Action Inheritance Within the Unified Planning Framework

Alan Lindsay, Ronald P. A. Petrick

Heriot-Watt University, Edinburgh, UK
alan.lindsay@hw.ac.uk

Abstract

The PDDL modelling problem is known to be challenging, time consuming and error prone. This has led researchers to investigate methods of supporting the modelling process. This has included the Unified Planning Framework, which allows planning structures to be specified and manipulated in code. Another recent proposal involved extending the PDDL modelling language with support for action inheritance. The motivation in the extension is to help to organise and provide structure, with an aim to making PDDL models easier to read, debug, and reuse. In this work we extend the Unified Planning Framework in order to support action inheritance. We consider whether action inheritance is useful in the context of the framework; in particular, we compare the use of action inheritance with the manipulation of PDDL structure in the framework. We conclude with a case study, where we demonstrate the benefit of multiple-inheritance.

Introduction

The problem of authoring PDDL domains has been identified as a major bottleneck in the adoption of planning. Traditionally authoring PDDL was carried out entirely in a text editor, but over the years tools and techniques have been developed to support the process. This includes a recent trend in the research field to develop community frameworks, bringing together tools and techniques for problem modelling, planning, and related processes (Muise 2016; Dolejsi et al. 2019). A recent example is the Unified Planning Framework[1] (UPF), which provides an intuitive and planner independent method for defining planning problems. Of particular relevance is that it provides access to planning structure as first-class objects to be manipulated in code.

Planning models can involve substantial duplication (e.g., context dependent actions) and complexity (e.g., models for real world applications). For example, context dependent actions are actions that can have several interpretations, or specific implementations, depending on the context that the action is executed. In the barman domain, there are different actions to fill a shot glass, which depend on whether the shot glass is empty or not. Of course, the actions representing these specific cases will typically share much of their representation. Moreover, redundancy and complexity can reduce legibility and increase the chances of introducing errors during model changes. In recent work, action inheritance was proposed as an extension to PDDL to allow the relationships between actions to be explicitly encoded. As a consequence, related actions can build from a shared representation of a more abstract action, allowing them to share structure. This allows for a more concise representation.

In this paper we consider action inheritance in the context of the UPF. In particular, we consider the comparison of using action inheritance over modelling similar actions using features of the UPF directly. The paper is structured as follows. We first introduce planning, PDDL, action inheritance in PDDL, and the UPF. We then demonstrate how the UPF can be extended to support single and then multiple action inheritance, and discuss their use. We conclude with a case study where we demonstrate how multiple inheritance can be used in practice to assist in incorporating a model of engagement into an existing planning model.

[1]https://github.com/aiplan4eu/unified-planning

Related Work

There are various approaches to supporting authoring PDDL models, including frameworks similar to IDEs for use by software engineers, e.g., the GIPO (Simpson, Kitchin, and McCluskey 2007), itSIMPLE (Vaquero et al. 2007) and KEWI (Wickler, Chrpa, and McCluskey 2014) systems. These modelling tools are useful for rapid development of domains by an experienced domain modeller. Alternatively, approaches to domain model acquisition aim to learn models from observations, e.g., (Wu, Yang, and Jiang 2007; Mourao et al. 2012; Lindsay et al. 2017) and approaches that provide assistance in refining (Lindsay et al. 2020) and extending (Porteous et al. 2021) existing planning models, each aim to reduce the burden of modelling a complete domain model.

The UPF is part of a recent trend in the automated planning research field to develop community frameworks. An early example, 'Planning.Domains' (Muise 2016), brings together a collection of resources and tools, including an online solver, and extensive sets of problem benchmarks. The VS Code PDDL extension (Dolejsi et al. 2019) provides functionality to support PDDL modelling in VS Code in a similar manner to programming languages, such as Python.

Inheritance in PDDL is used in the type hierarchy (McDermott et al. 1998). There are also approaches, e.g., (Hertle

```
(:action move
  :parameters (?t - truck ?l1 ?l2 - location)
  :precondition (and (at ?t ?l1)
    (not (at ?t ?l2)))
  :effect (and (at ?t ?l2) (not (at ?t ?l1))))

(:action constrained_move
  :super (move)
  :precondition (connected ?l1 ?l2))
```

Figure 1: Example PDDL actions for the `move` action and the `constrained_move` refinement (inheriting) action.

```
1  def get_traversal_model():
2    ...
3    Location = UserType('Location')
4    Truck = UserType('Truck')
5    at = Fluent('at', BoolType(), t=Truck, l=
         Location)
6    move = InstantaneousAction('move', t=Truck,
         l1=Location, l2=Location)
7    t = move.parameter('t')
8    l_from = move.parameter('l1')
9    l_to = move.parameter('l2')
10   move.add_precondition(at(t, l_from))
11   move.add_effect(at(t,l_from), False)
12   move.add_effect(at(t,l_to), True)
```

Figure 2: Example Python code for specifying location and truck types, a located predicate, and a move action in UPF.

et al. 2012), that support inheritance in an alternative representation language, which is subsequently compiled into PDDL. Linking supporting modules into PDDL has also been utilised in PMT (Gregory et al. 2012) and PDDL/M (Dornhege et al. 2009). However, in these cases modularity was introduced to mitigate language limitations, or to connect to external functions. In (Lindsay 2023) action inheritance was proposed as an extension to PDDL. In this work we have consider this work in the context of the UPF, and extend action inheritance to multiple inheritance.

Background

A classical planning problem can be defined as follows:

Definition 1. A Classical Planning Problem is a planning problem, $P = \langle F, A, I, G \rangle$, with fluents, F, actions, A, initial state, I, and goals, G. A solution (a plan) is a sequence of actions, $\pi = a_0, \ldots, a_n$, that transform the initial state, I, to a state, s_n, that satisfies the goals, $G \subseteq s_n$.

An action, a, is defined by a precondition ($Precs(a)$) and an effect, which we assume can be separated into sets of add ($Adds(a)$) and delete ($Dels(a)$) effects. An action is applicable in a state if its precondition is satisfied by the state. The aim in classical planning is typically to find short plans (unit cost).

Action Inheritance It is typical in planning to use action schema to specify planning actions (McDermott et al. 1998) (e.g., Figure 1 top). These schema detail the action's name, and first order precondition, and effect. They typically also include an ordered set of parameters (variables), which can used in the precondition and effect. Notice that on making a complete assignment of objects to variables leads to an action (as defined above). In (Lindsay 2023) they presented an extension that supports the use of action inheritance when specifying the planning model. This introduces an optional super slot, which can be used to indicate the super-action. For example, Figure 1 presents a PDDL representation of the `move` action, and the `constrained_move` action, which refines the `move` action with a constraint over traversable edges. If an action a inherits from a^S then the parameters, precondition, and effect of a^S are added to those of a. Please refer to (Lindsay 2023) for more details.

```
def make_restricted_move_model()
  connected = Fluent('connected', BoolType(),
      l_from=Location, l_to=Location)
  constrained_move = InstantaneousAction('
      constrained_move', _super=move)
  constrained_move.add_precondition(connected(
      move.parameter('l1'),
      move.parameter('l2')))

  traversal_problem = get_traversal_problem()

  with OneshotPlanner(problem_kind=
      traversal_problem.kind) as planner:
    result = planner.solve(traversal_problem)
    print("%s returned: %s" % (planner.name,
        result.plan))
```

Figure 3: Example Python code defining the `constrained_move` action, which inherits from the `move` action. The code also defines the `connected` predicate and uses it to add a constraint to the new action.

Unified Planning Framework

The Unified Planning Framework (UPF) was developed as part of the AIPlan4EU project[2], and aims to provide an intuitive and planner independent method for defining planning problems. It is provided as a Python library, allowing the user to define the domain and problem structure using a high-level API. For example, Figure 2 demonstrates how to specify the types, predicate and action for a simple traversal domain (e.g., line 6 defines a new action, defining its name and several typed parameters). UPF provides a convenient front-end to a collection of solvers, and gathers together functions, including model transformations, planning, and plan validation.

Inheritance in the Unified Planning Framework

In this section we present an extension to the standard UPF that allows an inheritance relationship to be defined between

[2]https://www.aiplan4eu-project.eu/

actions (e.g., as was shown in Figure 1). Our approach extends the action definition in the UPF with a *super* field: if the action has a super action then this field is an action object, otherwise it is *None*. This field can be populated as part of the action's instantiation, using a named argument.

In Figure 3, we present part of a script that extends the `move` action defined in Figure 2, with a `connected` predicate, which restricts the valid ground actions. In line 2 the `connected` predicate is defined, with two parameters of type `Location`. Then the constrained action is defined (line 3), using the optional `_super` keyword argument to define the `constrained_move` action as inheriting from the `move` action. Our implementation makes the action hierarchy accessible, supporting a `gather_supers` function for action types, which returns the hierarchy of super actions, ordered from most general to most specific. The hierarchy can be reported in text using the `get_action_hierarchy` function. For example, the function returns '`constrained_move←move`' for the `constrained_move` action.

In line 4, the `connected` predicate is used to add a precondition, restricting the valid `constrained_move` actions. A traversal problem is then generated (line 6), which creates a simple chain of connected locations, with a random initial state and goal location for the traverser. Lines 8-10 demonstrate typical code for selecting a planner and generating a plan using the UPF. Selecting an appropriate planner (line 8) relies on analysing the problem structure and determining the requirements of the model. This analysis was extended to consider the action hierarchies (the chain of super actions) for each action in the problem.

The problem is then solved using the planner (line 9). As part of the solve function, the problem is checked for action inheritance (e.g., it determines whether any actions have super actions), and if they do then the model is compiled before planning. This involves creating an action that includes the parameters, preconditions, and effects of all the actions in the action's inheritance hierarchy (see (Lindsay 2023) for more details). After compilation, the resulting model will not use action inheritance, and standard planners can be used.

In this case, the plan generated by Fast Downwards (Richter and Westphal 2010) was:

- `constrained_move(r0, l5, l6)`
- `constrained_move(r0, l6, l7)`
- `constrained_move(r0, l7, l8)`

which is an optimal plan given the connection constraints.

Multiple-Inheritance in the Unified Planning Framework

In this section, we consider the extension of action inheritance to allow an action to inherit from multiple actions. The approach builds from the previous section, and the *super* field becomes either an action, a list of actions, or None. The main implementation change is in the definition of the `gather_supers` function (see 'Inheritance in the Unified Planning Framework'), which determines how the parameters, preconditions, and effects are composed. In the case of

```
1  def gather_supers(a):
2    hierarchy = list()
3    dfs(a, hierarchy)
4    return hierarchy
5
6  def dfs(a, hierarchy):
7    early_visit(a)
8    for sa in get_supers(a):
9      dfs(sa, hierarchy)
10   late_visit(a, hierarchy)
11
12 def get_supers(a):
13   if a.super == None:
14     return []
15   if isinstance (a, Action):
16     return [a.super]
17   return a.super
18
19 def late_visit(a, hierarchy):
20   hierarchy.append(a)
```

Figure 4: Example code using a depth first search to gather the action hierarchy in the `hierarchy` variable.

preconditions and effects this should make no difference (although it might vary in practice (Vallati et al. 2015)). Our choice is to gather the super actions using a depth first search –using the ordering in the declaration of the actions to determine the order at each node. Figure 4 shows the Python code to gather the actions in order. It is based on a depth first search (lines 6-10), with the actions gathered when the algorithm has finished with the action (lines 19-20). The intuition here is so that the parameters for related aspects are grouped together in the action's parameters. To compile an action once the actions in the hierarchy are gathered, the parameters, preconditions, and effects are composed and a new compiled action is created. We assume that all actions (and their parameters) in the hierarchy are unique.

In the following section we provide an example from a recent Human-Robot Interaction (HRI) project.

Case-Study: Engagement in Human-Robot Interaction

In a related project, we are developing a plan-based social robot system for use in a medical setting (Lindsay et al. 2022; Foster et al. 2023; Lindsay et al. 2024; Ramírez-Duque et al. 2024). As part of that project we have developed a planning model that captures the human-robot interaction. This model was modelled using action inheritance and modularity (Lindsay 2023). The generated interactions capture alternative interaction sequences based on input from the user's preferences and choices, variation in the medical pathway, and sensed valuation of the user's anxiety level. As examples, the model includes an introduction action (`intro`) and a calming action (`am_calm`), which is intended to be used as part of an anxiety management intervention – useful in the type of interactions we are considering. In recent work [submitted], we have also extended this planning model, and

```
(:action  point_of_engagement
  :precondition (not (engaged))
  :effect (engaged))

(:action  sustain_engagement
  :precondition (engaged))

(:action  disengagement
  :precondition (engaged)
  :effect (not (engaged)))
```

Figure 5: PDDL representations of the three actions in the engagement model presented in (O'Brien and Toms 2008).

```
1  def extend_hri_model_with_engagement()
2    ...
3    po_eng = eng_domain.action('point_of_eng')
4    intro = hri_domain.action('intro')
5    eng_intro = InstantaneousAction('eng_intro',
         _super=[po_eng,intro])
6
7    sus_eng = eng_domain.action('sustain_eng')
8    am_calm = hri_domain.action('am_calm')
9    eng_calming = InstantaneousAction('
         eng_amcalm', _super=[sus_eng,am_calm])
```

Figure 6: Python code extending actions from the `hri_domain` with an engagement model (`eng_domain`).

the robot system in order to support the management of user engagement. In this section, we use the example of extending the existing planning model with a model of engagement, as a demonstration of the use of multiple inheritance.

Model of Engagement

The starting point for this is an existing model of engagement (O'Brien and Toms 2008), which identifies three stages: point of engagement, sustained engagement, and disengagement. We can represent the three stage model of engagement in a planning model by using a representation of the user's engagement: the `engaged` proposition that is true when the user is engaged. In Figure 5 we present a PDDL representations of the three stages. The `point_of_engagement` action has the precondition of `engaged` being false and transitions it to true. The `sustain_engagement` action simply insists that the `engaged` proposition holds. And finally, the `disengagement` action transitions the `engaged` proposition back to false.

Extending an Interaction with Engagement

In Figure 6, we present Python code that defines new actions that inherit from both the HRI and engagement models. In lines 3-4, the `intro` and `point_of_engagement` actions are identified, and in line 5 these are used in the definition of a new action, which inherits from both. In lines 7-9, a specific engaged calming action is created, both inheriting from the `am_calm` action, and the `sustaining_engagement` action. The generated type tree for the `eng_amcalm` action is shown here:

```
eng_amcalm <- sustainengagement
              amcalm <- calming <- doactivity
```

This was generated using an extension to the code in Figure 4 using the `early_visit` function.

Discussion

It is worth considering what is natively possible in the UPF without using action inheritance. For example, consider the action `move` from the example above in the context of single inheritance. If we want to extend this action with a `connected` predicate we can clone the `move` action, and then add a `connected` predicate. In this way the UPF library provides a natural method of sharing structure within and between models.

However, from a representational perspective, this approach does not make explicit the connection between the actions. As a consequence, if later the `move` action is changed, the changes will not be reflected in the `constrained_move` action. Internally the PDDL structure is duplicated, and if the user wants the model recorded in an external representation (e.g., PDDL) then this representation will also duplicate the structure. Notice in (Lindsay 2023), it is also demonstrated that the action hierarchy can be useful in reducing the effort of specifying behaviours, and improving the robustness, in a robot system.

The extension to multiple inheritance generalises the approach, allowing an action to inherit from multiple actions. This can allow richer relationships to be represented explicitly, which could lead to more concise representations, and more reuse. As an example, we demonstrated how multiple inheritance can be used to combine two planning models, by extending the actions of an interaction model for HRI with appropriate transitions from a model of user engagement. In this work we have assumed that the actions and parameters are unique in the hierarchy. Further work is required to identify appropriate relaxations of this constraint.

Conclusion

In this paper we have considered action inheritance in the context of the Unified Planning Framework (UPF). We presented an extension to the UPF that supports action inheritance. We extended the approach to allow multiple inheritance, which supports declaring inheritance relationships between actions and sets of actions. The UPF gives access to planning model structure, and can therefore be used to manipulate the structures directly in code. In this work, we observed that even in this context that establishing the explicit inheritance relationship between actions is still a useful modelling concept, which can help in action reuse, debugging, and making a concise representation. We used a case study from human-robot interaction to demonstrate the use of multiple inheritance to demonstrate how an existing planning model can be extended using a model of user engagement.

Acknowledgements

The authors wish to acknowledge the SSHRC-UKRI Canada-UK Artificial Intelligence Initiative (UKRI grant ES/T01296/1) for financial support of this project. We would also like to acknowledge all patient partners, research and clinical staff as well as the youth and families who have made this project possible.

References

Dolejsi, J.; Long, D.; Fox, M.; and Muise, C. 2019. From a Classroom to an Industry From PDDL “Hello World” to Debugging a Planning Problem. In *International Conference on Automated Planning and Scheduling, System Demonstrations*.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Nineteenth International Conference on Automated Planning and Scheduling*.

Foster, M. E.; Candelaria, P.; Dwyer, L. J.; Hudson, S.; Lindsay, A.; Nishat, F.; Pacquing, M.; Petrick, R. P. A.; Ramírez-Duque, A. A.; Stinson, J.; Zeller, F.; and Ali, S. 2023. Co-Design of a Social Robot for Distraction in the Paediatric Emergency Department. In *Companion of the 2023 ACM/IEEE International Conference on Human-Robot Interaction*, HRI ’23, 461–465. New York, NY, USA: Association for Computing Machinery. ISBN 9781450399708.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

Hertle, A.; Dornhege, C.; Keller, T.; and Nebel, B. 2012. Planning with Semantic Attachments: An Object-Oriented View. In *ECAI*, volume 242, 402–407.

Lindsay, A. 2023. On Using Action Inheritance and Modularity in PDDL Domain Modelling. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Lindsay, A.; Franco, S.; Reba, R.; and McCluskey, T. L. 2020. Refining Process Descriptions from Execution Data in Hybrid Planning Domain Models. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*.

Lindsay, A.; Ramírez-Duque, A. A.; Petrick, R. P. A.; and Foster, M. E. 2022. A Socially Assistive Robot using Automated Planning in a Paediatric Clinical Setting. In *Proceedings of the AAAI Fall Symposium on Artificial Intelligence for Human-Robot Interaction (AI-HRI)*.

Lindsay, A.; Ramírez-Duque, A. A.; Petrick, R. P. A.; and Foster, M. E. 2024. A Socially Assistive Robot using Automated Planning in a Paediatric Clinical Setting. In *Proceedings of the International Symposium on Technological Advances in HRI*.

Lindsay, A.; Read, J.; Ferreira, J. F.; Hayton, T.; Porteous, J.; and Gregory, P. J. 2017. Framer: Planning models from natural language action descriptions. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical report, Yale University.

Mourao, K.; Zettlemoyer, L.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Uncertainty in Artifical Intelligence*, 614 – 623.

Muise, C. 2016. Planning.Domains. In *26th International Conference on Automated Planning and Scheduling, System Demonstrations*.

O’Brien, H. L.; and Toms, E. G. 2008. What is user engagement? A conceptual framework for defining user engagement with technology. *Journal of the American Society for Information Science and Technology*, 59(6): 938–955.

Porteous, J.; Ferreira, J. F.; Lindsay, A.; and Cavazza, M. 2021. Automated Narrative Planning Model Extension. *Journal of Autonomous Agents and Multi-Agent Systems*.

Ramírez-Duque, A. A.; Lindsay, A.; Foster, M. E.; and Petrick, R. P. 2024. A Lightweight Artificial Cognition Model for Socio-Affective Human-Robot Interaction. In *Proceedings of the ACM/IEEE International Conference on HRI*.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowledge Engineering Review*.

Vallati, M.; Hutter, F.; Chrpa, L.; and McCluskey, T. L. 2015. On the effective configuration of planning domain models. In *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI press.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. In *International Conference on Automated Planning and Scheduling*, 336–343.

Wickler, G.; Chrpa, L.; and McCluskey, T. L. 2014. KEWI - A Knowledge Engineering Tool for Modelling AI Planning Tasks. In *International Conference on Knowledge Engineering and Ontology Development*, 36–47.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review*.