Action Usability via Deadend Detection

Qianyu Zhang and **Christian Muise**
School of Computing, Queen's University
qianyu.zhang@queensu.ca, christian.muise@queensu.ca

Abstract

The vast majority of planning problem models are incorrect, incomplete, or simply inconsistent. This is particularly an issue during the development process of a planning model. However, there are few existing debugging tools for modeling to aid with this issue. In this paper, we introduce a method for detecting actions that are deemed *unusable*, which may naturally be a result of modelling errors. We use action usability (or reachability) detection for on-the-fly diagnostics of planning models. In our technique, each action is checked for usability via problem reformulation and unsolvability detection. Through the analysis of usability, this technique could improve the modeling process directly, and we have demonstrated this capability through the tight integration with the online PDDL editor at Planning.Domains.

1 Introduction

The vast majority of PDDL models that exist are just plain wrong. This simple observation often goes overlooked, as we typically only publish the final product of a lengthy modelling process. However, until that final stage is reached, every intermediate planning model necessarily is incomplete, inconsistent, or just plain incorrect. Our work offers a diagnostic tool for planning modelling to help identify a certain type of incorrect models: *actions that are unusable*.

Planning systems are typically designed to expect files in the Planning Domain Definition Language (PDDL) (Haslum *et al.* 2019). The planning tasks are separated into two input files: a domain file and a problem file. The domain file describes a system with invariant rules, like the types of objects, states of the system, and potential actions that change the states. The problem file includes a concrete task based on the system that is specified by the domain file, including the specific objects, an initial state, and goal to achieve.

Unusable actions – i.e., those that can *never* be executed – are a tell tale sign of an error in the domain or problem model. It may be indicative of an incorrect precondition on the action, or incorrect effects of another action. Either way, it highlights a common assumption that *if an action is specified, the modeller expects it to be usable in the domain*. This is not necessarily always the case (e.g., some domains may be automatically generated and some instances may not require all actions), but it is generally true of many modelling situations. Note our reliance on the *problem configuration* in addition to the domain. This is a concious decision, as (1) this notion of unusable actions are a superset of those that are unusable regardless of problem; and (2) the analysis can lead to evidence of errors in both the domain and problem.

The output of a planner usually only shows whether the planner finds a plan or not. If it finds a plan, it means that the final goal is satisfied. However, if it does not find a plan, there is no in-depth explanation to tell us why the error occurs, or which part of the PDDL might be causing the error. It is time-consuming for modellers to figure out the reasons for a lack of solution and how to fix it. Contrasting with other programming language, PDDL modelling tools rarely have their own syntax checking and analysis to aid in the modelling. There are a variety of possible planning-specific analysis that can be done, and we present one such analysis: detecting unusable actions.

We achieve this through the repeated reformulation of the original model. The key insight is that we can introduce a special `(goal)` fluent that an action achieves, and replace the original goal with `(goal)`. Doing so provides us with a new model that has a solution if and only if the action is usable: the goal can be achieved only if the action in question can be executed. This provides an immediate diagnostic for each of the lifted actions, and allows us to report those action specifications that may be the symptom of an error. Note, however, that we do not require the use of a lifted planner for this – we are effectively testing if every grounding is unusable, and if at least one grounding of the action is usable then we assume the lifted action is as well.

We have implemented this approach and exposed it as a service integrated with the online PDDL editor (Muise 2016).[1]. Users of the online editor can directly analyze their PDDL models, and the problematic actions will be indicated directly in the editor itself. This work represents an appealing aspect of knowledge engineering for planning. Namely, we can use planning technology to indirectly aid in the modelling of planning problems themselves.

[1] http://editor.planning.domains/

The rest of the paper is structured as follows. We review some necessary background notation in Section 2. In Section 3 we describe the theoretic foundation for our approach and follow with its implementation in Section 4. Sections 5 and 6 cover the related work and conclusions respectively.

2 Background

STRIPS For the purposes of this paper, we formulate the planning problems using the common STRIPS formalism (Fikes and Nilsson 1971). In STRIPS, a planning problem is a tuple $\Pi = \langle F, I, G, A \rangle$, where F is a finite set of fluents, $I \subseteq F$ is the initial state, and $G \subseteq F$ is the goal state, and A is the finite set of actions. Each action $a_1 \in A$ has a name $name(a)$, precondition $pre(a) \subseteq F$, add effects $add(a) \subseteq F$, and delete effects $del(a)$. An action a is applicable in state $s \subseteq F$ only if $pre(a) \subseteq s$, and the resulting state is computed as $s' = (s \setminus del(a)) \cup add(a)$.

Unsolvability/Dead-ends Unsolvability (aka Dead-end detection) in the field of Automated Planning (AP) is the task of determining if the goal can be reached from a given state (Eriksson et al. 2017). It indicates that there is no solution to the problem, which means not all goals are achievable. It might be because the problem domain exists deadlock states or unreachable actions. As for the deadlock states, two actions are executed infinitely times preventing each other from accessing the next step so that the planner can not find a valid solution. If an unusable action is an indispensable step to reach a final goal, then the planner will not find a solution. However, problems can be unsolvable even if all actions are usable. Problems might be solvable if there exists an unusable action when this action is useless.

Goal and Action Usability A goal g is reachable if there are a sequence of actions such that (1) every action is executable in the state reached by executing all previous actions; and (2) g holds in the final state reached by the sequence of actions. Action reachability (or usability) for action a is the question of whether or not any state can be reached by a sequence of executable actions such that $pre(a)$ holds in the state resulting from executing the sequence.

Online PDDL editor and plugins The online PDDL Editor helps in the creation of planning tasks written in PDDL (Muise 2016). Integrations include import functionality of existing models and a remote solver for testing the models. To enhance the functionality of the PDDL editor, customized plugins can be developed, installed, and invoked by the users of the system. We expose our work through this framework.

3 Approach

Once a planning model is incorrect, it is difficult and time-consuming to evolve users' mental models and let them figure out the reasons for no valid solution and how to fix it without any assisted automated tool. To enhance the PDDL modeling process, we introduce a mechanism to automatically detect if key actions are unusable in the current specification. While this may not indicate an issue exists, quite often it is largely indicative of a problem with the model.

We check whether an action can be executed using a sound and complete planner. To accomplish this, we set the new goal as executing the testing action; the effect of the action is modified to achieve an additional auxiliary fluent. The other actions remain unchanged. If the result is a solvable task, then the action is deemed usable. If not, the action is unusable (likely indicating a bug in the domain). To guarantee the new goal is satisfied if and only if the testing action is executed, their new value has to be different from any existing state. Otherwise, it is possible that the goal is achieved because of an alternative action.

We refer to the problem reformulation process as the Action Usability Algorithm shown in Figure 1. Each action is evaluated in turn. We take one action $a \in A$ as an example to explain the details of the evaluation. To set the execution of a as a temporary goal, we assign a new fluent f_0 to both the effect of action a as well as the goal: $add(a) = G = \{f_0\}$. f_0 is supposed to be different from any existing predicate (i.e., $f_0 \cap F = \emptyset$) so that no more than one action can be the final step of reaching the goal. Since f_0 does not exist in the original domain, it is important to append it to the set of predicates $F' = F \cup f_0$ before using f_0. The variable *usable* stores the usability status of each of the actions. Then planner() calls a sound and complete planner to find a plan for the modified problem. If a valid plan is returned, a is considered as a usable action. If not, then a is unusable.

Algorithm 1: Action Usability Algorithm

Input: F, I, G, A
Output: *usable*

$F' \leftarrow F \cup \{f_0\}$;
$G \leftarrow \{f_0\}$;
usable $\leftarrow \{\}$;
for $a \in A$ **do**
  old_add $\leftarrow add(a)$;
  $add(a) \leftarrow \{f_0\}$;
  usable$[name(a)] \leftarrow$*planner*(F', I, G, A);
  $add(a) \leftarrow$*old_add*;
return *usable*;

We want to test only one action each time to make sure that each action has its independent testing result. Therefore, before checking the next action, the effect of current action needs to be reverted back to its original value, `old-add`. We have the following simple result:

Theorem 1 (Correctness)**.** Algorithm 1 identifies precisely which actions are usable and which are not.

Proof sketch. Both soundness and completeness assume a sound and complete planning process.
Soundness: If an action is deemed unusable, then the new goal of achieving f_0 must not be feasible. If there were some sequence of actions from A that ends with the candidate ac-

tion, then f_0 must be achieved. Thus, any action deemed unusable must in fact be unusable in the domain.

Completeness: Similarly, if the planner *does* find a solution, then the sequence of actions must include the candidate one, as it is the only action that can achieve the goal fluent f_0. Thus, the algorithm detects every action that is unusable. □

If an unusable action can be detected during modelling, then it will improve the efficiency of the process. Possible fixes to such a situation may include revising the problematic action preconditions or the effects of another action. Suggesting such targeted fixes is beyond the scope of this work, but an interesting path for future work.

4 Implementation

To realize the approach presented in the previous section, we built a plugin for the online editor at Planning.Domains. The "Action Usability" plugin can be accessed immediately in the online editor, and the following is a link that already has the plugin enabled and an an (erroneous) version of the common logistics domain ready to analyze:

http://editor.planning.domains/#read_session=7izsskx26H

By developing for the online editor, the plugin is now widely accessible to other researchers and students looking to build and diagnose their PDDL models with the online editor. The plugin was built using the framework defined by the Planning.Domains system. In this section we describe the implementation, including some of the key design decisions, and the code for the plugin can be found here:

https://github.com/AI-Planning/action-usability-detection

The message flow of the action usability plugin is shown in Figure 1. There are 5 components involving in the complete action usability plugin:

1. the editor, editor.planning.domains which is the client-side of our plugin;
2. the plugin, a JavaScript file that is used to call other classes;
3. the wrapper, a python file that is used to implement the action usability algorithm;
4. the solver, solver.planning.domains;
5. the parser, a python file that is used to print out the results.

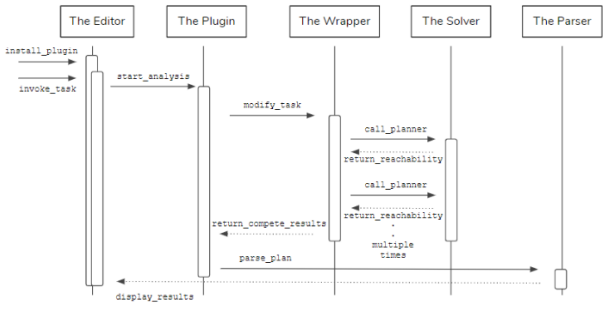

Figure 1: The message flow of the action usability plugin.

Once the plugin and the domain/problem files are loaded, the editor calls the plugin and starts the analysis. The wrapper is called to modify the task and then invoke the remote solver multiple times. The exact number of solver calls correspond to the number of existing action schemas (not ground actions), so as to do the usability detection for each and every action. The wrapper receives the usability result of each action after calling the solver. After results for all of the actions are gathered, the plugin calls the parser to parse the result in JSON format and send back to the interface of the editor. The final display on the editor shows which actions are usable and which ones are not.

The planner service at http://solver.planning.domains is used for the individual solver calls. Because there is a strict resource limit placed on the service, this is a sound-but-incomplete approach. To be conservative in our analysis, we consider an action to be usable if the solver does not return an answer in time. This way, our labeling of "unusable" for any particular action is correct. For those actions that are not flagged, they may or may not be usable.

The link provided to an online editor session at the start of this section contains two versions of the domain – one with the correct precondition of the `load-airplane` action and one with the incorrect precondition. The difference is minor, and reflects a common mistake of PDDL modelling (reversing the arguments of a predicate):

```
; Correct
; (at ?obj ?loc)
;
; Incorrect
(at ?loc ?obj)
```

When the plugin is invoked on the incorrect domain, the analysis that is conducted identifies two unusable actions (cf. Figure 2). The problematic action is correctly identified, but note also that the `unload-airplane` action is also identified as unusable. This makes intuitive sense, as you are only able to unload an airplane in this domain if there is something already loaded. Since the initial state doesn't begin with any package loaded, then the unload action becomes unusable as well. Down the road, it would be interesting to perform a deeper analysis on the unusable actions to detect which ones are the root cause of others not being usable. For example, if we were to remove all of the preconditions from the `load-airplane` action, then we would detect that `unload-airplane` is now usable; thus indicating a precedence in the (un)usability of these two actions (note that the reverse is not the case).

Considerations and Discussion

We originally had planned on using a state-of-the-art detector for deadends – specifically, Aidos (Seipp *et al.* 2016) – in order assess if an action was usable. We ultimately decided to forgo the use of this approach for two reasons: (1) hosting the Cplex library, which is required by the deadend detector, would violate the licensing agreement; and (2) we expect the majority of the sub-problems to actually be solvable, and trivially so. We were thus able to make do with direct calls to the online solver.

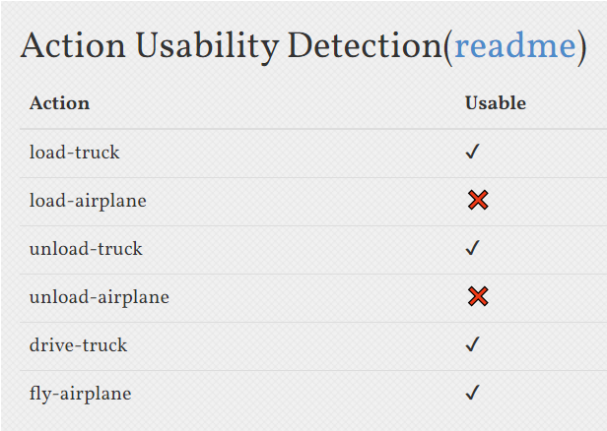

Figure 2: Usability analysis of the actions in a logistics domain with one action's precondition incorrectly specified.

The action usability plugin is an open-source project for educational and research purposes. The plugin project is listed above, and the online service that performs the repeated call can be found at:

https://github.com/QuMuLab/action-usability-via-deadend-detection

We welcome feedback on the project, and continue to improve the interface. In the future, we will expose the action usability result on the editor by automatically highlighting unusable actions, rather than displaying the analysis a new tab. This feature is more user-friendly and in line with common linter functionality found in many popular IDEs.

5 Related Work

The original idea of how to detect unusable actions is similar to the D3WA+ system (Sreedharan *et al.* 2020). It points out a problem caused by the declarative feature of dialogue planning: how hard it is for designers to understand the system without knowing how it works inside in a declarative programming paradigm. They introduce the D3WA+ system which is an updated version of D3WA. D3WA+ adds a suite of debugging tools upon the model acquisition framework using Explainable AI planning (EAIP) techniques. This is used to help users to grasp the imperative consequences to answer questions such as why there is no solution, why this is not a solution, and why this is a solution. To answer the first question, D3WA+ uses repeated solvability checks to find a subset of the domain that is unsolvable or solvable.

In our work, we consider unreachable actions as one of the most common bugs in PDDL modeling. With a similar deadend detection mechanism as D3WA+, we use repeated solvability checks for identifying if particular actions are usable. The two approaches are solving different high-level problems, but are doing so with a similar technique: repeated unsolvability checks on refinements to the planning model. In contrast with the rich history of verification techniques, our approach is specifically designed for the planning setting. We rely on the high-level planning language representation, and the affordances it provides, to detect unusable actions.

6 Summary

Unusable actions are a common symptom of buggy domains. It is difficult and inefficient to do a debugging without any automated assisted tool. In this paper, we presented a simple yet powerful model debugging technique for detecting when actions are unusable, and additionally provided an integrated solution into modern modelling tools. The action usability plugin makes use of the online editor plugin interface, as well as the online solver service. It helps users to quickly understand why there is no valid plan and improves the efficiency of the PDDL modelling process.

When we are able to detect whether an action is usable/reachable, the next step is to consider if an unusable action exists, why it occurs and how to fix it. There may be various reasons for its existence. For example, if there is something wrong with the precondition of the action itself, we can just change the precondition to a suitable one. If the unusable action itself is defined correctly, it might because there are more than one unusable actions, or the effects of other actions are erroneous. Being able to automatically detect and suggest possible fixes for an action being unusable is an exciting area of future research with this line of work.

Another key limitation is the assumption of independence between the actions. It may be that one action will necessarily "disable" the usability of another. This may be intended, but also may be an indication of a modelling error. The techniques we introduce apply equally well to this setting (searching through pairs of lifted actions instead of each action individually).

Additionally, in the future we will consider various types of errors that occur in the PDDL developing process and design a more comprehensive error checking library for PDDL. For instance, *useless* actions are those that are technically usable but will never help in the quest to achieve the goal. Identifying these for the modeller is another signal they can use to introspect the model they are developing.

References

Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool, 2019.

Christian Muise. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*, 2016.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett, and Yusra Alkhazraji. Fast downward aidos. *Unsolvability International Planning Competition: planner abstracts*, pages 28–38, 2016.

Sarath Sreedharan, Tathagata Chakraborti, Christian Muise, Yasaman Khazaeni, and Subbarao Kambhampati. D3WA+: A case study of XAIP in a model acquisition task. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.