# PDDL Templating and Custom Reporting: Generating Problems and Processing Plans

**Peter Gregory**
Schlumberger Cambridge Research Center
High Cross, Cambridge, UK
pgregory@slb.com

**Abstract**

Much of the literature in the Knowledge Engineering subfield of Automated Planning has focussed on domain engineering: for example, how to assist non-experts to construct useful PDDL domain files. In addition to domain engineering, non-experts also have to be able to construct planning problems that are correct with respect to this domain, and also interpret plans in a way that makes sense in the particular application domain of study.

This work provides a collection of methods for constructing planning problems, and interpreting plans, based on Jinja2 templating. The focus of the problem construction templating is connecting to data sources and building problems based on these sources. Template functions range from allowing time units and ISO times to set timed initial literals and durations to methods for constructing initial state conditions based on database queries. We provide a unified data structure that combines the information in the domain, problem and plan. Combined with helper functions, we demonstrate how this can ease the process of creating plan reports of various kinds.

## 1 Introduction

Much of the literature in the Knowledge Engineering subfield of Automated Planning has focussed on domain engineering: for example, how to assist non-experts to construct useful PDDL domain files. For example, the MyPDDL system [1] is motivated by the lack of existing systems for ‘modeling complex, real-world problems’. In addition to the task of domain model construction, existing tools also assist modellers to create problem instances. The GIPO system [2] and the itSIMPLE system [3] provide visual methods for constructing both planning domains and problems.

This work provides a collection of methods for constructing planning problems, and interpreting plans, based on Jinja2 templating. The focus of the problem construction template is connecting to data sources and building problems based on these sources. Rather than the GUI-based aspect of previous work, we are concerned with how business systems can interact with planners.

Beyond problem construction, we are interested in plan interpretation. Previous systems do not generally consider what happens to plans after generation. At face-value, a plan is simply a list of instructions that achieve a goal, if executed. When combined with the domain model and problem, the combined structure can convey much more information. A plan does not include details of alternative actions that were not performed, objects that are unused in the plan, timed initial literals, which actions achieve the goals, etc. We provide a unified data structure that combines the information in the domain, problem and plan. Combined with helper functions, we demonstrate how this can ease the process of creating plan reports of various kinds.

## 2 Background

This work relies heavily on Jinja templating [4]. Figure 1 shows a JSON [5] object that will be used to demonstrate Jinja templating. The object records a list of three people and their pets. Jinja templating is a method of transforming a JSON object into another, typically textual, representation. Templates are built by decorating a textual document with references to the input JSON. Jinja offer ways of accessing attributes of JSON objects in a programmatically natural way, provides typical programming features like looping and conditional statements, provides many functions and is extensible such that a developer can add more functions.

Two examples of templates and their outputs are provided in Figure 2 and Figure 3. Figure 2 shows simple looping and attribute access: the template is shown at the top of the figure, the output is shown underneath. As can be seen, attributes are accessed using a double brace syntax, and control statements are accessed using a brace / percent pair syntax. Figure 3 shows a more involved template and its output. In addition to the attribute access that we saw in the previous example, this example shows the use of functions that manipulate the input data: these functions are called ‘filters’ in Jinja terminology, and are invoked using the pipe operator.

This example shows several of these filters: for example the ‘selectattr’ filter that selects a subset of of the elements of a list, based on a provided test; the ‘length’ filter that returns the number of elements in a list and the ‘join’ filter that joins each element of a list with a delimiter. The result of this template is to list the different owners of the pet types.

```
{"data" : [
    {
        "name" : "Peter",
        "pet"  : "cat"
    },
    {
        "name" : "Jonathan",
        "pet"  : "dog"
    },
    {
        "name" : "Ben",
        "pet"  : "cat"
    }
]}
```

Figure 1: An example JSON object: in this case, recording the pets of three people: Peter, Jonathan and Ben.

## Related Work

There are several existing plan reporting tools. As we will see, these offer limited customisation.

**Visual Studio Code Extension** The VS Code Extension [6] is the most sophisticated plan reporting tool that we are aware of. The purpose of the extension is developing planning domains and problems. The plan reporting tool, therefore is domain independent, and shows the plan in a Gannt chart, then each object's timeline, and finally a graph of each of the numeric fluents. An example of the output is shown in Figure 5. This report can either be viewed within Visual Studio Code itself, or exported as HTML. The report can be customised in the following ways:

- The width of the report can be altered.
- The line charts can be displayed either as one chart per grounded function, or one chart per lifted function with one line per grounded function.

Only aesthetic customisations can therefore be made to the report.

**VAL Plan Validation Reporting** The VAL plan validator [7] has the ability to create a LaTeX validation report. This is a comprensive report on the plan validity, state changes that occur in the plan, graphs of numeric fluents and a Gannt chart view of the plan. The report is customisable only to the extent that the LaTeX source is available for editing.

```
{% for person in data %}
    {{person.name}} owns a {{person.pet}}.
{% endfor %}
```

```
    Peter owns a cat.
    Jonathan owns a dog.
    Ben owns a cat.
```

Figure 2: An example Jinja template and its output, given the data in Figure 1. Jinja is a templating language that allows typical programming constructs to produce custom views of data.

```
{% set cats =
  data|selectattr("pet","eq","cat")|list %}
{% set dogs =
  data|selectattr("pet","eq","dog")|list %}
There are {{cats|length}} cat owners:
 {{cats|map(attribute='name')|join(", ")}}.
There are {{dogs|length}} dog owners:
 {{dogs|map(attribute='name')|join(", ")}}.
```

```
There are 2 cat owners:
   Peter, Ben.
There are 1 dog owners:
   Jonathan.
```

Figure 3: A slightly more involved Jinja template and its output, given the data in Figure 1. This example demonstrates the use of filters: essectially functions that manipulate the input data. The filters in this example are `selectattr`, `list`, `length`, `map` and `join`.

**planning.domains** By using the planning.domains website [8], plans are reported as a list of actions. If an action is selected, then the grounded operator is shown in an adjacent panel. This shows the grounded preconditions and effects of the action, which presumably could aid in both the understanding of a domain, and in the debugging of a domain (if, for example, it was not expected that a certain action was applicable). This is the only way in which a plan can be viewed, and there is no customisation available. However, the action selection does show how a dynamic aspect to a plan report can allow more information to be presented as and when the user requires it.

## 3 Problem Construction

The Visual Studio Code PDDL Extension [6] allows the construction of planning problems using a templating language. In this section, we describe some additional filters to enrich this templating language. These filters are designed to make the bridge between business data and the planning model as small as possible. Reducing the steps between original data and a planning model correspondingly reduces the potential to introduce translation errors.

### Table-Based Initial State Generators

The initial state generators that we present are conceived on the notion that the true valuations of a particular predicate type, or numeric function type, can be described in their ex-

| Location | Road |
|---|---|
| Manchester | M6 |
| Cambridge | M11 |
| Birmingham | M6 |

```
(connected Manchester M6)
(connected Cambridge M11)
(connected Birmingham M6)
```

Figure 4: A table of data on the left, and an equivalent set of predicates on the right. Much data is stored, or queried, in tabular form. This table shows motorways that connect to a few British cities.

tensional form. That is, in the form of a table, with all values enumerated. For example, the table in Figure 4 shows a table view of a predicate type, along with the corresponding predicates of this type in the initial state of a planning problem. Figure 6 shows the equivalent for numeric fluent assignments in the initial state.

Database queries typically return tables: this is true both of relational databases in SQL queries and linked databases in SPARQL queries. Spreadsheets form another source of table-based data; Microsoft Excel forms the primary data format in many business areas. As such, we provide three types of filter for constructing initial state predicates from tables (we have corresponding filters for generating fluent assignments, which we omit here due to their similarity):

**`init_pred_sql(pred_name,query)`** This filter takes a file or URL containing the predicate name and an SQL query that generates the initial state. It then outputs the result as predicates. The SQL server, username and password must be supplied either through a separate configuration file, or through the additional filter `sql_config(server, username, password)`.

**`init_pred_sparql(pred_name,query)`** This filter takes the predicate name and a file or URL containing and an SPARQL query that generates the initial state. It then outputs the result as predicates. The SPARQL endpoint must be supplied either through a separate configuration file, or through the additional filter `sparql_config(sparql_endpoint)`.

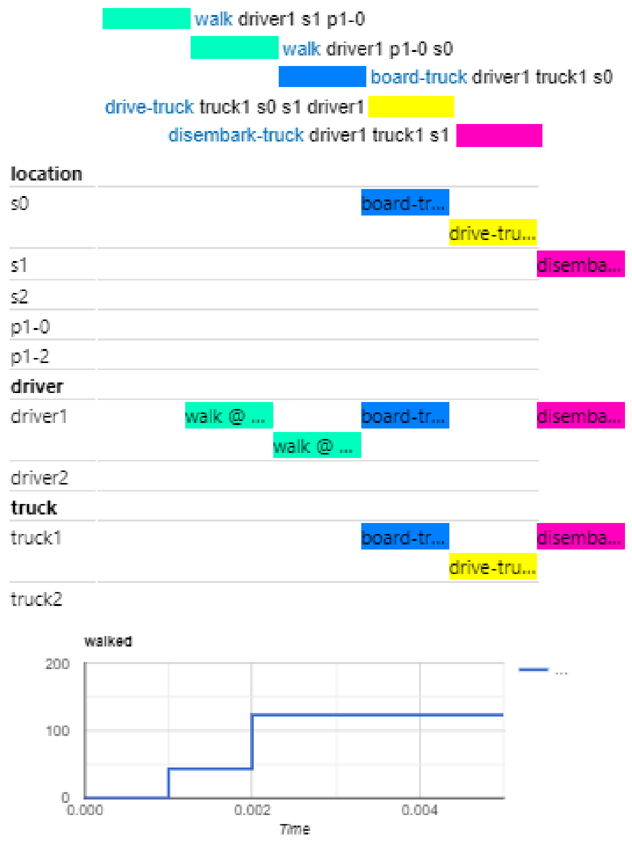


Figure 5: An example of the VS Code plan report. This report displays a Gannt Chart view of the plan, a swim-lane view for each object, and a line chart for each numeric fluent.

| Location | Population |
|---|---|
| Manchester | 555,000 |
| Cambridge | 124,000 |
| Birmingham | 1,140,000 |

```
(= (pop Manchester) 550000)
(= (pop Cambridge)  124000)
(= (pop Birmingham) 1140000)
```

Figure 6: A table of data on the left, and an equivalent set of numeric fluent assignments on the right. This table shows the populations of the cities mentioned in Figure 4.

**`init_pred_excel(pred_name, excel_file, worksheet, range)`** This filter takes the predicate name, an Excel file or URL, the worksheet name, and a range of cells that form the initial state of the predicate (in Excel, ranges are specified in "A1:C10", for example).

These filters will be extended to allow bindings to other data sources, and to bind to named tables in Excel, for example. Currently no support is developed for CSV and HTML tables, for example.

## Time Filters

In PDDL, durations and other times are encoded as unit-less numbers. In the real world, in contrast to PDDL, durations of actions have actual time units, deadlines refer to specific points in time and actions start and end at specific points in time. To reflect this, we introduce a number of filters to deal with ISO time, and time units.

**`pddl_time(ISO_time)`** This filter takes a time encoded as an ISO time (e.g. "2020-01-03T00:00:30") and returns the corresponding unitless PDDL time. This function requires a reference timestamp that signifies the start of the plan, which can be passed in a configuration file, or with the additional filter `timestamp(ISO_time)`.

**`seconds(num)`, `minutes(num)`, `hours(num)`, `days(num)`** and **`months(num)`** This family of filters take a number of a given time unit, and convert that to unitless PDDL time. This filter relies on a standard time unit being used, and this can either be set in a configuration file, by the `time_unit(unit)` filter, or will default to minutes.

These filters can be employed in the duration of actions (for example, in the turn_to action in Figure 7) or to timed initial literals (as in the initial state fragment of Figure 7.

## 4 Combined PDDL Representation

Much information is contained in a PDDL plan. However, without the domain and problem files, it is not possible to interpret its meaning in any useful way. For example: without the problem file, it isn't possible to identify object types (especially when subtyping is used); without the problem

```
(:durative-action turn_to
 :parameters (?s - satellite ?d_new ?d_prev - direction)
 :duration (= ?duration {{1|hours + 25|minutes}})
 :condition (at start (pointing ?s ?d_prev))
 :effect (and
   (at end (pointing ?s ?d_new))
   (at start (not (pointing ?s ?d_prev))))
)
```

```
(:init
 (calibration_target instrument0 groundstation2)
 (at {{"2020-01-04T00:15:00"|pddl_time}}
   (not (calibration_target instrument0 groundstation2)))
 (at {{"2020-01-04T00:15:00"|pddl_time}}
   (calibration_target instrument0 groundstation2))
 (= (slew_time phenomenon4 groundstation2) {{15|minutes}})
 ...
)
```

Figure 7: Examples of the time-related filters developed in order to avoid errors when converting real times into PDDL times. See the turn_to action for an example of combining time units, and the initial state fragment for examples of the pddl_time filter.

file, the timed initial literals are not seen; without the domain file, it is impossible to understand if a change to the plan is valid. Put in the most general terms, the plan cannot be interpreted without the context of the domain model and problem. We propose a unified object that contains all relevant information, in order to be able to interpret a plan in context.

## Combined PDDL JSON Object

In this section we describe the current Combined PDDL JSON [5] Object. The object has been built organically for applications, attributes being added to as required. In this sense, it is not supposed to be a definitive model, but a working model that captures what we have found useful so far. We feel, however, that it already captures a significant amount of information. Figure 8 describes the object in a schema-like representation. This is not formal JSON Schema, as this would be more cumbersome to present in this report, but is intended to be a more human-readable alternative.

There are definitions for the domain, problem, plan and two extra attributes that are derived from the plan: the intervals over which the predicates are true and the sequence of values for each numeric fluent in the problem. These new attributes reflect the fact that there is hidden structure contained within the plan, that can only be recovered by interpreting the plan, problem and domain simultaneously.

When interpreting a plan either for execution, or simply for reporting and storage, it is often the predicates and fluents that are interesting, rather than just the actions. For example, if a fluent represents the fuel level of a vehicle, by looking at the plan alone it isn't possible to see that there is a fuel level fluent at all. In order to know the initial value, the problem is required. To understand how the value changes requires the domain file. By using the combined PDDL JSON object, we have access to this information. This information then allows us to understand our fuel consumption or to chart fuel use over time, for example.

Finally, there is also the 'params' attribute, which allows arbitrary external data to be passed to the template. Examples could include database server information, query paths, timestamps and mappings between PDDL object names to some external reference. Since these will be application specific, we simply define the attribute, without committing to what will be contained within it.

## PDDL Specific Jinja Filters

The PDDL Templating tool is built on top of the VAL plan validation tool, using a C++ Jinja implementation, jinja2cpp. This means that custom filters can be coded in C++ that provide useful computations that could be needed in a custom report of a plan. These filters can then be used within any template. Some of these filters compute information that could form part of the combined PDDL JSON object. Conversely, some information in the PDDL JSON object (for example, the predicate intervals), could be replaced by a filter that would compute these values on-the-fly. The decisions about whether to encode the information, or expose filters to compute it, will need to be made as per user requirements as more plan templates are developed.

**The `state_at` Filter** This filter takes a time as a parameter, and returns the state at that time, in the same format as the state is defined in the initial state of the combined PDDL JSON object. This provides the means to display the state at any time during the plan. This could be useful, for example, if in Driverlog the plan was to be presented as a roster for the drivers. A time-slice of each day could be taken, and the state would contain the truck that each driver was driving.

**The `applicable_at` Filter** This filter takes a time as a parameter, and returns a list of actions that are applicable in this state. This filter allows alternative actions to those found within the plan to be reported. This could help when debugging a domain model, for example, ensuring that those actions that were expected to be applicable actually are.

**The `eval_pddl` Filter** This filter takes an arbitrary PDDL expression, and returns the valuation of that expression over the length of the plan. For example, it may be useful for reporting purposes to monitor those times in which a truck was low on fuel. You could evaluate the following:

```
{{"(<= (fuel truck1)
     (* (capacity truck1) 0.05))"|eval_pddl}}
```

And a list of valuations of the expression will be returned, one for each time-point in the plan the expression changes. In this case, a list of booleans, though numeric fluents are equally possible. Going forwards, we would like to add more general expressions. For example, it might be useful to count the number of drivers who are in a truck at any one time. Currently, this is possible from the Jinja side, for example here:

```
{{state_at(0.0)
    | selectattr("predicate","eq","driving")
    | length }}
```

```
{"data" : {
  "domain":  {"name":  "name"},
  "problem": {"name": "problem_name",
              "domain_name":   "domain_name",
              "objects":       [ { "name": "object name", "type": "object type" } ],
              "initial_state": [ "predicate" ],
              "initial_state_assignments": [
                {"function": "function name",
                 "value":    "initial assignment"} ],
              "timed_initial_state":
                           {"adds": ["predicate"],
                            "dels": ["predicate"]},
              "goal_state": ["predicate"]}
  },
  "plan":    [ { "name":     "action name",
                 "args":     "action arguments",
                 "time":     "start time",
                 "duration": "action duration",
                 "end_time": "end time" } ] ,
  "intervals": [{"variable":  "predicate name",
                 "intervals": [ {"start": "time predicate is added",
                                 "end":   "time predicate is deleted" } ] } ],
  "function_values": [{"function":  "function name",
                       "values": [ {"time":       "time of function value change",
                                    "value":      "new function value",
                                    "continuous": "was the change continuous" } ] } ],
  "params":    {}
}
```

Figure 8: The Combined PDDL JSON Object. The object combines the domain model, the problem, the plan, the predicate intervals and the function values. By combining this information, it becomes easier to interpret the plan in context, and therefore makes building custom reports easier.

However, we imagine that some expressions are better posed as PDDL, and would like to allow some of these natural functions, such as counting, into the `eval_pddl` filter.

Of course, all of these PDDL evaluations could be mechanically added to the domain model itself. However, often the expressions that are used in reporting a plan are different to those that are required in order to generate the plan. In this sense, these extra predicates and fluents would at best slow down the planner through redundant computation, and at worst affect the performance of the planning heuristic.

**The `iso_time` Filter** This filter take a unitless PDDL time value as input, and returns an ISO formatted time as a string. The filter has two optional parameters of a reference timestamp (the start of the plan) and a time unit (seconds, minutes, hours or days) that the value should be interpreted as. These parameters can also be passed in the 'param' attribute of the combined object, since they should remain constant for all queries to the filter.

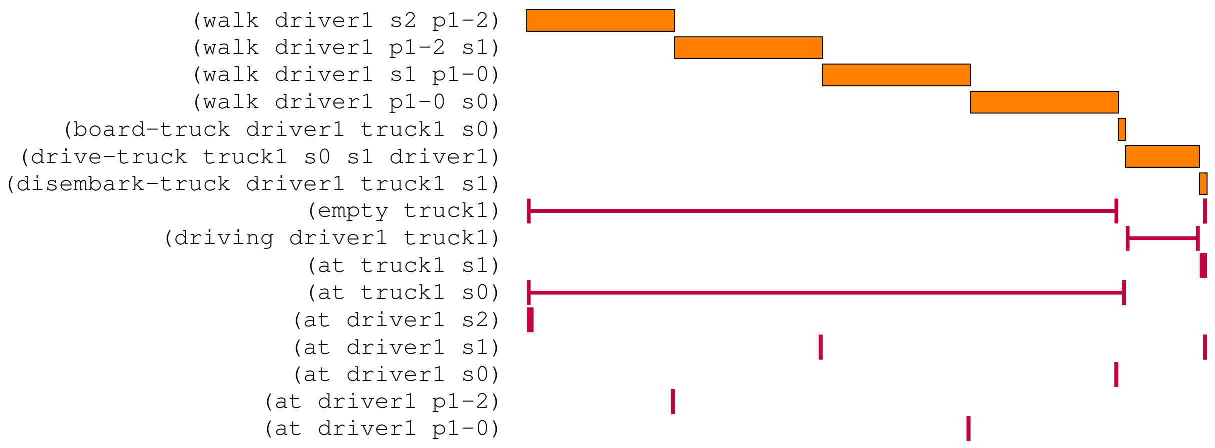
**The `is_static` Filter** This filter takes a predicate as input, and returns true if the predicate is static, and false if it is not. Static, for the purposes of this filter, means static in the context of the plan: i.e. maintains the same value for the duration of the plan. Of course, it would be possible to encode the more common notion of static, meaning not possible to change by any plan. However, for reporting purposes, the former definition has proven more useful practically.

## 5 Custom Plan Reporting

In this section we detail several existing custom plan reports, both domain-independent and domain-dependent, generated by PDDL Jinja templates. The purpose of this is to demonstrate the flexibility of plan templating, and how different applications can be served by the combined PDDL JSON object.

**LaTeX Reporting** Figure 9 shows a generic plan visualisation that shows a Gannt chart view of the plan, and also the intervals over which the predicates hold in the plan. The Jinja template that generates the LaTeX/Ti*k*Z code is shown directly below it. This example demonstrates several of the custom filters described above. Note, for example, that the static predicates are not displayed in the predicates because we filter them out using the 'is_static' filter.

**Driverlog Roadmap Generation** The LaTeX report shown in Figure 9 is a domain independent template. However, since we are interested in custom reporting, we also demonstrate the ability to build a domain-dependent report: this time, creating a dot representation of the roadmap. Recall, in the Driverlog domain, there are paths that the drivers can walk on, and roads that that can only be traversed in a truck. Figure 11 shows this roadmap with the paths in thin edges, and roads in thick edges. The template code is listed below the image.



```
{% set yscale = 0.4 %}
{% set xscale = 0.1 %}
{% for a in plan -%}
{% set idx = plan|length - loop.index %}
\node[left] at (0,{{(idx+0.5)*yscale}}) {\texttt{ {{a.action}} } };
\draw [fill=orange]
    ({{a.time|time_pct*xscale}},{{(idx+0.1)*yscale}})
        rectangle
    ({{a.end_time|time_pct*xscale}},{{(idx+0.9)*yscale}});
{% endfor %}

{% for interval in intervals|rejectattr('variable','is_static') -%}
  {% set idx = -(intervals|rejectattr('variable','is_static')|length - loop.index) %}
  \node[left] at (0,{{(idx-0.5)*yscale}}) {\texttt{ {{interval.variable}} } };
  {% for i in interval.intervals -%}
  \draw [|-|, ultra thick, purple]
    {{i.start|time_pct*xscale}},{{(idx-0.5)*yscale}})
        --
    ({{i.end|time_pct*xscale}},{{(idx-0.5)*yscale}}) ;
  {% endfor -%}
{% endfor -%}
```

Figure 9: Plan visualisation for Driverlog Problem 1 (above) and the corresponding PDDL Template Jinja code (below) that generated the visualisation in Ti*k*Z code. This example demonstrates the use of the `is_static` filter which checks if a predicate is static, and the `time_pct` filter, which returns a time as a percentage of the overall length of the plan.

**Database Code Generation** Interaction with business systems typically involves storing results in some kind of database. Figure 10 shows a template that generates a SPARQL query to insert the times that each package is picked up and delivered in a Driverlog problem. A delivery system would likely need its information in a database in order to communicate estimated times of arrival with customers.

If the plan templating was not available, a developer would be required to write a plan parser, and also interpret the meaning of the PDDL times. By having the conversion performed directly on the PDDL JSON object, the potential for introducing mistakes through misunderstanding the plan, or in incorrectly interpreting the time unit, is reduced.

**Interactive Plan Viewer** Figure 12 shows a screenshot of a web-based interactive plan viewer, showing a Gannt chart and predicate timelines. This visualisation is domain-independent, but it would be straightforward to add domain-specific components. Each component in this visualisation is generated by a separate template. In order to enable this, we built a web service that invokes the plan template system on a domain / problem / plan / template combination. The interactive components of the visualisation are: a search field that restricts the Gannt chart and timelines to a particular object or object type; hovering with the mouse over the plan will reveal the state where the cursor is placed as a tooltip.

Note that the goal predicates are highlighted in green. Short of building a PDDL problem parser, the only way in which this is possible is be having the combined PDDL JSON object, which contains the goals in the problem attribute. We highlight the fact that previous plan reports do not report things like the predicate timelines, and suggest one reason behind this is that these reports primarily report information explicitly reported in the plan.

```
PREFIX dl: <http://www.driverlog.com/>
CONSTRUCT {
 dl:package1 dl:pickup
      "2020-01-02T10:00:00"^^xsd:datetime.
 dl:package1 dl:dropoff
      "2020-01-02T12:00:00"^^xsd:datetime.
 dl:package2 dl:pickup
      "2020-01-02T11:00:00"^^xsd:datetime.
 dl:package2 dl:dropoff
      "2020-01-02T12:00:00"^^xsd:datetime.
}
```

```
PREFIX dl: <http://www.driverlog.com/>
CONSTRUCT {
{% for a in plan|selectattr('action','eq','load') %}
 {{a.args[0]}} dl:pickup
      "{{a.start_time|iso_time}}"^^xsd:datetime.
{% endfor %}
{% for a in plan|selectattr('action','eq','unload') %}
 {{a.args[0]}} dl:dropoff
      "{{a.start_time|iso_time}}"^^xsd:datetime.
{% endfor %}
}
```

Figure 10: An example of a template for generating a SPARQL query to append an ontology with the pickup and dropoff times of each package.

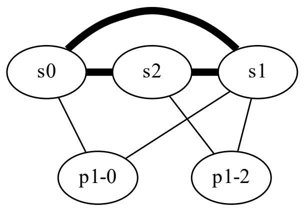
## 6 Future Directions

In this section, we propose several interesting projects that could be enabled by the plan templating system. These typically require additional information to be added to the combined PDDL JSON object.

**Automated Abstractions** Abstractions of the form described in [9] and [10], for example, can often be described as transformations of planning domains and problems, based on certain rules. Think of the delete relaxation heuristic of FF [11], it is possible to transform a domain to a 'relaxed' version of that domain simply by removing the delete effects of the domain operators. Currently, the operators are not exposed in the PDDL JSON object. Once they are, then many abstractions can be computed automatically.

**Validation Reports** One of the most important ways to debug PDDL is to have automated feedback on domain models, problem files and plan validity. The VAL [7] plan validation system provides useful feedback. However, we envisage a more structured feedback report, with error and warning codes that could be more easily exploited in systems like the Visual Studio Code extension [6].

**Interactive Plan Editor** In Section 5 we described an interactive plan visualisation tool. We would like to extend this system to be a plan editing tool, where actions can be added, removed, dragged to different places in the plan, etc. The mechanisms to allow this are already in place: the most important requirement is the 'applicable_at' filter, which returns the applicable actions at a give point in the plan. We feel that the ability to quickly modify plans to satisfy subjective preferences, whilst guaranteeing that plans remain valid will be useful in many applications.



```
digraph x {
{% for pred in problem.initial_state
  |selectattr('predicate','eq','path')|list -%}
  "{{pred.arguments[0]}}" -> "{{pred.arguments[1]}}"
        [arrowhead=none]
{% endfor %}
{% for pred in problem.initial_state
  |selectattr('predicate','eq','link')|list -%}
  "{{pred.arguments[0]}}" -> "{{pred.arguments[1]}}"
        [arrowhead=none, penwidth=5]
{% endfor %}
}
```

Figure 11: Driverlog Map and corresponding template for pfile1. Driveable links are shown in the figure as bold edges, walkable paths are shown as the lighter edges.

## 7 Limitations

The combined PDDL JSON object is a work in progress. The ultimate intention is that it makes explicit all of the implicit richness of a domain model, problem and plan. Additions have currently been made to the object in as as-needed ad hoc manner. We describe some potential additions:

- Operator object definition in the domain model. Having operators made explicit will allow domain transformations within the PDDL templating system. We feel this is the most important omission currently, and we are working to address this presently.
- Causal links between the actions. For many reporting purposes, it will be important to understand the underlying causal structure of the plan. For example, in the interactive visualisation, it would be desirable to focus on an individual goal, and filter out any actions that do not contribute to this goal in some way.

For any new piece of data that could be added, there is a tension between adding the information explicitly in the combined PDDL JSON object, and allowing computation of the information via filters. By adding the information to the object, we add richness to the object, but we also increase its size, potentially making it bloated.

## 8 Conclusions

In this work, we have introduced several Jinja filters related to problem generation and the combined PDDL JSON object that combines information from the domain model, the problem definition and the plan. The idea in this object is to encapsulate as much of the richness of a plan as possible in one place, such that it is possible to interpret plans with all relevant context.

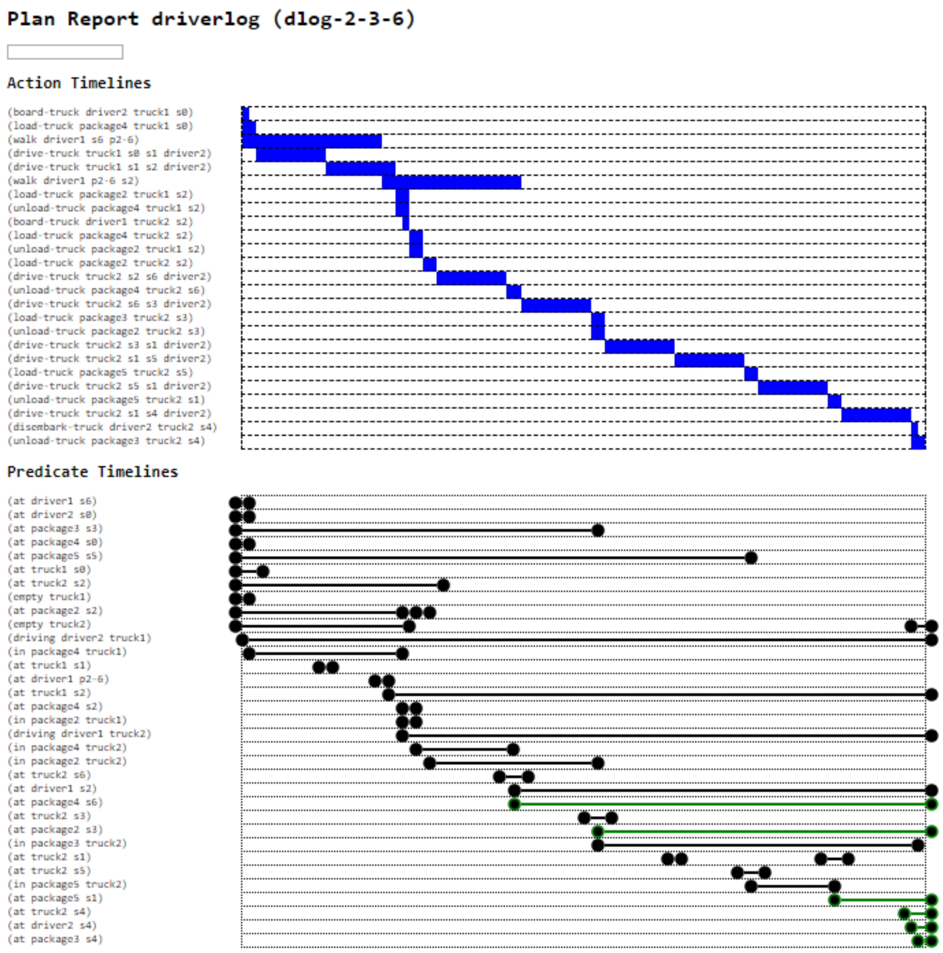

Plan Report driverlog (dlog-2-3-6)
Action Timelines
(board-truck driver2 truck1 s0)
(load-truck package4 truck1 s0)
(walk driver1 s6 p2-6)
(drive-truck truck1 s0 s1 driver2)
(drive-truck truck1 s1 s2 driver2)
(walk driver1 p2-6 s2)
(load-truck package2 truck1 s2)
(unload-truck package4 truck1 s2)
(board-truck driver1 truck2 s2)
(load-truck package4 truck2 s2)
(unload-truck package2 truck1 s2)
(load-truck package2 truck2 s2)
(drive-truck truck2 s2 s6 driver2)
(unload-truck package4 truck2 s6)
(drive-truck truck2 s6 s3 driver2)
(load-truck package3 truck2 s3)
(unload-truck package2 truck2 s3)
(drive-truck truck2 s3 s1 driver2)
(drive-truck truck2 s1 s5 driver2)
(load-truck package5 truck2 s5)
(drive-truck truck2 s5 s1 driver2)
(unload-truck package5 truck2 s1)
(drive-truck truck2 s1 s4 driver2)
(disembark-truck driver2 truck2 s4)
(unload-truck package3 truck2 s4)
Predicate Timelines
(at driver1 s6)
(at driver2 s0)
(at package3 s3)
(at package4 s0)
(at package5 s5)
(at truck1 s0)
(at truck2 s2)
(empty truck1)
(at package2 s2)
(empty truck2)
(driving driver2 truck1)
(in package4 truck1)
(at truck1 s1)
(at driver1 p2-6)
(at truck1 s2)
(at package4 s2)
(in package2 truck1)
(driving driver1 truck2)
(in package4 truck2)
(in package2 truck2)
(at truck2 s6)
(at driver1 s2)
(at package4 s6)
(at truck2 s3)
(at package2 s3)
(in package3 truck2)
(at truck2 s1)
(at truck2 s5)
(in package5 truck2)
(at package5 s1)
(at truck2 s4)
(at driver2 s4)
(at package3 s4)

Figure 12: A screenshot of an interactive plan viewer. This shows similar information to exiting plan reporting systems. In addition to a Gannt chart and the variable fluent graphs, there are predicate timelines..

## References

[1] Volker Strobel and Alexandra Kirsch. "Planning in the wild: modeling tools for PDDL". In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2014, pp. 273–284.

[2] Thomas Leo McCluskey, Donghong Liu, and Ron M Simpson. "GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment." In: *ICAPS*. Vol. 3. 2003, pp. 92–101.

[3] Tiago Stegun Vaquero et al. "itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains." In: *ICAPS*. 2007, pp. 336–343.

[4] Pallets. "Jinja2 Documentation". In: *Jinja2 Documentation (2.11.1)* (2020).

[5] Ben Smith. *Beginning JSON*. Apress, 2015.

[6] Derek Long, Jan Dolejsi, and Maria Fox. "Building support for PDDL as a modelling tool". In: *KEPS 2018*. 2018, p. 78.

[7] Richard Howey, Derek Long, and Maria Fox. "VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL". In: *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*. IEEE Computer Society, 2004, pp. 294–301. DOI: 10.1109/ICTAI.2004.120. URL: https://doi.org/10.1109/ICTAI.2004.120.

[8] Christian Muise. "planning.domains". In: *ICAPS system demonstration* (2016).

[9] Peter Gregory et al. "Exploiting path refinement abstraction in domain transition graphs". In: *Twenty-Fifth AAAI Conference on Artificial Intelligence*. 2011.

[10] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in artificial intelligence and complex systems*. Vol. 456. Springer, 2013.

[11] Jörg Hoffmann. "FF: The fast-forward planning system". In: *AI magazine* 22.3 (2001), pp. 57–57.