, users could transfer the ideas to other text editors.

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 semantic errors besides syntactic errors `\{ plch2012inspect }` 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.

In the diagram, predicates are only added to the types that are explicitly mentioned in the argument of the predicate. However, as subtypes of types declared in the predicate arguments, can also be used as argument to the predicate, this means, that all specializations of a type

can also be used for this predicate. This can be seen in Figure xyz ...:For example, a the PDDL domain file could declare `(hungry ?p - person)`, although men and women can be hungry.

7.2 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.

7.3 Conclusion

MYPDDL - Modular Auxiliary for the Planning Domain Definition Language, has been designed to support knowledge engineers in modeling planning tasks as well as in understanding, modifying, extending and using existing planning domains.

MYPDDL has been implemented as an interface between Clojure and PDDL, where PDDL editing features are fulfilled in the text editor Sublime Text. It is designed as an modular architecture, which is extensible, customizable and easy usable system. myPDDL-gen can visualize any PDDL domain, without making semantic assumptions and n-ary predicates.

Implemented features comprise code editing features, namely syntax highlighting and code snippets, a type diagram generator and a distance calculator,

The user study shows some initial evidence that the syntax highlighting feature (MYPDDL-SUB) and the type diagram generator(MYPDDL-GEN) can support knowledge engineers in the design and analysis process, in particular in error detection and in keeping track of the domain structure, the type hierarchy and grasping predicates using these types.

A faster understanding of the domain structure could be beneficial for the maintenance and application of existing domains and problems, and, possibly for the communication between engineers. Finally, real

world usage of PDDL can be promoted so that the focus of artificial intelligence planning can also be shifted towards the design of plans, following the citation "Plans are worthless, but planning is everything".

Bibliography

Paper Sources

- [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] Rich Hickey. “The clojure programming language”. In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM. 2008.
- [5] Chih-Wei Hsu and Benjamin W Wah. “The sgplan planning system in ipc-6”. In: *Proceedings of IPC*. 2008.
- [7] DL Kovacs. “BNF definition of PDDL 3.1”. In: *Unpublished manuscript from the IPC-2011 website* (2011).
- [8] 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.
- [9] MacroMates Ltd. *TextMate 2 Manual*. 2013. URL: <http://manual.macromates.com/en/> (visited on 02/24/2014).
- [10] Drew McDermott et al. “PDDL-the planning domain definition language”. In: (1998).
- [11] 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.

- [12] Tomas Plch et al. “Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio”. In: *and Exhibits* (2012), p. 15.
- [13] Jon Skinner. *Sublime Text 2*. <http://www.sublimetext.com/2>. Release 2.0.2. 2013.
- [14] Jon Skinner. *Sublime Text 3*. <http://www.sublimetext.com/3>. Beta Build 3059. 2013.
- [15] Hankz Hankui Zhuo et al. “Learning complex action models with quantifiers and logical implications”. In: *Artificial Intelligence* 174.18 (2010), pp. 1540–1569.

Website Sources

- [6] 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).

Chapter 8

Appendix

This code can also be found on the enclosed CD, and on the Internet page <https://github.com/pold87/sublime-pddl> (most recent version).

The website <http://pold87.github.io/sublime-pddl/> is the accompanying website for this project.

```
(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 \textsc{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 \textsc{pddl} file from a list of objects and locations"
  [objs-file objs-type]
  (str
   "(define (domain domainName)"

   (:requirements
    ; domain-specific actions
```



```

# [PackageDev] target_format: plist, ext: tmLanguage
---
name: \textsc{pddl}
scopeName: text.pddl
fileTypes: [pddl]
uuid: 2aef09fc-d29e-4efd-bf1a-974598feb7a9

patterns:

#####
### Customization ###

- include: '#domain'
- include: '#problem'
- include: '#comment'

#####
### Repository ###

repository:

#####
### General specifications ###
#####

built-in-var:
  match: \?duration
  name: variable.language.pddl

variable:
  match: '(?:~|\s+)(\?[a-zA-Z](?:\w|-|_)*)'
  # name: variable.other.pddl
  name: keyword.other.pddl # TODO: changeback again to variable.other.pddl
  # this is just a dirty hack for highlighting

pddl-expr:
  match: '(?:~|\s+)([a-zA-Z](?:\w|-|_)*)(?!:|\?)\b'
  captures:
    '1': {name: string.unquoted.pddl}
  #name: string.unquoted.pddl

comment:
  comment: "Comments beginning with ';'."
  name: comment.line.semicolon.pddl
  match: ;.*

number:
  name: constant.numeric.pddl
  match: \b((0(x|X)[0-9a-fA-F]*)|(([0-9]+\.[0-9]*)|(\.[0-9]+))((e|E)(\+|-)?[0-9-]))

keyword:
  name: storage.type.pddl # TODO: UPDATE

```

