Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)

On Using Action Inheritance and Modularity in PDDL Domain Modelling

Alan Lindsay

Automated Planning Lab,
Department of Computer Science,
Heriot-Watt University, Scotland, 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. One particular avenue is to adapt tools and techniques that have proven useful in software engineering to support the modelling process. We observe that concepts, such as inheritance and modularity have not been fully explored in the context of modelling PDDL planning models. Within software engineering these concepts help to organise and provide structure to code, which can make it easier to read, debug, and reuse code. In this work we consider inheritance and modularity and their use in PDDL action descriptions, and how these can have a similar impact on the PDDL modelling process. We define an extension to PDDL and develop appropriate tools to compile models using these extensions, both directly from the command line and through the Visual Studio Code PDDL extension. We report on our use of inheritance and modularity when modelling a planning model for a companion robot scenario. We also discuss the benefits of exploiting the inheritance hierarchy in other modules within our robot system.

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. These include frameworks similar to Integrated Development Environments (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. 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.

We observe that concepts that have proven useful in software engineering, such as inheritance and modularity, have not been fully explored in the context of modelling planning models. Inheritance and modularity are used in software engineering as organisational strategies, which can be used to organise complex code into hierarchical structures, simplifying writing and reading code. Inheritance is used in object orientated programming (OOP) (Stefik and Bobrow 1985), and allows groups of similar features that use similar interfaces, to share core code. In the case of OOP, the shared code is captured in an abstract object, and it is written once. Each specific feature extends this abstract object, adding the specific details of the implementation that distinguish the feature. The key is that all of the functionality provided by the shared code is inherited by each of these features. Modularity allows for code to be organised amongst files or modules, with the necessary dependencies explicitly declared between these modules.

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 this work we consider using inheritance and modularity within PDDL domain definitions. We make an extension to the PDDL syntax to allow their use in the definition of domain models and use benchmark domains to demonstrate their use. We present an approach for compiling domain definitions that use inheritance and modularity into standard PDDL, allowing planners to be used on these domains. We present a case study in which we use inheritance and modularity as part of the development of a domain model for a companion robot for use in a medical setting. We also consider how the inheritance structure can be used in other parts of the robot system to increase robustness in the system and reduce redundancy.

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. In this work we chose the Visual Studio PDDL plugin (Dolejsi et al. 2019) to extend with our approach.

Inheritance in PDDL is used in the type hierarchy (McDermott et al. 1998) and action inheritance has been considered previously in (Tenenberg 1989), where the motivation was on reducing the planner's search space through abstraction. In their approach, abstractions are derived retrospectively from a complete model description. There are approaches, e.g., (Hertle 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. Finally, our work has some relation to decomposition in hierarchical task network planning, e.g., (Nau et al. 2003), and object focused approaches (Simpson, Kitchin, and McCluskey 2007).

Background

In this section we introduce the planning background and two benchmark planning domains that will be used to motivate our approach.

Planning Model

The description of a planning problem is separated into two parts: the definition of the planning domain that defines the world and its behaviours; and an explanation of the specific problem to be solved within that world. A domain is a tuple, $\mathbb{D} = \langle \mathbb{T}, \mathbb{P}, \mathbb{A} \rangle$, defining the sets of types, $\mathbb{T}$, predicates, $\mathbb{P}$, and actions, $\mathbb{A}$. Types are defined as a series of pairs of types (indicating type inheritance), which describe a hierarchy, and both action and predicate arguments can be typed. A planning problem is a tuple, $\mathbf{P} = \langle \mathbf{O}, s_{init}, g \rangle$, with the set of typed objects, $\mathbf{O}$, an initial state, s_{init}, and a goal, g, which is a partial state. Predicates and actions are instantiated over the objects of a particular problem (instantiated predicates are called propositions). Actions are represented by three sets of predicates: the precondition (a_{PRE}) and the add (a_{ADD}) and delete (a_{DEL}) effects. An instantiated action, a, is applicable in a state, s, when its precondition is satisfied in s ($a_{PRE} \subseteq s$), and if selected its effects are applied resulting in a new state ($s' = (s \setminus a_{DEL}) \cup a_{ADD}$). We will use $\texttt{eff}(a)$ in this work to denote the combined effects of the action, and assume set operators that appropriately respect the add and delete sets structure. States are sets of propositions and the set of *reachable states* is defined as any state that can be reached from s_{init} through repeated application of applicable actions. A solution to a planning problem, $\mathbf{P}$, is a sequence of instantiated actions (i.e., a plan), $a_1, \ldots, a_n$, which when applied in sequence to s_{init}, leads to a state that satisfies g.

Barman Domain The barman domain is a benchmark domain, introduced in the seventh International Planning Competition (IPC), which includes deterministic actions such as `fill_shot`, `pour_shot_to_clean_shaker` and `shake`. The problems involve creating a selection of drinks, requiring ingredients to be poured into shakers, mixing the drink and the drink being poured out in to shot glasses.

Driverlog Domain Driverlog is a benchmark domain introduced for the third IPC. Problems involve redistributing packages amongst a set of locations, using trucks to collect and deliver the packages. The trucks are operated by drivers and both drivers and trucks navigate constrained maps. The domain includes actions to move trucks and drivers, and pickup and drop off packages and drivers.

PDDL with Inheritance: A Monotonic Refinement Approach

Inheritance in OOP allows for an object —a structure containing code and data— to be based on an existing object (or objects), thus sharing some implementation. This is used in order to organise complex code, and to reduce redundancy and potentially improve robustness through code sharing and reuse. In this section we propose the use of inheritance in the definition of actions. We define the process of action inheritance as action refinement, which allows for related actions to share part of their representation. We examine its benefit in context dependent actions (see the 'Case Study' for another example) and we describe how it is supported through an extension to the PDDL language.

Action Inheritance in Domain Modelling

We extend the definition of actions to allow each action in a domain to inherit from an optional *super* action. Our interpretation is that an action inherits the super action's structure, including parameters, preconditions and effects. Inheritance between action a_1 and its super action a_0, is therefore an explicit declaration that a_1, with parameters $\texttt{params}(a_1)$ preconditions $\texttt{prec}(a_1)$ and effects $\texttt{eff}(a_1)$, is a refinement of a_0.

Definition 1 (An Action Refinement)**.** An action a_1 is a refinement of an action a_0 if $\texttt{params}(a_0) \subseteq \texttt{params}(a_1)$ and $\texttt{prec}(a_0) \subseteq \texttt{prec}(a_1)$ and $\texttt{eff}(a_0) \subseteq \texttt{eff}(a_1)$.

As a consequence, a super action a_0 is at least as general as an inheriting action a_1 (e.g., where a_1 is applicable, then a_0 is applicable). A super action might also inherit from an action, although the resulting chain must not contain cycles.

Context Dependent Actions

We observe that context dependent actions form natural groups, where a set of actions each capture the effects of a specific action under a different context. As a consequence, it is often the case that the actions share part of their representation. For example, the barman domain involves actions, such as filling a shot and pouring a shot to a shaker, where the details of the action depend on the specific context. Figure 1 presents the PDDL representation of the `fill_shot`

```
(:action fill_shot
  :parameters (?s - shot ?i - ingredient
               ?h1 ?h2 - hand ?d - dispenser)
  :precondition (and (holding ?h1 ?s)
    (handempty ?h2) (dispenses ?d ?i)
    (empty ?s) (clean ?s))
  :effect (and (not (empty ?s))
    (contains ?s ?i) (not (clean ?s))
    (used ?s ?i)))
```

Figure 1: The `fill_shot` action from the barman benchmark domain. Context shared with the `refill_shot` action highlighted in red.

```
(:action base_fill_shot
  :parameters (?s - shot ?i - ingredient
               ?h1 ?h2 - hand ?d - dispenser)
  :precondition (and (holding ?h1 ?s)
    (handempty ?h2) (dispenses ?d ?i)
    (empty ?s))
  :effect (and (not (empty ?s))
    (contains ?s ?i)))

(:action fill_shot
  :super (base_fill_shot)
  :precondition (clean ?s)
  :effect (and (not (clean ?s))
    (used ?s ?i)))

(:action refill_shot
  :super (base_fill_shot)
  :precondition (used ?s ?i))
```

Figure 2: The `fill_shot` and `refill_shot` actions represented using a super action `base_fill_shot`.

action, which involves filling a shot with an ingredient from a dispenser. There are two variants of the `fill_shot` action. The `fill_shot` action requires that the shot glass is clean, whereas the `refill_shot` action requires that the shot glass has already been used. Figure 1 demonstrates the majority of the parameters and predicates are shared between the `fill_shot` and `refill_shot` actions (highlighted in red).

Action inheritance can therefore be used to build abstract actions that capture core structure and conceptual hierarchies in the action implementation. In the barman domain we could propose a new `base_fill_shot` action, which captured the main impact of filling the shot (see top of Figure 2). Both the `fill_shot` and `refill_shot` actions would be refinements of this action, each adding the detail required for implementing the action in a different context.

PDDL Action Inheritance Syntax

An extension to PDDL is proposed, introducing a new *:inheritance* keyword into the PDDL requirements, and the *super* keyword to be used as an optional entry in action descriptions. It allows the modeller to declare that an action inherits from a super action. The interpretation of an action a_1 inheriting from an action a_0 is that a_1 is an action refinement of

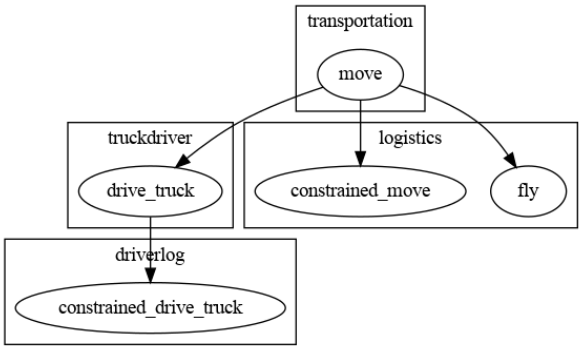

Figure 3: A graph showing a partial inheritance tree for the `move` action in a simple transportation domain. The boxes indicate different modules, each of which may be associated with problem files.

a_0, and as a consequence will include all of the parameters, preconditions and effects of a_0. Figure 2 presents the use of inheritance in PDDL (additional syntax emphasised in red) in order to declare the `fill_shot` and `refill_shot` using a super action `base_fill_shot`, which provides the core functionality.

The declaration of inheritance between actions can be used to create an action hierarchy, which is a directed acyclic graph that captures the relationships between actions in the domain. For example, Figure 3 presents part of the action hierarchy for the actions that inherit from a basic `move` action from a transportation domain. The `move` action freely moves a transporter between locations, the `drive_truck` action adds the requirement that a driver is in the truck and the `constrained_drive_truck` action also adds a map.

In the current version we allow single and multilevel inheritance and we do not allow cyclical dependencies. We also assume that action names are distinct. Explicit declaration of the super action and the definition of inheritance as action refinement means that any path in the action hierarchy will always describe a monotonic refinement, with predecessors always describing more general actions.

Modularity

Modularity in software engineering allows a separation of code into separate modules, which can allow organising code into conceptual units, or providing alternative implementations for a certain aspect. In this section we consider its use within PDDL and our proposed syntax for supporting modularity.

Modularity in Domain Modelling

Our interpretation of modularity is to allow the definition of a domain model to be separated into various domain model files with explicit dependencies made between these partial definitions. A declaration of dependency between a domain d_i and d_{i-1}, means that structure defined in d_{i-1} is available in d_i. The type and predicate definitions in d_i are extended

by the definitions in d_{i-1}. The action definitions in d_{i-1} are available for extension in d_i, but are not part of its domain definition. In particular, action inheritance is through explicitly defining an inheritance relationship, as was described above. We allow action definitions to declare inheritance between an action a_i in domain d_i and an action a_{i-1} from domain d_{i-1}, where domain d_i is dependent on domain d_{i-1}.

Our approach to modularity allows a domain model to declare a dependency for multiple domain models, where the dependency hierarchy of a domain must be representable by a directed acyclic graph.

Modularity in Transportation Domains Transportation underlies many planning domains, including briefcase, zeno, driverlog, logistics, and trucks. In (Long and Fox 2002) the authors identified a behaviour hierarchy that explained the existing transportation domains. This included alternative implementations, such as dynamic and static maps, and refinement of behaviour, e.g., from a mobile (e.g., a truck), to driven mobile (e.g., a truck that needs a driver). Currently, this hierarchy is instantiated in a variety of individual domain models. However, several partly share the same, or very similar structure. We observe that modularity provides a framework for developing a series of modules that can be combined in different ways. This allows these overlaps in structure to be extracted into modules and reused.

The basic structure of a transportation domain can be defined (e.g., consider an extended briefcase domain with explicit transporters), including actions representing moving a transporter (a carrier), and loading and unloading to/from the transporter. This basic structure can then be extended in different domain modules. For example, we can define the truckdriver domain that introduces drivers into the domain. This extension module adds an additional constraint to the move action (i.e., that a driver must be in the transporter), and adds actions for a driver to board and disembark the transporter, and to walk between locations. The truckdriver domain can then be further extended to the driverlog domain (see the 'Background' section) by adding maps to restrict the movement of trucks and drivers. From this point we may also extend the model with alternative cost model or temporal interpretations (e.g., see domains in the third IPC (Fox and Long 2002)). Alternatively, the logistics benchmark domain, would extend the basic transportation domain with alternative typed transporters with constrained maps. Figure 3 presents a graph of actions that inherit from the `move` action in the transportation domain, and indicates how these are organised into different modules. This demonstrates one way that modules can help sharing model structure, even between different domains.

Supporting Modularity in PDDL

In order to support modularity we allow model aspects to be organised in different domain files. We introduce a new *:modularity* keyword into the PDDL requirements and we add a *dependencies* entry in the PDDL definition, which is used to connect to additional domain files. For example, the graph in Figure 3 shows that action `move` is defined in a domain called transportation, which captures the basic transportation actions (e.g., move, load, unload). The truckdriver domain extends this domain —it includes an entry (:dependencies transportation.pddl ..)— and declares an inheriting action `drive_truck`.

Algorithm 1: ACTION COMPILER: Given an action a from domain $\mathbb{D}$ recursively build compiled action $\hat{a}$.

function ACTIONCOMPILER($\mathbb{D}, a, \hat{a}$)
  if $\texttt{super}(a)$ **then**
    $\mathbb{D}', a^{super} \leftarrow \texttt{findAction}(\texttt{super}(a))$
    $\texttt{ActionCompiler}(\mathbb{D}', a^{super}, \hat{a})$
  end if
  $\texttt{params}(\hat{a}) \leftarrow \texttt{params}(\hat{a}) \cup \texttt{params}(a)$
  $\texttt{prec}(\hat{a}) \leftarrow \texttt{prec}(\hat{a}) \cup \texttt{prec}(a)$
  $\texttt{eff}(\hat{a}) \leftarrow \texttt{eff}(\hat{a}) \cup \texttt{eff}(a)$
end function

Algorithm 2: MODULE COMPILER: Given a domain model, $\mathbb{D}$ with possible dependencies $\texttt{dep}(\mathbb{D})$, returns the compiled model $\hat{\mathbb{D}}$: in standard PDDL.

function MODULECOMPILER($\mathbb{D}$)
  $\hat{\mathbb{T}^{\mathbb{D}}} \leftarrow \mathbb{T}^{\mathbb{D}} \cup \bigcup_{\mathbb{D}' \in \texttt{dep}(\mathbb{D})} \hat{\mathbb{T}^{\mathbb{D}'}}$
  $\hat{\mathbb{P}^{\mathbb{D}}} \leftarrow \mathbb{P}^{\mathbb{D}} \cup \bigcup_{\mathbb{D}' \in \texttt{dep}(\mathbb{D})} \hat{\mathbb{P}^{\mathbb{D}'}}$
  $\hat{\mathbb{A}^{\mathbb{D}}} \leftarrow \texttt{list}()$
  for all $a \in \mathbb{A}^{\mathbb{D}}$ **do**
    $\hat{a} \leftarrow \texttt{Action}(\texttt{name}(\texttt{a}))$
    $\texttt{ActionCompiler}(\mathbb{D}, \texttt{a}, \hat{\texttt{a}})$
    $\hat{\mathbb{A}^{\mathbb{D}}} \leftarrow \hat{\mathbb{A}^{\mathbb{D}}} \cup \{\hat{a}\}$
  end for
  $\hat{\mathbb{D}} \leftarrow \texttt{Domain}(\hat{\mathbb{T}^{\mathbb{D}}}, \hat{\mathbb{P}^{\mathbb{D}}}, \hat{\mathbb{A}^{\mathbb{D}}})$
  return $\hat{\mathbb{D}}$
end function

PDDL Action Inheritance Toolkit

We have developed both command line and IDE based toolkits for managing domain models that use inheritance and modularity. The core functionality is a model compiler (implementing Algorithms 1 and 2), which takes a domain file (with possible dependencies to other domain files) and flattens it to a single domain file with no inheritance, allowing the model to be used by standard PDDL planners. However, we also include standard IDE features, and useful visualisations, to assist in the design and maintenance process. In this section we present these tools.

Compiling a Domain Model with Modularity and Inheritance

In order to create a single domain file, suitable for standard PDDL planners, we implement a compilation step, which builds the model from the modules in the dependency hierarchy. The pseudocode for the compilation is presented in function `ModuleCompiler` in Algorithm 2). For a domain $\mathbb{D}$, the dependencies are denoted $\texttt{dep}(\mathbb{D})$ and the compiled domain is denoted $\hat{\mathbb{D}}$. The definition of types, predicates and

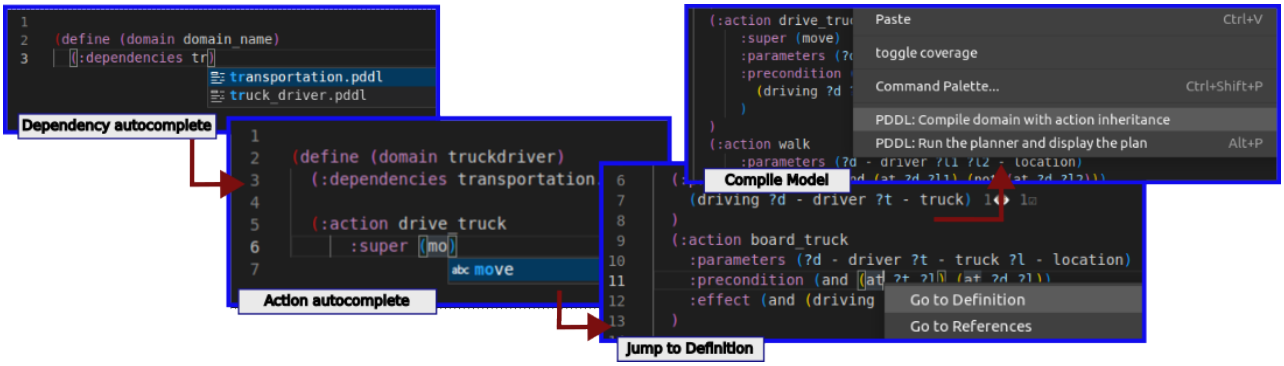

Figure 4: Screenshots from the extended VS Code PDDL extension.

actions within a domain file are denoted $\mathbb{T}$, $\mathbb{P}$, and $\mathbb{A}$, while the compiled sets are denoted $\hat{\mathbb{T}}$, $\hat{\mathbb{P}}$, and $\hat{\mathbb{A}}$. For compiling types (line 2) and predicates (line 3), the definition recursively combines (set union) the definition in the current domain with the compiled definition from each of the domain modules that the current domain depends on. As a result the compiled domain includes the type and predicate definitions from all domains in the dependency hierarchy. For each action, a new action is created retaining the action's name and an empty list of parameters and empty sets of preconditions and effects. The action is compiled (line 7) using the procedure presented in Algorithm 1. These compiled structures are then combined to create the compiled domain model, which is then returned (lines 10-11).

Each action is compiled (`ActionCompiler` in Algorithm 1) by recursively building its representation by descending the inheritance hierarchy (lines 2-4). If the current action references a super action (line 2) then the function `findAction` resolves the reference and returns both the action and its module (line 3). This is achieved by examining the current domain and all domains in the dependency hierarchy. Each of the current action's parameters, preconditions, and effects are then added (lines 6-8) to create the compiled action representation. The approach is organised so that structure from the more general actions is ordered first (in particular, parameters are added in order).

Within the PDDL VS Code Extension

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. For domain modelling this includes syntax highlighting, code snippets, autocompletion, hover over tooltips and jump to definitions. These features rely on lightweight parsing, operating on partial documents. This approach exploits the expected structure of PDDL definitions in order to build an interpretation over a partial model description and it is capable of maintaining key information about the model, including types, predicates and actions.

We have built on the VS Code extension to support editing multi-module domain models that use action inheritance. Figure 4 illustrates some of the functions that are implemented, including highlighting, autocomplete and jump to definition, and their use in the creation of the truckdriver domain (see 'Modularity'). In order to support the additional syntax, the parsing, syntax highlighting and code snippets were extended. Supporting the autocompletes and jump to definition required that the scope of relevant content to be redefined to include appropriate content from relevant modules. This relies on maintaining an up-to-date interpretation of each of the domain files that is currently in the dependency hierarchy for the domain in focus. The potentially recursive tree of dependencies is then descended to gather the appropriate scope. For example, in the case of creating the list of autocompletes for an action effect, the system builds the list from the predicates in the current domain file and all of the domains that the domain depends on. In the case of autofilling the dependencies, only `.pddl` files in or under the current model's directory are suggested.

Command Line Tool

We have also developed an alternative command line approach, which includes a parser, a model compiler, and several visualisation tools (e.g., see Figure 7). As in the VS Code extension, the core functionality is a model compiler, which takes a domain model file (with possible dependencies to other domain files) and flattens it to a single model with no inheritance, allowing the model to be used by standard PDDL planners. For example, Figure 1 presents the `fill-shot` action, compiled from the schema in Figure 2. This tool provides support to non-VS Code users.

Case Study: A Companion Robot

In this section we use a case study to examine the use of inheritance and modularity in modelling a PDDL domain model. We first introduce a project that aims to develop a companion robot for children during painful procedures. We introduce the scenario and overview our current system. We then describe the model that we have developed and how inheritance and modularity were used to organise the model's complexity. Finally, we describe the use of the model's structure in other parts of the system.

A Clinical Setting Scenario

Children regularly experience pain and distress in clinical settings, which can produce negative effects in both the short

```
(:action do_activity
  :parameters (?a - activity)
  :effect (done_activity ?a))

(:action do_calming_activity
  :super (do_activity)
  :precondition (calming ?a)
  :effect (hascalmed))

(:action calm_during_anxiety_management
  :super (do_calming_activity)
  :precondition
  (and
    (amrequirescalming)
    (amperforminganxietymanagement))
  :effect
  (and
    (not (amrequirescalming))
    (amrequiresanxietyretest)))
```

Figure 5: Example PDDL actions for the `do_activity` action and two refinement (inheriting) actions.

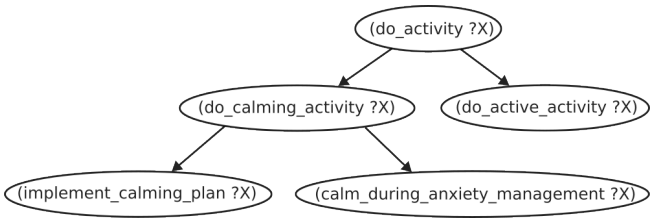

Figure 6: Part of the action hierarchy for the `do_activity` action.

term (e.g., fear, distress, inability to perform procedures) and the long term (e.g., needle phobia, anxiety) (Stevens et al. 2011). Recent studies have demonstrated that social robots can be used to manage child pain and distress during medical procedures (Ali et al. 2019; Trost et al. 2019). We are part of a team developing a social robot underpinned by a planning system to be used in this setting (Foster et al. 2020; Lindsay et al. 2022). The scenario presents a significant challenge for a social robot: the system must coexist with multiple humans engaged in numerous high-priority and dynamic tasks. The robot behaviour must be sensitive to the situation, as inappropriate behaviour may impact patient safety and wellbeing.

To address these challenges, we underpin the robot's behaviour with an automated planning system that uses observed social signals, together with the robot's state, to select appropriate behaviour: the planner makes high-level decisions as to which spoken, non-verbal, and task-based actions should be taken next by the system. Within this context, the robot must be able to adapt to different roles throughout the interaction, including mediator (e.g., introducing or explaining parts of a procedure), an assistant (e.g., performing actions alongside humans), and tutor/interviewer (e.g., in the debrief phase). The planning model must represent knowledge and constraints from the robot, the medical setting and the interaction, adding to the complexity of its representation.

Our System Overview

Our system architecture is composed of several components, including social signal processing, an interaction manager, a planning system, and a robot platform. The target robot platform is the SoftBank NAO, which is a humanoid robot with 25 degrees of freedom, which enables it to move and perform a large variety of actions. At the centre of the architecture is the interaction manager, which ensures synchronised transitions between the internal states of the system/robot. The interaction manager integrates the information from the social signal components to estimate the affective state. It also makes requests of the planning module, which is used during the interaction to determine the next action based on the current state and the goal. Finally, the social stimuli module interprets high-level actions and generates specific signals for each communication channel, whether through synthesised speech or non-verbal communication through gestures and body language.

The Planning Model

The planning model underpins the interaction, and splits the interaction into six stages: from introduction and preprocedure, optional site check, through the procedure, debrief and goodbye. The model includes actions that represent robot behaviour scripts (e.g., dancing, providing information or meditation), and sensing actions (e.g., for anxiety and engagement levels, or for user preferences). The model also captures procedural knowledge, which includes strategies for anxiety management, making a plan with the user for diversion during the procedure, and managing the social interaction. An example of this procedural knowledge is demonstrated by the `(do_activity meditation)` action. This action can be performed as a straightforward diversion during the interaction (e.g., see top of Figure 5). However, it also can form a component of the anxiety management procedure –a procedure that combines certain specific interventions with monitoring of social signals, with an aim of managing the patient's anxiety. In this case, the action has additional conditions and effects to ensure it is carefully placed within the procedure (e.g., see bottom of Figure 5).

The final domain model is organised into 13 modules (see Figure 7). The compiled domain declares 34 actions, of which 30 use at least a single layer of inheritance. Throughout the modules there are 52 inheritance declarations, and the action hierarchy has a maximum depth of five. The average number of predicates involved in action declarations in the compiled model is 7.62, while it is 3.32 in the modules, providing a measure of how inheritance is impacting on the complexity of the representation.

Utilising Action Inheritance in Model Authoring

During the design of the model it became apparent that the relatively small number of NAO, web-server and internal actions, each had several interpretations during the interactions, leading to action models with considerable repetition

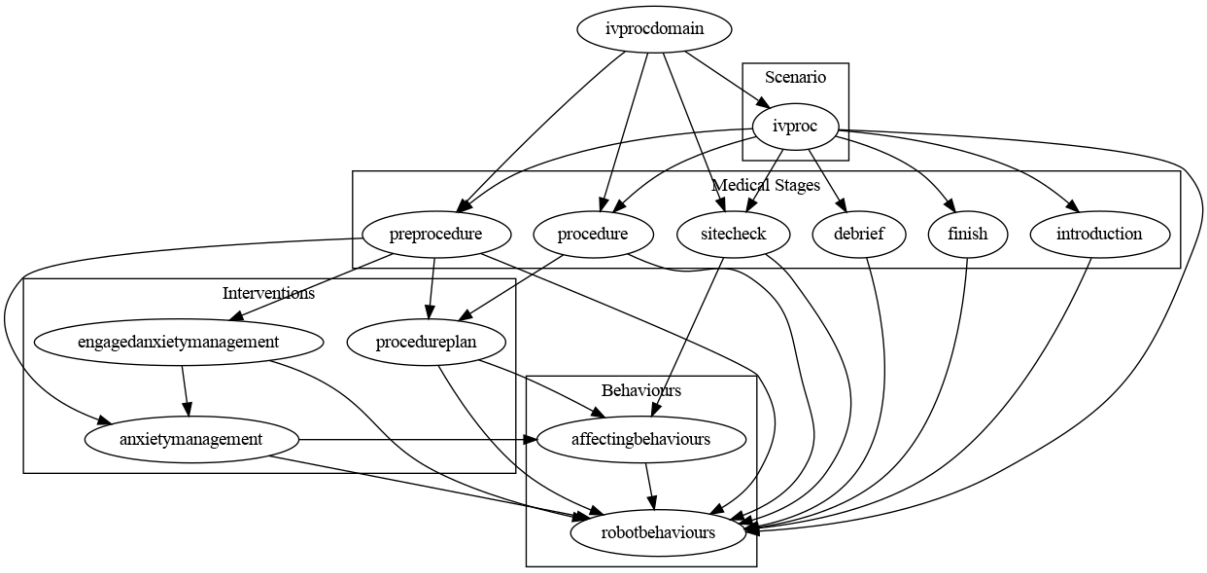

Figure 7: A graph illustrating the inheritance between different modules in our companion robot domain model.

between them. As an example, we can consider actions used in the anxiety management procedure, which is defined in the `anxietymanagement` module, and brings together the testing and management of the patient's anxiety level. This intervention requires refinements to testing, and robot activity actions that ensure their use in the required way for the intervention. For example, the intervention can include the `calm_during_anxiety_management` action, which is part of the action hierarchy for the robot activity action (see Figure 6). Figure 6 presents a fragment of the action hierarchy for the robot activity actions (`do_activity`), and identifies five actions, which are used in different contexts within plans in our domain. In the original model these actions each repeated parts of their representation. This made debugging the model challenging as changes had to be copied to all of the relevant actions, and identifying the important differences between actions was not always obvious.

We have used action inheritance in our model, allowing the sharing of action representation, and improving the model's organisation. Figure 5 demonstrates the use of inheritance in three actions in the `do_activity` action hierarchy. These actions demonstrate how inheritance can be used to build up abstraction layers. The use of inheritance also has practical implications on how the modeller interacts with the representation: firstly changes to the model can be implemented at a single layer in the hierarchy, and those changes impact on all of the inheriting actions directly; secondly, the size of the part of the precondition and effect described at each level is often small (e.g., see 'The Planning Model'), simplifying the comparison between different actions, which can assist in debugging.

Using Modularity in Model Authoring

During modelling we used separate domain files to separate out certain aspects of the model. The domain model is divided into 13 modules (see Figure 7). At the bottom are the basic behaviours (e.g., `do_activity` in Figure 5). Building on these basic behaviours are interventions (such as the anxiety management procedure), and their associated actions (e.g., see `calm_during_anxiety_management` in Figure 5), that combine the behaviours towards some purpose. Each of the stages of the medical procedure has its own domain file, which refine actions from the lower levels to ensure that certain requirements for the stage are fulfilled. For example, during the preprocedure the robot will have a number of goals, such as establishing a diversion plan with the patient (an agreement of how the robot can help during the procedure), and managing their social signals (e.g., performing an appropriate intervention if they are anxious). The support for modularity has allowed us to organise the representation and develop individual aspects of the procedure separately. It allows the potentially complex structure that is necessary to constrain parts of the procedure to be isolated to relevant modules. This has simplified debugging and has made reading the PDDL more manageable.

Using the Model in Planning

The compilation process (Algorithm 2) was used to compile the modules into a single domain model in standard PDDL. Problem models were then defined for the individual scenarios, and solved using a standard planner.

Exploiting the Hierarchy in a Robot System

The motivation for using inheritance in the description of actions was to support the modelling process. Although not the main motivation, we have found the action hierarchy —an output of the use of inheritance in the PDDL model— useful in our current companion robot project system. We now consider how the hierarchy has led to more concise code and

improved robustness in the action selection and interaction manager components.

Exploiting Levels of Granularity for Defining Behaviour The action selection and implementation component of our robot system takes the planning actions and selects appropriate low-level behaviours to be executed on the robot. As a consequence, each time a new action is implemented, which is often simply to reflect a new context, the component needs updated to include an appropriate mapping for the action.

Of course, in many cases, including e.g., context dependent actions, the mapped behaviours will often be the same. The hierarchy therefore provides a natural structure, allowing the robot behaviours to be associated with actions at different levels of the hierarchy. For example, in the barman domain, an appropriate behaviour could be attached to the `base_fill_shot` action (see Figure 2), as the required behaviour will be the same whether the shot glass is used or clean.

In order to implement the action selection, the component first reads the hierarchy information, which is stored within a central parameter server. It is static (generated from the domain model), so it can be read once and used throughout the interaction. When required to select a behaviour the component ascends the action hierarchy stopping at the first action that has a mapped behaviour. This can reduce redundancy, while allowing refinement of behaviours where necessary.

Robustness to Missing Action Specification The interaction manager brings together the other components of the system. As part of its role, it makes calls to parts of the system (e.g., a planner, a web-server and sensors) and makes internal updates, which can impact on the planner's state. As with the action selection component, we have exploited the action hierarchy when defining these behaviours. For example, the required steps for the anxiety test action (a sensing action that determines whether the patient's anxiety level is OK) is associated with a base level action, and context dependent refinements (e.g., an anxiety test during the anxiety management procedure) use the same behaviour.

As a consequence, when the system is run with a new refining action, even if it has not been explicitly specified in the system, the system can still operate, and provide some functionality. For example, when developing a particular scenario a base action `wait`, which idles until it receives an input, was extended with an action that provided two options (continue or abort). When the new action was selected by the planner, the system implemented a typical wait action and was able to continue. The specialised behaviour could then be attached later. The use of the hierarchy is therefore providing some robustness, allowing the specification of default behaviours, which can catch actions that have not been associated with more specific behaviour.

Discussion

The extension provides modelling support, and has minimal impact on performance: compilation is trivial (dominated by parsing), and the model combines the structures defined in the modules (no additional structure is required for the compilation). Of course, the modeller may choose to make a more complete representation of each abstraction layer. However, modern planners are typically effective at identifying and removing redundant and irrelevant structure during preprocessing.

Another aspect of OOP that might be worth considering is encapsulation, which might be used in e.g., privacy preserving planning (Maliah, Shani, and Brafman 2016). In particular, we may wish to declare certain actions and predicates as private, so that they are hidden from inheriting classes. Options for the syntax would include special private/public modal operators in domain definitions, or dedicated slots for visibility. Similarly, supporting a distinction for predicate declarations between being visible (can be used in dependent domain precondition) and modifiable (can be used in dependent domain effects) might be worthwhile.

It might also be useful to declare types, actions and domains, as abstract or final. Within the context of the current work, the main use of declaring these elements abstract would be in indicating the modeller's intention (e.g., that these elements should not be associated with objects, plans and problems respectively), rather than to force structure on inheriting structures. We currently support actions to be declared as abstract, meaning that they are omitted from the compiled domain (but can still be inherited from). For example, `base_fill_shot` in Figure 2 can be declared as an abstract action using `abstract-action`.

Conclusions and Future Work

In this work we have considered the use of inheritance between action descriptions and modularity in the Planning Domain Definition Language (PDDL). We proposed an interpretation of action inheritance as action refinement, and the definition of domain models as a collection of separate modules. We used benchmark domains to demonstrate how inheritance and modularity can be used to support sharing representation in different domain models and in different actions in the same domain. An extension to PDDL was proposed to allow inheritance between planning actions and dependencies to be defined between domain model modules. We have built tools (both command line and as part of the VS Code PDDL extension) to support modelling, and to enable these domains to be compiled to standard PDDL domains. We presented a case study in which we are using a planning model to underpin action selection in a robot companion system. We demonstrated how our approach has supported the organisation of knowledge and constraints relating to different aspects of the scenario, and allowed the description of planning actions to be refined through inheritance, removing redundancy and improving the legibility of the actions. The resulting action hierarchy has also proven useful within other parts of the robot system, making its implementation more concise and robust. In the future we will consider how other types of inheritance and other organisational techniques from software engineering can be utilised to impact on the PDDL authoring process and whether the structure introduced through the use of modularity and inheritance can be exploited in planning or related processes e.g., organising explanation content (Lindsay 2019).

Acknowledgments

The authors wish to acknowledge the SSHRC-UKRI Canada-UK Artificial Intelligence Initiative (UKRI grant ES/T01296/1) for financial support of this work.

References

Ali, S.; Manaloor, R.; Ma, K.; Sivakumar, M.; Vandermeer, B.; Beran, T.; Scott, S.; Graham, T.; Curtis, S.; Jou, H.; et al. 2019. LO63: humanoid robot-based distraction to reduce pain and distress during venipuncture in the pediatric emergency department: a randomized controlled trial. *Canadian Journal of Emergency Medicine*, 21(S1): S30–S31.

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 *Proceedings of the International Conference on Automated Planning and Scheduling*.

Foster, M. E.; Ali, S.; Litwin, S.; Parker, J.; Petrick, R. P. A.; Smith, D. H.; Stinson, J.; and Zeller, F. 2020. Using AI-Enhanced Social Robots to Improve Children's Healthcare Experiences. In *Social Robotics*.

Fox, M.; and Long, D. 2002. The Third International Planning Competition: Temporal and Metric Planning. In *AIPS*.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the 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 *Proceedings of the European Conference on Artificial Intelligence*.

Lindsay, A. 2019. Towards Exploiting Generic Problem Structures in Explanations for Automated Planning. In *Proceedings of the International Conference on Knowledge Capture*.

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 International Conference on Automated Planning and Scheduling (ICAPS)*.

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

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)*.

Long, D.; and Fox, M. 2002. Planning with generic types. In Lakemeyer, G.; and Nebel, B., eds., *Exploring Artificial Intelligence in the New Millennium*, Morgan Kaufmann Series in Artificial Intelligence, 103–138. Morgan Kaufmann.

Maliah, S.; Shani, G.; and Brafman, R. 2016. Online Macro Generation for Privacy Preserving Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

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 *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of artificial intelligence research*, 20: 379–404.

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

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowledge Eng. Review*, 22(2): 117–134.

Stefik, M.; and Bobrow, D. G. 1985. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4): 40.

Stevens, B. J.; Abbott, L. K.; Yamada, J.; Harrison, D.; Stinson, J.; Taddio, A.; Barwick, M.; Latimer, M.; Scott, S. D.; Rashotte, J.; et al. 2011. Epidemiology and management of painful procedures in children in Canadian hospitals. *Cmaj*, 183(7): E403–E410.

Tenenberg, J. D. 1989. Inheritance in Automated Planning. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*.

Trost, M. J.; Ford, A. R.; Kysh, L.; Gold, J. I.; and Matarić, M. 2019. Socially assistive robots for helping pediatric distress and pain: a review of current evidence and recommendations for future research and practice. *The Clinical journal of pain*, 35(5): 451.

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

Wickler, G.; Chrpa, L.; and McCluskey, T. L. 2014. KEWI - A Knowledge Engineering Tool for Modelling AI Planning Tasks. In *Proceedings of the 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*, 22(2): 135–152.