MyPDDL: Tools for efficiently creating PDDL domains and problems

Volker Strobel[1] [0000−0003−2974−9827] and Alexandra Kirsch[2] [0000−0002−5663−1798]

[1] IRIDIA, Université Libre de Bruxelles, Belgium
[2] Independent Scientist
vstrobel@ulb.ac.be

Abstract. The Planning Domain Definition Language (PDDL) is the state-of-the-art language for specifying planning problems in artificial intelligence research. Writing and maintaining these planning problems, however, can be time-consuming and error prone. To address this issue, we present myPDDL—a modular toolkit for developing and manipulating PDDL domains and problems. To evaluate myPDDL, we compare its features to existing knowledge engineering tools for PDDL. In a user test, we additionally assess two of its modules, namely the syntax highlighting feature and the type diagram generator. The users of syntax highlighting detected 36 % more errors than non-users in an erroneous domain file. The average time on task for questions on a PDDL type hierarchy was reduced by 48 % when making the type diagram generator available. This implies that myPDDL can support knowledge engineers well in the PDDL design and analysis process.

Keywords: PDDL · Planning · Knowledge Engineering

1 Introduction

Being a key aspect of artificial intelligence, *planning* is concerned with devising a sequence of actions to achieve a desired goal [10]. It can be both a tool to create automated systems and a means to support and understand human behavior [14]. However, the effectiveness of planning largely depends on the quality of the problem formalization [23,1]. To ensure a standardized modeling format, PDDL (*Planning Domain Definition Language*) [16] was developed and has become the de facto standard for the description of planning tasks [13]. The discipline that deals with the integration of world information into a computer system via a human expert is called knowledge engineering [7]. While automated planning could save vast amounts of time if everything works as intended, creating the planning task specifications is a complex task that can be error-prone and cumbersome.

In this chapter, we describe myPDDL (Figure 1), a knowledge engineering toolkit that supports the knowledge engineer in the entire design cycle of specifying planning tasks. In the initial stages, it allows for the creation of structured PDDL projects that should encourage a disciplined design process. With the help of snippets, that is, code templates, often used syntactic constructs can be inserted into PDDL files. A syntax highlighting feature that speeds up the error-detection can come in handy in intermediate stages. Understanding the textual representation of complex type hierarchies in domain files can be confusing, so an additional tool enables their visualization. PDDL's limited modeling capabilities were bypassed by developing an interface that converts PDDL code into code of the functional programming language Clojure [11] and vice versa. Within this project, the interface was employed for a feature that calculates distances between objects specified in a problem model, but the interface provides numerous other possibilities and could also be used to further automate the modeling process. All of the features were integrated into the customizable and extensible Sublime Text [2] editor. Since the main aim in the development of the toolkit was for it to be easy to use and maintain, it is evaluated with regard to these criteria. The usability was assessed by means of a user test with eight subjects that had no prior experience with AI planning. The results indicate that both error-detection and the understanding of a given domain can be facilitated by myPddl. This chapter is an extended version of work published in a previous paper [25].

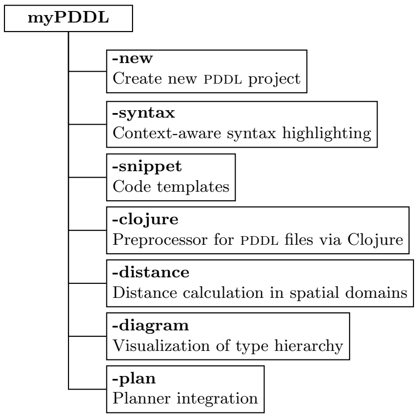

myPDDL
-new
Create new PDDL project
-syntax
Context-aware syntax highlighting
-snippet
Code templates
-clojure
Preprocessor for PDDL files via Clojure
-distance
Distance calculation in spatial domains
-diagram
Visualization of type hierarchy
-plan
Planner integration

Fig. 1: myPddl is a highly customizable and extensible modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promoting the collaboration between knowledge engineers and the use of PDDL in real-world applications. It consists of the parts shown in the figure.

The remainder of this chapter is structured as follows. Section 2 compares PDDL knowledge engineering tools to lay a foundation for myPddl. Section 3 describes the different modules of myPddl and their design principles. Section 4 evaluates myPddl via a user test. Section 5 concludes the chapter and outlines future work.

2 Related Work

This section introduces, compares, and discusses knowledge engineering tools that allow text-based editing of PDDL files to set the stage for MYPDDL (Table 1).

PDDL STUDIO [?,20] is an IDE (*integrated development environment*) for creating and managing PDDL projects, that is, a collection of PDDL files. 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 syntactic (missing keywords, parentheses, etc.) and semantic (wrong type of predicate parameters, misspelled predicates, etc.) errors. This means that PDDL STUDIO can detect errors due to a mismatch between domain and problem file in real time. The code completion feature offers recommendations for standard standard PDDL constructs as well as for previously used terms. Code folding allows the knowledge engineer to hide currently not needed code blocks. In this case only the first line of the block is displayed. All these above mentioned features of PDDL STUDIO utilize the parser. To keep track of all files, to display them in a tree structure, and to save them upon compilation, a project manager is integrated. Furthermore, this feature is necessary to detect interfile errors and to use the code completion functionality. Lastly, a command-line interface allows the integration of planners in order to run and compare different planning software.

Unlike PDDL STUDIO, which provides a text based editor for PDDL, the ITSIMPLE [27] editor has, as its main feature, a graphical approach that allows designing planning tasks in an object-oriented approach using UML (*Unified Modeling Language*). In the process leading up to ITSIMPLE, UML.P—UML in a Planning Approach—was proposed, a UML variant specifically designed for modeling planning domains and problems [26].

The main purpose of ITSIMPLE is supporting knowledge engineers in the initial stages of the design phase by making tools available that help transform the informality of the real world to 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" [27]. 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. Compared to PDDL STUDIO, ITSIMPLE has a different approach to planner integration: domain and problem can be fed to the planner by pressing a button, whereas in PDDL STUDIO, the user has to know and input commands in a command-line interface.

With itSimple, it is possible to directly input the domains and problems into a planner and to inspect the output from the planning system using the built-in plan analysis. This consists of a plan visualization, which shows the interaction between the plan and the domain by highlighting every change caused by an action. itSimple's modeling workflow is unidirectional as changes in the pddl domain do not affect the uml model and uml models have to be modeled manually, meaning that they cannot by generated using pddl. Starting in version 4.0 [28] itSimple expanded its features to allow the creation of pddl projects from scratch (i.e. without the 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.

Starting in version 4.0, itSimple expanded its features to allow the creation of pddl projects from scratch, that is, without the uml to pddl translation process [28]. So far, a basic syntax highlighting feature recognizes pddl keywords, variables, and comments. itSimple also provides templates for pddl constructs, such as requirement specifications, predicates, actions, initial state, and goal definitions.

pddl-mode[3] for Emacs builds on the sophisticated features of the widely used Emacs editor and uses its extensibility and customizability. pddl-mode 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, 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.

pddl-mode for Emacs supports pddl versions up to 2.2, which includes derived predicates and timed initial predicates [5], but does not recognize later features like object-fluents.

The online tool editor.planning.domains allows for editing pddl files in a web browser. Its feature comprise syntax highlighting, code folding, pddl-specific auto-completion, and multi-tab support. The editor is part of the Planning.Domains initiative which aims at providing three pillars to the planning community: (1) an API to access existing pddl domains and problems; (2) a planner-in-the-cloud service; and (3) an online PDDL editor. The online editor is also connected to the planner-in-the-cloud.

editor.planning.domains's syntax highlighting features are basic; so far, it does not support context-aware syntax highlighting: for example, editor.planning.domains highlights pddl key words even if they are misplaced. In contrast, myPddl is aware of the scope of a construct and highlights key words accordingly.

The PDDL plugin vscode-pddl[4] for the editor Visual Studio Code offers a wide range of editing functions, such as syntax highlighting, code completion,

[3] http://rakaposhi.eas.asu.edu/planning-list-mailarchive/msg00085.html
[4] https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl

and code snippets. It offers a mature planner integration and plan visualization. Thanks to a PDDL parser integration, it is possible to detect semantic errors immediately when they are made.

2.1 Critical Review

All the above-mentioned tools provide environments for the creation of PDDL code. Their advantages and disadvantages are reviewed in this section. At the end of each discussed feature, the approach that was used in MYPDDL is introduced.

PDDL STUDIO, ITSIMPLE, and editor.planning.domains for the most part do not build on existing editors and therefore cannot fall back on refined implementations of features, such as selection of tab size, defining custom key shortcuts, customizing the general look and feel, and bracket matching. In contrast, vscode-PDDL is integrated into the mature editor Visual Studio Code and can be used in combination with other plugins. To have both basic editor features 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 (up to PDDL 2.2) and editor.planning.domains (up to PDDL 3.1), and vscode-PDDL (up to PDDL 3.1) keywords, 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 and Visual Studio Code. editor.planning.domains provides two fixed color schemes. 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. 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.

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. 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. Both editor.planning.domains and vscode-PDDL provide many code snippets. 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.

PDDL STUDIO, PDDL-mode for Emacs, and editor.planning.domains do not provide 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. vscode-PDDL does not provide domain visualizationa but is able to visualize a found plan.

Searching for errors can be one of the most time consuming parts of the design process. Hence, any tool that is able to help detect errors faster is of great value to the knowledge engineer. While PDDL-mode for Emacs, ITSIMPLE, and editor.planning.domains facilitate error detection only by basic syntax highlighting, both PDDL STUDIO and vscodepddl are able to detect errors via a PDDL parser. In MYPDDL, syntactic errors are not be highlighted by the syntax highlighting feature, while all correct PDDL code is to get highlighted.

A major drawback of PDDL STUDIO and PDDL-mode for Emacs especially is that they are not updated regularly to support the most recent PDDL versions. PDDL STUDIO's parser is only able to parse PDDL 1.2 [?], while the latest PDDL version is 3.1. PDDL has significantly 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 [?]. PDDL-mode for Emacs is only compatible 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*. It follows that the range of functions specified in the domain file cannot include object-types in addition to numbers. ITSIMPLE, editor.planning.domains, vscode-PDDL, and MYPDDL support the latest PDDL version.

An important features of any software is the possibility of extending and customizing it [?]. PDDL STUDIO falls short of satisfying this requirement as the customization options 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. Since both Emacs and VS Code are established editors, PDDL-mode and vscode-PDDL are 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 offers such features as customizable key bindings, display of line numbers, and multi-line selection.

All in all, MYPDDL must be understood as complementary to the other existing knowledge engineering tools. MYPDDL is distributed as a package for Sublime Text and provides context-aware syntax highlighting, code snippets, syntactic error detection, and type diagram visualization. Additionally, it allows for the automation of modeling tasks due to an interface with Clojure that supports the conversion of PDDL code into Clojure code and vice versa. Therefore, MYPDDL is intended to support 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, the visualization capabilities of MYPDDL are meant to facilitate collaboration among knowledge engineers.

3 MyPDDL

MYPDDL is a highly customizable and extensible modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promoting the collaboration between knowledge engineers and the use of PDDL in real-world applications. The modules of MYPDDL are described in the next section.

3.1 Modules

myPDDL-IDE is an integrated development environment (IDE) for the use of MYPDDL in the text and code editor *Sublime Text*[5]. Since MYPDDL-SNIPPET and -SYNTAX are devised explicitly for Sublime Text, their integration is implicit. The other tools described below (MYPDDL-new, -diagram, -distance, -plan) can be used independently of Sublime Text via the command-line but can also be called from the editor.

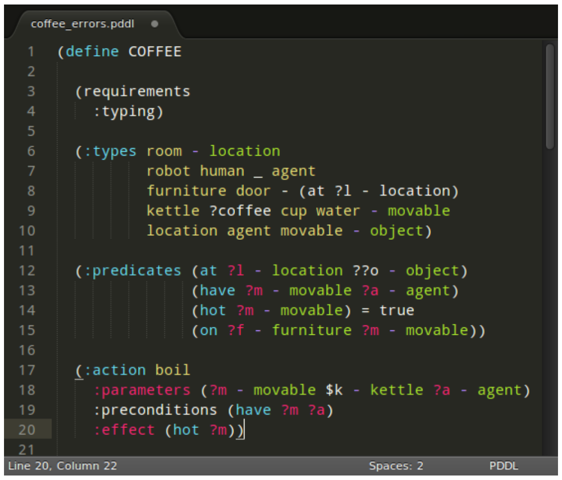
myPDDL-syntax is a context-aware syntax highlighting feature for Sublime Text. It recognizes all PDDL constructs up to version 3.1, such as comments, variables, names, and keywords and highlights them in different colors. Using regular expressions and a sophisticated pattern matching heuristic, it detects both the start and the end of PDDL code blocks and constructs. It then divides them into *scopes*, that is, named regions. Sublime Text colorizes the code elements via the assigned scope names and in accordance with the current color scheme. These scopes allow for a fragmentation of the PDDL files, so that constructs are only highlighted if they appear in the correct context. Thus missing brackets, misplaced expressions and misspelled keywords are visually distinct and can be identified (see Figure 2).

myPDDL-new helps to organize PDDL projects. In many cases PDDL domains are created ad hoc [24]. However, each implementation of a PDDL task specification comprises one domain and at least one corresponding problem file. Since several team members may be working on these files, keeping PDDL projects organized will facilitate collaboration. An automatically created, standardized project folder structure could facilitate the collaboration between users and the maintenance of consistency across projects. To this end,

[5] http://www.sublimetext.com

Table 1: Comparison of knowledge engineering tools and their features.

Feature	Function	PDDL STUDIO	ITSIMPLE	PDDL-mode	planning.domains	VS Code	MYPDDL
latest supp. PDDL version	considering recent PDDL features	1.2	3.1	2.2	3.1	3.1	3.1
syntax highlighting	supporting error detection and code navigation	yes	basic	basic	basic	yes	yes
semantic error detection	supporting error detection	yes	no	no	no	yes	no
automatic indentation	supporting readability and navigation	no	no	yes	no	yes	yes
code completion	speeding-up the knowledge engineering process	yes	no	yes	yes	yes	yes
code snippets	speeding-up the knowledge engineering process externalizing user's memory	no	yes	yes	basic	yes	yes
code folding	supporting keeping an overview of the code structure	yes	no	yes	yes	yes	yes
domain visualization	supporting fast understanding of the domain structure	no	no	no	no	no	yes
project management	supporting keeping an overview of associated files	yes	yes	no	yes	yes	yes
UML to PDDL translation	supporting initial modeling	no	yes	no	no	no	yes
planner integration	allowing for convenient planner access	basic	yes	no	yes	yes	yes
plan visualization	supporting understanding and crosschecking the plan	no	yes	no	no	yes	no
dynamic analysis	supporting dynamic domain analysis	no	yes	no	no	no	no
declaration menu	supporting code navigation	no	no	yes	no	yes	no
interface with programming language	automating tasks extending PDDL's modeling capabilities	no	no	no	no	no	yes
customization features	acknowledging individual needs and preferences	basic	no	yes	basic	yes	yes



```
(define COFFEE

  (requirements
    :typing)

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

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

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

```

Fig. 2: The figure shows the use of myPddl in the text editor Sublime Text. Syntax errors in the domain are detected by myPddl-syntax's context-aware syntax highlighting feature and displayed in white.

myPddl-new creates the following folder structure when creating a new pddl project:

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

All of the templates to create the files can be customized and new templates can be added. The domain file `domain.pddl` and the problem file `p01.pddl` initially contain corresponding pddl skeletons. Additionally the project name is used as the domain name within the files `domain.pddl` and `p01.pddl`. All problem files that are associated with one domain file are collected in the folder `problems/`. `README.md` is a Markdown file, which is intended for information about the authors of the project, contact information, informal domain and problem specifications, and licensing information. Markdown files can be converted to html by various hosting services (like GitHub or Bitbucket). The basic planner integration myPddl-plan provided by the file `plan` is described below.

myPDDL-snippet provides code skeletons, that is, templates for often used PDDL constructs such as domains, problems, type and function declarations, and actions. They can be inserted by typing a triggering keyword. Table 2 displays descriptions of all available snippets and the corresponding trigger.

Table 2: The snippets that can be inserted into PDDL files by typing the trigger.

snippet description	trigger
domain skeleton	`domain`
problem skeleton	`problem`
type declaration	`t1, t2, ...`
typed predicate declaration	`p1, p2, ...`
typed function declaration	`f1, f2, ...`
action skeleton	`action`, `durative-action`

For example, typing `action` and pressing the tabulator key inserts the action skeleton in Listing **??**. PDDL constructs with a specified arity can be generated by adding the arity number to the trigger (`p2` would insert the binary predicate template `(pred-name ?x - object ?y - object)`.
Every snippet is stored in a separate file, located in the folder `Packages/PDDL/` of Sublime Text. New snippets can be added and existing snippets can be customized by changing the templates in this folder.

myPDDL-clojure provides a preprocessor for PDDL files to bypass PDDL's limited mathematical capabilities, thus reducing modeling time without overcharging planning algorithms. Since PDDL is used to create more and more complex domains [8,9], one might need the square root function for a distance optimization problem or the logarithmic function for modeling an engineering problem. However, PDDL's calculating capabilities are limited [19]. While these mathematical operations are currently not supported by PDDL itself, preprocessing PDDL files in a programming language and then hardcoding the results back into the file seem to be a reasonable workaround. With the help of such an interface, the modeling time can be reduced. We decided to use the functional programming language Clojure [11], a modern Lisp dialect that runs on the Java Virtual Machine (JVM) [15], facilitating input and output of the Lisp-style PDDL constructs. Once a part is extracted and represented in Clojure, the processing possibilities are diverse and the full capacities of Clojure can be used. It can be used for generating PDDL constructs, reading domain and problem files, handling, using and modifying the input, and generating PDDL files as output.
The interface is provided as a Clojure library and based on two methods described below.

read-construct(keyword, file) This methods allows for the extraction of code blocks from PDDL files. The following code block shows an example in which the goal state `(:goal (exploited magicfailureapp))` is extracted.

Clojure command:

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

add-construct(file, block, part) This methods provides a means for adding constructs to a specified code block in PDDL problem files. This is illustrated in the following two code blocks where the predicate `(hungry gisela)` is added to the `(:init ...)` block.

Clojure command:

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

Updated PDDL file:

```
(:init (hungry gary)
       (in pizza-box big-pepperoni)
       (has-access gisela magicfailureapp))
       (hungry gisela))
```

myPDDL-distance provides special preprocessing functions for distance calculations. In some domains, every object needs a location specified by x and y coordinates. While the location of objects can be implemented using the predicate `(location ?o - object ?x ?y - number)`, with `x` and `y` being the spatial coordinates of an object, calculating the Euclidean distance requires using the square root function. However, PDDL 3.1 supports only the four basic arithmetic operators.

Parkinson and Longstaff [19] describe a workaround for this drawback. By writing an action `calculate-sqrt`, they bypass the missing square root function by making use of the Babylonian root method. Although this method approximates the square root function, it requires many iterations and would most likely have an adverse effect on plan generation [19].

More usable and probably faster results can be achieved by using the interface between PDDL and Clojure as a distance calculator, implemented in the tool MYPDDL-DISTANCE. It reads a problem file into Clojure and extracts all locations, defined in the `(:init ...)` code block. The Euclidean distances between these locations are then calculated and written back into a new and now extended copy of the problem file, using the predicate `(distance ?o1 ?o2 - object ?n - number)`, which specifies the distance between two objects. Listing 1 and 2 show the `(:init ...)` code block of a extended version of *Gary's Huge Problem* before and after using MYPDDL-distance, respectively.

The calculator works on any arity of the specified location predicate, so that locations can be specified in 1D, 2D, 3D, and even used in higher dimensions. A disadvantage of this method is that the calculated distances have to be stored in the PDDL problem file, potentially requiring many lines of code. If the number of locations is n, the number of calculated distances is n^2, so that every location has a distance to every other location and itself. Therefore, a sensible next step would be to extend PDDL by increasing its mathematical expressivity [19], perhaps by declaring a requirement `:math` that specifies further mathematical operations.

```
(:init ...
       (location gary 4 2)
       (location pizza 2 3))
```

Listing 1: Excerpt of the `(:init ...)` block of the extended file *Gary's Huge Problem* before using MYPDDL-distance.

```
(:init ...
       (location gary 4 2)
       (location pizza 2 3)
       (distance gary gary 0.0)
       (distance gary pizza 2.2361)
       (distance pizza gary 2.2361)
       (distance pizza pizza 0.0))
```

Listing 2: After the application of MYPDDL-distance, the calculated distances are inserted in the `(:init ...)` code block in a copy of the problem file.

myPDDL-diagram generates a PNG image from a PDDL domain file as shown in Figure 3. The diagrammatic representation of textual information helps to quickly understand the connection of hierarchically structured items and should thus be able to simplify the communication and collaboration between developers. In the diagram, types are represented with boxes, with every box consisting of two parts:

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

Generalization relationships express that every subtype is also an instance of the illustrated super type (e.g. "a laptop *is a* computer"). This relationship is indicated in the diagram with an arrow from the subtype (here: *laptop*) to the super type (here: *computer*).
In order to create the diagram, myPddl-diagram utilizes dot from the Graphviz package [6] and takes the following steps:

1. A copy of the domain file is stored in the folder `domains/`.
2. The `(:types ...)` block is extracted via the PDDL/Clojure interface.
3. In Clojure, the types are split into super types and associated subtypes using regular expressions and stored in a Clojure hashmap.
4. Based on the hashmap, the description of a directed graph in the dot language is created and saved in the folder `dot/`.
5. The dot file is passed to dot, creating a png diagram and saving it in the folder `diagrams/`.
6. The png diagram is displayed in a window.

Every time myPddl-diagram is invoked, these steps are executed and, optionally, the names of the saved files are extended by an ascending revision number. Thus, one cannot only identify associated pddl, dot and png files, but also use this feature for basic revision control. Figure **??** displays the folder structure after invoking myPddl-diagram twice on the *Hacker World*.

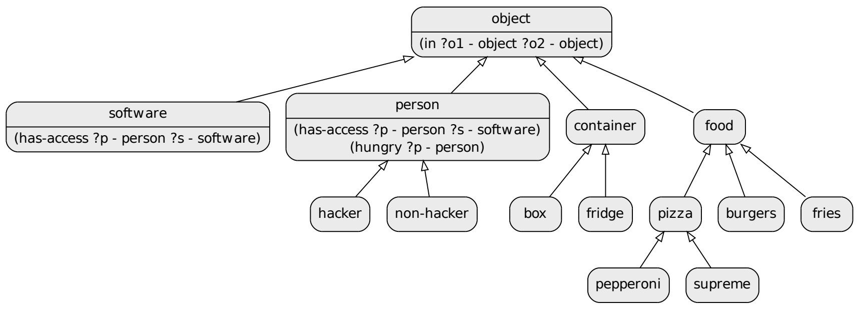


Fig. 3: The type diagram generated by myPddl-diagram helps to grasp the relationship between types in the domain file. Additionally, it displays all predicates that use the corresponding type at least once as a parameter.

myPDDL-plan is a basic planner integration for myPddl. After creating a new project with myPddl-new, the file `plan` in the project folder contains a shell script for executing a planner with the new domain and problem files as input. The desired planner can be specified in the file `plan` or by editing the templates of myPddl-new. Due to the versatility of shell scripts, any planner can be used and arbitrary command line options can be specified. The planner can be invoked from Sublime Text or via the command line.

In order to provide easy installation and maintenance, myPddl-ide can be installed using Sublime Text's Package Control[6]. The project source code is hosted on GitHub[7], providing the possibility to actively participate in the design process. myPddl-clojure is hosted at [8] Additionally, the project site[9] provides room for discussing features and reporting bugs.

4 Validation and Evaluation

To assess the utility of myPddl, we evaluated its performance in terms of collaboration, experience, efficiency, and debugging in a user test. We analyzed the user performance both with and without using myPddl-syntax and myPddl-diagram.

4.1 User Evaluation

The two most central modules of myPddl are myPddl-syntax and myPddl-diagram, since they support collaboration, efficiency, and debugging independently of the user's experience with pddl. To evaluate their usability, they were evaluated in a user study. To this end, we compared the user performance regarding several tasks, both with and without using the respective module.

Participants In Usability Engineering, a typical number of participants for user tests is five to ten. Studies have shown that even such small sample sizes identify about 80 % of the usability problems [17,12]. Our study design required eight participants. Three female and five male participants took part in the study (average age of 22.9, standard deviation of age 0.6). All participants were required to have basic experience with at least one Lisp dialect (in order not to be confused with the many parentheses), but no experience with pddl or AI planning in general.

Approach 24 hours before the experiment was to take place, participants received the web link[10] to a 30-minute interactive video tutorial on AI planning and pddl. This method was chosen in order not to pressure the participant with the presence of an experimenter when trying to understand the material.

[6] https://sublime.wbond.net/about
[7] https://github.com/Pold87/myPDDL
[8] https://github.com/Pold87/pddl-clojure-interface
[9] http://pold87.github.io/myPDDL/
[10] Tutorial in German: https://www.youtube.com/watch?v=Uck-K8VnNOU&list=PL3CZzLUZuiIMWEfJxy-G6OxYVzUrvjwuV

Procedure We defined four tasks: two debugging tasks for testing the syntax highlighting feature and two type hierarchy tasks for testing the type diagram generator. A within subjects design was considered most suited dueto the small number of participants. Therefore, it was necessary to construct two tasks (matched in difficulty) for each of these two types to compare the effects of having the tools available. Each participant started either with a debugging or type hierarchy task and was given the MYPDDL tools either in the first two tasks or the second two tasks, so that each participant completed each task type once with and once without MYPDDL. This results in 2 (first task is debugging or hierarchy) $\times$ 2 (task variations for debugging and hierarchy) $\times$ 2 (starting with or without MYPDDL) = 8 individual task orders, one per participant.

– Debugging Tasks
  For the debugging tasks, participants were given six minutes (a reasonable time frame tested on two pilot tests) to detect as many of the errors in the given domain as possible. They were asked to record each error in a table using pen and paper with the line number and a short comment. Moreover, they were instructed to immediately correct the errors in the code if they knew how to, but not to dwell on the correction otherwise. For the type hierarchy task, participants were asked to answer five questions concerning the domains, all of which could be facilitated with the type diagram generator. One of the five questions (Question 4) also required looking into the code. Participants were told that they should not feel pressured to answer quickly, but to not waste time either. Also they were asked to say their answer out loud as soon as it became evident to them. They were not told that the time it took them to come up with an answer was recorded, since this could have made them feel pressured and thus led to more false answers.
– Type Hierarchy Tasks
  The two tasks to test syntax highlighting presented the user with domains that were 54 lines in length, consisted of 1605 characters and contained 17 errors each. Errors were distributed evenly throughout the domains and were categorized into different types. The occurrence frequencies of these types were matched across domains as well, to ensure equal difficulty for both domains. To test the type diagram generator, two fictional domains with equally complex type hierarchies consisting of non-words were designed (five and six layers in depth, 20 and 21 types). The domains were also matched in length and overall complexity (five and six predicates with approximately the same distribution of arities, one action with four predicates in the precondition and two and three predicates in the effect).
– System Usability Scale
  At the end of the usability test the participants were asked to evaluate the perceived usability of MYPDDL using the system usability scale [4].

Analysis

– Debugging Tasks To compare differences in the debugging tasks, a paired sample *t*-test was used; normality was tested with a Shapiro-Wilk test. to compare the arithmetic means (Ms) of detected errors. The test was performed two-tailed, since syntax highlighting might both help or hinder the participants. Arithmetic standard deviations (SDs) were calculated for each condition.

- Type Hierarchy Tasks For the type hierarchy tasks, t-tests were performed on the logarithms of the data values to compare the geometric means for the two conditions for each question; normality was tested with a Shapiro-Wilk test on the log-normalized data values. The geometric mean is a more accurate measure of the mean for small sample sizes as task times have a strong tendency to be positively skewed [22]. The geometric standard deviation (GSD) was calculated for each question and condition. Only those task completion times were included in the calculation of the t-values, where the respective participant gave a correct answer for both occurrences of a question. This approach should reduce the influence of random guessing. Again, two-tailed t-tests were used to account for both, improvements and drawbacks, of using myPddl-diagram.
- System Usability Scale
  The arithmetic mean and standard deviation for the score on the System Usability Scale was calculated.

Results

- Debugging Tasks
  The participants detected more errors using the syntax highlighting feature, ($M = 7.6$, $SD = 2.07$) than without it ($M = 10.3$, $SD = 3.45$); $t(7) = 2.68$, $p = 0.03$. That is, approximately 36 % more errors were found with syntax highlighting. The arithmetic means are displayed in Figure 4, where each cross ($\times$) represents the data value of one participant.
- Type Hierarchy Tasks
  Figure 5 shows the geometric mean of the completion time of successful tasks for each question with and without the type diagram generator. With the type diagram generator participants answered all questions (except Question 4) on average nearly twice as fast ($GM = 33.0$, $GSD = 2.23$) as without it ($GM = 57.8$, $GSD = 2.05$); $t(32) = -3.34$, $p = 0.002$. This difference slightly increases, if Question 4 is excluded from the calculations: with type generator: $GM = 31.1$, $GSD = 2.17$, without: $GM = 58.1$, $GSD = 2.07$; $t(30) = -3.68$, $p < 0.001$. Table 3 gives an overview of geometric means, geometric standard deviations, t-values, and p-values for each question.
- System Usability Scale myPddl reached a score of 89.6 on the system usability scale [4], with a standard deviation of 3.9.

Discussion The user test shows that myPddl-syntax and -diagram provide useful tools for novices in AI planning and pddl. Below, we will discuss each part of the user test in turn.

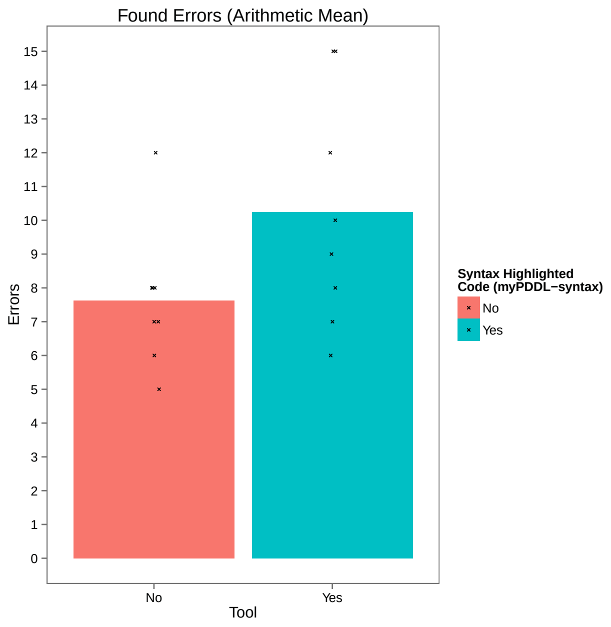


Fig. 4: Comparison of detected errors with and without the syntax highlighting feature. Each cross (×) shows the data value of one participant. The bars display the arithmetic mean.

- Debugging Tasks
  While, in general, the syntax highlighting feature was considered very useful, two participants remarked that the used colors confused them and that they found them more distracting than helpful. One of them mentioned that the contrast of the colors was so low that they were hard for her to distinguish. She found the same number of errors with and without syntax highlighting. The other of the two was the only participant who found less errors with syntax highlighting than without it. With myPddl-syntax, two participants found all errors in the domain, while none achieved this without syntax highlighting.
- Type Hierarchy Tasks
  In spite of the rather large difference between the *GM*s for Question 3, a high p-value is obtained ($p = 0.43$). This might be due to the high *GSD* for the *with* condition and the rather small degrees of freedom ($df = 5$). Testing more participants would probably yield clearer results here. The fact that the availability of tools did not have a positive effect on task completion times for Question 4 can probably be attributed to the complexity of this question (see Appendix, **??** and **??**): in contrast to the other four questions, here, participants were required to look at the actions in the domain file in addition to the type diagram. Most participants were confused by this,

	Type Diagram Generator with		without				
Question	*GM*	*GSD*	*GM*	*GSD*	*df*	*t*	*p*
Q1	21.8	1.52	40.0	2.26	7	-1.86	0.11
Q2	23.8	1.49	50.8	2.16	7	-1.91	0.10
Q3	48.0	3.49	83.2	2.20	5	-0.86	0.43
Q4	84.3	2.22	54.1	1.93	1	4.48	0.14
Q5	41.2	2.24	78.0	1.48	7	-2.75	0.03

Table 3: Overview of geometric means (*GM*s), geometric standard deviations (*GSD*s), degrees of freedom (*df*), *t*-values, and *p*-values. Please note, that the calculation for Q4 is based on only two paired data values ($df = 1$). Please note further that this table only considers paired data values, this means only if a participant answered the question correctly in both domains, the data value is considered (since *paired t*-tests are calculated). In contrast, Figure 5 displays the geometric means for all correct answers.

because they had assumed that once having the type diagram available, it alone would suffice to answer all questions. This initial confusion cost some time, thus negatively influencing the time on the task.

Visualization tools such as myPddl-diagram can improve the understanding of unknown pddl code and thus support collaboration. But users may be unaware of the limitations of such tools. A possible solution is to extend myPddl-diagram to display actions, but this can overload the diagram and, especially for large domains, render it unreadable. Different views for different aspects of the domain or dynamically displayed content could integrate more data, but this also hides functionality, which is generally undesired for usability [18].

– Sytem Usability Scale

Since the overall mean score of the system usability scale has an approximate value of 68 with a standard deviation of 12.5 [21], the score of myPddl is well above average with a small standard deviation. A score of 89.6 is usually attributed to superior products [3]. Furthermore, 89.6 corresponds approximately to a percentile rank of 99.8 %, meaning that it has a better perceived ease-of-use than 99.8 % of the products in the database used by Sauro [21].

In summary, the user test shows that customizability is important, as not all users prefer the same colors or syntax highlighting at all and their personal preferences seem to correlate with the effectiveness of the tools.

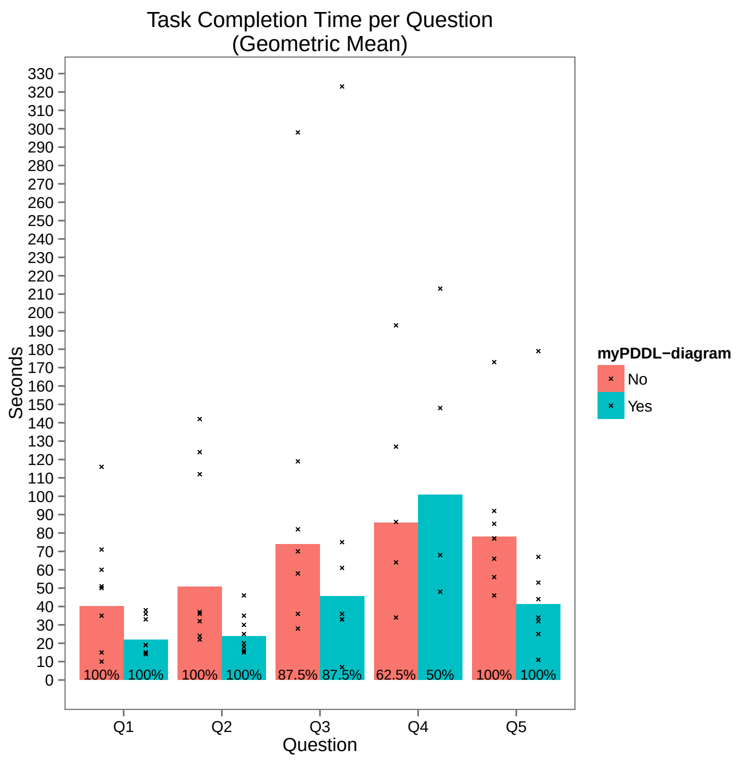


Fig. 5: Task completion time for the type hierarchy tasks. The bars display the geometric mean averaged over all participants; each cross (×) represents the data value of one participant. The percent values at the bottom of the bars show the percentage of users that completed the task successfully. The question can be found in the Appendix (**??**, **??**).

5 Conclusion

We designed myPddl to support knowledge engineers in creating, understanding, modifying, and extending planning domains. myPddl's code editing features such as syntax highlighting and code snippets, as well as a type diagram generator and an interface with the programming language Clojure, can help in the various stages of working with planning domains. myPddl's extensible and customizable architecture helps to fulfill the different preferences and requirements of knowledge engineers. In the conducted user test, myPddl user were able to grasp the domain structure of a pddl file more quickly than non-users and also found more errors in a . Moreover, the users found it as easy and pleasant to use.

In future work, myPddl's set of features could be extended in several directions. The interface between pddl and Clojure offers a basis for creating dynamic planning scenarios. Applications could be the modeling of learning and forgetting by adding facts to or retracting facts from a pddl file or the modeling of an ever changing real world via dynamic predicate lists. Another way of putting the interface to use would be by making the planning process more interactive, allowing for the online interception of planning software in order to account for the needs and wishes of the end user. Since myPddl is command line-based, interfaces with other editors could be developed. There is a basic integration into the code editor Atom[11].

All in all, the overall increase of efficiency due to facilitated collaboration and support in maintaining an overview should encourage a shift of focus toward real world problems in knowledge engineering. The full modeling potential can only be reached with appropriate tools, with myPddl hopefully leading to a broader acceptance and use of pddl for planning problems.

References

1. Ws 5: Knowledge engineering for planning and scheduling (KEPS). http://icaps14.icaps-conference.org/workshops_tutorials/keps.html (2014), accessed: 2019-06-03
2. Sublime text 3. http://www.sublimetext.com/ (2019), accessed: 2019-06-24
3. Bangor, A., Kortum, P.T., Miller, J.T.: An empirical evaluation of the system usability scale. Intl. Journal of Human–Computer Interaction **24**(6), 574–594 (2008)
4. Brooke, J.: Sus – a quick and dirty usability scale. Usability evaluation in industry **189** (1996)
5. Edelkamp, S., Hoffmann, J.: PDDL2.2: The language for the classical part of the 4th International Planning Competition. 4th International Planning Competition (IPC-04) (2004)
6. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz - open source graph drawing tools. In: Graph Drawing. pp. 483–484. Springer (2002)
7. Feigenbaum, E.A., McCorduck, P.: The Fifth Generation. Addison-Wesley Reading (1983)
8. Goldman, R.P., Keller, P.: "type problem in domain description!" or, outsiders' suggestions for PDDL improvement. WS-IPC 2012 p. 43 (2012)
9. Guerin, J.T., Hanna, J.P., Ferland, L., Mattei, N., Goldsmith, J.: The academic advising planning domain. WS-IPC 2012 p. 1 (2012)
10. Helmert, M.: Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition, vol. 4929. Springer (2008)
11. Hickey, R.: The Clojure programming language. In: Proceedings of the 2008 symposium on Dynamic languages. ACM (2008)
12. Hwang, W., Salvendy, G.: Number of people required for usability evaluation: the 10±2 rule. Communications of the ACM **53**(5), 130–133 (2010)
13. Ilghami, O., Murdock, J.W.: An extension to pddl: Actions with embedded code calls. In: Proceedings of the ICAPS 2005 Workshop on Plan Execution: A Reality Check. pp. 84–86 (2005)

[11] https://github.com/Pold87/myPDDL-Atom

14. Konar, A.: Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of the Human Brain, vol. 1. CRC press (1999)
15. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java virtual machine specification. Pearson Education (2014)
16. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: Pddl - the planning domain definition language (1998)
17. Nielsen, J.: Estimating the number of subjects needed for a thinking aloud test. International journal of human-computer studies **41**(3), 385–397 (1994)
18. Norman, D.A.: The design of everyday things. Basic books (2002)
19. Parkinson, S., Longstaff, A.P.: 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)
20. Plch, T., Chomut, M., Brom, C., Barták, R.: Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL Studio. ICAPS12 System Demonstration (2012)
21. Sauro, J.: A practical guide to the system usability scale: Background, benchmarks & best practices. Measuring Usability LLC (2011)
22. Sauro, J., Lewis, J.R.: Quantifying the user experience: Practical statistics for user research. Elsevier (2012)
23. Shah, M., Chrpa, L., Jimoh, F., Kitchin, D., McCluskey, T., Parkinson, S., Vallati, M.: Knowledge engineering tools in planning: State-of-the-art and future challenges. Knowledge Engineering for Planning and Scheduling (2013)
24. Shah, M.M., Chrpa, L., Kitchin, D., McCluskey, T.L., Vallati, M.: 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. pp. 2373–2379. AAAI Press (2013)
25. Strobel, V., Kirsch, A.: Planning in the wild: Modeling tools for PDDL. In: Lutz, C., Thielscher, M. (eds.) KI 2014: Advances in Artificial Intelligence, LNCS, vol. 8736, pp. 273–284. Springer, Cham, Switzerland (2014)
26. Vaquero, T.S., Tonidandel, F., de Barros, L.N., Silva, J.R.: On the use of UML.P for modeling a real application as a planning problem. In: ICAPS. pp. 434–437 (2006)
27. Vaquero, T.S., Tonidandel, F., Silva, J.R.: The itSIMPLE tool for modeling planning domains. Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, California, USA (2005)
28. Vaquero, T., Tonaco, R., Costa, G., Tonidandel, F., Silva, J.R., Beck, J.C.: 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)