Journal of Artificial Intelligence Research 61 (2018) 215–289 Submitted 5/17; published 1/18

From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning

George Konidaris GDK@CS.BROWN.EDU
Brown University, Providence RI 02912
Duke University, Durham NC 27708

Leslie Pack Kaelbling LPK@CSAIL.MIT.EDU
Tomas Lozano-Perez TLP@CSAIL.MIT.EDU
MIT CSAIL, 32 Vassar Street Cambridge MA 02139

Abstract

We consider the problem of constructing abstract representations for planning in high-dimensional, continuous environments. We assume an agent equipped with a collection of high-level actions, and construct representations provably capable of evaluating plans composed of sequences of those actions.

We first consider the deterministic planning case, and show that the relevant computation involves set operations performed over sets of states. We define the specific collection of sets that is necessary and sufficient for planning, and use them to construct a grounded abstract symbolic representation that is provably suitable for deterministic planning. The resulting representation can be expressed in PDDL, a canonical high-level planning domain language; we construct such a representation for the Playroom domain and solve it in milliseconds using an off-the-shelf planner.

We then consider probabilistic planning, which we show requires generalizing from sets of states to distributions over states. We identify the specific distributions required for planning, and use them to construct a grounded abstract symbolic representation that correctly estimates the expected reward and probability of success of any plan. In addition, we show that learning the relevant probability distributions corresponds to specific instances of probabilistic density estimation and probabilistic classification. We construct an agent that *autonomously learns* the correct abstract representation of a computer game domain, and rapidly solves it.

Finally, we apply these techniques to create a physical robot system that autonomously learns its own symbolic representation of a mobile manipulation task directly from sensorimotor data—point clouds, map locations, and joint angles—and then plans using that representation. Together, these results establish a principled link between high-level actions and abstract representations, a concrete theoretical foundation for constructing abstract representations with provable properties, and a practical mechanism for autonomously learning abstract high-level representations.

1. Introduction

A core challenge in artificial intelligence is the problem of high-level reasoning in a low-level world: agents often face environments that offer only low-level sensing and actuation, but where planning and learning at that level cannot be feasible. Imagine a hungry student attempting to find lunch. It is immediately infeasible to do so by planning in terms of the thousands of individual footsteps required to reach a nearby restaurant, all while predicting in detail the gigabytes of image data that might be observed on the way; nevertheless, any such plan must ultimately be executed using those actions and those sensor inputs. This difficulty is particularly acute when designing intelligent robots which, like humans, can only sense the world via a constant stream of noisy, high-dimensional sensations, and act in it by sending control signals to a collection of motors. High-level abstract decision-making is clearly desirable, but low-level sensing and control are ultimately unavoidable; the question is how to develop high-level abstractions that facilitate low-level action.

Hierarchical reinforcement learning (Barto & Mahadevan, 2003) attempts to address this problem by abstracting away the low-level details of control so that an agent can learn and plan using higher-level skills, echoing and modeling similar approaches in robotics based on building and sequencing a collection of low-level motor controllers (Nilsson, 1984; Malcolm & Smithers, 1990; Brooks, 1991; Huber & Grupen, 1997; Gat, 1998). These approaches are founded on the hypothesis that behavior is principally *modular and compositional:* it is largely composed of sequences of a finite number of discrete behavioral components, assembled and re-assembled as necessary to complete a particular task. An agent should therefore solve problems—and generate behavior—by identifying goals and then constructing plans composed of high-level skills in order to reach them.

However, while the use of high-level skills can significantly improve performance (Sutton, Precup, & Singh, 1999; Silver & Ciosek, 2012), planning in the high-dimensional, continuous state spaces that characterize robotics problems (and many other applications) remains very difficult. We hypothesize that adding high-level actions can only be half of the solution; effective planning for devices as complex as robots requires *both* procedural abstraction (in the form of high-level skills) *and* state abstraction (in the form of symbolic representations). Moreover, we propose a specific relationship between the two types of abstraction: *high-level skills induce abstract representations.* Consider the example in Figure 1.

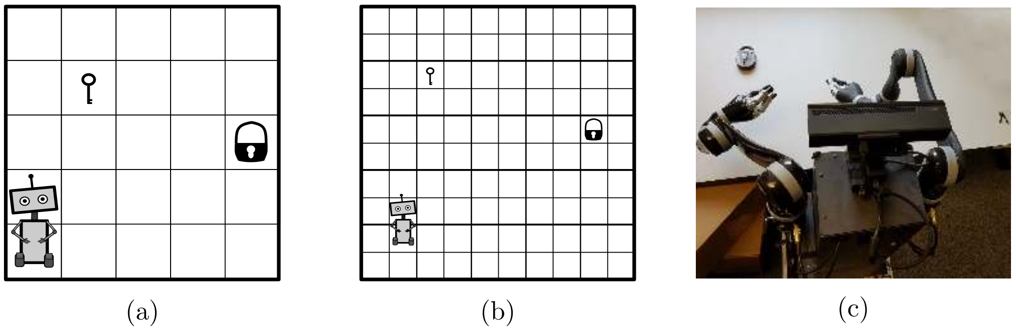


Figure 1: A robot must open a lock; similar tasks in (a) a small state space, (b) a larger state space, and (c) using a real robot.

Here we see three similar problems where a robot must obtain a key and use it to open a lock. In the first, an abstract agent operates within a small grid; in the second, the grid is much larger, substantially increasing the size of the state space; in the third, the agent is a real robot with a complex sensorimotor space. Without high-level actions, each problem is increasingly difficult—the robot must learn or plan in a larger state space in the second problem, and a high-dimensional continuous state and action space in the third. Now assume that each agent is given high-levels skills for approaching the lock and the key, picking up the key (when close enough to it), and using the key to open the lock (when adjacent to the lock and holding the key). Given these motor skills, *all three problems are equivalent*—in each case, a very simple state space is sufficient for planning: four boolean state variables describing whether the robot is near the key, whether it is adjacent to the lock, whether it holds the key, and whether the lock is open. Enlarging the state space or adding sensors and actuators makes creating the high-level skills harder, but once they are present, that space no longer matters—the high-level skills absorb the complexity of the robot's sensorimotor space and induce an abstract representation that reflects the underlying difficulty of the problem itself (which is trivial in this case).

We formalize this intuition by considering the problem of *automatically constructing abstract representations suitable for evaluating plans composed of sequences of high-level actions in a continuous, low-level environment.* Our broad approach is constructive: we define the set of questions that the abstract representation will be used to answer, and then construct a representation provably capable of answering them. We first consider the deterministic planning case, where the appropriate question set is whether any plan composed of high-level actions can be executed with certainty, or not. We show that the relevant abstract quantities are sets of states, and the relevant abstract operators are set operations, quantities which correspond to the classical definition of a propositional symbol. We define the specific collection of such symbols that is necessary and sufficient for planning, and define two types of generic skills for which those representations can be concisely described. The first type of skill, which reaches some *subgoal* and then terminates, leads to an abstract graph representation. The second, where the skill sets some low-level state variables to a subgoal and leaves others unchanged, leads to a representation similar to a factored MDP and which we show can be converted into PDDL (McDermott, Ghallab, Howe, Knoblock, Ram, Veloso, Weld, & Wilkins, 1998), a canonical STRIPS-style planning representation. In both cases the resulting representation always correctly evaluates a plan. We apply the resulting framework to the continuous playroom domain (Singh, Barto, & Chentanez, 2005; Konidaris & Barto, 2009a), a problem with a 33-dimensional state space, to build a high-level abstract representation of the domain that enables an off-the-shelf planner to solve it in a few milliseconds.

We then consider the probabilistic planning case, where an abstract representation must support determining the probability that a plan succeeds, along with its expected reward. We show that the appropriate abstract representation is analogous to the deterministic planning case, but requires a generalization from sets of states to distributions over states. We identify the specific distributions required for planning, and show how they can be used to construct a grounded abstract symbolic representation that correctly estimates the expected reward and probability of success of any plan. In addition, we show that learning the relevant probabilistic distributions corresponds to specific instances of the well-studied probabilistic density estimation and probabilistic classification tasks. We use these results to construct an agent that *autonomously learns* the correct abstract representation of a computer game domain, and uses an off-the-shelf probabilistic planner to rapidly solve it.

Finally, we apply these techniques to create a physical robot system (pictured in Figure 2) that autonomously learns its own symbolic representation of a mobile manipulation task directly from sensorimotor data—point clouds, map locations, and joint angles—and then

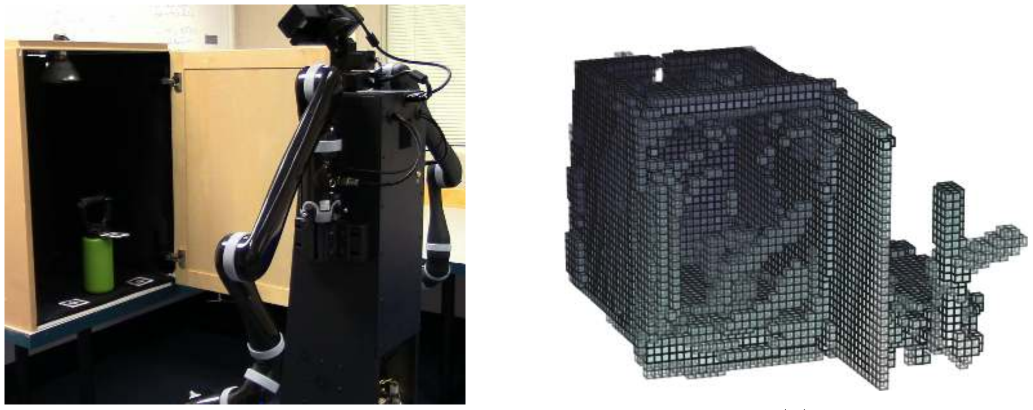


Figure 2: The system described in Section 5. (a) Anathema Device, a mobile manipulation robot, positioned in front of a cupboard and preparing to pick up the bottle inside it. (b) A visualization of a grounded symbol—learned autonomously by the robot—that indicates that the door is open. This particular symbol is necessary for determining whether or not Anathema can successfully pick up the bottle.

uses that representation to perform high-level planning. This result shows that our framework is capable of learning grounded, symbolic representations that bypass the underlying complexity of the robot and focus on the inherent complexity (or lack thereof) of the problem the robot is trying to solve. Together, these results establish a principled link between high-level actions and abstract representations, a theoretical foundation for constructing abstractions with provable properties, and a practical mechanism for autonomously learning abstract high-level representations.[1]

2. Background and Setting

We adopt the Markov-decision process (MDP) formalism of agent decision-making, which models the agent's low-level state and action spaces with a tuple:

$$(S, A, R, T, \gamma),$$

where S is a high-dimensional state space, each element $s \in S$ of which is a real-valued vector; A is an action space, each element $a \in A$ of which is also a real-valued vector; R is the reward function, with $R(s, a, s')$ expressing the real-valued scalar reward obtained by executing action a in state s and subsequently arriving in state s'; T is the transition function, where $T(s'|s, a)$ is the distribution over states in which the agent may find itself after having executed action a in state s; and $\gamma \in (0, 1]$ is a discount factor expressing a preference for immediate over future reward.

1. This article unifies and substantially extends material originally published in two conference papers (Konidaris, Kaelbling, & Lozano-Perez, 2014, 2015).

In an MDP the agent's goal is typically to select actions that maximize *return*, or the discounted sum of future rewards:

$$\mathcal{R} = \sum_{i=0}^{\infty} \gamma^i R(s_i, a_i, s_{i+1}).$$

Typically, the agent is concerned with maximizing $\mathcal{R}$ in a single MDP. However, we are interested in the *multi-task* reinforcement learning setting (Wilson, Fern, Ray, & Tadepalli, 2007; Brunskill & Li, 2013) where the agent must optimize expected reward over a distribution over tasks. Specifically, we are interested in changing goals over a fixed environment; we model the environment as a *base MDP* that specifies the environmental dynamics and a background reward function (typically representing action costs). The agent must solve multiple tasks in this environment, each of which is obtained by adding a set of *task goal states*, which are high-reward and in which execution terminates. This setting makes it advantageous for the agent to learn a model of its environment (the base MDP) which it can use in future tasks to minimize sample complexity by planning rather than learning.

2.1 Hierarchical Reinforcement Learning

Solving MDPs with high-dimensional state and action spaces can be prohibitively difficult, especially in domains (like robotics) where very long sequences of low-level actions may be required to achieve a goal. Hierarchical reinforcement learning (Barto & Mahadevan, 2003) aims to overcome this difficulty through the use of temporally extended high-level actions.

The *options framework* (Sutton et al., 1999) is a hierarchical reinforcement learning framework that models high-level actions as *options*. An option o consists of three components: an *option policy*, π_o, which is executed when the option is invoked, and maps low-level states to low-level actions; an *initiation set*, $I_o = \{s | o \in O(s)\}$, which describes the low-level states in which the option may be executed; and a *termination condition*, $\beta_o(s) \rightarrow [0, 1]$, which describes the probability that an option will terminate upon reaching low-level state s.

An option is thus a natural minimal model of a robot motor controller that can be executed in certain situations, whereupon it runs a control program (of whatever form) that issues motor commands until some termination condition is reached. We will therefore assume an agent that has a collection of options available to it, along with knowledge of each option's policy, initiation set, and termination condition. However, we will assume that the agent can only evaluate initiation set membership for the state it is currently in. This implicit definition of the initiation set models the case where a robot motor controller can determine whether or not it can be executed in a specific situation, but does not have complete knowledge in advance of all situations in which execution could occur.

Replacing an agent's low-level actions with higher-level options in this way results in a semi-Markov decision process (SMDP), described by the tuple:

$$M = (S, O, R, P, \gamma),$$

where S is the original low-level state space; $O(s)$ is the finite set of options available in state $s \in S$; $R(s', \tau | s, o)$ is the reward received when executing action $o \in O(s)$ at state $s \in S$, and arriving in state $s' \in S$ after τ time steps; $P(s', \tau | s, o)$ describes the probability of arriving in state $s' \in S$, τ time steps after executing action $o \in O(s)$ in state $s \in S$ (we often write $P(s'|s, o)$ when unconcerned with time); and $\gamma \in (0, 1]$ is the discount factor.

The combination of reward model, $R(s', \tau|s, o)$, and transition model, $P(s', \tau|s, o)$, for an option o is known as an *option model*. We will *not* assume that the agent has access to option models; instead, it must gather data about each option's performance and execution characteristics by actually executing it in the world.

One way to do so would be to use option execution experience to learn the relevant option models themselves. An agent possessing such models is capable of sample-based SMDP planning (Sutton et al., 1999; Kocsis & Szepesvári, 2006). However, although options can speed up planning by shortening the effective search depth (Sorg & Singh, 2010; Powley, Whitehouse, & Cowling, 2012), SMDP-style planning in the kinds of large, continuous state spaces that characterize robot problems is still very difficult. Value function methods are plagued by the curse of dimensionality, and sample-based methods cannot guarantee that a plan is feasible in the presence of low-level stochasticity, especially when abstracting over start states. Our goal is to show how the addition of options to an MDP allows us to drastically reduce its state space in addition to its action space, resulting in an *abstract representation* that affords high-level planning.

2.2 High-Level Planning

In contrast to hierarchical reinforcement learning, which is focused on procedural abstraction, high-level planning methods take a complementary approach focused on abstract, symbolic *state* representations. These methods have been the subject of intense study for several decades (Ghallab, Nau, & Traverso, 2004), both as an efficient means for solving general planning problems and specifically for abstract planning for robotics (Fikes & Nilsson, 1971).

The simplest high-level planning formalism is the *set-theoretic representation* (Ghallab et al., 2004). Here, a planning domain is described by a set of propositions $\mathcal{P} = \{p_1, ..., p_n\}$ and a set of actions $\mathcal{A} = \{\alpha_1, ..., \alpha_m\}$. A proposition is an abstract boolean state variable; a complete high-level state is obtained by assigning a truth value to every $p_i \in \mathcal{P}$. Each action α_i is described by a tuple:

$$\alpha_i = (\text{precond}_i, \text{effect}_i^+, \text{effect}_i^-),$$

where $\text{precond}_i \subseteq \mathcal{P}$ lists all propositions that must be true in a state for the action to be applicable at that state, and positive and negative effects $\text{effect}_i^+ \subseteq \mathcal{P}$ and $\text{effect}_i^- \subseteq \mathcal{P}$ list the propositions set to true or false, respectively, as a result of applying the action. All other propositions retain their values. A planning problem is obtained by augmenting the domain description with a start state s_0 and a set of goal states S_g. This separation between domain and task is equivalent to the distinction between an environment and a task in our multi-task reinforcement learning setting.

For example, a planning problem involving opening a door that could be locked might be described using the propositions `DoorUnlocked` and `DoorClosed`, each of which must be assigned a boolean truth value in every high-level state. The problem description might also include the action `OpenDoor`, which describes the preconditions for executing it, and the positive and negative effects of doing so:

```
action OpenDoor
  precondition: DoorUnlocked and DoorClosed
  effect+:
  effect-: DoorClosed.
```

In addition to specifying other operators, completing the problem description also requires specifying a single start state (perhaps that `DoorUnlocked = false` and `DoorClosed = true`), and a goal set (e.g., that `DoorClosed = false`, and the value of `DoorUnlocked` is immaterial).

A more common formulation is the *classical representation* (Ghallab et al., 2004), also known as STRIPS planning (Fikes & Nilsson, 1971), which uses a relational representation to more compactly describe the planning domain. The set of propositions is replaced by a set of atoms consisting of predicates and objects. A predicate is a parametrized proposition, which can be instantiated (resulting in a *ground predicate*) by binding objects to its parameters, and a high-level state is a list of ground predicates assigned to be true (the remainder are assumed to be false). Each action description, precondition, and positive and negative effect list is also parametrized to obtain an *operator*.

For example, if the planning problem described earlier involved multiple doors we could introduce objects `DoorA`, `DoorB`, and `DoorC` to refer to them, and generalize our previous propositions to predicates `DoorUnlocked(X)` and `DoorClosed(X)`, which can be instantiated by binding the variable `X` to an object. For example, we can instantiate `DoorUnlocked` using `DoorA` to obtain ground predicate `DoorUnlocked(DoorA)`, which must be true or false in each high-level state. The resulting representation allows us to write a relational (or *lifted*) operator that can be instantiated to refer to any door:

```
operator OpenDoor(X)
  precondition: DoorUnlocked(X) and DoorClosed(X)
  effect+:
  effect-: DoorClosed(X).
```

The agent may now bind an object to the parameter `X` to evaluate whether it is possible to execute, say, `OpenDoor(DoorB)`, in a particular high-level state, and to derive the resulting state change.

The classical and set-theoretic representations are equivalently expressive, though the classical representation may have an exponentially smaller domain description than the set-theoretic one (Ghallab et al., 2004), and both are essentially factored MDPs (Koenig, 1991; Boutilier, Dearden, & Goldszmidt, 1995). Problems in both formulations are typically described using the *Planning and Domain Definition Language* or PDDL (McDermott et al., 1998), which serves as the input format for most general-purpose planners. For simplicity, we use the set-theoretic representation and leave parametrizing it to future work.

2.3 The Semantics of Symbolic Representations

The high-level planning representations described above are considered abstract or symbolic because, when applied to real-world problems, the truth values of the propositions or predicates are not directly sensed by the agent. For example, a robot solving the door opening problem will likely sense the world as an image—a collection of pixel values. The distinction between the door being open or closed, the specific boundary where that distinction occurs, and the notion that that specific distinction must be made in the first place, are not properties of the world as sensed by the robot. Instead, they are an invented abstraction imposed on the robot's own low-level sensorimotor space.

A robot that employs high-level abstract planning must therefore equip itself with an appropriate symbolic representation, which it must link to values that it *can* sense. This link, which establishes the semantic meaning of the symbol, is known as a *grounding*. The classical definition of a propositional symbol is that it is a name for a set of low-level states:

Definition 1. *A propositional symbol σ_Z is the name for a test, τ_Z, and the corresponding set of states, $Z = \{s \in S \mid \tau_Z(s) = \textit{True}\}$.*

A symbol σ_Z is therefore a name for both a set of states, $Z \subseteq S$, and a test (or classifier), τ_Z, that the robot can run at any particular low-level state s to determine whether $s \in Z$. Either τ_Z or Z specify the symbol's semantic meaning; we refer to Z as its *grounding set*, and τ_Z as its *grounding classifier*.

For example, consider the symbolic proposition `AtTable13`, which indicates that a robot is near a specific table. The symbol `AtTable13` is just a name, but its semantic meaning can be represented as either all of the possible world states in which the robot could be considered to be at that particular table (its grounding set), or alternatively by a classifier that the robot can run at any state, to determine whether it is close enough to that particular table in that particular state to be considered "at" it (its grounding classifier).

In addition to a collection of symbols, a symbolic representation also includes a collection of *operators* (or rules) that support reasoning over those symbols. Each operator defined over symbols has a corresponding operation defined over their groundings. For example, the operators classically associated with propositional symbols are:

1. The `or` of two symbols, corresponding to the union of their grounding sets.
2. The `and` of two symbols, corresponding to the intersection of their grounding sets.
3. The `not` of a symbol, corresponding to the complement of its grounding set.
4. Whether or not a symbol is `null`, corresponding to whether or not a grounding set is empty.[2]

A reasoning process can be considered symbolic if it can evaluate symbolic expressions using solely symbols and symbolic operators, without resorting to the relevant groundings. Such a representation is only likely to be useful if the grounding of a composite expression is equal to the result of performing the corresponding set operations on the grounding of its components. This process is depicted in Figure 3.

A symbolic representation—or *model*—of a problem therefore consists of a *vocabulary* of symbols, plus a collection of operators that specify rules for manipulating those symbols to describe the effects of actions in the world. While such systems—often, but not necessarily,

2. This allows us to determine whether one classifier is a subset of another: $A \subseteq B$ if and only if $A \cap \neg B$ is empty; we can thereby test implication.

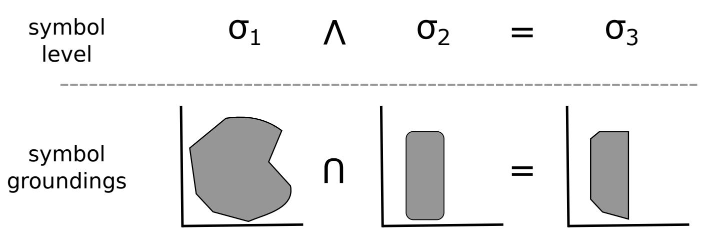


Figure 3: A logical operation on two symbols, with its corresponding grounding. For a reasoning process to be considered symbolic, it must be able to determine that, in this case, the result of a logical `and` of σ_1 and σ_2 is σ_3, without reference to their groundings. Such a process is only likely to be useful if σ_3's grounding is equal to the set intersection of the groundings of σ_1 and σ_2.

using the classical semantics of propositional symbols—have been used in robotics and AI before, it has almost always been the case that at least the symbolic vocabulary has been given to the robot by a designer. In other words, the question of *which abstract symbols* a robot should use has been left unanswered.

In the next section, we show that the right abstract representation for deterministic planning is indeed one that uses abstract propositional symbols with classical semantics, that we can precisely identify which symbols the agent should have in its vocabulary, and finally that, given that specific vocabulary and its grounding classifiers, we can construct a model that supports completely symbolic planning.

3. Constructing Symbolic Representations for Deterministic Planning

A critical question when constructing a representation is: *what is that representation for*? In other words, what reasoning process should it support? A precise answer to this question identifies the set of questions that the abstract representation will be required to evaluate. Our approach is to use the properties of the solutions to those questions to construct a representation provably capable of answering them.

Our goal is to support high-level planning, and we begin with its simplest instantiation: deterministic planning, which aims to find a sequence of actions that with probability $p = 1$ moves the agent from some start state to some goal. This may seem to be an unrealistic goal for agents acting in a stochastic SMDP. However, in many cases the agent's skills are based on feedback policies that absorb the low-level stochasticity present in the domain, resulting in the reliable achievement of subgoals. Deterministic planning is also the classical aim of high-level planning (Fikes & Nilsson, 1971), though like that field we will move on to the probabilistic setting in Section 4.

The central question that a deterministic planner must answer in order to operate is whether a plan is *satisficing*—whether it succeeds in reaching a goal from a start state with certainty. We define a plan as follows:

Definition 2. *A* plan $p = \{o_1, ..., o_{p_n}\}$ *from a state set* $Z \subseteq S$ *is a sequence of options to be executed from some state in* Z.

A feasible plan is one in which no sequence of states reachable by executing part of the plan would leave the agent unable to execute the remainder:

Definition 3. *A plan is* feasible *when the probability of the agent being able to execute it is 1, i.e.,* $\nexists \bar{S} = \{s_1, ..., s_j\}$ *for any* $j < p_n$ *such that* $s_1 \in Z$, $s_i \in I_{o_i}$, $P(s_{i+1}|s_i, o_i) > 0, \forall i < j$, *and* $s_j \notin I_{o_j}$.

Both definitions use a *set* of start states, Z, rather than a single start state, s_0. We do this for consistency, as will become become clear shortly, although of course it is possible to plan from a single start state by setting $Z = \{s_0\}$.

Our definition so far lacks the notion of a goal; for simplicity and uniformity we treat the agent's goal g as a terminating option that can only be executed when the agent has actually reached g. Given a set of goal states g, option o_g has initiation set $I_g = g$, termination condition $\beta_g(s) = 1 \forall s$, and a null policy. We therefore obtain a definition of a satisficing plan that simply requires a feasibility test as before:

Definition 4. *A plan tuple is* satisficing *for goal* g *if it is feasible and its final action is goal option* o_g.

We next define the *plan space*: the set of all plans that the agent could form.

Definition 5. *The* plan space *for an SMDP is the set of all tuples (*Z*,* p*), where* $Z \subseteq S$ *is a set of states in the SMDP and* p *is a plan.*

The purpose of an abstract representation in this setting is to determine whether any plan in the plan space is satisficing for goal g, which reduces to determining whether it is feasible. How might we determine whether or not a plan tuple (Z, p) is satisficing?

Consider a very simple two-step plan, $p = \{o_1, o_2\}$. The agent can definitely execute option o_1 from start set Z if, and only if, every state in Z is also in o_1's initiation set, I_{o_1}. Therefore, the agent can execute o_1 with certainty if and only if $Z \subseteq I_{o_1}$. Similarly, let Z_1 be the set of states that the agent could find itself in after executing o_1 from anywhere in Z; the agent can definitely execute o_2 if and only if $Z_1 \subseteq I_{o_2}$. This is depicted in Figure 4.

This simple example can be generalized to generate an expression that determines the feasibility of any plan in the plan space, of any length; the resulting family of expressions has the following important properties:

1. Each expression consists *only* of sets and set operators. An abstract representation whose task it is to evaluate such expressions must therefore be grounded in sets and set operators, which matches the classical semantics of propositional symbols (Definition 1, as discussed in Section 2.3).

2. The expressions are an *if-and-only-if* proof of feasibility; any other such proof (using any other such expression) must be either incorrect or isomorphic.

3. The only sets that occur in any such expression are the initiation set of an option, the start set Z, and the computed sets Z_i.

Building on these properties, we can now define a set of symbols and an operator that are necessary and sufficient for evaluating whether any plan is feasible.

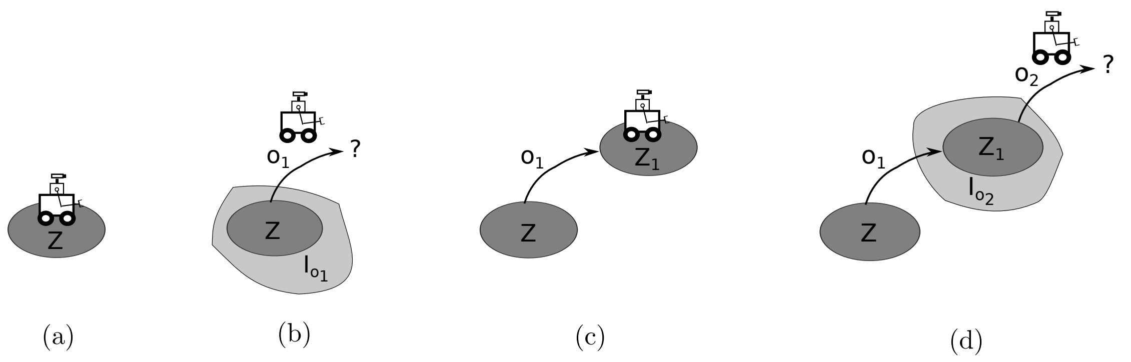


Figure 4: Verifying that an agent can definitely execute the plan $p = \{o_1, o_2\}$. (a) The agent begins in one of a set of states states, Z. (b) It must first determine whether or not it can execute o_1; it can do so if and only if $Z \subseteq I_{o_1}$. Having done so, it must determine whether or not it can execute o_2. (c) Let Z_1 be the set of states in which it might find itself after having executed o_1 from any state in Z. (d) The robot can execute o_2 if and only if $Z_1 \subseteq I_{o_2}$. The entire plan can therefore be executed if and only if $(Z \subseteq I_{o_1}) \wedge (Z_1 \subseteq I_{o_2})$.

3.1 Symbols for Deterministic Planning

First, we define a set of symbols that represent each option's initiation set:

Definition 6. *The* precondition *of option o is the symbol naming its initiation set: $Pre(o) = \sigma_{I_o}$.*

We also define an operator expressing the consequences of executing an option from one of a set of states, corresponding to each Z_i in the argument given above:

Definition 7. *Given an option o and a set of states $X \subseteq S$, the image of o from X is: $Im(X, o) = \{s' | \exists s \in X, P(s'|s, o) > 0\}$.*

The image operator $\text{Im}(X, o)$ computes the set of states that might result from executing o from a state in X. These two quantities (the precondition symbol and image operator) are defined using, and correspond closely to, the definition of an option model.

A representation that can express both each option's precondition and the image operator symbolically is sufficient to evaluate the feasibility of any plan:

Theorem 1. *Given an SMDP, the ability to represent the precondition of each option and to compute the image operator is sufficient for determining whether any plan tuple (Z, p) is feasible.*

Proof. Consider an arbitrary plan tuple (Z, p) with plan length n. We set $Z_0 = Z$ and repeatedly compute $Z_{j+1} = \text{Im}(Z_j, p_j)$, for $j \in \{1, ..., n\}$. The plan tuple is feasible if and only if $Z_i \subseteq I_{o_{i+1}}, \forall i \in \{0, ..., n-1\}$. □

To test the feasibility of a plan, the agent begins with its set of start states Z and sequentially computes the set of states that can be reached by executing each option in the plan, checking in turn that the resulting set is a subset of the initiation set of the following option. The representation must therefore contain symbols naming the initiation set of each option, and rules that enable the agent to compute symbols naming the set of states that it may find itself in after executing an option from any of a set of initial states.

Since the feasibility test in the above proof is biconditional, any other feasibility test must express exactly these conditions for each pair of successive options in a plan. Representing the image operator and precondition symbols are therefore also necessary for abstract planning. We call the symbols required to name an option's initiation set and express its image operator its *characterizing symbols*.

The ability to perform symbolic planning thus hinges on the ability to symbolically compute the image operator. This can be arbitrarily hard. Consider an option that maps each state in its initiation set to a single (but arbitrary and unique) successor state. In this case we can do no better than than expressing $\text{Im}(Z, o)$ as an infinite number of singleton mappings with no structure or regularity, with which we cannot hope to form a useful abstract representation.

We must therefore rely on some form of simplicity or regularity in the image operator, which in turn must come from a similar property of the options themselves. Fortunately, we can identify such a property, and thereby concisely represent the image operator, for at least two classes of options in common use—subgoal options, and abstract subgoal options.

3.1.1 Subgoal Options

Options are often—especially in research on skill discovery methods—constructed to reach a *subgoal* (Precup, 2000). A subgoal is typically a compact set of states; the option policy is constructed to drive the agent to that set, and the option's initiation set is the set of states from which option execution will successfully reach it. In such cases, the set of states in which the agent has a non-zero probability of arriving after executing the option may not practically depend on where it starts.

To capture the subgoal, we define an option's *effect set*, and the corresponding symbol naming it, its *effect symbol*, which we will for convenience often shorten to simply *effect*:

Definition 8. *The* effect set *of option o is the set of all states that an agent can possibly find itself in after executing o:* $\mathit{Effect}(o) = \{s' | \exists s \in I_o \ s.t.\ P(s'|s, o) > 0\}$.

Definition 9. *The* effect symbol *of option o names its effect set:* $\mathit{Eff}(o) = \sigma_{\mathit{Effect}(o)}$.

The effect set is the set of all possible states the agent can find itself in after executing the option from anywhere; it expands the start set to be all states in the initiation set, and so it can be viewed as the union of all possible images: $\text{Effect}(o) = \bigcup_X \text{Im}(X, o)$. It is also necessarily a subset of the option's termination condition: $\text{Effect}(o) \subseteq \beta_o$; it might not be identical to β_o because some terminal states may not be reachable from any start state.

Given the centrality of subgoals in many hierarchical reinforcement learning systems, it is natural to ask whether or not we could simply *substitute* the effect set in place of the image—discarding the complex image operator in favor of a single effect symbol per option. There are two conditions under which this results in a useful representation. In the first, substituting the effect set for the image never changes the result of a subset test with the initiation set of any option:

Definition 10. *The* weak subgoal condition *holds for option o when $(Effect(o) \subseteq I_{o_i}) = (Im(X, o) \subseteq I_{o_i})$, for all start sets $X \subseteq I_o$ and options o_i.*

The weak subgoal condition states that substituting the effect set for the image—effectively enlarging the results of the image operator to match the effect set—never changes the outcome of a feasibility test for any plan tuple the agent may wish to evaluate. A second, stronger condition, will be useful later:

Definition 11. *The* strong subgoal condition *holds when the effect set and the result of the image operator are always equal: $Effect(o) = Im(X, o)$, for all start sets $X \subseteq I_o$.*

The strong subgoal condition goes even further, requiring that the effect set *is exactly equal to* the outcome of the image operator. This very strong condition implies the weak subgoal condition; any representational property proven for the weak subgoal condition also holds for the strong (but not necessarily vice versa).

Although the weak subgoal condition may seem restrictive, many robot controllers either fulfill it in practice, or nearly so; for example, many robot motor skills use a feedback controller to reach a specific, and relatively small, set of goal states. However, an equally likely situation is one in which an option does *not* fulfill a subgoal condition, but in which it can be *partitioned* into suboptions which do. For example, consider an option that we might describe as *walk through the door you are facing.* In this case, the image of the option—and consequently the options that can follow on from it—strongly depends on the state from which the option is executed, because there may be many such doors, each leading to very different rooms. However, if there are a small number of doors, then the option could be split into separate suboptions for each door by splitting the initiation set and treating each resulting suboption as distinct. The resulting partitioned options may then naturally fulfill a subgoal condition.

Using subgoal options severely restricts the potential goals an agent may plan for: all feasible goals must necessarily be a superset of one of the effect sets. However, it has some major advantages. First, knowing which option will be executed at time t is a sufficient statistic for determining which options can subsequently be executed at time $t+1$; the effect set the agent is in at time t can therefore serve as a *Markov state descriptor* in the abstract model we will shortly construct. Second, the characterizing symbols for each subgoal option o are just $\text{Pre}(o)$ (naming I_o) and $\text{Eff}(o)$ (naming $\text{Effect}(o)$). Therefore, the *only* type of expression the symbolic representation need ever evaluate is of the form $\text{Effect}(o_i) \subseteq I_{o_j}$, for any pair of options o_i and o_j. This leads to a particularly simple abstract representation:

Definition 12. *A* plan graph *for an SMDP with n subgoal options is a directed graph $G = (E, V)$ with n vertices $v_1, ..., v_n$, and edge (v_i, v_j) if $Effect(o_i) \subseteq I_{o_j}$.*

A plan graph (augmented with an appropriate reward function) is an abstract deterministic MDP, where each state is a vertex, v_i, that grounds to the agent being in $\text{Effect}(o_i)$. Each edge (v_i, v_j) is an action, representing the ability to execute option o_j immediately after having executed option o_i, which is feasible if and only if $\text{Effect}(o_i) \subseteq I_{o_j}$. An entire plan tuple (p, Z), where $p = \{o_1, ..., o_{p_n}\}$ is therefore feasible if and only if $Z \subseteq I_{o_1}$, and a path exists in G from o_1 to o_{p_n}. An example plan graph is depicted in Figure 5.

Computing G requires time in $O(n^2)$ for n subgoal options, but once it has been computed the agent may dispose of its grounding classifiers and perform symbolic planning

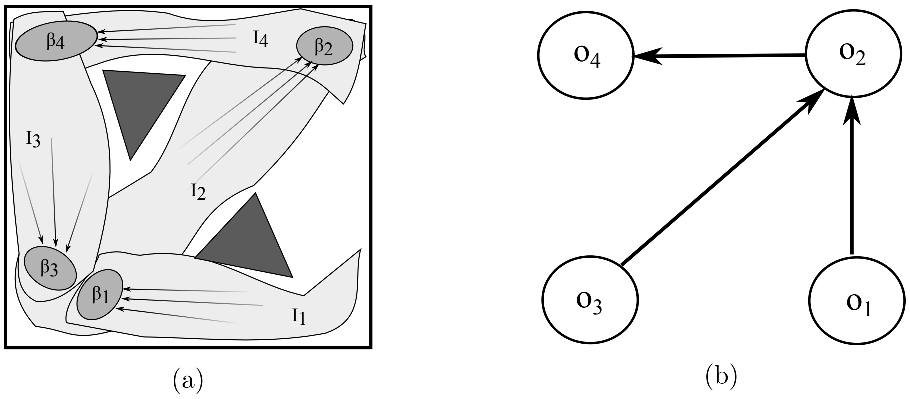


Figure 5: (a) A domain with four subgoal options, each with initiation set I_i and termination condition β_i. (b) The corresponding plan graph. Note that there is no edge from o_4 to o_3 because $\beta_4 \not\subseteq I_3$ (even though they overlap).

exclusively over G, without reference to the low-level state space. The subgoal property therefore leads directly to both an appropriate symbolic representation (a graph) and a class of planning mechanisms (graph search) for reasoning using that representation. We now turn to a more general class of options that leads to a more general class of representations, requiring a more powerful class of planning mechanisms.

3.1.2 Abstract Subgoal Options

The key assumption of subgoal options is that they drive *all aspects of the low-level state* to the target subgoal. That is a reasonable model for simpler robots performing tasks like navigation, where variables describing the robot itself constitute the entirety of the state space. However, consider a mobile manipulation robot in a large room full of many objects. Since the robot is able to manipulate the objects, each object's state variables must be present in the overall state space. Here it is unreasonable to require the robot's motor skills to change every object's state at once; the robot may only have one or two grippers, and the objects may be far apart. Instead, it is much more likely that a motor skill will only change *some* state variables, and leave the remainder unchanged.

We model such motor skills as *abstract subgoal options*. An abstract subgoal option o has a *mask* that identifies which low-level state variables the option drives towards a subgoal; the remaining state variables are left unchanged. In other words, mask(o) returns a list of the indices of all of the state variables that executing o modifies, and the new values of those variables are for practical purposes independent of the low-level state from which the option was executed. The image of an abstract subgoal option therefore has a specific structure—it can be broken into the part of the state space modified by option execution, and the part that remains untouched.

To formalize this, we first define an operator that relaxes the conditions under which membership of a set is based, by removing conditions that depend on specific variables:

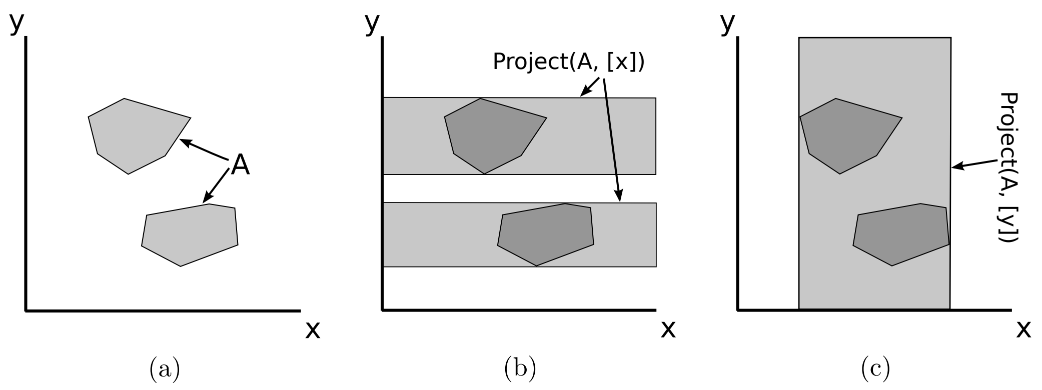


Figure 6: An example of applying the projection operator. (a) The input to the operator in this instance is a set, A, defined over two variables, x and y. A is not a compact set and consists of two component compact sets. (b) Projecting x out of A eliminates restrictions based on its value, effectively widening the set to span the entire x dimension. Note that the result is also not compact. (c) If we instead project y out of A, the result is a compact set spanning the y dimension.

Definition 13. *Given a set of states $X \subseteq S$ and a list of state variables v, the projection of v out of X is: Project$(X, v) = \{s | \exists x \in X, s[i] = x[i], \forall i \notin v\}$.*

The projection operator removes restrictions based on the state variables in v. For membership in the resulting set, a state need only agree with some state in X *for variables not in v*. Projection therefore makes X completely permissive about the state variables in v: *any* setting of those variables is acceptable, provided that it is coupled with values corresponding to the remaining variables for some state in X. This is depicted in Figure 6.

We now use projection to define an operator that removes restrictions on the state variables in option o's mask:

Definition 14. *Given set X and option o, the* remainder *of X after the execution of o is Remainder(X, o) = Project$(X, mask(o))$.*

The assumption that the value of the state variables in the mask after execution are practically independent of the low-level state from which the option was executed allows us to write the image of an abstract subgoal option as:

$$\text{Image}(X, o) = \text{Effect}(o) \cap \text{Remainder}(X, o).$$

The two conditions under which this substitution is appropriate for the subgoal case can be straightforwardly generalized to deal with abstract subgoals:[3]

Definition 15. *The* weak abstract subgoal condition *holds for option o when (Effect$(o) \cap$ Remainder$(X, o) \subseteq I_{o_i}$) = (Image$(X, o) \subseteq I_{o_i}$), for all start sets $X \subseteq I_o$ and options o_i.*

Definition 16. *The* strong abstract subgoal condition *holds if (Effect$(o) \cap$Remainder(X, o)) = Image(X, o), for all start sets $X \subseteq I_o$.*

3. As before, we can generalize abstract subgoal options to partitioned abstract subgoal options.

An abstract representation that supports planning using options that obey at least the weak abstract subgoal condition must therefore contain at least a precondition and effect symbol for each option, and additionally the symbols required to express the output of all instances of the remainder operator that could occur when computing the image.

3.2 Constructing a STRIPS-Like PDDL Domain Description

As we have seen above, an agent with subgoal options can construct an abstract graph representation that is sufficient for planning. The use of *abstract* subgoal options, however, requires a more structured representation capable of modeling option executions that only change some state variables. These properties suggest a representation where the abstract state is described by a vector of indivisible factors, in a similar fashion to a factored MDP (Boutilier et al., 1995), and an abstract model describes how these factors change with option execution. We now show how such a representation can be constructed using only each option's initiation set, effect set, and mask; no further information is required.

Our strategy is as follows. First, we assume that we are given abstract subgoal options, possibly as a result of a partitioning process. We then partition the low-level state variables into *factors*—sets of low-level state variables that, if changed by an option execution, are always changed simultaneously. The factors identify the smallest lists of state variables that could be projected out of a set of states when computing an image. We then enumerate the sets that describe all reachable projections of each factor out of every effect set. The resulting sets—which can all be derived from the initiation and effect sets for each option—can be used to describe every set of states reachable by a sequence of option executions. Symbols naming them therefore form a complete symbolic vocabulary for the SMDP. Finally, we use the grounding sets of that vocabulary and each option's initiation set to compute a collection of operator descriptions that forms a complete symbolic model of the domain. This process is depicted in Figure 7.

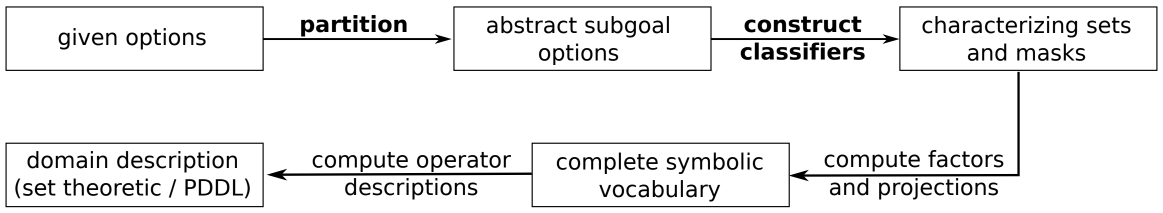

given options
partition
abstract subgoal options
construct classifiers
characterizing sets and masks
compute factors and projections
complete symbolic vocabulary
compute operator descriptions
domain description (set theoretic / PDDL)

Figure 7: The complete process for moving from an SMDP with abstract subgoal options to a symbolic representation. First, the options are partitioned so that the abstract subgoal property holds. Classifiers are then constructed representing the initiation and effect sets of each option. These characterizing sets are used to construct a set of factors describing state-variable changes in the domain, which are used to compute a collection of projections that form a complete symbolic vocabulary for the domain. Finally, this vocabulary is used to compute a domain description. The processes written in bold require either domain knowledge or data, while the remainder are purely computational processes that require no further data.

We will show that if the strong abstract subgoal condition holds, an abstract state computed using the symbolic model grounds to exactly the set of states reachable by the relevant plan. We also show that, while that grounding property is lost if only the weak abstract subgoal condition holds, the symbolic representation nevertheless returns the correct feasibility result for any plan tuple.

We formulate our model as a set-theoretic high-level domain specification expressed using PDDL, the *Planning and Domain Definition Language* (McDermott et al., 1998), which is the input format for most off-the-shelf general purpose planners. As described in section 2.2, a set-theoretic specification consists of a set of propositional symbols $\mathcal{P} = \{p_1, ..., p_n\}$ (to which we associate grounding classifiers $\mathcal{G}(p_i)$) and a set of operators $\mathcal{A} = \{\alpha_1, ..., \alpha_m\}$. A state $\mathcal{P}_t$ is obtained by assigning a truth value $\mathcal{P}_t(i)$ to every $p_i \in \mathcal{P}$, so we may consider abstract states to be binary vectors.

Each operator α_i is described by the tuple $\alpha_i = (\text{precond}_i, \text{effect}_i^+, \text{effect}_i^-)$, where $\text{precond}_i \subseteq \mathcal{P}$ lists all propositions that must be true in a state for the operator to be applicable at that state, and positive and negative effects $\text{effect}_i^+ \subseteq \mathcal{P}$ and $\text{effect}_i^- \subseteq \mathcal{P}$ list the propositions set to true or false, respectively, as a result of applying the operator. All other propositions remain unchanged when the operator is applied. Each operator describes the circumstances under which an option can be executed, and the resulting effect, though there may be multiple operators for each option. We assume that all options have abstract subgoals.

3.2.1 Grounding Semantics

A symbolic model must have semantics—a correspondence between any given abstract state $\mathcal{P}_t$ and a grounding set of states in the low-level SMDP. The semantics of the model allow us to reason about its correctness, even though once the model is built the grounding sets are unnecessary for planning.

Concretely, each state $\mathcal{P}_t$ must map to some grounded set of states; we denote that set of states by $\mathcal{G}(\mathcal{P}_t)$, and use the following grounding scheme:

$$\mathcal{G}(\mathcal{P}_t) = \cap_{i \in I} \mathcal{G}(p_i), \; I = \{i | \mathcal{P}_t(i) = \text{True}\}.$$

The grounding classifier of state $\mathcal{P}_t$ is the intersection of the grounding classifiers corresponding to the propositions set to true in that state. One can consider the propositions set to true as "on", in the sense that they are included in the grounding intersection, and those set to false as "off", and not included. Other schemes are possible—for example, intersecting with the negation of the propositions set to false—but this choice leads to natural operator models, as we describe later.

The key property of a deterministic PDDL domain description is that the operator descriptions and propositional symbols are sufficient for planning. Specifically, we should be able to correctly determine whether or not an option o_i can be executed at $\mathcal{P}_t$, and if so to determine a successor state $\mathcal{P}_{t+1}$, solely using the operator descriptions and the elements of $\mathcal{P}_t$, for any option o_i and time t.

The most direct way to guarantee this is to construct a set of operators that depend only on $\mathcal{P}$, and then show that the resulting model is *sound*:

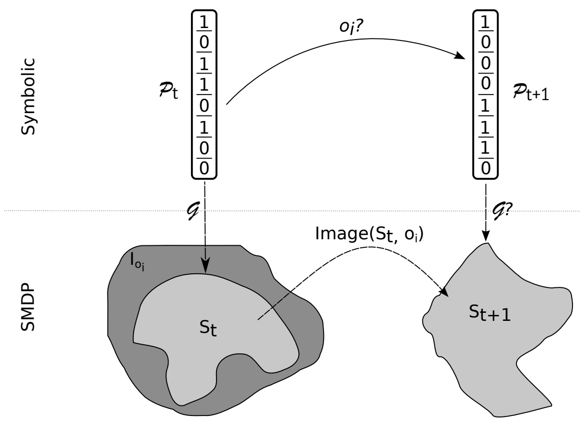


Figure 8: The soundness property for a symbolic model of a planning domain. The agent may be in some set of states S_t at time t, represented by the abstract state (binary vector) $\mathcal{P}_t$. We assume that the representation at time t grounds correctly: $\mathcal{G}(\mathcal{P}_t) = S_t$. The agent must be able to correctly calculate whether $S_t \subseteq I_{o_i}$, and compute the subsequent abstract state $\mathcal{P}_{t+1}$, using only its operator definitions. If o_i can be executed, the agent could find itself within the set of states S_{t+1}. The representation is sound if it computes $\mathcal{P}_{t+1}$ such that $\mathcal{G}(\mathcal{P}_{t+1}) = S_{t+1}$, for any o_i.

Definition 17. *A model is* sound *if, for any plan p, the grounding of the abstract state that the symbolic model predicts after executing p equals the actual set of states the agent could find itself in after executing p.*

This is depicted in Figure 8. If we can show that one step of computation with the abstract representation is sound for any option, then we can apply a simple induction argument to show that it holds for all the steps of any plan.

While soundness is convenient, a weaker condition suffices for abstract reasoning:

Definition 18. *A model is* suitable *if, for any plan p (of arbitrary length t) and option o_i, $(\bar{S}_{t+1} \subseteq I_{o_i}) = (S_{t+1} \subseteq I_{o_i})$, where $\bar{S}_{t+1} = \mathcal{G}(\mathcal{P}_{t+1})$ is the grounding of the abstract state $\mathcal{P}_{t+1}$ that the model predicts after executing p, and S_{t+1} is the actual set of states the agent could find itself in after executing p.*

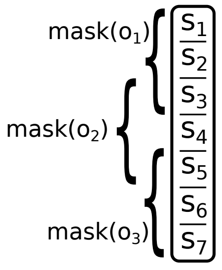
Rather than requiring that $\mathcal{G}(\mathcal{P}_{t+1}) = S_{t+1}$, suitability simply requires that the representation correctly predicts whether or not any option o_i can be executed after any sequence of other option executions. Here the representation loses some fidelity—$\bar{S}_{t+1}$ may not be equal to S_{t+1}—but retains the ability to correctly answer any feasibility question. The model we now construct is sound for options with the strong abstract subgoal property, and suitable for options with the weak abstract subgoal property.

3.2.2 Defining Factors

The first step in our approach is to identify the portions of the state space that change together when an option is executed. Given a low-level state vector $\mathbf{s} = [s_1, ..., s_n]$, we define a mapping from each individual low-level state variable to the set of options that change its value:

$$\text{modifies}(s_i) = \{o_j | i \in \text{mask}(o_j), o_j \in O\}.$$

We next partition $\mathbf{s}$ into factors, $f_1, ..., f_m$, each of which is the collection of all state variables modified by the same list of options. An example factorization is shown in Figure 9.



(a)

Factor	State Variables	Options
f_1	s_1, s_2	o_1
f_2	s_3	o_1, o_2
f_3	s_4	o_2
f_4	s_5	o_2, o_3
f_5	s_6, s_7	o_3

(b)

Figure 9: (a) An example low-level state vector with 7 variables, in a system with three option masks. (b) This collection of options results in five factors, each of which is a collection of low-level state variables modified by the same list of options.

We denote the set of options modifying the variables in factor f_i as options(f_i), and similarly the set of factors containing state variables that are modified by option o_j as factors(o_j). In a slight abuse of notation, we also denote the factors over which the grounding classifier for symbol σ_k is defined as either factors(σ_k) or factors(k).

3.2.3 Building the Symbolic Vocabulary

Executing abstract subgoal option o_i projects the factors it modifies (i.e., those that include the variables in its mask) out of the current grounding set; intersecting the result with Effect(o_i) computes the image. Future execution of any option o_j whose mask overlaps with those factors (i.e., where factors(o_i) $\cap$ factors(o_j) $\neq \emptyset$) will project the overlapping factors out of Effect(o_i). We therefore require a symbol naming the projection of each possible combination of factors out of each option's effects set, which can result in a combinatorial explosion in the number of symbols. However, we can represent the effect of projecting out a factor compactly if it obeys the following independence property:

Definition 19. *Factor f_s is independent in effect set Effect(o_i) iff Effect(o_i) can be written as Project(Effect(o_i), f_s) $\cap$ Project(Effect(o_i), factors(o_i) $\setminus$ f_s).*

Independence holds when the effect set can be decomposed into the intersection of two sets, one obtained by projecting f_s out of the effect set, and one obtained by projecting out all the other variables.[4] Let Effect$_r$(o_i) denote the set remaining after projecting out all

4. This might seem similar to the condition under which two random variables are statistically independent: we can write their joint probability distribution as the product of their individual distributions. We will see in Section 4, when we generalize sets to distributions, that the probabilistic version of this condition is exactly statistical independence.

independent factors from $\text{Effect}(o_i)$, and $\text{factors}_r(o_i)$ denote the remaining factors (if any). The effect set $\text{Effect}(o_i)$ for option o_i can now be rewritten as:

$$\text{Effect}(o_i) = \text{Effect}_r(o_i) \cap (\cap_j F_j),$$

where each set F_j is obtained by projecting all factors except independent factor f_j out of the effect set. The key advantage that results from independence is that projecting out independent factor f_j from this expression is achieved by simply deleting F_j from the conjunction. We therefore need not construct separate symbols naming $\text{Effect}(o_i)$ with every combination of independent factors projected out.

We will therefore require one proposition for each independent factor, and additionally propositions for $\text{Effect}_r(o_i)$ with every possible subset of $\text{factors}_r(o_i)$ projected out. Therefore, we construct a vocabulary $\mathcal{P}$ containing the following propositional symbols:

1. For each option o_i and factor f_s independent in $\text{Effect}(o_i)$, create a propositional symbol with grounding classifier $\text{Project}(\text{Effect}(o_i), \text{factors}(o_i) \setminus f_s)$.

2. For each collection of factors $f_r \subseteq \text{factors}_r(o_i)$, create a propositional symbol with grounding classifier $\text{Project}(\text{Effect}_r(o_i), f_r)$.

In the above process we discard duplicate propositions (those with the same grounding set as an existing proposition), and propositions naming the entire state space (which do not affect the grounding intersection). We will now show that this collection of symbols is a sufficient vocabulary for modeling the low-level SMDP by using them to construct a complete set of operators.

3.2.4 Constructing Operators

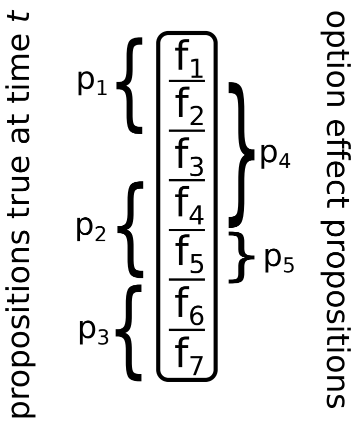
Executing option o_i results in the following effects:

1. All propositional symbols with grounding classifiers for each independent component of $\text{Effect}(o_i)$, and an additional proposition with grounding symbol $\text{Effect}_r(o_i)$ if necessary, are set to true.

2. All propositional symbols with grounding classifier $A \subseteq I_{o_i}$ such that $\text{factors}(A) \subseteq \text{factors}(o_i)$ are set to false.

3. All currently true propositional symbols with grounding classifier $B \subseteq I_{o_i}$, where $f_{bi} = \text{factors}(B) \cap \text{factors}(o_i) \neq \emptyset$ but $\text{factors}(B) \not\subseteq \text{factors}(o_i)$, are set to false. For each such B, the proposition with grounding symbol $\text{Project}(B, f_{bi})$ is set to true.

Recall that we compute the image of an abstract subgoal option o_i from a set of states Z using the equation $\text{Image}(Z, o_i) = \text{Project}(Z, o_i) \cap \text{Effect}(o_i)$. The first effect type in the above list corresponds to the $\text{Effect}(o_i)$ component of the image equation, while the remaining two types model the $\text{Project}(Z, o_i)$ component. The second type negates propositions defined entirely using variables within $\text{mask}(o_i)$—they are completely overwritten by the projection operator. The third type of effect models the side-effects of the option, where a true proposition defined over variables that overlap with $\text{mask}(o_i)$ has those variables projected out of its grounding classifier. This is depicted in Figure 10.



At time t, the following propositions are true:

$$p_1,\ p_2,\ \text{and}\ p_3.$$

After executing the option, the following propositions are true:

$$p_3,\ p_4,\ p_5,\ \text{and}\ p_6,$$

where p_6 is the proposition naming Project($\mathcal{G}(p_1)$, f_2). Propositions p_1 and p_2 are set to false.

Figure 10: An example effect computation. At time t, propositions p_1, p_2, and p_3 are true. The option's effect set is represented by propositions p_4 and p_5 (which is defined over independent factor f_5). After execution, proposition p_3 remains true because its factors lie completely outside of the option's mask, while p_2 is set to false because its factors lie completely inside the mask. The factors used by p_1 are only partially covered, so its proposition is set to false and proposition p_6, which names Project($\mathcal{G}(p_1)$, f_2), is set to true. p_4 and p_5 name the option's direct effects and are set to true.

We now formalize our description above to show that the resulting one-step domain model is sound for an executable option that obeys the strong abstract subgoal property:

Theorem 2. *Given abstract state descriptor $\mathcal{P}_t$, strong abstract subgoal option o_j such that $\mathcal{G}(\mathcal{P}_t) \subseteq I_{o_j}$, and $\mathcal{P}_{t+1}$ computed as described above, $\mathcal{G}(\mathcal{P}_{t+1}) = \text{Image}(\mathcal{G}(\mathcal{P}_t), o_j)$.*

Proof. Recall our grounding semantics:

$$\mathcal{G}(\mathcal{P}_t) = \cap_{i \in I} \mathcal{G}(p_i),\ I = \{i | \mathcal{P}_t(i) = \text{True}\},$$

and the definition of the image operator for strong abstract subgoal options:

$$\text{Image}(\mathcal{G}(\mathcal{P}_t), o_j) = \text{Project}(\mathcal{G}(\mathcal{P}_t), o_j) \cap \text{Effect}(o_j).$$

Substituting, we see that:

$$\begin{aligned}\text{Image}(\mathcal{G}(\mathcal{P}_t), o_j) &= \text{Project}(\cap_{i \in I} \mathcal{G}(p_i), o_j) \cap \text{Effect}(o_j) \\ &= \cap_{i \in I} \text{Project}(\mathcal{G}(p_i), o_j) \cap \text{Effect}(o_j).\end{aligned}$$

We can split I into three sets: I_u, which contains indices for propositions with grounding symbols whose factors do not overlap with option o_j; I_r, which contains propositions with grounding symbols whose factors are a subset of mask(o_j); and I_o, the remaining propositions with grounding symbols whose factors overlap. Projecting out mask(o_j) leaves the symbols in I_u unchanged, and replaces those in I_r with the universal set, so:

$$\text{Image}(\mathcal{G}(\mathcal{P}_t), o_j) = \cap_{i \in I_u} \mathcal{G}(p_i) \cap_{i \in I_o} \text{Project}(\mathcal{G}(p_i), o_j) \cap \text{Effect}(o_j).$$

For each $i \in I_o$, we can find a k such that $\mathcal{G}(p_k) = \text{Project}(\mathcal{G}(p_i), o_j)$, because we have explicitly constructed all such propositions. Let K be the set of such indices. Then we can write:

$$\text{Image}(\mathcal{G}(\mathcal{P}_t), o_j) = \cap_{i \in I_u} \mathcal{G}(p_i) \cap_{k \in K} \mathcal{G}(p_k) \cap \text{Effect}(o_j).$$

The first term on the right hand side of the above equality corresponds to the propositions untouched by option execution; the second term to the third class of effects propositions (those resulting from a partial overwrite by the option); the third term to the first class of effects propositions (the option's direct effects); and I_r to the second class of effects propositions (those completely overwritten as they fall inside the option's mask). These are exactly the effects enumerated by our operator model, so:

$$\text{Image}(\mathcal{G}(\mathcal{P}_t), o_j) = \mathcal{G}(\mathcal{P}_{t+1}).$$

□

Finally, we must compute the operator preconditions, which express the conditions under which the option can be executed, again in terms of our symbolic vocabulary. To do so, we consider I_{o_i} for each option o_i, and the factors it is defined over, factors(I_{o_i}).

Our operator effects model ensures that no two propositions with grounding symbols defined over the same factor can be true at the same time. We can therefore enumerate all possible "assignments" of factors to symbols, compute the resulting grounded state classifier for each, and determine whether it is a subset of I_{o_i}. If so, the option can be executed, and we output an operator description with the appropriate preconditions and effects.

Note that the third type of effect (expressing the option's side effects) depends on propositions other than those used to evaluate whether or not the option can be executed. We can express these cases either by creating a separate operator description for each possible assignment of relevant propositions, or more compactly via PDDL's support for conditional effects.

The proof that the generated preconditions are correct follows directly from the definition, so we omit it here. Together with Theorem 2 the correctness of the generated preconditions means that, given strong abstract subgoal options, the model constructed above both correctly determines when an option can be executed and computes a symbolic expression that grounds to exactly the correct set of states. The resulting model is therefore sound for strong abstract subgoal options. While weak abstract subgoal options may not result in the correct grounding (specifically because Theorem 2 does not hold), the definition of the weak abstract subgoal property ensures that the resulting inaccuracy never changes the feasibility of a plan; the resulting representation is therefore suitable.

Since the procedure for constructing operator descriptions given in this section uses only the finite set of propositions enumerated in section 3.2.3, it follows that $\mathcal{P}$ is a sufficient symbolic vocabulary for describing the system. Note also that all of the elements of $\mathcal{P}$ can be defined in terms of only option initiation sets, effect sets, and masks.

3.2.5 The Algorithm

Algorithm 1 formalizes the procedures described above in pseudo-code, proceeding in three stages: factor generation, symbol enumeration, and operator description generation.

Algorithm 1 Generate a PDDL Domain Description From Characterizing Sets

```
1: ▷ Initialize.
2: $\mathcal{P}, F \leftarrow \emptyset$

3: ▷ Compute factors.
4: for $i \in \{1, ..., n\}$ do
5:     if $\exists f_j \in F$ s.t. $\text{options}(f_j) = \text{modifies}(s_i)$ then
6:         $f_j \leftarrow f_j \cup s_i$
7:     else
8:         $F \leftarrow F \cup \{s_i\}$
9:     end if
10: end for

11: ▷ Generate symbol set.
12: for all $o_i \in O$ do
13:     $f \leftarrow \text{factors}(o_i)$, $e \leftarrow \text{Effect}(o_i)$

14:     ▷ Identify independent factors.
15:     for all $f_i \in f$ s.t. $\text{Project(e, } f_i) \cap \text{Project(e, } f \setminus f_i) = e$ do
16:         $\mathcal{P} \leftarrow \mathcal{P} \cup \sigma_{\text{Project(e,}f \setminus f_i)}$, e $\leftarrow$ Project(e, $f_i$), $f \leftarrow f \setminus f_i$
17:     end for

18:     ▷ Project out all combinations of remaining factors.
19:     for all $f_s \subset f$ do
20:         $\mathcal{P} \leftarrow \mathcal{P} \cup \sigma_{\text{Project(e,}f_s)}$
21:     end for
22: end for

23: ▷ Generate operator descriptors.
24: for all $o_i \in O$ do
25:     ▷ Direct effects.
26:     $\text{effects}_+(o_i) \leftarrow \{\sigma | \sigma \in \mathcal{P}, \text{refersToEffect}(\sigma, o_i)\}$
27:
28:     ▷ Side effects: full and partial overwrites.
29:     $\mathcal{P}_{nr} = \{\sigma | \sigma \in \mathcal{P}, \neg\text{refersToEffect}(\sigma_r, o_i)\}$
30:     $\text{effects}_-(o_i) \leftarrow \{\sigma | \sigma \in \mathcal{P}_{nr}, \mathcal{G}(\sigma) \subseteq I_{o_i}, \text{factors}(\sigma) \subseteq \text{factors}(o_i)\}$
31:     $\text{conditionalEffects}_-(o_i) \leftarrow \{\sigma_1, \sigma_2 | \sigma_1, \sigma_2 \in \mathcal{P}_{nr}, \mathcal{G}(\sigma_1) \subseteq I_{o_i},$
                                     $\text{factors}(\sigma_1) \cap \text{factors}(o_i) \neq \emptyset,$
                                     $\mathcal{G}(\sigma_2) = \text{Project}(\mathcal{G}(\sigma_1), \text{factors}(o_i))\}$
32:     ▷ Compute preconditions.
33:     for all $\mathcal{P}_c \subseteq \mathcal{P}$ s.t. $\text{factors}(I_{o_i}) \subseteq \cup_{\sigma \in \mathcal{P}_c} \text{factors}(\sigma)$,
34:                         $\cap_{\mathcal{G}(\sigma \in \mathcal{P}_c)} \subseteq I_{o_i}$, $\neg\text{factorsOverlap}(\mathcal{P}_c)$ do
35:         $\text{preconditions}(o_i) \leftarrow \text{preconditions}(o_i) \cup \mathcal{P}_c$
36:     end for
37: end for
```

Computing the factors requires time in $O(n|F||O|)$, where n is the dimensionality of the original low-level state space, and F, the set of factors, has at most n elements. Enumerating the symbol set requires time in $O(|O||F|c)$ for the independent factors and $O(|O|2^{|F|}c)$ for the dependent factors in the worst case, where each set operation is $O(c)$. This results in a symbol set of size $|\mathcal{P}| = O(F_I + 2^{F_D})$, where F_I indicates the number of symbols resulting from independent factors, and F_D the number of factors referred to by dependent effects sets. Computing the operator effects requires time $O(|\mathcal{P}||O|)$, and the preconditions requires time $O(|O||\mathcal{P}|^{|F|})$ in the worst case.

Although Algorithm 1 is guaranteed to terminate and will output a correct PDDL domain description for any SMDP with abstract subgoals, it has a potentially disastrous worst-case complexity, and the description it produces can be exponentially larger than the number of options. We expect that conversion will be most useful in problems where the number of factors is small compared to the number of original state variables ($|F| \ll n$), most or all of the effects sets are independent in all of their factors ($F_D \ll F_I$), the symbol enumeration process generates many of the same symbols repeatedly (keeping $|\mathcal{P}|$ low), and the the majority of precondition computations either fail early, or succeed with a small number of factors. These properties are typically necessary for a domain to have a compact PDDL encoding; as we shall see in the following section, they hold and allow very fast planning in a hierarchical reinforcement learning domain originating outside of the planning literature.

3.3 High-Level Planning in the Continuous Playroom Domain

In the continuous playroom domain (Singh et al., 2005; Konidaris & Barto, 2009a), an agent with three effectors (an eye, a hand, and a marker) is placed in a room with five objects (a light switch, a bell, a ball, and red and green buttons) and a monkey; the room also has a light (initially off) and music (also initially off). The agent is given options that allow it to move a specified effector over a specified object (always executable, moving the effector toward the object until is within 0.05 units of it in both x and y), plus an "interact" option for each object (for a total of 20 options). The effectors and objects are arranged randomly at the start of every episode; one arrangement is depicted in Figure 11.

Interacting with the buttons or the bell requires the light to be on, and the agent's hand and eye to be placed over the relevant object. The ball and the light switch are brightly colored and can be seen in the dark, so they do not require the light to be on. Interacting with the green button turns the music on; the red button turns it off. Interacting with the light switch turns the lights on or off. Finally, if the agent's marker is on the bell and it interacts with the ball, the ball is thrown at the bell and makes a noise. If this happens when the lights are off and the music is on, the monkey cries out, and the episode ends.

The domain description given above is high-level and abstract; however, it does not reflect the problem's actual state space, which is continuous and high dimensional. The problem state can be described by 19 real-valued variables: the x and y position of each of the eight objects, the observed light level (0 when the light is off, dropping from a value of 1 by the square of the eye's distance to the center of the room when it is on), the music volume (selected at random from the range $[0.3, 1.0]$ when the green button is pressed), and whether the monkey has cried out. We use 33 continuous features computed from the state

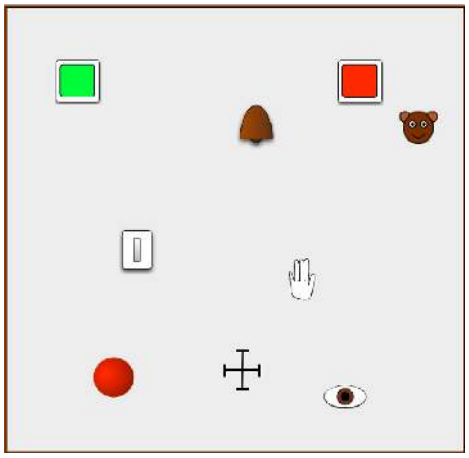


Figure 11: An instance of the continuous playroom domain.

variables, describing the x and y distance between each of the agent's effectors and each object instead of their absolute positions, plus the music, light, and monkey scream, as is standard for this domain (Konidaris & Barto, 2009a). As an illustrative example, we now show—by hand, using decision trees as grounding classifiers for legibility—how to construct an abstract symbolic model from these low-level features.

3.3.1 Constructing an Abstract Representation

We begin with the background knowledge that all of the options can be partitioned into options that obey the weak subgoal property. For example, when moving the hand over the red button, the start location of the hand does determine where exactly it stops, since it determines the direction from which it approaches the button—therefore, the strong abstract subgoal condition does not hold. However, that difference—for example, being 0.05 units to the left or to the right of the red button—is immaterial when deciding whether another option (e.g., interact) can be executed.

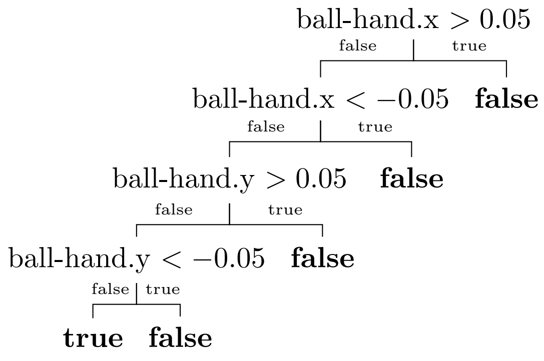
We begin with the options that move an effector over an object, as these are the simplest to model. They have the following properties:

- Effector movement options are executable in every state.
- Moving an effector changes the x and y distance between that effector and every object; it places the effector over the target object, and sets the distances to every other object to some other value that can be considered random (because it will depend on the random arrangement of objects in that particular task instance).
- If the light is on, moving the eye changes the observed light level. Other effector movement options leave the observed light level unchanged.

Thus, for all effector movement options except those involving the eye, the initiation set is simply true everywhere. Figure 12a lists the variables that are changed by moving the hand effector over the ball—its mask—and Figure 12b shows the resulting effect set

```
greenbutton-hand.x
greenbutton-hand.y
redbutton-hand.x
redbutton-hand.y
bell-hand.x
bell-hand.y
ball-hand.x
ball-hand.y
lightswitch-hand.x
lightswitch-hand.y
```

(a)



(b)

Figure 12: (a) The list of variables that can change and (b) the effect set decision tree for the option that moves the hand over the ball. The variables indicate that the distance between the effector and any other object in the domain can be changed; the decision tree indicates that option execution leaves the x and y distances between the hand and the ball within a particular range.

decision tree. These two pieces of information allow us to compute the image of the option from any start state set: we project out the variables that change from the decision tree representing the start set, and then intersect the result with the effect set decision tree.

Modeling the options involving the eye is slightly more involved, because the perceived light level changes with eye movement if the light is on. The option has two different masks depending on which state it is executed from; we must therefore partition it based on light level. This results in the two characterizing sets shown in Figure 13.

The collection of all of these decision tree classifiers captures all the information necessary to plan in the domain. We converted the classifiers to PDDL using a short program written in Java, which executed in approximately half a second. This process identified 6 factors, three containing variables describing the distance between an effector and all of the objects, and one each for the music, the lights, and the monkey's cry. These were used to generate 20 unique grounded symbols—all effects sets were composed only of independent factors—and a PDDL domain description file. An example operator description, along with the relevant symbol groundings, is shown in Figure 14.

Figure 15 shows the grounding classifier for a start symbol and three example goal symbols. Each plan starts from the set of all states where the monkey is not crying out and the lights and music are off, and the planner must return a plan guaranteed to reach each goal with probability 1 from any state in the start set, for any random configuration of the domain. We know of no sample-based SMDP planner that can provide similar guarantees.

Timing results for planning using the automatically generated PDDL description as input to version 2.3 of the FF planner (Hoffmann & Nebel, 2001) are given in Table 1. We also compare to a simple breadth-first planner that directly uses the grounding classifiers themselves at runtime, performing the corresponding logical operations as necessary; this

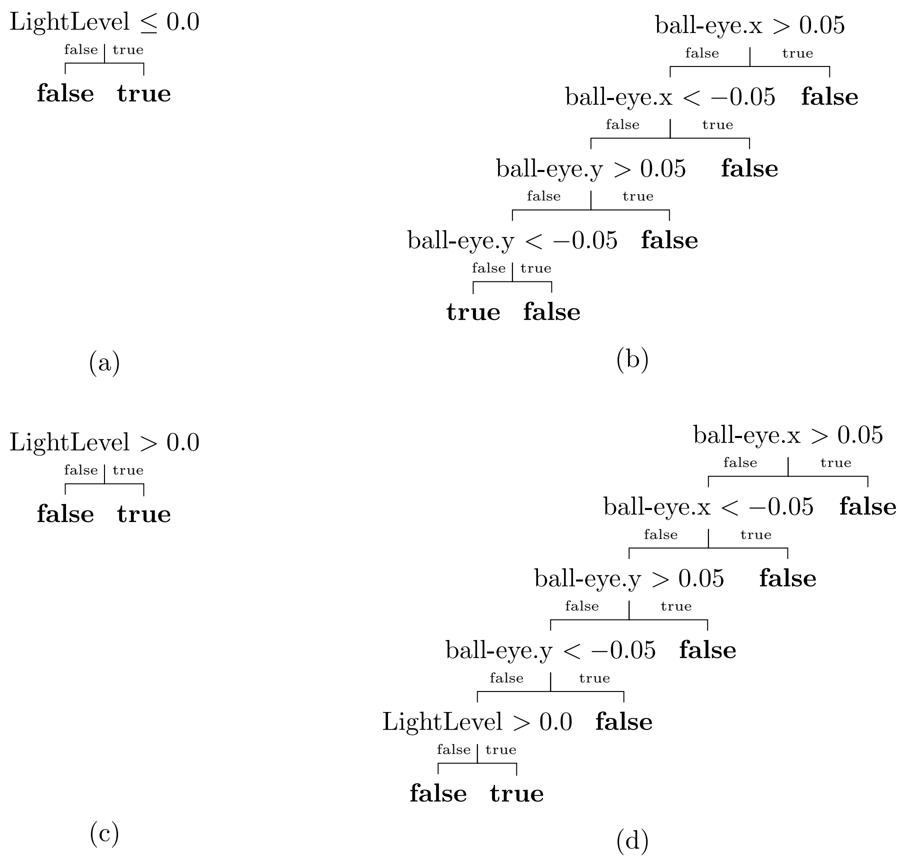


Figure 13: The two characterizing sets for the option that moves the eye over the ball. (a) The grounding classifier for the precondition of the first characterizing set includes only states where the light is off. Its effect mask does not include `LightLevel`, and its effects set classifier (b) indicates that the relationship between the eye and the ball is set to a specific range. (c) The grounding classifier for the precondition of the second characterizing set includes only states where the light is on. Its effects mask includes the `LightLevel` variable, which its effect set classifier (d) indicates changes to some positive value after option execution (due to eye movement).

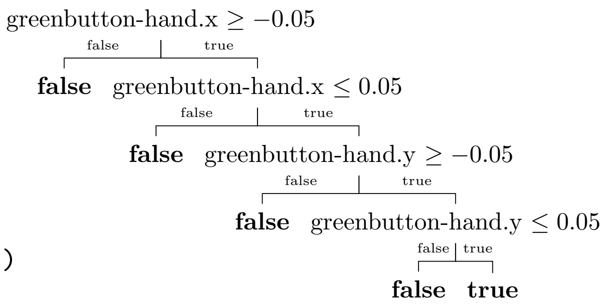
corresponds to semi-symbolic planning using the grounding sets themselves, rather than an abstract model computed using them. Both systems can solve the resulting search problem within a few seconds, though FF (which is a modern planner with a sophisticated heuristic search strategy) can solve it roughly a thousand times faster than the (admittedly naive) planner that operates directly on the grounding sets.

3.3.2 Learning a Symbolic Representation From Experience

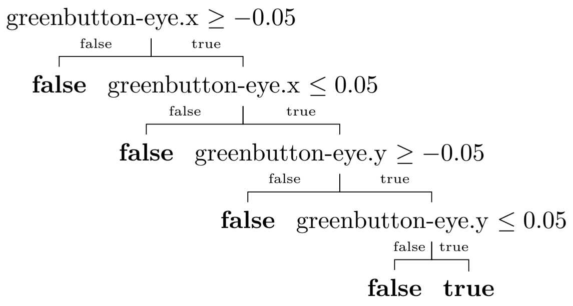
A key property of our framework is that it depends only on identifying the relevant initiation set and effect set classifiers, plus partitioning so that they obey at least the weak abstract

```
(:action interact_greenbutton
 :parameters ()
 :precondition (and (symbol1)
                    (symbol2)
                    (symbol3))
 :effect (and (symbol17)
              (not (symbol18))))
```

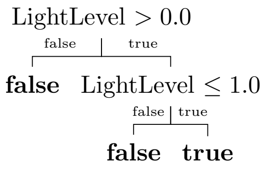
(a) Generated PDDL Operator


greenbutton-hand.x ≥ −0.05
false
true
false
greenbutton-hand.x ≤ 0.05
false
true
false
greenbutton-hand.y ≥ −0.05
false
true
false
greenbutton-hand.y ≤ 0.05
false
true
false
true

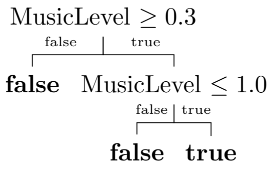
(b) `symbol1`



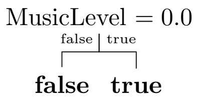
(c) `symbol2`



(d) `symbol3`



(e) `symbol17`



(f) `symbol18`

Figure 14: (a) The automatically generated PDDL descriptor for interacting with the green button, along with the groundings of the five automatically generated symbols it refers to (b–f). The operator symbolically expresses the precondition that *the hand and eye are over the green button, and the light is on*, and that, as a result of executing the operator, *the music is on, and the music is no longer off.*

subgoal property; building the completely symbolic PDDL description is then just a process of offline computation. If an agent is able to determine whether or not an option can be run at a particular state, then it can simply learn its initiation set by interacting with the domain and collecting data of the form $(s, I_o(s))$ for each option, and using it as training data for a classifier. Similarly, it can learn the effect set by simply executing the option and collecting transition data of the form (s, o, s'), and using the s' data as input for a one-class classifier.

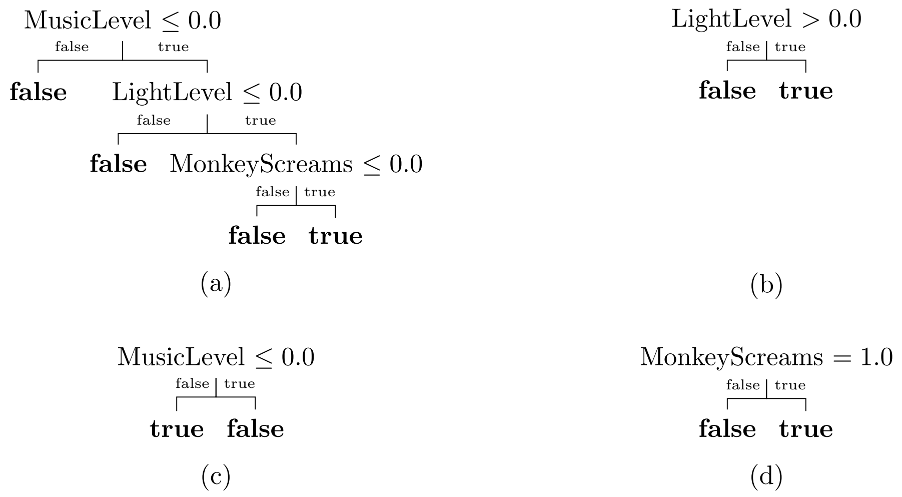


Figure 15: The start (a) and goal (b–d) symbols for three example planning problems. Each plan starts from the set of all states where the monkey is silent and the lights and music are off. The first plan requires the agent to switch on the lights; the second, to switch on the music; and the third, to make the monkey cry out.

		BFS with Grounding Sets		**PDDL (FF)**	
Goal	**Depth**	**Visited**	**Time (s)**	**Visited**	**Time (ms)**
Lights On	3	199	1.35	4	1.99
Music On	6	362	2.12	9	2.30
Monkey Cry	13	667	3.15	30	2.77

Table 1: The time required, and number of nodes visited, for the example planning problems given in Figure 15. Results were obtained on a MacBook Pro with a 2.5Ghz Intel Core i5 processor and 8GB of RAM.

To demonstrate this, we gathered $5,000$ positive and negative examples[5] of each option's initiation set and effects set by repeatedly creating a new instance of the playroom domain and sequentially executing options at random. For the effects set, we used option termination states as positive examples, and states encountered during option execution but before termination as negative examples. We used the WEKA toolkit (Hall, Frank, Holmes, Pfahringer, Reutemann, & Witten, 2009) C4.5 decision tree (Quinlan, 1993) learner to produce easy to visualize decision trees. A representative learned initiation set, for interacting with the green button, is given in Figure 16; an example learned effect set for moving the marker over the red button is shown in Figure 17.

A comparison between the numeric values in Figures 12 and 13 (where the classifiers are constructed by hand) and Figures 16 and 17 (where they are learned) shows that classifiers

5. This number was chosen arbitrarily, and does not reflect the difficulty of learning the relevant classifiers.

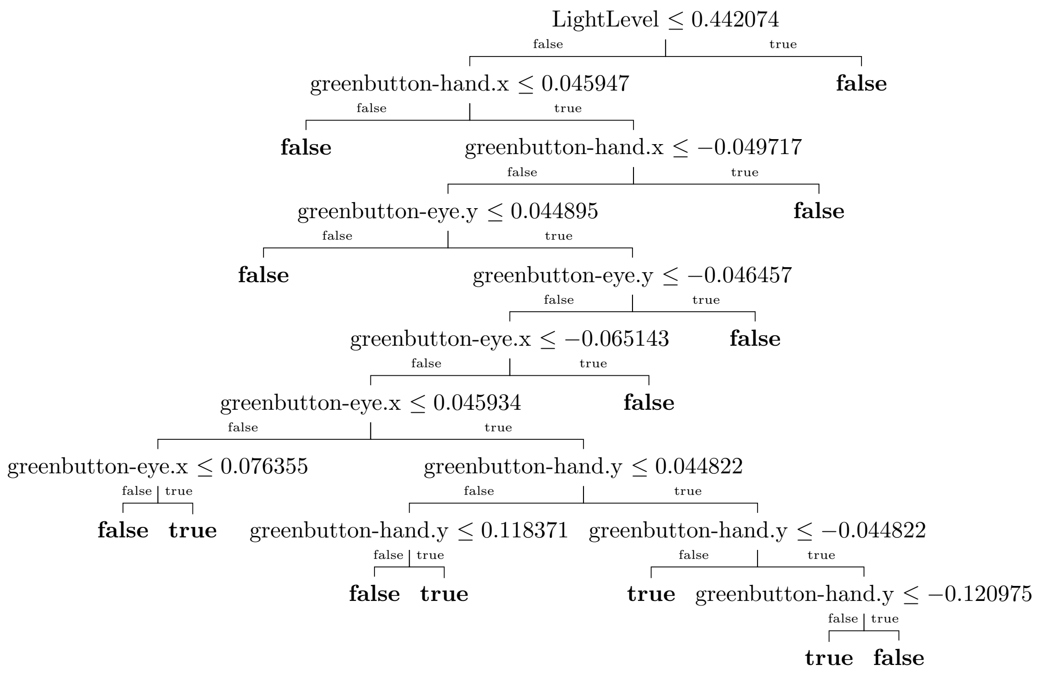

LightLevel ≤ 0.442074
false
true
greenbutton-hand.x ≤ 0.045947
false
greenbutton-hand.x ≤ −0.049717
greenbutton-eye.y ≤ 0.044895
greenbutton-eye.y ≤ −0.046457
greenbutton-eye.x ≤ −0.065143
greenbutton-eye.x ≤ 0.045934
greenbutton-eye.x ≤ 0.076355
greenbutton-hand.y ≤ 0.044822
greenbutton-hand.y ≤ 0.118371
greenbutton-hand.y ≤ −0.044822
greenbutton-hand.y ≤ −0.120975

Figure 16: The decision tree representing the learned initiation set for interacting with the green button. It approximately expresses the set of states we might label *the hand and eye are over the green button, and the light is on.*

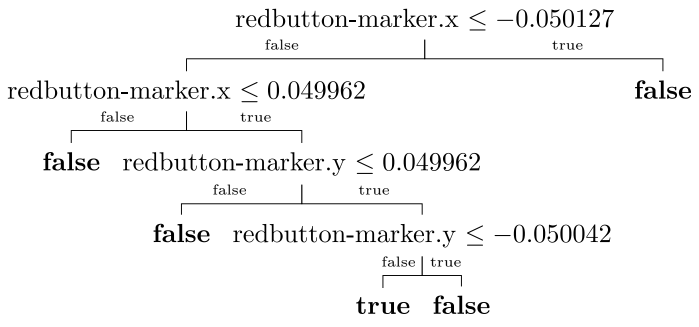

redbutton-marker.x ≤ −0.050127
false
true
redbutton-marker.x ≤ 0.049962
redbutton-marker.y ≤ 0.049962
redbutton-marker.y ≤ −0.050042

Figure 17: The decision tree representing the learned effects set of moving the marker to the red button. It approximately expresses the set of states we might label *the marker is over the red button.*

learned in this way will be unsuitable for building a symbolic representation. Even in a deterministic domain with no observation noise, learned symbols are in practice always approximate. This can easily result in plans being evaluated as infeasible when they are in fact feasible. For example, in the continuous playroom the range of x-distances required for interaction between an object and effector is exactly equal to the range of x-distances in the effects set for the option that moves that same effector over that same object. If the approximation of that range is too wide in the effects set or too narrow in the initiation set—even in only one variable, even by only a small amount—then we will incorrectly determine that the two options cannot be chained.

There are three causes of this difficulty. The first is that the formalization of symbols as sets effectively requires us to assume that all transitions with a non-zero probability of occurring are equally likely. The second is that the set-based formalism does not account for the uncertainty inherent in *learning* the symbols themselves. Instead, planning proceeds as if the estimated characterizing sets are exactly correct. Finally, deterministic planning asks only whether or not a plan can be executed with probability 1. None of these assumptions are well suited to real robotics problems; we therefore now turn to probabilistic symbols and plans.

4. Constructing Symbolic Representations for Probabilistic Planning

The previous section has shown that if the task we wish to solve with an abstract representation is deterministic planning, then that representation should be based on the classical semantics of propositional logic: it should be based on symbols naming sets of states. However, we have also seen that deterministic planning and the formalisms it demands are not well suited for real problems.

We now consider the question of constructing a representation suitable for *probabilistic planning*. As we shall see, this requires generalizing our formalism from symbols naming sets (manipulated using set operators) to symbols naming probability distributions (manipulated using probabilistic operations). The resulting formalism includes two important notions of uncertainty: it can express *the probability that a plan can be executed*, and it can deal with *uncertain symbols* (such as are obtained during learning) in a principled way. In addition, it supports computing the *expected reward* obtained when executing a plan.

As before, we first precisely define the class of questions that the target representation must answer, by defining a probabilistic plan and the corresponding plan space:

Definition 20. *A probabilistic plan $p = \{o_1, ..., o_{p_n}\}$ from a start state distribution Z is a sequence of options $o_i \in O$, $1 \leq i \leq p_n$, to be executed from a state drawn from Z.*

Here we have replaced a *set* of start starts with a *distribution* over start states. This is again largely for uniformity, as will become clear, although it could also reflect uncertainty in the agent's current state (e.g., when it is estimated from noisy, partially observable data).

Definition 21. *The* probabilistic plan space *for an SMDP is the set of all tuples (Z, p), where Z is a* start distribution *and p is a plan.*

The essential function of a symbolic representation for probabilistic planning is to compute, on demand, the probability that an agent can execute any element of the plan space to completion, and the expected reward received for successfully doing so. An example process for computing the probability of execution is shown in Figure 18.

In the deterministic setting we aimed to compute feasibility, and showed that the relevant computation consists exclusively of sets and set operations; in Figure 18 we see that

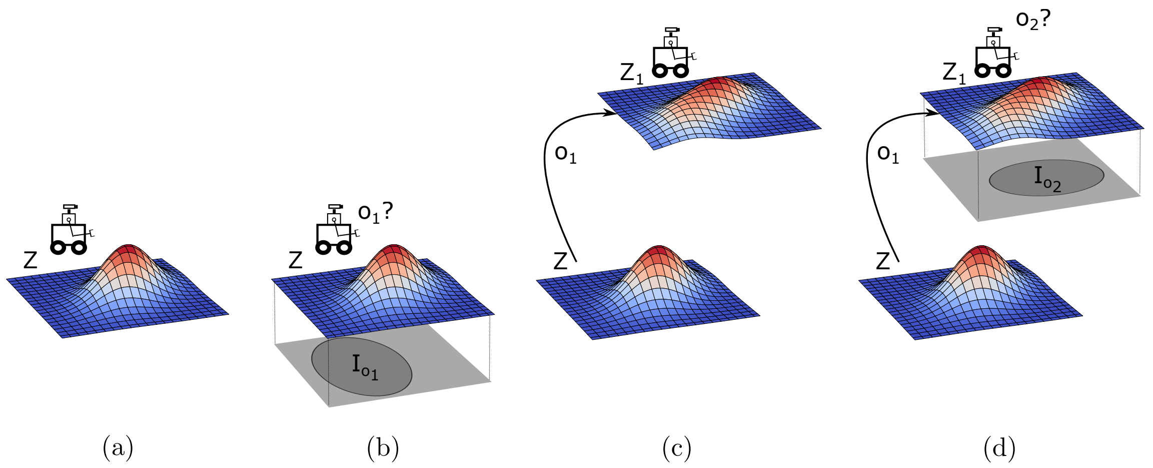


Figure 18: Determining the probability that an agent can execute the plan $p = \{o_1, o_2\}$. (a) The agent begins in a state drawn from a start distribution, Z. (b) It must first determine the probability that it can execute o_1; this is the probability that a state drawn from Z is in I_{o_1}: $e_1 = P(s \in I_{o_1} | s \sim Z)$. Having done so, it must determine the probability of executing o_2. (c) Let Z_1 be the distribution of states it might find itself in after having successfully executed o_1 from a state drawn from Z. (d) The probability with which it can execute o_2 is the probability that a state drawn from Z_1 is in I_{o_2}: $e_2 = P(s \in I_{o_2} | s \sim Z_1)$. The probability of being able to execute the entire plan is the product of the probabilities of being able to execute each step: $e_1 \times e_2$.

computing the probability of execution involves a product of probabiities:

$$P(e_1|Z) \times P(e_2|Z, e_1) \times P(e_3|Z, e_1, e_2) \times ... \times P(e_n|Z, e_1, ..., e_{n-1}),$$

where e_i is the probability of successfully executing the ith option in the plan, conditioned on having successfully executed all previous options starting from state distribution Z. These probabilities can always be expressed using only two types of objects: a distribution over states corresponding to the image operator, and (given that it may be learned from data and therefore uncertain) a probabilistic classifier corresponding to the initiation set. These two types of objects can be viewed as generalizations of *sets* to *probability distributions* in two different different senses; we now formalize these generalizations and the symbols naming them.

4.1 Symbols for Probabilistic Planning

In one sense, a set is a collection of states which the agent could find itself in, but with no information about which states are more likely and which are less. We can generalize this to a distribution over states:

Definition 22. *A distributional symbol σ_Z is the name associated with a distribution, Z, over states.*

The task of learning the grounding distribution of a distributional symbol is the problem of *density estimation*. Distributional symbols are used to express a distribution over start states and the probabilistic image operator which, given a start state distribution and *assuming successful option execution*, returns the distribution over states in which the agent may find itself:

Definition 23. *Given a start distribution Z and an option o, we define the probabilistic image of o from Z as:*

$$Image(Z, o) = \frac{\int_S P(s'|s, o)Z(s)P(s \in I_o)\,\mathrm{d}s}{\int_S Z(s)P(s \in I_o)\,\mathrm{d}s}.$$

The probabilistic image operator takes as input a start distribution and an option, and returns a distribution over states (i.e., a function of s).

In another sense, a set is a collection of states in which some test or condition holds. We generalize this to the probability that a condition holds in each state:

Definition 24. *A conditional symbol σ_E is the name associated with a function $P(C(s) = True)$ that returns the probability of a condition C holding at state $s \in S$.*

Conditional symbols name *probabilistic classifiers*. They are used to define the probabilistic precondition symbol, which expresses the probability that an option can be executed from each state:

Definition 25. *The* probabilistic precondition *is a conditional symbol defined as Pre(o) = $\sigma_{P(s \in I_o)}$.*

The probabilistic precondition operator takes as input an option and returns a probabilistic classifier—a function mapping s to a probability. Since the agent operates in an SMDP, a particular state is always either in I_o or not. However, the agent must necessarily generalize when learning an explicit representation of its initiation set, and here the probabilistic classifier expresses the agent's uncertainty about states it has not yet encountered.

These generalizations of the precondition set and image operator allow us to prove the probabilistic generalization of Theorem 1:

Theorem 3. *Given an SMDP, the ability to represent the probabilistic preconditions of each option and to compute the probabilistic image operator is sufficient to determine the probability of being able to execute any probabilistic plan tuple (σ_Z, p).*

Proof. Consider an arbitrary plan tuple (σ_Z, p), with plan length n. To determine the probability of executing p from σ_Z, we can set $Z_0 = Z$ and repeatedly compute $Z_{j+1} = \text{Image}(Z_j, p_j)$, for $j \in \{1, ..., n\}$. The probability of being able to execute the plan is given by $\Pi_{j=1}^{n} \left[\int P(s \in I_{o_j})Z_{j-1}(s)\,\mathrm{d}s\right]$. □

Computation proceeds analogously to the discrete, deterministic case—starting with an initial distribution over states, the agent repeatedly applies the image operator to obtain the distribution over low-level states it expects to find itself in after executing each option in the plan. It can thereby compute the probability of being able to execute each successive option, and multiply these probabilities to compute the probability of successfully executing the entire plan.

To obtain the expected reward obtained when successfully executing the plan, we require one additional operator:

Definition 26. *Given a start distribution Z and an option o, the reward operator is:* $J(Z, o) = \int_S \int_S \int_{\mathbb{R}_+} P(s', \tau | s, o) R(s', \tau | s, o) Z(s) \, \mathrm{d}\tau \, \mathrm{d}s' \, \mathrm{d}s$.

The expected reward of a plan tuple (σ_{Z_0}, p) of length n is then $\sum_{i=1}^{n} J(Z_{i-1}, p_i)$, where each state distribution Z_i is defined as in Theorem 3. Although the definition of the reward operator involves three integrals, during learning we use the following equation:

$$J(s, o) = \mathbb{E}_{s', \tau} \left[R(s', \tau | s, o) \right],$$

which simply estimates the expected reward for executing an option from each state. The reward obtained after option execution from a state is a sample of the right hand side of this equation, resulting in a standard supervised learning regression problem. Computing the expected reward now requires integration over the distribution over start states only.

The symbols and operators described above are defined in terms of option models. One approach to planning in SMDPs is to learn the option models themselves, and use them to perform sample-based planning. However, when the characteristics of the available options support it, the agent can go further and construct completely symbolic models it can use to plan—after which its grounding distributions are no longer required.

4.1.1 Subgoal Options and Abstract Subgoal Options

As in the deterministic planning case, computing the probabilistic image operator is hard in general. Instead, we might wish to substitute an effect distribution that is independent of the start state:

Definition 27. *The* effect distribution *of option o is the distribution over states that an agent may find itself in after executing o from anywhere: Effect$(o) = P(s'|o)$. The* effect symbol *is a distributional symbol naming the effect distribution.*

The effect distribution is obtained by integrating out the start state from the probabilistic image operator. If we could use it in place of the image operator, an abstract representation using only the precondition and effect symbols for each option would be sufficient for probabilistic planning.

As before, there are two cases in which this substitution is reasonable. In the *strong probabilistic subgoal condition*, the distribution returned by the image operator and the effect distribution are equal:

$$\text{Image}(X, o) = \text{Effect}(o),$$

for all reachable distributions over states (i.e., distributional symbols) X. This models the case where a feedback controller or learned policy guides the agent to a specific target distribution (with some completion noise) before completing execution. In this case the distribution over outcome states is statistically independent of the start distribution—a stronger condition than in the set-based case, which merely requires the two distributions have the same support (rather than the same density).

Alternatively, the *weak probabilistic subgoal condition* is that, for any starting distribution Z_0 and option o_j:

$$\int \text{Image}(Z_0, o)(s) P(s \in I_{o_j}) ds = \int \text{Effect}(o)(s) P(s \in I_{o_j}) ds.$$

Here the two distributions differ, but the probability of being able to execute any follow-on option o_j is the same. This models the case where the termination distribution of an option *does* depend on its start state, but that difference is immaterial to the probability of subsequently successfully executing any other option; in other words, the effect distribution is a sufficient statistic for predicting the probability with which any successive option can be executed, and is therefore suitable as a Markov state descriptor in our abstract model.

An agent with options that all obey either the weak or strong subgoal property can construct a very simple abstract data structure that suffices for probabilistic optimal planning. Given a collection of options and their effects distributions, define an abstract plan graph G, describing an abstract Markov decision process, as follows. Each option o_i has corresponding vertex v_i, with edges e_{ij} between v_i and v_j with reward $J(o_j, \text{Effect}(o_i))$ and probability of success $\int_S P(s \in I_{o_j}) \text{Effect}(o_i)(s)\, \mathrm{d}s$. The edges for the synthetic goal option and start distribution symbol have no incoming or outgoing edges, respectively. After computing G the agent can discard its grounding distributions and evaluate the reward and probability of a plan succeeding by adding the rewards and multiplying the probabilities along the corresponding path in the graph.

In the case where the option modifies some state variables and leave others unchanged, we can obtain an expression analogous to the image computation in the deterministic planning case by defining a mask listing the state variables modified by the option as before, and writing the probabilistic image operator as:

$$\text{Image}(X, o) = \text{Effect}(o) X_r,$$

where Effect(o) is a distribution over only the variables in the mask, $X_r = \int X(s) \mathrm{d}b$, and $b = \text{mask}(o)$. Here, X_r is the distribution obtained after integrating the variables in o's mask out of X.[6] This process is depicted in Figure 19.

The multiplication of these two distributions corresponds to assuming that the distributions over state variables inside and outside the mask are statistically independent of each other after execution. This models the assumption that the distribution over the values of state variables in the mask after execution is independent of start state. This assumption is reasonable when the equivalent strong abstract probabilistic subgoal condition holds (Image(X, o) = Effect$(o) X_r$) or similarly when the equivalent weak abstract probabilistic subgoal condition holds.

In the deterministic case there could be only one mask and only one distribution. However, in the probabilistic case, an abstract subgoal option may be best modeled using multiple *effect outcomes*: executing the option may lead to one of a number of distinct state distributions, each with an associated probability of occurrence. The effect outcome distributions can be modeled as a single mixture distribution if they all have the same mask; if not, they must be modeled individually. In such cases an option o_i is characterized by a

6. This corresponds to the Remainder operator in section 3.1.2.

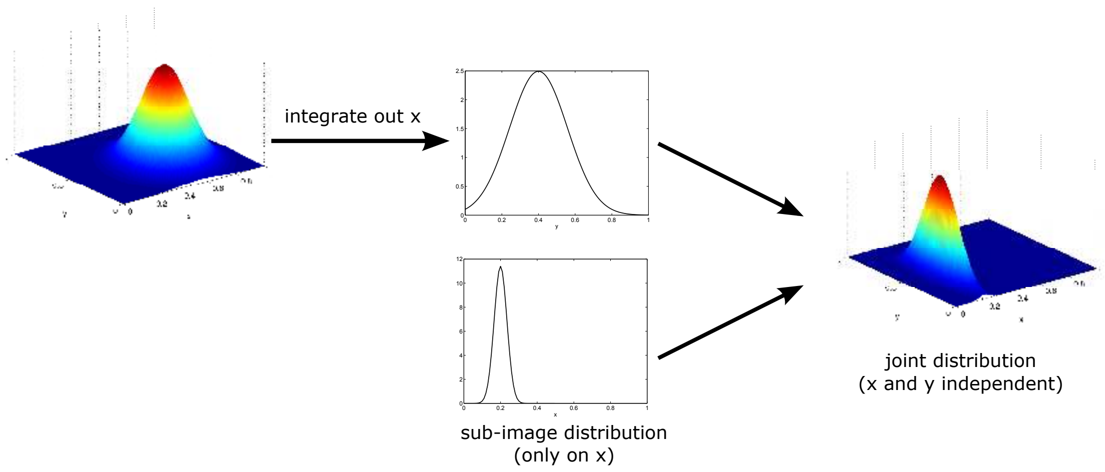


Figure 19: Computing the image of an abstract subgoal option in a two-dimensional state space. The option does not change y, so we integrate out the dimension that does change, x, to obtain X_r, a distribution over only y. X_r is multiplied by the sub-image distribution over x to obtain the image distribution. Note that x and y become statistically independent in the computed image.

single precondition symbol $\text{Pre}(o_i)$ (naming its initiation classifier I_{o_i}) and a set of effect outcomes symbols $\text{Eff}^j(o_i)$ (naming effect distributions), each with an associated mask and probability ρ_j of occurrence.

As before, we may generalize these two types of options to partitioned subgoal or abstract subgoal options. Identifying such partitions is a major challenge when learning a probabilistic symbolic representation in practice. Fortunately, partitioning can be interpreted statistically. If neither abstract subgoal condition holds, we must find a partition so that one of them does for each individual partition. To match the strong subgoal condition, for example, we must find a partitioning C such that:

$$P(s'|o, s) = P(s'|o, C(s)).$$

In other words, if s' is not independent of s, then we find a partition C such that it becomes independent conditioned on the value of C.

An abstract representation that supports planning using options obeying at least the weak abstract subgoal property must therefore contain at least a probabilistic precondition and effect symbol for each option, plus the symbols required to compute all reachable instances of the remainder distribution X_r. We now construct such a representation.

4.2 Constructing a STRIPS-Like PPDDL Domain Description

As before, we will construct a STRIPS-like factored symbolic representation. Our target in this case is PPDDL, the *Probabilistic Planning and Domain Definition Language* (Younes & Littman, 2004), a probabilistic extension of PDDL often used as input to off-the-shelf planners.

In PPDDL, we still require a set of propositional symbols $\mathcal{P} = \{p_1, ..., p_n\}$ and a set of actions $\mathcal{A} = \{\alpha_1, ..., \alpha_m\}$; a state $\mathcal{P}_t$ is still obtained by assigning a truth value $\mathcal{P}_t(i)$ to every $p_i \in \mathcal{P}$, so we may still consider abstract states to be binary vectors. However, each operator α_i now describes multiple possible action outcomes, along with the probability of each occurring:

$$\alpha_i = \left(\text{precond}_i, \left\{(\rho_1, \text{effect}^+_{i1}, \text{effect}^-_{i1}), ..., (\rho_k, \text{effect}^+_{ik}, \text{effect}^-_{ik})\right\}\right),$$

where each $\rho_j \in [0, 1]$ is an outcome probability such that $\sum_{j=1}^k \rho_j = 1$, and effect$^+_{ij}$ and effect$^-_{ij}$ are the positive (propositions set to be true) and negative (propositions set to be false) effects of outcome j occurring, respectively.

We must also choose a new grounding scheme that specifies the semantics of our symbolic representation in terms of probability distributions rather than sets. We choose to use distributional symbols so that each "propositional" symbol refers to a distribution over states. Thus, each proposition p_i has associated grounding probability distribution $\mathcal{G}(p_i)$. We then define the grounding distribution of an abstract state as the multiplication of the grounding distributions of the propositions set to the true in that state:

$$\mathcal{G}(\mathcal{P}_t) = \Pi_{i \in I} \mathcal{G}(p_i), \; I = \{i | \mathcal{P}_t(i) = 1\}.$$

In the set-based case, a high-level state grounded out to a a set of low-level states, and generalizing that set to a distribution may at first seem unnatural; it is obviously meaningful to treat a set of states as equivalent in order to ignore the differences between them, but it is less obvious what purpose placing a distribution over those states serves. However, we take the view that the key mechanism of planning is projecting the agent's state forward in time; a high-level state can be interpreted as representing the distribution of states in which the agent may find itself in the future. The abstract state $\mathcal{P}_t$ can therefore be interpreted as representing the distribution of states in which the agent expects to find itself at time t.

Given this grounding scheme, we proceed broadly as before. We first identify an appropriate set of factors using each option mask, exactly as in section 3.2.2, making use of the functions modifies(s_i) (the list of options that modify a low-level state variable), factors(o_i) (the list of factors affected by executing o_i), factors(A) and factors(σ_A) (the list of factors that probability distribution A or symbol σ_A is defined over), and options(f_i) (the set of options whose execution modifies factor f_i).

Constructing the symbol set requires the redefinition of the notion of independence used in section 3.2.3 to determine whether or not an effect distribution can be broken into independent parts. The corresponding definition for a probability distribution is:

Definition 28. *Factor f_s is independent in effect distribution Effect(o_i) iff:*

$$\textit{Effect}(o_i) = \left[\int \text{Effect}(o_i) \text{d}f_s\right] \times \left[\int \text{Effect}(o_i) \text{d}\bar{f}_s\right],$$

where $\bar{f}_s = \textit{factors}(o_i) \setminus f_s$.

Here, f_s is independent in Effect(o_i) if the two sets of random variables f_s and $\bar{f}_s$ are statistically independent in the joint distribution Effect(o_i). When that is the case, Effect(o_i) can be broken into independent factors with separate propositional symbols for each. Let Effect$_r(o_i)$ denote the effect distribution that remains after integrating out all independent factors from Effect(o_i), and factors$_r(o_i)$ denote the remaining factors. Our method requires a separate propositional symbol for Effect$_r(o_i)$ with each possible subset of factors$_r(o_i)$ integrated out. The vocabulary $\mathcal{P}$ thus contains the following symbols (with approximate duplicates merged):

1. For each option o_i and factor f_s independent in Effect(o_i), create a propositional symbol with grounding distribution $\int \text{Effect}(o_i)\text{d}\bar{f}_s$.
2. For each set of factors $f_r \subseteq \text{factors}_r(o_i)$, create a propositional symbol with grounding distribution $\int \text{Effect}_r(o_i)\text{d}f_r$.

Given that it can be executed, each option o_i with possible effect outcome Effect$^j(o_i)$ results in the following effects:

1. All propositional symbols with grounding distributions for each factor independent in Effect$^j(o_i)$, and an additional proposition with grounding distribution Effect$^j_r(o_i)$ if necessary, are set to true.
2. All propositional symbols (except the above) σ_j such that factors$(\sigma_j) \subseteq$ factors(o_i) and $\int \mathcal{G}(\sigma_j)(s)P(s \in I_{o_i})\text{d}s > 0$, are set to false.
3. All currently true propositional symbols σ_j where $f_{ij} = \text{factors}(\sigma_j) \cap \text{factors}(o_i) \neq \emptyset$ but factors$(\sigma_j) \not\subseteq$ factors(o_i), and $\int_S \mathcal{G}(\sigma_j)(s)P(s \in I_{i_o})\text{d}s > 0$, are set to false. For each such σ_j, the predicate with grounding distribution $\int \mathcal{G}(\sigma_j)\text{d}f_{ij}$ is set to true.

Each such potential outcome is listed with the probability ρ_j of it occurring.

This computation is analogous to the effects computation in Theorem 3, computing Image(o_i, X) = Effect$(o_i)X_r$, where $X_r = \int X \,\text{d}b$ (integrating out b, the variables in o_i's mask). The first effect in the above list corresponds to Effect(o_i), and the remaining two types of effect model X_r. The second type of effect removes predicates defined entirely using variables within mask(o_i), whose distributions are completely overwritten. The third models the side-effects of the option, where an existing distribution has the variables in mask(o_i) integrated out. The proof that this image computation is correct—more precisely, that it is sound for options with the strong abstract subgoal property and suitable for those with the weak abstract subgoal property—closely follows the one given in section 3.2.4, and we omit it here.

We face one additional complication in the probabilistic setting. Given high-level state $\mathcal{P}_t$ and option o, the agent can compute:

1. A grounding distribution $G = \mathcal{G}(\mathcal{P}_t)$.
2. The probability that o can be executed from $\mathcal{P}_t$: $\rho = \int P(s \in I_o)G(s)\text{d}s$.
3. The abstract effects of executing o (given above), *given that it can be executed.*

This creates a difficulty. The agent can compute the probability, ρ, that o can be executed from $\mathcal{P}_t$, and the positive and negative effects for that outcome. But that is a probabilistic *precondition*, and PPDDL only allows for probabilistic *outcomes*. This does not occur in the deterministic case, because the subset relationship is always either true or not.

The appropriate way to handle this mismatch depends on our aim. If we wish to determine the probability that a given plan reaches a goal state, we can model the failure to execute an option as a null operation that leaves the state unchanged. During execution, the agent may be unable to execute o_t, but may nevertheless be able to skip to executing o_{t+1} and progress to the goal. In that case we could simply output a null effect with probability $(1-\rho)$.[7] However, the strictly correct way to proceed—which will be critical if we wish to go further and evaluate conditional plans[8]—is to determine the state distribution *conditioned on the action not being executable.*

Conditional planning would require us to compute $P(s|s \notin I_{o_i}) \propto (1 - P(s \in I_{o_i}))P(s)$. We leave this for future work, as our stated aim is evaluating the probability of a single, straight-line plan succeeding, defined as every option in the plan being executable. Consequently, when constructing a PPDDL representation in the following section we add a virtual proposition named `notfailed`, which is set to true during initialization and is a precondition for every operator. We then add an effect of $\neg$`notfailed` to each action with probability $(1 - \rho)$ (we may omit the operator entirely if ρ is very close to 0).

4.3 Learning a Probabilistic Symbolic Representation for the Treasure Game

We now use our framework to design an agent that autonomously learns an abstract symbolic representation of a computer-game like domain. The Treasure Game features an agent in a 2D, 528×528 pixel world whose task is to obtain treasure and return to its starting position on a ladder at the top left of the screen (see Figure 20). The agent's path may be blocked by closed doors. Flipping the direction of either of the two handles (on the right middle and top left of the screen) switches the status of the two doors on the top right of the screen (only one can be open—shown as an open doorframe—at a time, and flipping one switch also flips the other). The agent must obtain the key (two thirds of the way up the screen, on the left) and use it in the lock (bottom left) to open the door on the bottom right of the screen, allowing it to reach the treasure.

The primitive actions available to the agent are moving up, down, left, and right, jumping, interacting, and a no-op. Left and right movement is available when the agent's way is not directly blocked by a closed door or a wall (shown in dark gray), while up and down are only available when the agent is on or above, or on or below, respectively, a ladder. These actions move the agent between 2 and 4 pixels (chosen uniformly at random)—note that though the game screen is drawn using large discrete image tiles, the agent moves at the pixel level. The interact action is available when the agent is standing in front of a handle (flipping the handle's position from right to left, or vice versa, with probability 0.8), or when it possesses the key and is standing in front of the lock (whereupon the agent loses the key). The no-op action lets the game dynamics continue for one time step, and is useful after a jump action, or when the agent is falling due to moving over an empty space.

7. We will do this in Section 5.
8. A conditional plan contains structures of the form `if we cannot execute` o_i`, switch to alternate plan` p_2. This is distinct from a *policy*, which specifies an action to execute from every state, and which our abstract representation supports via value-function based planning.

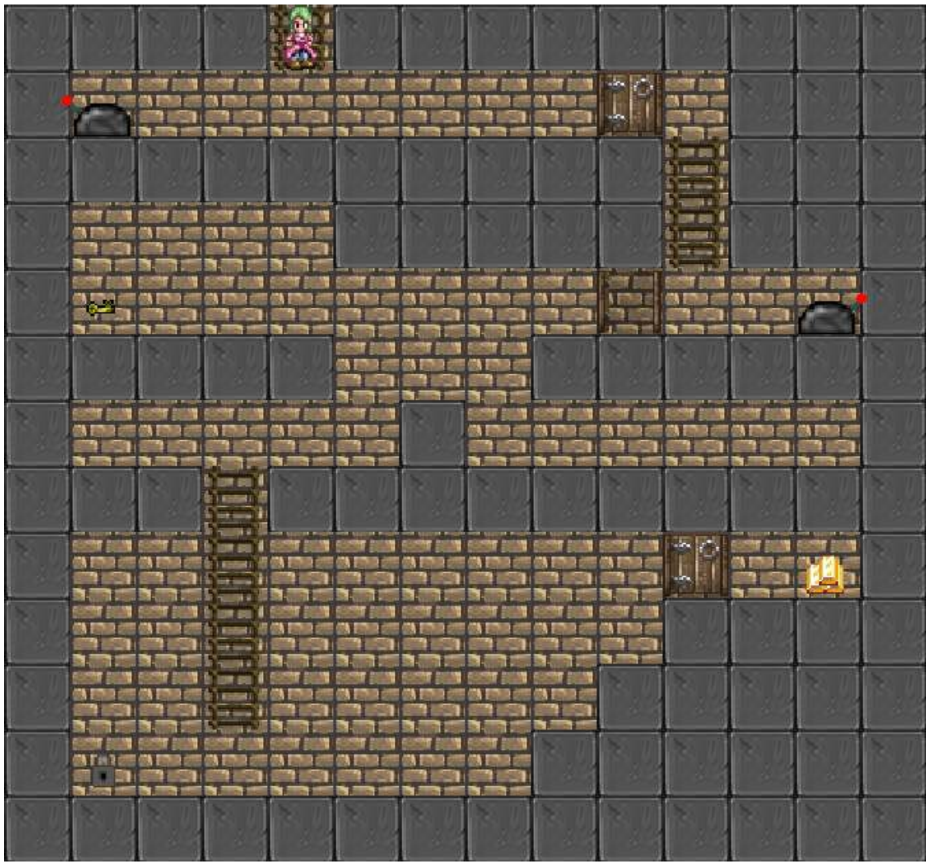


Figure 20: The Treasure Game domain. Sprites courtesy of Hyptosis and `opengameart.org`, Creative Commons license CC-BY 3.0. Although the game screen is drawn using large discrete image tiles, sprite movement is at the pixel level.

Each action has a reward of -1, except for the jump action, which receives a reward of -5. Returning to the top ladder with the treasure ends the episode.

The low-level state space is 9-dimensional, featuring the x and y positions of the agent, key, and treasure, the angles of the two handles, and the state of the lock. When the agent possesses the key or the treasure, it is located in the lower-right corner of the screen.

The agent has access to the following 9 high-level options, implemented using simple control loops:

- `go-left` and `go-right`, which move the agent continuously left or right, respectively, until it reaches a wall, an edge, an object with which it can interact (a handle or the lock), or a ladder. These options can only be executed when they would succeed (i.e., they cannot be executed when the agent's path is blocked by a closed door, it is up against a wall, or it would fall off an edge).

- `up-ladder` and `down-ladder`, which cause the agent to ascend or descend a ladder, respectively, and can only be executed when the agent is in a position to do so.
- `down-left` and `down-right`, which cause the agent to execute a controlled fall off an edge onto the nearest solid cell on its left or right, respectively. These options are only available when the agent is standing on an appropriate edge.
- `jump-left` and `jump-right`, which cause the agent to jump and move left, or right, respectively, for about 48 pixels. These options are only available to the agent when the area above its head, and above its head and to the left and right, are clear. These options are usually able to allow the agent to reach the floor at a level above it and to one side, but this sometimes fails due to the stochasticity in the amount of horizontal movement per step.
- `interact`, which executes the primitive interaction action, and is only available when such an interaction is possible.

All of the options have stochastic termination conditions which, when combined with the stochasticity present in the primitive actions, result in outcome variance ranging from a few pixels (for the `go-left` and `go-right` options) to a much larger amount (e.g., in the case where the `jump-left` option can miss the ledge, causing the agent to fall). The shortest plan that solves the Treasure Domain with non-zero probability consists of 42 high-level actions, which executes approximately 3800 low-level actions.

4.3.1 Learning a Symbolic Representation From Experience

To obtain training data, we executed 100 randomly selected options sequentially, gathering one set of data that recorded whether each option could run at states observed before and after option execution, and another that recorded the transition data $x_i = (s_i, o_i, r_i, s'_i)$ for each executed option. This procedure was repeated 40 times, for a total of 4000 option executions.

First, the options must be partitioned so that the abstract subgoal property approximately holds. A complete and principled solution to this problem requires us to partition the start states into classes C such that $P(s'|s, c) = P(s'|c)$ (the distribution of end states s' is independent of the distribution of start states s, conditioned on the classification $c \in C$). However, for our purposes a much simpler procedure sufficed. For each option o:

1. The mask m_i was computed for each sample transition x_i, and the data was partitioned by mask.
2. For each mask m_j, the effect states $s'_i\,[m_j]$ were clustered and each cluster was assigned to its own partition. The data was now partitioned into distinct effect distributions, but may have been over-partitioned because distinct effects may occur from the same start state partition.
3. For each pair of partitions, the agent determined whether their start-state samples s_i overlapped substantially by clustering the combined start state samples s_i from each partition, and determining whether each resulting cluster contained data from both partitions. If so, the common data was merged into a single partition.

4. When merging, an outcome was created for each effect cluster (which could be distinct due to clustering, or due to a different mask) and assigned an outcome probability based on the fraction of the samples assigned to it.

Clustering was performed using the DBSCAN algorithm (Ester, Kriegel, Sander, & Xu, 1996) in the `scikit-learn` toolkit (Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot, & Duchesnay, 2011), with parameters `min_samples = 5` and $\epsilon = 0.4/14$ (for partitioning the effects) or $\epsilon = 0.8/14$ (for merging the start states). This resulted in 39 partitioned options. We now illustrate a few example partitioned option models.

Figure 21 shows a visualization of the 6th partition of the `go-left` option, capturing the conditions under which the agent may move through one of the doors. Fuzzy or transparent figures indicate variance in the distribution. The initiation set visualization shows that one door *must* be open and that the agent has to be roughly underneath the rightmost ladder, but the gold (for example) could be in either possible location. When these conditions hold, the resulting effect set shows that the agent's x position changes so that it is past the door, and right before a ledge.

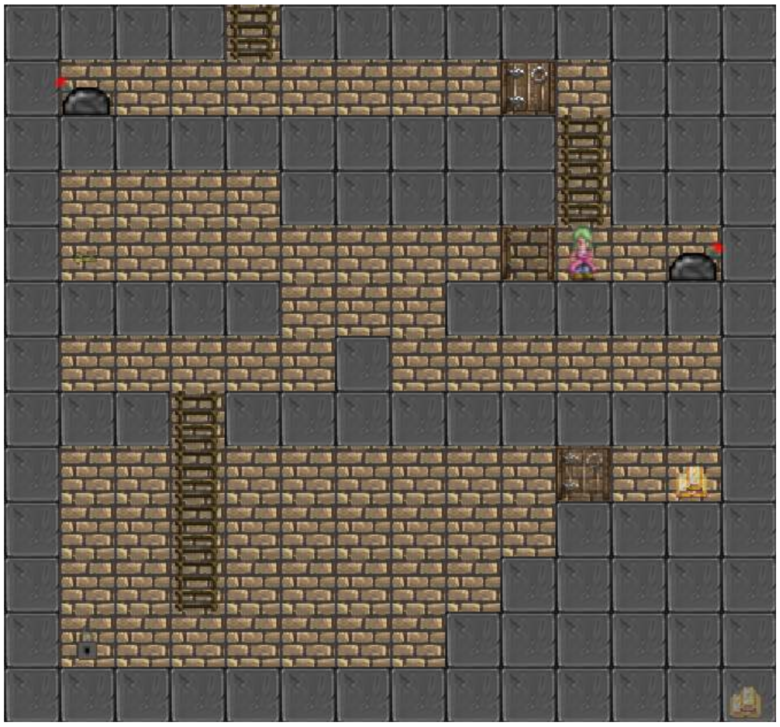
Figure 22 shows the two partitions of the `jump-left` option (we have zoomed in on the relevant area to show the visualization in more detail). The first partition shows that the agent can jump left when it is standing just to the left of the block in the centre of the domain, and that doing so leaves it standing atop that block. The second partition shows that it can also jump left from atop the central block, which results in a probabilistic outcome: the jump succeeds, leaving the agent atop a nearby ledge, about half the time; otherwise it finds itself on a nearby floor.

Figure 23 shows the first partition from the `interact` option, expressing the transition that occurs when an agent interacts with a handle when the lower door is already open. It models the 20% probability that the handle will move slightly but the door will not close, and the 80% probability that the handle will move all the way over (which causes the other handle to move too), closing the door. Another similar partition models the effect when the handle starts on the other side, and also for each similar case when the agent is standing in front of the other handle. The final partition of the `interact` option models the case when the agent is standing in front of the lock and is holding the key.

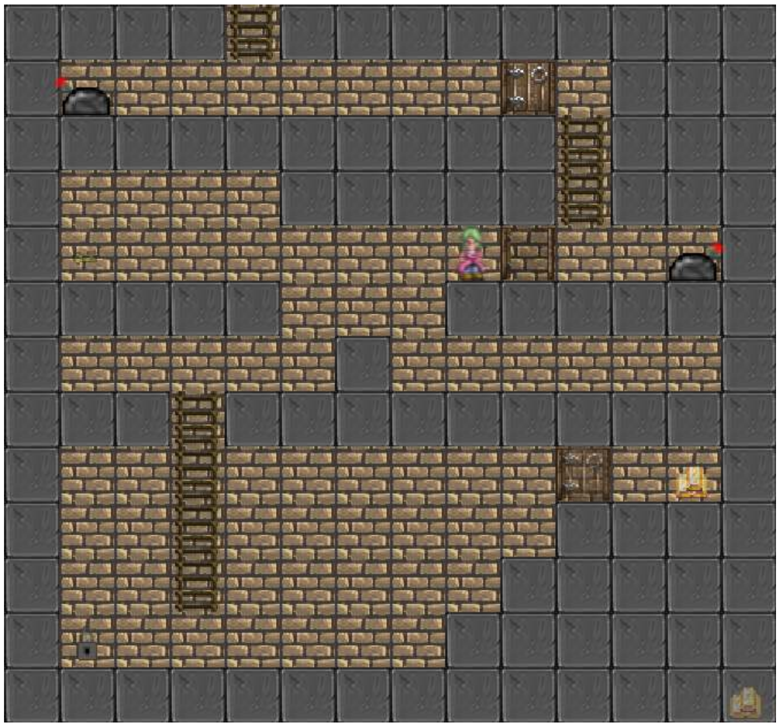
Given these partitions, the agent created learned probabilistic symbols for each of the partitioned options with groundings determined as follows:

1. A *precondition mask*, which listed the low-level state variables that the precondition classifier depended on, was computed using a simple feature selection procedure, as follows. A 3-fold cross-validation score was computed using the support vector machine (Cortes & Vapnik, 1995) classifier implementation in `scikit-learn`, with an RBF kernel, automatic class reweighting, and parameters selected by a grid search with 3-fold cross-validation. We then tested whether leaving out each state variable independently damaged the score, keeping only variables that did. Finally, we added each state variable back when doing so improved the score.

2. A support vector machine was used as a probabilistic precondition classifier, using states assigned to that partition as positive examples, and all other states (including



(a)



(b)

Figure 21: A visualization of the 6th partition of the `go-left` option, capturing the conditions under which the agent may move through one of the doors. The image is averaged over the initiation classifier (a) and the effect distributions (b).

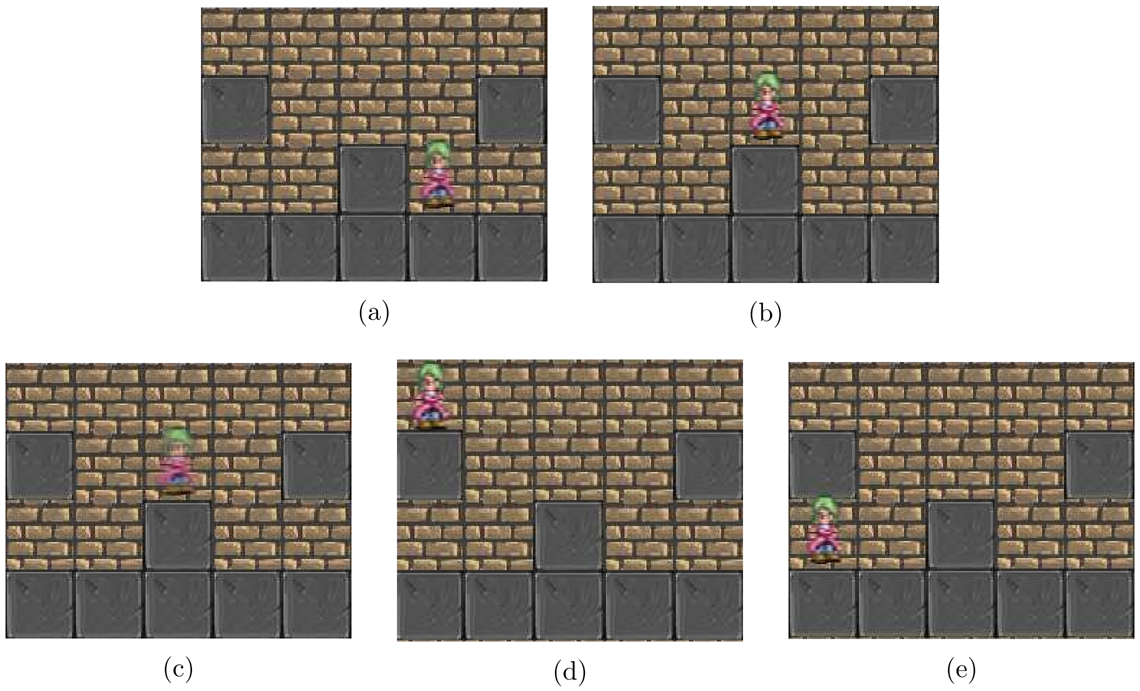


Figure 22: A visualization of both partitions of the `jump-left` option. (a) The first probabilistic precondition shows the agent standing next to a block. (b) Jumping left leaves the agent standing on top of the block. (c) The second probabilistic precondition has the agent standing on the block. Jumping left leaves the agent either standing atop to ledge to its left (d, with probability 0.53) or, having missed, on the floor (e, with probability 0.47).

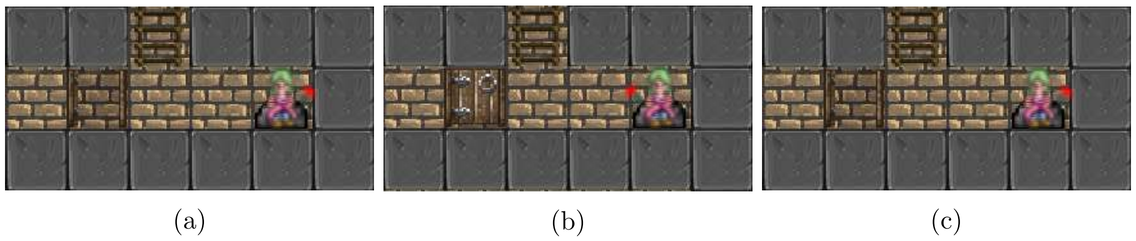


Figure 23: A visualization of the first partition (of five) of the `interact` option. (a) The probabilistic precondition shows the agent in front of a handle, the handle set to the right, and the door open. Interaction results in an effect distribution where the handle is pushed to the other side and the door is closed (b, with probability 0.795), or where the handle moves only slightly and the door remains open (c, with probability 0.204). Note that the two effect distributions have different masks—one changes the the angle of only one handle, and the other changes both (the other handle is not shown).

those in which another partition of the same option could be executed) as negative examples. We used the same support vector machine implementation, but this time enabling probabilistic class estimation using Platt scaling (Platt, 1999).

3. Kernel density estimation (Rosenblatt, 1956; Parzen, 1962) was used to model each effect distribution, with a Gaussian kernel and parameters fit using a grid search and 3-fold cross-validation.

4. A reward model was learned using support vector regression (Drucker, Burges, Kaufman, Smola, & Vapnik, 1997), using an RBF kernel and parameters fit using a grid search and 3-fold cross-validation.

These algorithms were used primarily because good implementations were already available in `scikit-learn`, and because as nonparametric methods they avoid assumptions about the shape of decision boundaries and probability distributions.

4.3.2 Constructing and Planning with a PPDDL Representation

The agent identified 7 factors from the resulting partitioned probabilistic symbols. The effect distributions were split into 30 distinct distributional symbols (duplicates were detected by a simple coverage interval and mean similarity test). The factors extracted, and the number of effect symbols defined over each factor, are shown in Table 2.

Factor	**State Variables**	**Symbols**
1	`player.x`	10
2	`player.y`	9
3	`handle1.angle`	2
4	`handle2.angle`	2
5	`key.x, key.y`	3
6	`bolt.locked`	2
7	`goldcoin.x, goldcoin.y`	2

Table 2: Factors identified automatically in the partitioned options extracted from the Treasure Domain, along with the number of probabilistic symbols defined over each.

The agent constructed a PPDDL representation by recursing through possible combinations of symbols that overlapped with each partitioned option's precondition mask. To compute the probability of being able to execute a partitioned option given a set of symbols, the grounding distribution of the precondition—a conjunction of the distributional symbols—was first computed, followed by the probability that a state drawn from the resulting distribution lies within the probabilistic precondition of the partitioned option. Finally, the expected reward of executing the option from the precondition's grounding distribution was computed using the learned reward model. All of these operations were carried out using Monte Carlo sampling ($m = 100$ samples). Rules estimated to be executable with a probability of less than 5%,were discarded, and those with an estimated probability of execution of greater than 95% were upgraded to certainty. This resulted in

345 action rules; an example action rule along with the relevant grounding distributions is given in Figure 24.

Note that many of the resulting rules were for combinations of symbols which are not reachable in the domain, but where our learned preconditions indicate the skill could be executed. For example, consider the option for descending a ladder; one partition captures the act of going down the top-most ladder. Its precondition classifier sees no samples near the very top of the domain, except for samples from the only open cell at that level, which are all positive. All samples with a lower y-coordinate for the player are negative examples. Therefore, the precondition classifier learns that that option can be executed if the player has a high enough y location (the precondition mask in this case still includes the player's x coordinate, but its effect is minor enough that it can be ignored). The agent therefore derived many operators that include preconditions for a high y value but x locations that the player cannot reach, because the cells they land in are filled with a concrete block. Similar extraneous rules occur for x and y combinations that are reachable by the player using low-level actions, but that never occur as a result of option executions. These can be viewed simply as over-generalizations by the agent; were they in states the agent could reach, their effect would shortly be learned away.

Once the PPDDL description had been constructed by the agent, it was free to discard its grounding distributions and plan solely using the completely symbolic PPDDL representation. Table 3 shows the time required to compute a policy for the resulting PPDDL problem, using the off-the-shelf mGPT planner (Bonet & Geffner, 2005) with the built-in `lrtdp` method and min-min relaxation heuristic. All policies were computed in less than one fifth of a second.

Goal	**Min. Depth**	**Time (ms)**
Obtain Key	14	35
Obtain Treasure	26	64
Treasure & Home	42	181

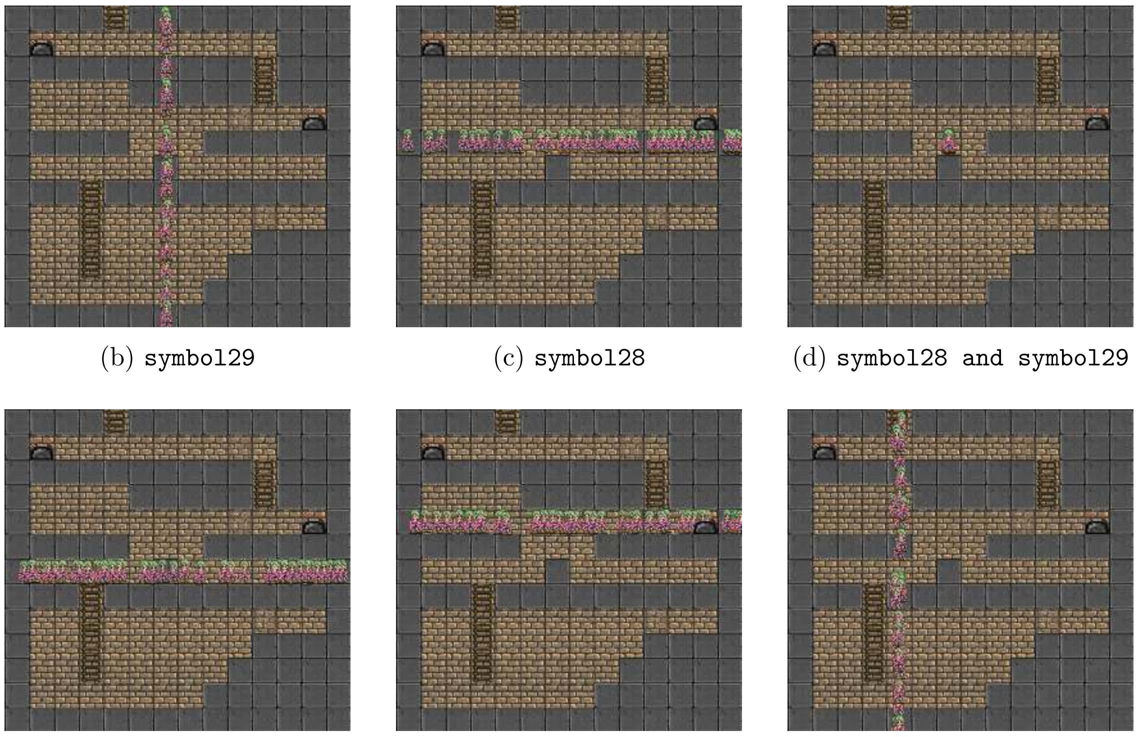
Table 3: The time required and minimum solution depth (in terms of option executions) for a few example Treasure Game planning problems. Results were obtained on an iMac with a 3.2Ghz Intel Core i5 processor and 16GB of RAM.

4.4 Summary

Generalizing the grounding of a symbol from a set of states to a probability distribution provides several advantages: it allows us to evaluate the probability of a plan succeeding, rather than determine whether it will always succeed; it allows us to evaluate the expected reward obtained by a plan; and it can cope with the uncertainty that is a fundamental to learning, thereby allowing us to learn the symbolic representation autonomously from experience.

```
(:action jump_left_option319
 :parameters ()
 :precondition (and (notfailed) (symbol29) (symbol28) )
 :effect (probabilistic
          0.4723 (and (symbol17) (symbol1) (not (symbol28)) (not (symbol29))
                      (decrease (reward) 62.39))
          0.5277 (and (symbol20) (symbol1) (not (symbol28)) (not (symbol29))
                      (decrease (reward) 36.32))
         )
)
```

(a) Generated PDDL Operator



(b) `symbol29` (c) `symbol28` (d) `symbol28 and symbol29`

(e) `symbol17` (f) `symbol20` (g) `symbol1`

Figure 24: (a) The automatically generated PPDDL operator for one partition of the `jump-left` option, together with 50 samples drawn from each symbol's grounding distribution. The precondition distributions are named by `symbol29` and `symbol28` (b and c), which together indicate that the agent should be atop the concrete block in the center of the domain (d). Executing the option results in one of two y coordinate outcomes, named by `symbol17` and `symbol20` (e and f). Both outcomes set the agent's x coordinate according to the distribution named by `symbol1` (g). Note that the two outcomes have different rewards—the failed jump costs more because the agent has to wait until it has fallen past the bottom of the ledge before it can finish moving left.

5. Learning a Symbolic Representation of a Robot Manipulation Task

The preceding section has demonstrated that our framework enables an agent to learn a symbolic representation using only data obtained through interaction with its environment, and thereby compute plans very quickly using an off-the-shelf probabilistic high-level planner. That demonstration took place in a relatively simple computer game environment, but we are motivated primarily by achieving abstract planning on real robots. We now show that our framework can achieve that goal, by describing a robot manipulation system that acquires a symbolic representation directly from low-level sensor outputs—point clouds, map locations, and joint positions—and uses it to rapidly plan to achieve its goals.

5.1 Experimental Task Design

The experiment uses a robot named Anathema Device, an Adept Pioneer Mobile Manipulator. Anathema consists of an Adept Pioneer LX base and a fixed torso upon which is mounted a pair of Kinova Jaco-2 robot arms, and a pan-tilt head with a Kinect-2 RGBD sensor. The base performs mapping, localization, and path planning, while each arm has a 3-finger gripper for manipulation. Computing was performed partially onboard and partially on a workstation connected by a wireless network, via ROS (Quigley, Conley, Gerkey, Faust, Foote, Leibs, Wheeler, & Ng, 2009) and a collection of open-source software packages (Niekum, 2011; Wiedemeyer, 2014; Sucan & Chitta, 2013; Rusu & Cousins, 2011; Beeson & Ames, 2015).

Anathema was placed in a room, shown in Figure 25, that contains a cupboard and a cooler—either of which may contain a green water bottle—and a switch. The switch controlled a bright light inside the cupboard, turning it on (when the switch was turned to the right) or off (when the switch was upright); the light was sufficiently bright to cause perception of objects inside of the cupboard to fail when it was lit. The robot was tasked with having to move the bottle from one container to the other on demand.

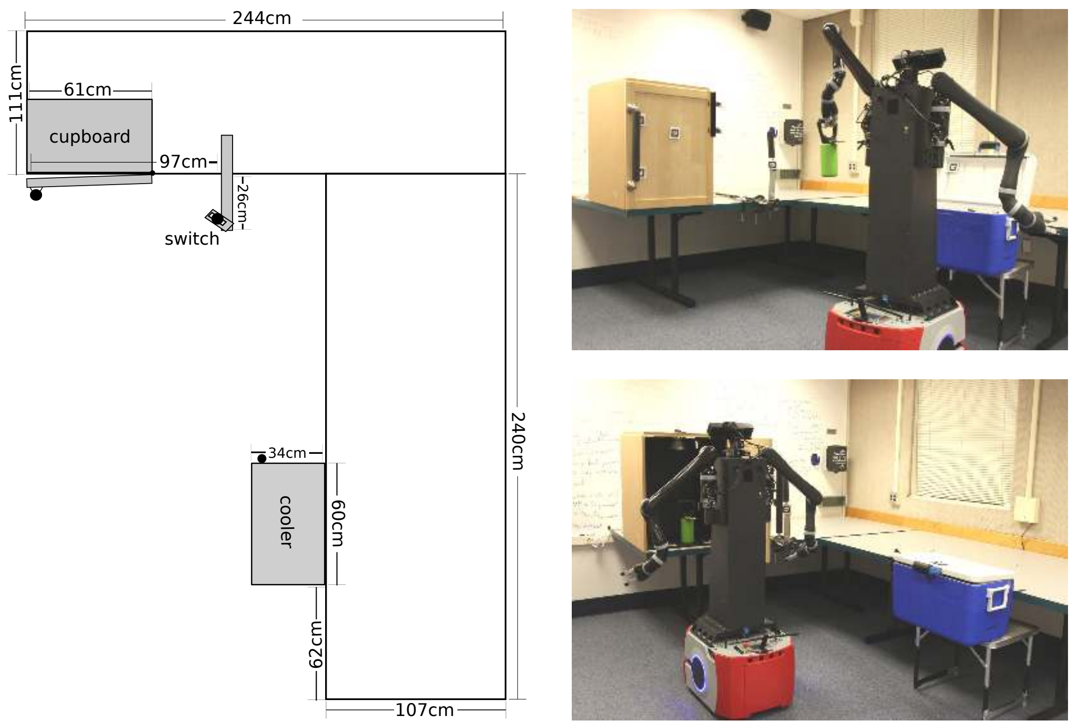
We provided Anathema with a collection of motor skills, some of which are pictured in Figure 26. To simplify the motor skill programming process, we used augmented-reality tags (or ARTags; Kato & Billinghurst, 1999) as a stand-in for skill-specific perception. For example, rather than using computer vision to detect the cooler lid handle, we simply placed an ARTag next to it and used the presence and pose of that tag as indicators for whether or not the handle was visible, and where to grasp it, respectively.[9]

The motor skills supplied to the robot were:

- `navigate-to-cupboard` and `navigate-to-cooler` used a path planner to move the robot to a pre-designated map pose, corresponding to the robot being in front of and facing the relevant object, while avoiding a collision between the robot's arms and the cupboard or cooler. These motor skills were always available for execution, unless the robot was already very close to the target pose.

- The `open-cooler` motor skill used the left arm to lift the cooler lid up by its handle, and then pushed it open with the right arm. It was executable when the ARTag placed next to the handle (and on top of the lid) was visible and within reach. The lid is

9. Note that the tags were not used in the symbol learning process, which used the depth cloud and joint data directly.



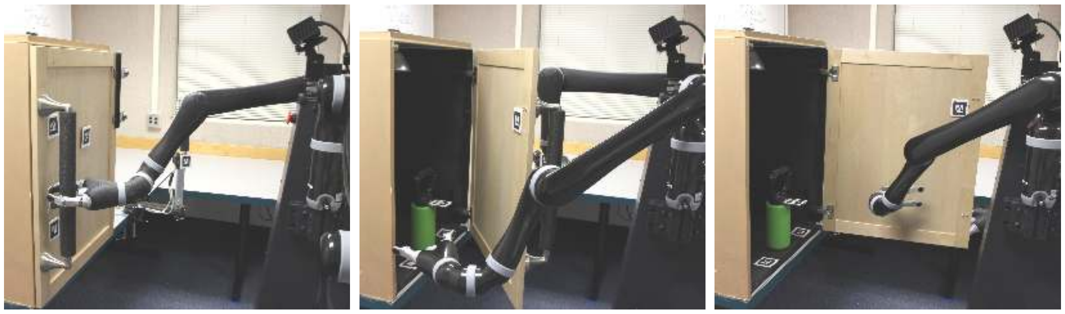
(a) (b)

Figure 25: (a) A schematic of the bottle task. (b) Anathema during task execution. In the top image, Anathema is holding the bottle, the cupboard is closed, the light switch is off, and the cooler is open; in the lower image, the cupboard is open and contains the bottle, the light switch is off, and the cooler is closed.

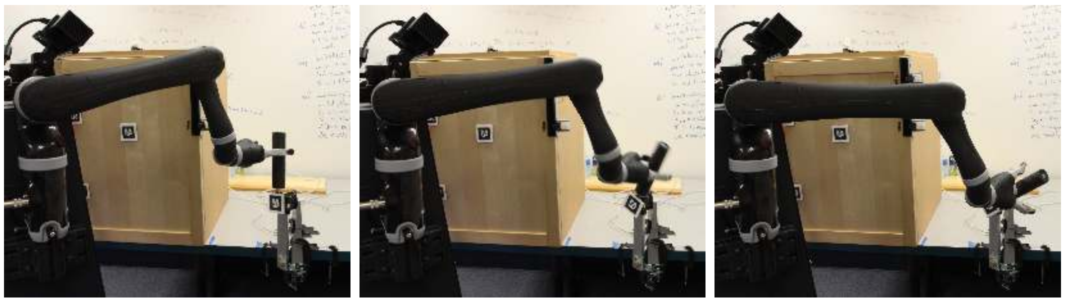
beyond Anathema's reach when open, so the `cooler-close` motor skill pushed down on a specially-attached lever to close it; it was executable when the ARTag placed next to the base of the lever was visible and within range. In both cases, the robot's two grippers had to be be available for use, i.e., not carrying an object.

- The `open-cupboard` motor skill used the right arm to grasp the cupboard handle and partially open the door, before withdrawing to allow the left arm to push the door fully open. It was executable when the ARTags marking the door handle were visible and within reach of the robot, and both grippers were free. The `close-cupboard` motor skill used the right arm to push the door closed, and was executable when the ARTag marking the inside of the door was visible and reachable, and neither gripper contained an object.

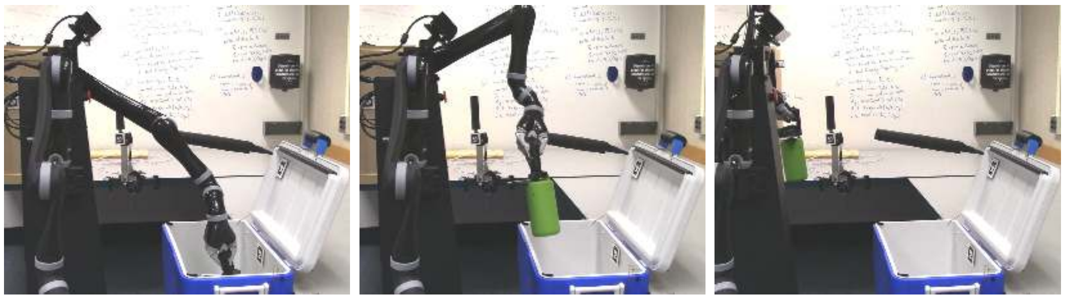
- `switch-on` and `switch-off` used the right arm to turn the switch so that it leaned to the right, or stood upright, respectively. This had the effect of turning the bright light in the cupboard on or off. The two motor skills were executable when the ARTag marking the base of the switch was visible and reachable, and its orientation was not



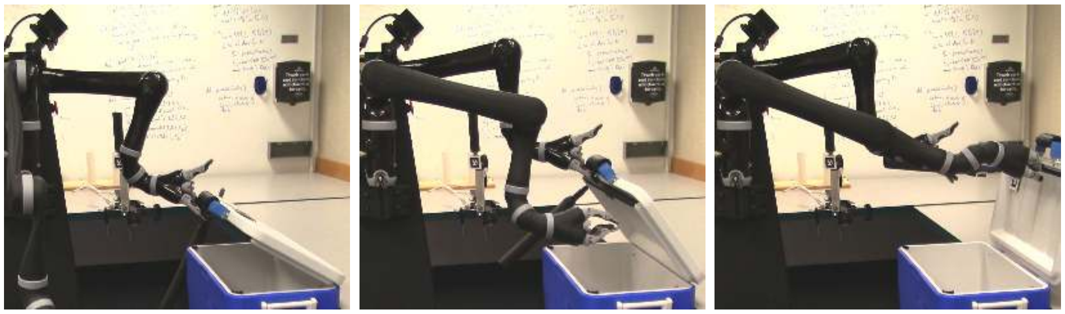
(a)



(b)



(c)



(d)

Figure 26: Anathema executing motor skills: (a) opening the cupboard door, (b) turning the switch, (c) retrieving the bottle from the cooler, and (d) opening the cooler.

already near the relevant target orientation. Since these motor skills used only the right arm, they were executable whenever the right gripper did not carry an object.

- The `pick-up` motor skill used Anathema's left arm to pick up the bottle, and was executable when the ARTag attached to the bottle was visible and reachable. Grasping was performed by repeatedly generating candidate grasp plans using the left arm until an executable one was found. After the gripper reached the pre-grasp pose it was controlled directly to the grasp pose, closed, withdrawn to the pre-grasp pose, and then finally withdrawn to a holding position approximately 30cm to the left of the shoulder.
- The `put-down` motor controller used a procedure similar to grasp generation (but opening the gripper at the target pose, rather than closing it) to place the bottle near a location marked by a surface ARTag. Such tags were placed on the inside of the cooler and the cupboard, and the motor skill was executable when one of them was visible, and when the bottle was present in the left gripper.

The task and motor controllers result in constraints over when the motor skills could be executed. These were not explicitly coded in the motor skills, but are instead a consequence of the physical layout of the task or the capabilities of the robot:

- The robot cannot open or close either container when it is holding the bottle.
- The robot must be located in front of a container to open or close it.
- The robot must be located in front of an open container to retrieve the bottle from it, or place the bottle in it.
- An open cupboard door blocks access to the switch.
- The light inside the cupboard is very bright when lit, causing the perception system to fail to perceive the ARTags inside the cupboard. The bottle therefore cannot be placed in or retrieved from the cupboard unless the light is off.

While these properties can be easily expressed abstractly, it is not at all obvious how to represent them in terms of the sensorimotor data available to the robot. A major advantage of a learned abstract representation is that it can concisely capture these relationships between the robot's sensorimotor space, its motor skills, and the dynamics of the task.

Perception (apart from proprioception) was performed using Anathema's Kinect-2 sensor. Depth images (960x540 pixels) capturing the scene directly in front of the robot were obtained before and after executing each motor skill. A single view was used when the robot was in front of the cooler, but clouds from two different views were captured, aligned using ICP (Besl & McKay, 1992), and merged when it was in front of the cupboard, in order to capture both the switch and the cupboard door or interior. The depth clouds at each location were all aligned over time (again using ICP, and based on the visibility of location ARTags) to compensate for map localization errors and variance in the outcome of the navigation skills.

Our framework is based on a Markov assumption, and so cannot yet reason about latent variables. Consequently, the robot must have immediate access to all of the relevant details

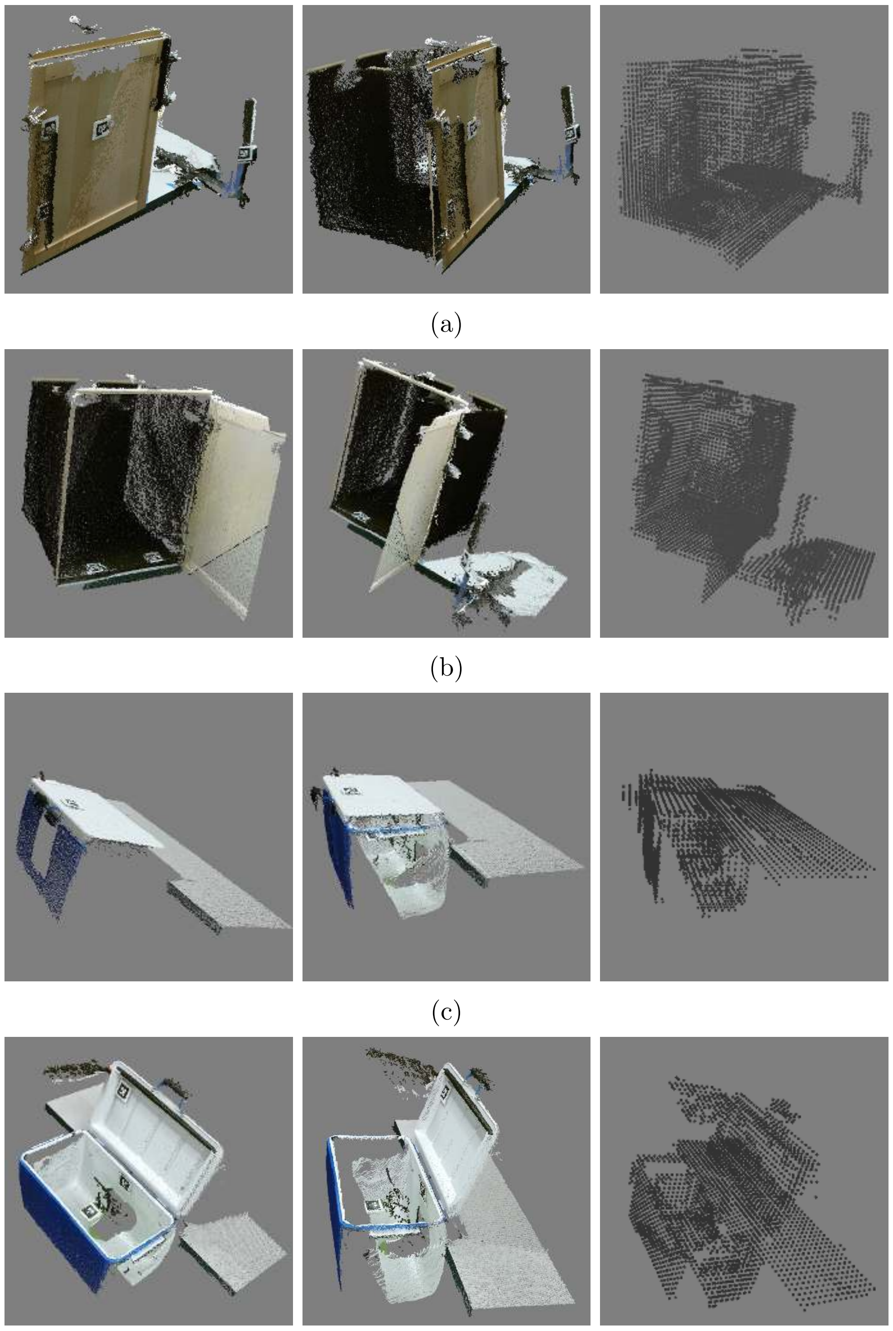


Figure 27: Example point clouds observed when facing (a) the cupboard while closed, (b) the cupboard while open, (c) the cooler while closed, and (d) the cooler while open. Each example is shown (from left to right) as observed, with unobserved pixels inserted to deal with occlusion, and as an occupancy grid.

of the environment. However, in many cases (e.g., when the cupboard door was closed, occluding the bottle inside the cupboard) this assumption did not hold. We therefore implemented a very simple form of perceptual memory: the depth clouds were "completed" by saving and copying over occluded regions from the last cloud in which they were visible. The robot also assumed that the location it was not occupying (i.e., the cooler region when it was at the cupboard, and vice versa) remained unchanged while it was not there. Finally, the depth clouds observed at each location were converted to occupancy grids with a voxel size of 2.5cm^3. This perceptual processing pipeline is depicted in Figure 27.

5.2 Learning a Symbolic Representation

We now describe the procedure used to learn an abstract symbolic representation of the bottle task. As before, we gathered two types of data: option initiation set data of the form $E_i = (s_i, I_i)$, where I_i is a vector representing whether each motor skill was executable in state s_i, and transition data $T_i = (s_i, o_i, r_i, s'_i)$. These were obtained from 167 motor skill executions over several sessions. The actions were selected by hand, rather than randomly as in previous sections, in order to obtain even coverage; our later work (Andersen & Konidaris, 2017) shows how to automate action selection for coverage. The skill executions resulted in 189 initiation set vectors.[10]

Each state was represented by an occupancy grid representation of the room (covering the cupboard and the cooler), an estimate of the robot's position (x, y, and θ) in the room obtained via onboard SLAM, and the 12 joint angles corresponding to each of the six degrees of freedom on both arms. Since our framework assumes that the input states are vectors, we compressed the occupancy grids to vectors using sparse PCA (Zou, Hastie, & Tibshirani, 2006), which we found was able to produce low-error reconstructions using 10 basis functions for the cupboard area and 5 for the cooler.[11] Each state was therefore represented by a 30-dimensional real-valued vector.

Partitioning and PPDDL generation were completed as in Section 4, with the exception of an iterative feature-selection process necessary due to the large number of variables. First, the initiation set of each motor skill was learned using feature selection, which resulted in a set of relevant variables. Each option mask was computed using only these variables, and partitions were identified and learned. If the resulting partition classifiers were inaccurate, each additional variable was tested for relevance and added back in to the set of relevant variables if it substantially improved the classifier. This process was repeated until the partitions and relevant variables converged (in this case, after 2 iterations). The process output 12 partitions, each of which produced a single operator.

5.2.1 Symbol Visualization

We visualized the distributional symbols used in each operator by sampling their grounding distributions. Since the grounding distributions are abstract, this required us to also

10. The discrepancy between the number of samples of each type was because the transitions were obtained over several sessions, each of which resulted in one extra sample initiation vector.
11. We compressed each area separately. It should be possible to use a single room-wide grid to obtain similar results, but doing so would be very memory-intensive since the resulting grid would be very large (and mostly empty).

generate values for the state variables *not* referred to by the distribution. For the purposes of visualization we split the symbols into groups for map pose, arm pose, and the cupboard and cooler grids. The cooler and cupboard contained several factors that could be set independently, so we drew the variables not referred to by the distribution from a kernel density estimator fit to the marginal distribution. Since the map and robot poses have only one factor each, the remaining variables were highly correlated to the grounding distribution, so we used rejection sampling to draw them from the conditional distribution.

The visualizations were produced by averaging 200 samples for each symbol or symbolic expression. For symbols or expressions referring to variables in the map pose group, we drew a red circle with a black stripe representing the robot's pose and heading for each sample and imposed them over a map obtained from the robot's SLAM software. For the robot pose group, we visualized each sample pose using `rviz`, and averaged the resulting images to obtain a single composite image. Finally, for the occupancy grid groups, we used each sample to reconstruct a complete grid using the component vectors constructed by PCA. Since an occupancy grid is binary, each voxel in a sampled grid was set to 1 if it had a value above 0.5. We produced composite images by shading the grid using the squared distance from a point in front of the scene to aid interpretability, and assigned each voxel an opacity proportional to the fraction of samples in which it was occupied.

5.2.2 Learned Operators

We begin with the two navigation operators, which are shown with their associated symbol visualizations in Figure 28. The operators express the learned knowledge that, to execute `nav_to_cooler`, the robot must be positioned at the cupboard, and that execution will result in the robot no longer being positioned at the cupboard, but instead being positioned in front of the cooler. Symbols `symbol0` and `symbol1` can be interpreted as representing the robot being positioned at the cooler and the cupboard, respectively. Note that, while this interpretation is intuitive, the robot has learned—completely autonomously—both that "at the cupboard" and "at the cooler" are the relevant symbolic abstractions and learned the appropriate grounding distributions for each.

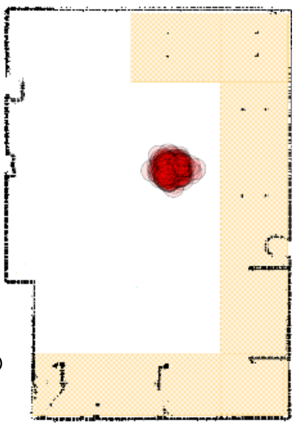
Similarly, to execute `nav_to_cupboard`, the robot should be positioned at the cooler, and will afterward be positioned at the cupboard. The `nav_to_cupboard` operator also includes an erroneous outcome—which it estimates will occur 5% of the time—whereby moving from the cooler to the cupboard causes the light switch to flip from down to up. This part of the operator originates in a compression artifact in one of the transitions, which made it appear that such a change had taken place. Errors of this type are an inevitable result of the noise and complexity of robot learning.

Figure 29 shows the operators for opening and closing the cupboard door, both of which require the robot to be in front of the cupboard (`symbol1`), with a particular arm pose (`symbol3`). That pose is the "stowed" position, to which the arms are moved at the end of every motor skill, except when an object is held in a gripper, in which case it is kept up high to avoid collisions with the environment during navigation. Consequently, `symbol3` indicates that the robot is not holding an object—both arms are free to manipulate the cupboard door. The two operators switch between `symbol4`, which can be interpreted as the cupboard door being closed, and `symbol5`, which can be interpreted as it being open. Notice that

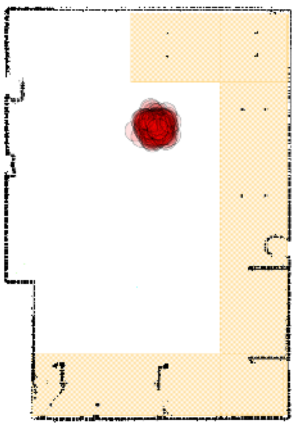
```
(:action nav_to_cooler1
 :parameters ()
 :precondition (and (symbol1))
 :effect       (and (symbol0) (not (symbol1))
                    (decrease (reward) 37.25))
)

(:action nav_to_cupboard1
 :parameters ()
 :precondition (and (symbol0))
 :effect (probabilistic
    0.9583 (and (symbol1) (not (symbol0))
                (decrease (reward) 30.92))
    0.0417 (and (symbol1) (symbol6)
                (not (symbol0)) (not (symbol7))
                (decrease (reward) 37.16))
         )
)
```

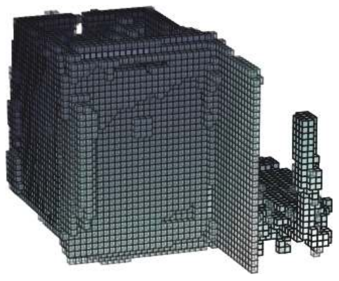
(a) Operator code



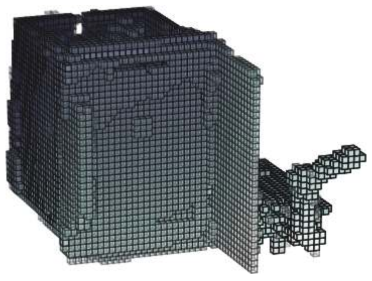
(b) `symbol0`



(c) `symbol1`



(d) `symbol6`



(e) `symbol7`

Figure 28: Learned operators (a) for Anathema's navigation motor skills, along with visualizations of the learned symbols they refer to. The operators switch between `symbol0` (b), which refers to a distribution of robot map poses in front of the cooler, and `symbol1` (c), which refers to a distribution of robot map poses in front of the cupboard. In addition, the `nav_to_cupboard` motor skill has a 5% chance of setting `symbol6` (d) to true and `symbol7` (e) to false. These two symbols refer to the cupboard, and indicate that the switch is up (`symbol6`) or down (`symbol7`). The presence of these two symbols is an artifact of compression error.

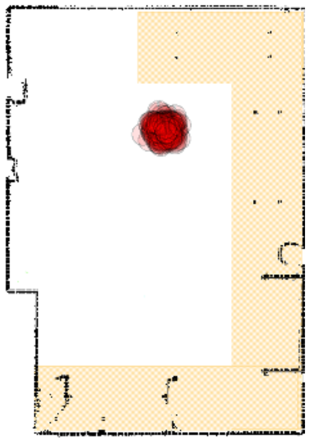
both visualizations are somewhat noisy—`symbol4` has some residual doorframe voxels, and `symbol5` has some residual door panel voxels—and that both symbols are agnostic as to the position of the switch and whether or not an object is present the cupboard (evident in Figure 29e, where the inside of the cupboard is visible).

Figure 30 shows the learned operators for turning the switch to the on (down) and off (up) positions. Both operators require the robot to be positioned in front of the cupboard, but additionally that `symbol4`, which we have previously seen can be interpreted as requiring that the cupboard door be closed, be true. This represents the fact that the switch is not visible to the robot when the cupboard door is open. The operators switch between `symbol6` and `symbol7`, which as we have seen can be interpreted as the switch being up and down, respectively. Figures 30c and 30d visualize the cupboard when both `symbol4` and `symbol6`,

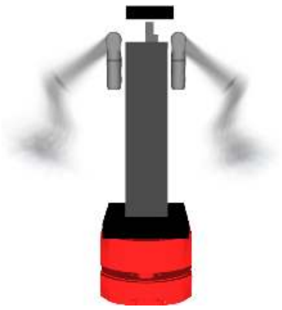
```
(:action cupboard_close1
 :parameters ()
 :precondition (and (symbol1) (symbol3) (symbol5))
 :effect (probabilistic
    0.7647 (and (symbol4) (not (symbol5))
                (decrease (reward) 9.12))
    0.2353 (and (symbol4) (symbol7)
                (not (symbol5)) (not (symbol6))
                (decrease (reward) 9.12))
         )
)

(:action cupboard_open1
 :parameters ()
 :precondition (and (symbol1) (symbol3) (symbol4))
 :effect       (and (symbol5) (not (symbol4))
                    (decrease (reward) 67.44))
)
```

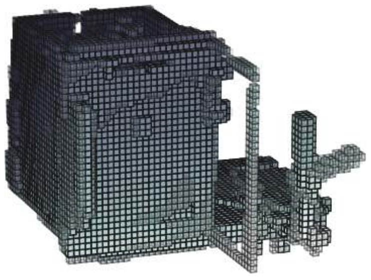
(a) Operator code



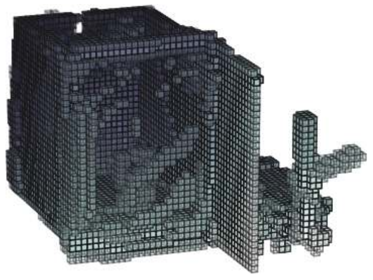
(b) `symbol1`



(c) `symbol3`



(d) `symbol4`



(e) `symbol5`

Figure 29: Learned operators (a) for opening and closing the cupboard, and the learned symbols they refer to. Both operators require Anathema to be positioned in front of the cupboard (`symbol1`, b) with its arms in the stowed position, which indicates that it is not carrying an object (`symbol3`, c). The two operators switch between `symbol4`, which indicates that the cupboard is closed (d), and `symbol5`, indicating that it is open (e).

and both `symbol4` and `symbol7`, are true, respectively. Since `symbol4` is a precondition of both operators, these grounding distributions are the precondition and effects distributions (for the cupboard occupancy grid) of both operators.

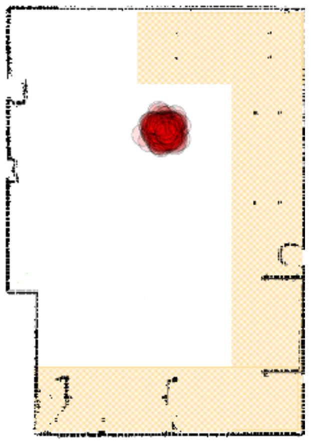
The operators in Figure 30 also both have a probability of failure (14% for `switch_on` and 24% for `switch_off`), which are errors reflecting inaccurate generalizations of either the effect or the precondition distributions for their motor skills. Additionally, if `switch_off` is executable, the learned operator gives it a 9% chance of not changing the state of the switch. This is due to a compression artifact in a single transition, which made it appear that a switch that was down was actually up.

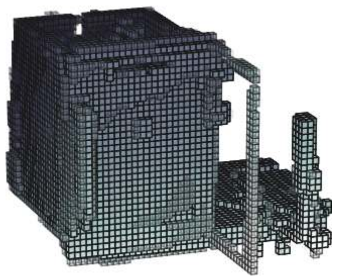
Figure 31 shows the operator and symbol visualizations for opening and closing the cooler. These operators both require the robot to be positioned in front of the cooler (`symbol0`), and not carrying an object (`symbol3`, because opening the cooler, like the cup-

```
(:action switch_on1
 :parameters ()
 :precondition (and (symbol1) (symbol4)
                    (symbol6))
 :effect (probabilistic
    0.1412 (and)
    0.8588 (and (symbol7) (not (symbol6))
                (decrease (reward) 47.40))
         )
)

(:action switch_off1
 :parameters ()
 :precondition (and (symbol1) (symbol4)
                    (symbol7))
 :effect (probabilistic 0.2377 (and)
    0.7623 (probabilistic
      0.9091 (and (symbol6) (not (symbol7))
                  (decrease (reward) 47.80))
      0.0909 (and (decrease (reward) 46.08))
          )
        )
)
```

(a) Operator code


(b) `symbol1`


(c) `(and (symbol4) (symbol6))`

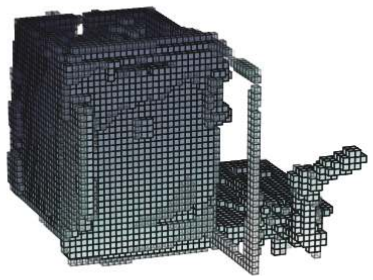

(d) `(and (symbol4) (symbol7))`

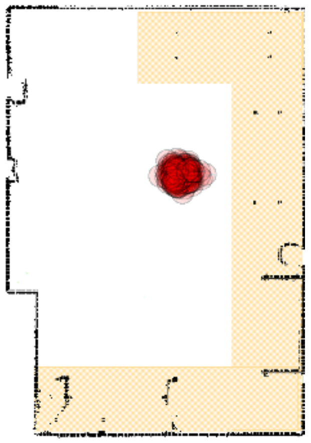
Figure 30: Learned operators (a) for turning the switch. Both operators require Anathema to be in front of the cupboard (`symbol1`). The `switch_on` operator requires both the cupboard door to be closed (`symbol4`) and the switch to be up (`symbol6`); the conjunction of these two symbols is visualized in (c). Executing `switch_on` results in the cupboard door remaining closed and the switch being down (`symbol7`), the conjunction of which is shown in (d). The `switch_off` operator performs similarly, but in reverse. Note that both operators have a probability of failure, and the `switch_off` operator may in addition be executable but (with probability 9%) result in no state change, both of which are errors due to learning and compression artifacts conflating switch states.

board, requires both grippers). The operators switch between symbols indicating that the cooler is open (`symbol8`) and that it is closed (`symbol9`). The `cooler_open` operator has a stochastic outcome, accurately reflecting the possibility (estimated here at 7.6%) that the motor skill may fail to open the cooler. This occurred because the robot first lifts the

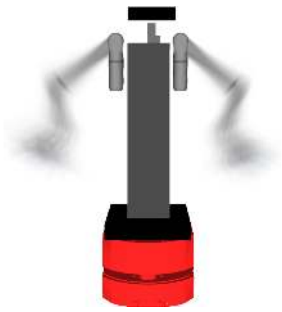
```
(:action cooler_open1
 :parameters ()
 :precondition (and (symbol0) (symbol3)
                    (symbol9))
 :effect (probabilistic
    0.9231 (and (symbol8) (not (symbol9))
                (decrease (reward) 65.40))
    0.0769 (and (decrease (reward) 65.76))
         )
)

(:action cooler_close1
 :parameters ()
 :precondition (and (symbol0) (symbol3)
                    (symbol8))
 :effect (and (symbol9) (not (symbol8))
              (decrease (reward) 45.38))
)
```

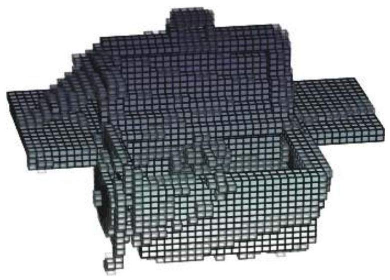
(a) Operator code



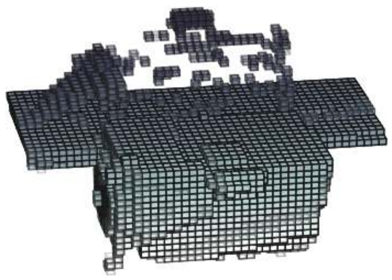
(b) `symbol0`



(c) `symbol3`



(d) `symbol8`



(e) `symbol9`

Figure 31: Learned operators (a) for opening and closing the cooler. Both operators require Anathema to be in front of the cooler (`symbol0`, b) and not carrying an object (`symbol3`, c). The operators switch between `symbol8`, indicating that the cooler is open (d), and `symbol9`, indicating that it is closed (e). The stochastic outcome for `cooler_open` reflects its occasional failure to open the cooler.

cooler lid with the left arm and then pushes it open with the right; in one out of thirteen executions the push was too forceful, and the cooler lid bounced shut again.

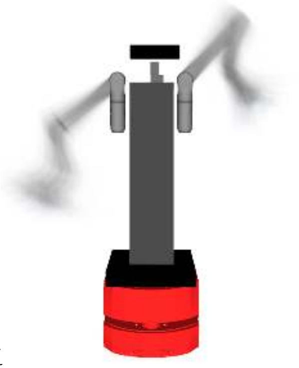
Note that an object may or may not be in the cooler, best observed in `symbol8`'s visualization (Figure 31d). Also note the extra noise voxels most evident in `symbol9`, which are artifacts of compression and generalization, and the smudge at the top left of each visualization, which occurs because the handle used to close the cooler was a reflective black, which caused depth estimation errors on the Kinect.

Figures 32 and 33 show the operators for picking up and putting down the bottle at the cooler and cupboard, respectively. All four operators require the robot to be at the relevant location, and switch between it holding an object in its left hand (`symbol2`) and not holding one (`symbol3`). Picking an object up at the cooler requires both `symbol8`, which indicates that the cooler is open, and `symbol12`, which can be interpreted as an object being present in the cooler (the conjunction of these two symbols is visualized in Figure 32e). It

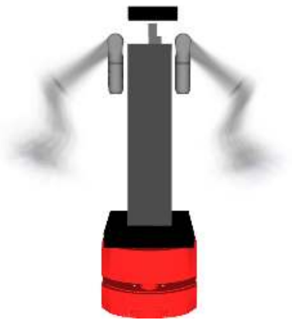
```
(:action pick_up1
 :parameters ()
 :precondition (and (symbol0) (symbol8)
                    (symbol12))
 :effect (and (symbol11) (symbol2)
              (not (symbol3)) (not (symbol12))
              (decrease (reward) 52.62))
)

(:action put_down1
 :parameters ()
 :precondition (and (symbol0) (symbol2)
                    (symbol8))
 :effect (probabilistic
    0.0554 (and)
    0.9446 (and (symbol12) (symbol3)
                (not (symbol2)) (not (symbol11)).
                (decrease (reward) 49.49))
         )
)
```

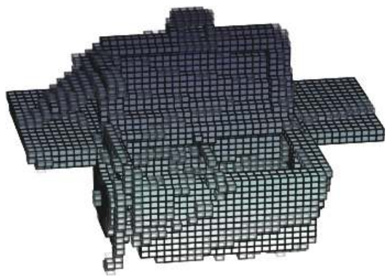
(a) Operator code.



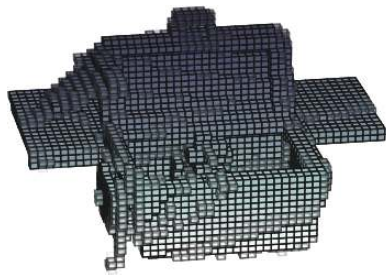
(b) `symbol2`



(c) `symbol3`



(d) `(and (symbol8) (symbol11))`



(e) `(and (symbol8) (symbol12))`

Figure 32: The learned operators (a) and relevant symbolic expressions for picking up and putting down the bottle from the cooler. Both operators require the robot be at the cooler (`symbol0`, shown previously) and that the cooler be open (`symbol8`). The operators switch between a combination of `symbol2` (indicating that an object is in the robot's gripper, b) and `symbol11` (which indicates that there is no object in the cooler, visualized with the cooler open, d) and a combination of `symbol3` (c, no object in gripper) and `symbol12` (which can be interpreted as the bottle being in the cooler; visualized with the cooler open, e).

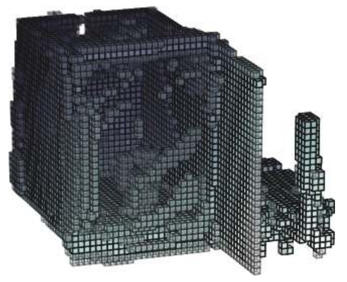
results in the robot holding the bottle (`symbol3`), and sets `symbol12` to false and `symbol11`, which can be interpreted as there not being an object in the cooler, to true (visualized, in conjunction with the open cooler, in Figure 32d). Since there is only one bottle in the task, the `put_down` operator for the cooler only requires that the robot be holding any object, and does not mention the redundant condition (represented by `symbol11`) that the cooler is already empty.

Picking up and putting down the object in the cupboard requires both the cupboard to be open and the switch to be up (`symbol5 and symbol6`, Figure 33b), because when the switch is up it turns the light inside the cupboard off, allowing Anathema to perceive

```
(:action pick_up2
 :parameters ()
 :precondition (and (symbol1) (symbol3)
                    (symbol5) (symbol6) (symbol11))
 :effect (probabilistic
    0.0559 (and)
    0.9441 (and (symbol2) (not (symbol3))
                (decrease (reward) 53.42))
         )
)

(:action put_down2
 :parameters ()
 :precondition (and (symbol1) (symbol2) (symbol5) (symbol6))
 :effect (probabilistic
    0.0697 (and)
    0.9303 (and (symbol3) (not (symbol2))
                (decrease (reward) 49.67))
          )
)
```

(a) Operator code.



(b) `(and (symbol5) (symbol6))`

Figure 33: The learned operators (a) for picking up and putting down the bottle from the cupboard. Both operators require the robot be at the cupboard (`symbol1`, shown previously), and that both the switch is up and the cupboard door is open (`symbol5 and symbol6`, shown in b). The two operators switch between the robot holding and not holding the bottle (`symbol2` and `symbol3`). The `pick_up` operator additionally requires that the bottle is not in the cooler (`symbol11`).

the ARTags marking the bottle and the bottom of the cupboard. Otherwise the bright light washes out the tags, and the skills cannot be run. Operator `put_down` requires the robot to be holding the bottle (`symbol2`), and results in it not holding the bottle (`symbol2`). Operator `pick_up` requires the robot to not be holding the bottle, and, in addition, for the bottle to be absent from the cooler (`symbol11`). Note that there is no symbol representing whether or not the bottle is in the cupboard—that would be redundant, because in this task there are only three places the bottle can be: if it is not in the cooler or the robot's gripper then it must be in the cupboard.

The learned symbolic representation presented above is a complete (though approximate) abstract description of the bottle task. Although we have shown the grounding distributions for each symbol, they are unnecessary for planning, which requires only the generated PPDDL code. Our system has autonomously reduced a high-dimensional, continuous state space (expressing the complexity of the robot's own sensorimotor space) to a much simpler discrete abstract state space—abstracting the details of the robot away to expose the underlying simplicity of the task itself.

5.3 Planning Using the Learned Symbolic Representation

We now demonstrate the use of our learned symbolic representation for planning in two test planning problems, and describe how the resulting plan execution unfolds on the robot. In the first plan, whose PPDDL code is given in Figure 34a, the robot must move the bottle from the cooler to the cupboard. The robot begins execution in front of the cooler, with the light switched on. Both containers are closed. The goal does not specify the location of the robot, but does specify that the light should be off when execution ceases.

```
(define (problem object_to_cupboard)
  (:domain ANA_TASK)
  (:init  (symbol0) ; At the cooler
          (symbol3) ; Arm stowed
          (symbol4) ; Cupboard closed
          (symbol7) ; Switch on
          (symbol9) ; Cooler closed
          (symbol12); Object in cooler
   )
  (:goal (and
          (symbol3)  ; Arm stowed
          (symbol4)  ; Cupboard closed
          (symbol6)  ; Switch off
          (symbol9)  ; Cooler closed
          (symbol11) ; Object not in cooler
         )
  )
)
```

(a)

```
(define (problem object_to_cooler)
  (:domain ANA_TASK)
  (:init  (symbol1) ; At the cupboard
          (symbol3) ; Arm stowed
          (symbol4) ; Cupboard closed
          (symbol7) ; Switch on
          (symbol9) ; Cooler closed
          (symbol11); Object not in cooler
   )
  (:goal (and
          (symbol3)  ; Arm stowed
          (symbol4)  ; Cupboard closed
          (symbol7)  ; Switch on
          (symbol9)  ; Cooler closed
          (symbol12) ; Object in cooler
         )
  )
)
```

(b)

Figure 34: PPDDL code for Anathema's first (a) and second (b) planning tasks.

In the second, whose PPDDL code is given in Figure 34b, the robot must do the reverse: move the bottle from the cupboard to the cooler. As before, both containers are closed and the light is on, but this time the robot's starting location is in front of the cupboard. The goal again does not specify the end location of the robot, but this time requires the light to be on when execution ceases.

We first evaluated the time required to find each plan, using the mGPT planner used in the previous section (Bonet & Geffner, 2005), on a 3Ghz Intel Xeon with 16GB of RAM. The two plans were found in 5 and 4 milliseconds, respectively. This is made possible because both of these planning tasks are fundamentally very simple; this simplicity can be exploited by abstracting away the complexity of the sensorimotor space in which they must be performed, and planning in a much simpler—but still essentially complete—space.

Having the robot execute the resulting plans requires a mechanism for interfacing the planner (and its high-level state space) with the robot at execution time. When a decision must be made, this process must determine which propositional symbols are true, then use the planner to generate a plan, and then execute the first action, repeating the process until the goal is reached. The only difficulty here is determining which propositional symbols should be considered true given sensor data, since our symbolic semantics are probability distributions over future states. Given the current low-level state (after perceptual processing), we simply used the proposition with the grounding distribution that evaluated to the highest density, repeating this process for each factor. All components of this process were extremely fast except the image alignment portions of the processing pipeline, which in some cases required up to 45 seconds of computation.

Plan execution for the first problem (cooler to cupboard) is shown in Figure 35. Even though Anathema starts in front of the cooler in which the bottle is located and immediately opens the cooler, it does not immediately retrieve the bottle. Instead, it moves to the

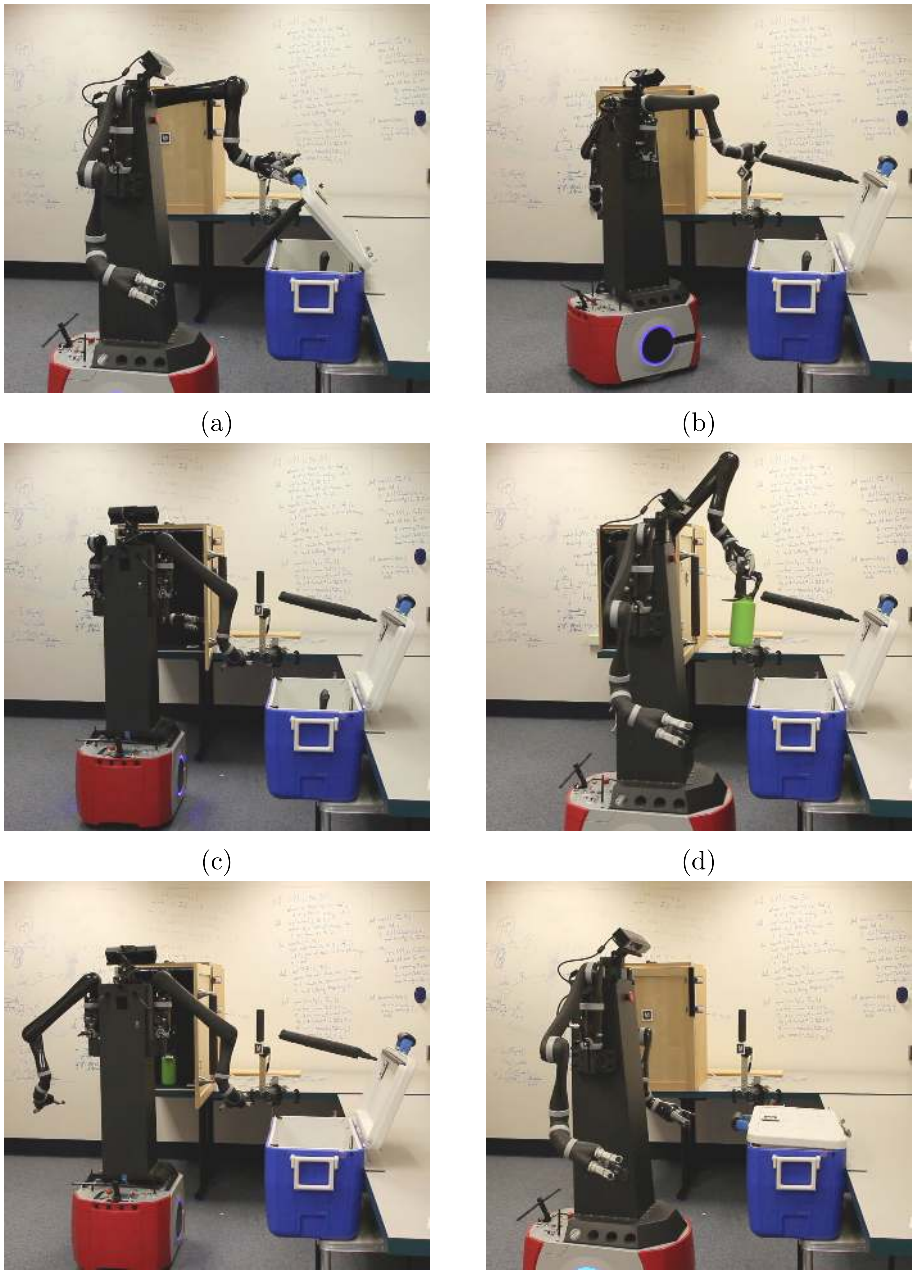

(a) (b) (c) (d) (e) (f)

Figure 35: Executing a plan to move the bottle from the cooler to the cupboard. Anathema first opens the cooler (a), then moves to the cupboard, and switches off the light (b) before opening the cupboard (c). Only then does the robot return to the cooler to pick up the bottle (d) which is placed in the cupboard (e), before the cooler and cupboard are closed (f).

cupboard, because the cupboard cannot be opened once it has an object in its gripper. Once there, it *first* turns the switch to the "off" position, and only *then* opens the cupboard. It must do this because the open cupboard blocks access to the switch, which must be off so that the robot can place the bottle inside the cupboard. Only once the cupboard has been appropriately configured does Anathema return to the cooler, pick up the bottle, and move it to the cupboard, before closing both containers.

Similarly, for the second planning task (Figure 36), Anathema starts in front of the cupboard that contains that bottle. Instead of immediately opening the cupboard, the robot moves to the cooler and opens it, and then returns to the cupboard, first switching the light off, and only then opening the cupboard. The robot then picks up the bottle and moves it to the cooler, before closing both containers and turning the switch back to the "on" position.

Anathema's behavior is characteristic of an agent performing high-level planning—it identifies and deals with future dependencies ahead of time. In both cases Anathema turns off the light before opening the cupboard, because the abstract representation available to it allows it to infer both that the light must be turned off, and that it cannot do so with the cupboard open. Similarly, both containers are opened before the bottle is ever picked up, because the planner realizes that neither container can be open when the robot is already holding the bottle. These kinds of behaviors are very difficult to generate when reasoning at the level of pixels and motors, but easy to generate given the right high-level representation. Our results have shown how such a representation can be learned for a complex robot, directly from sensorimotor data, and subsequently used to plan—bridging the gap between low-level sensorimotor embodiment and high-level planning.

6. Related Work

Several researchers have learned symbolic models of the preconditions and effects of pre-existing controllers for later use in planning (Drescher, 1991; Schmill, Oates, & Cohen, 2000; Pasula, Zettlemoyer, & Kaelbling, 2007; Amir & Chang, 2008; Kruger, Geib, Piater, Petrick, Steedman, Wörgötter, Ude, Asfour, Kraft, Omrčen, Agostini, & Dillmann, 2011; Lang, Toussaint, & Kersting, 2012; Mourão, Zettlemoyer, Patrick, & Steedman, 2012); similar approaches have have been applied under the heading of relational reinforcement learning (Džeroski, De Raedt, & Driessens, 2001). One such approach uses *Object-Action Complexes* (Kruger et al., 2011), a form of action schema that can composed hierarchically and used at any level of abstraction. However, in all of these cases, the high-level symbolic predicates (or attributes) used to learn the model were pre-specified; our work shows how to learn them.

We know of very few systems where a high-level, abstract state space is formed using low-level descriptions of skills. The most closely related work is that of Jetchev, Lang, and Toussaint (2013), which uses the same operational definition of a symbol as our set-based planning formalism. Their method finds relational, probabilistic STRIPS operators by searching for symbol groundings that maximize a metric that balances transition and reward predictability with model size. They are able to find small numbers of symbols in real-world data, but are hampered by the size of the search space, which contains a

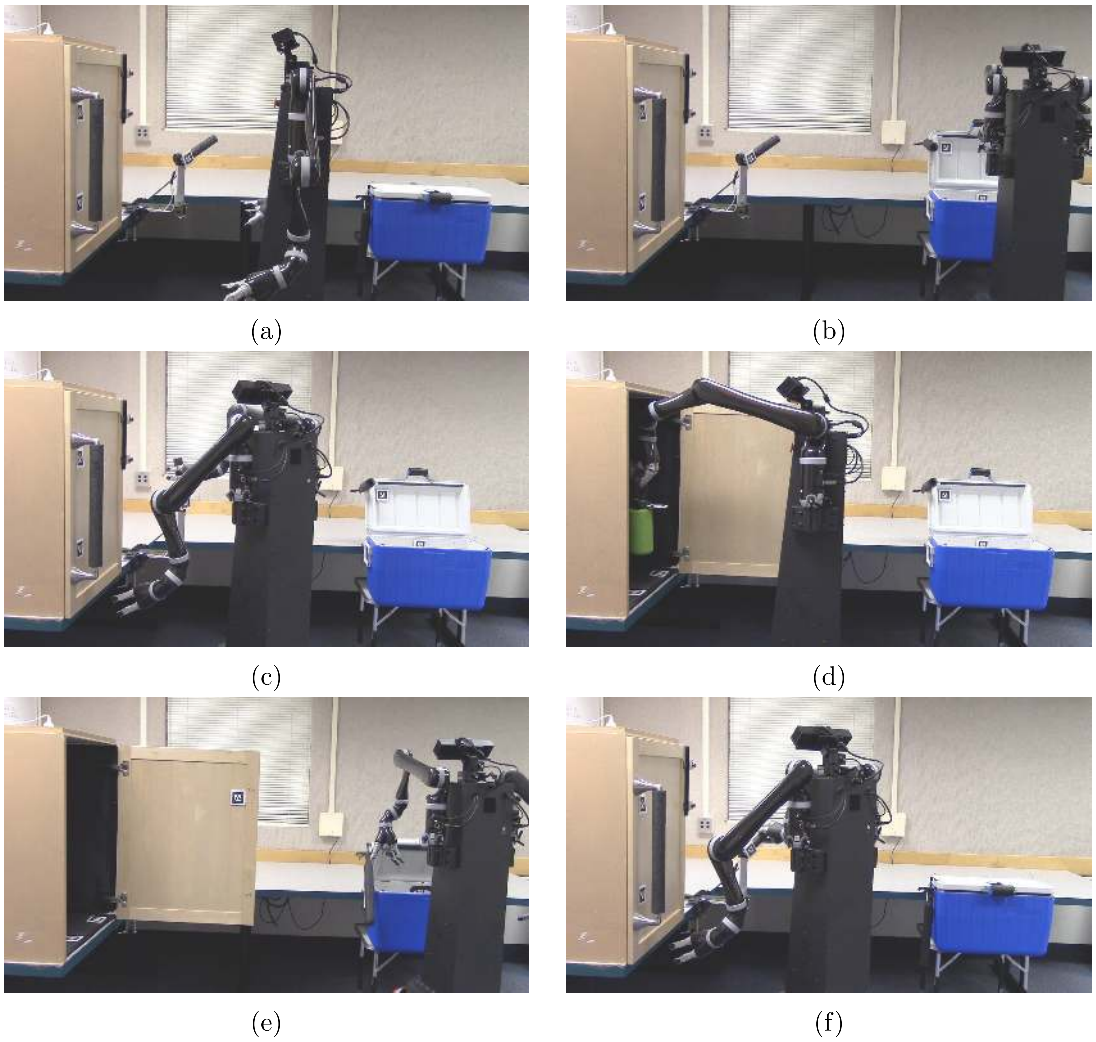


Figure 36: Moving the bottle from the cupboard to the cooler. Anathema starts in front of the cupboard (a), but first moves away from the location of the bottle to open the cooler (b). Once that is done, the robot returns to the cupboard and switches the light off (c) before opening the cupboard and retrieving the bottle (d). The bottle is placed in the cooler, which is then closed (e). Finally, Anathema returns to the cupboard and closes it, before turning the light back on (f).

dimension for every parameter of every grounding classifier. We obviate the need for this search by directly specifying the precise symbol groundings required for planning.

Ugur and Piater (2015a, 2015b) introduced an object-centric approach to learning symbolic representations of manipulation tasks. Their approach executes a collection of motor controllers and observes the resulting changes in the low-level states of the objects to which they are applied. These changes are clustered to form outcome classes (similar to our effect outcomes), and objects are grouped into types based on their outcome profile. The resulting model is converted to PDDL and used for planning. This method learns parametrized symbolic representations (as opposed to our simpler propositional representations) since it can generalize by object type. However, it relies on pre-specified (though quite generic) object relations, requires the objects to be identified and their object-specific features extracted prior to learning, and may result in representations that are neither sound nor suitable for planning.

Huber (2000) introduced an approach where the state of the system is described by the (discrete) state of a set of controllers (equivalent to our options, or skills). For example, if a robot has three controllers, each one of which can be in one of four states (cannot run, can run, running, completed), then we can form a discrete state space made up of three attributes, each of which can take on four possible values. Hart (2009) subsequently used this formulation to learn hierarchical manipulation schemas. This approach was one of the first to construct a symbolic description using motor controller properties. However, the resulting symbolic description assumes that the state of each controller is a sufficient statistic for planning. It thus only results in correct reasoning when all of the low-level states in which a controller can enter a particular state are equivalent. When that is not the case—for example, when a grasp can converge in two different configurations (perhaps depending on the robot's pose when it was initiated), only one of which enables a subsequent controller to execute—it may incorrectly evaluate the feasibility of a plan.

Finally, Modayil and Kuipers (2008) show that a robot can learn to distinguish objects in its environment via unsupervised learning, and then use a learned model of the motion of the object given an action to perform high-level planning. However, the learned models are still in the original state space. Later work by Mugan and Kuipers (2009, 2012) uses *qualitative distinctions* to adaptively discretize a continuous state space in order to acquire a discrete model suitable for planning; here, discretization is based on the ability to predict the outcome of executing an action. New actions are created to "set" the values of new variables, and then the discretized model is refined if necessary in order to predict the outcome of executing an action. This work is closely related to ours in that it builds a symbolic representation used to predict the outcome of executing options, though its propositions are defined only over single variables and the process begins with an initial discretization, rather than a set of skills.

Other approaches to MDP abstraction have focused on discretizing large continuous MDPs into abstract discrete MDPs (Munos & Moore, 1999) or minimizing the size of a discrete MDP model (Dean & Givan, 1997).

7. Discussion and Future Work

Section 5 provides an example of a robot autonomously acquiring its own symbolic representation of the world, directly from low-level sensorimotor data, and then using it to plan. This very limited demonstration used only a small set of skills in an environment engineered for simplicity; we have only begun to understand how to design robot systems that can reliably acquire symbolic representations in an open-ended and completely autonomous fashion. However, the framework developed here has some immediate implications which suggest directions that may prove useful in designing more sophisticated systems.

7.1 Implications for Skill Design and Acquisition

Our representation construction method depends critically on the availability of a suitable set of skills. In many cases, it is reasonable—or even desirable—for these to be designed. For example, a robot might be required to have a specific set of capabilities, each of which can be implemented using motion planning and modeled as an option. However, in many cases finding the appropriate set of skills is part of the problem.

Fortunately, a large body of research on skill acquisition (Menache, Mannor, & Shimkin, 2002; Şimşek & Barto, 2009; Mugan & Kuipers, 2009; Konidaris & Barto, 2009b; Vigorito & Barto, 2010; Mugan & Kuipers, 2012) is concerned with the problem of agents discovering skills autonomously, though these have been applied to robots in only a few cases (Hart, 2009; Konidaris, Kuindersma, Grupen, & Barto, 2011; Riano & McGinnity, 2012; Kompella, Stollenga, Luciw, & Schmidhuber, 2017). Although we assume that the skills are already present, our work does suggest some guidelines for designing or acquiring skills that enable effective high-level planning. For a subgoal option to be useful in a plan graph, its effects set should be a subset of the initiation set of any option that the agent may wish to execute after it. This suggests that options should be constructed so that their termination conditions lie inside the initiation sets for potential successor options. This is the principle underlying pre-image backchaining (Lozano-Perez, Mason, & Taylor, 1984; Burridge, Rizzi, & Koditschek, 1999), the LQR-Tree feedback motion planner (Tedrake, 2009), and the skill chaining skill discovery method (Konidaris & Barto, 2009b). More generally, initiation sets should be as large as possible, to increase the likelihood of a feasible plan; conversely, an option's image should should be compact, for the same reason. This argument supports the broadly accepted idea that skills should be "funnel-shaped" (Lozano-Perez et al., 1984; Burridge et al., 1999), reliably moving the agent from a large number of start states to a small number of end states. Similarly, skills should be designed so that their policies largely subsume the stochasticity of the underlying task, in order to reduce the size of each option's image. This is necessary even in the probabilistic setting: the probability of an agent being able to successfully execute a plan drops very quickly with the plan's length, unless each action succeeds with a very high probability.

Finally, the theory developed here suggests that the abstract subgoal and effect independence properties are fundamental to efficient STRIPS-style planning. STRIPS-style planning implicitly assumes that some form of the abstract subgoal property is true, while the effect independence property prevents a combinatorial explosion in the number of symbols and rules. Consequently, skill acquisition algorithms that discover options with abstract subgoals that exhibit the independence property may prove very useful in practice.

7.2 Complex Images and Preconditions

This work has assumed that the appropriate symbol groundings can always be well represented by some family of classifiers or distributions. However, that is not the case in many practical applications. For example, consider a controller that uses a motion planner to grasp an object. The precondition for that skill depends on the geometry of the scene facing the robot—whether such a path exists (and therefore whether or not the skill can be run) depends on the exact details of the geometric layout of the scene facing the robot, and may be hard (or even impossible) to compute. In addition, the planner does not obey the strong abstract subgoal condition—the end state of the robot's hand may depend very much on its start state, or on the details of the environment the robot is operating in (such as the height of a shelf). Attempts to resolve this difficulty lie at the heart of several recent approaches to combining high-level and geometric motion planning (Cambon, Alami, & Gravot, 2009; Choi & Amir, 2009; Dornhege, Gissler, Teschner, & Nebel, 2009; Wolfe, Marthi, & Russell, 2010; Kaelbling & Lozano-Pérez, 2011); similar issues may also arise for other classes of symbols.

This problem is fundamental and intractable, and some form of compromise is inevitable. If perfect accuracy is required we cannot expect to do any better than planning or learning at the lowest level available to the robot. In practice, we must hope that a potentially inaccurate approximation of these symbol groundings—perhaps augmented with a low-level planner that can cause high-level planning to backtrack—will be sufficient. We see two reasons for optimism. The first is that, while we cannot expect the strong abstract subgoal condition to hold, in many cases it is reasonable to expect that the weak abstract subgoal condition will, especially when creating motor skills with relatively small images. Second, we can use probability distributions to express uncertainty about approximating complex, hard-to-compute symbol groundings. For example, we might compute a quick-and-dirty visibility measure as a stand-in for the set of states from which a reaching motion may be executable, and build the uncertainty associated with that proxy measure into the distribution itself. In such cases, the distribution is doing triple duty—it incorporates uncertainty due to stochasticity in the world, uncertainty due to learning, and uncertainty due to abstraction—but it affords us the ability to make more robust plans by expressing the possibility that we may be wrong.

7.3 Learning Generalized Symbols

Our framework focuses on individual symbols defined directly over the state space of one particular domain. However, if symbol acquisition is to be feasible for more interesting robot problems, all opportunities for data efficiency should be exploited.

Consider a scenario in which a mobile robot with a gripper operates in an office building, with motor skills for picking up objects, navigating to various locations in the building (possibly while holding an object), and putting objects back down. Our framework will work for these skills, but will require a large number of object- and location-specific partitions, each resulting in preconditions and effects that must be learned separately. The robot would therefore have to gather multiple samples of the outcome of executing the pick-up action for every object in every location it might navigate to, and of navigating to every location with every object in its gripper.

Such a scenario rapidly becomes impractical. To avoid unnecessary learning, we would prefer to learn a representation that generalizes across these obviously related motor skill instantiations. There are at least two opportunities for generalization in this example:

1. *Generalization across locations.* Consider the robot picking up an object. We should be able to model the precondition (the object is visible, within reachable distance, and has sufficient free space around it) and the results of skill execution (the object is in the robot's gripper) independently of the robot's location in the building.

2. *Generalization across objects.* Consider the robot navigating to a location while holding an object. We should be able to model the effect of executing this skill for *any* object, without having to re-learn the effect for each object independently.

Such generalization would produce a portable definition of the preconditions and effects of the pick-up and navigation skills that can be bound to each object or location as necessary. A formulation that cleanly captures this kind of generalization will drastically reduce the sample complexity of learning a symbolic representation.

7.4 Actions Entail Representation

The most interesting implication of our results is that motor skills play a central role in determining the representational requirements of an intelligent agent. If generating behavior via symbolic planning is an agent's goal, its representation must directly model properties of its skills; *a suitable symbolic description of a domain depends on the skills available to the agent.*

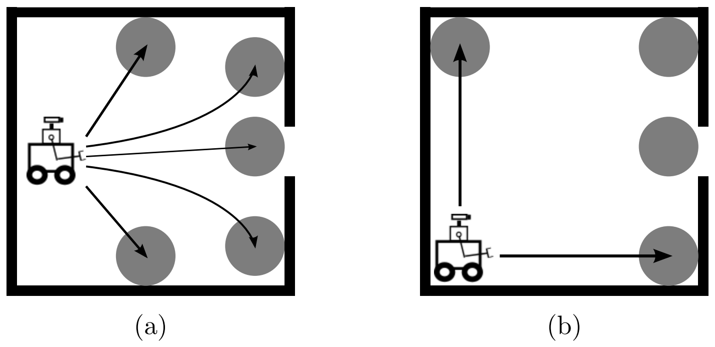


Figure 37: A robot navigating a room using a) subgoal options for moving to distinct walls, or b) wall-following subgoal options. The effect set for each option is shown in gray. Here we see that changing the options available to the agent changes the symbols it requires for planning, even if the environment remains unchanged.

Figure 37 shows two cases where the same robot must navigate the same room, but requires different symbolic representations because it uses different navigation controllers. In Figure 37a, the robot uses motor skills that drive it to the centroid of each wall. As a consequence, the symbols required to represent the effect of executing its options might be described as $AtLeftWall$, $AtRightWall$, $AtEntrance$, etc. In Figure 37b, the robot executes a wall-following controller and stops when it reaches a corner or a door. As a consequence, the symbols required to represent the effect of executing its options could be described as $LowerLeftCorner$, $LowerRightCorner$, etc.

In some sense, only an agent's skills and low-level perception are truly grounded in its environment; it is free to invent whatever other aspects of its control system that it likes. Those structures can only be *useful*, however, by virtue of improving its ability to act in the world. Thus, we view it as natural that the agent's symbolic representations reflect the demands of control—indeed, all mechanisms ultimately must.

8. Conclusions

Our approach can be contrasted with the standard method of merging high-level planning and low-level control, where human designers specify a symbolic description of the world and then write the perceptual and motor control programs necessary to ground and execute plans formed using that description. Such approaches can be considered *symbol-down*, because they first construct an abstract description of the world, and then design the remainder of the control system to support it. One difficulty posed when constructing symbol-down systems is that we are not afforded much guidance as to the appropriate symbolic description of the world; we generally design it by a combination of intuition and introspection, and then attempt to ground it in a robot with very different sensors and actuators than ourselves. How do we know when our symbolic representation is correct? While a motor skill directly generates behavior that enables us to empirically evaluate its performance, no such evaluation process is available for abstract symbolic representations.

By contrast, our approach can be considered *skill-up*, where we use an existing set of skills—which could be provided by a designer, or learned by the robot autonomously (Hart, 2009; Konidaris et al., 2011; Riano & McGinnity, 2012; Kompella et al., 2017) or by demonstration (Jenkins & Matarić, 2004; Kulić, Takano, & Nakamura, 2009; Chiappa & Peters, 2010; Konidaris, Kuindersma, Grupen, & Barto, 2012; Niekum, Osentoski, Konidaris, & Barto, 2012)—as a basis for building a representation appropriate for planning. Rather than defining a representation and then constructing a robot system to support it, our approach is based on clearly defining behavioral goals based on skills and then constructing a representation that is guaranteed to meet those goals. This approach has exposed the intimate relationship between the skills an agent has available and the representation required for planning using them. When the entire robot control system (including its available symbols) is to be engineered for a specific purpose, this relationship eliminates the symbol design problem that often plagues such systems in practice. But our formalism allows us to go even further, and design agents that learn their own high-level, symbolic representation—one that is correct, grounded, and useful by construction—completely autonomously.

Our work follows a long tradition in AI of structuring agent control architectures around procedural abstraction. The idea that the complexity of generating intelligent behavior can be managed by breaking it into discrete components that recur, in much the same way that programs are broken into subroutines, reflects an implicit foundational hypothesis that *behavior is modular and compositional.* The procedural abstraction hypothesis is fundamental to classical planning and hierarchical reinforcement learning, as well as virtually all robot architectures, from the STRIPS-based classical planning of Shakey (Nilsson, 1984) to the reactive behavioral modules of the subsumption architecture (Brooks, 1991). Our results show that procedural abstraction is even more fundamental, because it entails representational abstraction: the appropriate representation for reasoning about behavior follows from the procedural abstractions it is composed of.

Acknowledgments

The authors would like to thank Alejandro Perez, Patrick Barragán, and Lawson Wong for their critical assistance in designing the robot experiment, and Cameron Allen, Barrett Ames, Garrett Andersen, Ben Burchfiel, Steven James, and Yuu Jinnai for their useful comments and suggestions. We thank the remainder of the LIS and IRL, Greg Lewis, Siddharth Srivastava, Frank Guerin, and Marc Bellemare for thoughtful conversations. George Konidaris was partially supported during this work by an Intelligence Initiative Fellowship from MIT, and by a DARPA Young Faculty Award under agreement number D15AP00104.

References

Amir, E., & Chang, A. (2008). Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, *33*, 349–402.

Andersen, G., & Konidaris, G. (2017). Active exploration for learning symbolic representations. In *Advances in Neural Information Processing Systems 30*, pp. 5016–5026.

Barto, A., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*, 41–77.

Beeson, P., & Ames, B. (2015). TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *Proceedings of the IEEE RAS Humanoids Conference*, pp. 928–935, Seoul, Korea.

Besl, P., & McKay, N. (1992). Method for registration of 3-D shapes. *Pattern Analysis and Machine Intelligence*, *14*, 239–256.

Bonet, B., & Geffner, H. (2005). mGPT: a probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, *24*, 933–944.

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1104–1113.

Brooks, R. (1991). Intelligence without representation. In Haugeland, J. (Ed.), *Mind Design II*, pp. 395–420. MIT Press, Cambridge, Massachusetts.

Brunskill, E., & Li, L. (2013). Sample complexity of multi-task reinforcement learning. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, pp. 122–131.

Burridge, R., Rizzi, A., & Koditschek, D. (1999). Sequential composition of dynamically dextrous robot behaviors. *International Journal of Robotics Research*, *18*(6), 534–555.

Cambon, S., Alami, R., & Gravot, F. (2009). A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research*, *28*(1), 104–126.

Chiappa, S., & Peters, J. (2010). Movement extraction by detecting dynamics switches and repetitions. In *Advances in Neural Information Processing Systems 23*, pp. 388–396.

Choi, J., & Amir, E. (2009). Combining planning and motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 4374–4380.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297.

Dean, T., & Givan, R. (1997). Model minimization in Markov decision processes. In *In Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 106–111.

Dornhege, C., Gissler, M., Teschner, M., & Nebel, B. (2009). Integrating symbolic and geometric planning for mobile manipulation.. In *IEEE International Workshop on Safety, Security and Rescue Robotics*.

Drescher, G. (1991). *Made-Up Minds: A Constructivist Approach to Artificial Intelligence*. MIT Press.

Drucker, H., Burges, C., Kaufman, L., Smola, A., & Vapnik, V. (1997). Support vector regression machines. In *Advances in Neural Information Processing Systems 9*, pp. 155–161.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine learning*, *43*(1), 7–52.

Ester, M., Kriegel, H., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pp. 226–231.

Fikes, R., & Nilsson, N. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Gat, E. (1998). On three-layer architectures.. In Kortenkamp, D., Bonnasso, R., & Murphy, R. (Eds.), *Artificial Intelligence and Mobile Robots*. AAAI Press.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated planning: theory and practice*. Morgan Kaufmann.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. (2009). The WEKA data mining software: An update. *SIGKDD Explorations*, *11*(1), 10–18.

Hart, S. (2009). *The Development of Hierarchical Knowledge in Robot Systems*. Ph.D. thesis, University of Massachusetts Amherst.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Huber, M. (2000). A hybrid architecture for hierarchical reinforcement learning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3290–3295.

Huber, M., & Grupen, R. (1997). Learning to coordinate controllers - reinforcement learning on a control basis. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1366–1371.

Jenkins, O., & Matarić, M. (2004). Performance-derived behavior vocabularies: data-driven acquisition of skills from motion. *International Journal of Humanoid Robotics*, *1*(2), 237–288.

Jetchev, N., Lang, T., & Toussaint, M. (2013). Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the 2013 ICRA Workshop on Autonomous Learning*.

Kaelbling, L., & Lozano-Pérez, T. (2011). Hierarchical task and motion planning in the Now. In *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1470–1477.

Kato, H., & Billinghurst, M. (1999). Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, pp. 282–293.

Koenig, S. (1991). Optimal probabilistic and decision-theoretic planning using Markovian decision theory. Master's thesis, Computer Science Department, University of California at Berkeley, Berkeley, CA.

Kompella, V. R., Stollenga, M., Luciw, M., & Schmidhuber, J. (2017). Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots. *Artificial Intelligence*, *247*, 313–335.

Konidaris, G., & Barto, A. (2009a). Efficient skill learning using abstraction selection. In *Proceedings of the Twenty First International Joint Conference on Artificial Intelligence*, pp. 1107–1112.

Konidaris, G., & Barto, A. (2009b). Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems 22*, pp. 1015–1023.

Konidaris, G., Kaelbling, L., & Lozano-Perez, T. (2014). Constructing symbolic representations for high-level planning. In *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence*, pp. 1932–1940.

Konidaris, G., Kaelbling, L., & Lozano-Perez, T. (2015). Symbol acquisition for probabilistic high-level planning. In *Proceedings of the Twenty Fourth International Joint Conference on Artificial Intelligence*, pp. 3619–3627.

Konidaris, G., Kuindersma, S., Grupen, R., & Barto, A. (2011). Autonomous skill acquisition on a mobile manipulator. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pp. 1468–1473.

Konidaris, G., Kuindersma, S., Grupen, R., & Barto, A. (2012). Robot learning from demonstration by constructing skill trees. *International Journal of Robotics Research*, *31*(3), 360–375.

Kruger, N., Geib, C., Piater, J., Petrick, R., Steedman, M., Wörgötter, F., Ude, A., Asfour, T., Kraft, D., Omrčen, D., Agostini, A., & Dillmann, R. (2011). Object-action complexes: Grounded abstractions of sensory-motor processes. *Robotics and Autonomous Systems*, *59*, 740–757.

Kulić, D., Takano, W., & Nakamura, Y. (2009). Online segmentation and clustering from continuous observation of whole body motions. *IEEE Transactions on Robotics*, *25*(5), 1158–1166.

Lang, T., Toussaint, M., & Kersting, K. (2012). Exploration in relational domains for model-based reinforcement learning. *Journal of Machine Learning Research*, *13*, 3691–3734.

Lozano-Perez, T., Mason, M., & Taylor, R. (1984). Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, *3*(1), 3–24.

Malcolm, C., & Smithers, T. (1990). Symbol grounding via a hybrid architecture in an autonomous assembly system. *Robotics and Autonomous Systems*, *6*(1-2), 123–144.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). PDDL—the planning domain definition language. Tech. rep. CVC TR98003/DCS TR1165, Yale Center for Computational Vision and Control.

Menache, I., Mannor, S., & Shimkin, N. (2002). Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the Thirteenth European Conference on Machine Learning*, pp. 295–306.

Modayil, J., & Kuipers, B. (2008). The initial development of object knowledge by a learning robot. *Robotics and Autonomous Systems*, *56*(11), 879–890.

Mourão, K., Zettlemoyer, L., Patrick, R., & Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proceedings of Conference on Uncertainty in Articial Intelligence*, pp. 614–623.

Mugan, J., & Kuipers, B. (2009). Autonomously learning an action hierarchy using a learned qualitative state representation. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pp. 1175–1180.

Mugan, J., & Kuipers, B. (2012). Autonomous learning of high-level states and actions in continuous environments. *IEEE Transactions on Autonomous Mental Development*, *4*(1), 70–86.

Munos, R., & Moore, A. (1999). Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pp. 1348–1355.

Niekum, S. (2011). ar_track_alvar. `http://wiki.ros.org/ar_track_alvar`.

Niekum, S., Osentoski, S., Konidaris, G., & Barto, A. (2012). Learning and generalization of complex tasks from unstructured demonstrations. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5239–5246.

Nilsson, N. (1984). Shakey the robot. Tech. rep., SRI International.

Parzen, E. (1962). On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, *33*(3), 1065.

Pasula, H., Zettlemoyer, L., & Kaelbling, L. (2007). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, *29*, 309–352.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Platt, J. (1999). Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pp. 61–74. MIT Press.

Powley, E., Whitehouse, D., & Cowling, P. (2012). Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, pp. 234–241.

Precup, D. (2000). *Temporal Abstraction in Reinforcement Learning*. Ph.D. thesis, Department of Computer Science, University of Massachusetts Amherst.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., & Ng, A. (2009). ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

Quinlan, J. (1993). *C4.5: programs for machine learning*, Vol. 1. Morgan Kaufmann.

Riano, L., & McGinnity, T. (2012). Automatically composing and parameterizing skills by evolving finite state automata. *Robotics and Autonomous Systems*, *60*(4), 639–650.

Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, *27*(3), 832.

Rusu, R., & Cousins, S. (2011). 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China.

Schmill, M., Oates, T., & Cohen, P. (2000). Learning planning operators in real-world, partially observable environments. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pp. 245–253.

Silver, D., & Ciosek, K. (2012). Compositional planning using optimal option models. In *Proceedings of the Twenty-Ninth International Conference on Machine Learning*, pp. 1063–1070.

Şimşek, Ö., & Barto, A. (2009). Skill characterization based on betweenness. In *Advances in Neural Information Processing Systems 21*, pp. 1497–1504.

Singh, S., Barto, A., & Chentanez, N. (2005). Intrinsically motivated reinforcement learning. In *Advanced in Neural Information Processing 17*, pp. 1281–1288.

Sorg, J., & Singh, S. (2010). Linear options. In *Proceedings of the Ninth International Conference on Autonomous Systems and Multiagent Systems*, pp. 31–38.

Sucan, I., & Chitta, S. (2013). MoveIt!. `http://moveit.ros.org`.

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1-2), 181–211.

Tedrake, R. (2009). LQR-Trees: Feedback motion planning on sparse randomized trees.. In *Robotics: Science and Systems V*, pp. 18–24.

Ugur, E., & Piater, J. (2015a). Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2627–2633. IEEE.

Ugur, E., & Piater, J. (2015b). Refining discovered symbols with multi-step interaction experience. In *Proceedings of the 15th IEEE-RAS International Conference on Humanoid Robots*, pp. 1007–1012. IEEE.

Vigorito, C., & Barto, A. (2010). Intrinsically motivated hierarchical skill learning in structured environments. *IEEE Transactions on Autonomous Mental Development*, *2*, 132–143.

Wiedemeyer, T. (2014). IAI Kinect2. `https://github.com/code-iai/iai\_kinect2`. Accessed June 12, 2015.

Wilson, A., Fern, A., Ray, S., & Tadepalli, P. (2007). Multi-task reinforcement learning: a hierarchical Bayesian approach. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pp. 1015–1022.

Wolfe, J., Marthi, B., & Russell, S. J. (2010). Combined Task and Motion Planning for Mobile Manipulation. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pp. 254–257.

Younes, H., & Littman, M. (2004). PPDDL 1.0: an extension to PDDL for expressing planning domains with probabilistic effects. Tech. rep. CMU-CS-04-167, Carnegie Mellon University.

Zou, H., Hastie, T., & Tibshirani, R. (2006). Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, *15*(2), 265–286.