Integrating Acting, Planning, and Learning in Hierarchical Operational Models

Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, Dana Nau

Integrating Acting, Planning, and Learning in Hierarchical Operational Models

Sunandita Patra[1], **James Mason**[1], **Amit Kumar**[1], **Malik Ghallab**[2], **Paolo Traverso**[3], **Dana Nau**[1]
[1] University of Maryland, College Park, MD 20742, USA
[2] LAAS-CNRS, 31077, Toulouse, France
[3] Fondazione Bruno Kessler, I-38123, Povo-Trento, Italy
{patras@, jmason12@terpmail., akumar14@terpmail.}umd.edu, malik@laas.fr, traverso@fbk.eu, nau@cs.umd.edu

Abstract

We present new planning and learning algorithms for RAE, the Refinement Acting Engine (Ghallab, Nau, and Traverso 2016). RAE uses hierarchical operational models to perform tasks in dynamically changing environments. Our planning procedure, UPOM, does a UCT-like search in the space of operational models in order a near optimal method to use for the task and context at hand. Our learning strategies acquire, from online acting experiences and/or simulated planning results, a mapping from decision contexts to method instances as well as a heuristic function to guide UPOM. Our experimental results show that UPOM and our learning strategies significantly improve RAE's performance in four test domains using two different metrics: efficiency and success ratio.

1 Introduction

The "*actor's view of automated planning and acting*" (Ghallab, Nau, and Traverso 2014) advocates a hierarchical organization of an actor's deliberation functions, with online planning throughout the acting process. Following this view, (Patra et al. 2019) proposed RAEplan, a planner for the Refinement Acting Engine (RAE) of (Ghallab, Nau, and Traverso 2016, Chap. 3), and showed on test domains that it improves RAE's efficiency and success ratio. This approach, on which we rely, is appealing for its powerful representation and seamless integration of reasoning and acting.

RAE's operational models are specified as a collection of hierarchical *refinement methods* giving alternative ways to perform tasks and react to events. A method has a *body* that can be any complex algorithm, without the restrictions of HTN methods. It may contain the usual programming constructs, as well as subtasks that need to be refined recursively, and primitive actions that query and may change the world nondeterministically. RAE uses a collection of methods for closed-loop online decision making to perform tasks and react to events. When several method instances are available for a task, RAE may respond purely reactively, relying on a domain specific heuristic. It may also call an online planner such as RAEplan, to get a more informed decision.

RAEplan offers advantages over similar planners (see Sec. 2), but it is not easily scalable for demanding real-time applications, which require an anytime procedure supporting a receding-horizon planner. We propose here a new planning algorithm for RAE, which relies on a UCT-like Monte-Carlo tree search procedure called UPOM (UCT Planner for Operational Models). It is a progressive deepening, receding-horizon anytime planner. Its scalability requires heuristics. However, while operational models are needed for acting and can be used for planning, they lead to quite complex search spaces not easily amenable to the usual techniques for domain-independent heuristics.

Fortunately, the above issue can be addressed with learning. A learning approach can be used to acquire a mapping from decision contexts to method instances, and this mapping can be used as the base case of the anytime strategy. Learning can also be used to acquire a heuristic function to guide the search. The contributions of this paper include:

- A Monte-Carlo tree search procedure that extends UCT to a search space containing *disjunction* nodes, *sequence* nodes, and *statistical sampling* nodes. The search uses progressive deepening to provide an anytime planning algorithm that can be used with different utility criteria.
- Learning strategies to acquire, from online acting experiences and/or simulated planning results, both a mapping from decision contexts to refinement methods and a heuristic evaluation function to guide UPOM.
- An approach to integrate acting, planning and learning for an actor in a dynamic environment.

These contributions are backed-up with a full implementation of RAE and UPOM and extensive experiments on four test domains, to characterize the benefits of two different learning modalities and compare UPOM to RAEplan. We do not claim any contribution on the learning techniques *per se*, but on the integration of learning, planning, and acting. We use an off-the-shelf learning library with appropriate adaptation for our experiments. The learning algorithms do not provide the operational models needed by the planner, but they do several other useful things. First, they speed up the planner's search, thereby improving the actor's efficiency. Second, they enable both the planner and the actor to find

better solutions, thereby improving the actor's success ratio. Third, they allow the human domain author to write refinement methods without needing to specify a preference ordering in which the planner or actor should try those methods.

In the following sections we discuss the related work, then introduce informally the operational model representation and RAE. UPOM procedure is detailed in Section 4. Section 5 presents three ways in which supervised learning can be integrated with RAE and UPOM. In Section 6, we describe our experiments and show the benefits of planning and learning with respect to purely reactive RAE.

2 Related work

Most of the works that extend operational models with some deliberation mechanism do not perform any kind of learning. This is true for RAEplan (Patra et al. 2019), its predecessor SeRPE (Ghallab, Nau, and Traverso 2016), and for PropicePlan (Despouys and Ingrand 1999), which brings planning capabilities to PRS (Ingrand et al. 1996). It is also true for various approaches similar to PRS and RAE, which provide refinement capabilities and hierarchical models, e.g., (Verma et al. 2005; Wang et al. 1991; Bohren et al. 2011), and for (Musliner et al. 2008; Goldman et al. 2016), which combine online planning and acting.

Works on probabilistic planning and Monte Carlo tree search, e.g., (Kocsis and Szepesvári 2006), as well as works on sampling outcomes of actions, see, e.g., FF-replan (Yoon, Fern, and Givan 2007), use descriptive models (that describe *what* actions do but not *how*) rather than operational models, and provide no integration of acting, learning, and planning.

Our approach shares some similarities with the work on planning by reinforcement learning (RL) (Kaelbling, Littman, and Moore 1996; Sutton and Barto 1998; Geffner and Bonet 2013; Leonetti, Iocchi, and Stone 2016; Garnelo, Arulkumaran, and Shanahan 2016), since we learn by acting in a (simulated) environment. However, most of the works on RL learn policies that map states to actions to be executed, and learning is performed in a descriptive model. We learn how to select refinement methods in an operational model that allows for programming control constructs. This main difference holds also with works on hierarchical reinforcement learning, see, e.g., (Yang et al. 2018; Parr and Russell 1997; Ryan 2002). Works on user-guided learning, see e.g., (Martínez, Alenyà, and Torras 2017; Martínez et al. 2017), use model based RL to learn relational models, and the learner is integrated in a robot for planning with exogenous events. Even if relational models are then mappped to execution platforms, the main difference with our work still holds: learning is performed in a descriptive model. (Jevtic et al. 2018) uses RL for user-guided learning directly in the specific case of robot motion primitives.

The approach of (Morisset and Ghallab 2008) addresses a problem similar to ours but specific to robot navigation. Several methods for performing a navigation task and its subtasks are available, each with strong and weak points depending on the context. The problem of choosing a best method for starting or pursuing a task in a given context is stated as a receding horizon planning in an MDP for which a model-explicit RL technique is proposed. Our approach is not limited to navigation tasks; it allows for richer hierarchical refinement models and is combined with a powerful Monte-Carlo tree search technique.

The Hierarchical Planning in the Now (HPN) of (Kaelbling and Lozano-Perez 2011) is designed for integrating task and motion planning and acting in robotics. Task planning in HPN relies on a goal regression hierarchized according to the level of fluents in an operator preconditions. The regression is pursued until the preconditions of the considered action (at some hierarchical level) are met by current world state, at which point acting starts. Geometric reasoning is performed at the planning level (i) to test ground fluents through procedural attachement (for truth, entailment, contradiction), and (ii) to focus the search on a few suggested branches corresponding to geometric bindings of relevant operators using heuristics called geometric suggesters. It is also performed at the acting level to plan feasible motions for the primitives to be executed. HPN is correct but not complete; however when primitive actions are reversible, interleaved planning and acting is complete. HPN has been extended in a comprehensive system for handling geometric uncertainty (Kaelbling and Lozano-Perez 2013).

Similarly, the approach of (Wolfe and Marthi 2010) also addresses the integration of task and motion planning problem. It uses an HTN approach. Motion primitives are assessed with a specific solver through sampling for cost and feasibility. An algorithm called SAHTN extends the usual HTN search with a bookkeeping mechanism to cache previously computed motions. In comparison to this work as well as to HPN, our approach does not integrate specific constructs for motion planning. However, it is more generic regarding the integration of planning and acting.

In (Colledanchise 2017; Colledanchise and Ögren 2017), Behavioural Trees (BT) are synthesized by planning. In (Colledanchise, Parasuraman, and Ögren 2019) BT are generated by genetic programming. Building the tree refines the acting process by mapping the descriptive action model onto an operational model. We integrate acting, planning, and learning directly in an operational model with the control constructs of a programming language. Moreover, we learn how to select refinement methods, a natural and practical way to specify different ways of accomplishing a task.

Learning planning domain models has been investigated along several approaches. In probabilistic planning, for example Ross et al., 2011,Katt, Oliehoek, and Amato, or 2017 learn a POMDP domain model through interactions with the environment, in order to plan by reinforcement learning or by sampling methods. In these cases, no integration with operational models and hierarchical refinements is provided.

Learning HTN methods has also been investigated. HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) learns methods given a set of actions, a set of solutions to classical planning problems, and a collection of annotated tasks. This is extended for nondeterministic domains in (Hogg, Kuter, and Muñoz-Avila 2009). (Hogg, Kuter, and Muñoz-Avila 2010) integrates HTN with reinforcement learning, and estimates the expected values of the learned methods by performing Monte Carlo updates. The methods used in RAE and UPOM are different because the operational models needed

for acting may use rich control constructs rather than simple sequences of primitives as in HTNs. At this stage, we do not learn the methods but only how to chose the appropriate one.

3 Acting with operational models

In this section, we illustrate the operational model representation and present informally how RAE works. The basic ingredients are tasks, actions and refinement methods. A method may have several instances depending on the values of its parameters. Here are a few simplified methods from one of our test domains called S&R.

Example 1. *Consider a set R of robots performing search and rescue operations in a partially mapped area. The robots' job is to find people needing help and bring them a package of supplies (medication, food, water, etc.). This domain is specified with state variables such as* $\mathsf{robotType}(r) \in \{\mathit{UAV}, \mathit{UGV}\}$, *with* $r \in R$; $\mathsf{hasSupply}(r) \in \{\top, \bot\}$; $\mathsf{loc}(r) \in L$, *a finite set of locations. A rigid relation* $\mathsf{adjacent} \subseteq L^2$ *gives the topology of the domain.*

These robots can use actions such as DETECTPERSON$(r,$ *camera) which detects if a person appears in images acquired by camera of* r, TRIGGERALARM(r, l), DROPSUPPLY(r, l), LOADSUPPLY(r, l), TAKEOFF(r, l), LAND(r, l), MOVETO(r, l), FLYTO(r, l). *They can address tasks such as:* $\mathsf{survey}(r$*,area), which makes a UAV* r *survey in sequence the locations in area,* $\mathsf{navigate}(r, l)$, $\mathsf{rescue}(r, l)$, $\mathsf{getSupplies}(r)$.

Here is a refinement method for the survey *task:*

m1-survey(r, l)
  task: survey(r, l)
  pre: robotType$(r) = \mathit{UAV}$ and loc$(r) = l$
  body:
    for all l' in neighbouring areas of l:
      moveTo(r, l')
      for *cam* in cameras(r):
        if DETECTPERSON(r, *cam*) $= \top$ then:
          if hasSupply(r) then rescue(r, l')
          else TRIGGERALARM(r, l')

The above method specifies that the UAV r flies around and captures images of all neighbouring areas of location l. If it detects a person in any of the images, it proceeds to perform a rescue task if it has supplies; otherwise it triggers an alarm event. This event is processed (by some other method) by finding the closest UGV not involved in another rescue operation and assigning to it a rescue task for l'. Before going to rescue a person, the chosen UGV replenishes its supplies via the task getSupply. *Here are two of its refinement methods:*

m1-GetSupplies(r)
  task: GetSupplies(r)
  pre: robotType$(r) = \mathit{UGV}$
  body: moveTo$(r$,loc$(BASE))$
    REPLENISHSUPPLIES(r)

m2-GetSupplies(r)
  task: GetSupplies(r)
  pre: robotType$(r) = \mathit{UGV}$
  body: $r_2 = \text{argmin}_{r\prime}\{\text{EuclideanDistance}(r, r\prime) \mid$ hasMedicine$(r\prime) = \text{TRUE}\}$
    if r_2 = None then FAIL
    else:
      moveTo$(r, loc(r_2))$
      TRANSFER(r_2, r) □

We model an acting domain as a tuple $\Sigma = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$ where S is the set of world states the actor may be in, $\mathcal{T}$ is the set of tasks and events the actor may have to deal with, $\mathcal{M}$ is the set of method templates for handling tasks or events in $\mathcal{T}$ (we get a method instance by assigning values to the free parameters of a method template), $\mathsf{Applicable}(s, \tau)$ is the set of method instances applicable to τ in state s, $\mathcal{A}$ is the set of primitive actions the actor may perform. We let $\gamma(s, a)$ be the set of states that may be reached after performing action a in state s.

Acting problem. The deliberative acting problem can be stated informally as follows: given Σ and a task or event $\tau \in \mathcal{T}$, what is the "best" method $m \in \mathcal{M}$ to perform τ in a current state s. Strictly speaking, the actor does not require a plan, i.e., an organized set of actions or a policy. It requires an online selection procedure which designates for each task or subtask at hand the best method instance for pursuing the activity in the current context.

The current context for an incoming external task τ_0 is represented via a *refinement stack* σ which keeps track of how much further RAE has progressed in refining τ_0. The refinement stack is a LIFO list of tuples $\sigma = \langle(\tau, m, i), \ldots, (\tau_0, m_0, i_0)\rangle$, where τ is the deepest current subtask in the refinement of τ_0, m is the method instance used to refine τ, i is the current instruction in body(m), with $i = \mathsf{nil}$ if we haven't yet started executing body(m), and $m = \mathsf{nil}$ if no refinement method instance has been chosen for τ yet. σ is handled with the usual stack push, pop and top functions.

When RAE addresses a task τ, it must choose a method instance m for τ. Purely reactive RAE make this choice with a domain specific heuristic, e.g., according to some a priori order of $\mathcal{M}$; more informed RAE relies on a planner and/or on learned heuristics. Once a method m is chosen, RAE progresses on performing the body of m, starting with its first step. If the current step $m[i]$ is an action already triggered, then the execution status of this action is checked. If the action $m[i]$ is still running, stack σ has to wait, RAE goes on for other pending stacks in its agenda, if any. If action $m[i]$ fails, RAE examines alternative methods for the current subtask. Otherwise, if the action $m[i]$ is completed successfully, RAE proceeds with the next step in method m.

$\mathsf{next}(\sigma, s)$ is the refinement stack resulting by performing $m[i]$ in state s, where $(\tau, m, i) = top(\sigma)$. It advances within the body of the topmost method m in σ as well as with respect to σ. If i is the last step in the body of m, the current tuple is removed from σ: method m has successfully addressed τ. In that case, if τ was a subtask of some other task, the latter will be resumed. Otherwise τ is a root task which has succeeded; its stack is removed from RAE's agenda. If

i is not the last step in m, RAE proceeds to the next step in the body of m. This step j following i in m is defined with respect to the current state s and the control instruction in step i of m, if any.

In summary, RAE follows a refinement tree as in Figure 1. At an action node it performs the action in the real world; if successful it pursues the next step of the current method, or higher up if it was its last step; if the action fails, an alternate method is tried. This goes on until a successful refinement is achieved, or until no alternate method instance remains applicable in the current state. Planning with UPOM (described in the next section) searches through this space by doing simulated sampling at action nodes.

4 UPOM: a UCT-like search procedure

UPOM performs a recursive search to find a method instance m for a task τ and a state s approximately optimal for a utility function U. It is a UCT-like (Kocsis and Szepesvári 2006) Monte Carlo tree search procedure over the space of refinement trees for τ (see Figure 1). Extending UCT to work on refinement trees is nontrivial since the search space contains three kinds of nodes (as shown in the figure), each of which must be handled in a different way.

UPOM can optimize different utility functions, such as the acting efficiency or the success ratio. In this paper, we focus on optimizing the efficiency of method instances, which is the reciprocal of the total cost, as defined in (Patra et al. 2019).

Efficiency. Let a method m for a task τ have two subtasks, τ_1 and τ_2, with cost c_1 and c_2 respectively. The efficiency of τ_1 is $e_1 = 1/c_1$ and the efficiency of τ_2 is $e_2 = 1/c_2$. The cost of accomplishing both tasks is $c_1 + c_2$, so the efficiency of m is:

$$1/(c_1 + c_2) = e_1 e_2/(e_1 + e_2). \quad (1)$$

If $c_1 = 0$, the efficiency for both tasks is e_2; likewise for $c_2 = 0$. Thus, the incremental efficiency composition is:

$$e_1 \oplus e_2 = e_2 \text{ if } e_1 = \infty, \text{ else } e_1 \text{ if } e_2 = \infty, \text{ else } e_1 e_2/(e_1 + e_2). \quad (2)$$

If τ_1 (or τ_2) fails, then c_1 is ∞, $e_1 = 0$. Thus $e_1 \oplus e_2 = 0$, meaning that τ fails with method m. Note that formula 2 is associative. When using efficiency as a utility function, we denote U(Success) = ∞ and U(Failure) = 0.

When RAE has to perform a task τ in a state s and a stack σ, it calls Select-Method (Algorithm 1) with two control parameters: n_{ro}, the number of rollouts, and d_{max}, the maximum rollout length (total number of sub-tasks and actions in a rollout). Select-Method performs an anytime progressive deepening loop calling UPOM n_{ro} times, until the rollout length reaches d_{max} or the search is interrupted. The selected method instance $\tilde{m}$ is initialized according to a heuristic h (line 1). UPOM performs recursively one Monte Carlo rollout.

When UPOM has a subtask to be refined, it looks at the set of its applicable method instances (line 4). If some method instances have not yet been tried, UPOM chooses one randomly among *Untried*, otherwise it chooses (line 5) a tradeoff between promising methods and less tried ones (Upper Confidence bound formula). UPOM simulates the execution of m_{chosen}, which may result in further refinements and actions. After the rollout is done, UPOM updates (line 7) the Q values of m_{chosen} according to its utility estimate (line 6).

When UPOM encounters an action, it nondeterministically samples one outcome of it and, if successful, continues the rollout with the resulting state. The rollout ends when there are no more tasks to be refined or the rollout length has reached d. At rollout length d, UPOM estimates the remaining utility using the heuristic h (line 3), discussed in Section 5.

The planner can be interrupted anytime, which is essential for a reactive actor in a dynamic environment. It returns the method instance $\tilde{m}$ with the best Q value reached so far. For the experimental results of this paper we used fixed values of n_{ro} and d, without progressive deepening. The latter is not needed for the offline learning simulations.

When d_{max} and n_{ro} approach infinity and when there are no dynamic events, we can prove that UPOM (like UCT) converges asymptotically to the optimal method instance for utility U. Also, the Q value for any method instance converges to its expected utility.[1]

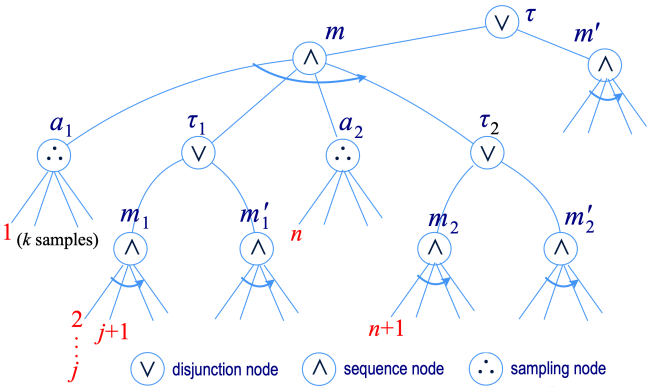

Figure 1: The space of refinement trees for a task τ. A *disjunction* node is a task followed by its applicable method instances. A *sequence* node is a method instance m followed by all the steps. A *sampling* node for an action a has the possible nondeterministic outcomes of a as its children. An example of a Monte Carlo rollout in this refinement tree is the sequence of nodes marked 1 (a sample of a_1), 2 (first step of m_1), ..., j (subsequent refinements), $j + 1$ (next step of m_1), ..., n (a sample of a_2), $n + 1$ (first step of m_2), etc.

Comparison with RAEplan. Other than UCT scoring and heuristic, UPOM and RAEplan (Patra et al. 2019) also differ in how the control parameters guide the search. RAEplan does exponentially many rollouts in the search breadth, depth and samples, whereas number of UPOM rollouts is linear in both n_{ro} and d. Select-Method has more fine-grained

[1] See proof at https://www.cs.umd.edu/~patras/UPOM_convergence_proof.pdf

Select-Method$(s, \tau, \sigma, d_{max}, n_{ro})$:
1 $\tilde{m} \leftarrow \text{argmax}_{m \in \text{Applicable}(s,\tau)} h(\tau, m, s)$
$d \leftarrow 0$
2 **repeat**
  $d \leftarrow d + 1$
  for n_{ro} *times* **do**
    UPOM (s, push($(\tau, nil, nil), \sigma$), d)
  $\tilde{m} \leftarrow \text{argmax}_{m \in M} Q_{s,\sigma}(m)$
until $d = d_{max}$ *or* searching time is over
return $\tilde{m}$

UPOM(s, σ, d):
if $\sigma = \langle\rangle$ **then** return U(Success)
$(\tau, m, i) \leftarrow \text{top}(\sigma)$
3 **if** $d = 0$ **then** return $h(\tau, m, s)$
if $m = nil$ *or* $m[i]$ *is a task* τ' **then**
  if $m = nil$ **then** $\tau' \leftarrow \tau$ # for the first task
  if $N_{s,\sigma}(\tau')$ *is not initialized yet* **then**
4   $M' \leftarrow \text{Applicable}(s, \tau')$
    if $M' = 0$ **then** return U(Failure)
    $N_{s,\sigma}(\tau') \leftarrow 0$
    for $m' \in M'$ **do**
      $N_{s,\sigma}(m') \leftarrow 0$; $Q_{s,\sigma}(m') \leftarrow 0$
  Untried $\leftarrow \{m' \in M' | N_{s,\sigma}(m') = 0\}$
  if *Untried* $\neq \emptyset$ **then**
    $m_{chosen} \leftarrow$ random selection from *Untried*
5 **else** $m_{chosen} \leftarrow \text{argmax}_{m \in M'} \phi(m, \tau')$
6 $\lambda \leftarrow$
   UPOM$(s, \text{push}((\tau', m, 1), \text{next}(\sigma, s)), d - 1)$
7 $Q_{s,\sigma}(m_{chosen}) \leftarrow$
   $\frac{N_{s,\sigma}(m_{chosen}) \times Q_{s,\sigma}(m_{chosen}) + \lambda}{1 + N_{s,\sigma}(m_{chosen})}$
  $N_{s,\sigma}(m_{chosen}) \leftarrow N_{s,\sigma}(m_{chosen}) + 1$
  return λ
if $m[i]$ *is an assignment* **then**
  $s' \leftarrow$ state s updated according to $m[i]$
  return UPOM$(s', \text{next}(\sigma, s'), d)$
if $m[i]$ *is an action* a **then**
  $s' \leftarrow \text{Sample}(s, a)$
  if $s' =$ *failed* **then** return U(Failure)
8 **else** return
    $U(s, a, s') \oplus$ UPOM$(s', \text{next}(\sigma, s'), d - 1)$

Algorithm 1: UPOM performs one rollout recursively down the refinement tree until depth d for stack σ. For $C > 0$,
$\phi(m, \tau) = Q_{s,\sigma}(m) + C\sqrt{\log N_{s,\sigma}(\tau)/N_{s,\sigma}(m)}$.

control of the tradeoff between running time and quality of evaluation, since a change to n_{ro} or d changes the running time by only a linear amount.

5 Integrating Learning, Planning and Acting

Purely reactive RAE chooses a method instance for a task using a domain specific heuristic. RAE can be combined with UPOM in a receding horizon manner: whenever a task or a subtask needs to be refined, RAE uses the approximately optimal method instance found by UPOM.

Finding efficient domain-specific heuristics is not easy to do by hand. This motivated us to try learning such heuristics automatically by running UPOM offline in simulation over numerous cases. For this work we relied on a neural network approach, using both linear and rectified linear unit (ReLU) layers. However, we suspect that other learning approaches, e.g., SVMs, might have provided comparable results.

We have two strategies for learning neural networks to guide RAE and UPOM. The first one, Learnπ, learns a policy which maps a context defined by a task τ, a state s, and a stack σ, to a refinement method m in this context, to be chosen by RAE when no planning can be performed. To simplify the learning process, Learnπ learns a mapping from contexts to methods, not to method instances, with all parameters instantiated. At acting time, RAE chooses randomly among all applicable instances of the learned method for the context at hand. The second learning strategy, LearnH, learns a heuristic evaluation function to be used by UPOM.

Learning to choose methods (Learnπ)

The Learnπ learning strategy consists of the following four steps, which are schematically depicted in Figure 2.

Step 1: Data generation. Training is performed on a set of data records of the form $r = ((s, \tau), m)$, where s is a state, τ is a task to be refined and m is a method for τ. Data records are obtained by making RAE call the planner offline with randomly generated tasks. Each call returns a method instance m. We tested two approaches (the results of the tests are in Section 6):

- Learnπ-1 adds $r = ((s, \tau), m)$ to the training set if RAE succeeds with m in accomplishing τ while acting in a dynamic environment.
- Learnπ-2 adds r to the training set irrespective of whether m succeeded during acting.

Step 2: Encoding. The data records are encoded according to the usual requirements of neural net approaches. Given a record $r = ((s, \tau), m)$, we encode (s, τ) into an input-feature vector and encode m into an output label, with the refinement stack σ omitted from the encoding for the sake of simplicity.[2] Thus the encoding is

$$((s, \tau), m) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau], w_m), \tag{3}$$

with w_s, w_τ and w_m being One-Hot representations of s, τ, and m. The encoding uses an N-dimensional One-Hot vector representation of each state variable, with N being the maximum range of any state variable. Thus if every $s \in \Xi$ has V state-variables, then s's representation w_s is $V \times N$ dimensional. Note that some information may be lost in this step due to discretization.

Step 3: Training. Our multi-layer perceptron (MLP) nn_π consists of two linear layers separated by a ReLU layer to account for non-linearity in our training data. To learn and

[2]Technically, the choice of m depends partly on σ. However, since σ is a program execution stack, including it would greatly increase the input feature vector's complexity, and the neural network's size and complexity.

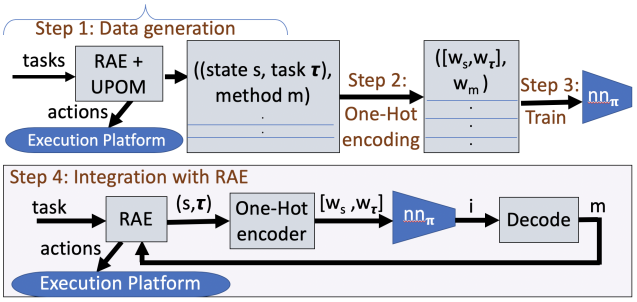

Figure 2: A schematic diagram for the Learnπ strategy.

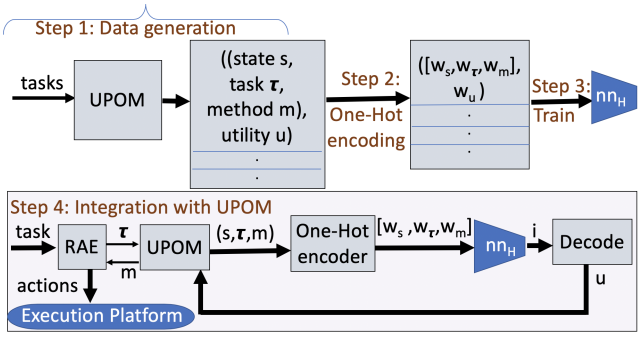

Figure 3: A schematic diagram for the LearnH strategy.

classify $[w_s, w_\tau]$ by refinement methods, we used a SGD (Stochastic Gradient Descent) optimizer and the Cross Entropy loss function. The output of nn_π is a vector of size $|M|$ where M is the set of all refinement methods in a domain. Each dimension in the output represents the degree to which a specific method is optimal in accomplishing τ.

Step 4: Integration in RAE. We have RAE use the trained network nn_π to choose a refinement method whenever a task or sub-task needs to be refined. Instead of calling the planner, RAE encodes (s, τ) into $[w_s, w_\tau]$ using Equation 3. Then, m is chosen as

$$m \leftarrow Decode(\text{argmax}_i(nn_\pi([w_s, w_\tau])[i])),$$

where $Decode$ is a one-one mapping from an integer index to a refinement method.

Learning a heuristic function (LearnH)

The LearnH strategy tries to learn an estimate of the utility u of accomplishing a task τ with a method m in state s. One difficulty with this is that u is a real number. In principle, an MLP could learn the u values using either regression or classification. To our knowledge, there is no rule to choose between the two; the best approach depends on the data distribution. Further, regression can be converted into classification by binning the target values if the objective is discrete. In our case, we don't need an exact utility value but only need to compare utilities to choose a method. Experimentally, we observed that classification performed better than regression. We divided the range of utility values into K intervals. By studying the range and distribution of utility values, we chose K and the range of each interval such

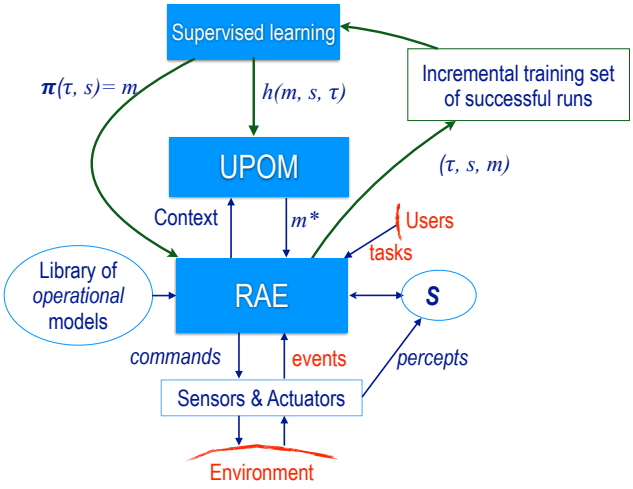

Figure 4: Integration of Acting, Planning and Learning.

that the intervals contained approximately equal numbers of data records. LearnH learns to predict $interval(u)$, i.e., the interval in which u lies. The steps of LearnH (see Figure 3) are:

Step 1: Data generation. We generate data records in a similar way as in the Learnπ strategy, with the difference that each record r is of the form $((s, \tau, m), u)$ where u is the estimated utility value calculated by UPOM.

Step 2: Encoding. In a record $r = ((s, \tau, m), u)$, we encode (s, τ, m) into an input-feature vector using N-dimensional One-Hot vector representation, omitting σ for the same reasons as before. If $interval(u)$ is as described above, then the encoding is

$$((s, \tau, m), interval(u)) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau, w_m], w_u) \quad (4)$$

with w_s, w_τ, w_m and w_u being One-Hot representations of s, τ, m and $interval(u)$.

Step 3: Training. LearnH's MLP nn_H is same as Learnπ's, except for the output layer. nn_H has a vector of size K as output where K is the number of intervals into which the utility values are split. Each dimension in the output of nn_H represents the degree to which the estimated utility lies in that interval.

Step 4: Integration in RAE. There are two ways to use nn_H with UPOM. One is for RAE to call the planner with a limited rollout length d, giving UPOM the following heuristic function to estimate a rollout's remaining utility:

$$h(\tau, m, s) \leftarrow Decode(\text{argmax}_i(nn_H([w_s, w_\tau, w_m])[i])),$$

where $[w_s, w_\tau, w_m]$ is the encoding of (τ, m, s) using Equation 4, and $Decode$ is a one-one mapping from a utility interval to its mid-point. The other way to use nn_H is to estimate the heuristic function in line 1 of Algorithm 1.

Incremental online learning

The proposed approach supports incremental online learning, although not yet implemented. The initialization can

be performed either by running RAE+UPOM online with $d = \infty$ without a heuristic, or with an initial heuristic from offline learning on simulated data. The online acting, planning and incremental learning is performed as follows:

- Augment the training set by recording successful methods and u values; train the models using Learnπ and LearnH with Z records, and then switch RAE to use either Learnπ alone when no search time is available, or UPOM with current heuristic h and finite d_{max} when there is some time available for planning.
- Repeat the above steps every X runs (or on idle periods) using the most recent Z training records (for Z about a few thousands) to improve the learning on both LearnH and Learnπ.

6 Experimental Evaluation

Domains. We have implemented and tested our framework on four simulated acting and planning domains (see Table 1).

Acting & planning domain	$\|\mathcal{T}\|$	$\|\mathcal{M}\|$	$\|\mathcal{A}\|$	Exogenous events	Dead ends	Sensing	Agent collaboration	Parallel tasks
Fetch	7	10	9	✓	✓	✓	–	✓
Explore	9	17	14	✓	✓	–	✓	✓
Nav	6	9	10	–	–	✓	✓	✓
S&R	8	16	14	✓	✓	✓	✓	✓

Table 1: Properties of our domains

In Fetch, several robots are collecting objects of interest. The robots are rechargeable and may carry the charger with them. They can't know where objects are, unless they do a sensing action at the object's location. They must search for an object before collecting it. A task reaches a dead end if a robot is far away from the charger and runs out of charge. While collecting objects, robots may have to attend to some emergency events happening in certain locations.

The Nav domain has several robots trying to move objects from one room to another in an environment with a mixture of spring doors (which close unless they're held open) and ordinary doors. A robot can't simultaneously carry an object and hold a spring door open, so it must ask for help from another robot. A free robot can be the helper. The type of each door isn't known to the robots in advance.

The S&R domain extends the search and rescue setting of Example 1 with UAVs surveying a partially mapped area and finding injured people in need of help. UGVs gather supplies, such as, medicines, and go to rescue the person. Exogenous events are weather conditions and debris in paths.

In Explore, several chargeable robots with different capabilities (UGVs and UAVs) explore a partially known terrain and gather information by surveying, screening, monitoring. They need to go back to the base regularly to deposit data or to collect a specific equipment. Appearance of animals simulate exogenous events.

Fetch, Nav and S&R have sensing actions. Fetch, S&R and Explore can have dead-ends, but Nav has none.[3]

[3]Full code is online at ⟨https://bitbucket.org/sunandita/upom/⟩.

Evaluation of planning with UPOM

To test whether planning with UPOM is beneficial for RAE, we compare its performance with purely reactive RAE and with the planner RAEplan (Patra et al. 2019) in our four simulated domains.[4] We configured UPOM to optimize the efficiency as its utility function, the same as RAEplan.

We created a test suite of 50 randomly generated problems for each domain. Each test problem consists of one to three tasks which arrive randomly chosen time points in RAE's input stream. For each test problem, we used a maximum time limit of 5 minutes for each call to the planner. We set n_{ro}, the maximum number of UCT rollouts of UPOM to be 1000, with $d_{max} = \infty$ in each rollout.[5] We ran each problem 20 times, to cover sufficiently the non-deterministic effects of actions. We ran the tests on a 2.8 GHz Intel Ivy Bridge processor.

Figure 5 shows the computation time for a single run of a task, averaged across all domains, an average of about 10^4 runs (4 domains $\times$ 50 problems/domain $\times$ 1-2 tasks/problem $\times$ 20 runs/task). We observe that RAE with UPOM runs more than twice as fast as RAE with RAEplan.

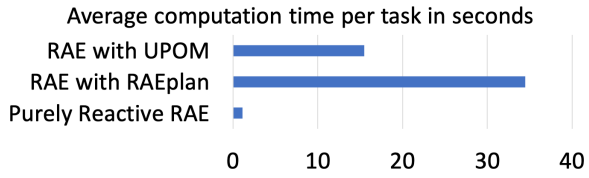

Figure 5: Computation time in seconds for a single run of a task, for RAE with and without the planners, averaged over all four domains, an average of about 10^4 runs. If RAE were running in the real world, the total time would be the computation time plus the time needed to perform the actions.

Efficiency. The average efficiency values for all four domains are presented in Figure 6, with the error bars showing a 95% confidence interval.We conclude that RAE with UPOM is more efficient than purely reactive RAE and RAE with RAEplan with 95% confidence in all four domains.

Success ratio. The success ratio is the proportion of incoming tasks successfully accomplished in each domain. Although RAEplan and UPOM both were configured to optimize efficiency rather than success, the success ratio is useful as a measure of robustness and is not directly proportional to efficiency. Suppose m_1 is always successful but has a very large cost, whereas m_2 sometimes fails but costs very

[4]We didn't compare UPOM with any non-hierarchical planning algorithms because it would be very difficult to perform a fair comparison, as discussed in (Kambhampati 2003).

[5]The table of N and Q is sparse in the lower parts of the search tree but pretty dense at the top, as in the standard UCT algorithm. Each time RAE wants to make a decision, UPOM reruns the MCT search starting at the current node, so there's no danger of the table becoming more sparse as RAE proceeds. $N_{s,\sigma}(m)$ is approximately in the range [50,200] at the top. In our experimental domains, the upper part of the search tree has a greater influence on the optimality, so $n_{ro} = 1000$ is found to be sufficient.

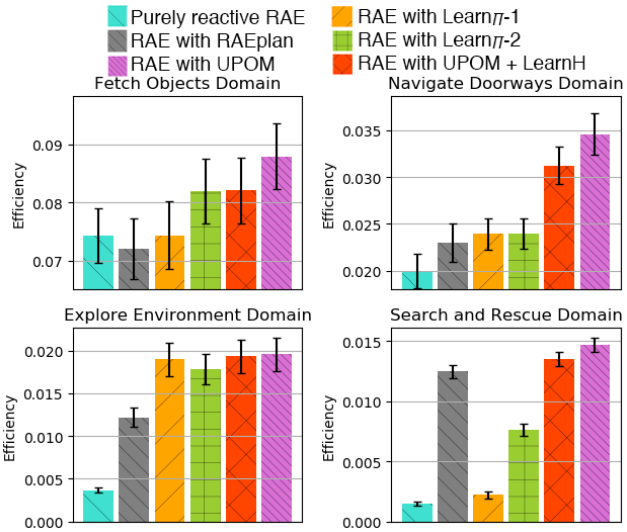

Figure 6: Efficiency (1/cost) for four domains each with six different ways of acting: purely reactive RAE, RAE calling RAEplan, RAE using the policy and heuristic learned by Learnπ and LearnH, and RAE using UPOM.

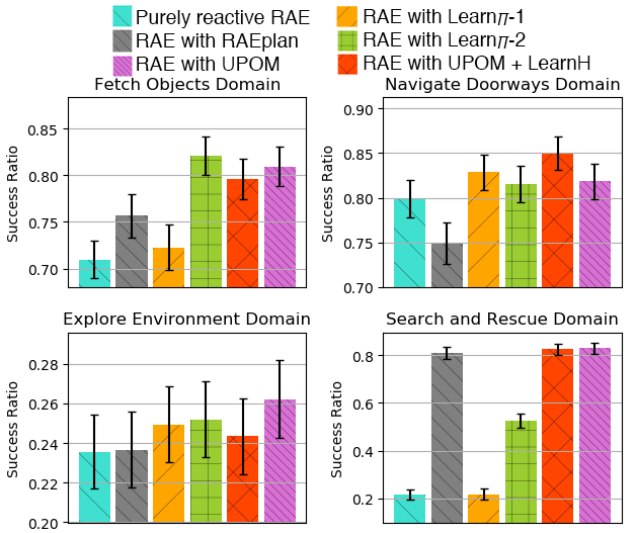

Figure 7: Success ratio (number of successful tasks / total number of incoming tasks) for four domains each with six different ways of acting: purely reactive RAE, RAE calling RAEplan, RAE using the policy and heuristic learned by Learnπ and LearnH, and RAE using UPOM.

little when it works. Then m_1 will have a higher probability of success, but m_2 will have higher expected efficiency.

Figure 7 shows RAE's success ratio both with and without the planners. We observe that planning with UPOM outperforms purely reactive RAE in Fetch and S&R with 95% confidence in terms of success ratio, whereas in Explore and Nav it does so with 85% confidence. Also, planning with UPOM outperforms planning with RAEplan in Fetch and Nav domains with a 95% confidence; in Explore domain with 85% confidence. The success ratio achieved is similar for RAEplan and UPOM in the S&R domain.

Asymptotically, UPOM and RAEplan should have near-equivalent efficiency and success ratio metrics. They differ because neither are able to traverse the entire search space due to computational constraints. Our experiments on simulated environments suggest that UPOM is more effective than RAEplan when called online with real-time constraints.

Evaluation of the learning benefits

We obtained data records for each domain by randomly generating incoming tasks and then running RAE with UPOM. The number of randomly generated tasks in Fetch, Explore, Nav and S&R domains are 123, 189, 132 and 96 respectively. We save the data records according to the Learnπ-1, Learnπ-2 and LearnH strategies, and encode them using the One-Hot schema. We divide the training set randomly into two parts: 80% for training and 20% for validation to avoid overfitting on the training data.

The training and validation losses decrease and the accuracies increase with increase in the number of training epochs (see Figure 8). The accuracy of Learnπ is measured by checking whether the refinement method instance returned by UPOM matches the template predicted by the MLP nn_π, whereas the accuracy of LearnH is measured by checking whether the efficiency estimated by UPOM lies in the interval predicted by nn_H. We chose the learning rate to be in the range $[10^{-3}, 10^{-1}]$. Learning rate is a scaling factor that controls how weights are updated in each training epoch via backpropagation. Table 2 summarizes the training set size, the number of input features and outputs after data records are encoded using the One-Hot schema, number of training epochs for the three different learning strategies. In the LearnH learning strategy, we define the number of output intervals K from the training data such that each interval has an approximately equal number of data records. The final validation accuracies for Learnπ are 65%, 91%, 66% and 78% in the domains Fetch, Explore, S&R and Nav respectively. The final validation accuracies for LearnH are similar but slightly lower. The accuracy values may possibly improve with more training data and encoding the refinement stacks as part of the input feature vectors.

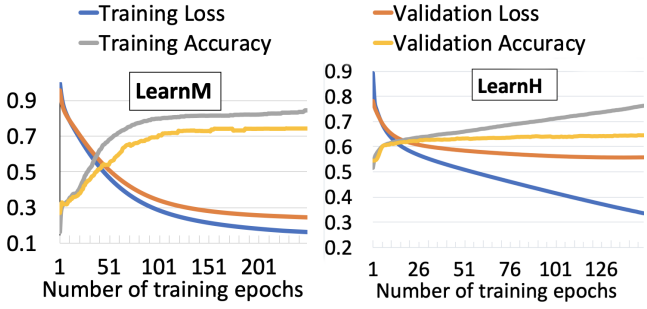

Figure 8: Training and validation results for Learnπ and LearnH, averaged over all domains.

To test the learning strategies (presented in Section 5) we

Domain	Training Set Size			#(input features)		Training epochs		#(outputs)	
	LM-1	LM-2	LH	LM-1 and -2	LH	LM-1 and -2	LH	LM-1 and -2	LH
Fetch	262	508	1084	97	104	430	250	10	100
Explore	2391	6883	10503	182	204	1000	250	17	200
Nav	1686	5331	16251	126	144	750	150	9	75
S&R	250	634	3542	330	401	225	250	16	10

Note:
LM-1 = Learnπ-1
LM-2 = Learnπ-2
LH = LearnH

Table 2: The size of the training set, number of input features and outputs, and the number of training epochs for three different learning strategies: Learnπ-1, Learnπ-2, and LearnH.

have RAE use nn_π-1, nn_π-2 (the models learned by Learnπ-1 and Learnπ-2) without a planner, and RAE use UPOM + nn_H (the model learned by LearnH), and measure the efficiency and success ratio. We use the same test suite as in our experiments with RAE using RAEplan and UPOM, and do 20 runs for each test problem. When using UPOM with nn_H, we set d_{max} to 5 and n_{ro} to 50, which has $\sim$88% less computation time compared to using UPOM with infinite d_{max} and $n_{ro} = 1000$. Since the learning happens offline, there is almost no computational overhead when RAE uses the learned models for online acting.

Efficiency. Figure 6 shows that RAE with UPOM + nn_H is more efficient than both purely reactive RAE and RAE with RAEplan in three domains (Explore, S&R and Nav) with 95% confidence, and in the Fetch domain with 90% confidence. The efficiency of RAE with nn_π-1 and nn_π-2 lies in between RAE with RAEplan and RAE with UPOM + nn_H, except in the S&R domain, where they perform worse than RAE with RAEplan but better than purely reactive RAE. This is possibly because the refinement stack plays a major role in the resulting efficiency in the S&R domain.

Success ratio. In our experiments, UPOM optimizes for the efficiency, not the success ratio. It is however interesting to see how we perform for this criteria even when it is not the chosen utility function. In Figure 7, we observe that RAE with UPOM + nn_H outperforms purely reactive RAE and RAE with RAEplan in three domains (Fetch, Nav and S&R) with 95% confidence in terms of success ratio. In Explore, there is only slight improvement in success-ratio possibly because of high level of non-determinism in the domain's design.

In most cases, we observe that RAE does better with nn_π-2 than with nn_π-1. Recall that the training set for Learnπ-2 is created with all method instances returned by UPOM regardless of whether they succeed while acting or not, whereas Learnπ-1 leaves out the methods that don't. This makes Learnπ-1's training set much smaller. In our simulated environments, the acting failures due to totally random exogenous events don't have a learnable pattern, and a smaller training set makes Learnπ-1's performance worse.

7 Conclusion

In this paper, we have presented algorithms to guide the acting procedure RAE on what methods to use. One is the UPOM procedure, which uses a search strategy inspired by the UCT algorithm, extended to operate in a more complicated search space. The others are learning functions: Learnπ, which learns a mapping from a task in a given context to a good method, and LearnH, which provides a domain independent strategy to learn a heuristic function in a task-based hierarchical operational model framework.

Recall that RAE can either run purely reactively, or can get advice from an online planner. In place of the planner, we experimented using UPOM, RAEplan (Patra et al. 2019), and the models learned by LearnH and Learnπ, on four simulated planning-and-acting domains. Our results show with 95% confidence that when RAE uses either UPOM or the LearnH model, it accomplishes tasks more efficiently than when it uses RAEplan or runs reactively.

Furthermore, with 90% confidence, when RAE uses UPOM and/or the functions LearnH and Learnπ, its success ratio (proportion of incoming tasks accomplished successfully) is higher than when it runs reactively, even though the success ratio was not the utility function UPOM and the learners were trying to optimize.

Future work

A significant limitation of Learnπ and LearnH is that they give method to use, not a method *instance*. Thus if they advise RAE to use method m, and several different instances of m are applicable in the current context, RAE chooses among them randomly. In future work, we may extend Learnπ and LearnH to give advice about method instances. Our final validation accuracy for the learning strategies is around 70%, which shows a large scope for improvement.

UPOM, just like RAEplan, uses efficiency (1/cost) as the utility function to optimize. UPOM can easily work with other utility functions. Theoretically, LearnH should also be able to estimate any utility function, but the properties of the utility function may affect how hard it is to learn, and we hope to test this empirically in our future work.

Currently, LearnH and Learnπ learn offline, by calling UPOM with randomly generated tasks. In future work, we intend to develop online learning to update the learned models while RAE is acting.

Acknowledgement This work has been supported in part by NRL grant N00173191G001. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Bohren, J.; Rusu, R. B.; Jones, E. G.; Marder-Eppstein, E.; Pantofaru, C.; Wise, M.; Mösenlechner, L.; Meeussen, W.; and Holzer, S. 2011. Towards autonomous robotic butlers: Lessons learned with the PR2. In *ICRA*, 5568–5575.

Colledanchise, M., and Ögren, P. 2017. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Trans. Robotics* 33(2):372–389.

Colledanchise, M.; Parasuraman, R.; and Ögren, P. 2019. Learning of behavior trees for autonomous agents. *IEEE Trans. Games* 11(2):183–189.

Colledanchise, M. 2017. *Behavior Trees in Robotics*. Ph.D. Dissertation, Department of Computer Science, KTH, Stockholm, Sweden.

Despouys, O., and Ingrand, F. 1999. Propice-Plan: Toward a unified framework for planning and execution. In *ECP*, 278–293.

Garnelo, M.; Arulkumaran, K.; and Shanahan, M. 2016. Towards deep symbolic reinforcement learning. *CoRR* abs/1609.05518.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence* 208:1–17.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Goldman, R. P.; Bryce, D.; Pelican, M. J. S.; Musliner, D. J.; and Bae, K. 2016. A hybrid architecture for correct-by-construction hybrid planning and control. In *NASA Formal Methods Symp.*, 388–394.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *IJCAI*, 1708–1714.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2010. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In *AAAI*.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: learning htns with minimal additional knowledge engineering required. In *AAAI*, 950–956.

Ingrand, F.; Chatilla, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, 43–49.

Jevtic, A.; Colomé, A.; Alenyà, G.; and Torras, C. 2018. Robot motion adaptation through user intervention and reinforcement learning. *Pattern Recognition Letters* 105:67–75.

Kaelbling, L. P., and Lozano-Perez, T. 2011. Hierarchical task and motion planning in the now. In *ICRA*, 1470–1477.

Kaelbling, L. P., and Lozano-Perez, T. 2013. Integrated task and motion planning in belief space. *International J. Robotics Research* 32:1194–1227.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *JAIR* 4:237–285.

Kambhampati, S. 2003. Are we comparing Dana and Fahiem or SHOP and TLPlan? A critique of the knowledge-based planning track at ICP. http://rakaposhi.eas.asu.edu/kbplan.pdf.

Katt, S.; Oliehoek, F. A.; and Amato, C. 2017. Learning in pomdps with Monte Carlo tree search. In *ICML*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, volume 6, 282–293.

Leonetti, M.; Iocchi, L.; and Stone, P. 2016. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence* 241:103–130.

Martínez, D. M.; Alenyà, G.; and Torras, C. 2017. Relational reinforcement learning with guided demonstrations. *Artificial Intelligence* 247:295–312.

Martínez, D. M.; Alenyà, G.; Ribeiro, T.; Inoue, K.; and Torras, C. 2017. Relational reinforcement learning for planning with exogenous effects. *J. Mach. Learn. Res.* 18:78:1–78:44.

Morisset, B., and Ghallab, M. 2008. Learning how to combine sensory-motor functions into a robust behavior. *Artificial Intelligence* 172(4-5):392–412.

Musliner, D. J.; Pelican, M. J. S.; Goldman, R. P.; Krebsbach, K. D.; and Durfee, E. H. 2008. The evolution of circa, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symp. Architectures for Intelligent Theory-Based Agents*, 43–48.

Parr, R., and Russell, S. J. 1997. Reinforcement learning with hierarchies of machines. In *NIPS*.

Patra, S.; Traverso, P.; Ghallab, M.; and Nau, D. 2019. Acting and planning using operational models. In *AAAI*, 7691–7698.

Ross, S.; Pineau, J.; Chaib-draa, B.; and Kreitmann, P. 2011. A bayesian approach for learning and planning in partially observable Markov decision processes. *J. Machine Learning Research* 12:1729–1770.

Ryan, M. R. K. 2002. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *ICML*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Verma, V.; Estlin, T.; Jónsson, A. K.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *Proc. International Symp. on Artificial Intelligence, Robotics and Automation in Space*, 1–8.

Wang, F. Y.; Kyriakopoulos, K. J.; Tsolkas, A.; and Saridis, G. N. 1991. A Petri-net coordination model for an intelligent mobile robot. *IEEE Trans. Systems, Man, and Cybernetics* 21(4):777–789.

Wolfe, J., and Marthi, B. 2010. Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling*, 254–257.

Yang, F.; Lyu, D.; Liu, B.; and Gustafson, S. 2018. PEORL: integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. In *IJCAI*.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *ICAPS*, 352–359.