
Defining Object Types and Options Using MDP Homomorphisms

Alicia Peregrin Wolfe

Andrew G. Barto

PIPPIN@CS.UMASS.EDU

BARTO@CS.UMASS.EDU

Computer Science Department, University of Massachusetts, Amherst, MA 01003 USA

Abstract

Agents in complex environments can have a wide range of tasks to perform over time. However, often there are sets of tasks that involve similar goals on similar objects, e.g., the skill of making a car move to a destination is similar for all cars. This paper lays out a framework for specifying goals that are parameterized with focus objects, as well as defining object *type* in such a way that objects of the same type share policies. The method is agnostic as to the underlying state representation, as long as simple functions of the state of the object can be calculated.

1. Introduction: Modeling Objects

We typically categorize objects in our environment into a set of types, defined by their behavior: cars are objects with wheels that we can drive; cups hold liquid; chairs can be used to sit on. This kind of object type definition is particularly useful to an agent learning to function in a complex environment: the type labels provide useful abstractions over the details of each object's features.

In this work, we use the MDP homomorphism (Ravindran, 2004) framework to determine which objects have the same type. Type is defined here only relative to some task or class of tasks: for example, objects sharing the same type for tasks involving their location may not share the same type for color. From this we show how to construct options that execute over objects of a variety of types, with one policy stored for each allowable type of object.

An option defined for a particular type can be reused on new objects, as long as they fit the criteria for that object type. Once we know how to pick up a cup and

drink, this skill can be applied to any cup.

2. MDP Homomorphisms

A Markov Decision Process (MDP) consists of tuple (S, A, T, R) comprising a state set (S), action set (A), transition function ($T : S \times A \times S \rightarrow [0, 1]$), and expected reward function ($R : S \times A \rightarrow \mathbb{R}$). The transition function defines the probability of transitioning from state to state given the current state and chosen action, while the reward function represents the expected reward the agent receives for being in a particular state and executing an action.

An MDP homomorphism (Ravindran, 2004) is a mapping, $h : S \times A \rightarrow S' \times A'$, from the states and actions of a base MDP, $M = (S, A, T, R)$, to an abstract model MDP $M' = (S', A', T', R')$. To be an MDP homomorphism, h must preserve both the reward function and some properties of the transition probabilities of M . Specifically, h consists of a set of mappings: $f : S \rightarrow S'$, and for each $s \in S$ a mapping $g_s : A \rightarrow A'$ that recodes actions in a possibly state-dependent way. The following properties must hold for all state and action pairs:

$$R'(f(s), g_s(a)) = R(s, a) \quad (1)$$

$$T'(f(s_i), g_{s_i}(a), f(s_j)) = \sum_{s_k | f(s_j) = f(s_k)} T(s_i, a, s_k). \quad (2)$$

Subgoal *options* (Sutton et al., 1999) provide a formalism for specifying multiple episodic subtasks. A subgoal option adds a termination condition $\beta : S \times A \rightarrow [0, 1]$ which specifies the probability that the option will terminate in any particular state. Homomorphisms for subgoal options add one additional constraint over the mapping h (Ravindran & Barto, 2003). For all $s \in S$ and $a \in A$:

$$\beta'(f(s), g_s(a)) = \beta(s, a). \quad (3)$$

When a mapping f can be found that is many-to-one,

the abstract MDP M' has fewer states than M . The homomorphism conditions mean that M' accurately tracks the transitions and rewards of M but at the resolution of blocks of states and actions. These properties guarantee that policies optimal for M' can be *lifted* to produce optimal policies of the larger MDP M (Ravindran, 2004).

Several algorithms exist for finding MDP homomorphisms given a model of an MDP. All proceed by partitioning the state set S in stages. At each stage, the states and actions are partitioned into two sets of blocks: a state (S) partition $\{B_1, \dots, B_m\}$ over states, and a state/action (SA) partition over (s, a) pairs, $\{P_1, \dots, P_n\}$. The S partition defines an f mapping: $s \in B_i \rightarrow f(s) = s'_i$. Similarly, the SA partition defines the set of g_s mappings: $(s, a) \in P_i \rightarrow g_s(a) = a'_i$.

The version of the homomorphism finding algorithm used in this paper is taken from (Ravindran, 2004), though similar examples exist in (Givan et al., 2003) and (Boutilier et al., 2001).

3. CMP Homomorphisms

A Controlled Markov Process (CMP) is an MDP without the latter's reward function. A CMP with output is a CMP together with an output function that maps its state set to a set of outputs, or observations, Y (Wolfe & Barto, 2006). We think of the output function as singling out some aspect of the CMP as being of interest for some reason. This function might be as simple as the location or color of an object in the state. Thus, a CMP with output is a tuple (S, A, T, y) , where S , A , and T are as in an MDP, and y is the output function $y : S \times A \rightarrow Y$. Given any function $r : Y \rightarrow \mathbb{R}$, $(S, A, T, r \circ y)$ is an MDP whose reward function is the composition of r and y . We say that this MDP is *supported by* y . This means that the reward depends only on the observations, not on the complete state. The termination conditions for the family of subgoal options supported by y have the form $\beta : Y \mapsto [0, 1]$.

A CMP homomorphism is a mapping from a CMP with output (S, A, T, y) to an abstract CMP with output (S', A', T', y') . The mapping functions h , f , and g_s are defined as for MDP homomorphisms. The conditions that must be satisfied for this to be a CMP homomorphism are similar to those for an MDP, with the constraints over the expected value of the reward for a state (Equation 1) and termination function β (Equation 3) replaced by a single constraint over the output function, for all $s \in S$ and $a \in A$:

$$y'(f(s), g_s(a)) = y(s, a). \quad (4)$$

The transition function constraints (Equation 2) re-

main the same. The model formed by a CMP homomorphism can be used to learn the value function for any supported reward $(r \circ y)$ and termination $(\beta \circ y)$ functions.

Existing methods for finding MDP homomorphisms apply with trivial modifications to the case of a CMP with output. The initial split on average reward must simply be replaced by a split that clusters the states according to the output function.

4. CMPs with Objects

At this point, we have most of the basic machinery we need to model environments with objects. The main addition made in this section is a transformation of the way we encode the state space: rather than being "global", the output function will now be associated with some object in the environment. The methods we have discussed so far enable us to determine which objects have similar behavior, no matter what state description we use.

4.1. Object CMPs

An Object CMP consists of a CMP, a set of object identifiers O , an object description set D (often factored into a set of features), and a set of functions, one per object, that maps states to object descriptions: $w_o : S \rightarrow D$. The output function z in this case maps object descriptions to outputs: $z : D \rightarrow Y$.

Specifying an object yields a CMP with output, $(S, T, A, z \circ w_o)$, which can be transformed into an MDP by specifying a reward function $(r \circ z \circ w_o)$ and termination function $(\beta \circ z \circ w_o)$ as in the previous sections.

The object/state specification may be as primitive or structured as the designer wishes, as long as there is some way to compute the desired output function, given an object pointer. Take two examples: one state space made up of pixels, one of features. The mapping w_o singles out some subset of pixels or features in each state as belonging to the object o . The features that belong to a particular object are typically fixed and are often named (*object1.position*, for example), whereas the set of pixels belonging to an object might change from state to state. Nonetheless, the mapping z for object position can be calculated from this set of pixels as well as it can from a set of features.

Since there is no guarantee that all objects will be present in all states, the output function $z \circ w_o$ evaluates to a special null output function value, \perp , for states in which the specified object is not present. Ob-

jects can be nested and can overlap.

We assume in this paper that the w_o mappings and function z are given. A homomorphic reduction of the state space for any individual CMP/object pair (M_i, o_i) for an output function z can be found as it would be for any CMP with output. The interesting question is: when do multiple objects o_i and o_j share the same abstract model, though their state/action mapping functions h_i and h_j are different? This equivalence will be key to constructing options that operate over a class of objects.

4.2. Object Options

Now we are ready to define options with object parameters. An object option subgoal consists of a reward and termination function: $(\beta \circ z, r \circ z)$, defined for a particular particular object description space, D . For example, blocks world CMPs would have an object description space of block descriptions. One option subgoal might be to move the focus block to a particular location in a blocks world. The reward function maps the output function $z = \text{“block position”}$ to a reward that is positive when the desired location is achieved and negative otherwise.

All other parts of the option structure follow from the reward function and termination function. Applying the subgoal to a CMP M_i and object o_j yields a subgoal option MDP $(S_i, A_i, T_i, \beta \circ z \circ w_{o_j}, r \circ z \circ w_{o_j})$, from which a policy can be calculated. The homomorphic mapping for the output function transforms the original state set to a state set having the correct “point of view” for the object o_j .

An object can be typed by its homomorphic reductions: if two objects can use the same abstract model for a given output function z , they have the same type for that output function. This definition implies that while the mapping functions h_i and h_j might be different for two objects o_i and o_j , they share the same policy (in the same abstract state/action space) for any reward function supported by z .

Because it is also possible that one object’s abstract model is a sub-model of another, our definition of object type induces a partial ordering over objects. If the function u maps objects in O onto a partially ordered set of type symbols, we would like to find a *consistent* type mapping for the output function z and set of CMPs $C = \{M_i\}$. Consistent type mappings have the following property: if $u(o_i) \preceq u(o_j)$ then for any M_k where h_i is a homomorphic mapping for $z \circ w_{o_i}$, there is a CMP M_l for which h_j is a homomorphic mapping for $z \circ w_{o_j}$ and produces a model with the following

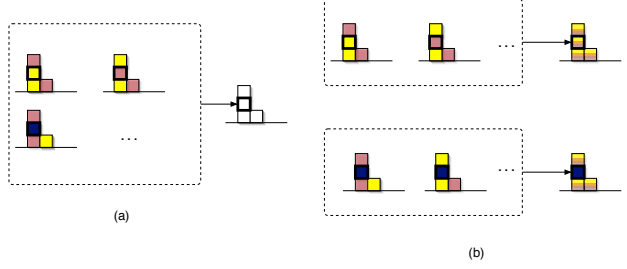


Figure 1. Example clusters from a blocks world with 4 blocks. Blocks in (a) all behave the same way, blue blocks (the dark blocks) in (b) are more slippery. Dashed squares represent blocks of the two state partitions.

property: $h_i(S_k, A_k) \subseteq h_j(S_l, A_l)$ (where \subseteq indicates partial or total model isomorphism).

Figure 1 compares partition blocks for two different blocks world state spaces: one in which all blocks have identical behavior, and one in which blue blocks are more slippery than others. One consistent u mapping for the output function “block position” for Figure 1b is $u(\text{blue blocks}) = \alpha$, $u(\text{other colors}) = \gamma$, $\alpha \neq \gamma$. In Figure 1a all blocks have the same type.

Policies learned for one object can be applied to any object of the same or lesser type. The option may specify a single type, or a range of types allowable for the focus object. The option must store a separate policy for each type of object which is allowable as a parameter. The generic policies are then mapped back to the true CMP by the same *lifting* process used in MDP homomorphisms.

Despite changes in the policy, the reward function is consistent across all possible parameter assignments. The goal of moving a block to position k is the same, whether the block is blue or green—only the method of executing that goal and the probability of success change if the type of the focus object changes.

4.3. Algorithm: Finding Object CMP Homomorphisms

To find reductions that tell us which objects have the same type, some modifications to the original homomorphism-finding algorithms are necessary.

We start with an output function z , and a set of one or more object CMP models $C = \{M_i\}$, each of which has a set of objects O_i . For each sample CMP M_i , we construct a set of $n = \|O_i\|$ CMPs of the form $(S_i, A_i, T_i, z \circ w_{o_k})$. The essential question is: which CMP/object pairs are isomorphic?

The algorithms for finding homomorphisms can be ex-

ecuted over multiple CMPs at once: the algorithm simply considers the combination of the two state and action spaces and proceeds. If, upon termination, the states and actions of two CMPs map to the same abstract states and actions (see Figure 1), their reduced models are isomorphic. If one CMP maps to a sub-model of another, it is partially isomorphic.

The space of all CMP/object pairs is large. It is therefore more efficient for the agent to construct a library of models for different objects as it encounters them, then match new CMP/object pairs encountered to this library. New CMPs can be combined with existing abstract models and the algorithm can be run on this slightly smaller combined CMP.

5. Experiments

All of these experiments used the algorithm above to create a library of models for 3-block blocks worlds. In each case the dynamics of the environment were changed to create a different type mapping.

For the 3-block blocks world with dynamics matching Figure 1b, the algorithm correctly finds two types of blocks, with 6 abstract CMP models, 3 used by blue blocks and 3 by focus blocks of other colors.

For our second example, consider a blocks world in which all blocks of any given color c stick to blocks of the same color. When a block sticks to the block beneath it, the probability of successfully lifting it and moving it to another pile is lower. While all blocks in this example have the same type, their abstract state mapping function is different: two yellow blocks with a blue focus block yields different dynamics than the same CMP with a yellow focus block. The algorithm finds the correct set of 4 abstract CMPs: M'_1 in which all three blocks have different colors (30 states), M'_2 in which the other two blocks match each other but not the focus block (30 states), M'_3 in which one other block matches the focus block color (60 states) and M'_4 in which all three blocks have the same colors (30 states).

Finally, for a simple example of types which are partially ordered, consider a blocks world in which blue and green blocks stick to blocks of the same color, while all other combinations of blocks interact normally. This results in 4 abstract CMPs, shown in Table 1, and two object types. The first type (blue or green focus blocks) uses all 4 models, while the second type (other blocks) uses only the two simpler models.

As expected, learning policies for supported reward functions in the reduced models is faster and has the

Table 1. Abstract CMPs for the blocks world in which blue and green blocks stick to other blocks of the same color.

Models Used by Focus Block				
Focus:	M'_1 (30)	M'_2 (30)	M'_3 (60)	M'_4 (30)
blue	✓	✓	✓	✓
green	✓	✓	✓	✓
red	✓	✓		
yellow	✓	✓		

same optimality guarantees as learning in the complete model, however, due to lack of space we do not include detailed results here.

6. Discussion

In order to use these parameterized options in the context of a higher level task, the agent must learn to assign objects to the pointers. This (along with the work in this paper) is closely tied to the notion of *deixis* (Agre & Chapman, 1987).

The algorithm used here uses a model of the entire MDP, which can be difficult to estimate from data. It would be more efficient to use methods similar to those in Wolfe and Barto (2006) to learn the model directly.

If there are some features of the object which cannot be observed which determine its true type (if the color of blue blocks in Figure 1 was unobserved) the agent is in a special type of POMDP in which the type of the object must be discovered through interaction.

References

- Agre, P. E., & Chapman, D. (1987). Pengi: An implementation of a theory of activity. *Proceedings of the 6th National Conference on Artificial Intelligence*.
- Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order mdps. *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (pp. 690–697).
- Givan, R., Dean, T., & Greig, M. (2003). Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147, 163–223.
- Kersting, K., & Raedt, L. D. (2003). Logical markov decision programs. *Proceedings of the 20th International Conference on Machine Learning*.
- Ravindran, B. (2004). *An algebraic approach to abstraction in reinforcement learning*. Doctoral dissertation, University of Massachusetts.
- Ravindran, B., & Barto, A. G. (2003). Smdp homomorphisms: An algebraic approach to abstraction in semi markov decision processes. *Proceedings of the 18th International Joint Conference on Artificial Intelligence* (pp. 1011–1016). AAAI Press.
- Sutton, R. S., Precup, D., & Singh, S. P. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112, 181–211.
- Wolfe, A. P., & Barto, A. G. (2006). Decision tree methods for finding reusable mdp homomorphisms. *Proceedings of the 21st National Conference on Artificial Intelligence*. To appear.