

PDDL Authoring and Validation Environment for Building end-to-end Planning Solutions

Jan Dolejsi and Derek Long and Maria Fox and Gilles Besançon
{JDolejsi, DLong6, MFox2, GBesancon}@slb.com
Schlumberger, UK

Abstract

This demonstration will show a suite of tools that form a end-to-end PDDL2.2 *developer environment*. Its shape was driven by dual purpose: to help the novice to learn faster and empower the expert to validate models at scale and integrate them to a larger software solution. This toolset takes a shape of an extension for a popular, cross-platform, light-weight and freely available Visual Studio Code editor.

1 Introduction

Planning Domain Definition Language (PDDL2.2) (Edelkamp *et al.*, 2004) has a simple and well described syntax, but learning to encode domain processes in it is still daunting especially when considering the numeric and temporal component of the language. It was already demonstrated by (Brom *et al.*, 2012) that such a PDDL integrated developer environment can be implemented, but it is hard to maintain over time and remains isolated from the other architectural components in the overall software solution.

It was also demonstrated by (planning.domains, 2015) how a light-weight authoring environment may be deployed to any device via a web browser, but as such it lacks advanced language support features available for other programming languages like C++ or Javascript.

While the impact of the two implementations may arguably still be limited, the drivers remain valid and their importance grows over time as the AI Planning technology proves viable in a growing number of industrial applications and the largest bottleneck became the speed of adoption.

2 Accelerating the Learning Curve

The fastest way to train people is to bring the training to the very environment where they do their work. While the syntax is clearly described in number of papers and tutorials and the most commonly used syntax fits one *cheat sheet*, it is far more effective to bring it directly to their finger tips and make it context sensitive to what they are doing, e.g. different syntax is applicable in different sections.

1.1 Understanding PDDL syntax

Example of the tools for the PDDL novice are:

- Syntax highlighting
- Snippets for :domain, :problem, :action, :durative-action, etc... blocks
- Auto-completion for keywords with intelligent snippets guiding them through the different parts,
- Auto-completion for predicate and function names including the developer documentation comment
- Hover-over tooltip for keywords, predicates and functions
- Jump to predicate/function declaration
- Find all references to a predicate/function/type

In addition, any syntax errors flagged by a PDDL Parser are flagged in the PDDL code using the red *squiggly* line and listed in a dedicated UI panel.

For convenience, a PDDL planner can be invoked right from the active editor.

1.2 Understanding PDDL Plans

Plans displayed as text demand human-intensive analysis to confirm whether the plan meets the expectation. Especially when it comes to temporal planning a picture is worth thousand printouts. The features demonstrated include:

- Simple clickable Gantt chart
- Swim lane view per object type and optionally
- Line plot displaying numeric values over time

If the configured planning engine is capable of outputting multiple plans, the PDDL modeler can review and compare all of them.

2 Empowering the Expert

PDDL domain models are usually implemented by gradually increasing their fidelity (and complexity). The testing is done by augmenting the problem file with the new scope and re-testing after every modification. This works for a single person prototyping a domain model for a demo, but is not sustainable when either:

- The scope and complexity grows,
- Multiple PDDL authors need to collaborate or
- Infrequent maintenance causes regression.

As the attention of the PDDL author shifts to the more and more complex aspects, it is essential to keep verifying that all the simple cases still work too (i.e. they should be regression tested). Besides that, once the PDDL domain model works for one or two problem files, they should be

tested on a range of test cases with increasing scale (i.e. scalability testing).

2.1 Regression testing

This PDDL toolset, as all modern integrated developer environments enable:

1. Discovering and enumerating all declared test cases in a workspace,
2. Executing them individually or as a suite,
3. Summarizing the results along with performance characteristics (pass/fail, elapsed time)
4. Defining test assertion (i.e. how is the expected plan supposed to look).

This is done by enumerating test cases in a manifest file using a simple JSON schema.

```
{
  "defaultDomain": "trucks.pddl",
  "cases": [
    {
      "label": "Problem #1",
      "problem": "trucksp0.pddl",
      "expectedPlans": [
        "trucksp0-1.plan"
      ]
    },
    ...
  ]
}
```

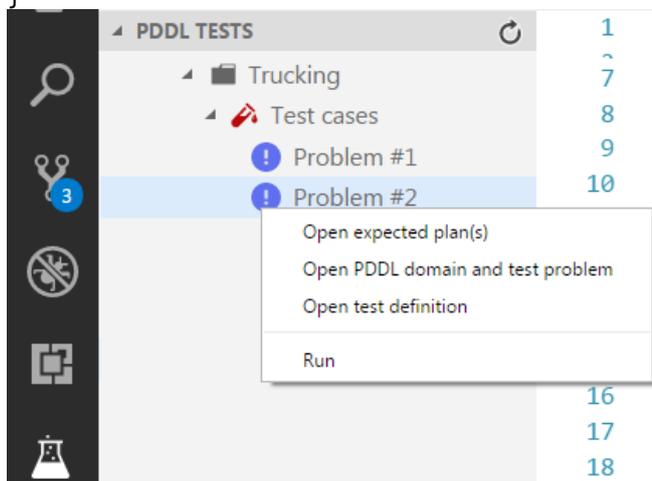


Figure 1 Interacting with regression test cases

2.2 PDDL Problem File Generation

While introducing PDDL into industrial software projects, we have discovered that authoring the domain is not as labor intensive as maintaining a steadily growing set of problem files to test it. Some re-curring patterns can be facilitated by a smart snippet:

- Symmetric predicate/function initialization i.e.: (connected a b)(connected b a)
- Sequence of predicates initialization i.e. (next stop1 stop2)...(next stop8 stop9)

However, this is very limited in scope and however smart those snippets are, they still generate static PDDL code, which is hard to maintain by hand afterwards.

Instead, we propose templating the PDDL problem file using an approach popular among web developers to generate web pages. To illustrate, this is how to generate the (:objects section from a list of items defined in a JSON file:

```
(:objects
{% for object in json.objects %}
  {{object.name}} - {{object.type}}
{% endfor %}
)
```

Several popular templating languages are supported out of the box (Jinja2 and Nunjucks), but the support also includes custom Python scripts or any shell script or custom program that may be appropriate to implement complex data transformations outside of the planning domain's scope. With both Jinja2 and Nunjucks, the data is expected in a .json file.

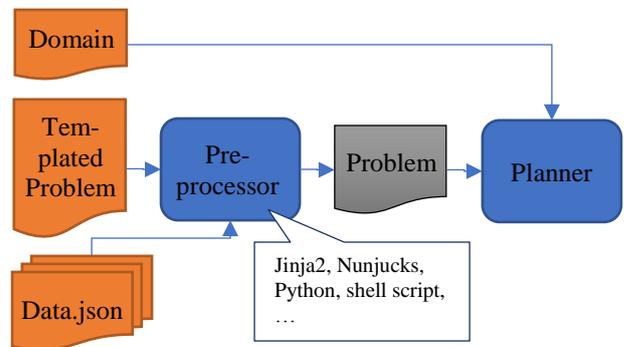


Figure 2 Flow chart of templated problem file generation

This approach is suitable in the context of prototyping the domain model, but can also be used at runtime as part of the overall software solution, where data may be extracted from an operational database instead of static .json files.

Finally, leveraging the regression test framework described above, templated PDDL helps auto-generating a suite of scalability tests that are easy to maintain.

3. Conclusion

To industrialize PDDL without over-investing into dedicated environment, we opted for (Visual Studio Code), which is a light-weight, free, cross-platform editor already fully featured for end-to-end developer workflows and implemented the (PDDL Extension for VS Code). This facilitated classroom training as well as PDDL self-study on one hand and enabled building a full scope plan-based automation solution coded in PDDL, Python and Javascript developed and tested cohesively in a single environment. To make it versatile, we made the PDDL Extension highly configurable for use with different (parser) and (planner) executables or online services.

References

- [Edelkamp *et al.*, 2004] Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: the language for the classical part of the 4th international planning competition. Technical Report 195, ALU Freiburg.
- [Brom *et al.*, 2012] Mgr. Cyril Brom Ph.D, PDDL Studio, <https://amis.mff.cuni.cz/PDDLStudio/>.
- [Strobel Kirsch, 2015] Volker Strobel, Alexandra Kirsch, MYPDDL, https://www.researchgate.net/publication/284788212_Planning_in_the_Wild_Modeling_Tools_for_PDDL
- [planning.domains, 2015] Andrew Coles, Christian Muise, Kristie Taylor-Muise, <http://planning.domains/>.
- [Visual Studio Code] Microsoft, <https://code.visualstudio.com/>.
- [PDDL extension for VS Code], <https://marketplace.visualstudio.com/items/jan-dolejsi.pddl/>
- [Jinja2] <http://jinja.pocoo.org/docs/2.10/templates/>
- [Nunjucks] <https://mozilla.github.io/nunjucks/>
- [parser] Parser configuration guidelines for PDDL VS Code extension: <https://github.com/jan-dolejsi/vscode-pddl/wiki/Configuring-the-PDDL-parser>
- [planner] Planner configuration guidelines for PDDL VS Code extension: <https://github.com/jan-dolejsi/vscode-pddl/wiki/Configuring-the-PDDL-planner>