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

Volker Strobel

April 11, 2014

Contents

Abstract

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 task specifications. With myPDDL, structured PDDL projects can be created and edited using code templates and syntax highlighting. One tool visualizes the type hierarchy in PDDL domains, allowing knowledge engineers to understand the representation structure at a glance and to also keep track of developments with a basic revision control system. Another tool allows calculating distances between objects specified by predicates in a problem file. These tools make use of an interface that provides a general way for reading and writing PDDL specifications by using the programming language Clojure, and thus presenting a means of bypassing PDDL's limited modeling capacity. They are made accessible in the customizable editor Sublime Text. A small user study, conducted with eight inexperienced PDDL users, shows some initial evidence that the syntax highlighting feature and the automated creation of type diagrams could support knowledge engineers in the design and analysis process. The users detected 35 % 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 was reduced by 52 %.

Chapter 1

Introduction

Have you ever struggled to find the optimal sequence of actions for a recurrent problem? While a task like this could take you hours, weeks or even a lifetime to complete, a planning software could possibly get the job done within milliseconds. Being a key aspect of artificial intelligence, planning is concerned with devising a plan, i.e. a sequence of actions, to achieve a desired goal [20]. It can be both a tool to create automated systems and a means to support and understand human behavior [27]. However, the effectiveness of planning largely depends on the quality of the problem formalization [44, 58]. To ensure a standardized modeling format, the Planning Domain Definition Language (PDDL) [32] was developed and has become the de facto standard for the description of planning tasks [25]. The discipline that deals with the integration of world information into a computer system via a human expert is called knowledge engineering [11]. While automated planning could save you a vast amount of time if everything works as intended, creating the planning task specifications is a complex task that can be error-prone and cumbersome. Searching for an error in a long PDDL file can be like looking for a needle in a haystack; debugging could take hours as some planners crash on erroneous PDDL files without providing any problem report.

The strenuous modeling process and the efficiency issues it brings gave rise to the need for tools designed for creating planning task specifications [44], so much so that the annual International Conference on Automated Planning and Scheduling [1] has workshops (such as Knowledge Engineering for Planning and Scheduling), tutorials and presentations dedicated to this topic. Therefore, developing such tools for PDDL and hence facilitating the knowledge engineering process is appreciated and worthwhile.

The modular toolkit myPDDL (*m*odeling efficientl*y* PDDL was developed,

[1]More information on ICAPS 2014, the 24th International Conference on Automated Planning and Scheduling can be found on `http://icaps14.icaps-conference.org/`.

within the scope of this thesis, in order to tackle frequent needs of knowledge engineers, like project management, efficient development, error-detection, team collaboration and to increase the acceptance and usage of PDDL in real-world domains [44, 58].

myPDDL is intended to support knowledge engineers throughout the entire design cycle of specifying planning tasks. In the initial stages, it allows for the creation of structured PDDL projects that should encourage a disciplined design process. With the help of snippets, i.e. code templates, often used constructs can be inserted in PDDL files. A syntax highlighting feature that speeds up the error-detection can come in handy in intermediate stages. Understanding the textual representation of complex type hierarchies in domain files can be confusing, so an additional feature enables their visualization. PDDL's limited modeling capabilities were bypassed by developing an interface that converts PDDL code into Clojure [21] code and vice versa. Within this project, the interface was employed for a feature that calculates distances between objects specified in a problem model, but the interface provides numerous other possibilities and could also be used to further automate the modeling process. All of these features were integrated into the customizable and extensible Sublime Text [47] editor. Since the main aim in the development of the toolkit was for it to be easy to use and maintain, it is evaluated with regard to these criteria. The usability was assessed by means of a user test with eight subjects that had no prior experience with artificial intelligence planning. The results indicate that both error-detection and the understanding of a given domain can be facilitated by myPDDL.

To lay the foundations for this thesis, a basic introduction to and design principles for PDDL will be given in Chapter 2. Existing tools will be critically reviewed in Chapter 3 to set the goals for myPDDL. The features and modules of myPDDL are then described in detail in Chapter 4. Details and results of the usability study are provided in Chapter 5. Finally, the implications are briefly discussed before an outlook for future research and developments in the field concludes this thesis in Chapter 6.

Chapter 2

Background and Basics

The human brain is an astonishing structure that allows us to get by in a highly complex world and give 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 tries to narrow this gap [2]. For this purpose, constructs are needed that can represent the information about the world and the problem. In automated planning, this is usually done [13] using a planning language, like PDDL.

To illustrate the usage and basics of PDDL, the remainder of this section presents a modeling walkthrough using a fictional example.

Consider the following world that is to be integrated into a computer system using PDDL:

> *Hacker World*
> If hackers are hungry, they have to eat some pizza in order to be able to work, that is to exploit (or hack into) 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, software can be exploited or not.

Actions: Hackers can exploit software and they can eat pizza.

This description of a world can be specified in PDDL using a domain file. The domain file can be compared to a stage setting, providing the framework for a specific problem scenario by way of general, abstract constructs and conditions.

In the world of hackers and pizzas, such a domain specific problem could be:

> *Gary's Huge Problem*
> Gary is a hungry hacker who should somehow exploit the vulnerable software MagicFailureApp. Some pepperoni pizza is lying around.

Again, several constructs can be identified:

Objects The hacker Gary (in PDDL all entities are objects, including persons), the pepperoni pizza, the software.

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

Goal state The MagicFailureApp is exploited.

Assume that Gary wants the help of an automated planning system to plan the sequence of required actions (*Who has to eat pizza?*, *What should be hacked?* and *In what order should these things be done?*), leading from the initial state to the goal state. These specifications must be formalized so that a planner can utilize them. In PDDL, this is done in problem files. In the end, Gary will be able to feed the domain and the problem file into a planner which will generate a sequence of actions that Gary can take to solve his problem.

Summing up, PDDL planning tasks specifications 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. Specific values are assigned to the templates provided by the domain file (instantiation).

This separation allows for a powerful task modeling process: while general world information is described in the domain file, specific instances of problems are created in the problem files. This means that one abstract model of a world can be used for solving many problem instances.

The rest of this section is to propose general design guidelines for, and give an introduction to PDDL [1], to serve as a basis for the rest of this thesis. To this end, the syntax of common constructs of domain and problem files is further investigated in a step-by-step approach, continuing with the above described example.

2.1 Analysis

How do you begin to model a planning task? The first, and possibly most significant step to integrate information into a computer system, is gaining an *understanding* of the problem [40, 20]. For modeling in PDDL, the following six general design principles [2] ought to lead to a thorough, stepwise, and iterative modeling process:

Analysis: Every task specification should begin with 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.

Type diagram: Based on the preceding analysis, the relationship of the identified categories or types is represented, using a diagram. This can be done on paper or with the help of a graph editor.

Domain definition: In this step, the 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. The initial and goal states are modeled using the predicates declared in the domain file.

Planning: Now, one can provide the domain and problem definition to a planner. The planner then generates a plan, i.e. a sequence of actions that leads to the goal state.

Plan analysis: Finally, the generated plan needs to be inspected. If any design mistakes or inconistencies are detected, it is advisable to restart at an earlier design step.

[1] More complete descriptions of PDDL, as well as formulations in Backus-Naur form (BNF) are provided by Fox and Long [13] for PDDL 2.2 and Kovacs [28] for PDDL 3.1.

[2] These design principles are based on *How to Design Classes (Draft)* that describes four steps of designing class hierarchies for object-oriented programs.

The following two sections deal with the creation of a domain and problem definition. At the end of the introduction of each construct, the corresponding code block of the *Hacker World* and *Gary's Huge Problem* is given.

2.2 Domain File

The domain file sets the framework for planning tasks. It models the world in which the problem occurs and hence determines which types and predicates are available and which actions are possible.

2.2.1 Domain Definition

We begin with the definition of the domain file. Every domain file starts with `(define (domain DNAME) ...)`, where `DNAME` specifies the name of the domain. A semicolon (`;`) declares the rest of the line as comment.

```
; Hacker World - A realistic example
(define (domain hacker-world)
```

Listing 1: The domain definition of the *Hacker World*

2.2.2 Requirements

PDDL is composed of feature subsets [32]. As most planners only support some of these subsets, the requirements block is useful for a planner to determine if it can act on a given problem. While basic specifications are used by default [32], further requirements have to be stated explicitly. For example, one requirement used by many planning domains [57] is:

:typing Enables the typification of variables (see 2.2.3 Types below), so that it is mandatory for variables to be of a particular type.

Besides `:typing`, the *Hacker World* will use a further requirement:

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

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

Listing 2: The requirements that are necessary to model the *Hacker World*

2.2.3 Types

Often in the real-world, there will be individual objects of the same kind or type. There may be many different desks, but all share common properties, like having a flat upper surface, and all are pieces of furniture.

PDDL allows for declaring types and thereby structuring the domain in the `(:types ...)` block. Relations can be expressed with a type hierarchy, in which any type can be a subtype of yet another type. Typed lists are used to assign types to variables. Parameters in actions, as well as arguments in predicates can be typed in this manner. Later, in the problem file, objects are assigned to types. Types are declared using 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`, and every defined type, in turn, is a subtype of `object`.

```
(:types hacker non-hacker - person
        desk chair - furniture
        laptop workstation - computer
        pizza burgers fries - food
        pepperoni supreme - pizza
        person furniture compiter food software - object)
```

Listing 3: The type hierarchy for the *Hacker World*, consisting of different types of persons, furniture, computers, hackers, food, pizza and software. The elements on the left-hand side (for example `hacker non-hacker`) are declared subtypes of the right-hand side (`person`) whereby the type hierarchy is expressed.

2.2.4 Predicates

How can we describe properties of objects and states of the world? Predicates are templates to represent logical facts and can be either true or false. In the `:(predicates ...)` block, predicate names and the number of arguments together with the corresponding types are declared. The general syntax for a predicate is `(pname ?v1 - t1 ?v2 - t2 ...)`, where `?` followed by a name (`v1`, `v2`) declares a variable, and the expression (`t1`, `t2`) following the hyphen (`-`) states the type of this variable. All the types that are used must be declared in the typing section first. The number of variables (or arguments) determines the arity of a predicate ranging from zero (nullary 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, meaning that `(p ?v1 - t ?v2 - t)` is equivalent to `(p ?v1 ?v2 - t)`.

```
(:predicates (has ?s - software ?p - person)
             (hungry ?p - person)
             (vulnerable ?s - software)
             (exploited ?s - software)
             (distance ?o1 ?o2 - object ?x - number)
             (location ?o - object ?x ?y - number))
```

Listing 4: This section declares five predicates: the unary predicates `hungry`, `vulnerable` and `exploited`, the binary predicate `has`, and the 3-ary predicates `location` that specifies x and y coordinates for objects, `distance` that expresses the distance between objects.

2.2.5 Actions

Now that we have predicates for describing world states, we still need a means for changing their value. This is done with action. Actions are operators in PDDL, because they can change properties of objects by changing predicate values, so that problems can be solved. 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 A combination of predicates, all of which must be true before an action can be executed. Therefore, this part describes the applicability of an action.

:effect Specifies the new values of the declared predicates, once the action has been completed. Therefore, it describes the post-condition of an action.

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

;; Move a piece of furniture
(:action move
  :parameters (?f - furniture ?old-x ?old-y ?new-x ?new-y)
  :effect (and (location ?f ?new-x ?new-y)
               (not (location ?f ?old-x ?old-y)))))
```

Listing 5: Three actions that can change logical values in the *Hacker World*. It is important to remember that predicate values keep being true if an effect adds a logical fact. This is often not desired. Consider the action move, that changes the location of a chair. Only having the effect `(location chair ?new-x ?new-y)` would result in the chair being located at two locations, at `?old-x` and `?old-y` and the new, specified coordinates. Therefore, the old coordinates have to be deleted, using `(not ...)`. The `:precondition` part can be omitted, if an action should be always applicable.

2.3 Problem File

A planning problem consists of a domain and a corresponding problem file. Within problem files, concrete objects are created (instantiated). Furthermore, the initial world state and the desired goal state that is to be reached are declared.

2.3.1 Problem Definition

Analogous to the domain definition, problem files are initiated with `(define (problem PNAME) ...)`, where `PNAME` declares the name of the problem.

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

Listing 6: Initiating the problem file with the name garys-huge-problem

2.3.2 Associated Domain

Problems occur in worlds. Therefore, problem files are designed with regard to domain files that need to be referenced at this point in the problem file. This means that `DNAME` in `(:domain DNAME)` and `DNAME` in `(define (domain DNAME) ...)` in the corresponding domain file have to be identical.

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

Listing 7: The domain "hacker-world" is the corresponding domain name to the problem garys-huge-problem

2.3.3 Objects

Since types are only empty shells, they need to be instantiated. This is done in the `(:objects ...)` block. Instantiating types means that concrete objects are assigned to the types.

```
(:objects big-pepperoni - pepperoni
          gary - hacker
          gisela - non-hacker
          magicfailureapp - software)
```

Listing 8: This part assigns concrete objects to the type templates. In this case, magicfailureapp - software means that the object magicfailureapp is of the type application.

2.3.4 Init

The `(:init ...)` block models the initial state of the world with a list of instantiated predicates that are declared as true. All other, non-specified predicates are assumed to be false. This is called the *closed-world assumption* [3].

```
(:init (hungry gary)
       (has big-pepperoni gary)
       (vulnerable magicfailureapp)
       (has magicfailureapp gisela))
```

Listing 9: The initial situation in Gary's Huge Problem consists of the hungry hacker Gary and the vulnerable application MagicFailureApp that belongs to Gisela.

2.3.5 Goal

The goal state is described by the logical fact that is desirable and should be reached with the execution of the plan. In PDDL, several goals are combined with `(and ...)`. All unspecified predicates are irrelevant, meaning that they can be either true or false in the goal state.

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

Listing 10: In the end, the software magicfailureapp should be exploited.

2.4 Planning

Finally, the effort of the formalization of the planning task will be rewarded with the automatic generation of a plan. There is a broad range of available planners [4]. However, most planners only support certain subsets of PDDL and have some peculiarities [5]. Additionally, the quality of error messages is very diverse, ranging from stating that an error occurred to displaying line number and found problem.

[3] By specifying `:open-world` in the requirements part, PDDL is also capable of using the open world assumption, where non-specified predicates can be both, true or false.

[4] For an overview of planners that participated in the 2011 International Planning Competition and their features, see `http://www.plg.inf.uc3m.es/ipc2011-deterministic/ParticipatingPlanners.html`.

[5] A short discussion on planners and their "excentricities" can be found at `http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html`.

This thesis uses the planner SGPlan$_5$ [22], a planner that supports many PDDL features and has comprehensive error messages that state the actual problem [6].

The planner SGPlan$_5$ can be used by specifying the domain file and the problem file in a command line interface.

The relevant output lines for *Gary's Huge Problem*, specified in the *Hacker World* look as follows:

```
0.001: (EAT-PIZZA BIG-PEPPERONI-PIZZA GARY) [1]
1.002: (EXPLOIT GARY MAGICFAILUREAPP GISELA) [1]
```

The numbers to the left (`0.001`, `1.002`) and to the right (`[1]`) specify the start time and the duration of the actions, respectively. Since we did not use any timestamps, they are dispensable in this case, because only the order of actions is relevant.

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

Since specifying PDDL files can be time-consuming the next chapter will compare knowledge engineering tools that support the user in effectively eneffective planning model.

[6] Additionally, SGPlan$_5$ was the winner of the 1st Prize Satisficing Planning in the Deterministic Part of the International Planning Competition (IPC) in 2006 `http://zeus.ing.unibs.it/ipc-5/results.html`.

Chapter 3

Related Work

This chapter is to introduce knowledge engineering tools that allow editing PDDL files in a textual environment to some extent. All tools provide features to support the user in writing correct PDDL code more efficiently. After introducing the tools, they are compared and their shortcomings are discussed to set the stage for myPDDL.

3.1 PDDL Studio

PDDL STUDIO [7, 39] is an application for creating and managing PDDL projects, i.e. a collection of PDDL files. The PDDL STUDIO integrated development environment (IDE) was inspired by Microsoft Visual Studio [34] and imperative programming paradigms. Its main features are syntax highlighting, error detection, context sensitive code completion, code folding, project management, and planner integration. Many of these features are based on a parser, which continuously analyzes the code and divides it into syntactic elements. These elements and the way in which they relate to each other can then be identified. The syntax highlighter is a tool that colors constructs according to their syntactical meaning within the code. In the case of PDDL STUDIO, it colors names, variables, errors, keywords, predicates, types and brackets each in a different customizable color. PDDL STUDIO's error detection can recognize both syntax errors (missing keywords, parentheses, etc.) and semantic errors (wrong type of predicate parameters, misspelled predicates, etc.). Since semantic errors can be of an interfile nature, meaning that there is a mismatch between domain and problem file, PDDL STUDIO can detect such errors in real time. The code completion feature allows for the selection of completion suggestions for standard PDDL constructs as well as for terms that have been used before within this file or other files in the same project. Code folding allows the knowledge engineer to hide certain code units or blocks that are currently not needed. Only the first line of the block is then displayed. PDDL STUDIO's code folding feature works on the basis of syntax. This means that it can tell different code blocks apart with the help of the parser and is thus able to fold the code accordingly. All these above mentioned features of PDDL STUDIO utilize the parser. Another important feature of the PDDL STUDIO project is a project manager. This keeps track of all files, displays them in a tree structure, saves them upon compilation and is also necessary for the interfile error detection and code completion functionalities. Lastly, a command-line interface allows the integration of planners in order to run and compare different planning software.

3.2 itSIMPLE

Unlike PDDL STUDIO, which provides a text based editor for PDDL, the ITSIMPLE [52] editor has, as its main feature, a graphical approach that allows for designing planning tasks in an object-oriented approach using Unified Modeling Language (UML) [5] diagrams. UML was invented in order to standardize modeling in software engineering (SE) and the latest version (UML 2.4.1) [51] consists of 14 different types of diagrams divided into two larger groups: structure and behavior diagrams. In the process leading up to ITSIMPLE, UML.P (UML in a Planning Approach) was proposed, a UML variant specifically designed for modeling planning domains and problems [53].

This variant specifies:

- Class Diagrams for static domain features
- Object Diagrams to describe the initial and the goal state in problem specifications
- StateChart Diagrams to represent dynamic characteristics such as actions in domain specifications.

Thus, ITSIMPLE uses both UML structure diagrams (Class and Object Diagrams) and UML behavior diagrams (StateChart Diagrams). The main purpose of ITSIMPLE is supporting knowledge engineers in the initial stages of the design phase by making tools available that help with the transition from the informality of the real world to the formal specifications of domain models. The professed aim of the project is to provide a means to a "{"disciplined process of elicitation, organization and analysis of requirements} [52]. However, subsequent design stages are also supported. Once domain and problem models have been created, PDDL representations can be generated from the UML.P diagrams, edited, and then used as input to a number of different integrated planning systems. Therefore, one of the tools already introduced within the scope of PDDL STUDIO, planner integration, is also implemented. However, unlike in PDDL Studio, ITSIMPLE has a more user-friendly approach to planner integration: domain and problem can be fed to the planner with the press of a button, while in PDDL Studio, the user has to know and input commands in a command-line interface.

Not only is it possible to directly input the domains and problems into a planner, another tool can inspect the output from the planning system using the built-in plan analysis. This consists of a plan visualization that shows the interaction between the plan and the domain by highlighting every change caused by an action. 's modeling 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 [55] 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. A minimal syntax highlighting feature recognizes PDDL keywords, variables, and comments. Furthermore, ITSIMPLE provides templates for PDDL constructs, such as requirement specifications, predicates, actions, initial and goal definitions.

3.3 PDDL-Mode for Emacs

GNU Emacs is a text editor, primarily written in, and customizable by using Emacs Lisp, a Lisp dialect [48, 29]. The core values of Emacs are its extensibility and customizability. PDDL-mode [46] is a major Emacs mode, which determines the editing behavior of Emacs, for browsing and editing PDDL files. It provides syntax highlighting by way of basic pattern matching of keywords, variables and comments. Additional features are automatic indentation and code completion as well as bracket matching. Code snippets for the creation of domains, problems and actions are also available. Finally, the PDDL-mode keeps track of action and problem declarations by adding them to a menu and thus intending to allow for easy and fast code navigation.

3.4 Critical Review

All three tools, that have been described above, provide environments for the creation of PDDL code. However, each comes with its own advantages and disadvantages that are to be reviewed in this section. At the end of each discussed feature, the approach that will be used in myPDDL is introduced.

First and foremost, it must be mentioned that both PDDL STUDIO and ITSIMPLE were made from scratch, i.e. they do not build on existing editors and therefore cannot fall back on refined implementations of features that have been modified and improved many times throughout their existence. Many of their features must be regarded against this backdrop.

For instance, PDDL STUDIO has a parser implemented that enables code folding on a syntactical basis. PDDL-mode for Emacs, on the other hand could be customized to be capable of code folding either on the basis of indentation or on a syntactic level. Not providing a simple way to automatically indent code is one of the drawbacks of PDDL STUDIO and ITSIMPLE, since "{"in a large program, no indentation would be a real hindrance and very difficult to use. The same is true for overly indented programs.} [33]. Furthermore, both ITSIMPLE and PDDL STUDIO specify horizontal tab sizes of about ten spaces, while two to four spaces generally seem to be adequate [33]. To have both basic editor features [1] and a high customizability, it was decided to use an existing, extensible text editor to integrate myPDDL into.

The tools can also be compared in terms of their syntax highlighting capabilities. In PDDL-mode for Emacs, keywords (up to PDDL 2.2), variables, and comments are highlighted. However, this is only done via pattern matching without controlling for context. This means that wherever the respective terms appear within the code they will get highlighted, regardless of the syntactical correctness. Therefore, it is useful when the knowledge engineer is familiar with PDDL syntax, but can also be misleading if this is not the case. Different colors can be chosen by customizing Emacs. ITSIMPLE's syntax highlighting for PDDL 3.1 is, except for the PDDL version difference, equally as extensive as that of PDDL-mode for Emacs, but does not allow for any customization. Despite placing a larger emphasis on the creation of PDDL code from scratch within the ITSIMPLE modeling environment, syntax highlighting did not get more advanced with the latest version. PDDL STUDIO has advanced syntax highlighting that distinguishes all different PDDL 1.2 constructs, depending on the context, and allows knowledge engineers to choose their preferred highlighting colors. One of the primary objectives of myPDDL

[1] Features such as automatic indentation, selection of tab size, defining custom key shortcuts, customizing the general look and feel, displaying line numbers, and bracket matching.

is to help users in keeping track of their PDDL programs. As a means to this end, it was decided to also implement sophisticated, context-dependent syntax highlighting.

Another feature that can be useful for fast programming, is the ability to insert larger code skeletons or snippets. This allows the knowledge engineer to focus on the specific domain and problem characteristics instead of having to worry about the PDDL formalities. PDDL STUDIO does not support the insertion of code snippets at all. ITSIMPLE features some code templates for predicates, derived predicates, functions, actions, constraints, types, comments, requirements, objects, and metrics. However, the templates are neither customizable nor extensible. PDDL-mode for Emacs provides three larger skeletons, one for domains, one for problems and one for actions. Further skeletons could be added. myPDDL aims to combine the best of these latter tools and support customizable and extensible snippets for domains, problems, types, predicates, functions, actions and durative actions. In addition, to allow users to easily navigate within snippets, the option of going from one blank to the next by pressing the tabulator key ⇥ on the keyboard is also provided.

When it comes to visualization, neither PDDL STUDIO nor PDDL-mode for Emacs provide any visualization options. ITSIMPLE, on the other hand, is based entirely on visually modeling domains and problems. Therefore, since the first version, the focus has mainly been on exporting from UML.P to PDDL. myPDDL is to reverse this design approach and enable type diagram visualization of some parts of the PDDL code.

At this point, it must be mentioned that Tonidandel, Vaquero, and Silva [50] present a translation process, from a PDDL domain specification to an object-oriented UML.P model as a possible integration for ITSIMPLE. This translation process makes extensive semantic assumptions for PDDL descriptions. Two default classes *Agent* and *Environment*, corresponding to PDDL types, are incorporated into the Class Diagram. The first parameter in the `:parameters` section of an action is automatically declared as a subclass of the class *Agent*. In addition, each action will be allocated to the corresponding class of its first parameter in the Class Diagram. Furthermore, the first argument of a predicate is considered to be its main argument, so depending on their arity, predicates would be visualized differently:

- Nullary predicates would be allocated as attributes of the type *Environment*.
- Unary predicates would be declared as attributes of the type of the specified parameter.

- Binary predicates would be regarded as associations, expressed by an labeled arrow from the type of the first parameter to the type of the second one.

The described method is limited, because predicates with an arity of three or higher cannot be visualized. There is currently no ITSIMPLE version with this feature, according to an email from one of the authors, Tiago Vaquero, dated March 11 2014. This approach makes relatively large semantic assumptions that could distort the visualization. In contrast, myPDDL allocates predicates to every mentioned type in the variable list, and therefore allows for a representation of arbitrary n-ary predicates ($n > 0$). Actions are not visualized in myPDDL.

Searching for errors can be one of the most time consuming parts of the design process [14]. Hence, any tool that is able to help detect errors faster is of great value to the knowledge engineer. While PDDL-mode for Emacs and ITSIMPLE facilitate error detection only by basic syntax highlighting, PDDL STUDIO not only has syntactic but also semantic error detection implemented. Errors are detected immediately when they are made, thanks to the parser, and a dynamic table keeps track of them and provides error descriptions. Even though the immediacy with which errors are highlighted and added to the table can be helpful, it can also be premature at times. For example just because the closing parenthesis was not typed yet, does not mean it was forgotten. Therefore, for myPDDL the goal was to implement a more subtle syntactic error detection. Syntactic errors are simply not highlighted by the syntax highlighting feature, while all correct PDDL code is highlighted. Even though checking for semantic errors online should allow finding such errors before feeding the program to a planner, and thus increase the probability of feeding correct files to the planner, planning software is also able to detect semantic errors. For this reason, it was decided not to implement semantic error detection in myPDDL yet.

Another major drawback of PDDL STUDIO and PDDL-mode for Emacs especially, is that they are apparently not updated regularly to work with the most recent PDDL versions. PDDL STUDIO's parser is only able to parse PDDL 1.2, one of the first PDDL versions. As of writing this thesis, the latest PDDL version is 3.1. It must be mentioned that PDDL has evolved since PDDL 1.2 and was extended in PDDL 2.1 to include durative actions to model time dependent behaviors, numeric fluents to model non-binary changes of the world state, and plan-metrics to customize the evaluation of plans [13]. PDDL-mode for Emacs only works with PDDL versions up to 2.2, which introduced derived predicates and timed initial predicates [8], but does not recognize later features like object-fluents, so that the range of functions, specified in the domain file, cannot include object-types in addition to numbers. ITSIMPLE on the other hand is more regularly maintained and ITSIMPLE 4.0 is in beta status since 2012 [54]. The release will be the first ITSIMPLE version intended to also support the creation of PDDL documents from scratch, meaning that the text editor plays a much larger role in this version compared to previous ones.

Finally, one of the most important features of any software is the possibility of extending and customizing it [23]. Different programmers need to work with many different tools and need them to have a similar look and feel; they have different use cases and thus need different plug-ins and extensions to meet their needs, or they may simply have different preferences. PDDL STUDIO falls short of satisfying this requirement as the customization features (without editing the source code) are limited to the choice of font style and color of highlighted PDDL expressions. Furthermore, PDDL STUDIO is written as standalone program, meaning that there are no PDDL independent extensions. The same holds true for ITSIMPLE which is also not customizable without editing the source code. Being an Emacs mode and Emacs being an established text editor, PDDL-mode is highly and easily customizable and extensible.

This is the other major reason why it was decided that myPDDL should be integrated into a existing, extensible, and customizable text editor. These requirements are intended to be met by Sublime Text, a text editor that sports such features as customizable key bindings, display of line numbers and multi-line selection. In addition, there is a broad range of extensions for Sublime Text, so that features like revision control via Git, file management with a sidebar, color highlighting of matching brackets or comparing and merging files can be added. Furthermore, Sublime Text supports the majority of common programming and markup languages, in order for users to use the same tool and settings for programming and PDDL specifications.

myPDDL is designed as a package for Sublime Text and provides sophisticated syntax highlighting, code snippets, syntactical error detection and type diagram visualization. Additionally, it allows for the automation of modeling tasks due to a Clojure interface that supports the conversion of PDDL code into Clojure code and vice versa. Therefore, the myPDDL shell supports both the initial design process of creating domains (with code snippets, syntax highlighting and the Clojure interface), and the later step of checking the validity of existing domains and problems with the type diagram generator. Lastly, since it is increasingly important that several people work on one project together, the visualization capabilities of myPDDL are meant to help users to understand each other's code faster and thus be able to work with it more efficiently.

Chapter 4

Knowledge Engineering Tools for Automated Planning

4.1 Statement of Problem

The *Hacker World* and *Gary's Huge Problem*, presented in Chapter 2, already indicated that writing and maintaining PDDL files can be cumbersome [30, 59]. Due to the amount of information that has to be integrated for specifying PDDL domains, files can get hard to overlook. PDDL's modeling capabilities have been developed further with the release of new PDDL versions [13, 28] and it is likely that it will be used for even more complex, realistic domains that are designed by a team of experts instead of a single person [44]. For these purposes, it is "generally accepted that effective tool support is required to build domain models" [44].

The following sections will present myPDDL, a highly customizable and extensible modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promoting the collaboration between knowledge engineers and the use of PDDL in real-world applications. It consists of the following, integral parts:

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

myPDDL-syn A context-aware syntax highlighting feature.

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

myPDDL-loc Automated distance calculation for PDDL locations, specified in a problem file.

myPDDL-dia A type diagram generator for analyzing the structure of PDDL type hierarchies.

myPDDL-ide An integrated development environment for the use of myPDDL in Sublime Text.

A general interface between PDDL and Clojure allows for bypassing PDDL's limited mathematical modeling capacity and serves as a basis for myPDDL-new, -dia and -loc.

4.2 General Interface between PDDL and Clojure

Since PDDL is used to create more and more complex domains [15, 19] using the square root function for a distance optimization problem or the logarithmic function for modeling an engineering problem seems to be likely. However, PDDL's calculating capabilities are limited [38]. While these features are currently not supported by PDDL itself, preprocessing PDDL files and then hardcode the results back into the file seems to be a reasonable workaround. This preprocessing needs to be done outside of the PDDL environment, thus necessitating an interface with a programming language that supports a wide range of mathematical operations. With the help of such an interface, the modeling time can reduced an even partly automated (see 4.6 the distance calculator myPDDL-loc).

Since PDDL's syntax is inspired by Lisp [13], using a Lisp dialect for the interface stood to reason. Consequently, file input and output methods can use s-expressions (i.e. parenthesized lists) instead of regular expressions so that parts of PDDL files can be accessed and represented in a convenient way. For this thesis, it was decided to Clojure [21], a modern Lisp dialect that runs on the Java Virtual Machine (JVM) [31].

The interface can be used for generating PDDL constructs, reading domain and problem files, handling, using and modifying the input, and generating PDDL files as output. Once a part is extracted and represented in Clojure, the processing possibilities are diverse and the full capacities of Clojure can be used.

The interface is provided as a Clojure library and based on two methods:

read-construct(keyword, file) Allows for the extraction of code blocks from PDDL files [1]. Listing 11 shows an example, where the goal state

[1] In doing so, it uses a safe reader method, provided by `https://github.com/clojure/tools.reader` that avoids that possibly harmful read-in constructs can be executed.

of *Gary's Huge Problem* is extracted.

```
(read-construct :goal "garys-huge-problem.pddl")
;;=> ((:goal (exploited magicfailureapp)))
```

Listing 11: Extracting the goal state of *Gary's Huge Problem* by using the interface.

add-construct(file, position, part) Provides a means for adding constructs to a specified position in PDDL problem files. Listing 12 displays the application of this method.

```
(add-construct "garys-huge-problem.pddl" :init '((hungry gisela)))
```

Listing 12: Adding the predicate (hungry gisela) to *Gary's Huge Problem* using the PDDL/Clojure interface.

Subsequently, the modified part of the problem file looks like follows:

```
...
 (:init
  (hungry gary)
  (vulnerable magicfailureapp)
  (has magicfailureapp gisela)
  (hungry gisela))
...
```

4.3 Create PDDL Projects (myPDDL-new)

In many cases, creating PDDL domains is an ad-hoc process [45]. However, each implementation of a PDDL task specification comprises the creation of one domain and at least a corresponding problem file. Since several team members may be working on these files, having a structure for these PDDL projects would be desirable. To this end, a standardized project folder structure could facilitate the collaboration and maintenance of consistency across projects and support a structured and organized design process.

myPDDL introduces such a standardized project folder structure by creating a new project folder on the basis of a project name. Figure 4.5) illustrates the created structure and files.

Figure 4.1: The project folder structure created by myPDDL-new. The project-name is chosen by the user and automatically used for the name of the created domain.

% .1 project-name. .2 dot. .2 diagrams. .2 domains. .2 problems. .3 p01.pddl. .2 solutions. .2 domain.pddl. .2 README.md. }

```
project-name/
├── dot/
├── diagrams/
├── domains/
├── problems/
│   └── p01.pddl
├── solutions/
├── domain.pddl
└── README.md
```

Figure 4.2: The project folder structure created by myPDDL-new. The project-name is chosen by the user and automatically used for the name of the created domain.

Within this project folder, the domain file `domain.pddl` and the problem file `p01.pddl` (in the folder `problems`) are filled with basic PDDL skeletons. The templates for the skeletons can be customized to the user's needs by editing the template.
The folders `domains/`, `dot/`, and `diagrams/` are created for the use with myPDDL-gen that saves its generated output to these folders and thereby allows for a basic version control system (see 4.7Type Diagram Generator (myPDDL-dia)).
As one domain file can have multiple problem files, the folder `problems/` is designed for the collection of all associated problem files.
Recognizing that knowledge engineers often do not write any documentation related to the specified planning task [44], `README.md` is a Markdown file (a plain text formatting syntax [18]), which is, among others, intended for information about the author(s) of the project, contact information, informal domain and problem specifications, and licensing information. Markdown files are converted to HTML by various hosting services that use the Git system (like GitHub, Bitbucket). This file can hence be used as a quick overview for PDDL projects, located at a hosting service (vllt die beiden sätze zusammenfügen) and tackles the need for team work [44]. The choice

of a folder structure (instead of a project file that contains information about the associated domain and problem files) has the advantage of being readable and customizable independent of the editor.

4.4 Syntax Highlighting (myPDDL-syn)

* Statement of Problem

Continually growing PDDL files can span several pages and consist of hundreds or thousands of lines of code. Thus detecting syntax errors and keeping track of the file structure can be become a real challenge for knowledge engineers. In order to recognize file elements quickly and detect errors at a glance, highlighting syntactical constructs is an established feature of text editors [41].

myPDDL-syn is a PDDL context-aware syntax highlighting feature for Sublime Text. It distinguishes all PDDL constructs up to version 3.1 [2], like comments, variables, names, and keywords and highlights them in different colors. By means of a sophisticated pattern matching heuristic that can both recognize the start and the end of code blocks, myPDDL identifies the current scope, i.e. code block [3]. These scopes allow for a fragmentation of the PDDL files, so that constructs are only highlighted, if they appear in the correct code block. Thus missing brackets, misplaced expressions and misspelled keywords are visually distinct and can be identified.

Figure `fig:syntax-highlighting` displays an extract of the domain *Coffee*, which was used in the user study for the evaluation of this tool.

Sublime Text then colorizes different parts of the code according to these names and the used color scheme, so that the look and feel can be alternated by changing Sublime Text's color scheme [4].

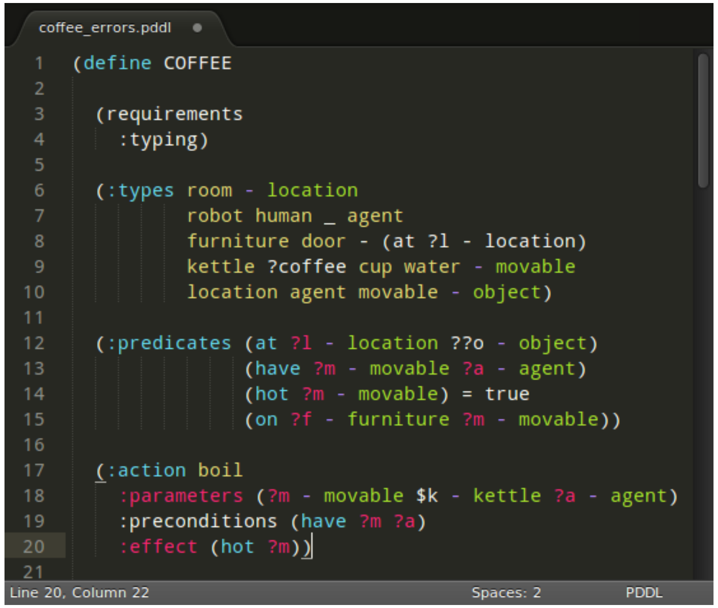
For the ease of creation, the PDDL syntax highlighter is implemented by the use of the ST plug-in *AAAPackageDev* [1]. So, the definitions can be written in YAML in converted to Plist XML later on.

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

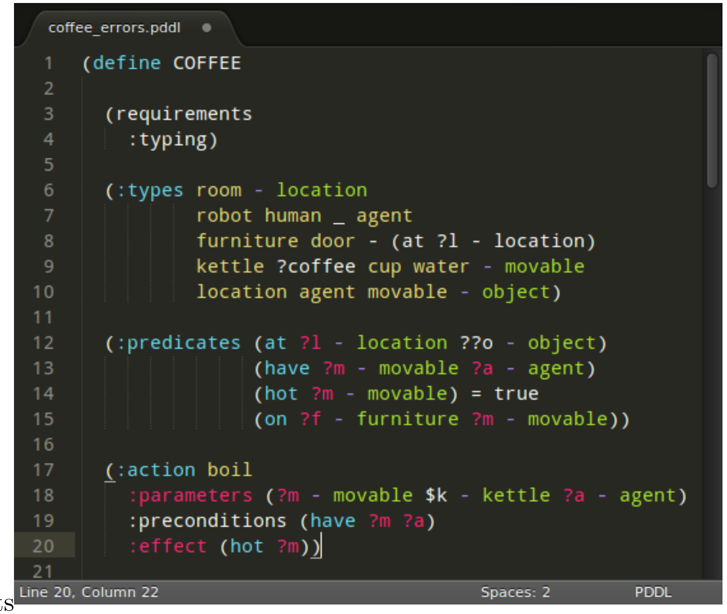
[2] The regular expressions that are used to detect different parts are based on the Backus-Naur Form (BNF) descriptions, formulated in Kovacs [28], Fox and Long [13], and McDermott et al. [32].

[3] Context is meant in the sense of code blocks here. That means is a construct written in ... `(:predicates ...)`) or `(:action ...)`

[4] The extend to which the syntax highlighting is supported is dependent on the used color scheme. By using the built-in Sublime Text color scheme *Monokai*, all syntactically correct PDDL constructs get highlighted. Non-highlighted (i.e. white) constructs either contain syntactical errors or are not specified by PDDL 3.1.



(a) Using the syntax highlighting feature incorrect PDDL constructs can be easily identified as they are not highlighted and therefore displayed in white.



(b) The same domain as on the left hand side without syntax highlighting.

Figure 4.3: The deliberately erroneous domain *Coffee* without 4.3a and with 4.3b syntax highlighting, using the Sublime Text color scheme /Monokai/.

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.

A key challenge of creating a context-aware syntax highlighter without the availability of a lexical parser, is the use of regular expressions for creating a preferably complete PDDL identification. 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.

4.5 Code Snippets (myPDDL-snp)

Consider again the basic skeleton of an action:

```
(:action action-name
        :parameters (?x - object)
        :precondition (and (pred-1))
        :effect (and ))
```

Listing 13: Skeleton of an PDDL action

Almost all PDDL actions consist of these same parts. Writing and extending PDDL files, knowledge engineers therefore have to to use the same constructs again and again. This is where code snippets come in. To facilitate and speed up the implementation of standard constructs, myPDDL-snp provides code skeletons, i.e. templates for often used PDDL constructs, like domain and problem definitions, predicates and actions. They can be inserted by typing a triggering keyword. Typing `action` and pressing the tabulator key (⇥), inserts the action specified in Listing 13. PDDL constructs with a specified arity can be inserted by adding the arity number to the trigger keyword (`p2` would insert the binary predicate template `(pred-name ?x - object ?y - object)`.

Once the snippets has been inserted, skipping from blank to blank is enabled by pressing tabulator key.

Therefore, having a means to easily navigate the snippets would become handy. For this purpose, the blanks can be filled by pressing ⇥on the keyboard and thereby navigating inside the snippet, so that the cursor will first mark the action-name .q inserted content contains fields with placeholders, that can be accessed and filled in consecutively.

Every snippet is stored in a separate file, located in the folder `Packages/PDDL/` of Sublime Text. New snippets can be added and existing snippets can be customized (change the template or the triggering keyword) in this folder.

4.6 Distance Calculation for PDDL Locations (myPDDL-loc)

Hacker World in Chapter 2, defines the predicate `(location ?f - furniture ?x ?y - number)`. A possible extension to this domain would be an action that is only applicable if a person is within a certain distance to an object. In order to determine this distance, it could be desirable to use the Euclidean distance which includes the square root function ($\sqrt{}$). However, PDDL 3.1 supports only four arithmetic operators (`+`, `-`, `/`, `*`). These operators can be used in preconditions, effects and durations. Parkinson and Longstaff [38] describe a workaround for this drawback. By writing 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. The square root can be approximated using the Babylonian method, requiring many iterations, this method would most likely have an adverse effect on plan generation [38].

The PDDL/Clojure interface reads a problem file and extracts all locations, defined in the `(:init ...)` code block. In Clojure, the Euclidean distances between all locations are then calculated and written back to an extended problem file.

The calculator works on any arity of the specified predicate, so that locations could be specified one, two and three dimensionally and even used in higher dimensions.

However, this workaround certainly has a major drawback, apart from the time required to calculate possibly unused distances. If the number of locations is n, the number of calculated distances is n^2, because every location has a distance to every other location. The calculated distances have to be stored in the PDDL problem file, potentially requiring a lot of space. Therefore, a sensible next step to extend PDDL would be to increase its mathematical expressivity [38]. One possibility would be to declare a requirement `:math` that specifies further mathematical operations and to extend PDDL in future versions.

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

Listing 14: Extract of the extended file /Gary's Huge Problem' before using myPDDL-loc.

```
(: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 15: After the application of myPDDL-loc, the calculated distances are inserted in the `(:init ...)` code block of a copy of the problem file.

4.7 Type Diagram Generator (myPDDL-dia)

TODO: Write something about diagrammatic reasoning
. . . furthermore, visual information can be transmitted more precisely [16] and remembered better [35]. The diagrammatic representation of textual information helps to quickly understand the connection of hierarchically structured items [49] and is supposed to simplify the communication and collaboration between developers.

Object types play a major role in typed PDDL domains: they constrain the types of arguments to predicates and determine the types of parameters used in actions. In order to use and extend available domains, a crucial part is understanding the involved types, their hierarchy and identifying the constructs that make use of them. However, this can be difficult by just reading the textual representation of the hierarchy, so a diagram that displays this hierarchy could be helpful.

Creating such a diagram manually each time a change is made can be unnerving and costly in terms of time. An automatically created graphical representation could save time and energy.

myPDDL-dia serves this purpose, by generating and displaying diagrams by means of domain files. Figure 4.4 shows the generated diagram from the

Hacker World in Chapter 2. In the diagram, types are represented with boxes, with every box consisting 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 as a parameter. The predicates are written just as they appear in the PDDL code.

Generalization relationships ("is a", e.g. "a laptop *is a* computer") express that every subtype is also an instance of the illustrated super type. This relationship is indicated in the diagram with arrow from the subtype (here: *laptop*) to the super type, where the arrow head aims at the super type (here: *computer*).

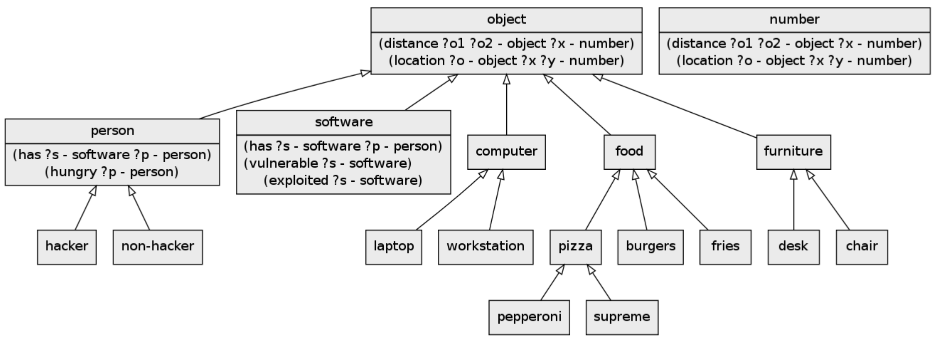


Figure 4.4: The type diagram that was generated from the *Hacker World* in Chapter 2 using myPDDL-dia.

In order to create the diagram, myPDDL-dia makes use of the PDDl/Clojure interface that extracts the `(:types ...)` block. Then, using regular expressions, the extracted types get split into super types and associated subtypes and stored in a Clojure hash-map.

Subsequently, the diagram is generated using dot from the Graphviz package [9], a collection of programs for drawing graphs. dot is a scriptable, graphing tool, that is able to generate hierarchical drawings of directed graphs in a variety of output formats (e.g. PNG, PDF, SVG). The input to dot are text files, written in the DOT language.

Based on the Clojure representation, the description of a directed graph in the DOT language is created and saved in the folder `dot/` that is located in the same folder as the PDDL domain file. The DOT file is then passed to dot, creating a PNG diagram and saving it in the folder `diagrams/`. Additionally, the diagram is immediately displayed in a window. Simultaneously, a copy of

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

Figure 4.5: Folder structure after two invocations of myPDDL-dia. Files and folders are automatically created and extended by a revision number (`0`,`1`) each time dia is used.

the domain file is stored in the folder `domains/`. Every time myPDDL-dia is invoked, these steps are executed and the names of the saved files are extended by an ascending revision number. Thus, one cannot only identify associated PDDL, DOT and PNG files, but also use this feature for basic revision control. Figure `fig:mypddl-new-project-folder` displays the folder structure after invoking myPDDL-dia twice on the *Hacker World*. The type hierarchy and predicate structure of a previous version of a domain file can thereby be identified by the corresponding type diagram (both files have matching revision numbers), and it is possible to revert to a previous revision, stored in the folder `domains/` .

4.8 Integrated Design Environment (myPDDL-ide)

The above presented tools provide a command-line interface for interacting with the user. This offers high flexibility, the possible automation of jobs by using scripts, and the possibility for integration into different software. However, the user has to be familiar with the underlying syntax in order to use the full spectrum of available myPDDL functions. By using Sublime Text as editor, all the language independent features are supported that are described in the Sublime Text 2 Documentation, like auto completion of words already used in this file, code folding and column selection. Sublime Text is used to combine the so far presented command-line tools, as well as the syntax highlighter and the code snippets to form an IDE. While myPDDL-snp and syn are devised explicitly for Sublime Text and therefore integrated from the outset, the other tools (new, dia, loc) can be used independently of ST with the command-line interface and any PDDL file. To provide an IDE for using myPDDL, *-sub* integrates new, dia and loc, aiming at a a user-friendly execution and use of the system.

This way a menu-driven interface is provided and 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 myPDDL-new requires the user to specify a project name in the then displayed input panel.

PDDL: Calculate Distances for myPDDL-loc Saves and

PDDL: Display Diagram for myPDDL-dia

myPDDL can be installed automatically via Sublime Text Package Control [4] or by copying the files of myPDDL[5] into the packages folder of Sublime Text [6]. Following, the features can be activated by changing Sublime Text's syntax to PDDL (View › Syntax › PDDL).

[5]The files can be downloaded from `https://github.com/Pold87/ba-thesis/`.

[6]Further information about Sublime Text packages can be found at `http://www.sublimetext.com/docs/3/packages.html`.

Chapter 5

Evaluation

To evaluate a software means to assess its quality. Appropriate criteria are supplied by Shah et al. [44] who evaluate different knowledge engineering tools in planning including ITSIMPLE. All in all, they identified seven criteria, the lead questions of which can be found in Table 5.1.

The first criterion (*operationality*) was not of interest when developing myPDDL, since it can be reduced to the question of whether planners perform well on PDDL files created using a text editor (instead of a graphical language). Therefore, it was decided to replace the criterion *operationality* with *functional suitability* from the ISCO/IEC 25010 standard. To assess the functional suitability and to illustrate where myPDDL fits in with similar tools, it was compared to the other three tools introduced and discussed in chapter 3, namely PDDL STUDIO, ITSIMPLE, and PDDL-mode for Emacs. Of the remaining six criteria in Table 5.1, *collaboration*, *experience*, and *debugging* were tested with a usability test. The other three criteria, *maintenance*, *efficiency*, and *support*, will simply be discussed.

5.1 Benchmarking

The comparison of myPDDL to the three tools from Chapter ch:related is intended to to show its appropriateness or "the degree to which the software product provides an appropriate set of functions for specified tasks and user objectives" (ISO 25010 6.1.1). Where does myPDDL fit in with existing tools for the same purpose? When and for which tasks it is best suited? The major user objective is identical for all four tools and can be summed up as the desire to integrate human knowledge into a knowledge based system, in particular to create domains and problems that can be fed to a planner. All tools intend to support this process in general and the various stages of the

Criteria	*Description*
Operationality	How efficient are models produced?
	Is the method able to improve the performances of planners on generated models and problems?
Collaboration	Does the method/tool help in team efforts?
	Is the method/tool suitable for being exploited in teams or is it focused on supporting the work of a single user?
Maintenance	How easy is it to come back and change a model?
	Is there any type of documentation that is automatically generated?
	Does the tool induce users to produce documentation?
Experience	Is the method/tool indicated for inexperienced users?
	Do users need to have good knowledge of PDDL?
	Is it able to support users and to hide low level details?
Efficiency	How quickly are acceptable models produced?
Debugging	Does the method/tool support debugging?
	Does it down on the time needed to debug?
	Is there any mechanism for promoting the overall quality of the model?
Support	Are there manuals available for using the method/tools?
	Is it easy to receive support?
	Is there an active community using the tool?

Table 5.1: The seven design criteria that were identified by Shah et al. [44]

design cycle to different degrees. However, sometimes knowledge engineers may only have to alter or develop already existing models further. myPDDL aims to also assist with the objective to quickly understand foreign code. Table 5.2 illustrates how the four tools compare in terms of features and how each of these features is helpful in the knowledge engineering process.

TODO!

5.2 Usability Test

The nightmare of any system development group is spending years and vast amounts of money on developing a system and finding, upon its release, that users cannot interact with it properly or do not see how it can help them. When designing and implementing a system intended to support humans, it is therefore of great importance to determine its usability [26]. An common method for doing so is by usability testing (inviting users to thoroughly test the software by means of a series of realistic tasks and asking their opinions). Therefore, two of the most important myPDDL features, syntax highlighting and type diagram generation, were tested in a small user study.

5.2.1 Participants

A total of eight participants (three female participants, average age: 22.9, standard deviation: 0.6) took part in this usability test [1]. Eight subjects was the minimum possible in this study in order to fully control for possible sequence and learning effects. Also, it was ensured that all participants were familiar with at least one Lisp dialect, so that no one would be confused by program code written as parenthesized lists. None of the participants had prior experience with planning in general or PDDL in particular. Furthermore, none of them had used Sublime Text before.

5.2.2 Material

It was decided to conduct the experiment at the home of the experimenter to have a more welcoming and relaxing atmosphere than in a university laboratory. A 30-minute interactive video tutorial for planning and PDDL was recorded to familiarize participants with the topic [2]. The tutorial made no reference to myPDDL. A second three minute video introduction to the functionality of the syntax highlighter (myPDDL-syn) and the usage of the type diagram generator (myPDDL-dia) was recorded. A preliminary questionnaire was designed to assess the prior experience with planning and Sublime Text. The system usability scale [6] was chosen as a post questionnaire to measure participants' attitudes concerning the two tested tools. The participants completed the actual tasks on a laptop computer (15.6 inch screen) with an additional screen (15.1 inch) for displaying the type diagram and the code side by side. All tasks were completed using the Sublime Text editor with the color scheme *Monokai* [3]. The times that participants took to answer questions were recorded using an online tool [4] as this allowed splitting the total

[1] Small sample sizes are sometimes criticized even in usability studies, because it is hard to detect issues that only few people have. For example, the number of people that are affected by hard-to-find information on vegan food served on a flight or by hard-to-find information on luggage constraints differs. Therefore, the latter will most likely be uncovered with small sample sizes, while the former might not. Reviewing the scientific discussion on this topic at this point is beyond the scope of this thesis, but testing more than the common five participants [36, 37, 56] is in line with more recent research [10, 24].

[2] `http://www.youtube.com/playlist?list=PL3CZzLUZuiIMWEfJxy-G6OxYVzUrvjwuV`

[3] *Monokai* is the default color scheme of Sublime Text. It has a dark gray background.

[4] `http://online-stopwatch.chronme.com`

time on task into smaller times for subtasks. Furthermore, the recorded times could be downloaded directly as a CSV file. To test the syntax highlighting and the type diagram generator, two different task types were needed. As a within subjects design was considered most suited (to control for individual differences within such a small sample), it was necessary to construct two tasks, matched in difficulty, for each of these two types to compare the effects of having the tools available. Subsequently, domains and instructions were written for these four tasks. The two tasks to test syntax highlighting presented the user with domains that were 54 lines in length, consisted of 1605 characters and contained 17 errors each. Errors were distributed evenly throughout the domains and were categorized into different types. The occurrence frequencies of these types were matched across domains as well, to ensure equal difficulty for both domains (see table 7.2 in the appendix). To test the type diagram generator, two fictional domains with equally complex type hierarchies consisting of non-words were designed (five and six layers in depth, 20 and 21 types). The domains were also matched in length and overall complexity (five and six predicates with approximately the same distribution of arities, one action with four predicates in the precondition and two and three predicates in the effect). All questionnaires and task descriptions can be found in the appendix. Lastly, participants were given pen and paper and a one page summary (*cheat sheet*) of PDDL domain syntax (see appendix) to help them solve tasks if they saw a need for it.

5.2.3 Method

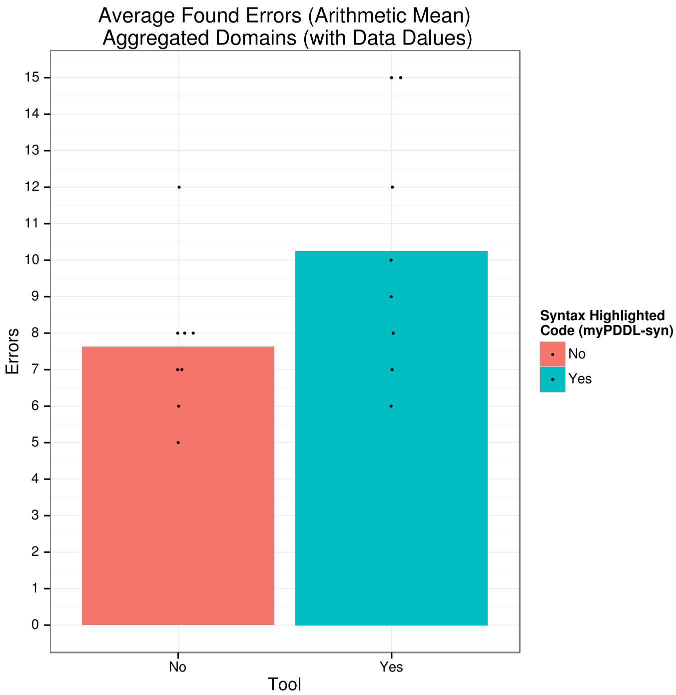
No earlier than 24 hours before the experiment was to take place, participants received the web link to the tutorial and were thus given the option to watch it in their own time if they felt so inclined. This method was chosen, because it was important that participants learn and understand the contents and this could be hindered by the presence of the experimenter or the testing situation, depending on the subject's personality. Upon their arrival, participants were handed a consent form and the preliminary questionnaire. If they had already watched the tutorial, they were asked if they had any questions concerning the tutorial and if they thought that they had understood everything. Otherwise, they proceeded to do so. After watching the tutorial, they were asked to complete the tasks in the order specified in table 7.1 in the appendix. Two factors were varied: whether the participant had the tools available for the first two tasks or for the second two tasks, and whether the participant started with a debugging task or a type hierarchy task. Directly before being given the tasks requiring the application of the tools, the three minute video introduction was shown. For the debugging tasks, participants were given six minutes [5] to detect as many of the errors as possible. They were asked to record each error in a table (pen and paper) with the line number and a short comment and to immediately correct the errors in the code if they knew how to, but not to dwell on the correction otherwise. For the type hierarchy task, participants were asked to answer five questions concerning the domains, all of which could be facilitated with the type diagram generator, but some of which still required looking in the code. Participants were told, that they should not feel pressured to answer quickly, but to not waste time either. Also they were asked to say their answer out loud, as soon as it became evident to them. They were not told that the time it took them to come up with an answer was recorded, since this knowledge could have made them feel pressured and thus led to more false answers. At the end of the usability test, they were asked to evaluate the perceived usability of myPDDL using the system usability scale.

5.2.4 Results

1. Debugging Tasks

   To test the syntax highlighter, participants were asked to find as many errors as they could in a given domain within a certain amount of time. On average, participants found 7.6 errors without syntax highlighting and 10.25 errors with syntax highlighting (i.e. approximately 35 % more errors were found with syntax highlighting). The difference is shown in Diagram `found-errors-combined`.

[5] A reasonable time frame tested on two pilots.



figure

Here, it is worth looking at the raw data, though, and more specifically the raw data in combination with the comments and remarks made by the participants during testing. Two participants (participants 5 and 6) remarked that the syntax highlighting colors confused them and that they found them more distracting than helpful. One of them even mentioned that the contrast of the colors used was so low that they were hard for her to distinguish. She found the same number of errors with and without syntax highlighting, the other of the two was the only participant that found less errors with syntax highlighting than without it. Excluding these two participants from the analysis results in 49 % more errors being found with the help of syntax highlighting. Since only eight participants took part in the study, splitting the data up further (to compare, for example, the errors found per domain) did not appear sensible as aggregated data values would then consist of data from four participants at the most. A between groups comparison with only four individuals per group is most likely not going to provide insightful and reliable results (TODO: source).

Type Hierarchy Tasks

Diagram `fig:task-completions-agg` shows the geometric mean [6] of the

[6]The geometric mean is a more accurate measure of the mean for small sample sizes as task times have a strong tendency to be positively skewed [43].

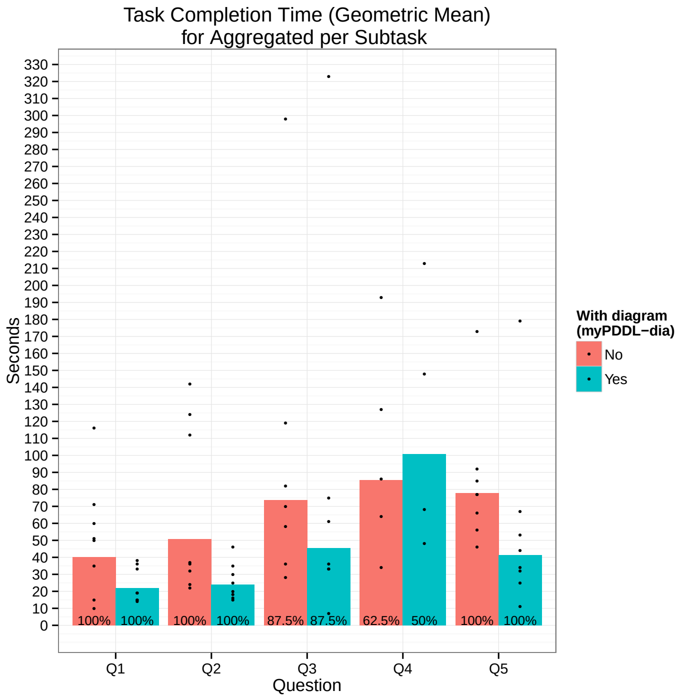


Figure 5.1: The diagram shows the geometric mean of the task completion time. The crosses $(\times) representoneparticipant.$

task completion time [7] for each question with and without the type diagram generator.

It can be seen that when having the type diagram generator available, participants answer all questions, except question 4, nearly twice as fast. The fact that the availability of tools did not have a positive effect on task completion times for question 4 can probably be attributed to the complexity of this question. In contrast to the other four questions, to answer question 4 correctly, the participants were required to look at the actions in the domain file in addition to the type diagram. Most participants were confused by this, because they had assumed that once having the type diagram available, it alone would suffice to answer all questions. This initial confusion cost some time, thus negatively influencing the time on task. Therefore, question four is excluded from Diagram `task-completions-agg-without-q4`.

Diagram `task-completions-agg-without-q4` illustrates the effect that the tool availability had on the total time participants spent on a task, with-

[7]The task completion time only includes task times of users who completed the task successfully.

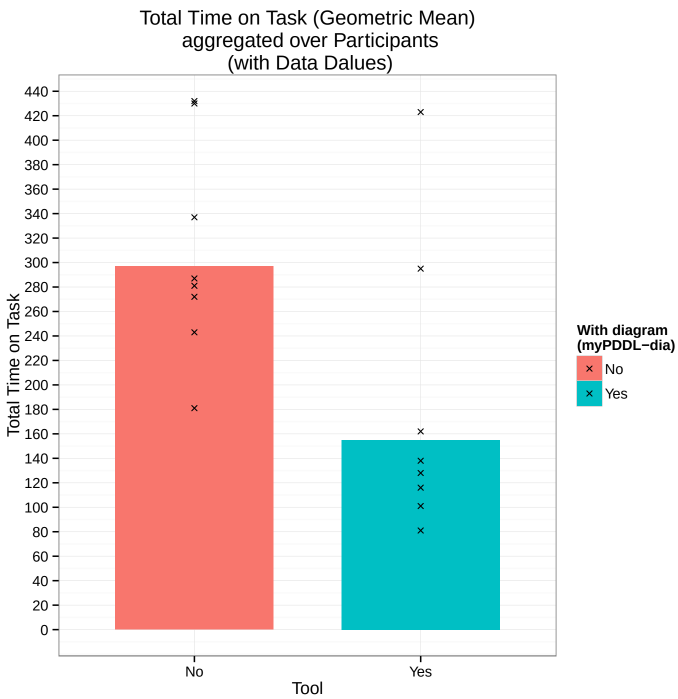


Figure 5.2: Geometric mean of the total time on task without Question 4

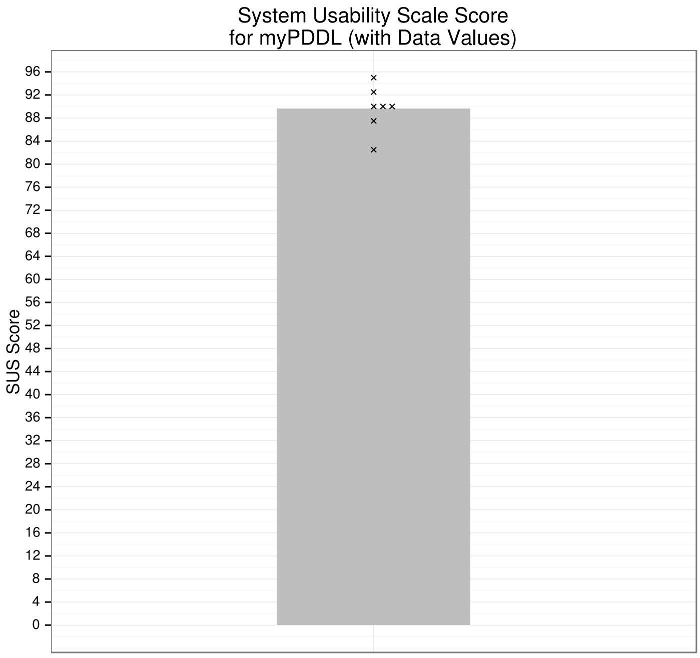


Figure 5.3: myPDDL got a score of 89.6 on the system usability scale that ranges from 0 - 100. Every cross $(\times) indicatesthesystemusabilityscalevalueofoneparticipant. Itisevidentthatallscoresrangeinthe$

out question 4. It only takes 52% of the time without tools to answer questions 1, 2, 3 and 5 once tools can be used.

System Usability Scale

myPDDL reached a score of 89.6 on the system usability scale [8], with a standard deviation of 3.9. Figure `fig:sus` displays this score and the data values of the seven participants who completed this questionnaire. Since the overall mean score of the system usability scale is approximately 68 with a standard deviation of 12.5 [42], the score of myPDDL is well above average with a small standard deviation. A score of 89.6 is usually assigned to superior products [3]. Furthermore, 89.6 corresponds approximately to a percentile rank of 99.8 %, meaning that it has a better perceived ease-of-use than 99.8 % of the products in the database used in Sauro [42].

[8] The range of possible values for the system usability scale is 0 to 100.

5.2.5 Discussion

Although participants' responses in the System Usability Scale seem to indicate that myPDDL is an enjoyable software with a high usability, it is still necessary that it is evaluated with regard to the afore mentioned criteria in order to reach an informed conclusion concerning its quality.

1. Functional Suitability

   When comparing the feature diversity of myPDDL with that of other tools for the same purpose, it becomes immediately evident that myPDDL is not only up to par, but, integrated into Sublime Text, it provides a rich developing environment for PDDL files. One of its strengths lies in being up-to-date and supporting the most recent PDDL version. Due to the Sublime Text integration, it also offers all basic editor features and a high customizability. Being the only one of the four tools capable of visualizing parts of the PDDL code, it must be understood as complementary to ITSIMPLE, which takes the opposite approach of transforming UML diagrams into PDDL files. The fact that myPDDL does not check for semantic errors is not actually a drawback, as planners will usually detect semantic errors. All in all, myPDDL combines the most useful tools of PDDL STUDIO, ITSIMPLE and PDDL-mode for Emacs and strives to support the knowledge engineer during all phases of the modeling process. Additionally, it features some unique tools, which have, in part, already been proven to be helpful in the user study. It can therefore be concluded that myPDDL provides an appropriate set of functions for developing PDDL files and is thus functionally suitable.

2. Collaboration

   With the growing importance of team work and team members not necessarily working in the same building or in the same country for that matter, there is an increasing need for tools supporting the collaboration effort. In developing myPDDL, this need was sought to be met by myPDDL-dia. Complex type hierarchies can be hard to overlook, especially if they were constructed by someone else. Therefore, a good way of tackling this problem seemed to be by providing a means to visualize such hierarchies in the form of type diagrams. But is this really able to help knowledge engineers? This was tested in the user study by having participants answer questions on complex type hierarchies that they had never seen before, made up entirely of non-words (to eliminate the interference of world knowledge). With the help of the diagram participants were able to answer questions correctly nearly twice as fast as without this service. This indicates that myPDDL-gen helps to understand foreign domain structures faster and more easily, therefore facilitating the collaboration between knowledge engineers.

3. Experience

   Although it is assumed that most people interested in using myPDDL have some experience in planning, unversed users should not be put off by a complex or incomprehensible interface. To test if myPDDL is usable even for novices, it was chosen to only allow for inexperienced participants in the user study. However they were all required to be familiar with at least one Lisp dialect because PDDL is Lisp-based. As described above, participants were asked to watch a 30-minute interactive planning and PDDL video tutorial, establishing the basics. Before having to use the myPDDL tools, another three minute video tutorial introduced myPDDL-syn and myPDDL-dia. None of the users seemed to have any problems in comprehending either of these tools after seeing the tutorial. Therefore, it can be concluded that a small time investment for inexperienced users, and an possibly even smaller investment for PDDL experts, results in proficient usage of myPDDL. Nonetheless, two subjects reported having trouble with myPDDL-syn, because they did not find it helpful or because they found the colors hard to distinguish. It seems, though, that this cannot be accounted to inexperience so much as to personal preference.

4. Debugging

   To help users find errors faster, the syntax highlighting feature myPDDL-syn was created. It highlights all syntactically correct constructs and leaves all syntactical errors non-highlighted. In order to assess if myPDDL helps in debugging, myPDDL-syn was included in the usability test. Users were asked to find as many errors as they could in a specified amount of time. Six of the eight subjects found more errors with syntax highlighting than without it, one person found the same number and one actually found one error less with the tool available. These latter two users reported being distracted by the colors. Syntax highlighting is optional in the Sublime Text integration, thanks to the high customizability, and what is more, different color schemes allow for the individualization of myPDDL-syn. However, this option was not made available in the user study, as it would have been an additional, possibly confounding variable, making the results harder to interpret. While eight people seem too few to generalize the numbers, it can be said that myPDDL with myPDDL-syn seems to help the majority of users in debugging.

5. Maintenance

   The possibility to maintain PDDL files is a key aspect of myPDDL. The automatically generated type diagram (myPDDL-dia) gives an overview of the domain structure, and thereby serves as a continuous means of documentation. In addition, the diagram supports the understanding of existing or extended domains provided by other knowledge engineers. Helping to understand foreign code, though, it follows logically that myPDDL-dia also helps in coming back and changing one's own models if some time has elapsed since they were last edited. The basic revision control feature of myPDDL-dia keeps track of changes, making it easy to revert to a previous domain version. Furthermore, myPDDL-new encourages adhering to an organized project structure and stores corresponding files at the same location. Text-based modeling facilitates using a common revision control system like Git [17]. The automatically created readme file can induce the user to provide further information and documentation about the PDDL project. When using a web-based hosting service, the readme file is usually prominently displayed on the home page of the project.

6. Efficiency

   All myPDDL tools are intended to increase the efficiency with which PDDL files are made. For one, code snippets enable the fast creation of large and correct code skeletons that only still need to be filled in. While the type diagram generator facilitates collaboration, it can also be used to double check if the type hierarchy and the predicates that use the types have been implemented as intended. Syntax highlighting can reduce the time spent on searching errors. Code folding allows users to hide currently irrelevant parts of the code and automatic indentation increases its legibility. To easily keep track of all the parts of a project, folders are automatically created and named. Lastly, it is also possible to customize myPDDL so as to adapt its look and feel to other programs one is already familiar with, or simply to make it more enjoyable to use. All these features should increase the efficiency, and especially skilled knowledge engineers should be able to produce acceptable models very quickly with the help of myPDDL.

7. Support

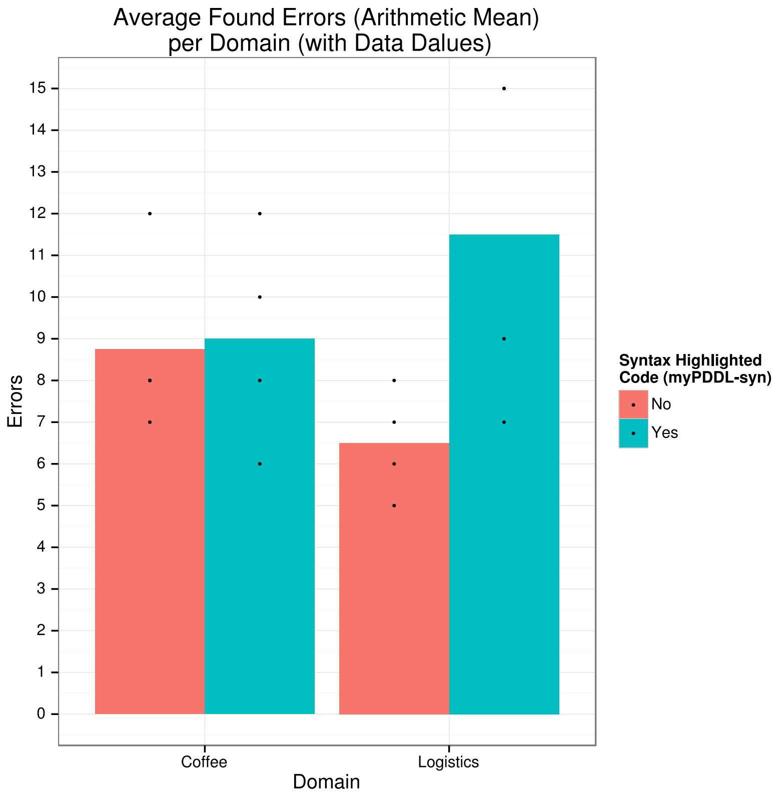


Figure 5.4: The Domain Planet Splisus.

myPDDL can be installed using Sublime Text's Package Control [4]. This allows for an easy installation and staying up-to-date with future versions. In order to provide global access and with it the possibility for developing an active community, the project source code is hosted on GitHub [9]. An additional project site [10] provides myPDDL video introductions, a manual and room for discussing features and reporting bugs.

[9] `https://github.com/Pold87/myPDDL`
[10] `http://pold87.github.io/sublime-pddl/`

	Function	PDDL STUDIO	ITSIMPLE	PDDL-mode	myPDDL
latest supported PDDL version	considering recent PDDL features	1.2	3.1	2.2	3.1
syntax highlighting	supporting error detection and navigation	Yes	Yes	Yes	Yes
syntactic error detection	supporting debugging	Yes	No	No	Yes
automatic indentation	supporting readability and navigation	No	No	Yes	Yes
semantic error detection	supporting debugging	Yes	No	No	No
code completion	speeding-up knowledge engineering process	Yes	No	Yes	Yes
code snippets	speeding-up knowledge engineering process	No	Yes	Yes	Yes
	externalizing user's memory				
code folding	supporting keeping an overview	Yes	No	Maybe	Yes
project management	supporting keeping an overview	Yes	Yes	No	Yes
PDDL code visualization	supporting fast understanding				
¨ to PDDL code translation	supporting initial modeling	No	Yes	No	Yes
customization features	acknowledging individual needs and preferences				
planner integration	allowing for easy planner access	Basic	Yes	No	Yes?
plan visualization	supports understanding and crosschecking the plan	no	yes	no	no
Petri net visualization	supporting dynamic domain analysis and model checking	no	yes	no	no
declaration menu	supporting easy code navigation	no	no	yes	no
interface with programming language	automating tasks				
	extending PDDL's modeling capabilities				

Table 5.2: Comparison of tools and their features for creating domains.

Chapter 6

Conclusion and Future Work

myPDDL has been designed to support knowledge engineers in modeling planning task specifications as well as in understanding, modifying, extending and using existing planning domains. myPDDL is written in a modular fashion, is extensible and customizable. 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 core features, syntax highlighting (myPDDL-sub) and type diagram generation (myPDDL-dia) 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 understandingq predicates using these types.

Knowledge engineers can customize and extend Sublime Text and use existing text editor features. Although myPDDL is made available in Sublime Text, users could transfer the ideas to other text editors.

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

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.

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.

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 knowledge engineering to the automated knowledge acquisition (KA). Perception systems, Nevertheless, a engineer who double-checks the generated tasks will be irreplaceable.

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.

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] Rajendra Akerkar. "Intelligent Systems: Perspectives and Research Challenges". In: *CSI Communications* (2012), p. 5.

[3] Aaron Bangor, Philip T Kortum, and James T Miller. "An empirical evaluation of the system usability scale". In: *Intl. Journal of Human–Computer Interaction* 24.6 (2008), pp. 574–594.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide*. Pearson Education India, 1999.

[6] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry* 189 (1996), p. 194.

[7] Miroslav Chomut. "Tool for editing PDDL projects". In: (2012), plch.

[8] Stefan Edelkamp and Jörg Hoffmann. "PDDL2. 2: The language for the classical part of the 4th international planning competition". In: *4th International Planning Competition (IPC'04), at ICAPS'04* (2004).

[9] John Ellson et al. "Graphviz—open source graph drawing tools". In: *Graph Drawing*. Springer. 2002, pp. 483–484.

[10] Laura Faulkner. "Beyond the five-user assumption: Benefits of increased sample sizes in usability testing". In: *Behavior Research Methods, Instruments, & Computers* 35.3 (2003), pp. 379–383.

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

[12] Matthias Felleisen et al. *How to Design Classes (Draft)*. 2010.

[13] 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.

[14] Stanley Gill. “The Diagnosis of Mistakes in Programmes on the EDSAC”. In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 206.1087 (1951), pp. 538–554.

[15] Robert P Goldman and Peter Keller. ““Type Problem in Domain Description!” or, Outsiders’ Suggestions for PDDL Improvement”. In: *WS-IPC 2012* (2012), p. 43.

[16] Paula Goolkasian. “Pictures, words, and sounds: From which format are we best able to Reason?” In: *The Journal of General Psychology* 127.4 (2000), pp. 439–459.

[17] Hans Grönniger et al. “Text-based Modeling”. In: *4th International Workshop on Software Language Engineering*. 2007.

[19] Joshua T Guerin et al. “The academic advising planning domain”. In: *WS-IPC 2012* (2012), p. 1.

[20] Malte Helmert. *Understanding planning tasks: domain complexity and heuristic decomposition*. Vol. 4929. Springer, 2008.

[21] Rich Hickey. “The Clojure programming language”. In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM. 2008.

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

[23] Bowen Hui, Sotirios Liaskos, and John Mylopoulos. “Requirements analysis for customizable software: A goals-skills-preferences framework”. In: *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*. IEEE. 2003, pp. 117–126.

[24] Wonil Hwang and Gavriel Salvendy. “Number of people required for usability evaluation: the 10±2 rule”. In: *Communications of the ACM* 53.5 (2010), pp. 130–133.

[25] Okhtay Ilghami and J William Murdock. “An extension to PDDL: Actions with embedded code calls”. In: *Proceedings of the ICAPS 2005 Workshop on Plan Execution: A Reality Check*. 2005, pp. 84–86.

[26] Jeff Johnson. *GUI bloopers: don’ts and do’s for software developers and Web designers*. Morgan Kaufmann, 2000.

[27] Amit Konar. *Artificial intelligence and soft computing: behavioral and cognitive modeling of the human brain*. Vol. 1. CRC press, 1999.

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

[29] Bil Lewis, Daniel LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual*. Free Software Foundation Cambridge, MA, 1990.

[30] 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.

[31] Tim Lindholm, Frank Yellin, and Gilad Bracha. "Virtual Machine Specification". In: (2011).

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

[33] Richard J Miara et al. "Program indentation and comprehensibility". In: *Communications of the ACM* 26.11 (1983), pp. 861–867.

[34] Microsoft. *Visual Studio*. URL: `http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=EN-US&k=k(MSDNSTART)&rd=true` (visited on 04/03/2014).

[35] Douglas L Nelson, Valerie S Reed, and John R Walling. "Pictorial superiority effect." In: *Journal of Experimental Psychology: Human Learning and Memory* 2.5 (1976), p. 523.

[36] Jakob Nielsen. "Estimating the number of subjects needed for a thinking aloud test". In: *International journal of human-computer studies* 41.3 (1994), pp. 385–397.

[37] Jakob Nielsen and Rolf Molich. "Heuristic evaluation of user interfaces". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1990, pp. 249–256.

[38] 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.

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

[40] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton university press, 2008.

[41] Hajo A Reijers et al. "Syntax highlighting in business process models". In: *Decision Support Systems* 51.3 (2011), pp. 339–349.

[42] Jeff Sauro. *A practical guide to the system usability scale: Background, benchmarks & best practices*. Measuring Usability LCC, 2011.

[43] Jeff Sauro and James R Lewis. *Quantifying the user experience: Practical statistics for user research*. Elsevier, 2012.

[44] MMS Shah et al. “Knowledge engineering tools in planning: State-of-the-art and future challenges”. In: *Knowledge Engineering for Planning and Scheduling* (2013), p. 53.

[45] Mohammad M Shah et al. “Exploring knowledge engineering strategies in designing and modelling a road traffic accident management domain”. In: *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. AAAI Press. 2013, pp. 2373–2379.

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

[48] Richard M Stallman. *EMACS the extensible, customizable self-documenting display editor*. Vol. 16. 6. ACM, 1981.

[49] Margaret-Anne D Storey, Davor Čubranić, and Daniel M German. “On the use of visualization to support awareness of human activities in software development: a survey and a framework”. In: *Proceedings of the 2005 ACM symposium on Software visualization*. ACM. 2005, pp. 193–202.

[50] Flavio Tonidandel, Tiago Stegun Vaquero, and José Reinaldo Silva. “Reading PDDL, writing an object-oriented model”. In: *Advances in Artificial Intelligence-IBERAMIA-SBIA 2006*. Springer, 2006, pp. 532–541.

[52] Tiago Stegun Vaquero, Flavio Tonidandel, and José Reinaldo Silva. “The itSIMPLE tool for modeling planning domains”. In: *Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, Califormia, USA* (2005).

[53] Tiago Stegun Vaquero et al. “On the Use of UML. P for Modeling a Real Application as a Planning Problem.” In: *ICAPS*. 2006, pp. 434–437.

[55] TS Vaquero et al. “itSIMPLE4. 0: Enhancing the modeling experience of planning problems”. In: *System Demonstration–Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)*. 2012.

[56] Robert A Virzi. “Refining the test phase of usability evaluation: how many subjects is enough?” In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 34.4 (1992), pp. 457–468.

[57] Gerhard Wickler. "Using planning domain features to facilitate knowledge engineering". In: *KEPS 2011* (2011), p. 39.

[59] Hankz Hankui Zhuo et al. "Learning complex action models with quantifiers and logical implications". In: *Artificial Intelligence* 174.18 (2010), pp. 1540–1569.

Chapter 7

Appendix

7.1 General

```
; Hacker World - A realistic example
(define (domain hacker-world)
        (:requirements :typing
                       :negative-preconditions
                       :fluents)

  (:types hacker non-hacker - person
          desk chair - furniture
          laptop workstation - computer
          pizza burgers fries - food
          pepperoni supreme - pizza
          computer food person furniture software - object
          number)

    (:predicates (has ?s - software ?p - person)
                 (hungry ?p - person)
                 (vulnerable ?s - software)
                 (exploited ?s - software)
                 (distance ?o1 ?o2 - object ?x - number)
                 (location ?o - object ?x ?y - number))

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

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

        (:domain hacker-world)

        (:objects big-pepperoni - pepperoni
                gary - hacker
                gisela - non-hacker
                magicfailureapp - software)

        (:init (hungry gary)
                   (has big-pepperoni gary)
                   (vulnerable magicfailureapp)
                   (has magicfailureapp gisela))

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

Listing 17: Garys' Huge Problem

7.2 Usability Test

7.2.1 Design

S	Order			
A	*Planet Splisus*	*Logistics*	Store	Coffee
B	Store	Coffee	*Planet Splisus*	*Logistics*
C	Planet Splisus	Logistics	*Store*	*Coffee*
D	*Store*	*Coffee*	Planet Splisus	Logistics
E	*Logistics*	*Planet Splisus*	Coffee	Store
F	Coffee	Store	*Logistics*	*Planet Splisus*
G	Logistics	Planet Splisus	*Coffee*	*Store*
H	*Coffee*	*Store*	Logistics	Planet Splisus

Table 7.1: The order of tasks in the experiment

Italic: Tools part

7.3 Fragen

Fehlertyp	**Logistics**	**Coffee**
Extra Zeichen	4	4
Fehlendes Zeichen	3	3
Tippfehler	2	2
Wortverwechslung	2	2
Zeichenvertauschung	2	2
Zusatz	1	1
Variablenfehler	1	1
Klammerfehler	1	1
Fehlende Syntax	1	1
Gesamt	17	17

Table 7.2: The errors in the domains *Logistics* and *Coffee*

Chapter 8

Lösung - Planet Splisus

1. Ja, `Flipsis` sind `Splis`, `Splis` sind `Gids` und `Gids` sind `Ruffisplisus`.
2. Nein, `Schloks` sind keine `Splis`.
3. Nein, `Merle` sind `Splis` aber keine `Splus`.
4. Nein, `Spleus` sind zwar `Splos` und `Hurpf` sind `Splis`, weshalb `Spleus` und `Hurpf` verheiratet (`married`) sein könnten, aber `Hurpf` sind keine `Sipsi` und nur `Sipsi` können Waffen (`has-weapon ?h - sipsi`) besitzen. Laut der Aktion `kill` muss man zum Umbringen eine Waffe besitzen.
5. - `Ruffisplisus` - Wesen
   - `Lech` - Orte
   - `Mak` - Nahrung

Chapter 9

Lösung - Store

1. Ja, `Lulas` sind `Nulls` und `Nulls` sind `Minis`.
2. Nein, nur `Nulls` sind `Zahls`.
3. Ja, `Iltre` sind vom Typ `Lala` (`Iltre`, `Nulls`, `Minis`, `Lala`) und können an einem Arbeitsplatz (`workplace`) vom Typ `Knozi` arbeiten, da diese wiederum `Lolas` sein können (`Lola`, `Zahls`, `Knozi`)
4. Nein, denn nur ein Objekt vom Typ `Spax` kann ein Produkt am Ende eines Verkaufsvorgangs besitzen und `Kostas` sind keine `Spax`.
5. - `Lala` - Menschen
   - `Lila` - Produkte
   - `Lola` - Orte

Chapter 10

Fehlerliste - Logistics Domain

#	Zeile	Fehler	Korrektur	Fehlertyp
1	3	?logistics	logistics	extra Zeichen
2	6	:types	:typing	Wortverwechslung
3	8	:typing	:types	Wortverwechslung
4	11	car 2	car2	extra Zeichen
5	16	= true	Komplett weg	Zusatz
6	18	??veh	?veh	extra Zeichen
7	22	?tr	?t	Variablenfehler
8	24	incity	in-city	fehlendes Zeichen
9	31	n0t	not	Tippfehler
10	35	?to airport	$?to - aiport$	fehlendes Zeichen
11	42	parameters:	:parameters	Zeichenvertauschung
12	43	precondition:	:precondition	Zeichenvertauschung
13	37	ay	at	Tippfehler
14	45	)	Klammer entfernen	Klammerfehler
15	49	p	?p	fehlendes Zeichen
16	51	?p ?v	(at ?p ?v)	fehlende Syntax
17	54	?p - ?l	?p ?l	extra Zeichen

Chapter 11

Fehlerliste - Coffee Domain

#	Zeile	Fehler	Korrektur	Fehlertypen
1	3	COFFEE	(domain COFFEE)	fehlende Syntax
2	6	requirements	:requirements	fehlendes Zeichen
3	9	_	-	Zeichenvertauschung
4	10	(at ?l – location)	location	Zusatz
5	11	?coffee	coffee	extra Zeichen
6	20	$k	?k	Zeichenvertauschung
7	21	:preconditions	:precondition	Wortverwechslung
8	25	_furniture	furniture	extra Zeichen
9	27	?fu	?f	Variablenfehler
10	32	:parameters:	:parameters	extra Zeichen
11	33	änd	and	Tippfehler
12	36	location	location wurde nicht definiert	Wortverwechselung
13	42	?fromr	?from-r	fehlendes Zeichen
14	44	?tor	?to-r	fehlendes Zeichen
15	47	cjp	cup	Tippfehler
16	52	?hand-over	hand-over	extra Zeichen
17	55	Klammer zu viel	Klammer entfernen	Klammerfehler

Chapter 12

Fehleranalyse

Fehlertyp	**Logistics**	**Coffee**
Extra Zeichen	4	4
Fehlendes Zeichen	3	3
Tippfehler	2	2
Wortverwechslung	2	2
Zeichenvertauschung	2	2
Zusatz	1	1
Variablenfehler	1	1
Klammerfehler	1	1
Fehlende Syntax	1	1
Gesamt	17	17

Chapter 13

Coffee Domain

```
;;;; Coffee Domain

(define COFFEE

  (requirements
    :typing)

  (:types room - location
          robot human _ agent
          furniture door - (at ?l - location)
          kettle ?coffee cup water - movable
          location agent movable - object)

  (:predicates (at ?l - location ?o - object)
               (have ?m - movable ?a - agent)
               (hot ?m - movable)
               (on ?f - furniture ?m - movable))

  (:action boil
    :parameters (?m - movable $k - kettle ?a - agent)
    :preconditions (have ?m ?a)
    :effect (hot ?m))

  (:action grip-some
    :parameters (?m - movable ?r - robot ?f - _furniture ?l - location)
    :precondition (and (at ?l ?r)
                       (on ?fu ?m)
                       (at ?l ?f))
```

```
  :effect (and (have ?m ?r)))

(:action move
  :parameters: (?m - movable ?a - agent ?from ?to - location)
  :precondition (or (änd (at ?from ?a)
                         (at ?from ?m))
                    (and (at ?from ?m)
                         (location ?from ?a)))
  :effect (and (not (at ?from ?m))
               (at ?to ?m)))

(:action change-room
  :parameters (?from-r ?to-r - room ?a - agent)
  :precondition (at ?fromr ?a)
  :effect (and (not (at ?from-r ?a))
               (at ?tor ?a)))

(:action prep-coffee
  :parameters (?a - agent ?c - cjp ?w - water ?cof - coffee)
  :precondition (and (have ?c ?a)
                     (hot ?w))
  :effect (have ?cof ?a))

(:action ?hand-over
  :parameters (?m - movable ?a1 - agent ?a2 - agent)
  :precondition (have ?m ?a1))
  :effect (and (not (have ?m ?a1))
               (have ?m ?a2))))
```

Chapter 14

Logistics Domain

```
;;;; Logistics domain

(define (domain ?logistics)

  (:requirements
    :types)

  (:typing truck airplane motorboat - vehicle
           package vehicle suitcase furniture - thing
           airport garage station - location
           car1 car 2 car3 - vehicle
           city location thing - object)

  (:predicates (in-city ?l - location ?c - city)
               (at ?obj - thing ?l - location)
               (key ?v - vehicle) = true
               (full ?v - vehicle)
               (in ?p - package ??veh - vehicle))

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

  (:action fly
```

```
  :parameters (?a - airplane ?from ?to - airport)
  :precondition (at ?a ?from)
  :effect (and (n0t (at ?a ?from))
               (at ?a ?to)))

(:action fuel
  :parameters (?v - vehicle ?c - city ?to airport)
  :precondition (and (not (full ?v))
                     (in-city ?to ?c)
                     (at ?v ?to))
  :effect (full ?v))

(:action load
  parameters: (?v - vehicle ?p - package ?l - location)
  precondition: (and (?v ?l)
                     (at ?p ?l))
  :effect (and (ay ?p ?l)
               (in ?p ?v)))

(:action unload
  :parameters (?v - vehicle p - package ?l - location)
  :precondition (and (at ?v ?l)
                         ?p ?v)
  :effects (and (not (in ?p ?v))
                (at ?p - ?l))))
```

Chapter 15

Planet Splisus

```
(define (domain splisus)

  (:requirements :typing)

  (:types splis - gid
          spleus - splos
          schprok schlok - splus
          rud mekle - lech
          hulpf hurpf - hupf
          sipsi flipsi hupf - splis
          schmok schkok - splus
          gid splos splus - ruffisplisus
          merle - hupf
          ruffisplisus mak lech - object)

  (:predicates (father-of ?r1 - ruffisplisus ?r2 - ruffisplisus)
               (married ?s1 - splos ?s2 - splis)
               (has-weapon ?h - sipsi)
               (dead ?r1 - ruffisplisus)
               (at ?l - lech ?r - ruffisplisus))

  (:action kill
    :parameters (?l - lech ?r1 - ruffisplisus ?s - splis)
    :precondition (and (at ?l ?r1)
                       (at ?l ?s)
                       (married ?r1 ?s)
                       (has-weapon ?s))
    :effect (and (dead ?r1)
```

```
(not (married ?r1 ?s)))))
```

Chapter 16

Store

```
(define (domain store)

  (:requirements :typing)

  (:types lala lila - zahls
          blisis blusis - ultri
          iltre lula - nulls
          zahls schwinds - knozi
          minis - lala
          ultri sopple schmitzl - lila
          ultres raglos wexis - lola
          kosta - nulls
          nulls spax - minis
          lola - zahls
          knozi schmus - object)

  (:predicates (product ?k - knozi) ; Produkt
               (workplace ?l1 - lola ?l2 - lala) ; Arbeitsplatz einer Person
               (product-at ?l1 - lola ?l2 - lila) ; Ort eines Produkts
               (cashier ?k - knozi) ; Kassierer / Verkäufer
               (customer ?s - spax) ; Kunde
               (owns ?l - lila ?s - spax)) ; Eigentum

  (:action sell
    :parameters (?p - lila ?z - zahls ?l - lola ?w - wexis ?s - spax)
    :precondition (and (product ?p)
                       (cashier ?z)
                       (product-at ?l ?p)
```

```
                    (customer ?s))
    :effect (and (product-at ?w ?p)
                 (not (product-at ?l ?p))
                 (owns ?p ?s))))
```

16.1 Code

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.

```clojure
(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]))
```

```clojure
(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)))
```

```clojure
(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)))
```

```clojure
(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
     :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>))
)"
   ))
```

```clojure
(defn split-up
  "Split a \textsc{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))))
```

```clojure
(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))
```

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

```clojure
(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)) "}"))
```

```clojure
(defn hash-map->TikZ-out
  "Converts complete \textsc{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}"))
```

```clojure
(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))) "}"))
```

```clojure
(defn types-hash-map->dot-language
  "Converts a \textsc{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)))
```

```clojure
;;; Read \textsc{pddl} predicates and generate UML 'type' diagram
(defn get-types-in-predicate
  "Takes a \textsc{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)))
```

```clojure
(defn pddl-pred->hash-map-long
  "Takes a \textsc{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)))
```

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

```clojure
;;; TODO: Create short version wiht prolog predicate style
;;; e.g. at/2
(defn all-pddl-preds->hash-map-long
  "Takes a list of \textsc{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 \textsc{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))

   "}"))
```

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

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

(defn get-\textsc{pddl}-init
  "Get all predicates in a \textsc{pddl} file"
  [pddl-file]
  (first (get-\textsc{pddl}-construct 'init pddl-file)))
```

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

```clojure
(defn \textsc{pddl}->dot
  "Takes a complete \textsc{pddl} file
and generates a UML type diagram"
  [pddl-file]
  (\textsc{pddl}->dot-with-style (get-\textsc{pddl}-predicates pddl-file)
                                (get-\textsc{pddl}-types pddl-file)))
```

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

```
(defn \textsc{pddl}->dot-file-input
  "Reads \textsc{pddl} file"
  [pddl-file-name]
  (\textsc{pddl}->dot pddl-file-name))
```

;;;; math helper functions

(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$_{ONCLOSE}$)))

(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))))))))))

;; 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)))))) #+END$_{SRC}$

```
# [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:
```