Deliberative Acting, Planning and Learning with Hierarchical Operational Models

Sunandita Patra[a,*], James Mason[a], Malik Ghallab[b], Dana Nau[a], Paolo Traverso[c]

[a] *Dept. of Computer Science and Inst. for Systems Research, University of Maryland, College Park, Maryland, USA*
[b] *LAAS-CNRS, Toulouse, France*
[c] *FBK-ICT, Povo-Trento, Italy*

Abstract

In AI research, synthesizing a plan of action has typically used *descriptive models* of the actions that abstractly specify *what* might happen as a result of an action, and are tailored for efficiently computing state transitions. However, executing the planned actions has needed *operational* models, in which rich computational control structures and closed-loop online decision-making are used to specify *how* to perform an action in a nondeterministic execution context, react to events and adapt to an unfolding situation. *Deliberative actors*, which integrate acting and planning, have typically needed to use both of these models together—which causes problems when attempting to develop the different models, verify their consistency, and smoothly interleave acting and planning.

As an alternative, we define and implement an integrated acting-and-planning system in which both planning and acting use the same operational models. These rely on hierarchical task-oriented *refinement methods* offering rich control structures. The acting component, called Reactive Acting Engine (RAE), is inspired by the well-known PRS system. At each decision step, RAE can get advice from a planner for a near-optimal choice with respect to an utility function. The anytime planner uses a UCT-like Monte Carlo Tree Search procedure, called UPOM, whose rollouts are simulations of the actor's operational models. We also present learning strategies for use with RAE and UPOM that 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. We demonstrate the asymptotic convergence of UPOM towards optimal methods in static domains, and show experimentally that UPOM and the learning strategies significantly improve the acting efficiency and robustness.

Keywords: Acting and Planning, Operational Models, Hierarchical Actor,

*Corresponding author

Email addresses: patras@umd.edu (Sunandita Patra), james.mason@jpl.nasa.gov (James Mason), malik@laas.fr (Malik Ghallab), nau@umd.edu (Dana Nau), traverso@fbk.eu (Paolo Traverso)

1. Introduction

Consider a system required to autonomously perform a diversity of tasks in varying dynamic environments. Such a system, referred to as an "*actor*" (following [36]), needs to be reactive and to act in a purposeful deliberative way. This requirements are usually addressed by combining a reactive approach and a plan-based approach using, respectively, *operational models* and *descriptive models* of actions.

Descriptive models specify *what* might happen as a result of an action. They are tailored to efficiently search a huge state space by representing state transitions with abstract preconditions and effects. This representation, inherited from the early STRIPS system [28], refined into SAS$^+$ [3, 57], the PDDL languages [78, 30, 31, 42] and their nondeterministic and probabilistic variants, e.g., PPDDL [123] and RDDL [101], is used by most planning algorithms.

Operational models specify *how* to perform an action. They are designed to take into account an elaborate context about ongoing activities, react to events and adapt to an unfolding situation. Using several computational paradigms with rich control structures (e.g., procedures, rules, automata, Petri nets), they allow for closed-loop online decision-making. They have been designed into a diversity of languages, such as PRS, RAPS, TDL, Plexil, Golex, or RMPL (see the survey [53, Section 4]).

The combination of descriptive and operational models raises several problems. First, it is difficult to take into account with two separate models the highly interconnected reasoning required between planning and deliberative acting. Second, the mapping between descriptive and operational models is very complex. A guarantee of the consistency of this mapping is required in safety-critical applications, such as self-driving cars [44], collaborative robots working directly with humans [114], or virtual coaching systems to help patients with chronic diseases [98]. However, to verify the consistency between the two different models is difficult (e.g., see the work on formal verification of operational models such as PRS-like procedures, using model checking and theorem proving [105, 9]). Finally, modeling is always a costly bottleneck; reducing the corresponding effort is beneficial in most applications.

Therefore, it is desirable to have a single representation for both acting and planning. If such a representation were solely descriptive, it wouldn't provide sufficient functionality. Instead, the planner needs to be able to reason directly with the actor's operational models.

This paper describes an integrated planning and acting system in which both planning and acting use the actor's operational models.[1] The acting component, called *Refinement Acting Engine* (RAE), is inspired by the well-known

[1] Prior results about this approach have been presented in [37, 93, 92, 94]. The last paragraph of Section 2 describes what the current paper adds to that work.

PRS system [52]. RAE uses a hierarchical task-oriented operational representation in which an expressive, general-purpose language offers rich programming control structures for online decision-making. A collection of *hierarchical refinement methods* describes alternative ways to handle *tasks* and react to *events*. A method can be any complex algorithm, including *subtasks*, which need to be refined recursively, and primitive *actions*, which query and change the world *nondeterministically*. We assume that methods are manually specified (approaches for learning method bodies are discussed in Section 8).

Rather than behaving purely reactively, RAE interacts with a planner. To choose how best to refine tasks, the planner uses a Monte Carlo Tree Search procedure, called UPOM, which assesses the utility of possible alternatives and finds an approximately optimal one. Two utility functions are proposed reflecting the acting *efficiency* (reciprocal of the cost) and *robustness* (success ratio). Planning is performed using the constructs and steps as of the operational model, except that methods and actions are executed in a simulated world rather than the real one. When a refinement method contains an action, UPOM takes samples of its possible outcomes, using either a domain-dependent generative simulator, when available, or a probability distribution of its effects.

UPOM is used by RAE as a progressive deepening, receding-horizon anytime planner. Its scalability requires heuristics. However, operational models lead to quite complex search spaces not easily amenable to the usual techniques for domain-independent heuristics. Fortunately, this issue can be addressed with a learning approach to acquire a mapping from decision contexts to method instances; this mapping provides the base case of the anytime algorithm. Learning can also be used to acquire a heuristic function to prune deep Monte Carlo rollouts. We use an off-the-shelf learning library with appropriate adaptation for our experiments. Our contribution is not on the learning techniques *per se*, but on the integration of learning, planning, and acting. The learning algorithms do not provide the operational models needed by the planner, but they do several other useful things. They speed up the online planning search allowing for an anytime procedure. Both the planner and the actor can find better solutions, thereby improving the actor's performance. The human domain author can write refinement methods without needing to specify a preference ordering in which the planner or actor should try instances of those methods.

Following a discussion of the state of the art in Section 2, Section 3 describes the actor's architecture and the hierarchical operational model representation. In Sections 4, 5, and 6, respectively, we present the acting component RAE, the planning component UPOM, and, the learning procedures for RAE and UPOM. We provide an experimental evaluation of the approach in Section 7, followed by a discussion and a conclusion. The planner's asymptotic convergence to optimal choices is detailed in Appendix A. The operational model for a search and rescue domain is described in Appendix B. A table of notation is presented in Appendix C. The code for the algorithms and test domains is available online.

2. Related Work

To our knowledge, no previous approach has proposed the integration of planning, acting and learning directly with operational models. In this section, we first discuss the relations with systems for acting, including those approaches that provide some (limited) deliberation mechanism. We then discuss the main differences of our approach with systems for on-line acting and planning, such as RMPL and Behaviour Trees. We continue the section by comparing our work with HTNs and with different approaches based on MDP and Monte Carlo search. We discuss the relation with work on integrating planning and execution in robotics, the work on approaches based on temporal logics (including situation calculus and BDIs), and the approach to planning by reinforcement learning. We conclude the section with a relation with our prior work on the topic of this paper.

(Deliberative) Acting. Our acting algorithm and operational models are based on the *Refinement Acting Engine*, RAE algorithm [37, Chapter 3], which in turn is inspired from PRS [52]. If RAE needs to choose among several eligible refinement method instances for a given task or event, it makes the choice without trying to plan ahead. This approach has been extended with some planning capabilities in PropicePlan [18] and SeRPE [37]. Unlike our approach, those systems model actions as classical planning operators; they both require the action models and the refinement methods to satisfy classical planning assumptions of deterministic, fully observable and static environments, which are not acceptable assumptions for most acting systems. Moreover, these works do not perform any kind of learning.

Various acting approaches similar to PRS and RAE have been proposed, e.g., [29, 107, 109, 5, 83, 85]. Some of these have refinement capabilities and hierarchical models, e.g., [115, 117, 7]. While such systems offer expressive acting environments, e.g., with real time handling primitives, none of them provides the ability to plan with the operational models used for acting, and thus cannot integrate acting and planning as we do. Most of these systems do not reason about alternative refinements, and do not perform any kind of learning.

Online Acting and Planning. Online planning and acting is addressed in many approaches, e.g., [84, 39, 38], but their notion of "online" is different from ours. For example, in [84], the old plan is executed repeatedly in a loop while the planner synthesizes a new plan, which isn't installed until planning has been finished. In UPOM, hierarchical task refinement is simulated to do the planning, and can be interrupted anytime when RAE needs to act.

The Reactive Model-based Programming Language (RMPL) [51] is a comprehensive CSP-based approach for temporal planning and acting, which combines a system model with a control model. The system model specifies nominal as well as failure state transitions with hierarchical constraints. The control model uses standard reactive programming constructs. RMPL programs are

transformed into an extension of Simple Temporal Networks with symbolic constraints and decision nodes [118, 14]. Planning consists in finding a path in the network that meets the constraints. RMPL has been extended with error recovery, temporal flexibility, and conditional execution based on the state of the world [21]. Probabilistic RMPL is introduced in [102, 73] with the notions of weak and strong consistency, as well as uncertainty for contingent decisions taken by the environment or another agent. The acting system adapts the execution to observations and predictions based on the plan. RMPL and subsequent developments have been illustrated with a service robot which observes and assists a human. Our approach does not handle time; it focuses instead on hierarchical decomposition with Monte Carlo rollout and sampling.

Behavior trees (BT) [11, 12, 13] can also respond reactively to contingent events that were not predicted. In [11, 12], BT are synthesized by planning. In [13] 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 and method instances in a natural and practical way to specify different ways of accomplishing a task.

Hierarchical Task Networks. Our methods are significantly different from those used in HTNs [87]: to allow for the operational models needed for acting, we use rich control constructs rather than simple sequences of primitives. Learning HTN methods has also been investigated. HTN-MAKER [49] 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 [47]. [48] integrates HTN with Reinforcement Learning (RL), and estimates the expected values of the learned methods by performing Monte Carlo updates. At this stage, we do not learn the methods but only how to chose the appropriate one.

MDP and Monte Carlo Tree Search. A wide literature on MDP planning and Monte Carlo Tree Search refers to simulated execution, e.g., [25, 26, 55] and sampling outcomes of action models e.g., RFF [112], FF-replan [121], or hindsight optimization [122]. In particular, our UPOM procedure is an adaptation of the popular UCT algorithm [66], which has been used for various games and MDP planners, e.g., in PROST for RDDL domains [63]. The main conceptual and practical difference with our work is that these approaches use descriptive models, i.e., abstract actions on finite MDPs. Although most of the papers refer to online planning, they plan using descriptive models rather than operational models. There is no integration of acting and planning, hence no concerns about the planner's descriptive models versus the actor's operational models. Some works deal with hierarchically structured MDPs (see, e.g., [90, 4, 43]), or hierarchical extensions of Hidden Markov Models for plan recognition (see, e.g., [20]). However, most of the approaches based on MDPs do not deal with hierarchical models and none of them is based on refinement methods.

Planning and Execution in Robotics. There has been a lot of work in robotics to integrate planning and execution. They propose various techniques and strategies to handle the inconsistency issues that arise when execution and planning are done with different models. [68] shows how HTN planning can be used in robotics. [33] and [34] integrates task and motion planning for robotics. The approach of [81] 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 instance for starting or pursuing a task in a given context is formulated as 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 [59] 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 [60].

The integration of task and motion planning problem is also addressed in [119], which 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.

Logic Based Approaches. Approaches based on temporal logics (see e.g., [19]) specify acting and planning knowledge through high-level descriptive models and not through operational models like in RAE. Moreover, these approaches integrate acting and planning without exploiting the hierarchical refinement approach described here. Models based on GOLOG (see, e.g., [41, 10, 27]) share some similarities with RAE operational models, since they extend situation calculus with control constructs, procedure invocation and nondeterministic choice. In principle, GOLOG can use forward search to make a decision at each nondeterministic choice. This resembles our approach where UPOM is used to choose a method to be executed among the available ones. However, no work based on GOLOG provides an effective and practical planning method such as UPOM,

which is based on UCT-like Monte Carlo Tree Search and simulations over different scenarios. It is not clear to us how GOLOG could be extended in such a direction.

There are many commonalities between RAE and architectures based on BDI (Belief-Desire-Intention) models [79, 15, 103, 16]. Both approaches rely on a reactive system, but with differences regarding their primitives as well as their methods or plan-rules. Several BDI systems rely on a descriptive model, e.g., specified in PDDL, whereas RAE can handle any type of skill (e.g., physics-based action simulators) with nondeterministic effects. Because of the latter, there are also differences about how they can do planning. Nondeterministic approaches are required in RAE for the selection of methods, and since we are relying on a reactive approach, we do not synthesize a policy per se, but only a receding-horizon best choice.

Reinforcement Learning and Learning Domain Models. Our approach shares some similarities with the work on planning by reinforcement learning (RL) [58, 111, 35, 71, 32], 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 method instances in an operational model that allows for programming control constructs. This main difference holds also with works on hierarchical reinforcement learning, see, e.g., [120], [89], [100]. Works on user-guided learning, see e.g., [76], [75], 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 mapped to execution platforms, the main difference with our work still holds: Learning is performed in a descriptive model. [56] uses RL for user-guided learning directly in the specific case of robot motion primitives.

Several approaches have been investigated for learning planning-domain models. In probabilistic planning, for example [99], or [62], 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.

Relation with our prior work. Here is how the current paper relates to our prior work on this topic. A pseudocode version of RAE first appeared in [37]. An implementation of RAE, and three successively better planners for use with it, were described in [93, 92, 94]. The current paper is based on [94], with the following additional contributions. We provide complete formal specifications and explanations of the actor RAE and planner UPOM. We present a learning strategy to learn values of uninstantiated method parameters, with experimental evaluation. We have an additional experimental domain, called Deliver. We propose a new performance metric, called Retry Ratio, and evaluate it on our five experimental domains. We perform experiments with success-ratio (or probability of success) as the utility function optimized by UPOM, We compare success ratio

with efficiency. We perform experiments with varying the parameters, number of rollouts and maximum rollout length, of UPOM. We provide a proof of convergence of UPOM to a plan with optimal expected utility.

3. Architecture and Representation

Our approach integrates reactive and deliberative capabilities on the basis of hierarchical operational models. It focuses on a reactive perspective, extended with planning capabilities. This section presents the global architecture of the actor and details the ingredient of the representation.

3.1. Architecture

The popular three-layer architectures for autonomous deliberative actors usually combine *(i)* a platform layer with sensory-motor modules, *(ii)* a reactive control layer, and *(iii)* a deliberative planning layer [67]. As motivated earlier, our approach merges the last two layers within a reactive-centered perspective.

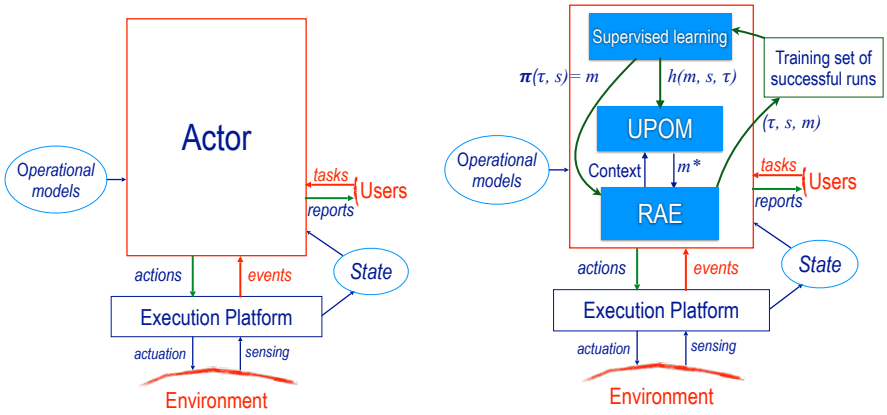

Figure 1: (a) Architecture of an actor reacting to events and tasks through an execution platform; (b) Integration of refinement acting, planning and learning.

The central component of the architecture (labelled "Actor" in Figure 1(a)) interacts with the environment for sensing and actuation through an execution platform, from which it receives events and world state updates. It also interacts with users getting tasks to be performed and reporting on their achievement.

RAE is the driving system within the actor. It reacts to tasks and events through hierarchical refinements specified by a library of operational models. At each decision step, RAE uses the planner UPOM to make the appropriate choice. UPOM performs a look-ahead by simulating available options in the current context. Supervised learning is used to speed-up UPOM with a heuristic, avoiding very deep and costly Monte Carlo rollouts; it also provides a base policy

for an anytime strategy when the actor has no time for planning (see Figure 1(b), the integration of planning, learning and refinement acting is detailed in subsequent sections).

3.2. Hierarchical Operational Models

We rely on a formalism described in [37, Chapter 3], which has been designed for acting and reacting in a dynamic environment. It provides a hierarchical representation of tasks through alternative refinement methods and primitive actions. Let us detail its main ingredients (see table of notation in Appendix C).

State variables. We rely on a representation with parameterized state variables. These are a finite collection of mappings from typed sets of objects of the domain into some range. For example, to describe the kinematic configuration of a two-arm robot r, its location with respect to a floor reference frame, and the status of a door d, one might use state variables $\mathsf{configuration}(r) \in \mathbb{R}^{17}$, $\mathsf{loc}(r) \in \mathbb{R}^2$, $\mathsf{door\text{-}status}(d) \in \{\mathsf{closed}, \mathsf{open}, \mathsf{cracked}\}$. Let X be a finite set of such state variables; variable $x \in X$ takes values from the set $\text{Range}(x)$. For each $x \in X$, to provide a convenient notation for handling partial knowledge, $\text{Range}(x)$ is extended to include the special symbol $\mathsf{unknown}$, which is the default value if x has not been set or updated to another value.

A state is a total assignment of values to state variables. The world state ξ is updated through observation by the execution platform, reflecting the dynamics of the external world. This update is continual for some state variables, e.g., $\mathsf{configuration}(r)$, with the usual propriosensing in robotics. Other variables may be updated to $\mathsf{unknown}$, unless they are in the range of a sensor or known to keep a value set by the actor. In the following algorithms, an update step, denoted as "observe current state ξ", precedes every reactive decision of the actor.

State abstraction. For the purpose of the planning lookahead, ξ is simplified into an abstract state $s \in S$ which evolves by reasoning. The state s, which gets updated from the observed state ξ each time the actor calls the planner (see Section 5), is a domain-dependent abstraction of ξ meeting the following conditions:

- The set of state variables of the abstract state s is a subset of X, i.e., some state variables in ξ may be ignored in s.
- For a variable x in s, $\text{Range}(x)$ can be a discretization of its range in Ξ. Continuing with the preceding robotics example, we might have $\text{Range}(\mathsf{configuration}(r)) = \{\mathsf{packed}, \mathsf{holding}, \mathsf{carrying}, \mathsf{manipulating}\}$, and $\text{Range}(\mathsf{loc}(r)) =$ a finite set of locations. Note that the formal proof of the asymptotic convergence of UPOM assumes S to be finite. However, this assumption is not a practical requirement of the planning algorithm.[2]

[2] We report (Section 7) on a test domain with continuous state variables in S.

Hence, a world state ξ is deterministically mapped to a single s=Abstract(ξ). An abstract state s may correspond to a subset of world states; but there is never a need to map back an abstract state. Indeed, the reasoning by simulation on s=Abstract(ξ) informs the actor about an appropriate choice of a method at some point. But the abstract states $s', s'', \ldots$, to which this reasoning led are not needed by the actor whose actions always rely on the current observed state.

Other variables and relations. The state variables in X are managed in the acting and planning algorithms as global variables. However, since methods embody programs, it is convenient to define *local variables*, which are generally derived from other variables. For example, one might use $\mathsf{stable}(o, pose) \in \{\top, \bot\}$, to mean that object o in some particular *pose* is stable; this property results from some geometric and dynamic computation. Local variables are updated by assignment statements inside methods. An assignment statement is of the form $x \leftarrow expr$, where *expr* may be either a ground value in Range(x), or a computational expression that returns a ground value in Range(x). Such an expression may include, for example, calls to specialized software packages.

It is convenient to define the unvarying properties of a domain through a set of *rigid relations* (as opposed to fluents, in our case parametrized state variables). For example, the adjacency of two locations or the color objects (if relevant) could be described with rigid relations.

Tasks. A task is a label naming an activity to be performed. It has the form $\mathsf{task\text{-}name}(args)$, where task-name designates the task considered, arguments *args* is an ordered list of objects and values. Tasks specified by a user are called *root* tasks, to distinguish them from the subtasks in which they are refined.

Events. An event designates an occurrence of some type detected by the execution platform; it corresponds to an *exogenous* change in the environment to which the actor may have to react, e.g., the activation of an emergency signal. It has the form $\mathsf{event\text{-}name}(args)$.

Actions. An action is a primitive function with instantiated parameters that can be executed by the execution platform through sensory motor commands. It has *nondeterministic* effects. For the purpose of planning, we do not represent actions with formal templates, as usually done with descriptive models. Instead, we assume there is a generative nondeterministic sampling simulator, denoted Sample. A call to $\mathsf{Sample}(a, s)$ returns a state s' randomly drawn among the possible states resulting from the execution of a in s. Sample can be implemented simply through a probability distribution of the effects of a (see Section 5).

When the actor triggers an action a for some task or event, it waits until a terminates or fails before pursuing that task or event. To follow its execution progress, when action a is triggered, there is a state variable, denoted $\mathsf{execution\text{-}status}(a) \in \{\mathsf{running}, \mathsf{done}, \mathsf{failed}\}$, which expresses the fact that the execution of a is going on, has terminated or failed. A terminated action returns a value of some type, which can be used to branch over various followup of the activity.

Refinement Methods. A refinement method is a triple of the form (*task*, *precondition*, *body*) or (*event*, *precondition*, *body*). The first field, either a task or an event, is its *role*; it tells what the method is about. When the *precondition* holds in the current state, the method is *applicable* for addressing the task or event in its role by running a program given in the method's *body*. This program refines the task or event into a sequence of subtasks, actions, and assignments. It may use recursions and iteration loops, but its sequence of steps is assumed to be finite.[3]

Refinement methods are specified as parameterized templates with a name and list of arguments *method-name*$(arg_1, \ldots, arg_k)$. An instance of a method is given by the substitution of its arguments by constants that are the values of state variables.

A method instance is applicable for a task if its role matches a current task or event, and its preconditions are satisfied by the current values of the state variables. A method may have several applicable instances for a current state, task, and event. An applicable instance of a method, if executed, addresses a task or an event by refining it, in a context dependent manner, into subtasks, actions, and possibly state updates, as specified in its body.

The body of a method is a sequence of lines with the usual programming control structure (if-then-else, while loops, etc.), and tests on the values of state variables. A *simple* test has the form $(x \circ v)$, where $\circ \in \{=, \neq, <, >\}$. A *compound* test is a negation, conjunction, or disjunction of simple or compound tests. Tests are evaluated with respect to the current state ξ. In tests, the symbol unknown is not treated in any special way; it is just one of the state variable's possible values.

The following example of a simplified search-and-rescue domain illustrates the representation.

Example 1. Consider a set R of robots performing search and rescue operations in a partially mapped area. The robots have to find people needing help in some area and leave them a package of supplies (medication, food, water, etc.). This domain is specified with state variables such as $\mathsf{robotType}(r) \in \{\text{UAV, UGV}\}$, $r \in R$, a finite set of robot names; $\mathsf{hasSupply}(r) \in \{\top, \bot\}$; $\mathsf{loc}(r) \in L$, 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 $\textsc{Detect}(r, \text{camera}, \text{class})$ (which detects if an object of some *class* appears in images acquired by *camera* of r), $\textsc{TriggerAlarm}(r, l)$, $\textsc{DropSupply}(r, l)$, $\textsc{LoadSupply}(r, l)$, $\textsc{Takeoff}(r, l)$, $\textsc{Land}(r, l)$, $\textsc{MoveTo}(r, l)$, and $\textsc{FlyTo}(r, l)$. They can address tasks such as:

[3] One way to enforce such a restriction would be as follows. For each iteration loop, one could require it to have a loop counter that will terminate it after a finite number of iterations. For recursions, one could use a *level mapping* (e.g., see [22, 46]) that assigns to each task t a positive integer $\ell(t)$, and require that for every method m whose task is t and every task t' that appears in the body of m, $\ell(t') < \ell(t)$. However, in most problem domains it is straightforward to write a set of methods that don't necessarily satisfy this property but still don't produce infinite recursion.

$\mathsf{search}(r, \mathit{area})$ (which makes a UAV r survey in sequence the locations in area), $\mathsf{survey}(l)$, $\mathsf{navigate}(r, l)$, $\mathsf{rescue}(r, l)$, and $\mathsf{getSupplies}(r)$.

Here is a refinement method for the survey task:

m1-survey(l, r)
  task: survey(l)
  pre: robotType$(r) = UAV$ and loc$(r) = l$ and status$(r) =$ *free*
  body: for all l' in neighbouring areas of l do:
    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')

This method specifies that in the location l the UAV r detects if a person appears in the images from its camera. In that case, it proceeds to a rescue task if it has supplies; if it does not it triggers an alarm event. This event is processed (by some other methods) by finding the closest robot not involved in a current rescue and assigning to it a rescue task for that location.

m1-GetSupplies(r)
  task: GetSupplies(r)
  pre: robotType$(r) = UGV$
  body: moveTo$(r$,loc$(BASE))$
    ReplenishSupplies(r)

m2-GetSupplies(r)
  task: GetSupplies(r)
  pre: robotType$(r) = UGV$
  body: $r_2 \leftarrow \mathrm{argmin}_{r'}\{\mathrm{Distance}(r, r') \mid \mathsf{hasMedicine}(r') = \textsc{True}\}$
    if $r_2 =$ None then Fail
    else:
      moveTo$(r, loc(r_2))$
      Transfer(r_2, r) □

Specification of an acting domain. We model an acting domain Σ as a tuple $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$ where:

- Ξ 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 methods for handling tasks or events in $\mathcal{T}$, $\mathsf{Applicable}(\xi, \tau)$ is the set of method instances applicable to τ in state ξ.
- $\mathcal{A}$ is the set of actions the actor may perform. We let $\gamma(\xi, a)$ be the set of states that may be reached after performing action a in state ξ.

We assume that Ξ, $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are finite.

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 instance $m \in \mathcal{M}$ to perform τ in a current state ξ. In Example 1, for a task $\mathsf{getSupplies}(r)$, the choice is between $\mathsf{m1\text{-}GetSupplies}(r)$ and $\mathsf{m2\text{-}GetSupplies}(r)$. Strictly speaking, the actor does not require a plan, i.e., an organized set of actions or a policy. It requires a selection procedure which designates for each task or subtask at hand the "best" method instance for pursuing the activity in the current context.

The next section describes a reactive actor which relies on a predefined preference order of methods in $\mathsf{Applicable}(\xi, \tau)$. Such an order is often natural when specifying the set of possible methods for a task. In Section 5 we detail a more informed receding-horizon look-ahead mechanism using an approximately optimal planning algorithm which provides the needed selection procedure.

4. Acting with RAE

RAE is adapted from [37, Chapter 3]. It maintains an *Agenda* consisting of a set of *refinement stacks*, one for each root task or event that needs to be addressed. A refinement stack σ is a LIFO list of tuples of the form $(\tau, m, i, \mathit{tried})$ where τ is an identifier for the task or event; m is a method instance to refine τ (set to *nil* if no method instance has been chosen yet); i is a pointer to a line in the body of m, initialized to 1 (first line in the body); and *tried* is a set of refinement method instances already tried for τ that failed to accomplish it. A stack σ is handled with the usual push, pop and top functions.

```
RAE:
Agenda ← empty list
while True do
1   for each new task or event τ to be addressed do
2       observe current state ξ
3       m ← Select(ξ, τ, ⟨(τ, nil, 1, ∅)⟩, d_max, n_ro)
4       if m = ∅ then output(τ, "failed")
        else Agenda ← Agenda ∪ {⟨(τ, m, 1, ∅)⟩}
5   for each σ ∈ Agenda do
        observe current state ξ
        σ ← Progress(σ, ξ)
6       if σ = ∅ then
            Agenda ← Agenda \ σ
            output(τ, "succeeded")
7       else if σ =retrial-failure then
            Agenda ← Agenda \ σ
            output(τ, "failed")
```

Algorithm 1: Refinement Acting Engine RAE

When RAE addresses a task τ, it must choose a method instance m for τ. This is performed by function Select (lines 3 of RAE, 5 of Progress, and 2 of Retry). Select takes five arguments: the current state ξ, task τ, and stack σ, and two control parameters d_{max}, n_{ro} which are needed only for planning. In purely reactive mode (without planning), Select returns the first applicable method instance, according to a pre-defined ordering, which has not already been tried (*tried* is given in σ). Note that this choice is with respect to the current world state ξ. Lines 2,4,1 in RAE, Progress and Retry respectively, specify to get an update of the world state from the execution platform. If $\mathsf{Applicable}(\xi, \tau) \subseteq$ *tried*, then Select returns $\emptyset$, i.e., there is no applicable method instances for τ in ξ that has not already been tried, meaning a failure to address τ.

The first inner loop of RAE (line 1) reads each new root task or event τ to be addressed and adds to the *Agenda* its refinement stack, initialized to $\langle(\tau, m, 1, \emptyset)\rangle$, m being the method instance returned by Select, if there is one. The root task τ for this stack will remain at the bottom of σ until solved; the subtasks in which τ refines will be pushed onto σ along with the refinement. The second loop of RAE progresses by one step in the topmost method instance of each stack in the *Agenda*.

```
   Progress(σ, ξ):
   (τ, m, i, tried) ← top(σ)
1  if m[i] is an already triggered action then
       case execution-status(m[i]):
           running:  return σ
2          failed:   return Retry(σ)
           done:     return Next(σ, ξ)
3  else if m[i] is an assignement step then
       update ξ according to m[i]
       return Next(σ, ξ)
   else if m[i] is an action a then
       trigger the execution of action a
       return σ
   else if m[i] is a task τ' then
4      observe current state ξ
5      m' ← Select(ξ, τ', σ, d_max, n_ro)
       if m' = ∅ then return Retry(σ)
       else return push((τ', m', 1, ∅), σ)
```

Algorithm 2: Progress returns an updated stack taking into account the execution status of the ongoing action, or the type of the next step in method instance m.

To progress a refinement stack σ, Progress (Algorithm 2) focuses on the tuple $(\tau, m, i, tried)$ at the top of σ. If the current line $m[i]$ is an action already triggered, then the execution status of this action is checked. If the action $m[i]$

is still running, this stack has to wait, but RAE goes on for other pending stacks in the *Agenda*. If $m[i]$ failed, Retry examines alternative method instances. Otherwise the action $m[i]$ is done: RAE will proceed in the following iteration with the next step in method instance m, as defined by the function Next (Algorithm 3).

```
Next (σ, ξ):
repeat
    (τ, m, i, tried) ← top(σ)
    pop(σ)
    if σ = ⟨⟩ then return ⟨⟩
until i is not the last step of m
j ← step following i in m depending on ξ
return push((τ, m, j, tried), σ)
```

Algorithm 3: Returns the next step in a method instance m for a stack σ and updates σ.

Next(σ, ξ) advances within the body of the topmost method instance 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 instance m has successfully addressed τ. If τ is a root task; Next and Progress return $\emptyset$, meaning that τ succeeded; its stack σ is removed from the *Agenda*. If i is not the last step of m, RAE proceeds to the next step j. Normally j is the next line after i, but if that line is a control instruction (e.g., an *if* or *while*) then j is the step to which the control instruction directs us (which of course may depend on the current state ξ).

Starting from line 3 in Progress, i points to the next line of m to be processed. If $m[i]$ is an assignment, the corresponding update of ξ if performed; RAE proceeds with the next step. If $m[i]$ is an action a, its execution is triggered; RAE will wait until a finishes to examine the Next step of m. If $m[i]$ is a task τ', a refinement with a method instance m', returned by Select, is performed. The corresponding tuple is pushed on top of σ. If there is no applicable method instance to τ', then the current method instance m failed to accomplish τ, a Retry with other method instances is performed.

```
   Retry(σ):
   (τ, m, i, tried) ← pop(σ)
   tried ← tried ∪ {m}                          ▷ m failed
1  observe current state ξ
2  m' ← Select(ξ, τ, σ, d_max, n_ro)
3  if m' ≠ ∅ then return push((τ, m', 1, tried), σ)
   else if σ ≠ ∅ then return Retry(σ)
4  else return retrial-failure
```

Algorithm 4: Retry examines untried alternative method instances, if any, and returns an updated stack.

Retry (Algorithm 4) adds the failed method instance m to the set of method instances that have been tried for τ and failed. It removes the corresponding tuple from σ. It retries refining τ with another method instance m' returned by Select which has not been already tried (line 3). If there is no such m' and if σ is not empty, Retry calls itself recursively on the topmost stack element, which is the one that generated τ as a subtask: retrial is performed one level up in the refinement tree. If stack σ is empty, then τ is the root task or event: RAE failed to accomplish τ.

RAE fails either (i) when there is no method instance applicable to the root task in the current state (line 4 of RAE), or (ii) when all applicable method instances have been tried and failed (line 7). A method instance fails either (i) when one of its actions fails (line 2 in Progress) or (ii) when all applicable method instances for one of its subtasks have been tried and failed (line 4 in Retry).

Note that Retry is not a backtracking procedure: it does not go back to a previous *computational node* to pick up another option among the candidates that *were* applicable when that node was first reached. It finds another method instance among those that are *now* applicable for the *current* state of the world ξ. RAE interacts with a dynamic world: it cannot rely on the set $\mathsf{Applicable}(\xi, \tau)$ computed earlier, because ξ has changed, new method instances may be applicable. However, the same method instance that failed at some point may succeed later on and may merit retrials. We discuss this issue in Section 8.

5. Planning for RAE

In Section 3, we informally defined the deliberative acting problem as the problem of selecting the "best" method instance $m \in \mathcal{M}$ to perform τ in a current state ξ for a domain $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$. A *refinement planning domain* is a tuple $\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$, where S is the set of states that are abstractions of states in Ξ, and $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are the same as in Σ.

Recall that if RAE is run purely reactively, Select chooses a refinement method instance from a predefined order of refinement methods, without comparing alternative options in the current context. In this section, we define a utility function to assess and compare method instances in $\mathsf{Applicable}(\xi, \tau)$ to select the best one. This utility function might, in principle, be used by an exact optimization procedure for finding the optimal method instance for a task. We propose a more efficient Monte Carlo Tree Search approach for finding an approximately optimal method instance. The planner relies on a procedure, called UPOM, inspired from the Upper Confidence bounds search applied to Trees (UCT). UPOM (*UCT Procedure for Operational Models*) is parameterized for rollout depth d and number of rollouts, n_{ro}. It relies on a heuristic function h for estimating the criterion at the end of the rollouts when $d < \infty$.

The proposed approach runs multiple simulations using the method instances and a *generative sampling model* of actions. This model is defined as a function Sample: $S \times \mathcal{A} \to S$. $\mathsf{Sample}(s, a)$ returns a state s' randomly drawn from $\gamma(s, a)$, with $\gamma : S \times A \to 2^S \cup \{\mathsf{failed}\}$. The transition function γ is augmented

with the token failed to account for possible failures of a. We assume, as usual, that the sampling reflects the probability distribution of the action's real-world outcomes.

A simulation of a method instance m for a task τ during planning goes successively through the steps of m, as required by the control flow for the current context, and generates a sequence of *simulated states* $\langle s_0, \ldots, s_i, \ldots \rangle$, where initially s_0 corresponds to an abstraction of the current real world state ξ. For instance in Example 1 this involves simulating for the method $\mathsf{m1\text{-}survey}(l, r)$ several $\mathsf{moveTo}(r, l')$ tasks, followed by DETECTPERSON(r, cam) action and $\mathsf{rescue}(r, l')$ tasks depending on the current context. The utility function is computed along such a sequence, taking into account the *deterministic* refinements of method instances and the *nondeterministic* outcomes of actions (see Figure 2). Simulation during planning does not Retry, as in RAE, but it takes into account possible failures. Further, we do not observe in UPOM nor consider possible changes in ξ during a simulation. These changes, when leading to events, are dealt with at the acting level through the main loop of RAE which remains concurrently active during planning. [MALIK SAYS: *I modified the point regarding ξ*]

5.1. Utility criteria and optimal approach

The appropriate utility function can be application dependent. One may consider a function combining rewards for desirable or undesirable states, and costs for the time and resources of actions. To keep the formal presentation simple, we assume that there are no rewards in states. We studied two utility functions measuring respectively the actor's efficiency and robustness. Regarding the former, instead of minimizing costs, the efficiency utility function maximizes values to easily account for failures. For the latter, the actor seeks a method instance that has a good chance to succeed.

We first define two value functions for actions, v_e and v_s, which lead to the two proposed utility functions for method instances.

Efficiency. Let $\mathsf{Cost} : S \times \mathcal{A} \times (S \cup \{\mathsf{failed}\}) \to \mathbb{R}^+$ be a cost function. $\mathsf{Cost}(s, a, s')$ is the cost of performing action a in state s when the outcome is s'. Note that the cost of an action a is finite even when a fails. This is the case since in general an actor is able to figure out that an attempted action failed to limit its cost. However, a failed action a in a method instance m leads to the failure of m; its eficiency is simply 0. Hence we define the efficiency value of an action as follows:

$$v_e(s, a, s') = \begin{cases} 0 & \text{if } s' = \text{“failed”}, \\ 1/\mathsf{Cost}(s, a, s') & \text{otherwise.} \end{cases} \tag{1}$$

If we let $v_{e1} \oplus v_{e2}$ denote the cumulative efficiency value of two successive actions whose efficiency values are $v_{e1} = 1/c_1$ and $v_{e2} = 1/c_2$, then

$$v_{e1} \oplus v_{e2} = 1/(c_1 + c_2) = 1/\left(\tfrac{1}{v_{e1}} + \tfrac{1}{v_{e2}}\right) = v_{e1} \times v_{e2}/(v_{e1} + v_{e2}). \tag{2}$$

Success Ratio. Here, we measure the utility of a method instance as its probability of success over all possible outcomes of its actions. Hence we simply take a value 0 for an action that fails, and 1 if the action succeeds.

$$v_s(s, a, s') = \begin{cases} 0 & \text{if } s' = \text{ “failed”}, \\ 1 & \text{otherwise.} \end{cases} \tag{3}$$

If we let $v_{s1} \oplus v_{s2}$ denote the cumulative success ratio for two successive actions in a method instance whose success ratios are v_{s1} and v_{s2}, then

$$v_{s1} \oplus v_{s2} = v_{s1} \times v_{s2}. \tag{4}$$

For both value functions v_e and v_s, the operator $\oplus$ is associative, which is needed for combining successive steps. For both value functions, we let $\mathbb{I}$ denote the identity element for operation $\oplus$, i.e., the element such that $x \oplus \mathbb{I} = \mathbb{I} \oplus x = x$:

- For v_e in Equation 1, we have $\mathbb{I} = \infty$, corresponding to a cost of $1/\mathbb{I} = 0$. If $v_{e1} = \mathbb{I}$, then $v_{e1} \oplus v_{e2} = 1/(0 + \frac{1}{v_{e2}}) = v_{e2}$ for every v_{e2}.
- For v_s in Equation 3, we have $\mathbb{I} = 1$, corresponding to success (task is already accomplished).

Note that if either of two actions in a method instance m fails, their combined value is 0, since m also fails.

Let us now define a utility function for method instances using either v_e or v_s. In order to compute the expected utility of a method instance m we need to consider possible *traces* of the execution of m for a task τ. In RAE, an execution trace was conveniently represented though the evolution of σ for the task τ. In planning, we similarly use σ as a LIFO list of tuples $(\tau, m, i, \textit{tried})$, as defined in RAE.[4] For a given simulation of m for τ, σ is initialized as a copy of the current stack in RAE.We progress in the simulation of m step by step using the function Next (Algorithm 3), pushing in σ a new tuple when a step requires a refinement into a subtask.

Let $\mathsf{top}(\sigma)$ be the stack tuple $(\tau, m, i, \textit{tried})$. The utility of a particular simulation of i^{th} step of m for τ is given by the following recursive equation:

$$U(m, s, \sigma) = \begin{cases} U(m, s', \mathsf{Next}(\sigma, s)) & \text{if } m[i] \text{ is an assignment,} \\ v(s, a, s') \oplus U(m, s', \mathsf{Next}(\sigma, s)) & \text{if } m[i] \text{ is an action } a, \\ U(m', s, \mathsf{push}((\tau', m', 1, \varnothing), \mathsf{Next}(\sigma, s)) & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if } \sigma = \varnothing, \end{cases} \tag{5}$$

where v is either v_e or v_s. An assignment step changes the state from s to s' but does not change the utility U. An action a changes the state nondeterministically to s'; the utility is the combined value of a and the utility of the remaining

[4] We do not need for the moment to keep track of already *tried* method instances, but we'll see in a moment the usefulness of this term

step. A refinement step does not change the state; it is addressed in this particular simulation by refining τ into τ' with m'. The function Next moves to the following step, and to the empty stack at the end of every simulated execution.

From Equation 5 we derive the *maximal expected utility* of m for τ by maximizing recursively over all possible refinements in m and averaging over all possible outcomes of actions, including failures:

$$U^*(m,s,\sigma) = \begin{cases} U^*(m,s',\mathsf{Next}(\sigma,s)) & \text{if } m[i] \text{ is an assignment,} \\ \sum_{s'\in\gamma(s,a)} \Pr(s'|s,a)[v(s,a,s') \oplus U^*(m,s',\mathsf{Next}(\sigma,s))] & \text{if } m[i] \text{ is an action } a, \\ \max_{m'\in\mathsf{Applicable}(s,\tau')} U^*(m',s,\mathsf{push}((\tau',m',1),\mathsf{Next}(\sigma,s)) & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if } \sigma = \varnothing. \end{cases} \tag{6}$$

In the above equation, $\gamma(s,a)$ includes the token "failed". We assume as usual that if $\mathsf{Applicable}(s,\tau) = \varnothing$ then $\max_{m\in\mathsf{Applicable}(s,\tau)} U^*(m,s,\sigma) = 0$, meaning a refinement failure. Instantiating v as either v_e or v_s gives the two utility functions, the efficiency and the success ratio of method instances, respectively.

The optimal method instance for a task τ in a state s for the utility U^* is:

$$m^*_{\tau,s} = \mathrm{argmax}_{m\in\mathsf{Applicable}(s,\tau)} U^*(m,s,\langle(\tau,m,1,\varnothing)\rangle) \tag{7}$$

It is possible to implement Equation 6 directly as a recursive backtracking optimization algorithm and to make the planning algorithm return $m^*_{\tau,s}$, as defined above. However, this would be too computationally demanding and not practical for an online planner. We propose instead to seek an approximately optimal method instance with an anytime controllable procedure using a Monte Carlo Tree Search algorithm in the space of operational models.

5.2. A planning algorithm based on UCT

To find an approximation $\tilde{m}$ of m^*, we propose a progressive deepening Monte Carlo Tree Search procedure with n_{ro} rollouts, down to a depth d_{max} in the refinement tree of a task τ (see Figure 2). A rollout in MCTS is an exploration of a path along a random branch from each nondeterministic node (i.e., an outcome of an action) down to depth d_{max}. The basic ideas of UPOM are the following:

- at an action node of the search tree, we average over the values of the corresponding n_{ro} rollouts;
- at a task node, we choose the refinement method instance with the highest expected utility;
- starting from $d = d_{\max}$,we decrease d for a refinement step and an action step, but not in an assignment step;

- we take a heuristic estimate of the utility of the remaining refinements at the tip of a rollout, i.e., at $d = 0$;
- we stop a rollout at a failure of an action or a refinement, and return a value $U_{\mathsf{Failure}} = 0$; we also stop when the stack is empty and return $U_{\mathsf{Success}} = \mathbb{I}$.

```
   Select(ξ, τ, σ, d_max, n_ro):
   (τ, m, i, tried) ← top(σ)
   M ← Applicable(ξ, τ) \ tried
   if M = ∅ then return ∅
   if |M = {m}| = 1 then return m
   s ← Abstract(ξ) ; σ ← copy of σ; d ← 0
1  m̃ ← argmax_{m∈M} h(τ, m, s)                    ▷ initialize m̃
2  repeat
   |  d ← d + 1                                   ▷ progressive deepening
3  |  for n_ro times do
   |  |  UPOM (s, push((τ, nil, 1, ∅), σ), d)
   |  m̃ ← argmax_{m∈M} Q_{σ,s}(m)
   until d = d_max or search time is over
   return m̃
```

Algorithm 5: A progressive deepening procedure using UPOM for finding an approximately optimal method instance.

This is detailed in algorithms 5 and 6. Select is called by RAE with five parameters: ξ, τ, and σ, and the control parameters d_{max}, the maximum rollout depth, and n_{ro}, the number of UCT rollouts. Recall that on a new root task τ, RAE calls Select with $\sigma = \langle(\tau, nil, 1, \varnothing)\rangle$. Select returns $\tilde{m}$, an approximately optimal method instance for τ, or $\varnothing$ if no method instance is found, i.e., if there is no applicable method instances for τ in ξ, but of those already tried by RAE for this task. Select uses a copy of RAE's current stack σ, and a simulation state s, which is an abstraction of the current execution state ξ (e.g., in Example 1, l can be a precise metric location for acting and topological reference for planning). It initializes $\tilde{m}$ with a heuristic estimates (line 1). It performs a succession of simulations at progressively deeper refinement levels using the function UPOM to evaluate the utility of a candidate method instance. The progressive deepening loop (line 2) is pursued until reaching the maximum rollout depth, or until the actor interrupts the search because of time limit or any other reason, at which point the current $\tilde{m}$ is returned and will be tried by RAE. Select is an *anytime* procedure: it returns a solution whenever interrupted. $Q_{\sigma,s}(m)$ is a global data structure that approximates the utility $U^*(m, s, \sigma)$.

UPOM (Algorithm 6) takes as arguments a simulation state s, a stack σ, and the rollout depth d. It performs one rollout over recursive calls for a method instance m and its refinements. On the first call of a rollout, $m = nil$, meaning that no method instance has yet been chosen. A method instance m_c is chosen among untried method instances (line 3). If all method instances have been

```
   UPOM(s, σ, d):
   if σ = ⟨⟩ then return U_Success
   (τ, m, i, tried) ← top(σ)
1  if d = 0 then return h(τ, m, s)
   if m = nil or m[i] is a task τ' then
       if m = nil then τ' ← τ
       if N_{σ,s}(τ') is not initialized yet then
2          M' ← Applicable(s, τ') \ tried
           if M' = 0 then return U_Failure
           N_{σ,s}(τ') ← 0
           for m' ∈ M' do
               N_{σ,s}(m') ← 0 ; Q_{σ,s}(m') ← 0
       Untried ← {m' ∈ M' | N_{σ,s}(m') = 0}
       if Untried ≠ ∅ then
3          m_c ← random selection from Untried
4      else m_c ← argmax_{m∈M'} {Q_{σ,s}(m) + C × [log N_{σ,s}(τ)/N_{σ,s}(m)]^{1/2}}
5      λ ← UPOM(s, push((τ', m_c, 1, ∅), Next(σ, s)), d − 1)
6      Q_{σ,s}(m_c) ← [N_{σ,s}(m_c) × Q_{σ,s}(m_c) + λ]/[1 + N_{σ,s}(m_c)]
       N_{σ,s}(m_c) ← N_{σ,s}(m_c) + 1
       return λ
   if m[i] is an assignment then
       s' ← state s updated according to m[i]
       return UPOM(s', Next(σ, s'), d)
   if m[i] is an action a then
7      s' ← Sample(s, a)
       if s' = failed then return U_Failure
8      else return v(s, a, s') ⊕ UPOM(s', Next(σ, s'), d − 1)
```

Algorithm 6: Monte Carlo tree search procedure UPOM; performs one rollout recursively down the refinement tree of a method instance to compute an estimate of its optimal utility.

tried, m_c is chosen (line 4) according to a tradeoff between exploration and exploitation. The constant $C > 0$ fixes this tradeoff for the exploration less sampled method instances (high C) versus the exploitation or more promising ones (low C).

$Q_{\sigma,s}(m)$ is calculated as follows. $Q_{\sigma,s}(m)$ combines the value of a sampled action with the utility of the remaining part of a rollout (line 8), and it updates Q by averaging over previous rollouts (line 6). The value function v (line 8) is either v_e or v_s depending on the chosen utility function, efficiency or success ratio. For both function, $U_{\mathsf{Success}} = \mathbb{I}$ and $U_{\mathsf{Failure}} = 0$.

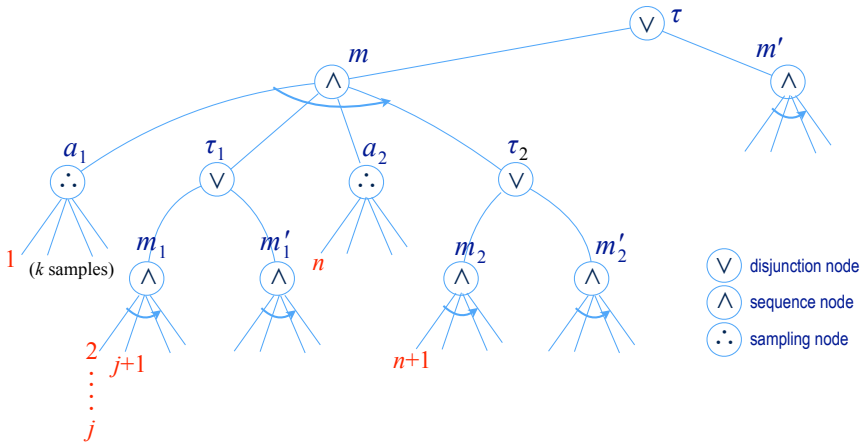

Figure 2: A refinement tree, with three types of nodes: *disjunction* for a task over possible method instances, *sequence* for a method instances over all its steps, and *sampling* for an action over its possible outcomes. A rollout can be, for example, the sequence of nodes marked 1 (a sample of a_1), 2 (first step of m_1), $\ldots, j$ (subsequent refinements), $j+1$ (next step of m_1), $\ldots, n$ (a sample of a_2), $n+1$ (first step of m_2), etc.

Differences from Equations 6 and 7. A significant difference between the pseudocode in Algorithm 6 and Equation 6 is the restriction of Applicable to method instances that have not been tried before by RAE for the same task. This is a conservative strategy, because at this point the actor has no means for distinguishing failures of tried method instances that require retrials from those that don't. We'll come back to a retrial strategy in Section 9.

Another difference shows up in the initialization of σ in Select. This is explained by going back to how Select is used by RAE. At a root task τ, when Select is called the first time (line 3 of RAE), $\sigma = \langle(\tau, nil, 1, \varnothing)\rangle$. If RAE proceeds for τ with a method instance m returned by Select, at the next refinement call of RAE, e.g., for τ_1 (see Figure 2) Select needs to consider the utility of the method instances for τ_1, but also their impact on the remaining steps in m, here on a_2 and τ_2. In other words, the actor requires the best method instance for τ_1 in the context of its current execution state, taking into account the remaining steps of the method instance m it is executing. This best method instance for τ_1 may be different from that given by Equation 7. The need to keep track of previously tried method instances and pending tasks explains why σ is taken as a copy of the current σ in RAE for the root task at hand. However, this does not lead to reconsider previously made choices of method instances the actor is currently executing, e.g., in Figure 2, m' is not reassessed. Note that UPOM does not pursue a rollout at an internal refinement node with the method instance maximizing the current utility evaluation Q, but with the best method instance according to the UCT exploration/exploitation tradeoff (line 4).

Asymptotic convergence. In Appendix A we prove the asymptotic convergence of UPOM towards an optimal method, i.e., as $n_{ro} \to \infty$ (Theorem 1). The proof assumes no depth cut-off ($d_{max} = \infty$) and static domains, i.e., domains without exogenous events.[5] It proceeds by mapping UPOM's search strategy into UCT, which has been demonstrated to converge on a finite horizon MDP with a probability of not finding the optimal action at the root node that goes to zero at a polynomial rate as the number of rollouts grows to infinity (Theorem 6 of [66]). To simplify the mapping, we first consider UPOM with an additive utility function, and show how to map UPOM's search space into an MDP. We then discuss how this can be extended to the efficiency and success ratio utility functions defined in 5, using the fact that the UCT algorithm is not restricted to the additive case; it still converges as long as the utility function is monotonic.

Control parameters. The effects of the two control parameters d_{max} and n_{ro} are not independent. This is because UCT exploration examines an untried method instance before pursuing a rollout on an already tried one. Exploration would be complete if $n_{ro} > \mu$, where [DANA SAYS: *I modified the following equation to try to improve readability. Please check whether it's OK.*]

$$\mu = \sum_{\tau_i \text{ is a subtask}} \max_s |\mathsf{Applicable}(s, \tau_i)|,$$

over all subtasks τ_i in the refinement tree (see Figure 2), down to a refinement depth of the root task. But μ increases with d_{max}. In our experiments, we keep a large constant n_{ro} and increase d in the progressive deepening loop until the max depth d_{max}. An alternative control of Select can be the following:

- for a given d, pursue the rollouts (line 3) until there are K successive exploitation rollouts, i.e., for which $Untried = \varnothing$, for some constant K;[6]
- pursue the progressive deepening loop (line 2) until no subtask is left unrefined for the K exploitation rollouts or until the search time is over.

This is an adaptive control strategy that requires only two constants C and K.

Search depth. Finally, let us discuss the important issue of the depth cutoff strategy. Two options may be considered: *(i)* d is the number of steps of a rollout (as in MDP algorithms), or *(ii)* d is the refinement depth of a rollout. The pseudocode in Algorithm 6 takes the former option: d decreases at every recursive call, for an action step as well as for a task refinement step. The advantage is that the cutoff at $d = 0$ stops the current evaluation. The difficulty is that the root method instance, and possibly its refinements, are only partially evaluated. For example in Figure 2, if $j > d_{max}$, steps a_2 and τ_2 of m will never be considered; similarly for the remaining steps in m_1: rollouts will go in deep

[5] It should be possible to extend the proof to dynamic domains if there are known probability distributions over the occurrence of exogenous events.

[6] The probabilistic roadmap motion planning algorithm uses a similar idea to stop after K configuration samples unsuccessful for augmenting the roadmap.

refinements and never assess all the steps of evaluated method instances. The value returned by UPOM can be arbitrarily far from U^*. The other issue of this strategy is that the heuristic estimate has to take into account remaining refinements lower down the cutoff point as well as remaining steps higher up in the refinement tree, i.e., what remains to be evaluated in σ.

In the alternative option where d is the refinement depth of a rollout, d decreases at a task refinement step only, not at an action step. The advantage is to allow each rollout to go through all the steps of every developed method instance. Furthermore, the heuristic estimate at a cutoff is focused in this case on a subtask and its applicable method instances, whose simulation will not be started (nondeveloped method instances). The disadvantage is that one needs an estimate of the state following the achievement of a task with a nondeveloped method instance in order to pursue the sibling steps. In Figure 2 with $d = 1$ for example, τ_1 will not be refined; a_2 and remaining steps of m will be based on an estimated state following the achievement of τ_1. The definition of a default state change following a task is domain dependent and might not be easily specified in general.

The modifications needed in UPOM to implement option *(ii)* are as follows:

- In order to be able to go back to higher levels of d when the simulation is pursued in parent method instances after a cutoff, it is convenient to maintain d as part of the simulation stack: a fifth term d is added in every tuple of σ.
- The arguments of UPOM are modified according to the previous point.
- Line 1 in UPOM has to pursue the evaluation higher up in σ:

  $$\textbf{if } d = 0 \textbf{ then return } h(\tau, m, s) \oplus \mathsf{UPOM}(g(s, \tau, m), pop(\sigma), b, k),$$

  where $g(s, \tau, m)$ is a default state after the achievement of τ with m in s.

For our experimental results (see Section 7), we have implemented a mixture of the two options: we take d as the refinement steps of a rollout (decreasing d at a task refinement step only), but we stop the evaluation when reaching $d = 0$, taking heuristic estimates for the remaining steps of pending method instances. This has the disadvantage of a partial evaluation, but its advantages are to allow easily defined heuristic and not require a following state estimate.

6. Learning for RAE and UPOM

Purely reactive RAE chooses a method instance for a task using an *a priori* ordering or a heuristic. RAE with anytime receding-horizon planning uses UPOM to find an approximately optimal method instance to refine a task or a subtask. At maximum rollout depth, UPOM needs also heuristic estimates

The classical techniques for domain independent heuristics in planning do not work for operational refinement models. Specifying by hand efficient domain-specific heuristics is not an acceptable solution. However, it is possible to learn 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.

We developed three learning procedures to guide RAE and UPOM:

- 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. In Example 1, for the task getSupplies, Learnπ is used to choose between m1-GetSupplies and m2-GetSupplies in the current context.
- Learnπ_i learns the values of uninstantiated parameters of refinement method m chosen by Learnπ. In Example 1, for the method m1-survey(l, r), Learnπ_i is used to choose the value of r. The value of l comes from the argument l in the task survey(l).
- LearnH learns a heuristic evaluation function to be used by UPOM.

6.1. Learning to choose methods (Learnπ)

In a first approach, Learnπ learns a mapping from contexts to partially instantiated methods. A parameter of a method instance can inherit its value from the task at hand. However, different instances of a method may be applicable in a given state to the same task. This is illustrated in Example 1 by method m1-survey(l, r) where l is inherited from the task, but r can be instantiated as any robot such that status$(r) = \mathit{free}$. Learnπ simplifies the learning by abstracting all these applicable method instances to a single class. To use the learned policy, RAE chooses randomly among all applicable instances of the learned method for the context at hand. Learnπ learning procedure consists of the following four steps, which are schematically depicted in Figure 3.

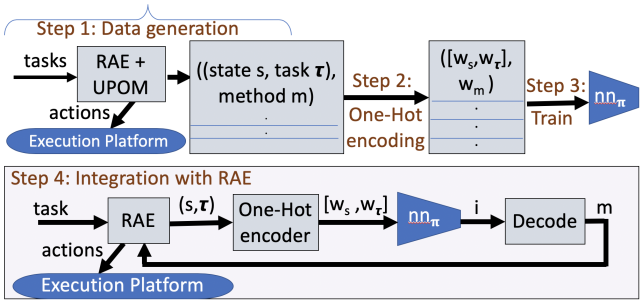

Figure 3: A schematic diagram for the Learnπ procedure.

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 of the method m. We tested two approaches (the results of the tests are in Section 7):

- 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.[7] Thus the encoding is

$$((s, \tau), m) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau], w_m), \tag{8}$$

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 $s \in S$ 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 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 $|\mathcal{M}|$ where $\mathcal{M}$ 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. RAE uses 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 8. Then m is chosen as

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

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

6.2. Learning to choose method instances (Learnπ_i)

Here, we extend the previous approach to learn a mapping from context to fully instantiated methods. The Learnπ_i procedure learns over all the values of uninstantiated parameters using a multi-layered perceptron (MLP). We have a separate MLP for each uninstantiated parameter.

[7] 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.

Step 1: Data generation. For each uninstantiated method parameter v_{un}, training is performed on a set of data records of the form $r = ((s, v_\tau), b)$, where s is the current state, v_τ is a list of values of the task parameters, and b is the value of the parameter v_{un}. Data records are obtained by making RAE call UPOM offline with randomly generated tasks. Each call returns a method instance m and the value of its parameters.

Step 2: Encoding. Given a record $r = ((s, v_\tau), b)$, we encode (s, v_τ) into an input-feature vector and encode b into an output label. Thus the encoding is

$$((s, v_\tau), b) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_{v_\tau}], w_b), \tag{9}$$

with w_s, w_{v_τ} and w_b being One-Hot representations of s, v_τ, and b.

Step 3: Training. We train a multi-layered perceptron (MLP) for each uninstantiated task parameter v_{un}. Each such MLP $nn_{v_{un}}$ consists of two linear layers separated by a ReLU layer to account for non-linearity in our training data. To learn and classify $[w_s, w_{v_\tau}]$ by the values of v_{un}, we used a SGD (Stochastic Gradient Descent) optimizer and the Cross Entropy loss function. The output of $nn_{v_{un}}$ is a vector of size $|Range(v_{un})|$. Each dimension in the output represents the degree to which v_{un} takes a specific value.

Step 4: Integration in RAE. After RAE has chosen a refinement method m for task τ, we have RAE use the trained network $nn_{v_{un}}$ to choose a value for each uninstantiated parameter v_{un}. RAE encodes (s, v_τ) into $[w_s, w_{v_\tau}]$ using Equation 9. Then, the value for v_{un}, b is chosen as

$$b \leftarrow Decode(\text{argmax}_j(nn_{v_{un}}([w_s, w_{v_{un}}])[j])),$$

where *Decode* is a one-one mapping from integer indices to $Range(v_{un})$.

With Learnπ_i, we choose the value of each uninstantiated parameter independently of the others, while remaining among currently applicable values (i.e., valid method instances). Certainly, parameters are not independent, and are not processed as such by RAE nor UPOM. This choice is simply *relaxation assumption* in Learnπ_i, which is common in the design of heuristics. Despite this relaxation, we found the guidance of Learnπ and Learnπ_i to be quite effective when compared to reactive RAE (see [94, Figures 6 and 7]). Furthermore, recall that Learnπ_i is needed only when methods have more parameters than the task they address, and it is used only as first choice in progressive deepening, when there is no time for planning.

*6.3. Learning a heuristic function (*LearnH*)*

The LearnH procedure tries to learn an estimate of the utility u of accomplishing a task τ with a method instance 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. We chose to use classification.[8] 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 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 are the following (see Figure 4):

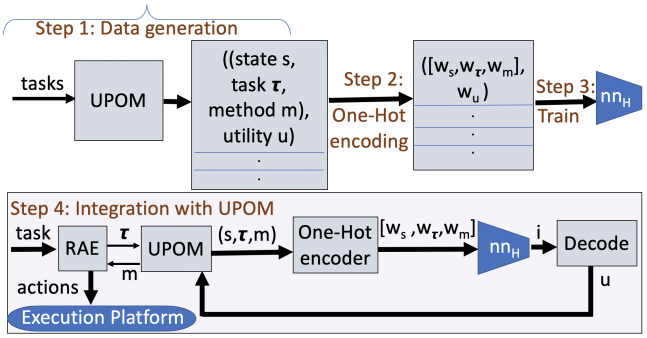

Figure 4: A schematic diagram for the LearnH procedure.

Step 1: Data generation. We generate data records in a similar way as in the Learnπ procedure, 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

$$((s, \tau, m), interval(u)) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau, w_m], w_u) \tag{10}$$

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 the 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.

[8] We chose classification because it outperformed regression in our preliminary experiments. It is possible that regression could be made to perform better, but that is beyond the scope of this paper.

Step 4: Integration in RAE. RAE calls 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 10, and $Decode$ is a one-one mapping from a utility interval to its mid-point. Before the progressive deepening loop over calls to UPOM, Select initializes $\tilde{m}$ in line 1 according to this heuristic h.

6.4. Incremental online learning

The previous learning procedures rely on synthetic training data obtained from a collection of states and tasks. However, even if one is careful, the training data may not reflect an actor's specific working conditions. This is a well known important issue in machine learning. It can be addressed by continual incremental online learning. Here is a procedure to do so in our framework.

- Initialization: either (i) without a heuristic by running RAE+UPOM online with $d_{max} = \infty$, or (ii) with an initial heuristic obtained from offline learning on simulated data.
- Online acting, planning and incremental learning:
  - Augment the training set by recording successful methods (with the values of uninstantiated parameters) 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 planning time available.
  - 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.

The issues for deploying this procedure in a practical application are discussed in Section 8.

7. Experimental Evaluation

7.1. Domains

We have implemented and tested our framework on five domains which illustrate service and exploration robotics scenarios with aerial and ground robots. All the agents are under a centralized control. In these domains, acting is performed in a simulated environment. Consequently, we did not need to abstract the acting state ξ into a planning state s; both are identical in our tests.

The S&R domain extends the search and rescue setting of Example 1 with several 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 localized persons. Exogenous events are weather conditions and debris in

paths. The complete set of tasks, commands, and refinement methods for the S&R domain is described in Appendix B.

In Explore, several chargeable UGVs and UAVs explore a partially known terrain and gather information by surveying, screening, monitoring, e.g., for ecological studies. They need to go back to the base regularly to deposit data or to collect a specific equipment. Exogenous events are appearance of animals in motion.

In Fetch domain, 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 Deliver domain has several robots in a shipping warehouse that must co-operatively package incoming orders, i.e., lists of items of different types and weights to deliver to customers. Items for a single order have be placed in a machine, which packs them together; packages have to be placed in the shipping doc. To process multiple orders concurrently, items can be moved to a pallet before transfer to a machine. Robots have limited capacities.

S&R, Explore, Nav and Fetch have sensing actions. S&R, Explore, Fetch and Deliver can have dead-ends. The features of these domains are summarized in Table 1, Table 2 and Table 3. Recall from Section 3 that $\mathcal{M}$ is the set of all refinement methods, $\overline{\mathcal{M}}$ is the set of all refinement method instances, and $\overline{\mathcal{T}}$ and $\overline{\mathcal{A}}$ are the sets of instances of tasks and actions. In Table 2, the upper bound on the size of the state space is $|S| < \prod_{x \in X} |Range(x)|$, since state variables are generally not independent.

Domain	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
S&R	✓	✓	✓	✓	✓
Explore	✓	✓	✓	✓	✓
Fetch	✓	✓	✓	–	✓
Nav	✓	–	✓	✓	✓
Deliver	✓	✓	–	✓	✓

Table 1: Features of the test domains.

7.2. Planning parameters

Here we analyze the effect of the two planning parameters, n_{ro} and d_{max}, on the two utility functions we considered, the efficiency, and the success ratio, as

Domain	Upper bound on $\lvert\mathcal{S}\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert\overline{\mathcal{T}}\rvert$	$\lvert\mathcal{M}\rvert$	$\lvert\overline{\mathcal{M}}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\overline{\mathcal{A}}\rvert$
S&R	9.6×10^{16}	8	2508	16	8766	14	2.7×10^{6}
Explore	1.93×10^{21}	9	358	17	756	14	843
Fetch	2.4×10^{17}	6	270	10	282	9	463
Nav	1.5×10^{11}	6	192	9	651	10	490
Deliver	∞	6	64	6	318	9	4442

Table 2: Sizes of the test domains. $\mathcal{M}$ is the set of all refinement methods, $\overline{\mathcal{M}}$ is the set of all refinement method instances, and $\overline{\mathcal{T}}$ and $\overline{\mathcal{A}}$ are the sets of instances of tasks and actions. In the Deliver domain, the number of states is infinite because some of the state variables may have real values, but the other numbers are finite because the task, methods, and instances do not have real-valued arguments.

Domain	Rollout length		Branching factor			
			Task nodes ($\vee$)		Action nodes ($\cdot$)	
	Avg	Max	Avg	Max	Avg	Max
S&R	19	27	2.4	4	1.1	2
Explore	20	112	1.5	3	1.2	2
Fetch	26	56	1.3	3	1.4	4
Nav	30	78	2	6	1.2	2
Deliver	42	52	1.1	64	1.1	2

Table 3: Estimates of the search space parameters of the test domains.

well as on the retry ratio of RAE. We tested $n_{ro} \in [0, 1000]$ and $d_{max} \in [0, 30]$. The case $n_{ro} = 0$ rollout corresponds to purely reactive RAE, without planning. We only report for $n_{ro} \in [0, 250]$ since no significant additional effect was observed beyond $n_{ro} > 250$. We tested each domain on 50 randomly generated problems. A problem consists of one or two root tasks that arrive at random time points in RAE's input stream, together with other randomly generated exogenous events. For each problem we recorded 50 runs to account for the nondeterministic effects of actions. We measured the following:

- the efficiency of RAE for a task, i.e., the reciprocal of the sum of the costs of the actions executed by RAE for accomplishing that task;
- the success ratio of RAE for a run, i.e., the number of successful tasks over the total of tasks for that run; and
- the retry ratio of RAE for a run, i.e., the number of call to Retry over the total of tasks for that run.

Since we are more concerned with the relative values than the absolute values of the efficiency, the success ratio, and the retry ratio, we rescaled in the following plots the Y axis with respect to the base case for $n_{ro} = 0$. Note that the measured efficiency takes into account the execution context with concurrent tasks and exogenous events; hence it is different for the corresponding utility function optimized in UPOM (i.e., the expected efficiency of Equation 6); similarly for the success ratio. We used a 2.8 GHz Intel Ivy Bridge processor. The cut-off time for a run was set to 30 minutes.

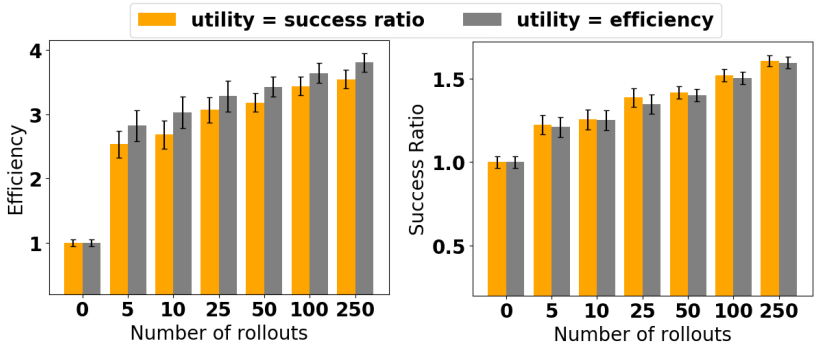

Figure 5: Efficiency and success ratio for two different utility functions (orange is expected success ratio and gray is expected efficiency) averaged over all five domains, with $d_{max} = \infty$. The Y axis is rescaled with respect to the base case of U for $n_{ro} = 0$.

Comparison of the two utility functions. We studied two utility functions that are not totally independent but assess different criteria. The success ratio is useful as a measure of robustness. Suppose method instance m_1 is always successful but has a large cost, whereas m_2 sometimes fails but costs very little when it works: m_1 has a higher success ratio, but m_2 has higher expected efficiency.

Figure 5 shows the measured efficiency and success ratio of RAE for the two utility functions, averaged over all domains. Each data point is the average of 12,500 runs, with the error bars showing 95% confidence interval; we plot relative values with respect the base case of U for $n_{ro} = 0$. As expected, the measured efficiency is higher when the optimized utility function of UPOM is the expected efficiency. Similarly for the success ratio. However, optimizing one criteria has also a good effect on the other one, since the two are not independent. We also observe that 5 rollouts have already a significant effect on the efficiency, with slight improvements as UPOM does more rollouts. In contrast, the success-ratio increases smoothly from no planning to planning with 250 rollouts. This can be due to the difference between the two criteria: a task that succeeds in its first attempt and a task that succeeds after several retries of RAE have both a success-ratio of 1, but the efficiency in the latter case is lower. This point is analyzed next.

Retry ratio. Figure 6 shows the *retry ratio*, i.e., the number of calls to Retry, divided by the total number of tasks. Recall that calling Retry is how RAE faces the failure of chosen methods; RAE may succeed but after numerous retrials, which is not desirable. Although the retry ratio criteria is not independent from the combined two utility functions, this ratio depicts very clearly how effective the guidance of RAE is. We observe that the retry ratio drops sharply from purely reactive RAE to calling UPOM with 5 rollouts. From then onwards, until 250 rollouts, the retry ratio continues to decrease gradually. The behavior is similar in all domains, so we have combined the results together to show the rescaled average values in a single plot.

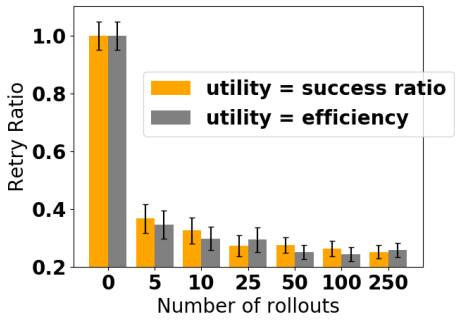

Figure 6: Retry ratio (# of retries / total # of incoming tasks) averaged over all five domains, for UPOM with $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

Efficiency across domains. In Figure 7 we detail for each domain the measured efficiency of RAE when the utility of UPOM was set to expected efficiency, for varying n_{ro} and $d_{max} = \infty$. Each data point is the average of 2500 runs. We observe that the efficiency generally improves with the number of rollouts. However, there is not much improvement with increase in n_{ro} in the Fetch domain, and in the Deliver domain, the efficiency drops slightly when $n_{ro} = 250$. We conjectured that this can be due to concurrent interfering tasks. Hence, we measured for Fetch and Deliver domains the efficiency for test cases with only one root task; the results in Figure 8 confirmed this conjecture.

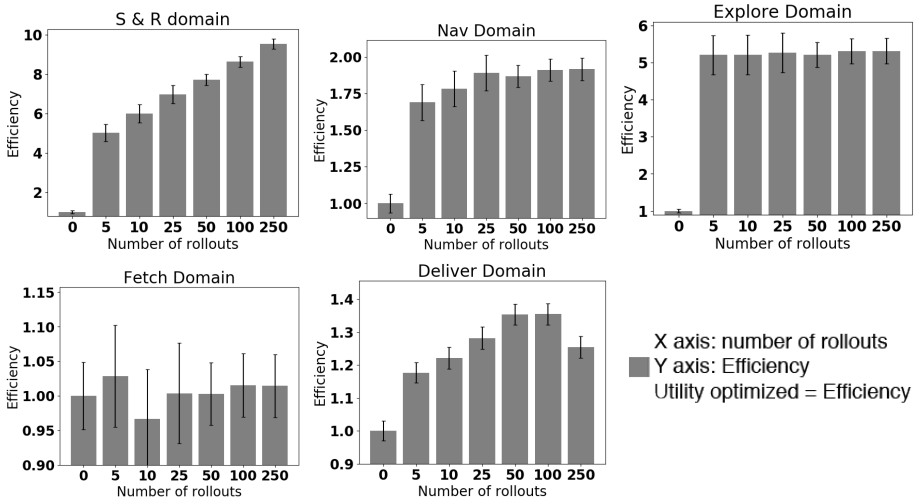

Figure 7: Measured efficiency of RAE for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$).

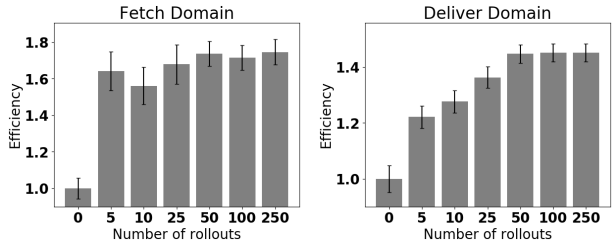

Figure 8: Measured efficiency averaged over only test cases with one root task, in Fetch and Deliver domains with $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

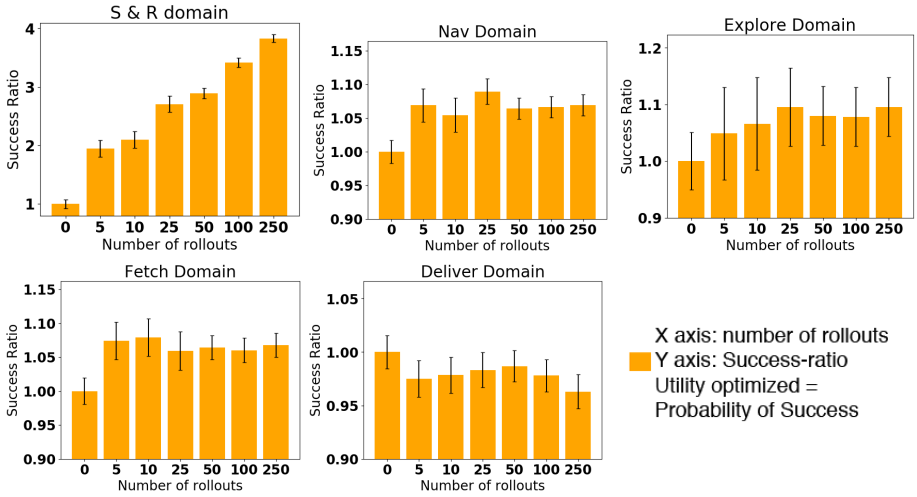

Figure 9: Measured success ratio (# of successful incoming tasks/ total # of incoming tasks) for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

Success ratio across domains. Figure 9 shows for each domain the measured success ratio of RAE when the utility of UPOM was set to expected success ratio, for varying n_{ro} and $d_{max} = \infty$. The success-ratio generally increases with increase in the number of rollouts. Again, a slight drop is observed in the Deliver domain. Figure 10 shows that for test cases with only one root task the success-ratio improves in the Fetch domain, and remains constant in the Deliver domain. The success ratio remains 1 in the Deliver domain because all test cases with one root task succeed eventually, with or without retries. In the domains with dead ends, the improvement in success ratio is more substantial than domains without dead ends because planning is more critical for cases where one bad choice of refinement method instance can lead to permanent failure.

Depth and Heuristics. We ran UPOM at different values of $d_{max} \in [0, 30]$, without progressive deepening in Select. At the depth limit, UPOM estimates the remaining efficiency using one of the following heuristic functions:

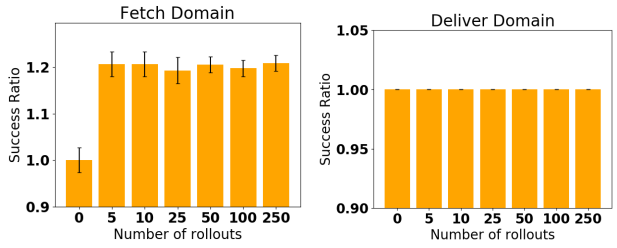

Figure 10: Measured success ratio averaged over only test cases with one root task, in Fetch and Deliver domains with $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

- h_0 always returns ∞;
- h_D is a hand written domain specific heuristic;
- h_{LearnH} is the heuristic function learned by the LearnH procedure (Section 6.3).

The results, in Figure 11, show that the efficiency generally increases with depth across all domains. In the Nav domain, the h_{LearnH} performs better than h_0 and h_D with 95% confidence at depths 2 and 3. In the Explore domain, h_{LearnH} performs better than h_0 and h_D at depth 1 with 95% confidence. The same is true for Fetch at depth 2. In the Deliver domain, the learned heuristic performs better than the others with 95% confidence for all depths $>= 1$. The performance difference between the three different heuristics are due to the properties of the domain, how the refinement methods are designed and how much of it is learnable by the LearnH procedure.

Measured vs expected efficiency. Each time RAE calls UPOM, UPOM uses its rollouts to make a prediction of expected efficiency. This predicted efficiency may differ from the measured efficiency that RAE achieves by time that it finishes. We will use the term *relative error* to denote the absolute value of this difference divided by the error in UPOM's initial prediction with a constant number of rollouts [9] (i.e., UPOM's prediction before RAE has performed any actions). Figure 12 shows the average relative error of UPOM's predictions, averaged over all 12,500 runs of our test problems, as a function of RAE's *progress*, i.e., how many actions RAE has performed since it began. Note that UPOM's relative error generally decreases as RAE's progress increases, and that it is quite small after just one or two actions.

7.3. Assessment of UPOM

We are not aware of any comparable planner for operational models, but of RAEplan [92], a Monte Carlo Tree Search procedure we developed earlier. We discuss

[9] We choose to plot the error relative to 5 UPOM rollouts.

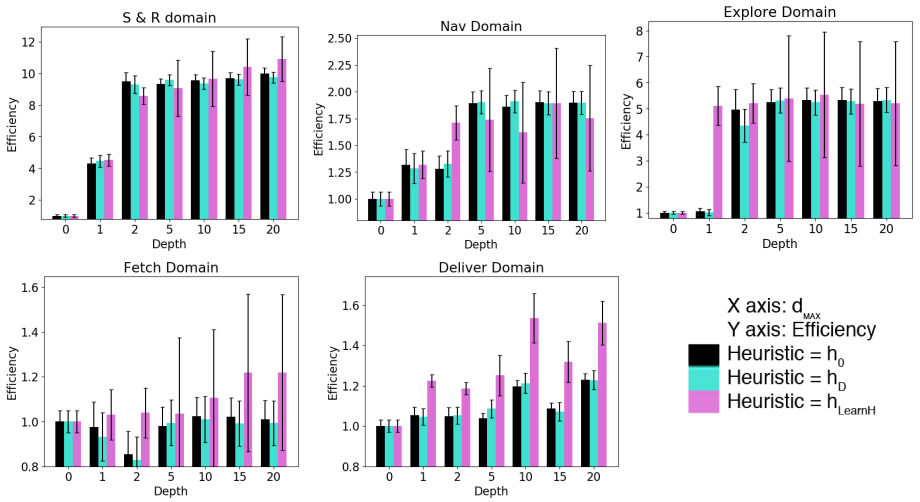

Figure 11: Measured efficiency with limited depth and three different heuristic functions. The utility function optimized is expected efficiency; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

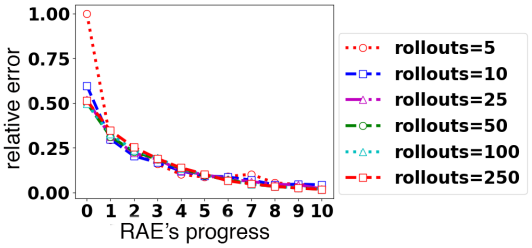

Figure 12: Average relative error of UPOM's predictions, for various numbers of rollouts by UPOM, shown as a function of RAE's progress (i.e., how many actions RAE has performed so far). Each data point is an average over all 12,500 runs of our test problems.

here RAE with UPOM *vs* RAEplan.[10] We configured UPOM to optimize the expected efficiency as its utility function, the same as RAEplan. In order not to favor the UCT strategy of UPOM with respect to the tree branching strategy of RAEplan, we set $n_{ro} = 1000$, with $d_{max} = \infty$ in each rollout.

Figure 13 shows the computation time for a single run of a problem (one or two root tasks), averaged across all domains and problems, i.e., over 12500 runs. RAE with UPOM runs more than twice as fast as RAE with RAEplan. Note that the computation time of RAE alone is negligible, since it is designed to be a fast reactive system, without search. However, in physical experiments, the total time includes sensing and actuation time, hence the planning overhead would not appear as significant as it is

[10] 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 [61].

here.

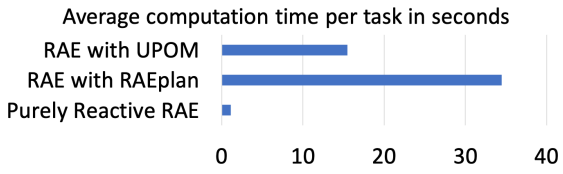

Figure 13: Average computation time in seconds for a single run of a problem, for RAE with and without the planners.

Efficiency. Figure 14 gives the measured efficiency for the five domains, with the 95% confidence intervals. It shows in all domains that RAE with UPOM is more efficient than purely reactive RAE and RAE with RAEplan.

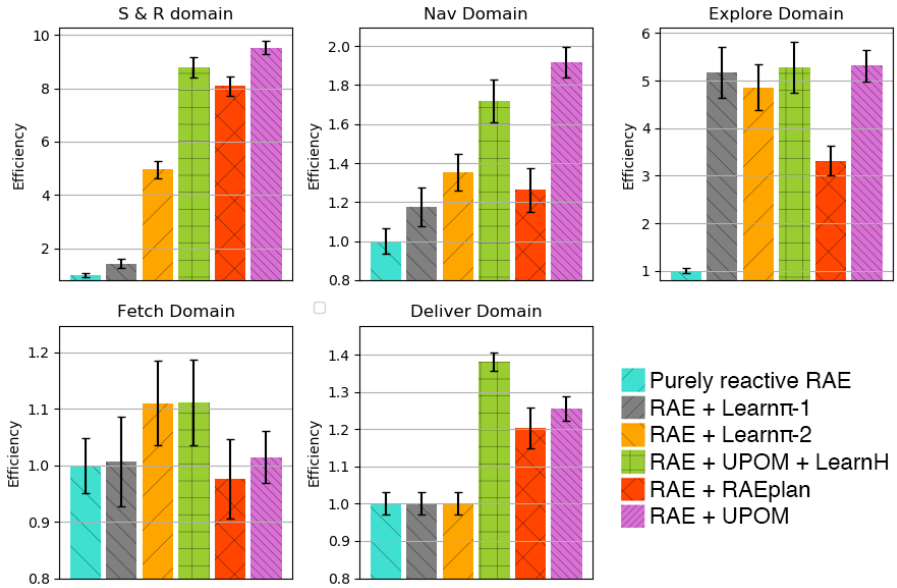

Figure 14: Measured efficiency for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

Success ratio. Figure 15 shows RAE's success ratio both with and without the planners. We observe that planning with UPOM outperforms purely reactive RAE in S&R and Fetch with 95% confidence, and Explore and Nav with 85% confidence. Also, UPOM outperforms RAEplan in Fetch and Nav domains with a 95% confidence, and Explore domain with 85% confidence. In the S&R domain, the success ratio is similar for RAEplan and UPOM.

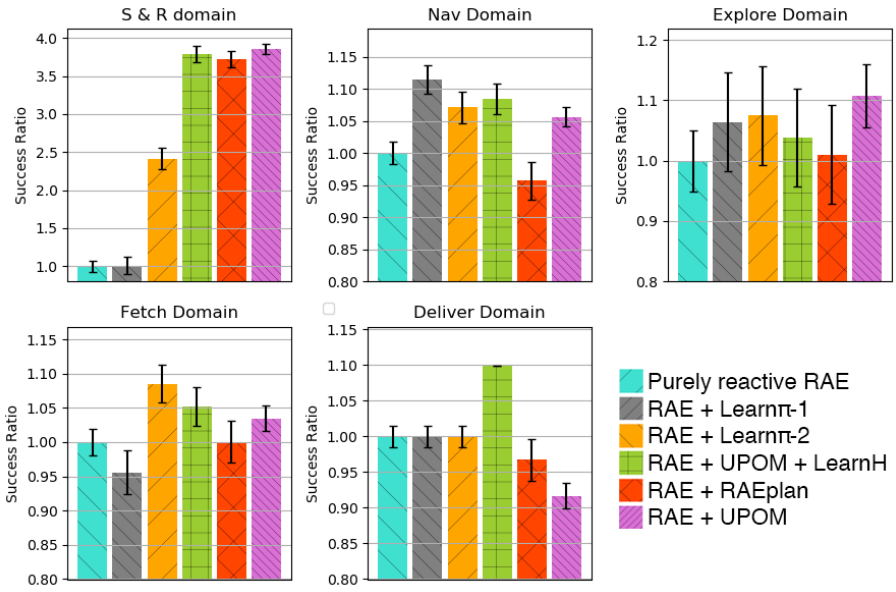

Figure 15: Measured success ratio for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$; Y axis rescaled with respect to the base case for $n_{ro} = 0$.

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.

7.4. Assessment of learning procedures

For training purposes, we synthesized data records for each domain by randomly generating root tasks and then running RAE with UPOM. The number of randomly generated tasks in S&R, Nav, Explore, Fetch, and Deliver domains are 96, 132, 189, 123, and 100 respectively. We save the data records according to the Learnπ-1, Learnπ-2, Learnπ_i and LearnH procedures, 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 accuracy increases with increase in the number of training epochs (see Figure 16).

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.

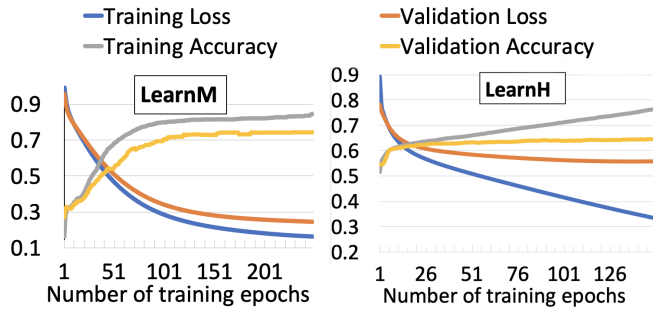

Figure 16: Training and validation results for Learnπ and LearnH, averaged over all domains.

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
S&R	250	634	3542	330	401	225	250	16	10
Nav	1686	5331	16251	126	144	750	150	9	75
Explore	2391	6883	10503	182	204	1000	250	17	200
Fetch	262	508	1084	97	104	430	250	10	100
Deliver	-	-	2001	-	627	-	250	-	10

Table 4: The size of the training set, number of input features and outputs, and the number of training epochs for three different learning procedures: Learnπ-1, Learnπ-2, and LearnH. We note LM-1 = Learnπ-1, LM-2 = Learnπ-2, and LH = LearnH.

Table 4 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 procedures. In the LearnH learning procedure, 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.

To test the learning procedures we measured the efficiency and success ratio of RAE with the policies learned by Learnπ-1 and Learnπ-2 without planning, and RAE with UPOM and the heuristic learned by LearnH. 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 LearnH, we set d_{max} to 5 and n_{ro} to 50, which has about 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 14 shows that RAE with UPOM + LearnH 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 Learnπ-1 and Learnπ-2 lies in between RAE with RAEplan and RAE with UPOM + LearnH, 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 these last 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 15, we observe that RAE with UPOM + LearnH 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 nondeterminism in the domain's design.

In a majority of the domains, the best efficiency and success ratio is achieved by either RAE with UPOM, or RAE with UPOM + LearnH. However, their computation times are quite different. Note that when UPOM is run with LearnH, the rollout lengths are quite shallow and the computation time is similar to purely reactive RAE (Figure 13). This makes it highy scalable. In contrast, RAE with UPOM explores each rollout to its maximum possible depth. This increases the computation time, making it less suitable for online usage with strict time constraints.

In most cases, we observe that RAE does better with Learnπ-2 than with Learnπ-1. Recall that the training set for Learnπ-2 is created with all methods 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 random exogenous events don't have a learnable pattern, and a smaller training set makes Learnπ-1's performance worse.

Domain	Method	Parameter	Training Set Size	#(input features)	#(outputs)
Nav	MoveThroughDoorway_M2	*robot*	404	150	4
	Recover_M1	*robot*	337	128	4
Deliver	Order_M1	*machine*	296	613	5
		objList	297	613	2
	Order_M2	*machine*	95	613	5
		objList	95	613	2
		pallet	95	613	4
	PickupAndLoad_M1	*robot*	244	637	7
	UnloadAndDeliver_M1	*robot*	219	625	7
	MoveToPallet_M1	*robot*	7	633	7

Table 5: The size of the training set, number of input features and outputs for learning method parameters in Learnπ_i.

Learning Method Instances. Two of our simulated domains, Nav and Deliver, have refinement methods with parameters that are not inherited from the task at hand. For these domains, Learnπ-1 and Learnπ-2 give only partially instantiated methods, while Learnπ_i is more discriminate. To test its benefit, we trained a MLP for each parameter not specified in the task. The size of the training set, number of input features and number of outputs are summarized in Table 5.

Figure 17 compares the efficiency of RAE with Learnπ_i vs purely reactive RAE and RAE with RAEplan, Learnπ-1, Learnπ-2, LearnH, and UPOM. In the Deliver domain, RAE with Learnπ_i is better than purely reactive RAE as well as RAE with Learnπ-1 or Learnπ-2 with 95% confidence. In the Nav domain, RAE with Learnπ_i also outperforms Learnπ-1 and purely reactive RAE with 95% confidence, but not Learnπ-2. The performance benefit is important in the Deliver domain because the refinement methods have several parameters that are not arguments of the task that the method

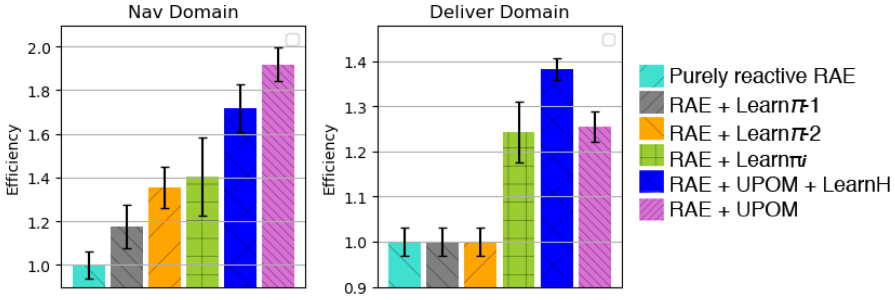

Figure 17: The cross-hatched blue bars show the performance of RAE with Learnπ_i (learning method instances) for the two domains, Nav and Deliver, which have methods with parameters not in tasks (Y axis rescaled with respect to the base case for $n_{ro} = 0$).

is used for, hence there are combinatorially many different ways to instantiate these parameters when creating method instances.

In summary, for all the domains, planning with UPOM and learning clearly outperforms purely reactive RAE.

8. Discussion

We now discuss several issues that readers may find helpful for using and deploying RAE and UPOM in practical applications. We also point out several limitations and topics for future work.

8.1. Retrial in RAE

As mentioned earlier, Retry is not a backtracking procedure. Since RAE interacts with a dynamic world, Retry cannot go back to a previous state. It selects a method instance among those applicable in the *current* world state, except for those that have been tried before and failed. This latter restriction may not always be necessary, since the same method instance that failed at some point may succeed later on. It can be complicated to analyze the conditions responsible for failures and ascertain whether they still hold. However, RAE can be adapted to retrial of method instances if they are vulnerable to noisy sensing, or if the execution context is one in which they should be retried. For example, one may give the methods additional parameters that are not needed for the logic of the method instance but that characterize the context (e.g., the pose of a sensor that may have changed between trials), while bounding the number of retrials.

Retrial can be applied more easily for actions. In RAE, a method instance fails when one of its actions fails. But if an action has nondeterministic outcomes, it may be worthwhile retrying the action as assessed by its expected utility. This may be implemented after a full analysis and the computation of an optimal MDP policy,[11] or

[11] This can be done with a sequence of dummy states s_{fail_i} such that the effects of action a in s include $s_1^{fail} \in \gamma(s, a)$, $s_2^{fail} \in \gamma(s_1^{fail}, a)$, ..., $s_{i+1}^{fail} \in \gamma(s_i^{fail}, a)$, with two actions being applicable to each s_i^{fail}: a and stop-with-failure.

simply with an ad-hoc loop on the execution-status of the actions that merit retrials. Furthermore, the body of a method is a procedure in which one can specify complex retrial loops. For example, a difficult grasp action in robotics may need several sequences of ⟨move, sense, grasp⟩ before succeeding or giving up.

In the Deliver domain, we implemented a retrial procedure by having the actions set a status describing if they should be retried after failing. This status depends on the nature of the failure, with likely unrectifiable failures such as unsatisfied preconditions not being retried. These actions were called from a wrapper task that can repeat the command up to two additional times.

8.2. Concurrency

RAE's *Agenda* contains several refinement stacks, one for each top-level task, and the purpose of RAE's main loop is to progress these stacks concurrently. Each of the experimental domains in Section 7 used this facility and involved concurrent tasks. However, our current implementation provides no built-in way to manage possible conflicts and needed synchronizations; instead they must be managed by the refinement methods. We handled conflict in our S&R, Nav, and Deliver domains by having state variables to specify the status of the resources. The task waiting for the resource executes no action until that resource is available again. We use semaphores to avoid race conditions. Such refinement methods could be made easier to write by extending RAE to provide synchronization constructs such as those used in TCA and TDL [108, 109], and rely on the execution-status of actions to handle waits. Since both UPOM and learning rely on the simulation of the method instances, they could support such extensions as long as Sample can simulate the duration of actions. More research is needed on how to extend RAE to permit formal verification of concurrency properties (liveness, deadlocks), e.g., as in the Petri-net based reactive system ASPiC [72].

Note that it is possible to extend RAE to allow the body of a method to specify concurrent subtasks (see [37, Sect. 3.2.4]). However, our current version of RAE does not include such a facility.

8.3. Learning operational models

The Learnπ and LearnH procedures are designed to improve the decision making of RAE, with or without planning. They are also of help to a domain author, who does not need to design a minimal set of methods associated to a preference ordering. However, assistance in acquiring operational models would be highly desirable. Let us raise few remarks about this important issue of future work.

Actions and methods, the two main components of operational models, demand different learning techniques. Execution models of actions are often domain dependent. For example, in robotics several approaches have been proposed for learning primitive actions, e.g., [96, 64, 45, 17], usually relying on *Reinforcement Learning* (RL), possibly supervised and/or with inverse RL (see survey [65]). Other techniques for learning actions as low-level skills may also be relevant, e.g., [13, 124]. These and similar techniques would provide operational models of primitive actions needed by RAE, as well as a domain simulator needed by UPOM for the function Sample (see line 7 of Algorithm 6). However, since UPOM may call Sample many times during its Monte Carlo rollouts, a detailed domain simulator may have a high computational overhead. A learned domain independent but shallow model of primitive actions, e.g., [91, 69], would provide an efficient Sample function.

Regarding refinement methods, several techniques have been developed for learning HTN methods, e.g., [49, 47, 48, 125, 124]. However, our refinement methods for operational models can be significantly more complex. Possible avenues of investigation for synthesizing these methods include program synthesis techniques [116, 40], partial programming and RL [1, 74, 110], as well as learning from the demonstrations of a tutor [2].

8.4. Potential and limitations

The work presented in this paper includes a reactive system (RAE), extended with capabilities for planning (UPOM) and learning (Learnπ, Learnπ_i, LearnH). It is intended to empower an autonomous system facing a diversity of tasks in nondeterministic dynamic environments with robust and efficient deliberation competences. It makes use of a hierarchical representation of tasks and refinement methods, and abstraction of the reactive system's states for use in the planning and learning algorithms.

Beyond the aspects stressed earlier, there are several possible ways to extend the work:

- In addition to planning and acting, a deliberative system needs also *monitoring*. This function fits naturally in our framework through methods for handling alarms and other surveilled events. Many fault detection, identification and recovery systems, e.g., in space and critical applications [50, 97], can use approaches like RAE. Furthermore, RAE and UPOM have been used in a prototype cybersecurity monitoring and recovery system (see Section 8.5) in which monitoring functions invoke RAE and UPOM when they are needed to plan recovery from cyber-attacks.
- Although *time* and temporal primitives have not been introduced in our relatively simple representation, these constructs are widely used in several reactive langages, such as TCA/TDL or Petri Net based systems. Offering similar facilities in RAE appears quite feasible, but the extensions in UPOM will require more work.
- Similarly, we have not covered space and motion planning. But these can be added in RAE, e.g., as external functions conditioning particular actions. They can be part of the simulations performed in UPOM.

There also are several limitations, two of which need to be underlined:

- The approach developed in this paper is not suitable for addressing intensive combinatorial search and optimization problems. For example, while the approach is adequate for a video game such as Starcraft; for board games such as chess or go, it is better to use game-tree search and learning techniques [88, 106]. Similarly, the organization of numerous resources over a long horizon in a well-modeled predictable environment are better addressed with temporal planning and scheduling tools, e.g., [23]. However, it is conceivable to connect such a tool for assigning tasks to actors that are driven by our approach (see [37, Section 4.5]).
- Our approach, as presented, does not integrate general inference-making mechanisms, e.g., in order to deduce from common sense or general knowledge that in some contexts, additional care is necessary for an activity to succeed. Right now, such a situation would require refinement methods that are adapted to that context; and in their absence, UPOM would find that all of the available methods fail. Possible avenues of response to this limitation include online continual learning of methods, and extending RAE to include an inference engine that uses general and domain-specific axioms [86, 113].

8.5. Deployment in new applications

To deploy the proposed approach in a new domain, it is necessary to have an execution platform or the equivalent collection of sensing and primitive actions, as well as a set of methods. If the tasks and events are well-defined and the techniques for handling them are known, human experts may be able to write the methods without too much difficulty.

RAE executes methods and triggers the execution of primitive actions by the platform. UPOM has to simulate both. Simulating refinement methods is not an issue, but more effort may be needed to simulate the execution platform and its environment. Clearly, the reliability of such a simulator affects the quality of UPOM's steering. It is not straightforward to develop a reliable simulator that reproduces the dynamics of a nondeterministic uncertain environment. In some cases, available simulation tools can be useful, e.g., physics-based simulations [6, 24, 8], robotic simulations [80, 70, 104], automated manufacturing simulations [54, 82] and so forth. Some of these tools are often used for the specification and design of an execution platform, which may simplify their use for the development of a simulator. A fallback option, easily applicable in most cases, would be to define the procedure Sample by sampling the possible outcomes of every action from probability distributions, which are initialized by a human expert, then refined by learning and experiments. The results in Section 7 show that this shallow simulation already provides substantial improvements in decision-making. Note that it is possible to combine detailed simulations for critical actions, for which tools might be available, and shallow simulations for the remaining actions.

The deployment of RAE and UPOM in a prototype application for security monitoring and recovery from attacks on Software-Defined Networks (SDN) is described in [95]. It defines an abstract representation of the SDN's state used by RAE and UPOM, hand-programmed primitive actions that could be executed on the SDN, tasks and events regarding attacks on an SDN, and a set of refinement methods giving possible recovery procedures for each of the attacks. Several refinement method are applicable to the same task or event; the human expert does not need to specify which applicable method is preferable in which contexts, since the purpose of UPOM is to make such choices online. The final system has been evaluated by SDN experts successfully.

The development effort required about three person-months of work by the human expert, but the human expert's ongoing experience with RAE and UPOM revealed software features that we needed to add so that his development effort could proceed. Given RAE and UPOM in their current form, we think such a development effort might take much less time.

9. Conclusion

We have presented a novel system for integrating acting and planning using hierarchical refinement operational models. The refinement acting engine, RAE, can either run purely reactively or it can get advice from an online planner to choose efficient method instances for performing a task. The planning procedure, UPOM, uses an anytime search strategy inspired by UCT, extended to operate in a more complicated search space. UPOM provides near-optimal method instances with respect to utility functions that may be quite general, and it converges asymptotically. We have proposed two distinct utility functions that favor efficiency and robustness, respectively. UPOM is integrated with RAE using a receding-horizon, anytime progressive-deepening procedure.

We have also presented three learning strategies: Learnπ, to learn a mapping from a task in a given context to a good method, Learnπ_i, to learn values of uninstantiated method parameters, and LearnH, to learn a domain specific heuristic function for our hierarchical refinement framework. We have shown how incremental learning can be integrated online with acting and planning.

We have presented empirical results over five domains that have challenging features such as dynamicity, dead-ends, exogenous events, sensing and information gathering actions, collaborative and concurrent tasks. Rather than just evaluating the system's planning functionality, we have devised simulations and measurements that assess its overall acting performance, with and without planning and learning, taking into account exogenous events and failure cases.

We have measured the actor's efficiency, success ratio and retry ratio, and discussed their relationships with respect to the planner's utility function, maximizing either the expected efficiency or the expected success ratio. Our results show that Learnπ improves the performance of reactive RAE with respect to the three performance measures, and they are improved even further when RAE is used either with UPOM and LearnH or with UPOM at unbounded search depth. Thanks to learning, the computational overhead remains acceptable for an online procedure, since in this case a small number of rollouts already bring a good benefit.

In summary, acting purely reactively in dynamic domains with dead ends can be costly and risky. Our proposed integration of acting, planning and learning can be of great benefit, reflected by a higher efficiency or robustness. An open source repository of all of the algorithms and test domains is available at [**?**].

Acknowledgements. We are very grateful to Félix Ingrand, LAAS-CNRS, for very fruitful discussions regarding the initial specification of RAE's algorithms, and to Amit Kumar, U. Maryland, for advices regarding the multi-layer perceptron design and training. Many thanks to the anonymous reviewers for their very positive and insightful remarks. This work has been supported in part by NRL grants N00173191G001 and N0017320P0399, ONR grant N000142012257, and AFOSR grant 1010GWA357. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

[1] Andre, D., Russell, S.J., 2002. State abstraction for programmable reinforcement learning agents, in: AAAI.

[2] Argall, B.D., Chernova, S., veloso, M.M., Browning, B., 2009. A survey of robot learning from demonstration. Robotics and Autonomous Systems 57, 469–483.

[3] Bäckström, C., Nebel, B., 1995. Complexity results for SAS+ planning. Computational Intelligence 11, 625–655.

[4] Barry, J.L., Kaelbling, L.P., Lozano-Pérez, T., 2013. A hierarchical approach to manipulation with diverse actions, in: ICRA, pp. 1799–1806.

[5] Beetz, M., McDermott, D., 1994. Improving robot plans during their execution, in: AIPS.

[6] Boeing, A., Bräunl, T., 2007. Evaluation of real-time physics simulation systems, in: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia, pp. 281–288.

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

[8] Bridson, R., 2015. Fluid Simulation for Computer Graphics. CRC Press.

[9] Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147, 35–84.

[10] Claßen, J., Röger, G., Lakemeyer, G., Nebel, B., 2012. Platas—integrating planning and the action language Golog. KI-Künstliche Intelligenz 26, 61–67.

[11] Colledanchise, M., 2017. Behavior Trees in Robotics. Ph.D. thesis. KTH, Stockholm, Sweden.

[12] Colledanchise, M., 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, 372–389.

[13] Colledanchise, M., Parasuraman, R., Ögren, P., 2018. Learning of behavior trees for autonomous agents. IEEE Transactions on Games 11, 183–189.

[14] Conrad, P., Shah, J., Williams, B.C., 2009. Flexible execution of plans with choice, in: ICAPS.

[15] De Silva, L., Meneguzzi, F., Logan, B., 2020. BDI agent architectures: A survey, in: IJCAI.

[16] De Silva, L., Meneguzzi, F.R., Logan, B., 2018. An operational semantics for a fragment of prs, in: IJCAI.

[17] Deisenroth, M.P., Neumann, G., Peters, J., 2013. A survey on policy search for robotics. Foundations and Trends in Robotics 2, 1–142.

[18] Despouys, O., Ingrand, F., 1999. Propice-Plan: Toward a unified framework for planning and execution, in: ECP.

[19] Doherty, P., Kvarnström, J., Heintz, F., 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. J. Autonomous Agents and Multi-Agent Syst. 19, 332–377.

[20] Duong, T.V., Phung, D.Q., Bui, H.H., Venkatesh, S., 2009. Efficient duration and hierarchical modeling for human activity recognition. Artificial Intelligence 173, 830–856.

[21] Effinger, R., Williams, B., Hofmann, A., 2010. Dynamic execution of temporally and spatially flexible reactive programs, in: AAAI Wksp. on Bridging the Gap between Task and Motion Planning, pp. 1–8.

[22] Erol, K., Nau, D.S., Subrahmanian, V.S., 1995. Complexity, decidability and undecidability results for domain-independent planning. Artificial Intelligence 76, 75–88.

[23] Estlin, T., Gaines, D., Chouinard, C., Castano, R., Bornstein, B., Judd, M., Nesnas, I., Anderson, R., 2007. Increased mars rover autonomy using ai planning, scheduling and execution, in: ICRA, IEEE. pp. 4911–4918.

[24] Faure, F., Duriez, C., Delingette, H., Allard, J., Gilles, B., Marchesseau, S., Talbot, H., Courtecuisse, H., Bousquet, G., Peterlik, I., et al., 2012. Sofa: A multi-model framework for interactive physical simulation, in: Soft tissue biomechanical modeling for computer assisted surgery. Springer, pp. 283–321.

[25] Feldman, Z., Domshlak, C., 2013. Monte-carlo planning: Theoretically fast convergence meets practical efficiency, in: UAI.

[26] Feldman, Z., Domshlak, C., 2014. Monte Carlo tree search: To MC or to DP?, in: ECAI, pp. 321–326.

[27] Ferrein, A., Lakemeyer, G., 2008. Logic-based robot control in highly dynamic domains. Robotics and Autonomous Systems 56, 980–991.

[28] Fikes, R.E., Nilsson, N.J., 1971. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 189–208.

[29] Firby, R.J., 1987. An investigation into reactive planning in complex domains, in: AAAI, pp. 202–206.

[30] Fox, M., Long, D., 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. J. Artificial Intelligence Research 20, 61–124.

[31] Fox, M., Long, D., 2006. Modelling mixed discrete-continuous domains for planning. J. Artificial Intelligence Research 27, 235–297.

[32] Garnelo, M., Arulkumaran, K., Shanahan, M., 2016. Towards deep symbolic reinforcement learning. arXiv:1609.05518 .

[33] Garrett, C.R., Lozano-Perez, T., Kaelbling, L.P., 2018a. FFRob: Leveraging symbolic planning for efficient task and motion planning. The International Journal of Robotics Research 37, 104–136.

[34] Garrett, C.R., Lozano-Pérez, T., Kaelbling, L.P., 2018b. STRIPStream: Integrating symbolic planners and blackbox samplers. arXiv:1802.08705 .

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

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

[37] Ghallab, M., Nau, D.S., Traverso, P., 2016. Automated Planning and Acting. Cambridge University Press.

[38] Goldman, R.P., 2009. A semantics for HTN methods, in: ICAPS.

[39] Goldman, R.P., Bryce, D., Pelican, M.J., Musliner, D.J., Bae, K., 2016. A hybrid architecture for correct-by-construction hybrid planning and control, in: NASA Formal Methods Symposium, Springer. pp. 388–394.

[40] Gulwani, S., Polozov, O., Singh, R., et al., 2017. Program synthesis. Foundations and Trends in Programming Languages 4, 1–119.

[41] Hähnel, D., Burgard, W., Lakemeyer, G., 1998. GOLEX – bridging the gap between logic (GOLOG) and a real robot, in: KI, Springer. pp. 165–176.

[42] Haslum, P., Lipovetzky, N., Magazzeni, D., Muise, C., 2019. An Introduction to the Planning Domain Definition Language. Morgan & Claypool.

[43] Hauskrecht, M., Meuleau, N., Kaelbling, L.P., Dean, T.L., Boutilier, C., 2013. Hierarchical solution of Markov decision processes using macro-actions. arXiv:1301.7381 .

[44] Henaff, M., Canziani, A., LeCun, Y., 2019. Model-predictive policy learning with uncertainty regularization for driving in dense traffic. arXiv:1901.02705 .

[45] Hester, T., Stone, P., 2012. TEXPLORE: Real-Time Sample-Efficient Reinforcement Learning for Robots, in: AAAI Spring Symposium, pp. 1–6.

[46] Hitzler, P., Wendt, M., 2005. A uniform approach to logic programming semantics. Theory and Practice of Logic Programming 5, 93–121.

[47] Hogg, C., Kuter, U., Muñoz-Avila, H., 2009. Learning hierarchical task networks for nondeterministic planning domains, in: IJCAI, pp. 1708–1714.

[48] Hogg, C., Kuter, U., Muñoz-Avila, H., 2010. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning, in: AAAI.

[49] Hogg, C., Muñoz-Avila, H., Kuter, U., 2008. HTN-MAKER: learning HTNs with minimal additional knowledge engineering required, in: AAAI, pp. 950–956.

[50] Hwang, I., Kim, S., Kim, Y., Seah, C.E., 2010. A survey of fault detection, isolation, and reconfiguration methods. IEEE Trans. Control. Syst. Technol. 18, 636–653.

[51] Ingham, M.D., Ragno, R.J., Williams, B.C., 2001. A reactive model-based programming language for robotic space explorers, in: i-SAIRAS.

[52] Ingrand, F., Chatilla, R., Alami, R., Robert, F., 1996. PRS: A high level supervision and control language for autonomous mobile robots, in: ICRA, pp. 43–49.

[53] Ingrand, F., Ghallab, M., 2017. Deliberation for Autonomous Robots: A Survey. Artificial Intelligence 247, 10–44.

[54] Jahangirian, M., Eldabi, T., Naseer, A., Stergioulas, L.K., Young, T., 2010. Simulation in manufacturing and business: A review. European Journal of Operational Research 203, 1–13.

[55] James, S., Konidaris, G., Rosman, B., 2017. An analysis of Monte Carlo tree search, in: AAAI, pp. 3576–3582.

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

[57] Jonsson, P., Bäckström, C., 1998. State-variable planning under structural restrictions: Algorithms and complexity. Artificial Intelligence 100, 125–176.

[58] Kaelbling, L.P., Littman, M.L., Moore, A.W., 1996. Reinforcement learning: A survey. JAIR 4, 237–285.

[59] Kaelbling, L.P., Lozano-Perez, T., 2011. Hierarchical task and motion planning in the now, in: ICRA, pp. 1470–1477.

[60] Kaelbling, L.P., Lozano-Perez, T., 2013. Integrated task and motion planning in belief space. Intl. J. Robotics Research 32, 1194–1227.

[61] 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.

[62] Katt, S., Oliehoek, F.A., Amato, C., 2017. Learning in POMDPs with Monte Carlo tree search, in: ICML.

[63] Keller, T., Eyerich, P., 2012. PROST: Probabilistic planning based on UCT. ICAPS , 119–127.

[64] Kober, J., 2012. Learning Motor Skills: From Algorithms to Robot Experiments. Ph.D. thesis. Darmstadt University.

[65] Kober, J., Bagnell, J.A., Peters, J., 2013. Reinforcement Learning in Robotics: A Survey. International Journal of Robotics Research .

[66] Kocsis, L., Szepesvári, C., 2006. Bandit based Monte Carlo planning, in: ECML, pp. 282–293.

[67] Kortenkamp, D., Simmons, R., 2008. Robotic Systems Architectures and Programming, in: Siciliano, B., Khatib, O. (Eds.), Springer Handbook of Robotics. Springer, pp. 187–206.

[68] Lallement, R., De Silva, L., Alami, R., 2014. HATP: an HTN planner for robotics. arXiv:1405.5345 .

[69] Lang, T., Toussaint, M., Kersting, K., 2012. Exploration in relational domains for model-based reinforcement learning. The Journal of Machine Learning Research 13, 3725–3768.

[70] León, B., Ulbrich, S., Diankov, R., Puche, G., Przybylski, M., Morales, A., Asfour, T., Moisio, S., Bohg, J., Kuffner, J., et al., 2010. Opengrasp: a toolkit for robot grasping simulation, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer. pp. 109–120.

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

[72] Lesire, C., Pommereau, F., 2018. ASPiC: an acting system based on skill Petri net composition, in: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE. pp. 6952–6958.

[73] Levine, S.J., Williams, B.C., 2014. Concurrent plan recognition and execution for human-robot teams, in: ICAPS.

[74] Marthi, B.M., Russell, S.J., Latham, D., Guestrin, C., 2005. Concurrent hierarchical reinforcement learning, in: AAAI, p. 1652.

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

[76] Martínez, D.M., Alenyà, G., Torras, C., 2017b. Relational reinforcement learning with guided demonstrations. Artificial Intelligence 247, 295–312.

[77] Mausam, A.K., 2012. Planning with markov decision processes: an ai perspective. Morgan & Claypool Publishers .

[78] McDermott, D.M., 2000. The 1998 AI planning systems competition. AI Mag. 21, 35.

[79] Meneguzzi, F., De Silva, L., 2015. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. The Knowledge Engineering Review 30, 1–44.

[80] Michel, O., 2004. Cyberbotics Ltd. Webots™: professional mobile robot simulation. International Journal of Advanced Robotic Systems 1, 5.

[81] Morisset, B., Ghallab, M., 2008. Learning how to combine sensory-motor functions into a robust behavior. Artificial Intelligence 172, 392–412.

[82] Mourtzis, D., Doukas, M., Bernidaki, D., 2014. Simulation in manufacturing: Review and challenges. Procedia Cirp 25, 213–229.

[83] Muscettola, N., Nayak, P.P., Pell, B., Williams, B.C., 1998. Remote Agent: To boldly go where no AI system has gone before. Artificial Intelligence 103, 5–47.

[84] Musliner, D.J., Pelican, M.J., Goldman, R.P., Krebsbach, K.D., Durfee, E.H., 2008. The evolution of CIRCA, a theory-based ai architecture with real-time performance guarantees., in: AAAI Spring Symposium: Emotion, Personality, and Social Behavior.

[85] Myers, K.L., 1999. CPEF: A continuous planning and execution framework. AI Mag. 20, 63–69.

[86] Nau, D.S., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F., 2003. SHOP2: An HTN planning system. J. Artificial Intelligence Research 20, 379–404.

[87] Nau, D.S., Cao, Y., Lotem, A., Muñoz-Avila, H., 1999. SHOP: Simple hierarchical ordered planner, in: IJCAI, pp. 968–973.

[88] Newborn, M., 2013. Deep Blue: an artificial intelligence milestone. Springer Science & Business Media.

[89] Parr, R., Russell, S.J., 1997. Reinforcement learning with hierarchies of machines, in: NIPS.

[90] Parr, R., Russell, S.J., 1998. Reinforcement learning with hierarchies of machines, in: Advances in neural information processing systems, pp. 1043–1049.

[91] Pasula, H.M., Zettlemoyer, L.S., Kaelbling, L.P., 2007. Learning symbolic models of stochastic domains. Journal of Artificial Intelligence Research 29, 309–352.

[92] Patra, S., Ghallab, M., Nau, D., Traverso, P., . Acting and planning using operational models, in: AAAI.

[93] Patra, S., Ghallab, M., Nau, D., Traverso, P., 2018. Ape: An acting and planning engine. Advances in Cognitive Systems .

[94] Patra, S., Mason, J., Kumar, A., Ghallab, M., Traverso, P., Nau, D., 2020. Integrating acting, planning, and learning in hierarchical operational models, in: ICAPS.

[95] Patra, S., Velasquez, A., Kang, M., Nau, D., 2021. Using online planning and acting to recover from cyberattacks on software-defined networks, in: IAAI.

[96] Peters, J., Kober, J., Nguyen-Tuong, D., 2008. Policy Learning–a unified perspective with applications in robotics. Recent Advances in Reinforcement Learning , 220–228.

[97] Pettersson, O., 2005. Execution monitoring in robotics: A survey. Robotics and Autonomous Systems 53, 73–88.

[98] Ramchandani, N., 2019. Virtual coaching to enhace diabetes care. Diabetes Technology and Therapeutics 21, S2–48–S2–51.

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

[100] Ryan, M.R.K., 2002. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies, in: ICML.

[101] Sanner, S., 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. Technical Report. NICTA.

[102] Santana, P.H.R.Q.A., Williams, B.C., 2014. Chance-constrained consistency for probabilistic temporal plan networks, in: ICAPS.

[103] Sardina, S., de Silva, L., Padgham, L., 2006. Hierarchical planning in BDI agent programming languages: A formal approach, in: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pp. 1001–1008.

[104] Shah, S., Dey, D., Lovett, C., Kapoor, A., 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles, in: Field and service robotics, Springer. pp. 621–635.

[105] de Silva, L., Meneguzzi, F., Logan, B., 2018. An Operational Semantics for a Fragment of PRS, in: IJCAI.

[106] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science 362, 1140–1144.

[107] Simmons, R., 1992. Concurrent planning and execution for autonomous robots. IEEE Control Systems 12, 46–50.

[108] Simmons, R., 1994. Structured control for autonomous robots. IEEE Trans. Robotics and Automation 10, 34–43.

[109] Simmons, R., Apfelbaum, D., 1998. A task description language for robot control, in: IROS, pp. 1931–1937.

[110] Simpkins, C., Bhat, S., Isbell, Jr., C., Mateas, M., 2008. Towards adaptive programming: integrating reinforcement learning into a programming language, in: ACM SIGPLAN Conf. on Object-Oriented Progr. Syst., Lang., and Applications (OOPSLA), ACM. pp. 603–614.

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

[112] Teichteil-Königsbuch, F., Infantes, G., Kuter, U., 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure, in: ICAPS.

[113] Thiébaux, S., Hoffmann, J., Nebel, B., 2005. In defense of PDDL axioms. Artificial Intelligence 168, 38–69.

[114] Veloso, M.M., Biswas, J., Coltin, B., Rosenthal, S., 2015. Cobots: Robust symbiotic autonomous mobile service robots., in: IJCAI, p. 4423.

[115] Verma, V., Estlin, T., Jónsson, A.K., Pasareanu, C., Simmons, R., Tso, K., 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences, in: i-SAIRAS.

[116] Walkinshaw, N., Taylor, R., Derrick, J., 2016. Inferring extended finite state machine models from software executions. Empirical Software Engineering 21, 811–853.

[117] Wang, F.Y., Kyriakopoulos, K.J., Tsolkas, A., Saridis, G.N., 1991. A Petri-net coordination model for an intelligent mobile robot. IEEE Trans. Syst., Man, and Cybernetics 21, 777–789.

[118] Williams, B.C., Abramson, M., 2001. Executing reactive, model-based programs through graph-based temporal planning, in: IJCAI.

[119] Wolfe, J., Marthi, B., 2010. Combined task and motion planning for mobile manipulation, in: International Conference on Automated Planning and Scheduling, pp. 254–257.

[120] Yang, F., Lyu, D., Liu, B., Gustafson, S., 2018. PEORL: integrating symbolic planning and hierarchical reinforcement learning for robust decision-making, in: IJCAI.

[121] Yoon, S.W., Fern, A., Givan, R., 2007. FF-Replan: A baseline for probabilistic planning., in: ICAPS, pp. 352–359.

[122] Yoon, S.W., Fern, A., Givan, R., Kambhampati, S., 2008. Probabilistic planning via determinization in hindsight., in: AAAI, pp. 1010–1016.

[123] Younes, H., Littman, M., 2004. PPDDL: The probabilistic planning domain definition language. Technical Report. CMU.

[124] Zhang, Q., Yao, J., Yin, Q., Zha, Y., 2018. Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution. Applied Sciences 8, 1077.

[125] Zhuo, H.H., Hu, D.H., Hogg, C., Yang, Q., Muñoz-Avila, H., 2009. Learning HTN method preconditions and action models from partial observations, in: IJCAI, pp. 1804–1810.

Appendix A. Asymptotic Convergence of UPOM

In this section we demonstrate the asymptotic convergence of UPOM towards an optimal method, as $n_{ro} \to \infty$. The proof assumes no depth cut-off ($d_{max} = \infty$) and static domains, i.e., domains without exogenous events. We believe it would be straightforward to extend the proof to dynamic domains if there are known probability distributions over the occurrence of exogenous events. However, we have not attempted to extend the proof to that case.

The proof proceeds by mapping UPOM's search strategy into UCT, which is demonstrated to converge on a finite horizon MDP with a probability of not finding the optimal action at the root node that goes to zero at a polynomial rate as the number of rollouts grows to infinity (Theorem 6 of [66]).

To simplify the mapping, we first consider UPOM with an additive utility function, and show how to map UPOM's search space into an MDP. We then discuss how this can be extended to the efficiency and success ratio utility functions defined in 5, using the fact that the UCT algorithm is not restricted to the additive case; it still converges as long as the utility function is monotonic.

A.1. Search Space for Refinement Planning

Let $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$ be an acting domain, as specified at the end of Section 3. Throughout this appendix, we will assume that Σ is static.

Recall from Section 5 that the space searched by UPOM is a simulated version of Σ. To talk about this formally, recall that a *refinement planning domain* is a tuple $\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$, where S is the set of states (recall that these are abstractions of states in Ξ), and $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are the same as in Σ. Recall from Section 3 that S, $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are all finite, and that every sequence of steps generated by the methods in $\mathcal{M}$ is finite.

For $s \in S$ and $a \in \mathcal{A}$, we let $\gamma(s,a) \subseteq S$ be the set of all states that may be produced by simulating a's execution in s. For each $s' \in \gamma(s,a)$, we let $P(s,a,s')$ be the probability that state s' will be produced if we simulate a's execution in state s.

Recall from Section 4 that a *refinement stack* is a LIFO stack in which each element is a tuple $(\tau, m, i, \mathit{tried})$, where τ is a task, m is a method, i is an instruction pointer that points to the i'th line of m's body (which is a computer program), and *tried* is the set of methods previously tried for τ. We will call the tuple $(\tau, m, i, \mathit{tried})$ a *stack frame*, and we will let $m[i]$ denote the i'th line of the body of m.

We now can define a *refinement planning problem* to be a tuple $\Pi = (\Phi, s_0, \sigma_0, U)$, where s_0 is the initial state, σ_0 is the initial refinement stack, and U is a utility function.

Rollouts. A *rollout* in Φ is a sequence of pairs

$$\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n) \rangle \tag{A.1}$$

satisfying the following properties:

- each s_i is a state, and each σ_i is a refinement stack;
- for each $i > 0$ there is a nonzero probability that s_j and σ_j are the next state and refinement stack after s_{i-1} and σ_{i-1};
- (σ_n, s_n) is a termination point for UPOM.

If the final refinement stack is $\sigma_n = \langle\rangle$, i.e., the empty stack, then the rollout ρ is successful. Otherwise ρ fails.

In a top-level call to UPOM, the initial refinement stack σ_0 would normally be

$$\sigma_0 = \langle (\tau_0, m_0, 1, \varnothing) \rangle, \tag{A.2}$$

where τ_0 is a task, and m_0 is a method that is relevant for τ_0 and applicable in s_0. In all subsequent refinement stacks produced by UPOM.

We will say that a refinement stack σ is *reachable* in Φ (i.e., reachable from a top-level call to UPOM) if there exists a rollout

$$\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n) \rangle$$

such that σ_0 satisfies Equation A.2 and $\sigma \in \{\sigma_0, \ldots, \sigma_n\}$. We let $\mathcal{R}(\Phi)$ be the set of all refinement stacks that are reachable in Φ. Since every sequence of steps generated by the methods in $\mathcal{M}$ is finite, it follows that $\mathcal{R}(\Phi)$ is also finite.

Additive utility functions. The utility function U is *additive* if there is either a reward function $R(s)$ or a cost function $C(s,a,s')$ (where (s,a,s') is a transition from s to s' caused by action a) such that U is the sum of the rewards or costs associated with the state transitions in ρ. These state transitions are the points in ρ where UPOM simulates the execution of an action.

For each pair (σ_j, s_j) in ρ, let $(\tau_j, m_j, i_j, \mathit{tried}_j)$ be the top element of σ_j. If $m_j[i_j]$ is an action, then the next element of ρ is a pair (σ_{j+1}, s_{j+1}) in which s_{j+1} is the state produced by executing the action $m_j[i_j]$. In Φ this corresponds to the state transition $(s_j, m_j[i_j], s_{j+1})$. Thus the set of state transitions in ρ is

$$\begin{aligned} t_\rho = \{(s_j, m_j[i_j], s_{j+1}) \mid (\sigma_j, s_j) \text{ and } (\sigma_{j+1}, s_{j+1}) \text{ are members of } \rho, \\ (\tau_j, m_j, i_j, \mathit{tried}_j) = \mathsf{top}(\sigma_j), \text{ and } m_j[i_j] \text{ is an action}\}. \end{aligned} \tag{A.3}$$

Thus if U is additive, then

$$U(\rho) = \begin{cases} \sum_{(s,a,s') \in t_\rho} R(s'), & \text{if } U \text{ is the sum of rewards,} \\ \sum_{(s,a,s') \in t_\rho} C(s,a,s'), & \text{if } U \text{ is the sum of costs.} \end{cases} \tag{A.4}$$

A.2. Defining the MDP

We want to define an MDP Ψ such that choosing among methods in Φ corresponds to choosing among actions in Ψ. The easiest way to do this is to let all of Φ's actions and methods be actions in Ψ. Based loosely on the notation in [77], we will write Ψ as

$$\Psi = (S^\Psi, \mathcal{A}^\Psi, s_0^\Psi, S_g^\Psi, \gamma^\Psi, P^\Psi, U^\Psi) \tag{A.5}$$

where

$$\begin{aligned} S^\Psi &= \text{stacks}(\Phi) \times S \text{ is the set of states,} \\ \mathcal{A}^\Psi &= \mathcal{M} \cup \mathcal{A} \text{ is the set of actions,} \\ s_0^\Psi &= (\sigma_0, s_0) \text{ is the initial state,} \\ S_g^\Psi &= \{(\langle\rangle, s) \mid s \in S\} \text{ is the set of goal states,} \end{aligned}$$

and the state-transition function γ^Ψ, state-transition probability function P^Ψ, and utility function U^Ψ are defined as follows.

State transitions. To define γ^Ψ and P^Ψ, we must first define which actions are applicable in each state. Let $(\sigma, s) \in S^\Psi$, and $(\tau, m, i, t) = \mathsf{top}(\sigma)$. Then the set of actions that are applicable to (σ, s) in Ψ is

$$\text{Applicable}^\Psi((\sigma, s)) = \begin{cases} \mathsf{Instances}(\mathcal{M}, m[i], s), & \text{if } m[i] \text{ is a task,} \\ \{m[i]\}, & \text{if } m[i] \text{ is an action.} \end{cases} \tag{A.6}$$

Thus if $a \in \text{Applicable}^\Psi((\sigma, s))$, then there are two cases for what $\gamma^\Psi(s, a)$ and $P^\Psi(s, a, s')$ might be:

- Case 1: $m[i]$ is a task in $\mathcal{M}$, and $a \in \mathsf{Instances}(\mathcal{M}, m[i], s)$. In this case, the next refinement stack will be produced by pushing a new stack frame $\phi = (m[i], a, 1, \varnothing)$ onto σ. The state s will remain unchanged. Thus the next state in Ψ will be $(\phi + \sigma, s)$, where '+' denotes concatenation. Thus

$$\begin{aligned} \gamma((\sigma, s), a) &= \{(\phi + \sigma, s)\}; \\ P^\Psi[(\sigma, s), a, (\phi + \sigma, s)] &= 1; \\ P^\Psi[(\sigma, s), a, (\sigma', s')] &= 0, \quad \text{if } (\sigma', s') \neq (\phi + \sigma, s). \end{aligned}$$

- Case 2: $m[i]$ is an action in $\mathcal{A}$, and $a = m[i]$. Then a's possible outcomes in Ψ correspond one-to-one to its possible outcomes in Φ. More specifically, if γ is the state-transition function for Φ (see Section 3), then

$$\gamma^\Psi((\sigma, s), a) = \{(\mathsf{Next}(\sigma, s'), s') \mid s' \in \gamma(s, a)\}$$

and

$$P^\Psi((\sigma, s), a, (\sigma', s'))) = \begin{cases} P(s, a, s'), & \text{if } (\sigma', s') \in \gamma^\Psi((\sigma, s), a), \\ 0, & \text{otherwise.} \end{cases}$$

Rollouts and utility. A *rollout* of Π^Ψ is any sequence of states and actions of Ψ,

$$\rho^\Psi = \langle (\sigma_0, s_0), a_1, (\sigma_1, s_1), a_2, \ldots, (\sigma_{n-1}, s_{n-1}), a_n, (\sigma_n, s_n) \rangle,$$

such that for $i = 1, \ldots, n$, $a_i \in \text{Applicable}(\sigma_{i-1}, s_{i-1})$ and

$$P^\Psi((\sigma_{i-1}, s_{i-1}, (\sigma_i, s_i)), a_i) > 0.$$

The rollout is *successful* if $(\sigma_n, s_n) \in S_g^\Phi$, and unsuccessful otherwise.

We can define U^Ψ directly from U. If ρ^Ψ is the rollout given above, then the corresponding rollout in Φ is $\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_{n-1}, s_{n-1}), (\sigma_n, s_n) \rangle$, and

$$U^\Psi(\rho^\Psi) = U(\rho).$$

If U is additive, then so is U^Ψ. In this case, Ψ satisfies the definition of an MDP with initial state (see [77]).

A.3. Mapping UPOM's Search to an Equivalent UCT Search

Let

$$\Pi = (\Phi, s_0, \sigma_0, U) \tag{A.7}$$

be a refinement planning problem, where

$$\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A}). \tag{A.8}$$

Suppose $\mathsf{UPOM}(s_0, \sigma_0, \infty)$ generates the rollout

$$\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n) \rangle, \tag{A.9}$$

where $\sigma_j = (\tau_j, m_j, i_j, \mathit{tried}_j)$, for $j = 1, \ldots, n$. UPOM generates ρ by choosing m_1 and then recursively calling $\mathsf{UPOM}(s_j, \sigma_j, \infty)$. Consequently, UPOM's probability of generating ρ is

$$p = p_1 \times \ldots \times p_n, \tag{A.10}$$

where each p_j is the probability that $\mathsf{UPOM}(s_j, \sigma_j, \infty)$ will choose m_j before making its recursive call. The value of p_j will depend on UPOM's *metadata* for Π, e.g., the number of times each method for a task τ has been tried in each state s, and the average utility obtained over those tries.

We want to show that UPOM's search of Π corresponds to an equivalent UCT search of Ψ. Below, Theorem 1 accomplishes this in the case where the utility function U is additive. After the proof of the theorem, we discuss how to generalize the theorem to cases where U is not additive.

Theorem 1. *Let Π, Φ, ρ and p be as in Equations A.7–A.10, let U be additive, let UPOM's metadata for Π be as described above, and let $\Psi = (S^\Psi, \mathcal{A}^\Psi, \gamma^\Psi, P^\Psi, U^\Psi)$ be the MDP corresponding to Π. If UCT searches Ψ using the same metadata that UPOM used, then the probability that UCT generates the rollout*

$$\rho^\Psi = \langle (\sigma_0, s_0), m_1, (\sigma_1, s_1), m_2, \ldots, (\sigma_{n-1}, s_{n-1}), m_n, (\sigma_n, s_n) \rangle$$

is the same probability $p = p_1 \times \ldots \times p_n$ as in Equation A.10.

Sketch of proof. The proof is by induction on n, the length of ρ. The base case is when $n = 0$, i.e., $\rho = \langle(\sigma_0, s_0)\rangle$. If $n = 0$ then it must be that $\mathsf{Applicable}(s_0) = \varnothing$. Thus $\mathsf{Applicable}^{\Psi}((\sigma_0, s_0)) = \varnothing$, so in this case the theorem is vacuously true.

For the induction step, suppose $n > 0$, and consider UPOM's recursive call to $\mathsf{UPOM}(s_1, \sigma_1, \infty)$. In this case, the refinement planning problem is $\Pi_1 = (\Phi, s_1, \sigma_1, U)$, and we let Ψ_1 be the corresponding MDP.

Given the same metadata as above, $\mathsf{UPOM}(s_1, \sigma_1, \infty)$ will generate the rollout $\rho_1 = \langle(\sigma_1, s_1), \ldots, (\sigma_n, s_n)\rangle$ with probability $p_2 \times \ldots \times p_n$. The induction assumption is that with that same probability, a UCT search of Ψ_1 will generate the rollout

$$\rho_1^{\Psi} = \langle(\sigma_1, s_1), m_2, \ldots, (\sigma_{n-1}, s_{n-1}), m_n, (\sigma_n, s_n)\rangle.$$

Before applying the induction assumption, we first need to show that if p_1 is the probability that $\mathsf{UPOM}(s_0, \sigma_0, U)$ chooses m_1 before making its recursive call, then a UCT search of Ψ_1 will choose m_1 with the same probability p_1. There are two cases:

- Case 1: m_1 is a method in Φ. As shown in Algorithm 6, $\mathsf{UPOM}(s_0, \sigma_0, U)$ chooses m_1 using the same UCB-style computation that a UCT search in Ψ would use at (σ_0, s_0). Thus, omitting the details about how to compute p_1 from the metadata, it follows that if $\mathsf{UPOM}(s_0, \sigma_0, U)$ chooses m_1 with probability p_1, then so does the UCT search.
- Case 2: m_1 is an action in Φ. Then UPOM's computation (in lines line 8 through the end of Algorithm 6) is *not* a UCT-style computation, but this does not matter, because there is only one possible choice, namely m_1. In this case, UPOM's probability of choosing m_1 is $p_1 = 1$, and the same is true for the UCT search.

In both cases, it follows from the induction assumption that in Π, UPOM's probability of generating ρ is $p_1 \times p_2 \times \ldots \times p_n$, and in Π^{Ψ}, UCT's probability of generating ρ^{Ψ} is also $p_1 \times p_2 \times \ldots \times p_n$. This concludes the sketch of the proof. □

Generalizing beyond MDPs. If the utility function U is not additive, Equation A.5 produces a probabilistic planning problem that looks similar to an MDP, the only difference being that the utility function U^{Ψ} is not additive. Furthermore, Theorem 1 still holds even when U is not additive, if we modify the proof to remove the claim that Ψ is an MDP.

We note that the UCT algorithm [66] is not restricted to the case where U^{Ψ} is additive; it will still converge as long as U^{Ψ} is monotonic. If U is monotonic, then so is U^{Ψ}. In this case it follows that UCT—and thus UPOM—will converge to an optimal solution. In particular, UPOM will converge to an optimal solution when using the *efficiency* and *success ratio* utility functions described in Section 5.1.

Appendix B. Operational model for the S&R domain

```
declare_commands([
    moveEuclidean, # UGV moves from a location to another following
     a Euclidean path
    moveCurved, # UGV moves from a location to another following a
    curved path
    moveManhattan, # UGV moves from a location to another following
     a Manhattan path
    fly, # UAV flies from one location to another
    giveSupportToPerson, # UGV helps one person
```

```
    clearLocation, # UGV removes debri from a location
    inspectLocation, # UAV surveys a location, searching for
    injured people
    inspectPerson, # UGV checks whether a person is injured or not
    transfer, # one UGV transfers medical supplies to another
    replenishSupplies, # UGV replenishes medical supplies at the
    base
    captureImage, # UAV captures an image using on of its cameras
    changeAltitude, # UAV changes its flying altitude
    deadEnd, # the agent reaches a dead end from which it can't
    recover from
    fail # the command always fails
    ])

declare_task('moveTo', 'r', 'l') # robot r moves to location l
declare_task('rescue', 'r', 'p') # robot r rescues person p
declare_task('helpPerson', 'r', 'p') # robot r helps a person p
declare_task('getSupplies', 'r') # UGV r refills its medical
    supplies from the base
declare_task('survey', 'r', 'l') # UAV r surveys the location l
declare_task('getRobot') # assigns a free robot to help a person
declare_task('adjustAltitude', 'r') # UAV r adjusts its flying
    altitude depending on the weather conditions



declare_methods('moveTo',  # Four possible methods for the \
    MoveTo_Method4, # task moveTo(r, l)
    MoveTo_Method3,
    MoveTo_Method2,
    MoveTo_Method1,
    )

declare_methods('rescue', # Two methods for the task rescue(r, p)
    Rescue_Method1,
    Rescue_Method2,
    )

declare_methods('helpPerson', # Two methods for the \
    HelpPerson_Method2, # task helpPerson(r, p)
    HelpPerson_Method1,
    )

declare_methods('getSupplies', # Two methods for the \
    GetSupplies_Method2, #  task getSupplies(r)
    GetSupplies_Method1,
    )

declare_methods('survey', # Two methods for the \
    Survey_Method1, # task survey(r, l)
    Survey_Method2
    )

declare_methods('getRobot', # Two methods for the \
    GetRobot_Method1, # task getRobot
    GetRobot_Method2,
    )
```

```
declare_methods('adjustAltitude', # Two methods for the \
    AdjustAltitude_Method1,  # task adjustAltitude(r)
    AdjustAltitude_Method2,
    )

# Full descriptions of the commands

def moveEuclidean(r, l1, l2, dist):
  ''' UGV r moves from a location l1 to location l2  following a
    Euclidean path. '''
    (x1, y1) = l1
    (x2, y2) = l2
    xlow = min(x1, x2)
    xhigh = max(x1, x2)
    ylow = min(y1, y2)
    yhigh = max(y1, y2)
    # r checks whether there are any obstacles in the path
    for o in rv.OBSTACLES:
        (ox, oy) = o
        if ox >= xlow and ox <= xhigh and oy >= ylow and oy <=
    yhigh:
            if ox == x1 or x2 == x1:
                Simulate("%s cannot move in Euclidean path because
    of obstacle\n" %r)
                return FAILURE
            elif abs((oy - y1)/(ox - x1) - (y2 - y1)/(x2 - x1)) <=
    0.0001:
                Simulate("%s cannot move in Euclidean path because
    of obstacle\n" %r)
                return FAILURE

    state.loc.AcquireLock(r)
    if l1 == l2:
        Simulate("Robot %s is already at location %s\n" %(r, l2))
        res = SUCCESS
    elif state.loc[r] == l1:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('moveEuclidean',
    start, r, l1, l2, dist) == False):
           pass
        res = Sense('moveEuclidean')
        if res == SUCCESS:
            Simulate("Robot %s has moved from %s to %s\n" %(r, str(
    l1), str(l2)))
            state.loc[r] = l2
        else:
            Simulate("Robot %s failed to move due to some internal
    failure.\n" %r)
    else:
        Simulate("Robot %s is not in location %d.\n" %(r, l1))
        res = FAILURE
    state.loc.ReleaseLock(r)
    return res

def moveCurved(r, l1, l2, dist):
  ''' UGV r moves from a location l1 to l2 following a curved path
```

```
    '''
    (x1, y1) = l1
    (x2, y2) = l2
    centrex = (x1 + x2)/2
    centrey = (y1 + y2)/2
     # r checks whether there are any obstacles in the path
    for o in rv.OBSTACLES:
        (ox, oy) = o
        r2 = (x2 - centrex)*(x2 - centrex) + (y2 - centrey)*(y2 -
    centrey)
        ro = (ox - centrex)*(ox - centrex) + (oy - centrey)*(oy -
    centrey)
        if abs(r2 - ro) <= 0.0001:
            Simulate("%s cannot move in curved path because of
    obstacle\n" %r)
            return FAILURE

    state.loc.AcquireLock(r)
    if l1 == l2:
        Simulate("Robot %s is already at location %s\n" %(r, l2))
        res = SUCCESS
    elif state.loc[r] == l1:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('moveCurved',
    start, r, l1, l2, dist) == False):
            pass
        res = Sense('moveCurved')
        if res == SUCCESS:
            Simulate("Robot %s has moved from %s to %s\n" %(r, str(
    l1), str(l2)))
            state.loc[r] = l2
        else:
            Simulate("Robot %s failed to move due to some internal
    failure.\n" %r)
    else:
        Simulate("Robot %s is not in location %d.\n" %(r, l1))
        res = FAILURE
    state.loc.ReleaseLock(r)
    return res

def moveManhattan(r, l1, l2, dist):
  ''' UGV r moves from a location l1 to l2 following a Manhattan
    path '''
    (x1, y1) = l1
    (x2, y2) = l2
    xlow = min(x1, x2)
    xhigh = max(x1, x2)
    ylow = min(y1, y2)
    yhigh = max(y1, y2)
    # r checks whether there are any obstacles in the path
    for o in rv.OBSTACLES:
        (ox, oy) = o
        if abs(oy - y1) <= 0.0001 and ox >= xlow and ox <= xhigh:
            Simulate("%s cannot move in Manhattan path because of
    obstacle\n" %r)
            return FAILURE
```

```
            if abs(ox - x2) <= 0.0001 and oy >= ylow and oy <= yhigh:
                Simulate("%s cannot move in Manhattan path because of
        obstacle\n" %r)
                return FAILURE

        state.loc.AcquireLock(r)
        if l1 == l2:
            Simulate("Robot %s is already at location %s\n" %(r, l2))
            res = SUCCESS
        elif state.loc[r] == l1:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('moveManhattan',
        start, r, l1, l2, dist) == False):
                pass
            res = Sense('moveManhattan')
            if res == SUCCESS:
                Simulate("Robot %s has moved from %s to %s\n" %(r, str(
        l1), str(l2)))
                state.loc[r] = l2
            else:
                Simulate("Robot %s failed to move due to some internal
        failure.\n" %r)
        else:
            Simulate("Robot %s is not in location %d.\n" %(r, l1))
            res = FAILURE
        state.loc.ReleaseLock(r)
        return res

    def fly(r, l1, l2):
      ''' UAV r flies from one location l1 to another location l2'''
        state.loc.AcquireLock(r)
        if l1 == l2:
            Simulate("Robot %s is already at location %s\n" %(r, l2))
            res = SUCCESS
        elif state.loc[r] == l1:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('fly', start) ==
        False):
                pass
            res = Sense('fly')
            if res == SUCCESS:
                Simulate("Robot %s has flied from %s to %s\n" %(r, str(
        l1), str(l2)))
                state.loc[r] = l2
            else:
                Simulate("Robot %s failed to fly due to some internal
        failure.\n" %r)
        else:
            Simulate("Robot %s is not in location %d.\n" %(r, l1))
            res = FAILURE
        state.loc.ReleaseLock(r)
        return res

    def inspectPerson(r, p):
      ''' UGV r helps one person p'''
        Simulate("Robot %s is inspecting person %s \n" %(r, p))
        state.status[p] = env.realStatus[p]
```

```
    return SUCCESS

def giveSupportToPerson(r, p):
    if state.status[p] != 'dead':
        Simulate("Robot %s has saved person %s \n" %(r, p))
        state.status[p] = 'OK'
        env.realStatus[p] = 'OK'
        res = SUCCESS
    else:
        Simulate("Person %s is already dead \n" %(p))
        res = FAILURE
    return res

def inspectLocation(r, l):
  ''' UAV r surveys a location, searching for injured people'''
    Simulate("Robot %s is inspecting location %s \n" %(r, str(l)))
    state.status[l] = env.realStatus[l]
    return SUCCESS

def clearLocation(r, l):
  ''' UGV r removes debri from a location l'''
    Simulate("Robot %s has cleared location %s \n" %(r, str(l)))
    state.status[l] = 'clear'
    env.realStatus[l] = 'clear'
    return SUCCESS

def replenishSupplies(r):
  ''' UGV r replenishes medical supplies at the base'''
    state.hasMedicine.AcquireLock(r)
    if state.loc[r] == (1,1):
        state.hasMedicine[r] = 5
        Simulate("Robot %s has replenished supplies at the base.\n"
     %r)
        res = SUCCESS
    else:
        Simulate("Robot %s is not at the base.\n" %r)
        res = FAILURE

    state.hasMedicine.ReleaseLock(r)
    return res

def transfer(r1, r2):
  ''' One UGV r1 transfers medical supplies to another UGV r2'''
    state.hasMedicine.AcquireLock(r1)
    state.hasMedicine.AcquireLock(r2)
    if state.loc[r1] == state.loc[r2]:
        if state.hasMedicine[r1] > 0:
            state.hasMedicine[r2] += 1
            state.hasMedicine[r1] -= 1
            Simulate("Robot %s has transferred medicine to %s.\n"
    %(r1, r2))
            res = SUCCESS
        else:
            Simulate("Robot %s does not have medicines.\n" %r1)
            res = FAILURE
    else:
        Simulate("Robots %s and %s are in different locations.\n"
```

```
    %(r1, r2))
        res = FAILURE
    state.hasMedicine.ReleaseLock(r2)
    state.hasMedicine.ReleaseLock(r1)
    return res

def captureImage(r, camera, l):
  ''' UAV r captures an image using on of its cameras at location l
    '''
    img = Sense('captureImage', r, camera, l)

    state.currentImage.AcquireLock(r)
    state.currentImage[r] = img
    Simulate("UAV %s has captured image in location %s using %s\n"
    %(r, l, camera))
    state.currentImage.ReleaseLock(r)
    return SUCCESS

def changeAltitude(r, newAltitude):
  ''' UAV r changes its flying altitude to newAltitude '''
    state.altitude.AcquireLock(r)
    if state.altitude[r] != newAltitude:
        res = Sense('changeAltitude')
        if res == SUCCESS:
            state.altitude[r] = newAltitude
            Simulate("UAV %s has changed altitude to %s\n" %(r,
    newAltitude))
        else:
            Simulate("UAV %s was not able to change altitude to %s\
    n" %(r, newAltitude))
    else:
        res = SUCCESS
        Simulate("UAV %s is already in %s altitude.\n" %(r,
    newAltitude))
    state.altitude.ReleaseLock(r)
    return res

def SR_GETDISTANCE_Euclidean(l0, l1):
'''Calculates the euclidean distance between two 2D points, l0 and
    l1'''
    (x0, y0) = l0
    (x1, y1) = l1
    return math.sqrt((x1 - x0)*(x1 - x0) + (y1 - y0)*(y1-y0))

def MoveTo_Method1(r, l):
  # A wheeled UGV robot r takes the straight path to reach l1
    x = state.loc[r]
    if x == l:
        Simulate("Robot %s is already in location %s\n." %(r, l))
    elif state.robotType[r] == 'wheeled':
        dist = SR_GETDISTANCE_Euclidean(x, l)
        Simulate("Euclidean distance = %d " %dist)
        do_command(moveEuclidean, r, x, l, dist)
    else:
        do_command(fail)

def SR_GETDISTANCE_Manhattan(l0, l1):
```

```
 ''' Calculates the Manhattan distance between two 2D points, l0
    and l1. '''
    (x1, y1) = l0
    (x2, y2) = l1
    return abs(x2 - x1) + abs(y2 - y1)

def MoveTo_Method2(r, l):
  ''' UGV r  takes a Manhattan path to location l '''
    x = state.loc[r]
    if x == l:
        Simulate("Robot %s is already in location %s\n." %(r, l))
    elif state.robotType[r] == 'wheeled':
        dist = SR_GETDISTANCE_Manhattan(x, l)
        Simulate("Manhattan distance = %d " %dist)
        do_command(moveManhattan, r, x, l, dist)
    else:
        do_command(fail)

def SR_GETDISTANCE_Curved(l0, l1):
 ''' Calculates the curved distance between two 2D points, l0 and
    l1. '''
    diameter = SR_GETDISTANCE_Euclidean(l0, l1)
    return math.pi * diameter / 2

def MoveTo_Method3(r, l):
  # UGV r  takes a curved path to reach location l
    x = state.loc[r]
    if x == l:
        Simulate("Robot %s is already in location %s\n." %(r, l))
    elif state.robotType[r] == 'wheeled':
        dist = SR_GETDISTANCE_Curved(x, l)
        Simulate("Curved distance = %d " %dist)
        do_command(moveCurved, r, x, l, dist)
    else:
        do_command(fail)

def MoveTo_Method4(r, l):
  ''' UAV r moves to location l '''
    x = state.loc[r]
    if x == l:
        Simulate("Robot %s is already in location %s\n." %(r, l))
    elif state.robotType[r] == 'uav':
        do_command(fly, r, x, l)
    else:
        do_command(fail)

def Rescue_Method1(r, p):
  ''' A UGV r helps a person after procuring medical supplies. '''
    if state.robotType[r] != 'uav':
        if state.hasMedicine[r] == 0:
            do_task('getSupplies', r)
        do_task('helpPerson', r, p)
    else:
        do_command(fail)

def Rescue_Method2(r, p):
  ''' A UAV r delegates the rescuing task to a free ground robot.
```

```
    '''
    if state.robotType[r] == 'uav':
        do_task('getRobot')
    r2 = state.newRobot[1]
    if r2 != None:
        if state.hasMedicine[r2] == 0:
            do_task('getSupplies', r2)
        do_task('helpPerson', r2, p)
        state.status[r2] = 'free'
    else:
        Simulate("No robot is free to help person %s\n" %p)
        do_command(fail)

def HelpPerson_Method1(r, p):
    # Robot r helps an injured person p
    do_task('moveTo', r, state.loc[p])
    do_command(inspectPerson, r, p)
    if state.status[p] == 'injured':
        do_command(giveSupportToPerson, r, p)
    else:
        do_command(fail)

def HelpPerson_Method2(r, p):
    # Robot r helps a person p trapped inside some debri but not
    injured
    do_task('moveTo', r, state.loc[p])
    do_command(inspectLocation, r, state.loc[r])
    if state.status[state.loc[r]] == 'hasDebri':
        do_command(clearLocation, r, state.loc[r])
    else:
        CheckResult(state.loc[p])
        do_command(fail)

def GetSupplies_Method1(r):
    # UGV r gets medical supplies from nearby robots
    r2 = None
    nearestDist = float("inf")
    for r1 in rv.WHEELEDROBOTS:
        if state.hasMedicine[r1] > 0:
            dist = SR_GETDISTANCE_Euclidean(state.loc[r], state.loc
    [r1])
            if dist < nearestDist:
                nearestDist = dist
                r2 = r1
    if r2 != None:
        do_task('moveTo', r, state.loc[r2])
        do_command(transfer, r2, r)

    else:
        do_command(fail)

def GetSupplies_Method2(r):
    # UGV r gets medical supplies from the base
    do_task('moveTo', r, (1,1))
    do_command(replenishSupplies, r)

def CheckResult(l):
```

```
    ''' Function to check whether a person is saved or not after
      performing the rescue operations.'''
      p = env.realPerson[l]
      if p != None:
          if env.realStatus[p] == 'injured' or env.realStatus[p] == '
      dead' or env.realStatus[l] == 'hasDebri':
              Simulate("Person in location %s failed to be saved.\n"
      %str(l))
              do_command(deadEnd, p)
              do_command(fail)

def Survey_Method1(r, l):
    ''' UAV r surveys location l with the help of the front camera.
      '''
      if state.robotType[r] != 'uav':
          do_command(fail)

      do_task('adjustAltitude', r)

      do_command(captureImage, r, 'frontCamera', l)

      img = state.currentImage[r]
      position = img['loc']
      person = img['person']

      if person != None:
          do_task('rescue', r, person)

      CheckResult(l)

def Survey_Method2(r, l):
    ''' UAV r surveys location l with the help of the bottom camera.
      '''
      if state.robotType[r] != 'uav':
          do_command(fail)

      do_task('adjustAltitude', r)
      do_command(captureImage, r, 'bottomCamera', l)

      img = state.currentImage[r]
      position = img['loc']
      person = img['person']
      if person != None:
          do_task('rescue', r, person)

      CheckResultl(l)

def GetRobot_Method1():
    ''' Finds a free robot to do some rescue task '''
      dist = float("inf")
      robot = None
      for r in rv.WHEELEDROBOTS:
          if state.status[r] == 'free':
              if SR_GETDISTANCE_Euclidean(state.loc[r], (1,1)) < dist
      :
                  robot = r
                  dist = SR_GETDISTANCE_Euclidean(state.loc[r], (1,1)
```

```
    )
    if robot == None:
        do_command(fail)
    else:
        state.status[robot] = 'busy'
        state.newRobot[1] = robot

def GetRobot_Method2():
  ''' Assigns the first robot from the list of UGVs to do a resue
    task '''
    state.newRobot[1] = rv.WHEELEDROBOTS[0]
    state.status[rv.WHEELEDROBOTS[0]] = 'busy'

def AdjustAltitude_Method1(r):
  ''' Changes the altitude of an UAV r from high to low.'''
    if state.altitude[r] == 'high':
        do_command(changeAltitude, r, 'low')

def AdjustAltitude_Method2(r):
  ''' Changes the altitude of an UAV r from low to high.'''
    if state.altitude[r] == 'low':
        do_command(changeAltitude, r, 'high')

# ONE PROBLEM INSTANCE INCLUDES THE FOLLOWING
# Rigid variables
rv.WHEELEDROBOTS = ['w1', 'w2']
rv.DRONES = ['a1', 'a2']
rv.OBSTACLES = { (100, 100)}

# initial values state variables
  state.loc = {'w1': (15,15), 'w2': (29,29), 'p1': (28,30), 'p2':
    (10,30), 'a1': (9,19), 'a2': (4,5)}
  state.hasMedicine = {'a1': 0, 'a2': 0, 'w1': 0, 'w2': 0}
  state.robotType = {'w1': 'wheeled', 'a1': 'uav', 'a2': 'uav', 'w2
    ': 'wheeled'}
  state.status = {'w1': 'free', 'w2': 'free', 'a1': UNK, 'a2': UNK,
     'p1': UNK, 'p2': UNK, (28,30): UNK, (15, 15): UNK, (10, 30):
    UNK}
  state.altitude = {'a1': 'high', 'a2': 'low'}
  state.currentImage = {'a1': None, 'a2': None}
  state.newRobot = {1: None}

  # properties of the environment
  env.realStatus = {'w1': 'OK', 'p1': 'OK', 'p2': 'injured', 'w2':
    'OK', 'a1': 'OK', 'a2': 'OK', (28, 30): 'hasDebri', (15, 15): '
    clear', (10, 30): 'hasDebri'}
  env.realPerson = {(28,30): 'p1', (15, 15): None, (10, 30): 'p2'}
  env.weather = {(28,30): "foggy", (15, 15): "rainy", (10, 30): "
    dustStorm"}

# tasks to accomplish
tasks = {
    8: [['survey', 'a1', (15,15)], ['survey', 'a2', (28,30)]],
    20: [['survey', 'a1', (10,30)]]
}
```

Appendix C. Table of Notation

Notation	Meaning	Page defined
$\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$	an acting domain	12
s, S	predicted state, set of states for the planner	9
X	set of state variables	9
ξ, Ξ	actual state, set of world states for the actor	9
τ, $\mathcal{T}$	task or event, set of tasks and events	12
m, $\mathcal{M}$	method/method instance, set of methods for $\mathcal{T}$	12
$\overline{\mathcal{M}}$	set of method instances of $\mathcal{M}$	12
$m[i]$	the ith step of m	14
$\mathsf{Applicable}(\xi, \tau)$	set of method instances applicable to τ in state ξ	12
a, $\mathcal{A}$	action, set of actions	12
$\gamma(\xi, a)$	possible states after performing a in ξ	12
σ	refinement stack with tuples of the form $(\tau, m, i, tried)$	13
v_e, v_s	value functions for efficiency and success ratio	17
$v_{e1} \oplus v_{e2}$	cumulative efficiency value of two successive actions	17
$v_{s1} \oplus v_{s2}$	cumulative success ratio of two successive actions	18
$\mathbb{I}$	the identify element for $\oplus$, i.e. $x \oplus \mathbb{I} = x$	18
$U(m, s, \sigma)$	the utility of m for τ and σ	18
$U^*(m, s, \sigma)$	the maximal expected utility of m for τ	19
U_{Success}, U_{Failure}	the utility of a success, the utility of a failure	21
$m^*_{\tau,s}$	the optimal method instance for τ in s for utility U^*	19
d, d_{max}, n_{ro}	depth, max depth, number of rollouts	19
$h(\tau, m, s)$	heuristic estimate to solve τ with m in s	20
h_0, h_D	always returns ∞, hand written heuristic	33
h_{LearnH}	learned heuristic	33
$Q_{\sigma,s}(m)$	approximation of $U^*(m, s, \sigma)$	21
C	tradeoff between exploration and exploitation	21
μ, K	suggested control parameters for n_{ro}	23
$g(s, \tau, m)$	default state after accomplishing τ with m in s	24
r	a data record of the form $((s, \tau), m)$	25
w_s, w_τ, w_m, w_u	One-Hot representations of s, τ, m, and $interval(u)$	26
v_{un}	uninstantiated method parameter	27
v_τ	list of values of task parameters	27
b	value of the parameter v_{ui}	27
V	number of state variables	26
nn_π	MLP for $\mathsf{Learn}\pi$	26
$nn_{v_{un}}$	MLP for each v_{un}	27
Z	Number of training records	29
$\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$	a refinement planning domain	16
$P(s, a, s')$	probability that a's execution in s returns s'	54
$\Pi = (\Phi, s_0, \sigma_0, U)$	a refinement planning problem	54
ρ	a rollout in Φ	56
$\mathcal{R}(\Phi)$	the set of all refinement stacks that are reachable in Φ	54
$R(s)$, $C(s, a, s')$	reward function, cost function	54
Ψ	an MDP used in the appendix	55