Interleaving Acting and Planning Using Operational Models

Sunandita Patra[1], **Malik Ghallab**[2], **Dana Nau**[1], **Paolo Traverso**[3]
patras@cs.umd.edu, malik@laas.fr, nau@cs.umd.edu, traverso@fbk.eu
[1]Department of Computer Science and Institute for Systems Research, University of Maryland, College Park, USA
[2]Centre national de la recherche scientifique (CNRS), Toulouse, France
[3]Fondazione Bruno Kessler (FBK), Povo - Trento, Italy

Abstract

In (Patra et al. 2019) we proposed and implemented a framework for planning with *operational models*, i.e., models that describe how to perform actions, with rich control structures for closed-loop online decision-making. As described in (Patra et al. 2019), the acting component RAE, inspired by the well-known PRS system, calls the planner RAEplan, which plans by doing Monte Carlo rollout simulations of the actor's operational models.

In this paper, we show how this framework can be used to interleave acting and planning with operational models in different ways. We extend the acting component RAE with heuristics to decide when and how to call the planning component RAEplan. This allows us to realize more or less reactive behaviors. For instance, the acting component RAE may decide to call the planner just when it fails or anytime a decision needs to be made. Moreover, RAE can decide whether to bound the depth of the search during planning, and whether to do acting and planing concurrently. The planning algorithm in this paper takes into consideration the depth of the search. We call the modified planning algorithm RAEplan-LookAhead. We implement the RAEplan-LookAhead algorithm and do its experimental evaluation on a simulated domain called Search and Rescue.

Introduction

Several approaches for the integration of planning, acting, and execution have been proposed so far, see, e.g., (Vaquero et al. 2018). Some of them (e.g., lookahead methods, see e.g., (Ghallab, Nau, and Traverso 2016) for a survey) are based on the idea of generating a partial plan, for example the next few "good" actions, and then acting, i.e., performing all or some of the generated actions, and repeating these two steps from the state that has been reached. In this way, the planner knows exactly which of the many possible states of the world has actually been reached, and the uncertainty as well as the search space is significantly reduced. Moreover, interleaving planning and acting provides the ability to deal with dynamic environments and exogenous events.

Most of the previous approaches to interleaving planning and execution perform planning with *descriptive models*, which represent actions at a rather abstract level, e.g., with preconditions and effects. This representation is tailored to efficiently compute, given some conditions on state variables (the action preconditions), how the values of the state variables change (the action effects). However, when planning needs to be interleaved with acting, most of the works highly underestimate the problem of mapping descriptive models to and from *operational models*, which describe *how* to perform actions, with rich control structures for closed-loop online decision-making. The mapping between the descriptive model and the operational model is often given for granted, simply assuming that the acting/execution mechanism returns the actual state (the values of state variables) at the abstract level in which the agent can start to do planning again.

In this paper we take a different approach. We build upon our work presented in (Patra et al. 2019) (see also (Patra et al. 2018)), in which we propose to use a single representation, the operational model, for both acting and planning, and to do planning by reasoning directly with the actor's operational models. In our approach, the agent does not start from planning and then calls the execution platform when needed. We rather start the other way around. The acting component RAE, inspired by the well-known PRS system, calls the planner RAEplan, which plans by doing Monte Carlo rollout simulations of the actor's operational models. RAE uses a hierarchical task-oriented operational representation. A collection of refinement *methods* describes alternative ways to handle *tasks* and react to *events*. RAE calls RAEplan to decide how to refine tasks or events. RAEplan does Monte Carlo rollouts with applicable refinement methods.

In this paper, we extend this framework and show how it can be used to interleave acting and planning in different ways. We extend the acting component RAE with heuristics to decide when an how to call the planning component RAEplan. This allows us to realize more or less reactive behaviours. For instance, the acting component RAE can decide to call the planner just when it fails or anytime a decision must be made, i.e., anytime one upon different applicable methods must be selected. Moreover, our extended planning algorithm RAEplan-LookAhead, when called by RAE, can decide whether to complete the Monte Carlo rollout, or to bound the depth of the search according to different heuristics, e.g., to save time. Finally, RAE and RAEplan-

LookAhead can be run concurrently and interact in different ways.

We implement different techniques for interleaving acting and planning using heuristics and evaluate them on the Search and Rescue domain. With this experimental evaluation, we show the benefits of interleaving acting and planning with operational models.

The paper is structured as follows. In the next section, we first provide some background on the key concepts described in (Patra et al. 2019) to keep the paper self-contained. Following that, we describe different techniques for the interleaving of acting and planning. We then describe the operational models for the Search and Rescue domain, and provide an experimental evaluation. We finally discuss the related and future work.

Background

In this section, we briefly review the key elements of the approach presented in (Patra et al. 2019), to make the paper self-contained. For the details of the acting component RAE and of the planning component RAEplan, please refer to (Patra et al. 2019).

RAE (Refinement Acting Engine) is from (Ghallab, Nau, and Traverso 2016, Chapter 3). It is is based on a hierarchical task-oriented operational representation with an expressive, general-purpose language offering rich control structures for closed-loop online decision-making. A collection of refinement methods describes alternative ways to handle *tasks* and react to *events*. Each method has a *body* that can be any complex algorithm. In addition to the usual programming constructs, the body may contain *subtasks*, which need to be refined recursively, and sensory-motor *commands*, which query and change the world non-deterministically. Notice that we assume that *methods, tasks, and subtasks are manually programmed*. We believe this assumption is the practical way to build agents that can behave reactively and deal with realistic and complex applications.

RAE implements a reactive system. At each loop, it gets in input a task or event that comes in from an external source, such as the user or the execution platform, and it creates a *refinement stack*, analogous to a computer program's execution stack. An agenda keeps the set of all current refinement stacks.

Task frames and refinement stacks. A *task frame* is a four-tuple $r = (\tau, m, i, \textit{tried})$, where τ is a task, m is the method instance used to refine τ, i is the current instruction in body(m), with $i = \mathsf{nil}$ if we haven't yet started executing body(m), and *tried* is the set of methods that have been already tried and failed for accomplishing τ.

A *refinement stack* is a finite sequence of stack frames $\textit{stack} = \langle \rho_1, \ldots, \rho_n \rangle$. If *stack* is nonempty, then top(*stack*) $= \rho_1$; rest(*stack*) $= \langle \rho_2, \ldots, \rho_n \rangle$; *stack* $=$ top(*stack*).rest(*stack*). To denote pushing ρ onto *stack*, we write $\rho.\textit{stack} = \langle \rho, \rho_1, \rho_2, \ldots, \rho_n \rangle$. Refinement stacks used during planning will have the same semantics, but we will use the notation *stack* instead of *stack* to distinguish it from the acting stack.

When an execution failure occurs with a method instance, then RAE calls a Retry procedure. Retry tries another applicable method instance that it hasn't tried already. Notice that when a Retry is called, the failed method has already been partially executed; it has changed the current state. In many application domains it is important to minimize the total number of retries, since recovery from failure may incur significant, unbudgeted amounts of time and expense. We call *retry ratio* the number of times that RAE had to call the Retry procedure, divided by the total number of tasks to accomplish.

Rather than behaving purely reactively, the agent interleaves acting with planning to decide how to refine tasks or events. Planning is performed by Monte Carlo rollouts with applicable refinement methods. Planning is therefore performed with all the same constructs and operations of the operational model used to act, all but a simulation of commands. Commands are indeed simulated when planning and performed by an execution platform in the real world when acting. During planning, when a refinement method contains a command, the planner takes samples of its possible outcomes, using either a domain-dependent generative simulator, when available, or a probability distribution of its possible outcomes.

RAEplan does a recursive search to optimize a criterion. It chooses a refinement method that has a refinement tree with a minimum expected cost for accomplishing a task (along with the remaining partially accomplished tasks in the current refinement stack). It minimizes the expected cost, i.e., the expected cost of the plan for accomplishing all the tasks in the refinement stack. In order to take into account possible failures, which would have an infinite cost, cost minimization is done by maximizing an *efficiency* criteria, which is the reciprocal of the cost.

Efficiency. We define the *efficiency* of accomplishing a task to be the reciprocal of the cost. Let a decomposition of a task τ have two subtasks, τ_1 and τ_2, with cost c_1 and c_2 respectively. The efficiency of τ_1 is $e_1 = 1/c_1$ and the efficiency of τ_2 is $e_2 = 1/c_2$. The cost of accomplishing both tasks is $c_1 + c_2$, so the efficiency of accomplishing τ is

$$1/(c_1 + c_2) = e_1 e_2/(e_1 + e_2). \tag{1}$$

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

$$\begin{aligned} e_1 \bullet e_2 = e_2 \text{ if } e_1 = \infty, \text{ else} \\ e_1 \text{ if } e_2 = \infty, \text{ else } e_1 e_2/(e_1 + e_2). \end{aligned} \tag{2}$$

If τ_1 (or τ_2) fails, then c_1 is ∞, $e_1 = 0$. Thus $e_1 \bullet e_2 = 0$, meaning that τ fails with this decomposition. Note that formula 2 is associative.

Moreover, RAEplan has two parameters b and k. Parameter b denotes how many different method instances to examine for each task. Parameter k denotes how large a sample size must be for each command. The estimated efficiency $E^*_{b,k}(s, \textit{stack})$ calculated in a given state s for a refinement stack *stack* depends on both b and k. The larger the values of b and k in $E^*_{b,k}$, the more plans RAEplan will examine. In the

planning algorithm proposed in this paper called RAEplan-LookAhead, we add an additional sub-script called d to efficiency in order to bound the depth of Monte Carlo rollouts that RAEplan-LookAhead examine. In RAEplan, d is always infinite. As $d \to \infty$, the behavior of RAEplan-LookAhead becomes similar to RAEplan. Please see the next section for details.

Interleaving Acting and Planning

In (Patra et al. 2019), the acting algorithm RAE always waits for the planner RAEplan to complete its search and return a refinement method for a task τ. The time taken by RAEplan to complete the search increases with the increase in size of the refinement tree for τ. However, because the planning is happening online, this may create a long wait time before the actor takes an action. We can have several strategies to reduce the wait time of the actor and discuss them below.

Strategy 1: Active Planning. This is the simplest case where RAE calls RAEplan every time it needs to refine a task τ. RAE waits for RAEplan to complete its search and refines τ according to what RAEplan suggested. This strategy is implemented in the paper (Patra et al. 2019). The advantage of this approach is that RAEplan returns the best possible suggestion for given values of b and k. The disadvantage is that the actor RAE need to wait until RAEplan returns. It has no control over the wait time.

Strategy 2: Include Heuristics. The idea is similar to a lookahead search. When RAEplan searches for the most efficient method for a task τ, it does several Monte Carlo rollouts. Every such rollout corresponds to a complete refinement tree for τ. Our idea is that instead of looking at complete rollouts like RAEplan, RAEplan-LookAhead only rolls out upto depth d. When the length of the rollout reaches depth d, we estimate the efficiency of the remaining part of the rollout using a heuristic function. The heuristic may be domain dependent or domain independent. We call the modified algorithm RAEplan-LookAhead. The pseudocode is as follows:

RAEplan-LookAhead$(s, \tau, \textit{tried}, \textit{stack}, d)$
  $M \leftarrow \textit{Candidates}(\tau, s) \setminus \textit{tried}$
  if $d = 0$ then return $M[1]$
  else
    $m_{opt} \leftarrow \text{argmax}_{m \in M} E^*_{b,k,d-1}(s, (\tau, m, 0, \textit{tried}).\textit{stack})$
    if $m_{opt} =$ None then return $M[1]$
    else return m_{opt}

Above, s is the current state, *tried* is the set of refinement method instances which has been tried by RAE to accomplish τ and failed. *stack* is the current refinement stack. $\textit{Candidates}(\tau, s)$ is the set of applicable method instances for τ in current state s. d is the maximum search depth. Note that $d = 0$ corresponds to the situation where RAE acts purely reactively and no planning is done. RAEplan-LookAhead optimizes a criterion called expected efficiency which is based on the definition in the previous section with an additional parameter d (in subscript). The expected efficiency is calculated depending on whether the current step is a task or command and also the current depth d. The definition of E^* is recursive and the value of d decreases by one at every recursive call.

Estimated efficiency. We now define $E^*_{b,k,d}(s, \textit{stack})$ as an estimate of expected efficiency of the optimal plan for the tasks in stack *stack* when the current state is s. The parameters b and k denote, respectively, how many different method instances to examine for each task, and how large a sample size to use for each command. d denotes how much further RAEplan-LookAhead is allowed to search.

If *stack* is empty, then $E^*_{b,k,d}(s, \textit{stack}) = \infty$ because there are no tasks to accomplish. Otherwise, let $(\tau, m, i, \textit{tried}) = \text{top}(\textit{stack})$. Then $E^*_{b,k,d}(s, \textit{stack})$ depends on whether i is a command, an assignment statement, or a task and whether the current depth d is greater than 0:

- If i is a command and $d > 0$, then $E^*_{b,k,d}(s, \textit{stack}) =$

$$\tfrac{1}{k} \textstyle\sum_{s' \in S'} \frac{1}{cost(s,i,s')} \bullet E^*_{b,k,d-1}(s', \mathsf{next}(s', \textit{stack})), \quad (3)$$

  where S' is a random sample of k outcomes of command i in state s, with duplicates allowed. $\mathsf{next}(s', \textit{stack})$ is the refinement stack after performing command i taking into account the effect of control statements like if-else or loops. Since S' has the probability distributions of the outcomes of the commands, it converges asymptotically to the expected value of E^*.

- If i is a command and $d = 0$, then

$$E^*_{b,k,0}(s, \textit{stack}) = \frac{1}{\text{Heuristic-Estimate}(s, \textit{stack})} \quad (4)$$

- If i is an assignment statement, then $E^*_{b,k,d}(s, \textit{stack}) = E^*_{b,k,d}(s', \mathsf{next}(s', \textit{stack}))$, where s' is the state produced from s by performing the assignment statement.

- If i is a task and $d > 0$, then $E^*_{b,k,d}(s, \textit{stack})$ recursively optimizes over the candidate method instances for i. That is:

$$E^*_{b,k,d}(s, \textit{stack}) = \max_{m \in M'} E^*_{b,k,d-1}(s, (i, m, \mathsf{nil}, \emptyset).\textit{stack}), \quad (5)$$

  where $M' = \textit{Candidates}(i, s)$ if $|\textit{Candidates}(i, s)| \leq b$, and otherwise M' is the first b method instances in the preference ordering for $\textit{Candidates}(i, s)$.

- If i is a task and $d = 0$, then $E^*_{b,k,0}(s, \textit{stack})$ is a heuristic estimate of accomplishing the remaining stack. That is:

$$E^*_{b,k,0}(s, \textit{stack}) = \frac{1}{\text{Heuristic-Estimate}(s, \textit{stack})}. \quad (6)$$

We have implemented RAEplan-LookAhead in this paper for a simulated domain called Search and Rescue with two domain dependent heuristics and various values of depth d. As d approaches infinity, the behavior of RAEplan-LookAhead should become similar to the RAEplan algorithm in (Patra et al. 2019). The results are presented in the Experimental Evaluation section.

Strategy 3: Lazy Planning. RAE calls RAEplan-LookAhead the first time it receives a task. RAEplan-LookAhead needs to be modified so that it returns the

most expected refinement tree because this strategy requires that we have a plan and not just one suggested refinement method instance. RAE executes the refinement tree via in-order traversal until it encounters a mismatch between the current state and the state expected from the refinement tree, or a command fails. The drawback of this approach is that in an environment with dynamic events, our plan might become too old and lead to dead-ends which we could have avoided if we used Strategy 1. The advantage of this approach is that it saves the time of computing a similar plan again in case their are no dynamic or exogenous events. However, if commands are non-deterministic, the chances of getting a similar plan are low and this is probably not a good approach.

Strategy 4: Concurrent Planning. RAEplan-LookAhead always runs in parallel to RAE and keeps track of the most recent plan for the current task at hand. As explained in the case of lazy Planning, RAEplan-LookAhead needs to be modified so that it returns a refinement tree instead of just a refinement method instance. Whenever RAE needs advice from RAEplan, RAEplan suggest the refinement method from its current refinement tree. One drawback of this strategy is that, the current refinement tree may not have taken into account the most recent dynamic or exogenous events of the environment. However, this should be better than the lazy strategy because the planner is never idle. This method can be useful when the actor has access to multiple cores and can do multi-tasking.

One could use any one of the above strategies or a combination of them depending on the domain and the nature of tasks that need to be accomplished.

Domain: Search and Rescue

Consider that some natural disaster has happened in a 2D area. People are trapped or injured at certain locations in this area which has no particular graph or map. UAVs continuously survey the area and find people who need help. The detection happens by capturing images via the front and bottom cameras that the UAVs are equipped with. The clarity of the image depends upon various weather conditions and the altitude at which the UAV is flying. We assume that a human expert or some computer vision algorithm identifies correctly whether a person needs help or not from the image. Once a person in need of help has been identified, the UAV transfers control to the UGVs operating on the ground. Ground locations are represented via integral coordinates. The UGVs navigate following certain patterns. In order to move from one location to another, UGVs may take a straight route, a curved route or a Manhattan route. There may or may not be obstacles in their path. If it finds an obstacle, it needs to take a different route to reach its destination. UAVs always fly from one location to the other via a straight route. They may fly in two different altitudes. UGVs are useful for transporting first-aid and medicine to doctors and volunteers or the person in need. First-aid and medicines can be picked up from the base camp or taken from other UGVs which have them. Once helper robot and/or human experts have reached the location of the injured person, they may not find the person immediately. They might need to do some sensing and searching, which can involve removing debris or looking around.

Example 1. *Consider a set R of robots performing search and rescue operations in a partially mapped area. The robots have to find persons in some area and leave them a package of supplies (medication, food, water, etc.). This domain is specified with state variables such as* $\mathsf{robotType}(r) \in \{$*UAV, UGV*$\}$, $r \in R$; $\mathsf{hasSupply}(r) \in \{\top, \bot\}$; $\mathsf{loc}(r) \in L$, *for* $L = \{(x, y) |$ *x and y are integers*$\} \cup \{$*BASE*$\}$.

These robots can use commands such as $\textsc{Detect}(r,$ *camera, class) which detects if an object of some class appears in images acquired by camera of* r, $\textsc{TriggerAlarm}(r, l)$, $\textsc{DropSupply}(r, l)$, $\textsc{LoadSupply}(r, l)$, $\textsc{Takeoff}(r, l)$, $\textsc{Land}(r, l)$, $\textsc{MoveTo}(r, l)$, $\textsc{FlyTo}(r, l)$. *They can address tasks such as:* $\mathsf{search}(r,$*area), which makes a UAV* r *survey in sequence the locations in area,* $\mathsf{survey}(r, l)$, $\mathsf{navigate}(r, l)$, $\mathsf{rescue}(r, l)$, $\mathsf{getSupplies}(r)$.

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

m1-survey(r, l)
  task: survey(r, l)
  pre: robotType$(r) =$ UAV
  body: if $\textsc{Detect}$(r,"base-camera","person")=$\top$ then:
    if hasSupply(r) then rescue(r, l)
    else $\textsc{TriggerAlarm}(r, l)$

This methods specifies that in the location l the UAV r detects if a person appears in the images from its base camera. In that case, it proceeds to a rescue task if it has supplies, otherwise 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.

Here are two possible methods for the task $\mathsf{rescue}(r, l)$*:*

m1-rescue(r, l)
  task: rescue(r, l)
  pre: robotType$(r) =$ UAV
  body: if hasSupply(r) then
    if loc$(r) = l$ then $\textsc{DropSupply}(r, l)$
    else do
      navigate(r, l)
      rescue(r, l)
  else do
    navigate$(r, BASE)$
    $\textsc{LoadSupply}(r, BASE)$
    rescue(r, l)

m2-rescue(r, p)
  task: rescue(r, p)
  pre: (robotType$(r) =$ UGV$) \wedge$ hasSupply(r)
  body: if loc$(r) = l$ then $\textsc{DropSupply}(r, l)$
  else do
    navigate(r, l)
    rescue(r, l)

Note that the above methods are recursive. □

Experimental Evaluation

For our experiments, we generated 96 problems for the search and rescue domain randomly. Every problem has one incoming task, 'survey' or 'rescue' which arrives at a randomly chosen time in RAE's input stream. A problem may

have one to four robots (consisting of UAVs and UGVs). The location of a robot consist of its x and y coordinates in a 2D area. x and y are chosen to be random integers from the range [5, 30]. The location from where a person needs to be rescued is also generated randomly in the same way. Because our commands are nondeterministic, every problem with a particular combination of parameters of RAEplan-LookAhead is run 20 times. The experiments are run on a 2.6 GHz Intel Core i5 processor.

RAEplan-LookAhead has three main parameters: b, k and d. We first did experiments by changing b and k with d set to ∞. We measured the performance using three different metrics: efficiency, success ratio and retry ratio. It is not easy to measure the performance of an integrated planning and acting system. These three metrics were developed after much thought and details can be found in (Patra et al. 2019). Efficiency is the same discussed in the previous section. Success ratio is the number of successful jobs divided by the total number of incoming jobs. A *job* is a task that arrives in the input stream of RAE and does not include the subtasks generated from it. Retry ratio is the number of times RAE retries a task before succeeding (details can be found in the Section *Background*). The results for efficiency, success ratio and retry ratio are shown in Figures 1, 2 and 3 respectively.

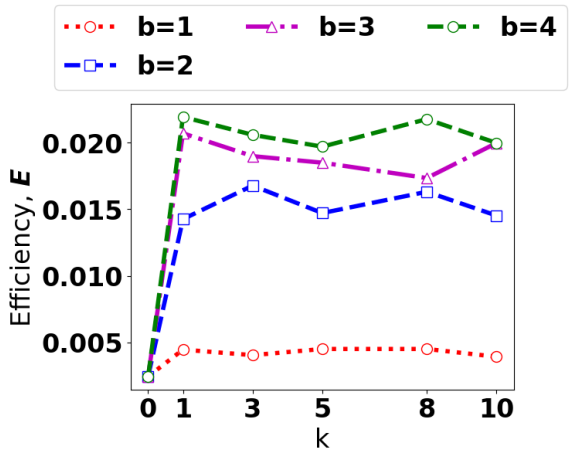

Figure 1: Efficiency E for various values of b and k in the Search and Rescue Domain. d is set to infinity.

From the plots of efficiency, success ratio and retry ratio, we observe that $k = 3$ is the most optimal value of k. Now, we fix k to the value 3 and do experiments by varying the parameter depth d of RAEplan-LookAhead. We do this with the following two heuristics:

1. **Zero Heuristic**: Heuristic is always 0.
2. **Distance Heuristic**: Heuristic is the distance of the agent trusted with the rescue operation and the location where the rescue operation needs to be performed.

We choose the value of depth d to be all values from the set $\{0, 3, 6, 9, 12, 15\}$. We also choose b to be all values from $\{1, 2, 3, 4\}$ because there can be a maximum of four method

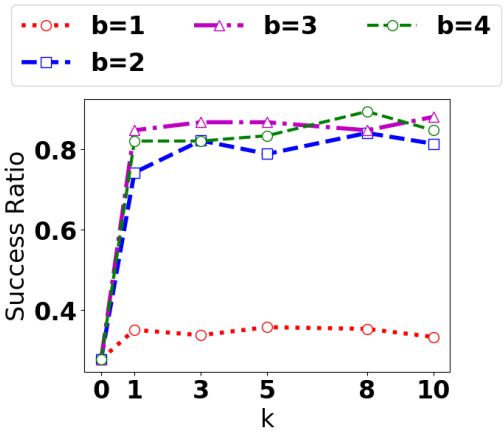

Figure 2: Success ratio (number of successful jobs/ total number of jobs) for various values of b and k in the Search and Rescue Domain.

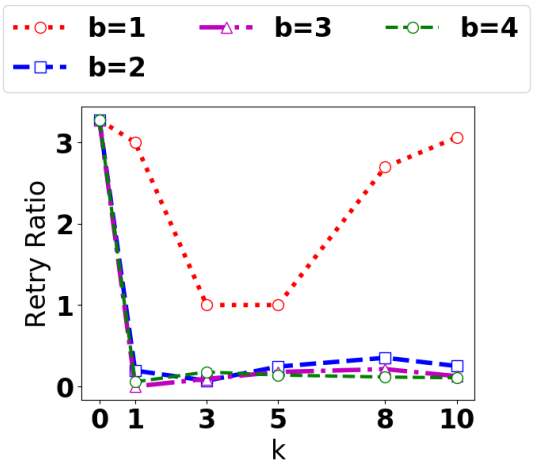

Figure 3: Retry Ratio (number of retries of RAE/ total number of jobs) for various values of b and k in the Search and Rescue Domain.

instances for any task in this domain. We set k to 3 as explained before.

Experiments with Zero Heuristic

We expect to see an improvement in efficiency with increase in depth d because the plans will be more accurate if we examine rollouts till a higher depth before using a heuristic estimate. It is indeed the case as observed in Figure 4. The success ratio also increases for the same reason as observed in Figure 5.

It is interesting to see that the retry ratio decreases upto depth $d = 6$ but then starts increasing with increase in d. In general, we would expect the retry ratio to decrease with increase in d because RAE should be able to accomplish tasks with fewer attempts when plans are more accurate. The increase in efficiency and success ratio confirms that. However, note that retry ratio is measured and compared only for

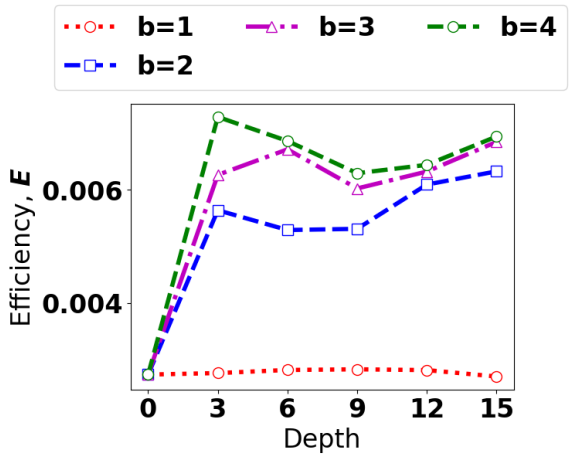

Figure 4: Efficiency E for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

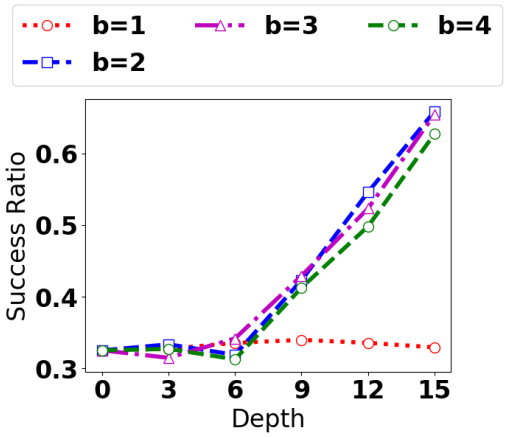

Figure 5: Success ratio (number of successful jobs/ total number of jobs) for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

the successful jobs, jobs that succeed for all values of b and d because it will be unfair to compare the retries of a failed job to the retries of a successful job. For this set of sub-problems in the Search and Rescue domain, it is possible that for some sub-task, RAEplan-LookAhead finds a method which is more efficient but not very reliable to succeed. The failure is not very dangerous because the success ratio does not suffer as seen in Figure 5.

Experiments with Distance Heuristic

Like in the case of Zero heuristic, the experiments with Distance heuristic also show that the efficiency increases with increase in b and depth d. This can be seen in Figure 7. The success ratio also increases with increase in b and d as seen in Figure 8. The behavior of retry ratio shown in Figure 9 is similar to that of zero heuristic. It decreases upto $d = 6$ and then increases. We believe the reason for this behavior is same as discussed in the case of zero heuristic.

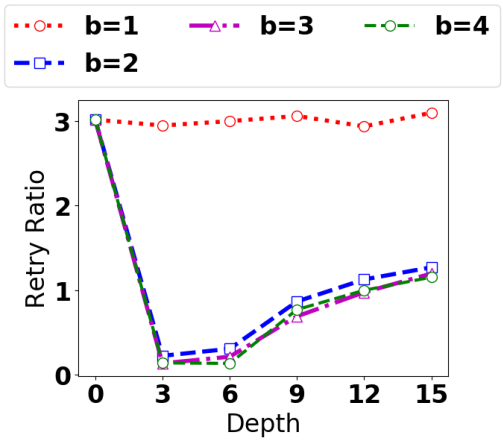

Figure 6: Retry ratio for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

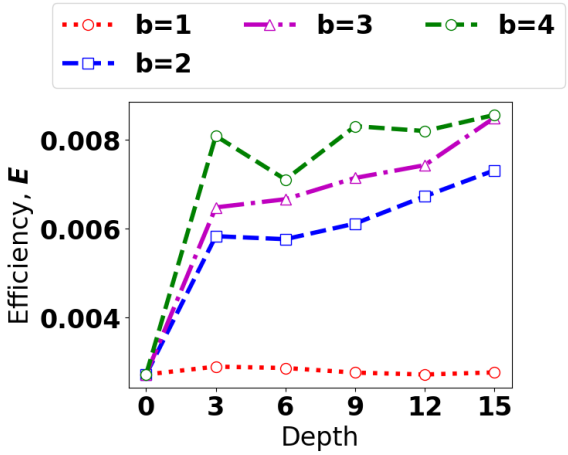

Figure 7: Efficiency E for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

Depth and Running Time

Figure 10 shows how the acting time and the planning time changes with depth for the zero heuristic. We measured the acting time and planning time separately with the bolder lines denoting the acting time. We observe that acting time decreases and planning time increases with increase in depth d of RAEplan-LookAhead. This is expected because the planner needs more time to roll out upto greater depths and returns more efficient methods which in turn reduces the acting time. The changes with depth are more pronounced for $b = 4$ than $b = 1$ because RAEplan-LookAhead examines for method instances for higher values of b. We also observe that somewhere between $d = 9$ and $d = 12$, the planning time starts to dominate the acting time. This is interesting and may help one decide the ideal value of depth in their domain. The acting time and planning time for the distance heuristics is shown in Figure 11. The value of d may be chosen depending on the desired trade-off between efficiency and running time of RAEplan with respect to RAE. Note that

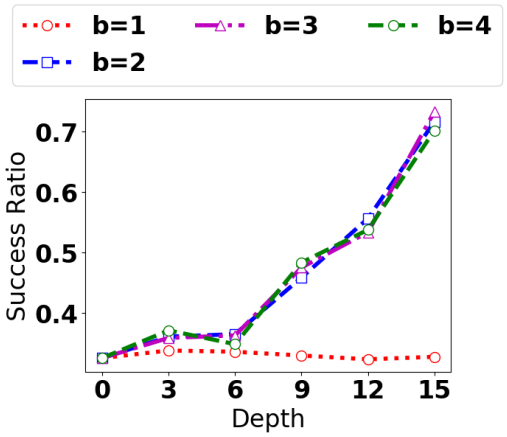

Figure 8: Success ratio (number of successful jobs/ total number of jobs) for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

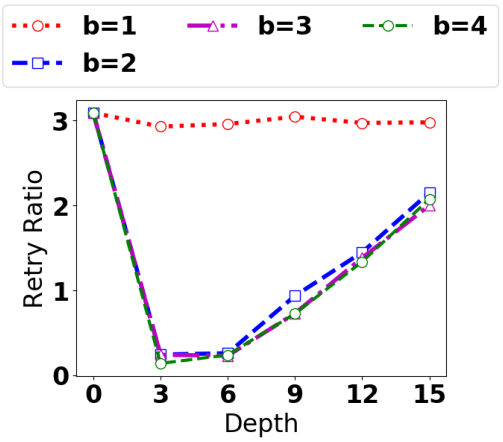

Figure 9: Retry ratio for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

we also tried to observe patterns in the total time by taking an weighted sum of the acting and planning times. However, the results were not very meaningful in this domain.

Related Work

The approach to do planning in an operational model, RAE, and RAEplan, have been presented in (Patra et al. 2019). In this paper we discussed how the framework proposed in (Patra et al. 2019) can be used to easily interleave acting and planning and provided a novel experimental evaluation that shows the advantage of interleaving acting with planning. Beyond our AAAI work, to our knowledge, no previous approach has proposed the integration of planning and acting directly within the language of an operational model.

Our acting algorithm and operational models are based on the RAE algorithm (Ghallab, Nau, and Traverso 2016, Chapter 3), which in turn is based on PRS. If RAE and PRS need to choose among several eligible refinement methods for a given task or event, they make the choice without trying

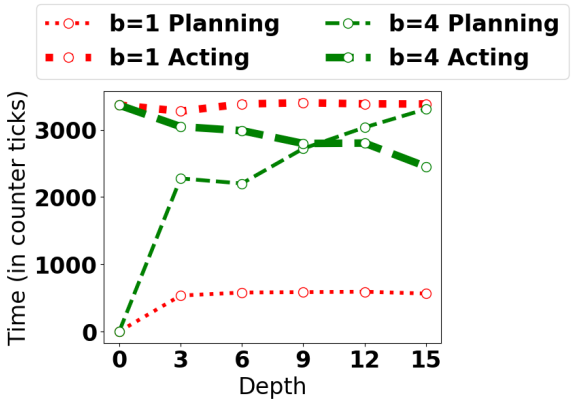

Figure 10: This figure shows that using the zero heuristic, the planning time (running time of RAEplan) increases with depth and the acting time (running time of RAE) decreases when it calls RAEplan with higher depth d. The time is measured in counter ticks. We do not show the plots for $b = 2$ and $b = 3$ here because they were similar to $b = 4$ and adding them made the figure more cluttered.

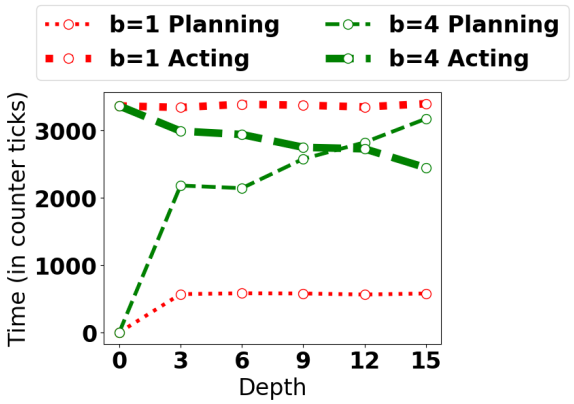

Figure 11: This figure shows that using the distance heuristic, the planning time (running time of RAEplan) increases with depth and the acting time (running time of RAE) decreases when it calls RAEplan with higher depth d. The time is measured in counter ticks.

to plan ahead. This approach has been extended with some planning capabilities in PropicePlan (Despouys and Ingrand 1999) and SeRPE (Ghallab, Nau, and Traverso 2016). Unlike our approach, those systems model commands 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.

Various acting approaches similar to PRS and RAE have been proposed, e.g., (Firby 1987; Simmons 1992; Simmons and Apfelbaum 1998; Beetz and McDermott 1994; Muscettola et al. 1998; Myers 1999). Some of these have refinement capabilities and hierarchical models, e.g., (Verma

et al. 2005; Wang et al. 1991; Bohren et al. 2011). While such systems offer expressive acting environments, e.g., with real time handling primitives, none of them provide 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.

(Musliner et al. 2008; Goldman et al. 2016; Goldman 2009) propose a way to do online planning and acting, but their notion of "online" is different from ours. In (Musliner et al. 2008), the old plan is executed repeatedly in a loop while the planner synthesizes a new plan (which the authors say can take a large amount of time), and the new plan isn't installed until planning has been finished. In RAEplan, hierarchical task refinement is used to do the planning quickly, and RAE waits until RAEplan returns.

The Reactive Model-based Programming Language (RMPL) (Ingham, Ragno, and Williams 2001) 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 (Williams and Abramson 2001; Conrad, Shah, and Williams 2009). 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 (Effinger, Williams, and Hofmann 2010). Probabilistic RMPL are introduced in (Santana and Williams 2014; Levine and Williams 2014) with the notions of weak and strong consistency, as well as uncertainty for contingent decisions taken by the environment or another agent. Our approach does not handle time; it focuses instead on hierarchical decomposition with Monte Carlo rollout and sampling.

Behavior trees (BT) (Colledanchise 2017; Colledanchise and Ögren 2017) can also respond reactively to contingent events that were not predicted. Planning synthesizes a BT that has a desired behavior. Building the tree refines the acting process by mapping the descriptive action model onto an operational model. Our approach is different since RAE provides the rich and general control constructs of a programming language and plans directly within the operational model, not by mapping from the descriptive to an operational model. Moreover, the BT approach does not allow for refinement methods, which are a rather natural and practical way to specify different possible refinements of tasks.

Approaches based on temporal logics and situation calculus (Doherty, Kvarnström, and Heintz 2009; Hähnel, Burgard, and Lakemeyer 1998; Claßen et al. 2012; Ferrein and Lakemeyer 2008) 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.

Our methods are significantly different from those used in HTNs (Nau et al. 1999): to allow for the operational models needed for acting, we use rich control constructs rather than simple sequences of primitives. The hierarchical representation framework of (Bucchiarone et al. 2013) includes abstract actions to interleave acting and planning for composing web services—but it focuses on distributed processes, which are represented as state transition systems, not operational models. It does not allow for refinement methods.

A wide literature on MDP-based probabilistic planning and Monte Carlo tree search refers to simulated execution, e.g., (Feldman and Domshlak 2013; 2014; Kocsis and Szepesvári 2006; James, Konidaris, and Rosman 2017) and sampling outcomes of action models e.g., RFF (Teichteil-Königsbuch, Infantes, and Kuter 2008), FF-replan (Yoon, Fern, and Givan 2007) and hindsight optimization (Yoon et al. 2008). 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 doing the planning online, they do the planning using descriptive models rather than operational models. There is no notion of integration of acting and planning, hence no notion of how to maintain consistency between the planner's descriptive models and the actor's operational models. Moreover, they have no notion of hierarchy and refinement methods.

Finally, 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. (Lallement, De Silva, and Alami 2014) shows how HTN planning can be used in robotics. (Garrett, Lozano-Perez, and Kaelbling 2018a) and (Garrett, Lozano-Pérez, and Kaelbling 2018b) integrates task and motion planning for robotics. (Coste-Maniere et al. 2017) integrates planning and execution for surgical planning algorithms used by surgical robots for laporoscopic and other minimally invasive surgery.

Conclusions and Future Work

In this paper, we discussed different ways to interleave acting and planning using operational models. Our actor is RAE and our planner is RAEplan-LookAhead. We came up with simple domain dependent heuristics in a simulated domain, called Search and Rescue and showed that performance improves with depth but the cost is that it also takes more time. Depending on the domain, the heuristics available and the performance requirements, the set of experiments done in this paper can help one identify the sweet spot.

In future, we plan to interleave acting and planning in some other domains and observe the changes in the performance of RAE and RAEplan-LookAhead. We also plan to do experiments using the lazy and concurrent strategies discussed in this paper.

References

Beetz, M., and McDermott, D. 1994. Improving robot plans during their execution. In *AIPS*.

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

Bucchiarone, A.; Marconi, A.; Pistore, M.; Traverso, P.; Bertoli, P.; and Kazhamiakin, R. 2013. Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*, 571–578.

Claßen, J.; Röger, G.; Lakemeyer, G.; and Nebel, B. 2012. Platas—integrating planning and the action language Golog. *KI-Künstliche Intelligenz* 26(1):61–67.

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

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

Conrad, P.; Shah, J.; and Williams, B. C. 2009. Flexible execution of plans with choice. In *ICAPS*.

Coste-Maniere, E.; Adhami, L.; Boissonnat, J.-D.; Carpentier, A.; and Guthart, G. S. 2017. Methods and apparatus for surgical planning. US Patent App. 15/397,498.

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

Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *J. Autonomous Agents and Multi-Agent Syst.* 19(3):332–377.

Effinger, R.; Williams, B.; and Hofmann, A. 2010. Dynamic execution of temporally and spatially flexible reactive programs. In *AAAI Wksp. on Bridging the Gap between Task and Motion Planning*, 1–8.

Feldman, Z., and Domshlak, C. 2013. Monte-carlo planning: Theoretically fast convergence meets practical efficiency. In *UAI*.

Feldman, Z., and Domshlak, C. 2014. Monte-carlo tree search: To MC or to DP? In *ECAI*, 321–326.

Ferrein, A., and Lakemeyer, G. 2008. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems* 56(11):980–991.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *AAAI*, 202–206. AAAI Press.

Garrett, C. R.; Lozano-Perez, T.; and Kaelbling, L. P. 2018a. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research* 37(1):104–136.

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2018b. Stripstream: Integrating symbolic planners and blackbox samplers. *arXiv preprint arXiv:1802.08705*.

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

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

Goldman, R. P. 2009. A semantics for htn methods. In *ICAPS*.

Hähnel, D.; Burgard, W.; and Lakemeyer, G. 1998. GOLEX – bridging the gap between logic (GOLOG) and a real robot. In *KI*, 165–176. Springer.

Ingham, M. D.; Ragno, R. J.; and Williams, B. C. 2001. A reactive model-based programming language for robotic space explorers. In *i-SAIRAS*.

James, S.; Konidaris, G.; and Rosman, B. 2017. An analysis of monte carlo tree search. In *AAAI*, 3576–3582.

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

Lallement, R.; De Silva, L.; and Alami, R. 2014. Hatp: An htn planner for robotics. *arXiv preprint arXiv:1405.5345*.

Levine, S. J., and Williams, B. C. 2014. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*.

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

Musliner, D. J.; Pelican, M. J.; Goldman, R. P.; Krebsbach, K. D.; and Durfee, E. H. 2008. The evolution of circa, a theory-based ai architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, volume 1205.

Myers, K. L. 1999. CPEF: A continuous planning and execution framework. *AI Mag.* 20(4):63–69.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *IJCAI*, 968–973.

Patra, S.; Gallab, M.; Nau, D.; and Traverso, P. 2018. Using operational models to integrate acting and planning. In *IntEx: ICAPS 2018 Workshop on Integrated Planning, Acting and Execution*.

Patra, S.; Gallab, M.; Nau, D.; and Traverso, P. 2019. Acting and planning using operational models. In *AAAI*. AAAI Press.

Santana, P. H. R. Q. A., and Williams, B. C. 2014. Chance-constrained consistency for probabilistic temporal plan networks. In *ICAPS*.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *IROS*, 1931–1937.

Simmons, R. 1992. Concurrent planning and execution for autonomous robots. *IEEE Control Systems* 12(1):46–50.

Teichteil-Königsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *ICAPS*.

Vaquero, T.; Roberts, M.; Bernardini, S.; Niemueller, T.; and Fratini, S., eds. 2018. *Proceedings of the 2nd Workshop on Integrated Planning, Acting, and Execution*, ICAPS 2018 Workshop.

Verma, V.; Estlin, T.; Jónsson, A. K.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *i-SAIRAS*.

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

Williams, B. C., and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*.

Yoon, S. W.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *AAAI*, 1010–1016.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, 352–359.