MyPDDL: Tools for efficiently creating PDDL domains and problems

Volker Strobel[1][0000−0003−2974−9827] and Alexandra Kirsch[2][0000000256631798]

[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

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, the Planning Domain Definition Language (PDDL) [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.

The specification of Artificial Intelligence (AI) planning can be time-consuming and cubersome. While there is a large spectrum of different AI languages, the de facto standard language is PDDL (*Planning Domain Definition Language*). However, so far its use is predominantly confined to academic labs. One of the reasons seems to be the missing support of knowledge engieering tools so simplify the task of specifying complex planning domains.

In this chapter, we describe MYPDDL (Figure 1), a knowledge engineering toolkit that supports the AI knowledge engineer in the development of complex AI planning problems. MYPDDL is intended to support knowledge engineers throughout the entire design cycle of specifying planning tasks. In the initial stages, it allows for the creation of structured PDDL projects that should encourage a disciplined design process. With the help of snippets, i.e. code templates, often used constructs can be inserted in PDDL files. A syntax highlighting feature that speeds up the error-detection can come in handy in intermediate stages. Understanding the textual representation of complex type hierarchies in domain files can be confusing, so an additional tool enables their visualization. PDDL's limited modeling capabilities were bypassed by developing an interface that converts PDDL code into Clojure [11] code and vice versa. Within this project, the interface was employed for a feature that calculates distances between objects specified in a problem model, but the interface provides numerous other possibilities and could also be used to further automate the modeling process. All of 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 artificial intelligence planning. The results indicate that both error-detection and the understanding of a given domain can be facilitated by MYPDDL.

The remainder of this chapter is structured as follows. Section 2 compares other PDDL editors and PDDL knowledge engineering tools. Section **??** describes the different modules of MYPDDL and their design principles. Section 4 compares MYPDDL to the other existing tools using a benchmark validation. Section 5 concludes the chapter and outlines future work.

2 Related Work

This section introduces knowledge engineering tools that allow editing PDDL files in a textual environment. After introducing the tools, they are compared and their shortcomings are discussed to set the stage for MYPDDL.

PDDL STUDIO [20] is an application for creating and managing PDDL projects, i.e. a collection of PDDL files. PDDL STUDIO's integrated development environment (IDE) was inspired by Microsoft Visual Studio and imperative programming paradigms. Its main features are syntax highlighting, error detection, context sensitive code completion, code folding, project management, and planner integration. PDDL STUDIO's error detection can recognize both syntactic (missing keywords, parentheses, etc.) and semantic (wrong type of predicate parameters, misspelled predicates, etc.) errors.

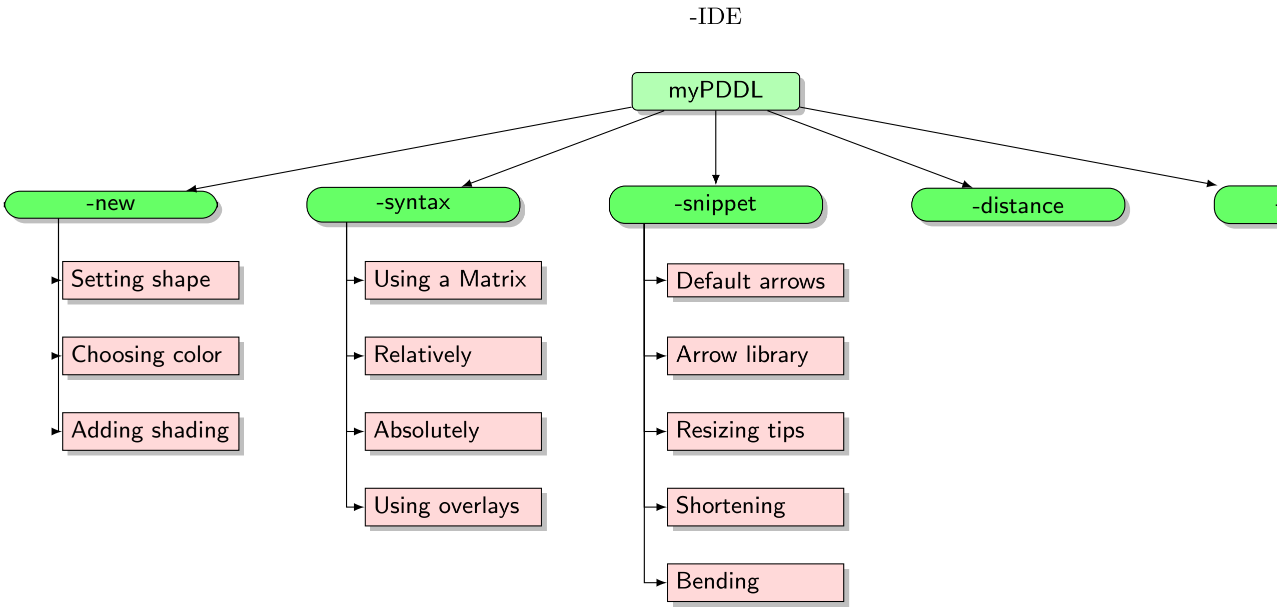


Fig. 1: myPddl is a highly customizable and extensible modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promoting the collaboration between knowledge engineers and the use of PDDL in real-world applications. It consists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

A major drawback of pddl studio is that it is not updated regularly and only supports pddl 1.2. Later pddl versions contain several additional features such as durative actions, numeric fluents, and plan metrics [5].

itSimple [26] follows a graphical approach using Unified Modeling Language (uml) diagrams. In the process leading up to itSimple, uml.p (uml in a Planning Approach) was proposed, a uml variant specifically designed for modeling planning domains and problems [25].

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 from pddl. However, [24] 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. For example, the first parameter in the `:parameters` section of an action is automatically declared as a subclass of the default class `Agent`, and the method is limited to predicates with a maximum arity of two. The current version of itSimple does not include the translation process from pddl to uml.

Starting in version 4.0, itSimple expanded its features to allow the creation of pddl projects from scratch (i.e. without the uml to pddl translation process)

[27]. Thus far, the PDDL editing features are basic. A minimal 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.

Both PDDL STUDIO and ITSIMPLE 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.

PDDL-mode[3] for the widely used Emacs editor builds on the sophisticated features of Emacs and uses its extensibility and customizability. 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, 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 tool editor.planning.domains provides to edit PDDL files online. 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 pillar to the planning community: (1) an API for existing planning problems; (2) a planner-in-the-cloud service; and (3) an online PDDL editor. In contrast to MYPDDL, editor.planning.domains does not support context-aware syntax highlighting: for example, editor.planning.domains highlights key words even if they are misplaced. In contrast, MYPDDL is aware of the scope of a construct and highlights key words accordingly.

The tool vscode-pddl offers a wide range of editing functions, such as syntax highlighting, code completion, and code snippets. Since it is implemented as a plugin for the editor Visual Studio Code, it can fall back on the the extensibility and maturity of this code editor. In contrast to MYPDDL, it does not offer domain visualization nor ab interface with a programming language in order to extend PDDL's limited modeling capacities.

In sum, there is currently no tool available supporting all features of PDDL 3.1, nor all the steps in the modeling process.

3 MyPDDL

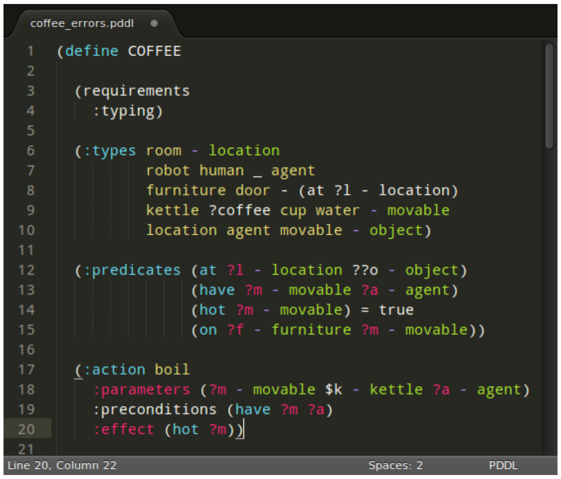
MYPDDL is a highly customizable and extensible modular system, designed for supporting knowledge engineers in the process of writing, analyzing and expanding PDDL files and thereby promoting the collaboration between knowledge engineers and the use of PDDL in real-world applications. MYPDDL consists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

[3] http://rakaposhi.eas.asu.edu/planning- list-mailarchive/msg00085.html

3.1 Modules

myPddl-ide is an integrated development environment (IDE) for the use of myPddl in the text and code editor *Sublime Text*[4]. Since myPddl-snippet and -syntax are devised explicitly for Sublime Text, their integration is implicit. The other tools (-new, -diagram, -distance) can be used independently of Sublime Text with the command-line interface and any pddl file, but were also integrated into the editor.

myPddl-syntax is a context-aware syntax highlighting feature for Sublime Text. It distinguishes all pddl constructs up to version 3.1. Using regular expressions that can recognize both the start and the end of code blocks by means of a sophisticated pattern matching heuristic, myPddl-syntax identifies pddl code blocks and constructs and divides them into so called scopes, i.e. 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).



```
(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: Syntax highlighting using myPddl-ide. White text contains errors.

myPddl-new helps to organize pddl projects by generating the following folder structure:

The domain file `domain.pddl` and the problem file `p01.pddl` initially contain corresponding pddl skeletons which can also be customized. All problem files that are associated with one domain file are collected in the folder

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

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

problems/. README.md is a Markdown file, which is intended for (but not limited to) information about the author(s) 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).

myPddl-snippet provides code skeletons, i.e. 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. Almost all PDDL actions consist of these same parts. And this is only an example. Knowledge engineers have to use the same constructs again and again when writing and extending PDDL files. This is where code snippets can come in handy. To facilitate and speed up the implementation of standard constructs, MYPDDL-snippet provides code skeletons, i.e. templates for often used PDDL constructs. They can be inserted by typing a triggering keyword (trigger). Table 1 displays descriptions of all available snippets and the corresponding 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`

Table 1: The snippets that can be inserted into PDDL files by typing the trigger.

For example, typing `action` and pressing the tabulator key (this function defaults to the tabulator key, but the key can be customized) 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)`.
Once the snippet has been inserted, skipping from blank to blank is enabled by pressing the tabulator key. This allows for a fast navigation within the snippet.

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 template or the trigger) 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], needing the square root function for a distance optimization problem or the logarithmic function for modeling an engineering problem does not seem to be a far-fetched scenario. 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 seems to be a reasonable workaround. With the help of such an interface, the modeling time can reduced an even partly automated (see **??** the distance calculator myPddl-distance). We decided to use 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. The interface is provided as a Clojure library and based on two methods:

read-construct(keyword, file) Allows for the extraction of code blocks from PDDL files. Listing 1 shows an example in which the goal state of *Gary's Huge Problem* is extracted.

add-construct(file, block, part) Provides a means for adding constructs to a specified code block in PDDL problem files. This is illustrated in Listing 2 and 3 where the predicate `(hungry gisela)` is added to the `(:init ...)` block of *Gary's Huge Problem*.

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

Listing 1: Extracting the goal state of *Gary's Huge Problem* by using the interface.

add-construct(file, block, part) Provides a means for adding constructs to a specified code block in PDDL problem files. This is illustrated in Listing 2 and 3 where the predicate `(hungry gisela)` is added to the `(:init ...)` block of *Gary's Huge Problem*.

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

Listing 2: Adding the predicate `(hungry gisela)` to the `(:init ...)` block of *Gary's Huge Problem* using the PDDL/Clojure interface.

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

Listing 3: The modified part of the problem file to which the predicate `(hungry gisela)` has been added.

myPddl-distance provides special preprocessing functions for distance calculations. Let us assume that Gary from Chapter **??** wants the *Hacker World* to be more realistic. Every object shall now have a location specified by a x and y coordinate. Imagine this domain now including a further precondition in the action `eat-pizza`, so that Gary can only eat the pizza if it is at most an arm's length away. 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 ($\sqrt{}$). Unfortunately, PDDL 3.1 supports only four arithmetic operators (`+`, `-`, `/`, `*`). How then is it possible to prevent hackers from starving?

[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)`[5], which specifies the distance between two objects. Listing 4

[5] This predicate has to be declared in the `(:predicates ...)` block of the corresponding domain file.

and 5 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 one, two and three dimensionally and even used in higher dimensions.
However, this preprocessing of problem file also has a drawback, apart from the time required to calculate possibly unused distances. If the number of locations is n, the number of calculated distances is n^2, because every location has a distance to every other location (and to itself). The calculated distances have to be stored in the pddl problem file, potentially requiring a lot of space. 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.
For domains with spatial components, the distance of objects is often important and should not be omitted in the domain model. However, calculating distances from coordinates requires the square root function, which is not supported by pddl (it only supports the four basic arithmetic operators). More sophisticated calculations can be achieved with the supported operators, but the solutions are rather inefficent and inelegant [19]. By calculating the distances offline and including them as additional predicates in the problem file using myPddl-distance, the distances between objects are given to the planner as part of the problem description.

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

Listing 4: 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 5: 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 process of generating the diagrams, an optional copy of the PDDL file can be created, providing a simple version control system. 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] [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 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.*

myPddl-plan Planner execution to get the sequence of actions based on the PDDL domain and problem files.

3.2 Design Principles

As guidelines for design decisions, we used the seven criteria for knowledge engineering tools proposed by Shah et al. [23] as well as general usability principles. Operationality instantiates, whether the generated models can improve the planning performance. This is not a design principle for myPddl, because we

[6] Graphviz is a collection of programs for drawing graphs. This includes dot, 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.

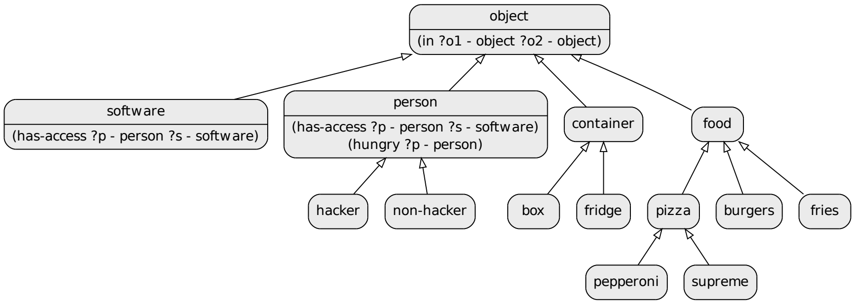


Fig. 3: Type diagram generated by myPddl-diagram

assume that myPddl does not improve the quality (with respect to planning performance) of the resulting pddl specifications. Therefore, we replaced this criterion with functional suitability from the iso/iec 25010 standard, which is defined as the degree to which the software product provides an appropriate set of functions for specified tasks and user objectives (iso 25010 6.1.1). myPddl supports the current version 3.1 of pddl. It encompasses and exceeds most of the functionality of the existing tools. It specifically supports basic editor features with a high customizability as well as visualization support.

Collaboration: With the growing importance of team work and team members not necessarily working in the same building, or in the same country, there is an increasing need for tools supporting the collaboration effort. In developing myPddl, this need was sought to be met by myPddl-diagram. Complex type hierarchies can be hard to overlook, especially if they were constructed by someone else. Therefore, a good way of tackling this problem seemed to be by providing a means to visualize such hierarchies in the form of type diagrams.

Experience: myPddl was designed specifically for users with a background in AI, but not necessarily in pddl. The tools are similar to standard software engineering tools and should thus be easily learnable. The user evaluation (Section TODO) confirms that myPddl helps novices in pddl to master planning task modeling. In addition, it is also possible to customize myPddl so as to adapt its look and feel to other programs one is already familiar with, or simply to make it more enjoyable to use. The project site[7] provides myPddl video introductions and a manual to get started quickly.

Efficiency: All myPddl tools are intended to increase the efficiency with which pddl files are created. myPddl-snippet enables the fast creation of large and correct code skeletons that only need to be complemented. myPddl-syntax can reduce the time spent on searching errors. Code folding allows

[7] http://pold87.github.io/myPDDL

users to hide currently irrelevant parts of the code and automatic indentation increases its readability. To easily keep track of all the parts of a project, folders are automatically created and named with myPddl-new. myPddl-clojure and -distance allow for a straightforward inclusion of numerical values in the problem definition.

Debugging: myPddl -syntax highlights all syntactically correct constructs and leaves all syntactical errors non-highlighted. In contrast, pddl-mode for Emacs and itSimple only provide basic syntax highlighting for emphasizing the structure. pddl studio explicitly detects errors, but the user is immediately prompted when an error is detected. Often, such error messages are premature, for example, just because the closing parenthesis was not typed yet, does not mean it was forgotten. myPddl indicates errors in a more subtle way: syntactic errors are simply not highlighted, while all correct pddl code is. The colors are customizable, so that users can choose how prominent the highlighting sticks out.

Maintenance: The possibility to maintain pddl files is a key aspect of myPddl. The automatically generated type diagram (myPddl-diagram) gives an overview of the domain structure and thereby serves as a continuous means of documentation. Helping to understand foreign code, though, it follows logically that myPddl-diagram also helps in coming back and changing ones own models if some time has elapsed since they were last edited. The basic revision control feature of myPddl-diagram keeps track of changes, making it easy to revert to a previous domain version. Furthermore, myPddl-new encourages adhering to an organized project structure and stores corresponding files at the same location. The automatically created readme file can induce the user to provide further information and documentation about the pddl project.

Support: myPddl-ide can be installed using Sublime Text's Package Control[8]. This allows for an easy installation and staying up-to-date with future versions. In order to provide global access and with it the possibility for developing an active community, the project source code is hosted on GitHub[9]. Additionally, the project site provides room for discussing features and reporting bugs.

4 Validation and Evaluation

To assess the utility of myPddl, we used the criteria listed in Section TODO. The functional suitability was evaluated using a benchmark validation, comparing myPddl's functionality with the tools described in Section 2. The criteria collaboration, experience, efficiency, and debugging were evaluated in a user test. We analyzed the user performance both with and without using myPddl-syntax and myPddl-diagram. The myPddl components supporting maintenance are the same ones that are used in the user test, but their long-term

[8] https://sublime.wbond.net/about

[9] https://github.com/Pold87/myPDDL

usage is difficult to evaluate. The support criterion depends primarily on the infrastructure, which has been established as explained in 3.2.

4.1 Benchmark Validation

Functional suitability encompasses the set of functions to meet the user objectives. The tools of Section 2 basically all follow the same objectives as myPddl: creating pddl domains and problems. The features offered by each tool are summarized in Table 2. Besides supporting the latest pddl version, a strength of myPddl is its high customizability, which comes with the Sublime Text editor. Being the only one of the six tools capable of visualizing parts of the pddl code, it must be understood as complementary to itSimple, which takes the opposite approach of transforming uml diagrams into pddl files. The fact that myPddl does not check for semantic errors is not actually a drawback because planners usually detect semantic errors. All in all, myPddl strives to support the planning task engineer during all phases of the modeling process. Additionally, it features some unique tools, such as domain visualization and an interface with a programming language. It can therefore be concluded that myPddl provides an appropriate set of functions for developing pddl files and is thus functionally suitable.

4.2 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 users 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 No earlier than 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.

[10] https://www.youtube.com/watch?v=Uck-K8VnNOU&list=PL3CZzLUZuiIMWEfJxy-G6OxYVzUrvjwuV (tutorial is in German)

	detection				
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	no	yes	yes	basic
yes	yes				
	externalizing user's memory				
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 code 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	no	no	no	no

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) × 2 (task variations for debugging and hierarchy) × 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 (×) 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.

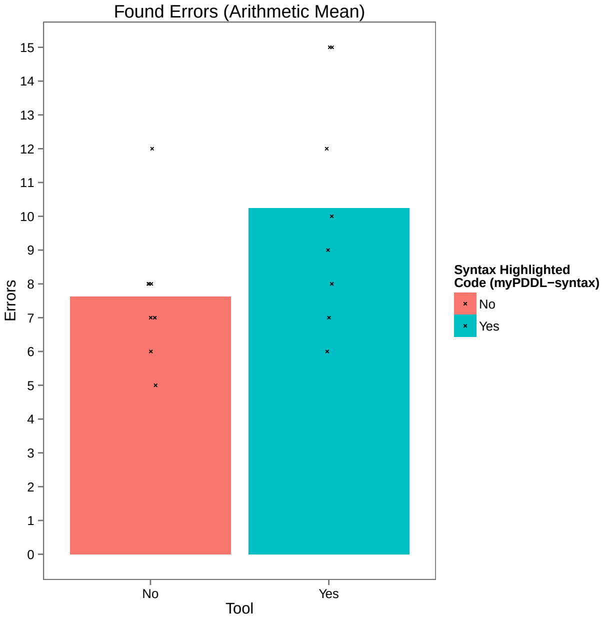


Fig. 4: Comparison of detected errors with and without the syntax highlighting feature. Each cross (×) shows the data value of one participant. The bars display the arithmetic mean.

- 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 GMs 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.

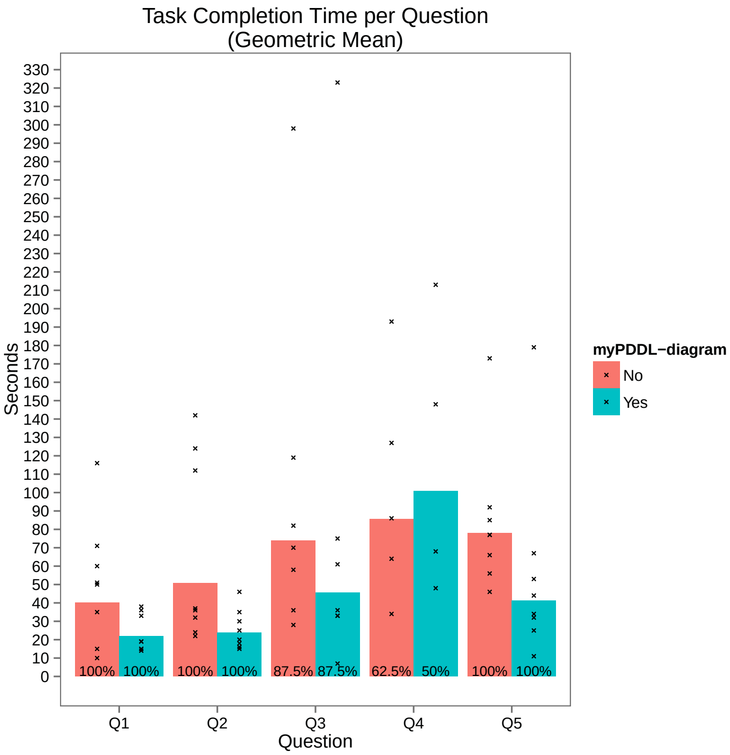


Fig. 5: Task completion time for the type hierarchy tasks. The bars display the geometric mean averaged over all participants; each cross (×) represents the data value of one participant. The percent values at the bottom of the bars show the percentage of users that completed the task successfully. The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

5 Conclusion

We designed myPddl to support knowledge engineers in all steps of developing a planning domain. Additionally, it support engineers to understand, modify, and extend existing pddl planning domains.

This was realized with a set of tools comprising code editing features, namely syntax highlighting and code snippets, a type diagram generator, and a distance calculator. To also have all tools accessible from one place, they were made available in the Sublime Text editor. The different needs and requirements of knowledge engineers are met by the modular, extensible and customizable architecture of the toolkit and Sublime Text. The evaluation of myPddl has shown some initial evidence that it allows a faster understanding of the domain structure, which could be beneficial for the maintenance and application of existing task specifications and for the communication between engineers. Users perceive it as easy and enjoyable to use, and the increase in their performance when using myPddl underpins their subjective impressions. It is hoped that the planning community not only accepts but actually uses myPddl intensely, leading to extensive feedback and continuous improvements.

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.

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)

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. Tonidandel, F., Vaquero, T.S., Silva, J.R.: Reading PDDL, writing an object-oriented model. In: Advances in Artificial Intelligence – IBERAMIA-SBIA 2006, pp. 532–541. Springer (2006)
25. 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)
26. 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)
27. 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)