Specifying State Abstractions and Representation Mappings

Ronen I. Brafman, Or Wertheim
Department of Computer Science
Ben Gurion University of the Negev
{brafman,orwert}@post.bgu.ac.il

Abstract

Languages are used to describe diverse aspects of planning formalism: classical domains (PDDL), stochastic domains (RDDL), hierarchical task networks (HDDL), and more. In this paper, we suggest that another component of planning-based systems – state representation mappings – should be singled out and specified explicitly so that it can be used and manipulated by other programs to provide added value. Our main motivation is the automated integration of planning and execution, where this mapping connects the more abstract, descriptive planning model with the actual code within the system that implements it. However, the language could also be used to describe mappings between different planning state spaces and, possibly, domain models. This paper motivates the need for state mapping languages and describes and illustrates a concrete language we developed.

Introduction

Planning systems are typically developed with the goal of being used to control systems in order to enhance these system's autonomy. The planner reasons about the impact on the environment of the various operations the system can perform and how they can lead to a goal state or desirable behavior. Eventually, a controller must carry out the operations recommended by the planner. However, while the planner manipulates relatively abstract descriptions that model the action's impact on properties of interest to the user, the system implements it using code in a programming language. This code typically manipulates less abstract variables. For example, if we implement an *open-door* routine on a robot, the planning model will usually model it in terms of propositions like *door-closed, door-open, door-locked, has-key, in-room(X)* etc, while the code for opening the door will be concerned with properties such as the robot's *position* and *pose*, *arm-joint angles, gripper status* and various properties of images or sonar readings. Similarly, the parameters of the planner model may be some door identifier or may assume the robot is facing the relevant door, while the code may require precise coordinates of the door and its handle in the robot's frame of reference.

Abstraction occurs not only when we map descriptive representations to procedural representations but also when we map between different levels of descriptive representations as we try to simplify a model in order to make it easier to solve. Indeed, abstraction mappings are the basis for popular heuristics such as the pattern-database heuristic (Culberson and Schaeffer 1998; Edelkamp 2001) and the merge-and-shrink heuristic (Helmert et al. 2014).

In this paper, we argue for the development of explicit descriptions of state mappings and describe one such language we developed. There are diverse languages for specifying planning models, starting from STRIPS (Fikes and Nilsson 1971), PDDL (Fox and Long 2003; Gerevini et al. 2009), PPDDL (Younes and Littman 2004) and RDDL (Sanner 2010). There are also languages for specifying action abstractions as HTN planning domains, such as HDDL (Höller et al. 2020), but we are unaware of any language for specifying state abstractions.

The reason we need a language for describing state mappings is as input to programs that manipulate these mappings and provide automation, standardization, and other added values. Whereas the programs that manipulate planning domain descriptions are, typically, planners, the primary use for state mappings is integration: generating code that can connect planners to systems. Presently, when one wishes to use a planner to control a system, one must manually write code that integrates the two systems: the planner and the controlled system. While the mapping itself may be relatively simple, the integration code can be complex and system-dependent. Once we have an agreed-upon language for these mappings, we can automate the integration process, greatly reducing the software engineering effort required to connect planners with systems.

More specifically, the main motivation for the language described here is recent systems developed for simplifying the integration of planning and robotics, especially robots that use ROS (Quigley et al. 2009) as their infrastructure, starting with the pioneering ROSPlan system (Cashmore et al. 2015). Robot code that implements various capabilities, often referred to as *skills*, such as navigation, diverse types of manipulation, and sensed-data analysis (e.g., object and face recognition), is becoming more and more widely available. ROSPlan, and many follow-up systems (e.g., (Martín et al. 2021; Rovida et al. 2017; Rao, Hu, and Jiang 2020; Albore et al. 2023; Doychev et al. 2021)) make it easier to build planner-based task-level controllers that can automatically activate such code as needed. Yet, many of these systems still require writing explicit mapping and integration code.

The AOS system (Wertheim, Suissa, and Brafman 2024) addresses this issue by adding an explicit state-mapping model to the planning model. Using this model, it is able to fully automate the integration process, greatly simplifying and reducing user effort.

The main contribution of this paper is to point out the need for an agreed-upon representation of state mappings, to suggest that they be specified explicitly and separately from the code that uses or embodies them, and to illustrate a candidate language. The article describing the AOS system (Wertheim, Suissa, and Brafman 2024) demonstrates the utility of this approach by describing diverse implemented use cases. through applications programmed using the AOS system.

Our focus within this paper, and the main current application discussed, is mapping descriptive models to procedural code. However, state mappings can also be used to specify abstractions, and we believe a promising future application of such a language could be as a target language for abstraction learning algorithms.

In the rest of this paper, we describe the abstraction mapping language we developed, which we denote by AM. AM was developed in the context of factored POMDP models, described next, of which classical deterministic models are a special case. Moreover, in POMDPs, one must model not only the mapping from planning actions to the code implementing them but also from code values to POMDP observations. While, in principle, this latter mapping is not needed in the context of deterministic classical models, in practice, all systems that rely on classical models realize that their model is imperfect and provide some way of updating the state based on observations. Most systems require the user to write explicit code for this.

Related Work

The two most closely related works known to us are: semantics attachments in planning (Dornhege et al. 2009) and embedded system bridges (ESBs) (Sadanandam et al. 2023).

Semantic attachment were introduced in the FOL reasoning system (Weyhrauch 1980) as a way of using LISP code to evaluate the value of predicates. They were adopted by (Dornhege et al. 2009) for use in planning. Instead of checking the validity of some ground predicate by checking an explicit list of all true ground predicates, as done in typical planners, a procedure (e.g., a path planner) is called to evaluate a predicate (e.g., *reachable(config1,config2)*) and return the ground predicate's truth value. The procedure is called during the planning phase and is used in forward-search planners during as part of planning. The novelty in (Dornhege et al. 2009) is the explicit extension of PDDL with the ability to specify such semantic extensions making this feature available to domain-independent planners. However, this planner must support this extended PDDL version. They support two types of attachments: procedures that check conditions and procedures that compute the effect values, hence handling both preconditions and effects.

Technically, we also consider two mapping directions: from the more abstract to the more concrete and from the more concrete back to the more abstract. The information that needs to be provided is similar. Except that our mapping are used to bridge model levels and not to help the planner. They are used to activate the concrete skill code abstracted by the planner's model as an action, and are used to send data back from the skill code to the planner. Hence, the planner is not involved in this process, and can be any general purpose planner. Or, of course, it could be a planner that also uses semantic attachment to compute its plan.

As an example, the action of moving a block might require complex computations to decide whether a precondition of having a clear path holds. Semantic attachment (calling a path planner) can compute whether this condition holds. But then, to actually move the arm, a call to some code, e.g., *move-it* with appropriate parameters is required, which is what our mapping provides. Similarly, semantic attachment could run a computation for updating the battery level after this action, while if there is a battery-level topic in ROS maintaining this information, our mapping will provide the information needed to map this value to planning model values.

ESB is a part of the AIPlan4EU project and is related to its Unified Planning Library (UP)(https://www.aiplan4eu-project.eu). UP offers an abstraction layer/user interface on top of standard planning definition languages for specifying planning language. As such, it offers mapping services. These are focused on mapping user input to and code specific planner's inputs. In principle, one could do with code whatever one likes, and more specifically, map actions to code calls or to a different abstraction level. However, this is not the focus of UP, nor does it give declarative tools for such specification. The AM language as used by the AOS attempts to remove the need for coding such mapping by specifying them declaratively.

ESBs attempt to extend the UP to the application domain by connecting the gap with orchestration. The bridge automatically maps executable functions from the application definition to the action instances returned by the planner, sensor data into fluent values, and action choices to code application. This is done in the context of some application domain. In this respect, the bridge provides the added value the AOS system provides by auto-generating integration code based on our mapping specification. The key difference is that this paper posits the explicit specification of a mapping function between representations as a separate object, separating the definition of the mapping from its application. We conjecture that an ESB could be auto-generated given an AM spec.

The AM mappings are language-dependent in the sense that they map one description to the other, so they must parse specific syntax. In the AOS system, they parse our POMDP specification syntax. But one nice thing about them is that they are compositional – one can map A to C by mapping A to B and B to C. Moreover, because factored POMDPs subsume MDPs and classical planning, any language for describing the latter can be mapped to the former.

POMDPs

POMDPs offer a realistic model for autonomous robots because they model the stochastic nature of robots' actions, partial and noisy sensing, and one can provide rich task specifications using the reward function. Formally, a POMDP is a tuple $\langle \mathbf{S}, \mathbf{A}, \mathbf{T}, \mathbf{R}, \Omega, \mathbf{O}, \gamma, \mathbf{I} \rangle$: $\mathbf{S}$ is the state space, $\mathbf{A}$ the action space, $\mathbf{T}$ the state transition model, $\mathbf{R}$ the reward model, Ω the observation space, $\mathbf{O}$ the observation model, $\gamma \in (0, 1]$ is the discount factor, and $\mathbf{I} \in \mathbf{B}$ is the initial belief state. A *belief state* is a distribution over $\mathbf{S}$ that models the likelihood of each concrete world state based on available information.

Following an action $a \in \mathbf{A}$, the environment transitions from its current state $s \in \mathbf{S}$ to state $s' \in \mathbf{S}$, with probability $\mathbf{T}(s, a, s')$. Then, the agent receives an observation $o \in \Omega$, with probability $\mathbf{O}(s', a, o)$, and a reward $r = \mathbf{R}(s, a) \in \mathbb{R}$. Now, one can update one's belief state b to $b' = Pr(s|a, o, b)$ using the model parameters.

We focus on factored models where a state is an assignment to variables $X_1, \ldots, X_k$, and each observation Ω is an assignment to observation variables $W_1, \ldots, W_d$. Thus, $S = Dom(X_1) \times \cdots \times Dom(X_k)$ and $\Omega = Dom(W_1) \times \cdots \times Dom(W_d)$. In that case, τ, O, and R can be represented compactly by, e.g., a dynamic Bayesian network (Boutilier, Dean, and Hanks 1999).

A *policy* for a POMDP is a mapping $\pi : \mathbf{B} \mapsto \mathbf{A}$ from belief states to actions. The goal of POMDP solvers is to find a policy π^* that maximizes the expected accumulated discounted reward, i.e., $\pi^* = \underset{\pi}{max}[\mathbb{E}_\pi[\sum_{t=1}^{\infty} \gamma^t r_t.]$. r_t is the reward at step t, discounted by γ^t, so that when $\gamma < 1$, receiving a reward earlier is preferred.

The AM language

The basic requirement from a planning-based controller is to be able to dispatch actions and update the state with their results. Dispatching requires the ability to activate the code with appropriate parameter values. As noted earlier, these could be quite different from the action parameters used by the planner. Update requires using code elements, such as code variable values or values returned by the code, to update the planner's state representation. In the case of POMDPs, the latter is captured by the concept of an observation. In both directions, it is useful to be able to define local variables that help generate the final result. Therefore, the AM file consists of three parts: (1) Local variable definitions. (2) Observation computation. (3) Code activation, including how to compute code parameters.

Local Variable Definition. The first part of an AM file contains definitions of local variables and how to initialize their value. These variables can be used in other assignment statements in the AM file. Local variables can be initialized using (1) action parameters (2) code parameters (3) code return values (4) Python code that manipulates any of these elements.

Observation. The second part of the AM file specifies how to use the value of local variables to compute the observation following the action execution. In a factored POMDP, the observations are represented through the values of the observation variables. A POMDP is a very general model, and the fully observable case is the special case where the observation variables correspond to all state variables and there is no noise. We allow two specification methods: The first is appropriate when we want to return one of a small set of values. The user specifies a sequence of rules (expressed in Python) with associated values. The return value is that associated with the first rule evaluated to *True*. The second option is particularly useful for large observation spaces. We simply return the value of some local variable defined in the first part.

Code Activation. The third part of the AM file specifies how to activate the code. This includes instructions for finding the relevant code and for computing the code parameters. Code parameters may be unrelated to the model, e.g., a camera's sampling frequency, or they can be derived from the action parameters. For example, a navigation action is likely to have the source and destination as its parameters, where their values are discrete locations such as *here, kitchen, office, lab*. The navigation code may specify some code parameters, such as the local planner used or the rate by which the cost map is updated, and the actual (x, y) coordinates of the source and destination locations. The AM file provides (1) A path to the code or some equivalent system-specific information, such as the name of a ROS service. (2) An assignment to the various code parameters using local variable values.

Example

In this section, we describe an AM file that maps between a *move* action at the model level and the *navigate* skill code that implements it. In our system, one AM file is associated with every (lifted) action to allow for incremental addition of skill code and their corresponding planning-model action. However, in principle, one can aggregate all mappings in a single file. Through this example, we will also understand the syntax of AM files. We also present parts of the model-level documentation needed to understand the AM file (see Listings 1 and 2). In the AM file described below, blue words mark the start of a section. Sections may appear in any order except project name, which appears first. Brown words specify section properties. Section properties may also appear in any order. Teal words further elaborate section properties. Reserved variable names are in red.

Move, which is described in the model, has one parameter of type tLocation whose name is oDesiredLocation (see Listing 2). It specifies the location's (x, y, z) coordinates. Its *Navigate*'s skill code is based on the well-known open-source ROS *MoveBase* package. It has a single parameter called *goal*, which is an object with three fields, the (x, y, z) coordinates of the navigation target.

The definition of the tLocation type is in Lines 1-5 of the domain's Environment File (see Listing 1). Variable types can be any C++ primitive, primitive vector, or compound type. Compound variable types or enums can be defined with C++ primitive types as the building blocks. Next, for brevity, Lines 6-8 define a single tLocation constant: $l1$. The real file in our application contains additional tLocations, of course.

```
project: example
define_type: tLocation
 variable: float x 0.0
 variable: float y 0.0
 variable: float z 0.0
const: tLocation l1
code:
state.l1.x = −1.01606154442; state.l1.y =
    0.660750925541; state.l1.z
    =−0.00454711914062; state.l1.discrete = 1
```

Listing 1: Domain's Environment File

```
project: example
parameter: tLocation oDesiredLocation
```

Listing 2: Move's Documentation File

The AM file starts with a declaration and definition of the local variables using the local_variable keyword (see Listing 1). The lines following this declaration and ending in the next definition, define its properties. For example, we see that *goal_reached* is a variable that is defined through a ROS topic. It describes the topic's name, the type of the message it contains, where this message type is defined, and the variable's type. The AM file initializes its value in Line 7 to *False*. The code property describes how its value is updated. If its value was *True*, it remains so. Otherwise, if the */rosout* topic published a message containing the "Goal reached" text, the *else* part will return *True*. In Lines 13-18, local variables are initialized based on the model parameter's oDesiredLocation *x,y*, and *z* values. The sd_parameter property tells us that what follows is based on the skill model's parameters. In Lines 19-24, we define the *skillSuccess* variable using *navigate* skill code's return value. The reserved word '__input' refers either to a topic message's recent value (as used in Line 12) or the skill code's returned value (as used in Line 24).

```
1  project: example
2   local_variable: goal_reached
3  topic: /rosout
4  message_type: Log
5  imports: from: rosgraph_msgs.msg import: Log
6  type: bool
7   initial_value: False
8  code:
9  if goal_reached == True:
10     return True
11 else:
12     return __input.msg.find('Goal reached') > −1
13  local_variable: nav_to_x
14 sd_parameter: oDesiredLocation.x
15  local_variable: nav_to_y
16 sd_parameter: oDesiredLocation.y
17  local_variable: nav_to_z
18 sd_parameter: oDesiredLocation.z
19  local_variable: skillSuccess
20 imports: from: std_msgs.msg import: Bool
21 type: bool
22  from_ros_service_response: true
23 code:
24  skillSuccess = __input.success
```

Listing 3: Move's Abstraction Mapping File

In Lines 25-28, we describe the mapping from skill code to observations. *Move* can receive two observation values: *eSuccess, eFailed*. We return *eSuccess* if skillSuccess and goal_reached are true and otherwise, *eFailed*. Another option, not shown, yet essential for large observation spaces, is to return a specified local variable value as the POMDP's observation.

```
25 response: eSuccess
26  response_rule: skillSuccess and goal_reached
27 response: eFailed
28  response_rule: True
```

Listing 4: Move's Abstraction Mapping File

Lines 29-36 specify how to activate the */navigate* ROS service, provide its path and name, and specify its code parameters' value using the local variables defined earlier.

```
29  module_activation: ros_service
30 imports: from: geometry_msgs.msg import: Point
31 imports: from: simple_navigation_goals.srv
       import: navigateResponse, navigate
32 path: /navigate_to_point
33 srv: navigate
34 parameter: goal
35 code:
36 Point(x= nav_to_x, y= nav_to_y, z= nav_to_z)
```

Listing 5: Move's Abstraction Mapping File

Implementation

The *AM file* format was defined as part of the AOS system for using model-based planning for task-level control of autonomous robots and systems (Wertheim, Suissa, and Brafman 2024). AOS assumes that each robotic skill is modeled as a POMDP action and that the skill is implemented as a ROS service. In this sense, it follows in the footsteps of similar systems, starting with ROSPlan (Cashmore et al. 2015), that uses an action description language to describe the impact of some skill code and use planning to decide which skill to apply and when. These systems are able to dispatch the action and initiate the execution of the relevant skill code, and they also offer some mechanism for updating the state based on observations. AOS is the first system of this type to model skills using a POMDP model and is the first to include an explicit mapping format between the POMDP action and the skill code in the form of the AM file defined above. The use of AM files provides two important advantages. First, we have a clear, structured description. Second, we achieve plug'n play behavior: the user need not specify any code and information beyond the POMDP model and the AM file. Using this, the AOS can automatically integrate a POMDP planner with the skills' code, and control the robot online. For more details see (Wertheim, Suissa, and Brafman 2024), where we describe an implementation of a tic-tac-toe playing robotic arm, and a mobile robot with an arm delivering a cup, both of which required the programmer to specify the POMDP model and the AM, only, using which it generated all the integration code and controlled the robot.

Extensions

Our current file format supports the mapping of factored POMDP state and observation spaces to different representations of these spaces. But representation mappings can also be used to specify state abstractions that are the basis of diverse planning heuristics. Abstraction heuristics require, in addition, a mapping between action spaces. Specifying action mapping for POMDPs is not straightforward because this requires maping distributions. However, in the context of classical planning, this may be easier.

The best-known abstraction heuristics in classical planning are *pattern database* (PDB) heuristics (Culberson and Schaeffer 1998; Edelkamp 2001). The mapping used there is simply a projection, where some variables are completely ignored and other variables are copied without change. This is easily captured by using an identity mapping for the variables used and ignoring (or mapping to *true*) all other variables. Because PDBs maintain the same set of labels for actions, the action mapping is automatically induced by the state abstraction.

More interesting and more general are *merge-and-shrink* (M&S) heuristics (Helmert et al. 2014). (Sievers and Helmert 2021) develops a comprehensive theory of transformations of factored transition systems that provides the foundation for understanding M&S and its properties. Two fundamental concepts are *factored representations and mappings*, and *composition of transformation*. The mapping described by the AM is essentially a factored mapping over a factored representation. As long as labels remain unchanged, it also automatically extends to a mapping between action representations. To capture mappings between domains with different label/action spaces, the AM language would have to be extended, but we do not see any conceptual challenge in this. Indeed, one interesting application mentioned in (Sievers and Helmert 2021) is that of domain reformulation, which is, in a sense, our main application.

An important element of the M&S heuristic is the idea of transformation compositions, where a sequence of domain transformations is applied. In complex settings, corresponding to robotics, and possibly multi-robot systems, such transformations could be complex, and added-value tools could help automate their compositions. Indeed, while classical planning typically considers discrete, finite-domain variables, in robotics and other applications, we have many numeric variables. We believe that the fundamental ideas behind M&S remain as relevant in such domains, but now tables must be replaced by more complex functions, as in AM specs. In fact, one of the fundamental tools of ROS is a coordinate transformations package, which is often used to automatically compose such transformations. A typical composition would be between the transformation from a fixed coordinate system (e.g., a map) to the coordinate system of the robot base and then to the coordinate system of the robot's gripper. Or from a coordinate system of the robot's camera to its base and then to the gripper. Our work on formally specifying the transformation enables similar mapping between more complex state spaces.

Summary

When integrating planning with execution, a planning model is used to make decisions that must then be dispatched by executing real code. This code interacts with the real world and returns new observations that must be integrated into the planner's world model. Making this process work requires non-trivial programming, much of which can be replaced by auto-generated code, given an explicit mapping between the state variables of the planner and those of the code. The AM format described here provides such a specification format, used by the AOS system to provide exactly this added value. The precise AM syntax, which can surely be improved, is not the core issue. Rather it is the idea of using such a formal specification and the demonstration of its utility. We hope that this work will stimulate additional development of this knowledge representation and its applications.

References

Albore, A.; Doose, D.; Grand, C.; Guiochet, J.; Lesire, C.; and Manecy, A. 2023. Skill-based design of dependable robotic architectures. *Robotics and Autonomous Systems*, 160.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *J. Artif. Int. Res.*, 11(1): 1–94.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. Rosplan: Planning in the robot operating system. In *ICAPS*.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

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*, 114–121.

Doychev, I. D.; Viehmann, T.; Hofmann, T.; Lakemeyer, G.; and Trimpe, S. 2021. Goal Reasoning with the CLIPS Executive in ROS2.

Edelkamp, S. 2001. Planning with Pattern Databases. In *ECP*, 13–24.

Fikes, R. E.; and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2: 189–208.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 20: 61–124.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *AIJ*, 173(5-6): 619–668.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *J. ACM*, 61(3).

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, 9883–9891. AAAI Press.

Martín, F.; Clavero, J. G.; Matellán, V.; and Rodríguez, F. J. 2021. Plansys2: A planning system framework for ros2. In *IROS*.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*.

Rao, D.; Hu, G.; and Jiang, Z. 2020. PRobPlan: A Framework of Integrating Probabilistic Planning Into ROS. *IEEE Access*.

Rovida, F.; Crosby, M.; Holz, D.; Polydoros, A. S.; Großmann, B.; Petrick, R.; and Krüger, V. 2017. SkiROS—A Skill-Based Robot Control Platform on Top of ROS. In *Robot Operating System (ROS): The Complete Reference (Volume 2)*, 121–160.

Sadanandam, S. H. S. S.; Stock, S.; Sung, A.; Ingrand, F.; Lima, O.; Vinci, M.; and Hertzberg, J. 2023. A Closed-Loop Framework-Independent Bridge from AIPlan4EU's Unified Planning Platform to Embedded Systems. In *ICAPS'23 Planning in Robotics (PlanRob) Workshop*.

Sanner, S. 2010. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. ANU*.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *Journal of AI Research*, 781–883orw.

Wertheim, O.; Suissa, D. R.; and Brafman, R. I. 2024. Plug'n Play Task-Level Autonomy for Robotics Using POMDPs and Probabilistic Programs. *IEEE Robotics and Automation Letters*, 9(1).

Weyhrauch, R. W. 1980. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artif. Intell.*, 13(1-2): 133–170.

Younes, H. L.; and Littman, M. L. 2004. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*.