

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

Volker Strobel

April 8, 2014

Contents

1	Introduction	1
2	Background and Basics	4
2.1	Analysis	6
2.2	Domain File	7
2.2.1	Domain Definition	7
2.2.2	Requirements	7
2.2.3	Types	8
2.2.4	Predicates	8
2.2.5	Actions	9
2.3	Problem File	10
2.3.1	Problem Definition	10
2.3.2	Associated Domain	11
2.3.3	Objects	11
2.3.4	Init	11
2.3.5	Goal	12
2.4	Planning	12
3	Related Work	14
3.1	PDDL Studio	14
3.2	itSIMPLE	15
3.3	PDDL-Mode for Emacs	16
3.4	Critical Review	16
4	Knowledge Engineering Tools for Artificial Intelligence Plan- ning	21
4.1	Statement of Problem	21
4.2	General Interface between PDDL and Clojure	22
4.3	Create PDDL Projects (myPDDL-new)	23
4.4	Syntax Highlighting (myPDDL-syn)	24
4.4.1	Implementation and Customization	26

4.4.2	Usage and Customization	27
4.5	Code Snippets (myPDDL-snp/)	29
4.6	Distance Calculation for PDDL Locations (myPDDL-loc) . . .	29
4.7	Type Diagram Generator (myPDDL-gen)	31
4.8	Integrated Design Environment (myPDDL-sub)	33
5	Evaluation	35
5.1	Functional Suitability	37
5.2	Design Goals	38
5.3	Empirical Study	38
5.4	User Study	39
5.4.1	Participants	39
5.4.2	Material	40
5.4.3	Method	41
5.4.4	Design	42
5.4.5	Procedure	42
5.4.6	Results	43
6	Conclusion and Future Work	45
7	Appendix	51

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 ..% 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 ..%.

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, that is a sequence of actions, to achieve a desired goal [12]. It can be both a tool to create automated systems and a means to support and understand human behavior [17]. However, the effectiveness of planning largely depends on the quality of the problem formalization [30, 44]. The Planning Domain Definition Language (PDDL) [24] is a formal language and the de facto standard for the description of planning tasks [16]. The discipline that deals with the integration of world information into a computer system via a human expert is called knowledge engineering [7]. To lay the foundations for this thesis, a basic introduction to and design principles for PDDL will be given in Chapter 2. Creating these planning task specifications is a complex task that can be error-prone and time consuming. It requires the analysis of the underlying problem itself and the systematic use of appropriate tools specialized for developing planning task specifications [30]. Therefore, developing tools for PDDL and hence facilitating the knowledge engineering process is worthwhile.

```

;; Logistics domain, PDDL 3.1 version.

(define (domain logistics-object-fluents)

  (:requirements :typing :equality :object-fluents)

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

  (:functions (city-of ?l - location) - city
              (vehicle-location ?v - vehicle) - location
              (package-location ?p - package) - place)

  (:action drive
    :parameters      (?t - truck ?l - location)
    :precondition     (= (city-of (vehicle-location ?t)) (city-of ?l))
    :effect           (assign (vehicle-location ?t) ?l))

  (:action fly
    :parameters      (?a - airplane ?l - airport)
    :effect           (assign (vehicle-location ?a) ?l))

  (:action load
    :parameters      (?p - package ?v vehicle)
    :precondition     (= (package-location ?p) (vehicle-location ?v))
    :effect           (assign (package-location ?p) ?v))

  (:action unload
    :parameters      (?p - package ?v - vehicle)
    :precondition     (= (package-location ?p) ?v)
    :effect           (assign (package-location ?p) (vehicle-location ?v)))

)

```

The modular toolkit myPDDL (*modeling efficiently* PDDL) was developed, within the scope of this thesis, in order to tackle frequent needs of knowledge engineers, like project management, efficient development, error-detection, team collaboration features and to increase the acceptance and usage of PDDL in real-world domains [30, 44]. Existing tools will be critically reviewed in Chapter 3 to set the goals for myPDDL.

myPDDL is intended to support knowledge engineers throughout the en-

tire design cycle of specifying planning tasks. In the initial stages, it allows for the creation of structured PDDL projects that should support 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 supports a faster error-detection and aids with understanding parts of planning tasks at a glance 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 supports the conversion of PDDL code into Clojure [13] code and vice versa. One use case for such an interface could be the partial automation of modeling tasks. Within this project, the interface was used for a feature that calculates distances between objects specified in a problem model. All of these features were integrated into the customizable and extensible Sublime Text [33] editor. They are described in detail in Chapter 4. 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 in Chapter 5. 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. 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 [8] 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 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. Figure ?? (TODO: Add figure) visualizes the workflow for planning in PDDL.

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 [29, 12]. For modeling in PDDL, the following six general design principles 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 inconsistencies are detected, it is advisable to restart at an earlier design step.

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.

¹More complete descriptions of PDDL, as well as formulations in Backus-Naur form (BNF) are provided by Fox and Long [8] for PDDL 2.2 and Kovacs [19] for PDDL 3.1.

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 [24]. 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 [24], further requirements have to be stated explicitly. For example, one requirement used by many planning domains [43] 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
      food person furniture software - object)
```

Listing 3: The type hierarchy for the *Hacker World*, consisting of different types of persons, furniture, computers, hackers, 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 $(p \text{ ?v1 } - \text{ t ?v2 } - \text{ t})$ is equivalent to $(p \text{ ?v1 ?v2 } - \text{ t})$.

```
(:predicates (has ?s - software ?p - person)
              (hungry ?p - person)
              (vulnerable ?s - software)
              (exploited ?s - software)
              (location ?f - furniture ?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 predicate **location** that specifies x and y coordinates for a furniture item.

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 (hungry ?p)
:effect (not (hungry ?p)))

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

;; Move a piece of furniture
(:action move
:parameters (?f - furniture ?old-x ?old-y ?new-x ?new-y)
:precondition ()
: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 ...)`.

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*².

²By 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.

```
(:init (hungry 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 ³. However, most planners only support certain subsets of PDDL and have some peculiarities ⁴. Additionally, the quality of error messages is very diverse, ranging from stating that an error occurred to displaying line number and found problem.

This thesis uses the planner SGPlan₅ [14], a planner that supports many PDDL features and has comprehensive error messages that state the actual problem ⁵.

The planner SGPlan₅ can be used by specifying the domain file and the problem file in a command line interface.

³For 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>.

⁴A short discussion on planners and their "excentricities" can be found at <http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html>.

⁵Additionally, SGPlan₅ 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>.

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.

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 [4, 28] 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 [26] 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 IT-SIMPLE [39] 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) [3] diagrams. UML was invented in order to standardize modeling in software engineering (SE) and the latest version (UML 2.4.1) [38] consists of 14 different types of diagrams divided into two larger groups: structure and behavior diagrams. In the process leading up to IT-SIMPLE, UML.P (UML in a Planning Approach) was proposed, a UML variant specifically designed for modeling planning domains and problems [40].

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} [39]. 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 be generated using PDDL. Starting in version 4.0 [42] 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 (TODO!), a Lisp dialect [35, 20]. The core values of Emacs are its extensibility and customizability. PDDL-mode [32] 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.} [25]. Furthermore, both ITSIMPLE and PDDL STUDIO specify horizontal tab sizes of about ten spaces, while two to four spaces generally seem to be adequate [25]. To have both basic editor features¹ 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 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.

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

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  (tab 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 [37] 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 [9]. 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 [8]. PDDL-mode for Emacs only works with PDDL versions up to 2.2, which introduced derived predicates and timed initial predicates [5], 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 [41]. 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 [15]. 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 Artificial Intelligence Planning

4.1 Statement of Problem

The erroneous domain in Chapter 1 and the *Hacker World* and *Gary's Huge Problem*, presented in Chapter 2, already indicated that writing and maintaining PDDL files can be time-consuming and cumbersome [21, 45]. But not only the modeling effort can be quite huge in contrast to the length of the generated output. Due to the amount of information that has to be integrated for specifying PDDL domains, files can get confusing and error-prone. PDDL's modeling capacities have been extended over the last years (source) 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 [30]. For these purposes, tools that support knowledge engineers at different design steps seems to be reasonable. The following sections will present myPDDL, an extensible, modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promote 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-gen A type diagram generator for analyzing the structure of PDDL type hierarchies.

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

myPDDL-syn A context-aware syntax highlighting feature.

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

myPDDL-sub A integrated development environment for the afore mentioned tools to be used 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 *new*, *gen* and *loc*. As stated before, myPDDL is focused on customizability and extensibility, ranging from the editor-dependent choice of key bindings and themes to adding a new module based on a general interface between PDDL and Clojure (TODO: write more beautiful).

4.2 General Interface between PDDL and Clojure

In accordance with programming languages, you might want to automate tasks, do sophisticated mathematics on parts of a planning specification or . However, PDDL's calculating capabilities are limited [27]. While these features are currently not supported by PDDL itself, pre-processing PDDL files and thereby integrate the information into the file seems to be a reasonable workaround. At this end, an interface with a programming language seems feasible. It can thus partly automate the modeling process as well as reduce the modeling time (see 4.6 the distance calculator myPDDL-loc). As PDDL is used to create more and more complex domains [10, 11].

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

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

As PDDL's syntax is inspired by Lisp [8, p. 64], using a Lisp dialect for the interface seems reasonable. This way, 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 in a convenient way, and amongst other, automated code layout and indentation. This thesis uses Clojure [13], a modern Lisp dialect that runs on the Java Virtual Machine (JVM) [22].

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

The interface is built on two methods:

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

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

The *read-construct* method thereby makes use of an safe reader that avoids that read-in constructs can be executed.

Once a part is extracted and represented in Clojure, the processing possibilities are manifold and the full capacities of Clojure can be used.

The interface is provided as a library, so that its methods can be included in any Clojure file.

4.3 Create PDDL Projects (myPDDL-new)

In many cases, creating PDDL domains is kind of an ad-hoc process [31]. However, prior to each implementation of a PDDL task specification stands the creation of a new *project*, consisting of at least one domain and a belonging problem file. As several team members may be working on this project, "Have a place for every thing, and keep every thing in its proper place" would be desirable. To this end, a standardized project folder structure could facilitate, collaboration, maintaining and and keep consistency over projects ... and the organization.

myPDDL-new creates PDDL project folders, on the basis of a project name (Figure 4.9).

In this project folder, the domain file `domain.pddl` and the problem file `p01.pddl` (in the folder `problems`) are filled with basic PDDL skeletons (TODO: remove this sentence or add functionality or even better: specify a template, which can be added!). The templates for the skeletons are located in the `myPDDL` folder and be customized to your need by editing and saving the template.

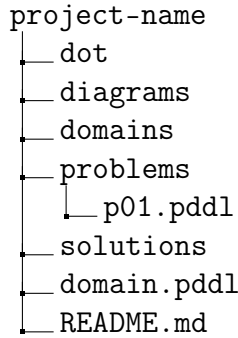


Figure 4.2: The project folder structure created by myPDDL-new. The project-name is chosen by the user and used for the name of the created domain.

The **domains**, **dot** and **diagrams** folders are created for the use with *myPDDL-gen*, which will save its generated output to these folders and thereby allows for a basic version control system (see 4.7Type Diagram Generator (myPDDL-gen)).

As one domain file can have multiple problem files, the **problems** folder is designed for the collection of all associated problem files.

Recognizing, that most knowledge engineers do not write any documentation related to the specified planning task ([30]), **README.md** is a Markdown (a plain text formatting syntax) file, which is, amongst 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. This way, this file can be used as an quick overview for PDDL projects, located at hosting service.

This approach should support an structured and organized design process. The choice of a folder structure (instead of a project file) has the advantage of being readable and customizable independently of the editor. So the need for team work [30] is tackled by using a structured project folder where and changes can be seen in the view of the diagram. This directory organization is intended to contain a single or just a few domain files in one project, stored in the project root directory, while problem files are stores in the subfolder problems.

4.4 Syntax Highlighting (myPDDL-syn)

* Statement of Problem

[b].5 A

Figure 4.3: A subfigure

[b].5 B

Figure 4.4: Another subfigure

Figure 4.5: A figure

Writing and maintaining PDDL files is an ongoing process. Continually growing, PDDL files can span several pages and contain hundreds or thousands of lines of code. In order to recognize parts of the file quickly and detect errors at a glance, distinguishing code parts by color seems to be a reasonable way for aiding the designer.

myPDDL-syn distinguishes comments, variables, subtypes, types, keywords, inbuilt-functions and highlights them in different colors by a sophisticated pattern matching heuristic that can both recognize the start and the end of a larger code block (like `:predicates ...`). This way, knowledge engineers can skim through the code and ignore parts of the code.

While syntax highlighting can be helpful in get along inside code and keep an overview, it could be especially powerful, if syntax errors could get quickly detected. Missing brackets or expression at the wrong position are possible sources of error. Knowledge Engineers can ignore larger parts of the code (e.g. comments) that are visually distinct.

Figure XYZ displays an extract of the *Coffee* world, a domain used in the user study for the evaluation of this tool.

By using the inbuilt Sublime Text color scheme *Monokai*, almost all proper PDDL constructs get highlighted in colors according to their meaning. Constructs written in white usually contain errors, are written at the wrong place or are not specified by PDDL 3.1. This way correct constructs and errors are visually distinct. Highlighted code could nevertheless contain semantic errors. This was tested by means of ...

Longer files can get quickly confusing. Therefore, it is convenient to have a tool that supports editing these files. The syntax highlighting feature displays code in different colors according to the category of terms. In order to facilitate editing PDDL files, a syntax highlighting plug-in for the Sublime Text editor [33, 34] is proposed.

For this reason, the implemented highlighter makes use of scopes, so that identify code parts by begin and end markers.

4.4.1 Implementation and Customization

For the ease of creation, the PDDL syntax highlighter is implemented by the use of the ST plug-in *AAAPackageDev* [1]. So, the definitions can be written in YAML in converted to Plist XML later on. *AAAPackageDev* [1] is a ST plugin, that helps to create, amongst others, ST packages, syntax definitions and 'snippets' (re-usable code).

By means of Oniguruma regular expressions [18], scopes are defined, that determine the meaning of the PDDL code block. ST themes highlight different parts of the code by the use of scopes. Scopes are defined by the use of regular expressions (regexes) in a tm-Language file. The scope naming conventions mentioned in the *TextMate 2 Manual* are applied here. By the means of the name, the colors are assigned according to the current used ST theme. That means that colors are not assigned per se, but dependently on the current scheme. Through that, experienced users can use their default theme and all can easily change the colors by changing the scheme. Different ST themes display different colors (not all themes support all naming conventions).

The syntax highlighting is intended for PDDL 3.1, but is backward compatible to previous versions. It's based on the Backus-Naur Form (BNF) descriptions, formulated in Kovacs [19], Fox and Long [8], and McDermott et al. [24].

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

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

The first part of the PDDL.YAML-tmlanguage describes the parts of the PDDL task that should be highlighted. By removing (or commenting) include statements, the syntax highlighter is adjustable the user's need.

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

```

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

```

Line 20, Column 22 Spaces: 2 PDDL

Figure 4.6: Coffee domain with and without syntax highlighting

4.4.2 Usage and Customization

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

```
sus-eps-converted-to.pdf
```

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

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

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

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

and starts to model his world. To this end, he uses the code snippets `domain` for creating the domain skeleton, navigates inside the domain file with `,` creates new type definitions with the snippets `t2` and `t3`. After completing his first draft, he presses `f8`, for saving his file and displaying the PDDL type diagram and sees the following diagram:

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

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

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

A final check show that everything is as expected:

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

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

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

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

```
$ sublime p01.pddl
```

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

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

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

4.5 Code Snippets (myPDDL-snp/)

Consider again the basic skeleton of an action:

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

Almost all PDDL actions consist of these same sections. Writing and extending pddl files, knowledge engineers are supposed to use the same constructs again and again. This is where code snippets come in. To facilitate and fasten the implementation of standard constructs, my-PDDL-snp provides code snippets. These snippets are templates for often used pddl constructs, like domain and problem definitions, predicates and actions. They can be inserted by typing a trigger keyword. Typing **action** and pressing the \rightarrow key, would exactly insert the action specified in Listing xyz.

Having inserted the skeleton, it still has to be filled in. That means a means for easily navigating inside the code snippet would become handy. For this purpose, the blanks can be filled by pressing \rightarrow 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. 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)).

p2hasspp

Figure 4.7: Example for the use of snippets. =p2= creates a binary predicate template that can be filled in.

Every snippet is stored in a separate file, located in the PDDL/ folder. New snippets can be added and existing snippets can be customized there (change the template or change the trigger keyword).

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 that includes the square root function (`\sqrt`). An Euclidean distance function that uses the square root would be convenient for distance modeling and measurement. However, PDDL 3.1 supports only four arithmetic operators (`+`, `-`, `/`, `*`). However, PDDL does only support basic arithmetic operations (`+`, `-`, `/`, `*`). These operators can be used in preconditions, effects (normal/continuous/conditional) and durations. Parkinson and Longstaff [27] describe a workaround for this drawback. By declaring an action ‘calculate-sqrt’, they bypass the lack of this function and rather write their own action that makes use of the Babylonian root method. While the square root could be approximately determined using the Babylonian method, requiring many iterations, this method would most likely have an adverse effect on plan generation [27].

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

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

However, this approach has certainly a major drawback, apart from the time required to calculate (non-used) distances. If the number of locations is n , the number of calculated distances is n^2 , as every location has a distance to every other. The calculated distances have to be stored in the PDDL problem file, so that possibly a lot of space is used. This is why it seems to be reasonable to extend `textsc{pddl}`’s mathematical operations. In accordance with Parkinson and Longstaff [27], extending a possibility would be a PDDL A 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 11: Problem file before using myPDDL

```
(: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 12: After

4.7 Type Diagram Generator (myPDDL-gen)

As stated by the adage "A picture is worth a thousand words" graphical representations can simplify textual representations. In a computer science, the graphical visualization of textual information should simplify the communication and collaboration between developers and help to quickly grasp the connection of hierarchical structured items [36].

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 understand, use and extend available domains, a crucial part is grasping involved types, understand their hierarchy, and identify the constructs that make use of them. However, this can be difficult by just reading PDDL the textual representation of the hierarchy, so a diagram that displays this hierarchy could be helpful.

Creating this diagram manually each time a change is made can be unhandy, takes time and can be a source of errors. To this way, an automated graphical representation, based on a PDDL file could save time and energy.

myPDDL-gen serves this purpose and generates and displays diagrams by means of domain files. Figure xyz shows the automatically generated diagram from the *Hacker World* in Chapter 2. In the diagram, types are represented with boxes, whereby every box consists of two parts:

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

Generalization relationships ("is a", for example "a laptop *is a* computer") are expressed by arrows from the subtype (here: *laptop*) to the super type, where the arrow head aims at the super type (here: *computer*). This

relationship expresses, that every subtype is also an instance of the illustrated super type.

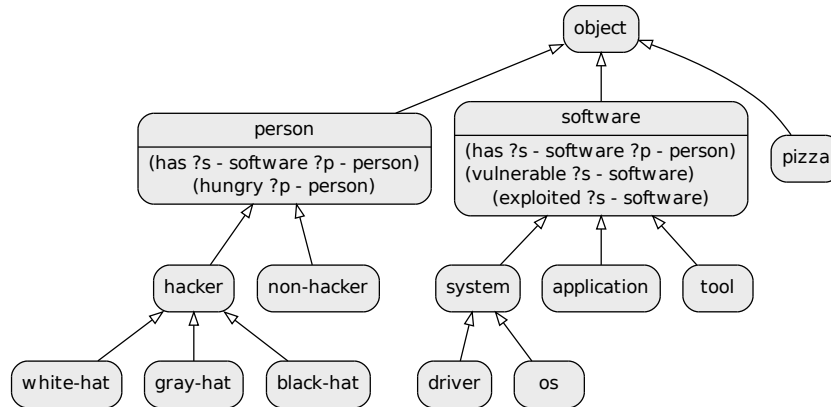


Figure 4.8: The type diagram that was generated from the *Hacker World* using myPDDL-gen.

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

Subsequently, the diagram is generated using `dot` from the Graphviz package [6], 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 (`digraph`) 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` and a PNG diagram is created and saved in the folder `diagrams/`. Additionally, the diagram will be immediately opened and displayed in a window. In addition, a copy of the domain file is stored in the folder `domains/`. Every time *myPDDL-gen* is invoked, these steps are executed and the saved file names are extended by a ascending revision number. Thus, one cannot only identify associated PDDL, DOT and PNG files, but also use this feature for basic revision control. Figure xyz displays the folder structure after invoking *dia* twice on the *Hacker World*. This way, the type hierarchy and predicate structure of a previous version of a domain file can

```

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 4.9: Folder structure after two invocations of `textitmyPDDL-gen`. Files and folders are automatically created and extended by a revision number (`=0=`, `=1=`) each time `/gen/` is used.

be identified by the correspondent type diagram (both files have matching revision numbers), and one can revert to a previous revision, stored in the `domains/` folder. All folders are created if necessary.

4.8 Integrated Design Environment (myPDDL-sub)

The so far presented tools provide a command-line interface for interacting with the user. This offers a high flexibility, the possible automation of jobs by using scripts, and the possibility for a integration in different software. However, the user has to be familiar with the underlying syntax, in order to use the full spectrum of available functions. By using ST as editor, language independent ST features are supported, like auto completion of words already used in this file, code folding and column selection, described in the Sublime Text 2 Documentation. Sublime Text is used to combine the so far presented command-line tools, as well as the syntax highlighter and the code snippets into an IDE. While *snp* and *syn* are devised explicitly for ST and therefore integrated from the outset, the other tools (*new*, *gen*, *loc*) can be used independently of ST utilizing the command-line interface and any PDDL file. To provide an IDE for using myPDDL, *-sub* integrates *new*, *gen* and *loc*, aiming at 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 for myPDDL-new *PDDL: Create Project* 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 or by placing the files of myPDDL¹ in the packages folder of Sublime Text ². Following, the features can be activated by changing Sublime Text's syntax to PDDL (View->Syntax->\pddl).

¹The files can be downloaded from <https://github.com/Pold87/ba-thesis/>.

²Further information about Sublime Text packages can be found at <http://www.sublimetext.com/docs/3/packages.html>.

Chapter 5

Evaluation

TODO: Spannender Einleitungssatz To evaluate a software means to assess its quality. Even though there exists an international standard (ISO/IEC 25010) to evaluate software, most of the eight characteristics that serve as quality criteria are irrelevant for this project. More appropriate criteria are supplied by Shah et al. [30] who evaluate different knowledge engineering tools in planning, including, among others, ITSIMPLE. All in all, they identified seven criteria, the lead questions of which can be found in table 5.1 5.1.

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?
Experience	Does the tool induce users to produce documentation? 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 depend 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. [30]

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 depend 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?

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?
Experience	Does the tool induce users to produce documentation?
	Is the method/tool indicated for inexperienced users?
	Do users need to have good knowledge of PDDL?
Efficiency	Is it able to support users and to hide low level details?
	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?

The first question that defines operationality ("How efficient are models produced?") is equivalent to the criterion efficiency and the second question was not of interest when developing myPDDL, since it can be reduced to the question of whether planners perform well on standard PDDL files. Therefore, it was decided to replace the criterion operationality with functional suitability from the ISO/IEC 25010 standard. To assess the functional suitability and to illustrate where myPDDL fits in with similar tools, it was considered appropriate to compare it 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 XX, collaboration, experience, and debugging were tested with a user test. The other three criteria, maintenance, efficiency, and support, will simply be discussed.

5.1 Functional Suitability

To assess the functional suitability, it was decided to compare myPDDL to the three tools already discussed in related work: PDDL-Studio, itSIMPLE,

and PDDL-mode for Emacs. This is 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 different stages of the process to different degrees. They do this via different features. However, sometimes knowledge engineers may only have to alter or develop already existing models further. myPDDL aims to also assist in the objective to quickly understand foreign code. Table XY is to illustrate how the four tools compare in terms of features and how each of these features is helpful in the knowledge engineering process.

5.2 Design Goals

	PDDL STUDIO	ITSIMPLE	PDDL-mode	myPDDL
latest supported version	PDDL 1.2	PDDL 3.1	PDDL 2.2	pddl 3.1
syntax highlighting	Yes	Yes	Yes	Yes
syntactic error detection	Yes	No	No	By Context
semantic error detection	Yes	No	No	No
code completion	Yes	No	Yes	Yes
code snippets	No	Yes	Yes	Yes
code folding	Yes	No	Yes	Yes
project management	Yes	Yes	No	Yes
visualization feature	No	Yes	No	Yes
planner integration	Basic	Yes	No	Yes?
automatic indentation	No	No	Yes	Yes
customization features	No	No	extensive	extensive

5.3 Empirical Study

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

The consistency and capability to highlight every PDDL construct in a

color according to its meaning, were checked by 320 (syntax error-free) PDDL files, consisting of 87 domain and 230 problem files (list of files). In that, no inconsistencies nor non-highlighted words could be found.

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

5.4 User Study

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. The best 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.

5.4.1 Participants

A total of eight participants (two female participants, Meanage=23, SDage=2 (TODO: Update)) took part in this usability test. 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 project, but testing more than the common five participants (Nielsen, J. Estimating the number of subjects needed for a thinking aloud test. *International Journal of Human-Computer Studies* 41, (1994), 385-397.9. Nielsen, J. and Molich, R. HE of user interface. In *CHI '90 Conference Proceedings*. ACM, 1990, 249-256. Virzi, R.A. Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors* 34, (1992), 457-468.) is in line with more recent research (Faulkner, L. (2003). Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3), 379-383.; Hwang, W., & Salvendy, G. (2010). Number of people required for usability evaluation: the 10 ± 2 rule. *Communications of the ACM*, 53(5), 130-

133.). 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.

5.4.2 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 myPDDL was recorded to familiarize participants with the topic. A preliminary questionnaire was designed to assess the prior experience with planning and Sublime Text. The system usability scale (QUELLE) 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. The times that participants took to answer questions were recorded with the web site online-stopwatch.chronme.com as this allowed splitting the total 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 (SH) and the type diagram (TD) 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 for each of these two types to compare the effects of having the tools available. For these four tasks, domains (matched in difficulty) and instructions were written. The two tasks to test syntax highlighting presented the user with domains that were 54 lines in length (WORTLAENGE) and contained 17 errors each. Errors were distributed evenly throughout the domains and were categorized into different types (WHAT TYPES) and the occurrence frequencies of these types were matched across domains as well, to ensure equal difficulty for both domains. To test the type diagram generator, two fictional (WHAT IS FICTIONAL IN THIS CASE) domains with equally complex type hierarchies (approximately same (5/6 layers) depth, approximately same number (20/21) of types) consisting of non-words were designed. The domains were also matched in length and overall complexity (same number (1) of actions, same number (4) of preconditions in action, approximately same number (2/3) of effects in action, approximately same number (5/6) of predicates with the same arity distributions). All questionnaires and task descriptions can be found in Appendix

X. Lastly, participants were given pen and paper to help them solve tasks if they saw a need for it.

5.4.3 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 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 had any questions concerning the tutorial and if they thought that they had understood everything. If they had not yet watched it at home, they proceeded to do so. After the tutorial, they were asked to complete the tasks in the order specified in table XY in Appendix Y. 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 an SH task or a TD task. For the SH tasks, participants were given six minutes (a reasonable time frame tested on two pilots) 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 them in the code if they knew how to, but not to dwell on the correction if they did not. For the TD 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, once they figured it out. They were not told that the time it took them to come up with an answer was recorded, since this knowledge could make them feel pressured and lead to more false answers. Once the participants had completed all four tasks, they were asked to evaluate the perceived usability of myPDDL using the SUS.

5.4.4 Design

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

Italic: Tools part

5.4.5 Procedure

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

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

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

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

The participants did not and that there will be a *tools* part. Immediately before the tools part, a three minute video introduction to the functionality of the syntax highlighter (myPDDL-syn) and the usage of (myPDDL-gen) was given. Directly after his, participants were asked to work on the tools parts. so that they faced the tools were not confronted with the tools before the actual test.

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

5.4.6 Results

1. Syntax Highlighting Tasks
2. Type Diagram Tasks

Syntax Highlighting Tasks Type Diagram Tasks Diagram XY shows the geometric mean of the task completion time (this excludes participants that did not complete the task. i.e. gave an incorrect answer) for the two domains, Planet Splisus and Store, with and without the type diagram generator for each question. It is evident, that when having the type diagram generator available, participants answer each question faster. However, subjects' time on question with the Planet Splisus domain is on average 37 seconds faster with the TDG than without it, while with the Store domain, this time difference only accumulates to 18 seconds, indicating an ordinal interaction effect of domain and tool availability. The ordinal interaction in this case means that the beneficial effect of having tools available depends on the domain at hand. Assuming that the domains and questions were successfully matched in difficulty, this interaction effect may be due to...

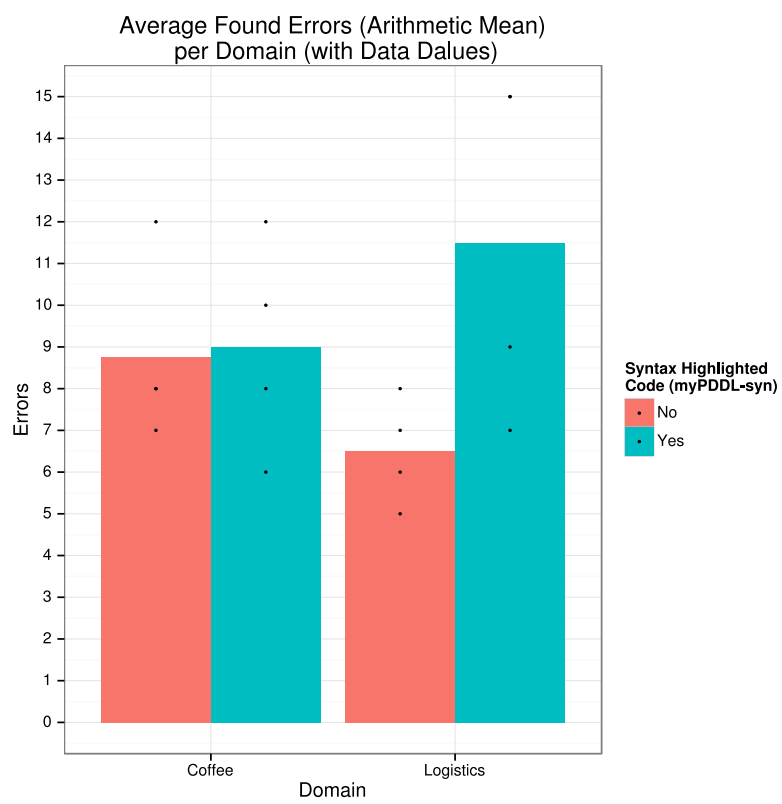


Figure 5.1: The Domain Planet Splisus.

Chapter 6

Conclusion and Future Work

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

Besides that, interactive planning

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

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

The plug-in for the editor ST could be further extended to provide features of common integrated developing environments (IDE). A build script for providing input to a planner for auto-matching domain and matching problem(s) (or problem and matching domain) in ST could be convenient.

Detecting semantic errors besides syntactic errors \{ plch2012inspect} Studio could be the next step to detecting errors fast and accurate. Possible semantic errors could be undeclared variables or predicates in a domain specification.

In the diagram, predicates are only added to the types that are explicitly mentioned in the argument of the predicate. However, as subtypes of types declared in the predicate arguments, can also be used as argument to the predicate, this means, that all specializations of a type can also be used for this predicate. This can be seen in Figure xyz ...:For example, a the PDDL domain file could declare (**hungry** ?p - **person**), although men and women can be hungry.

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.

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

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

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

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

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

A faster understanding of the domain structure could be beneficial for the maintenance and application of existing domains and problems, and, possibly for the communication between engineers. Finally, real world usage of PDDL can be promoted so that the focus of artificial intelligence planning can also be shifted towards the design of plans, following the citation "Plans are worthless, but planning is everything".

Bibliography

Paper Sources

- [1] *AAAPackageDev*. 2014. URL: <https://github.com/SublimeText/AAAPackageDev/> (visited on 02/24/2014).
- [2] Rajendra Akerkar. “Intelligent Systems: Perspectives and Research Challenges”. In: *CSI Communications* (2012), p. 5.
- [3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide*. Pearson Education India, 1999.
- [4] Miroslav Chomut. “Tool for editing PDDL projects”. In: (2012), plch.
- [5] 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).
- [6] John Ellson et al. “Graphviz—open source graph drawing tools”. In: *Graph Drawing*. Springer. 2002, pp. 483–484.
- [7] Edward A Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley Reading, 1983.
- [8] 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.
- [9] Stanley Gill. “The Diagnosis of Mistakes in Programmes on the ED-SAC”. In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 206.1087 (1951), pp. 538–554.
- [10] Robert P Goldman and Peter Keller. ““Type Problem in Domain Description!” or, Outsiders’ Suggestions for PDDL Improvement”. In: *WS-IPC 2012* (2012), p. 43.
- [11] Joshua T Guerin et al. “The academic advising planning domain”. In: *WS-IPC 2012* (2012), p. 1.

- [12] Malte Helmert. *Understanding planning tasks: domain complexity and heuristic decomposition*. Vol. 4929. Springer, 2008.
- [13] Rich Hickey. “The Clojure programming language”. In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM. 2008.
- [14] Chih-Wei Hsu and Benjamin W Wah. “The sgplan planning system in ipc-6”. In: *Proceedings of IPC*. 2008.
- [15] 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.
- [16] 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.
- [17] Amit Konar. *Artificial intelligence and soft computing: behavioral and cognitive modeling of the human brain*. Vol. 1. CRC press, 1999.
- [19] DL Kovacs. “BNF definition of PDDL 3.1”. In: *Unpublished manuscript from the IPC-2011 website* (2011).
- [20] Bil Lewis, Daniel LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual*. Free Software Foundation Cambridge, MA, 1990.
- [21] 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.
- [22] Tim Lindholm, Frank Yellin, and Gilad Bracha. “Virtual Machine Specification”. In: (2011).
- [23] MacroMates Ltd. *TextMate 2 Manual*. 2013. URL: <http://manual.macromates.com/en/> (visited on 02/24/2014).
- [24] Drew McDermott et al. “PDDL-the planning domain definition language”. In: (1998).
- [25] Richard J Miara et al. “Program indentation and comprehensibility”. In: *Communications of the ACM* 26.11 (1983), pp. 861–867.
- [26] Microsoft. *Visual Studio*. URL: [http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=EN-US&k=k\(MSDNSTART\)&rd=true](http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=EN-US&k=k(MSDNSTART)&rd=true) (visited on 04/03/2014).

- [27] 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.
- [28] Tomas Plch et al. “Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio”. In: *and Exhibits* (2012), p. 15.
- [29] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton university press, 2008.
- [30] 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.
- [31] 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.
- [33] Jon Skinner. *Sublime Text 2*. <http://www.sublimetext.com/2>. Release 2.0.2. 2013.
- [34] Jon Skinner. *Sublime Text 3*. <http://www.sublimetext.com/3>. Beta Build 3059. 2013.
- [35] Richard M Stallman. *EMACS the extensible, customizable self-documenting display editor*. Vol. 16. 6. ACM, 1981.
- [36] 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.
- [37] 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.
- [39] 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, California, USA* (2005).

- [40] 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.
- [42] 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.
- [43] Gerhard Wickler. “Using planning domain features to facilitate knowledge engineering”. In: *KEPS 2011* (2011), p. 39.
- [45] Hankz Hankui Zhuo et al. “Learning complex action models with quantifiers and logical implications”. In: *Artificial Intelligence* 174.18 (2010), pp. 1540–1569.

Website Sources

- [18] K. Kosako. *Oniguruma Regular Expressions Version 5.9.1*. 2007. URL: <http://www.geocities.jp/kosako3/oniguruma/doc/RE.txt> (visited on 02/24/2014).
- [32] Surendra Singhi. *Emacs mode for PDDL*. 2005. URL: <http://rakaposhi.eas.asu.edu/planning-list-mailarchive/msg00085.html> (visited on 04/03/2014).
- [38] *UML 2.4.1*. URL: <http://www.omg.org/spec/UML/2.4.1/> (visited on 04/03/2014).
- [41] TS Vaquero et al. URL: <https://code.google.com/p/itsimple/downloads/list> (visited on 04/03/2014).
- [44] *WS 5: Knowledge Engineering for Planning and Scheduling (KEPS)*. 2014. URL: http://icaps14.icaps-conference.org/workshops_tutorials/keps.html (visited on 04/03/2014).

Chapter 7

Appendix

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

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

```

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

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

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

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

52
(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

```



```

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

patterns:

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

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

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

repository:

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

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

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

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

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

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

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

```

