PDDL Planning Problems and GROOVE Graph Transformations: Combining two Worlds with a Translator

Ronald Meijer
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.m.meijer-1@student.utwente.nl

ABSTRACT

This paper describes how the AI planning language PDDL was combined with a graph-based model checking tool called GROOVE. Graph-based models are very intuitive to work with, yet powerful enough to deal with a great variety of problems. The visual representation provides a quick insight in the workings of a model and the available tools allows the user to explore the system with great ease. The built-in state space generator of GROOVE comes in handy during the analysis of a problem and its solution.

Traditional PDDL planners on the other hand are specialized in finding a solution without exploring the complete state space. This approach can be necessary for problems which state space is just too big to fully explore. Both approaches have shown to be useful, and the translator we built combines the forces of both: it makes for an easy interpretation of traditional planning problems formulated in PDDL, and exploration process for graph transformation systems with a big state space can be accelerated with an export to PDDL.

Keywords

PDDL, GROOVE, planning, graphs, translation

1. INTRODUCTION

1.1 Motivation

Graph transformations have shown to be a successful technique for modeling software and hardware systems, and they have a rich formal foundation for reasoning about them. To model changes in systems, graphs can be transformed by deleting existing nodes and edges, or creating new ones. Many types of transformations are formalized [3], and are flexible enough to deal with a great variety of models [8]. The visual representation of graphs makes it easy to gain a quick insight into the essence of a model. See section 5.2 for more details about graphs and graph transformation.

AI planners are tools, optimized to find the quickest solution to problems using heuristic search algorithms. For these planners to work properly, the problem must be described in a machine-readable language, including the actions that can be taken to lead to a solution. One language for doing this is PDDL [7], which describes both the problem and the domain in a formal way. See section 5.1 for more details about PDDL.

Edelkamp and Rensink [2] suggested that these two fields are more related than one would think, and graph transformation systems could be used to solve planning problems. Edelkamp [1] already explored the possibilities of using PDDL for model checking software and measured some promising performance results. Although he did the translations between graphs and PDDL manually, a translation has been proven possible and useful. Several examples of PDDL were translated to graphs and several models (represented as graphs) were translated into PDDL descriptions. In this paper, we describe a tool to do this automatically. Building an automated translation will combine the intuitive workings of graph-based modeling and solving with the heuristic exploration algorithms of traditional planners that rely on PDDL input.

1.2 Paper outline

In section 2, we will introduce the basic concepts of PDDL and graph transformations. In section 3, we will elaborate on the problem and what we want to achieve with this paper, followed by a survey of related work in section 4. In section 5, we will zoom in on the details of both PDDL and grapsh, followed by the description of the translation from PDDL to graphs in section 6. Section 7 will describe the translation the other way around. To conclude, the conclusion in section 8 contains a reflection: what do this research and translator contribute, and what is still left to do? The appendices consist of a list of definitions (A) that might help while reading, the graphs generated during the tests (B), the graphs used as input for the tests (C) and the generated PDDL output (D).

2. BACKGROUND

2.1 PDDL

PDDL is an acronym for Planning Domain Definition Language [7], and was an attempt to standardize AI planning languages. It was inspired by several forebears such as STRIPS [5] and ADL [13], and developed in 1998. By default, PDDL uses STRIPS syntax to describe both the domain and the problem (discussed in section 5.1). STRIPS uses preconditions and effects to describe possible actions. Some planners support more features, called requirements, like typing or quantified preconditions. This paper is limited to the basic STRIPS functionality. A piece of example code can be found in section 5.1, where we will discuss the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
17th Twente Student Conference on IT June 25st, 2012, Enschede, The Netherlands.
Copyright 2012, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

syntax. Using the PDDL description as input, planners can try to find a solution for the described problem. The format of the output is not specified, but it is usually a fully or partially ordered plan: a sequence of events (the application of an action, indicating which parameters are used) which will lead from the initial state to the goal.

2.2 Graphs

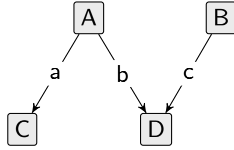


Figure 1. An example graph

A graph G can be defined as a 3-tuple: $G = \langle V, E, L \rangle$, with V being a set of nodes (sometimes called vertices) and E being a set of edges, represented as a 3-tuple with a source node, a label and a destination node: (v_{src}, l, v_{dst}). Finally, L is the complete set of labels that the edges have. Labels of nodes are represented by a self-edge of that node. All this can easily be represented visually by drawing the nodes as boxes, and the edges as arrows from node to node. The following formal description corresponds to the example graph in figure 1.

$$
\begin{aligned}
V &= \{0, 1, 2, 3\} \\
E &= \{(0, A, 0), (1, B, 1), (2, C, 2), (3, D, 3), \\
&\quad (0, a, 2), (0, b, 3), (1, c, 3)\} \\
L &= \{A, B, C, D, a, b, c\}
\end{aligned}
$$

The set of labels could be split up into a set of node-labels and a set of edge-labels. In the example figure however, node labels are just syntactic sugar for self-edges.

2.2.1 *Graph transformation*

To model changes in a non-static environment, it is necessary to change the graph accordingly. Changes in graphs using formal rules are called transformations. Transformations are used in modeling software and hardware systems. The formal rules to change a graph consist of nodes and edges, labeled as one of these four types:

Readers Edges and nodes that must be present to make a rule applicable, shown as a plain black (continuous thin) edge.

Embargoes Edges and nodes which presence prevents the rule from being applied, shown red (dashed fat).

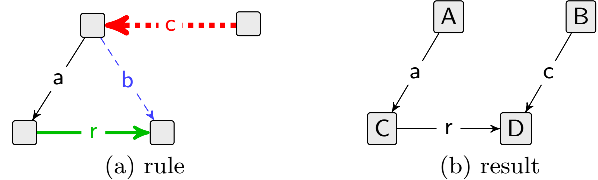
Creators Edges and nodes that will be *created* when the rule is applied, shown green (continuous fat).

Erasers Edges and nodes that will be *deleted* when the rule is applied, shown as a blue (dashed thin) edge.

These terms and colors are specific to GROOVE, the tool we use to model graph transformation systems (discussed later), but all graph transformations systems work with the same principle. An example rule is shown in figure 2a. `a` is a reader, `b` an eraser, `c` an embargo and `r` a creator.

In figure 2 the rule is applied to the example graph (figure 1). This is a valid action because the start graph satisfies the reader but not the embargo. By applying the rule, edge `b` is deleted and edge `r` is created, resulting in the graph in figure 2b.

3. PROBLEM STATEMENT



(a) rule (b) result

Figure 2. An example graph rule and transformation, starting with the graph in figure 1

To design and build a translator that combines the best of the two worlds, we address the following questions:

1. How can problem descriptions in PDDL and graph transformation systems be translated correctly?
    - How can problem descriptions in PDDL be translated correctly into graph transformation systems?
    - How can graph transformation systems be translated correctly into a domain and problem descriptions in PDDL?
2. When can a translation be called 'correct'?
3. How can a translation be implemented in GROOVE?

We define the relation between PDDL and graph transformation systems: what do they haven in common, what are the differences? We show how we use that common ground and work around some of the differences, and list the limitations of the translations.

Products.

As part of the research, we have produced a translator that uses all the results of these questions. By incorporating this translator into GROOVE, the tool will be able to handle PDDL descriptions as input. A graph transformation system will automatically be generated from the description and the user can work with it as if it were a graph system he built himself. This way, existing PDDL descriptions can be visualized, edited and improved.

As for the other way around, graph transformation systems (both generated and hand-made) can be exported to a PDDL description. Traditional planners can use their heuristic search algorithms to quickly explore the state space, so larger and more complex problems can be solved within reasonable time.

4. RELATED WORK

Not much research has been done in the relation between graphs and planning, but the following works are somewhat related:

- Edelkamp [1] already explored the possibilities of using PDDL for model checking software and measured some promising performance results. His translations were done manually.
- Hegedus et. al. [10] described a framework to explore the state space of graph transformation systems using heuristic search. They also measured great performance increase compared to 'normal' state space exploration.
- Hegedus et. al. [11] also developed a guided trajectory exploration algorithm based on the results of petri net [12] analysis. This again shows the potential of heuristic algorithms to explore a state space of graph transformation systems.

- Estler et. al. [4] have build a planning framework that uses heuristic search algorithms to direct the search in a state space of graph-based systems.
- Gerevini and Long [6] described the PDDL language with a BNF grammar, which is one of the first steps towards building the translator.
- Snippe [14] researched the possibility of using A* search to find a path to the end state in GROOVE. This algorithm was not implemented, but the relevance of a heuristic search to explore the state space was shown once again.

5. DETAILS

To explain the translation, we need to zoom in on the details of PDDL and graph transformation systems. For both, we will use 'wumpus world' as an example, a commonly used problem to get used to planning systems and their terminology. Wumpus is a generic 'bad guy' that guards a treasure. The purpose of this problem is to move an agent through a simple maze, defeat the wumpus with a spear, gather the treasure and move back to the starting point. A visual representation of this problem is shown in figure 3. The PDDL code used for these examples is derived from the examples of Patrik Haslum[1].

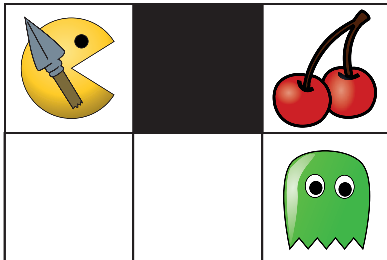


Figure 3. Visual representation of the wumpus world. The agent, carrying a spear, is in the top left corner, the wumpus in the bottom right, and the treasure in the top right.

A possible plan to solve this problem is to move the agent down and right, shoot his spear at the wumpus (killing it), move right and up, take the treasure and walk back (down, left, left, up).

5.1 PDDL

PDDL uses a strict separation between the *domain description* and the *problem description*, both of which will be discussed separately. An important principle in both is *predicates*. Predicates can assign properties to objects, and define relations between objects; for example the predicate `dead`, which puts a label on an object indicating it is considered dead, or `have`, which states that one object is in possession of another.

5.1.1 Domain description

The domain description describes the possible actions and their effects. It consists of the following parts:

Domain name The definition of the domain name; used to refer to the domain.

List of requirements Some AI planners might support more features than basic STRIPS (e.g. typing of objects) and to indicate whether the description uses these extra features, a list of 'requirements' is added to the domain description. If no requirements are given, STRIPS is implied.

List of predicates with each predicate denoting the possible properties of, and relations between objects in the domain.

List of actions The possible actions are described with their names, parameters, preconditions and effects. Both the preconditions and the effects are optional: an action without preconditions can always be applied, an action without effects doesn't change anything but only checks whether some conditions are true. Both the preconditions and the effects are described in terms of predicates, applied to the parameters of the action. These predicates can be negated (`not`), and multiple predicates are possible using `and`.

```
1  (define (domain wumpus)
     (:requirements :strips)
     (:predicates
       (at ?what ?square)
5      (adj ?square-1 ?square-2)
       (pit ?square)
       (wumpus-in ?square)
       (have ?who ?what)
       (is-agent ?who) (is-wumpus ?who)
10     (is-gold ?what) (is-arrow ?what)
       (dead ?who)
     )
     (:action move-agent
       :parameters (?who ?from ?to)
15     :precondition (and (is-agent ?who)
                          (at ?who ?from)
                          (adj ?from ?to)
                     (not (pit ?to))
                     (not (wumpus-in ?to))
20     )
       :effect   (and (not (at ?who ?from))
                          (at ?who ?to))
     )
     ;more actions: take, shoot, move-wumpus
25 )
```

Listing 1. Domain description in PDDL

Listing 1 shows an example *domain description*, where predicates are defined on lines `3-11`, and one possible action on lines `13-23`. The predicates with one parameter can assign a property to an object (e.g. `is-agent`), predicates with two or more parameters give a relation between objects (e.g. `at`). The actions are described by the required parameters, the preconditions and effects (both expressed in terms of predicates). In the example, `move-agent` can only be applied to three objects if one of them is an agent (`?who`) on a tile (`?from`), `?from` must be adjacent to another tile (`?to`) and that tile cannot be a pit and there may be no wumpus on it. When this action is applied, the agent is no longer on the first tile, but on the second.

Note that in the domain description, no assignments are made. The domain only indicates the possibilities and the consequences of actions when they are applied. The actual assignments of objects and predicates are done in the problem description:

5.1.2 Problem description

The *problem description* states which objects exist (e.g. puzzle pieces), which predicates each object satisfies (e.g.

[1]`http://users.cecs.anu.edu.au/~patrik/pddlman/wumpus.html`

shape and initial position of the pieces) and the goal of the problem (e.g. a solved puzzle). An example problem description can be found in listing 2.

The problem needs to have a name and a domain to which it applies, followed by a list of objects that exist. Next, predicates are assigned to to objects in the begin state. Finally, the goal is described with predicates. Not all objects need to be mentioned in the goal, the subset that describes the desired end state is enough.

Both the list of objects and the assignment of predicates are optional. A problem without objects is hardly thinkable, but a problem with no initial predicates is: the predicates could be built up with rules, making it part of the plan. However, in most cases, both will be given.

```
1  (define (problem wumpus-1)
     (:domain wumpus)
     (:objects s-1-1 s-1-2 s-1-3
               s-2-1 s-2-2 s-2-3
5              gold arrow
               agent wumpus
     )
     (:init (adj s-1-1 s-1-2) (adj s-1-2 s-1-1)
            (adj s-1-2 s-1-3) (adj s-1-3 s-1-2)
10          (adj s-2-1 s-2-2) (adj s-2-2 s-2-1)
            (adj s-2-2 s-2-3) (adj s-2-3 s-2-2)
            (adj s-1-1 s-2-1) (adj s-2-1 s-1-1)
            (adj s-1-2 s-2-2) (adj s-2-2 s-1-2)
            (adj s-1-3 s-2-3) (adj s-2-3 s-1-3)
15          (is-gold gold)     (at gold s-1-3)
            (is-agent agent)  (at agent s-1-1)
            (is-arrow arrow)  (have agent arrow)
            (is-wumpus wumpus)(at wumpus s-2-3)
            (wumpus-in s-2-3)  (pit s-1-2)
20   )
     (:goal (and (have agent gold)
                 (at agent s-1-1))
     )
   )
```

Listing 2. Problem description in PDDL

5.2 Graphs in GROOVE

To model all graphs and graph rules, a tool is needed. Several options are available, such as GROOVE[2], AGG[3] and AUGUR[4]. We chose to use GROOVE because this tool is being developed and used at the University of Twente, so there is a lot of knowledge about the internal workings of the tool. A great advantage is the built-in state space generator, which can be used to see if the model is correct and to find a plan that leads to (one of the) end state(s). The usage of the tool is very intuitive, immediately showing one of the powers of graph models.

The tool divides the case into a start graph and graph rules. The rules can be prioritized, so that certain rules will never be applied if other rules can be. For example, the pattern of the desired end state can be formulated as a rule with the highest priority, so no more rules will be applied if the graph matches this state.

Figure 4 shows a graph representation of the problem described in listing 2. All objects are represented as nodes, the predicates are shown as edges between them. The object names are shown inside the nodes, as node labels. Properties of the nodes are shown as proper self-edges, just for clarity. This has no influence on the working of the system.

[2]groove.cs.utwente.nl

[3]user.cs.tu-berlin.de/~gragra/agg

[4]www.fmi.uni-stuttgart.de/szs/tools/augur

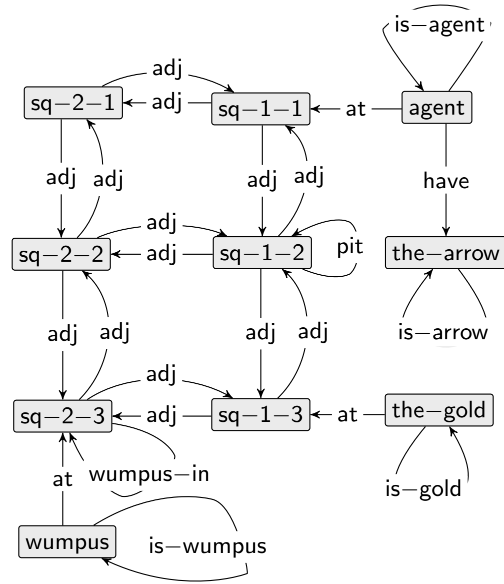


Figure 4. Graph representation of the wumpus problem in GROOVE. Object names are pictured as node labels, predicates as (self) edges

Similarly, figure 5 shows the graph representation of the action `move-agent` (in graphs, an action is called a rule), described in listing 1.

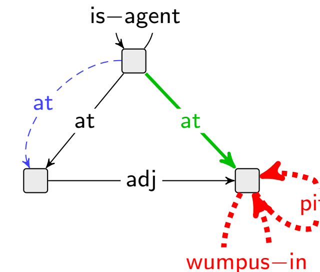


Figure 5. Graph representation of the move-agent action in GROOVE

A closer look learns that there is only one possible match of the rule to the start graph: the agent could move from `sq-1-1` to `sq-2-1`.

State Space.

One of the biggest advantages of GROOVE is the built-in state space generator. This view represents each possible graph of the system as a node, and each transition (rule application) as an edge. A rule application r on graph g_0 which leads to graph g_1 is displayed as $g_0 \xrightarrow{r} g_1$. There are several methods to explore the state space, like manually, depth-first, breadth-first and some (semi-)random algorithms. The complete state space can give a quick insight into the correctness of the model by looking for the presence of deadlock-states (no more possible actions), or the solution of the problem by finding (a) final state(s). The path toward such states can easily be followed to see where the real problem is. Each node can be clicked to see the corresponding graph.

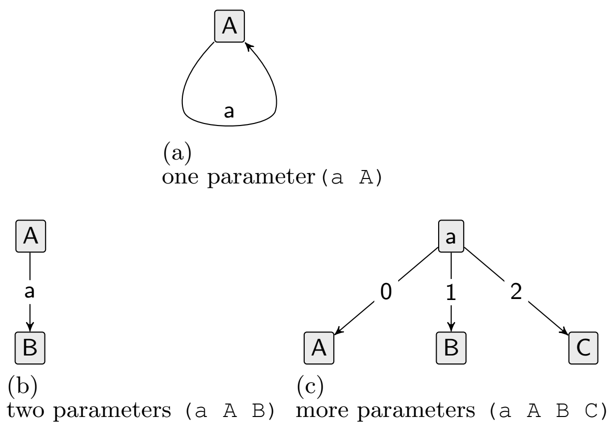
6. TRANSLATION PDDL TO GRAPHS

We will now describe the translation from PDDL to a graph transformation system in GROOVE. First we will explain the theoretical background for reasoning about its correctness (section 6.1), followed by a test plan and the results (section 6.2 and 6.2).

6.1 Theory

The way PDDL splits up problems into a *domain definition* and a *problem definition* very much resembles the way graph transformation systems are represented. In GROOVE in particular, there is a clear division between the rules (domain) and the start graph (problem). This similarity makes the translation fairly simple.

The predicates are easy to translate into edges: a self-edge for predicates with only one argument, an edge between two nodes for two arguments and an extra node with multiple edges to multiple existing nodes for predicates with three or more arguments. An example of these translations can be seen in figure 6.



(a) one parameter `(a A)`

(b) two parameters `(a A B)`

(c) more parameters `(a A B C)`

Figure 6. Example basic translations

The actions can all be translated to graph transformation rules by interpreting the preconditions as readers and embargoes, and the effects as creators and erasers, except for one difference: An eraser in GROOVE acts as a reader at the same time, allowing the rule application only when that edge is present. PDDL however, applies the action regardless of the presence of the predicate that is to be deleted. This might seem to cause no problems, as the result is the same. The resulting plan would differ between the systems however, and if the action/rule would include another effect, this action would produce another state than the corresponding rule. The desired behavior of the rule would be to erase the relation *if present*, but also apply the rule if it is not, as this is how PDDL is specified. GROOVE supports a feature that does this, using existential quantifiers. This feature is well-documented for nodes, but not so much for edges. The solution is to introduce a new *quantifier node* with a certain *level indicator*. The eraser rule should reference to that level indicator, resulting in the desired behavior. Each eraser needs its own, unique quantifier node to ensure they are matched independently. Figure 7a shows the 'simple' approach, figure 7b shows the rule with an existential quantifier. The extra node is shown as $\exists_z^?$, the label of the eraser edge reads `b@z`.

Another solution for this phenomenon would be to create two rules for each action: one *with* eraser and one *without*. This way, it is always possible to apply either one

```
:precondition (
      (a ?A ?B)
)
:effect (
   not (b ?A ?B)
)
```

Listing 3. PDDL code of a negative effect

(a) simple approach

(b) solution

Figure 7. Graph rules of an eraser

of these rules, and by giving them the same *transition label*, the resulting plan makes no difference between them, making both rules appear as one and the same. We chose not to use this method, because the system could become quite cluttered with two or more rules per action. Moreover, constructions like this are more difficult to detect when one would want to translate the graph transformation system back to PDDL.

Another difference between PDDL and graphs is 'injective matching' of objects. PDDL matches the parameters of an actions distinctively, i.e. all the used objects are unique. By default, GROOVE does not do this and one node in a graph can be used multiple times in a single rule. Fortunately, GROOVE has a setting to prevent this: injective matching should be turned on in the resulting graphs for the rules to behave the same as the PDDL actions.

6.2 Testing

To validate this approach, we made PDDL descriptions of a state, and for every atomic action the state that followed after applying each action. We translated these states and actions to graphs and graph rules using the suggested translation method. To put this in a formal way:

> Given translation T, from planner states to graphs and actions to graph rules. $T(S)$ is the graph corresponding to state S, $T(A)$ is the rule corresponding to action A.
> To test translation T, an action a is chosen and applied to a state s_0, resulting in a state s_1: $s_0 \xrightarrow{a} s_1$.
> These states and action are translated using T, resulting in graphs $T(s_0)$ and $T(s_1)$, and graph rule $T(a)$.
> If T is to be called correct for this action a, the rule application of $T(a)$ on graph $T(s_0)$ should result in graph $T(s_1)$: $T(s_0) \xrightarrow{T(a)} T(s_1)$
> T is considered correct if this holds for every action a.

See figure 8 for a visual representation of the test plan.

We performed this test on 12 test cases: A rule can have positive and negative preconditions, and it can have positive and negative effects. All the possibilities to combine are shown in table 1. All actions were applied to a state with two objects; `A` and `B`, with the predicate `a` from `A` to

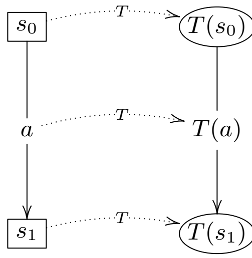


Figure 8. Correctness of the translation. The squares on the left are planner states, the circles on the right are graph states. The solid vertical arrows are action- and rule applications and the dotted horizontal arrows are translations

Table 1. Numbers of the test cases: + for positive preconditions/effects, − for negative, ± for both

test case		Preconditions		
		+	−	±
Effects		1	2	3
	+	4	5	6
	−	7	8	9
	±	10	11	12

B. We translated all actions to graph rules, applied those on the graph that corresponds with the start state and checked whether the result was similar to the result state. For example:

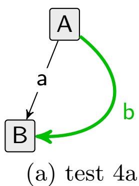
Test 4a & 4b
For all tests as of number 4, we figured there are multiple possibilities to implement the tests: When using multiple predicates in a rule, they can apply on the different relations between the objects (e.g. test 4a, below), or on the same (e.g. test 4b, below). The graph rules of both 4a and 4b are shown in figure 9a and 9b.

4a

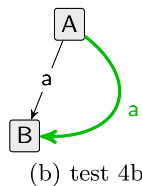
```
:precondition (
        (a ?A ?B)
)
:effect (
        (b ?A ?B)
)
```

4b

```
:precondition (
        (a ?A ?B)
)
:effect (
        (a ?A ?B)
)
```



(a) test 4a



(b) test 4b

Figure 9. Graph rules of test 4

Test 10b
As can be seen in listing 5, this action has two contradicting effects, which does not work in PDDL. Therefor, his action cannot be translated.

6.3 Results

Except for the issue mentioned above, the tests show that the approach appears to be correct, because the generated

```
:precondition (
        (a ?A ?B)
)
:effect (
    and((b ?A ?B)
    not (b ?A ?B))
)
```

Listing 4. PDDL code with two contradicting effects

graph rules for all other actions were applicable to the first graph and resulted in the respective end graphs. To fully test the possibilities, we have designed and built a compiler that reads PDDL-files into GROOVE, automatically representing the problem with graphs and graph rules. To test the compiler, we made GROOVE to come up with a plan for the wumpus problem as described in listing 1 and 2. The resulting graphs are shown in appendix B.

The translator worked properly on the given example: the resulting graph rules reflect the described actions in PDDL, the same goes for the start graph and the goal. The state space however, shows no less than 27 end states. A quick analysis of those states learns that the original PDDL description is not as complete as it appears:

- For the agent, it is forbidden to move to a square with a wumpus on it. The wumpus however, is not restricted in its moves and has no problem moving onto a square with the agent on it.
- The agent can not only take the treasure, but also the wumpus when it is on the same square. Even if the previously mentioned problem is solved, the agent can pick up a dead wumpus, because a wumpus stays on its square when dying. By taking a wumpus alive however, the predicate `wumpus-in` remains, causing unexpected results later as the square has become inaccessible for the agent.
- 'stays' on its square is not completely true. The move-wumpus action does not check whether the wumpus is still alive or not, so even a dead wumpus is able to wander around the field.

All these factors introduce new possible states in which the goal is satisfied, which were probably never found by traditional planners. This is a showcase of the advantages of graph-based problem solving, immediately showing the potential of a good integration between the two systems.

Limitations

The translator is limited to the use of STRIPS, without any of the additional requirements.

7. TRANSLATION GRAPHS TO PDDL

We will now describe the translation from graph transformation systems in GROOVE to PDDL descriptions. First we will explain the theoretical background (section 7.1), followed by a test plan and the results (section 7.2 and 7.3).

7.1 Theory

The translation as described in section 6.1 works pretty much the same way for the other way around. A big difference however, is that PDDL does not support the creation and/or deletion of objects, while it is possible to create nodes as creators and erasers in GROOVE. As

Edelkamp [1] mentioned, additional action effects that create or delete an object would not only help for this problem, but would also be desirable for several other planning purposes. But of course altering PDDL is not feasible within the scope of this research, so we will limit the translation to graph transformation systems without node creations. As a result, PDDL descriptions using predicates of 3 or more parameters can be translated to graphs, but because such a predicate means a node creation in graphs, the translation back to PDDL is impossible.

The fact that an eraser in GROOVE also acts as a reader, has to be taken into account in this translation as well. Luckily, this solution is much simpler, and an extra precondition is enough to simulate the reader-behavior of an eraser.

Another difference between the two systems is in the handling of the effects. GROOVE handles every eraser before it handles all the creators, but PDDL effects are formulated as a kind of post-condition, without any order of execution. Because of this, GROOVE can have 'contradicting' effects (e.g. a creator and an eraser for the same edge), but in PDDL this would mean the predicate is both present and absent after the application of an action. Ignoring the questionable usefulness of such rules, the solution is to detect these kind of structures and replace them with a reader (again, to simulate the reader-behavior of the eraser). The result is of course no problem for PDDL, while graphs that are produced stay the same. Figure 10a and 10b show two graph rules with the same behavior. The first one results in unusable PDDL code (listing 5), while the second one produces valid code.

(a) not translatable (b) translatable

Figure 10. Graph rules with the same behavior

```
:precondition (
      (a ?A ?B)
)
:effect (
   and((b ?A ?B)
   not (b ?A ?B))
)
```

Listing 5. translated PDDL code

7.2 Testing

The test plan of this translation is analogous to the one described in section 6.2, but of course the other way around as shown in figure 11:

7.3 Results

All these tests were executed and showed no other problems than the ones mentioned in section 7.2, again indicating the translation to be correct.

Additionally, we created a basic graph transformation system based on the ferryman problem[5], which can be found in appendix C, and exported it to PDDL descriptions. We chose not to use the wumpus problem again, because the generated graphs cannot be translated back because of the existential quantifiers for the erasers, and a new problem enabled us to test another example on-the-fly, as the generated PDDL code could be translated back to graphs. Parts of the generated PDDL files can be found in Appendix D.

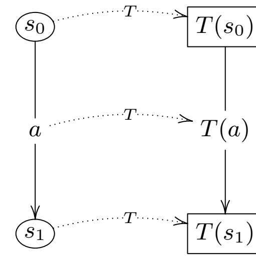


Figure 11. Correctness of the translation. The circles on left right are graph states, The boxes on the right are planner states. The solid vertical arrows are action- and rule applications and the dotted horizontal arrows are translations

Limitations

The translation from graph system to PDDL has some limitations, as mentioned earlier in this paper. Additionally, some extra clarifications are needed for the translator to work properly:

- Because GROOVE saves node labels as self-edges, the translator cannot distinguish the difference between the name of an object, and a predicate with one parameter. Therefor, it is necessary to make node labels stand out by adding the prefix `flag:`. These self-edges will be interpreted as the names of the objects, self-edges without this prefix are considered to be predicates.
- There can only be one start graph, and this graph should describe the problem using labeled nodes.
- One of the rules has to be named `goal` and must describe the desired end state, using only nodes with a flag label, readers and embargoes.
- Existential quantifiers are needed to make the erasers work properly, but these structures are not recognized by the translator from graphs to PDDL and will generate unforeseen results.
- Object creation and deletion is not possible in PDDL, so the graph rules cannot have any creator or eraser nodes.

8. CONCLUSION

8.1 Reflection on problem statement

How can problem descriptions in PDDL and graph transformation systems be translated correctly?

The translation from PDDL to graphs, as suggested in sections 6.1 and 7.1 did not only lead to a working translator, it even showcased one of the biggest advantages of a translator by unveiling several shortcomings in the example PDDL code. The other translation, from graphs to PDDL works as well, although it still has some of limitations.

[5]Also known as the Fox, goose and bag of beans puzzle, but in this case we use a wolf, a goat and cabbage.

When can a translation be called 'correct'?

The extensive process of validating and testing, as described in sections 6.2 and 7.2, indicates that the label 'correct' is in place.

How can a translation be implemented in GROOVE?

We used as much built-in methods of GROOVE as possible, so we didn't have to do anything ourselves that GROOVE already did.

8.2 Future work

This research only scratched the surface of the possibilities. There is a lot of potential in the produced translators, but there is still much left to do:

Extra requirements.

Several options (requirements) of PDDL are left unimplemented and untested. The translator should be extended to handle more requirements like `typing` and `equality`.

Optimization.

The translation is quite naive at some points. For example, an existential quantifier is created regardless of the presence of a reader which forces the edge to be present before applying the rule. This renders the quantifier useless. For the translation back however, this quantifier is quite a burden as it prevents the rule from being translated properly.

The same goes for predicates with 3 or more parameters. These are translated into a node with numbered edges to the object-nodes. This structure cannot be translated back because the generation of objects is not supported in PDDL, but the translator does not recognize this structure to interpret it as a multi-object predicate.

In short, if the translator would recognize patterns in the graph rules, it should be able to identify them and create the respective PDDL construction instead of giving an error. This would increase the number of graphs that are translatable to PDDL.

8.3 Availability

The translators will be incorporated in one of the next releases of the GROOVE tool (urlgroove.cs.utwente.nl). As GROOVE is an open source project, the sources of the translator will become public at the same time.

All the PDDL files and GROOVE grammars mentioned in this paper, both as input and generated, can be found on `www.ronaldm.nl/pddl-groove`.

9. REFERENCES

[1] S. Edelkamp. Limits and possibilities of PDDL for model checking software. *Edelkamp & Hoffmann*, 2003.

[2] S. Edelkamp and A. Rensink. Graph transformation and AI planning. *Knowledge Engineering Competition (ICKEPS), Rhode Island, USA*, 2007.

[3] H. Ehrig. *Fundamentals of algebraic graph transformation*. Springer-Verlag New York Inc, 2006.

[4] H. Estler and H. Wehrheim. Heuristic search-based planning for graph transformation systems. *KEPS 2011*, 2011.

[5] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1972.

[6] A. Gerevini and D. Long. BNF description of PDDL3.0. *technical report*.

[7] M. Ghallab, A. Howe, D. Christianson, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL-the planning domain definition language. *AIPS98 planning committee*, 78(4), 1998.

[8] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*.

[9] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187 – 198, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[10] A. Hegedus, A. Horvath, I. Rath, and D. Varro. A model-driven framework for guided design space exploration. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference*, pages 173 –182, 2011.

[11] A. Hegedus, A. Horvath, and D. Varro. Towards guided trajectory exploration of graph transformation systems. *ECEASST*, 40, 2010.

[12] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, 1989.

[13] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332. Morgan Kaufmann Publishers Inc., 1989.

[14] E. Snippe. Using heuristic search to solve planning problems in GROOVE. *14th Twente Student Conference on IT*, 2011.

APPENDIX

A. DEFINITIONS

The following terms may cause confusion, so to be clear we will briefly explain them.

Action Description of a possible action, using *preconditions* and *effects*.

Creator An edge or node that will be added to the graph by applying a *graph rule*.

Domain Description of the *problem* setting, including possible *predicates* and *actions*.

Effect Description of the effects of an *action*, using *Predicates* and logical operators such as `and` and `not`.

Embargo An edge or node that cannot be present to apply a *graph rule*.

Eraser An edge or node that will be removed from the graph by applying a *graph rule*.

Event The application of an *action*, indicating which *objects* are used as arguments.

Goal The desired end *state*, to which a *plan* should lead.

Graph rule Description of *graph transformations* to be taken when a (sub)graph matches the given *readers* and *embargoes*.

Graph transformation Transformations to be applied to a graph, expressed in *creators* and *erasers*.

Object Instances in the *domain* that can have *predicates*.

Plan A sequence of *events* that leads from the initial *state* to the *goal state*.

Planner A tool that delivers a *plan*, given a *domain* and a *problem*.

Precondition Description of a precondition of an *action*, using *predicates* and logical operators such as `and` and `not`.

Predicate An attribute of an *object*, or a relation between multiple *objects*.

Problem Description of the start *state*, using *predicates*

Reader An edge or node that must be present to apply a *graph rule*.

State A possible configuration of *objects* and *predicates*.

State space All possible *states*.

Translation A (systematic) way to express a *state* or *action* as a graph or *graph rule* respectively, or the other way around.

B. GRAPH OUTPUT

B.1 Start graph

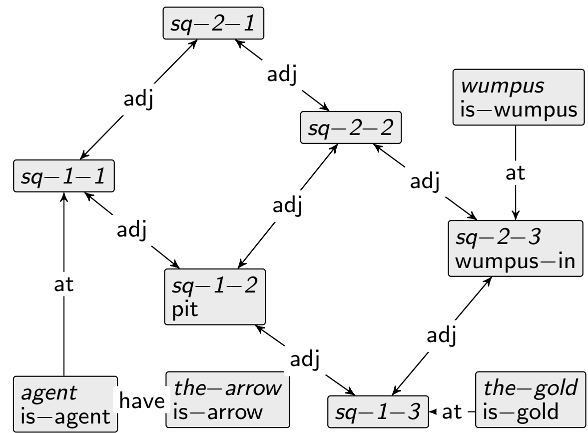


Figure 12. Generated start graph

B.2 Goal

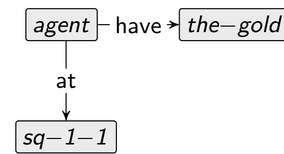


Figure 13. Generated goal rule

B.3 Rules

B.3.1 *move-agent*

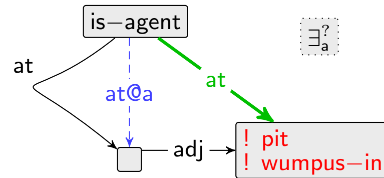


Figure 14. Generated rule graph: `move-agent`

B.3.2 *move-wumpus*

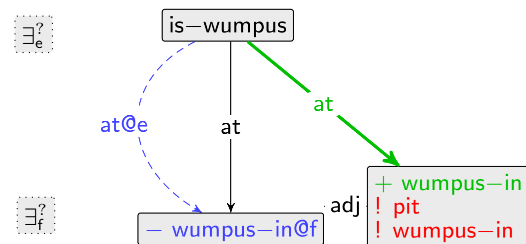


Figure 15. Generated rule graph: `move-wumpus`

B.3.3 *shoot*

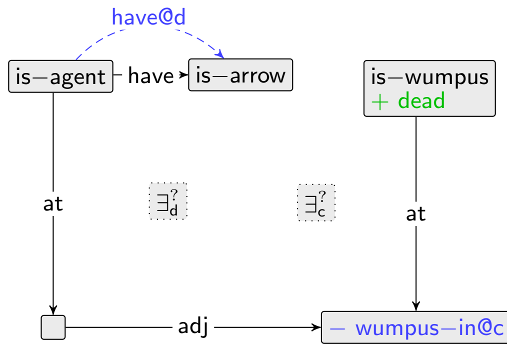


Figure 16. Generated rule graph: shoot

B.3.4 *take*

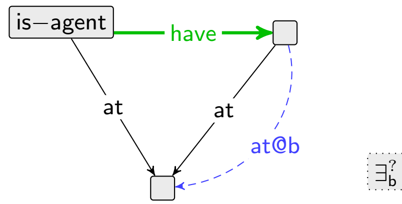


Figure 17. Generated rule graph: take

C. GRAPH INPUT

C.1 Start Graph

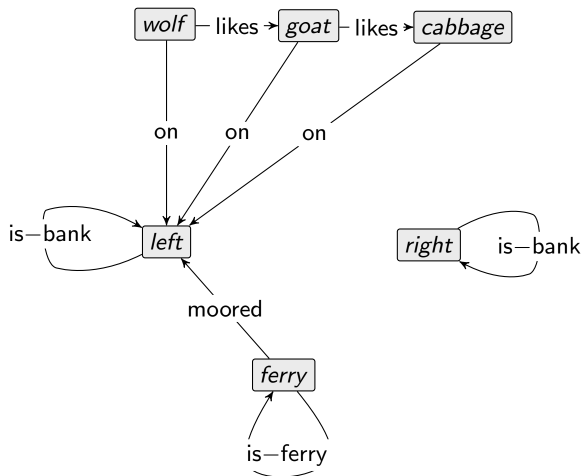


Figure 18. Start graph

C.2 Rule: eat

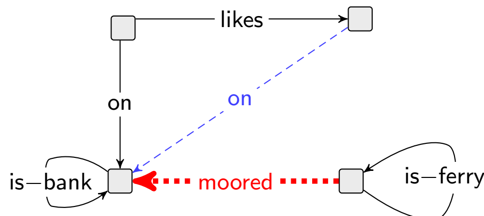


Figure 19. graph rule: eat

D. PDDL OUTPUT

D.1 Domain (fragment)

```
1  (define (domain toPDDL)
       (:requirements :strips)
       (:predicates    (is-bank ?a)
                       (is-ferry ?a)
5                      (moored ?a ?b)
                       (likes ?a ?b)
                       (on ?a ?b))

       (:action eat
10         :parameters ( ?n0 ?n1 ?n2 ?n3)
           :precondition (and ((is-bank n0)
                               (is-ferry n1)
                          (not (moored n1 n0))
                               (on n2 n0)
15                             (likes n2 n3)
                               (on n3 n0)))
           :effect (       (not (on n3 n0)))
       ;other actions: move, move-empty
   )
```

Listing 6. Generated domain description in PDDL

D.2 Problem

```
1  (define (problem start)
       (:domain toPDDL)
       (:objects left right ferry
                 wolf goat cabbage)
5      (:init
           (is-bank left)
           (is-bank right)
           (is-ferry ferry)
           (moored ferry left)
10         (likes wolf goat)
           (on wolf left)
           (on goat left)
           (likes goat cabbage)
           (on cabbage left)
15     )
       (:goal
           (and
               (on wolf right)
               (on goat right)
20             (on cabbage right)
           )
       )
   )
```

Listing 7. Generated problem description in PDDL