The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)

Acting and Planning Using Operational Models

Sunandita Patra,[1] Malik Ghallab,[2] Dana Nau,[1] Paolo Traverso[3]
[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
patras@cs.umd.edu, malik@laas.fr, nau@cs.umd.edu, traverso@fbk.eu

Abstract

The most common representation formalisms for planning are *descriptive models*. They abstractly describe *what* the actions do and are tailored for efficiently computing the next state(s) in a state transition system. But acting requires *operational models* that describe *how* to do things, with rich control structures for closed-loop online decision-making. Using descriptive representations for planning and operational representations for acting can lead to problems with developing and verifying consistency of the different models.

We define and implement an integrated acting-and-planning system in which both planning and acting use the same operational models, which are written in a general-purpose hierarchical task-oriented language offering rich control structures. The acting component is inspired by the well-known PRS system, except that instead of being purely reactive, it can get advice from the planner. Our planning algorithm, RAEplan, plans by doing Monte Carlo rollout simulations of the actor's operational models. Our experiments show significant benefits in the efficiency of the acting and planning system.

1 Introduction

Numerous knowledge representations have been proposed for describing and reasoning on actions. However, for the purpose of planning, the dominant representation is the one inherited from the early STRIPS system and formalized in various versions of the PDDL description language, including representations for planning under uncertainty, like PPDDL. This class of *descriptive models* of actions is tailored to efficiently compute the next states in a state transition system. It is quite limited. In particular it cannot reason about ongoing activities, react and adapt to an unfolding context. As argued by many authors, e.g., (Pollack and Horty 1999), plans are needed for acting deliberately, but they are not sufficient for realistic applications. Acting requires *operational models* that describe *how* to do things, with rich control structures for closed-loop online decision-making.

Most approaches for the integration of planning and acting seek to combine descriptive representations for the former and operational representations for the latter (Ingrand and Ghallab 2017). This has several drawbacks in particular for the development and consistency verification of the models. By 'consistency verification of the models', we mean a growing body of work (eg, in the context of certifying self-driving cars) for the formal verification of operational models, such as PRS-like procedures, using model checking and theorem proving. (de Silva, Meneguzzi, and Logan 2018) attempts to do this. It is highly desirable to have a single representation for both acting and planning. But this representation cannot be solely descriptive, because that wouldn't provide sufficient functionality. The planner needs to be able to reason directly with the actor's operational models.

We present an integrated planning and acting system in which both planning and acting use the actor's operational models. The acting component is inspired by the well-known PRS system (Ingrand et al. 1996). It uses 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. Commands are simulated when planning and performed by an execution platform in the real world when acting.

Rather than behaving purely reactively like PRS, our actor interacts with a planner, RAEplan. To decide how to refine tasks or events, RAEplan does Monte Carlo rollouts with applicable refinement methods. When a refinement method contains a command, RAEplan takes samples of its possible outcomes, using either a domain-dependent generative simulator, when available, or a probability distribution of its possible outcomes. We have implemented and evaluated our approach, and the results show significant benefits.

Next we discuss related work, followed by descriptions of the operational models, the acting and planning algorithms, our benchmark domains and experimental results, and a discussion of the results and concluding remarks.

2 Related 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 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. 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) (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.

Finally, 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.

3 Operational Models

Our formalism for operational models extends that of (Ghallab, Nau, and Traverso 2016, Chapter 3). Its features allow for dealing with a dynamic environment through an *execution platform* that handles sensing and actuation. The main ingredients are *state variables*, *tasks*, *events*, *refinement methods*, and *commands*. Let us illustrate the representation informally on simple examples.

Example 1. *Consider several robots (UGVs and UAVs) moving around in a partially known terrain, performing operations such as data gathering, processing, screening and monitoring. This domain is specified with the following:*

- *a set of robots,* $R = \{g_1, g_2, a_1, a_2\}$,

- *a set of locations,* $L = \{base, z_1, z_2, z_3, z_4\}$,
- *a set of tools,* TOOLS $= \{e_1, e_2, e_3\}$,
- loc$(r) \in L$ *and* data$(r) \in [0, 100]$, *for* $r \in R$, *are observable state variables that gives the current location and the amount of data the robot* r *has collected,*
- status$(e) \in \{\text{free}, \text{busy}\}$ *is an observable state variable that says whether the tool* e *is free or being used,*
- survey(r, l) *is a command performed by robot* r *in location* l *that surveys* l *and collects data.*

Let explore(r, l) *be a task for robot* r *to reach location* l *and perform the command* survey(r, l). *In order to survey, the robot needs some tool that might be in use by another robot. Robot* r *should collect the tool, then move to the location* l *and execute the command* survey(r, l). *Each robot can carry only a limited amount of data. Once its data storage is full, it can either go and deposit data to the base, or transfer it to an UAV via the task* depositData*(r). Here is a refinement method to do this.*

```
m1-explore(r, l)
   task: explore(r, l)
   body: getTool(r)
         moveTo(r, l)
         if loc(r) = l then:
            Execute command survey(r, l)
            if data(r) = 100 then depositData(r)
         else fail                                  □
```

Above, getTool(r), moveTo(r, l) *and* depositData(r) *are subtasks that need to be further refined via suitable refinement methods. Each robot can hold a limited amount of charge and is rechargeable. Depending on what locations it needs to move to,* r *might need to recharge by going to the base where the charger is located. Different ways of doing the task* get-Tool(r) *can be captured by multiple refinement methods. Here are two of them:*

```
m1-getTool(r)
   task: getTool(r)
   body: for e in TOOLS do
            if status(e)=free:
               l ← loc(e)
               moveTo(r, l)
               take(r, e)
               return
         // no tool is free
         fail
```

```
m2-getTool(r)
   task: getTool(r)
   body: for e in TOOLS do
            if status(e) = free:
               recharge(r)
               l ← loc(e)
               moveTo(r, l)
               take(r, e)
               return
         fail
```

UAVs can fly and UGVs can't, so there can be different possible refinement methods for the task moveTo(r, l) *based on whether* r *can fly or not.*

A refinement method for a task t specifies *how to* perform t, i.e., it gives a procedure for accomplishing t by performing subtasks, commands and state variable assignments. The procedure may include any of the usual programming constructs: if-then-else, loops, etc. Here is an example:

Example 2. *Suppose a space alien ☺ is spotted in one of the locations* $l \in L$ *of Example 1 and a robot has to react to it by stopping its current activity and going to* l. *Let us represent this with an event* alienSpotted*(l). We also need an additional state variable:* alien-handling*(r)*∈{T, F} *which indicates whether the robot* r *is engaged in handling an alien. A refinement method for this event is shown below. It can succeed if robot* r *is not already engaged in negotiating with another alien. After negotiations are over, the methods changes the value of* alien-handling*(r) to F.*

```
m-handleAlien(r, l)
   event: alienSpotted(l)
   body: if alien-handling(r) = F then:
            alien-handling(r) ← T
            moveTo(r, l)
            Execute command negotiate(r, l)
            alien-handling(r) ← F
         else fail                                  □
```

Commands correspond to programs to be run by the execution platform. They are used in RAEplan through a generative simulator or simply defined, as in our experiments, as probability distributions of their possible outcomes.

4 RAE

RAE (Refinement Acting Engine) is from (Ghallab, Nau, and Traverso 2016, Chapter 3). The first inner loop (line 1) reads each new *job*, i.e., each task or event that comes in from an *external source* such as the user or the execution platform, as opposed to the subtasks generated by refinement methods. For each such job τ, RAE creates a *refinement stack*, R_a (the a denotes acting), analogous to a computer program's execution stack. *Agenda* is the set of all current refinement stacks.

Algorithm 1: RAE (Refinement Acting Engine)

```
  Agenda ← empty list
  while True do
1     for each new task or event τ in the input stream do
          s ← current state; m ← RAEplan(s, τ, ∅, ⟨⟩)
          if m = failed then output("failed", τ)
          else
              R_a ← a new, empty refinement stack
              push (τ, m, nil, ∅) onto R_a
              insert R_a into Agenda
2     for each R_a ∈ Agenda do
          Progress(R_a)
3         if R_a is empty then remove it from Agenda
```

Task frames and refinement stacks. A *task frame* is a four-tuple $r = (\tau, m, i, tried)$, where τ is a task, m is the method instance used to refine τ, i is the current instruction in body(m), with i = 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 $R_a = \langle \rho_1, \ldots, \rho_n \rangle$. If R_a is nonempty, then top$(R_a) = \rho_1$; rest$(R_a) = \langle \rho_2, \ldots, \rho_n \rangle$; R_a = top(R_a).rest(R_a). To denote pushing ρ onto R_a, we write $\rho.R_a = \langle \rho, \rho_1, \rho_2, \ldots, \rho_n \rangle$. Refinement stacks used during planning will have the same semantics, but we will use the notation R_p instead of R_a to distinguish it from the acting stack.

Progression in methods. If $(\tau, m, i, \textit{tried}) = \text{top}(R_a)$ and s is the current state, then $\mathsf{next}(s, R_a)$ is the refinement stack produced by going to the next instruction after i. If m has more instructions after i, $\mathsf{next}(s, R_a) = (m, i', \tau, \textit{tried}).\text{rest}(R_a)$, where i' is the *next instruction* in body(m), taking into account the effects of control structures such as if-then-else, loops, etc. If body(m) has no more instructions to execute after i, then τ (the task in top(R_a)) is finished, hence $\mathsf{next}(s, R_a) = \text{rest}(R_a)$ modified with the topmost frame redirected to point the next instruction after τ. In other words, if rest$(R_a) = (\tau', m', j, \textit{tried}')$.rest(rest$(R_a)$) and j' is the *next instruction* in m' after j, then,

$$\begin{aligned}\mathsf{next}(s, R_a) &= (\tau', m', j', \textit{tried}').\text{rest}(\text{rest}(R_a)) \text{ if } j' \neq \text{NIL};\\ &= \mathsf{next}(s, \text{rest}(R_a)), \text{ otherwise.}\end{aligned}$$

In the second inner loop (RAE line 2), for each refinement stack in *Agenda*, RAE *progresses* the topmost *stack element* by one step, i.e., it executes the next instruction in the program (see the Progress subroutine at right). It may involve checking the status of a currently executing command (Progress line 1), following a control structure such as a loop or if-then-else (Progress line 2), executing an assignment statement, sending a command to the execution platform, or handling a subtask τ' by pushing a new stack element onto the stack (line 5 of Progress). If a method returns without failure, it has accomplished the task.

When RAE creates a stack element for a task τ, it must choose (line 1 of RAE, 4 of Progress, and 6 of Retry) a method instance m for τ. The version of RAE in (Ghallab, Nau, and Traverso 2016) chooses the methods from a *preference ordering* specified by the domain's author. Instead of that, we call a planner, RAEplan, to make an informed choice of m for accomplishing τ. m may have several subtasks. In line 3 of Progress, where m has a subtask τ', RAE calls RAEplan to choose a new method m' for τ'.

5 RAEplan

RAEplan does a recursive search to optimize a criterion. We first consider the simple case where the simulation of methods never fails; then we'll explain how to account for planning-time failures (which are distinct from running-time failures addressed by Retry) using an adequate criteria.

We choose a refinement method that has a refinement tree with a minimum expected cost for accomplishing a task τ (along with the remaining partially accomplished tasks in the current refinement stack).

Estimated Cost. Let $C^*(s, R_p)$ be the optimal expected cost, i.e., the expected cost of the optimal plan for accomplishing all the tasks in the refinement stack R_p in state s.

If R_p is empty, then $C^*(s, R_p) = 0$ because there are no tasks to accomplish. Otherwise, let $(\tau, m, i, \textit{tried}) = \text{top}(R_p)$. Then $C^*(s, R_p)$ depends on whether i is a command, an assignment statement, or a task:

<u>Progress(R_a):</u>
$(\tau, m, \textit{step}, \textit{tried}) \leftarrow \text{top}(R_a)$ `// step is the current step of m`
if *step* $\neq$ *nil* **then** `// m is running`
1   **if** *type*(*step*) = *command* **then**
    case execution-status(*step*):
      still-running: return;
      failed: Retry(R_a); return;
      successful: continue;
if *there are no more steps in* m **then**
  pop(R_a); return;
2 *step* $\leftarrow$ **Next Instruction** of m
case type(*step*):
  assignment: update s according to *step*; return;
  command: send *step* to execution platform; return;
  task: continue;
3 $\tau' \leftarrow$ *step*; $s \leftarrow$ current state ;
$R_p \leftarrow$ a copy of R_a
4 $m' \leftarrow \mathsf{RAEplan}(s, \tau', \emptyset, R_p)$
if m' = *failed* **then** Retry(R_a); return;
5 push $(\tau', m', \mathsf{nil}, \emptyset)$ onto R_a
<u>Retry(R_a):</u>
$(\tau, m, \textit{step}, \textit{tried}) \leftarrow \text{pop}(R_a)$
add m to *tried* `// m didn't succeed`
$R_p \leftarrow$ a copy of R_a
6 $s \leftarrow$ current state; $m' \leftarrow \mathsf{RAEplan}(s, \tau, \textit{tried}, R_p)$
if $m' \neq$ *failed* **then**
  push $(\tau, m', \mathsf{nil}, \textit{tried})$ onto R_a
else if R_a *is empty* **then**
  output("failed to accomplish", τ)
  remove R_a from *Agenda*
else Retry(R_a)

- If i is a command, then $C^*(s, R_p) =$

$$EV_{s' \in S'}\left[\{cost(s, i, s') + C^*(s', \mathsf{next}(s', R_p))\}\right], \quad (1)$$

  where S' is the set of outcomes of command i in s and EV stands for expected value.
- If i is an assignment statement, then $C^*(s, R_p) = C^*(s', \mathsf{next}(s', R_p))$, where s' is the state produced from s by performing the assignment statement.
- If i is a task, then $C^*(s, R_p)$ recursively optimizes over the candidate method instances for i. That is

$$C^*(s, R_p) = \min_{m' \in M'} C^*(s, (i, m', \mathsf{nil}, \emptyset).R_p),$$

  where $M' = \textit{Candidates}(i, s)$.

By computing $C^*(s, R_p)$, we can choose what method to use for a task. The algorithm for doing this is:

C*-Choice(s, τ, R_p)
  $M \leftarrow \textit{Candidates}(\tau, s)$
  return $\text{argmin}_{m \in M} C^*(s, (\tau, m, 0, \emptyset).R_p)$

Next, let us see how to account for planning failures. Note that C^* cannot handle failures because the cost of a failed command is ∞, resulting in an expected value of ∞ in equation 1 for all commands with at least one possibility of failure. In order to overcome this, we introduce the *efficiency* criteria, $\nu = 1/cost$, to measure the efficiency of a plan. RAEplan maximizes efficiency instead of minimizing 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

$$1/(c_1 + c_2) = e_1 e_2/(e_1 + e_2). \tag{2}$$

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

$$\begin{aligned} e_1 \bullet e_2 = e_2 \text{ if } e_1 = \infty, \text{ else} \\ e_1 \text{ if } e_2 = \infty, \text{ else } e_1 e_2/(e_1 + e_2). \end{aligned} \tag{3}$$

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 3 is associative.

Estimated efficiency. We now define $E^*_{b,k}(s, R_p)$ as an estimate of expected efficiency of the optimal plan for the tasks in stack R_p 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. Additional details are on the next page, in the *Experiments and Analysis* subsection.

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

- If i is a command, then $E^*_{b,k}(s, R_p) =$

  $$\tfrac{1}{k} \textstyle\sum_{s' \in S'} \tfrac{1}{cost(s,i,s')} \bullet E^*_{b,k}(s', \mathsf{next}(s', R_p)), \tag{4}$$

  where S' is a random sample of k outcomes of command i in state s, with duplicates allowed. Since S' has the probability distributions of the outcomes of the commands, it converges asymptotically to the expected value of E^*.
- If i is an assignment statement, then $E^*_{b,k}(s, R_p) = E^*_{b,k}(s', \mathsf{next}(s', R_p))$, where s' is the state produced from s by performing the assignment statement.
- If i is a task, then $E^*_{b,k}(s, R_p)$ recursively optimizes over the candidate method instances for i. That is:

  $$E^*_{b,k}(s, R_p) = \max_{m \in M'} E^*_{b,k}(s, (i, m, \mathsf{nil}, \emptyset).R_p), \tag{5}$$

  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 $\textit{Candidates}(i, s)$.

As we did with C*-Choice, by computing $E^*_{b,k}(s, R_p)$ we can choose what method to use for a task. The RAEplan algorithm is as follows, with b and k being global variables:

RAEplan$(s, \tau, \textit{tried}, R_p)$
  $M \leftarrow \textit{Candidates}(\tau, s) \setminus \textit{tried}$
  return $\text{argmax}_{m \in M} E^*_{b,k}(s, (\tau, m, 0, \textit{tried}).R_p)$

The larger the values of b and k in $E^*_{b,k}(s, R_p)$, the more plans RAEplan will examine. It can be proved that when $b = \max_{\tau,s}\{|\textit{Candidates}(\tau, s)|\}$ (call it b_{max}) and $k \to \infty$, the method instance returned by RAEplan converges to one with the maximum expected efficiency. We now outline the proof. It is by induction on the number of remaining push operations in R_p. In the base case, the number of remaining push operations in R_p is 1. This has to be a command, because if it were a task, then it would further refine into more commands, resulting in more push operations. The maximum expected efficiency for a command is just its expected value. The induction hypothesis is that for any stack R_p with n remaining push operations, $E^*_{b_{max},\infty}$ gives the maximum expected efficiency. In the inductive step, we show that equations 4 and 5 converge to the maximum expected efficiency for any R_p with $n + 1$ remaining push operations.[1]

6 Experimental Evaluation

Domains

We have implemented and tested our framework on four domains. The Explorable Environment domain (EE) extends the UAVs and UGVs setting of Example 1 with a total of 8 tasks, 17 refinement methods and 14 commands. It has dead ends because robots may run of charge in isolated locations.

The Chargeable Robot Domain (CR) consists of several robots moving around to collect objects of interest. Each robot can hold a limited amount of charge and is rechargeable. It may or may not carry the charger. They use Dijkstra's shortest path algorithm to move between locations. They don'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. The environment is dynamic due to emergency events as in Example 2. A task reaches a dead end if a robot is far away from the charger and runs out of charge. CR has 6 tasks, 10 methods and 9 commands.

The Spring Door domain (SD) has several robots trying to move objects from one room to another in an environment with a mixture of spring doors and ordinary doors. Spring doors close themselves unless they are held. A robot cannot simulataneously carry an object and hold a spring door open, so it must ask for help from another robot. Any robot that's free can be the helper. The environment is dynamic because the type of door is unknown to the robot. There are no dead ends. SD has 5 tasks, 9 methods and 9 commands.

The Industrial Plant domain (IP) consists of an industrial workshop environment, as in the RoboCup Logistics League competition. There are several fixed machines for painting, assembly, wrapping and packing. As new orders for assembly, paint, etc., arrive, carrier robots transport the necessary objects to the required machine's location. An order can be compound, e.g., paint two objects, assemble them together, and pack the resulting object. Once the order is done, the product is delivered to the output buffer. The environment is dynamic because the machines may get damaged and need repair before being used again; but there are no dead ends. IP has 9 tasks, 16 methods and 9 commands.

[1] Full proof at http://www.cs.umd.edu/~patras/theorems.pdf

Domain	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
CR	✓	✓	✓	–	✓
EE	✓	✓	–	✓	✓
SD	✓	–	–	✓	✓
IP	✓	–	–	✓	✓

Figure 1: Properties of our domains

Figure 1 summarizes the different properties of these domains. CR includes a model of a *sensing* action that a robot can use to identify what objects are at a given location. SD models a situation where robots need to collaborate, and can ask for help from each other. EE models a combination of robots with different capabilities (UGVs and UAVs) whereas in the other three domains all robots have same capabilities. It also models collaboration. In the IP domain, the allocation of tasks among the robots is hidden from the user. The user just specifies their orders; the delegation of the sub-tasks (movement of objects to the required locations) is handled inside the refinement methods. CR and EE can have dead-ends, whereas SD and IP do not have dead-ends.[2]

Experiments and Analysis

To examine how RAE's performance might depend on the amount of planning, we created a suite of test problems for the four domains. Each test problem consists of a job to accomplish, that arrives at a randomly chosen time point in RAE's input stream. For each such time point, we chose a random value and held it fixed throughout the experiments.

Recall that RAE's objective is to maximize the expected efficiency of a job's refinement tree, and the number of plans generated by RAEplan depends on b (how many different methods to try for a task) and k (how many times to simulate a command). The number of plans examined by RAEplan is exponential in b and k. As a special case, $k = 0$ runs RAE purely reactively, with no planning at all.

We ran experiments with $k = 0, 3, 5, 7, 10$. In the CR, EE and IP domains we used $b = 1, 2, 3$ because each task are at most three method instances. In the SD domain, we used $b = 1, 2, 3, 4$ because it has four methods for opening a door.

In the CR, EE, SD and IP domains, our test suites consisted of 15, 12, 12, and 14 problems respectively. We ran each problem 20 times to account for the effect of probabilistic non-deterministic commands. In our experiments, we used simulated versions of the four environments, running on a 2.6 GHz Intel Core i5 processor. The average (over 20 runs) running time for our experiments ranged from one minute to 6-7 minutes per test suite.

Efficiency. Figures 2 and 3 show how the average efficiency E depends on b and k. We see that efficiency increases with increase in b and k as expected. This is true for all four domains. In the CR domain, efficiency increases considerably as we move from $b = 1$ to $b = 2$, then (specifically when $k = 3$ and 5) decreases slightly as we move to $b = 3$. This is possibly because the commands present in the third method

[2]Full code is online at https://bitbucket.org/sunandita/raeplan.

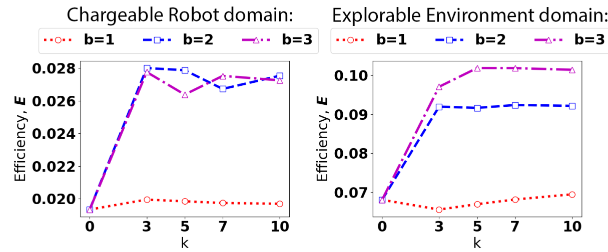

Figure 2: Efficiency E averaged over all of the jobs, for various values of b and k in domains with dead ends.

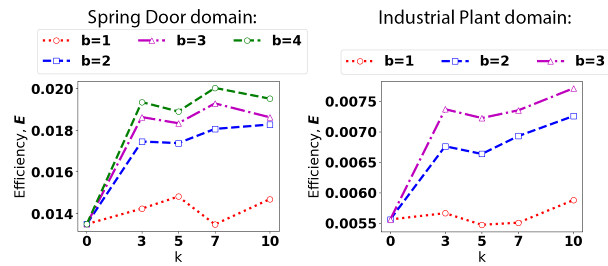

Figure 3: Efficiency E averaged over all of the jobs, for various values of b and k in domains *without* dead ends.

require more sampling to make accurate predictions. Indeed, with more samples, $k = 7$ and 10, $b = 3$ has better efficiency than $b = 2$. In the EE domain, we see that the efficiency improves up to $k = 5$ and then remains stable, indicating that 5 samples are enough for this domain. In the domains without dead ends (SD and IP), we see a gradual increase in efficiency with k. In Figure 2, the large increase in efficiency between $b = 1$ and $b = 2$ (as opposed to a more uniform increase) is because RAEplan explores methods according to a preference ordering specified by the domain's author. For many of the problems in our test suite, the 2^{nd} method in the preference ordering turned out to be the one with the largest expected efficiency. These experiments confirm our expectation that efficiency improves with b and k.

Success ratio. We wanted to assess how robust RAE was with and without planning. Figures 4 and 5 show RAE's *success ratio*, i.e., the proportion of jobs successfully accomplished in each domain. For the domains with dead ends (CR and EE), the success ratio increases as b increases. However, in the CR domain, there is some decrease after $k = 3$ because we are optimizing efficiency, not robustness. Formulating an explicit robustness criterion is non-trivial and will require further work. For the success ratio experiments, when we say we're not optimizing robustness, we mean we're not optimizing a specific criterion that leads to better recovery if an unexpected event causes failure. RAEplan looks for the most efficient plan. In our efficiency formula in Eqs. (2,3), a plan with a high risk of failure will have low efficiency, but so will a high-cost plan that always succeeds.

In the SD domain, b or k didn't make very much difference in the success ratio. In fact, for some values of b and k, the success ratio decreases. This is because in our prefer-

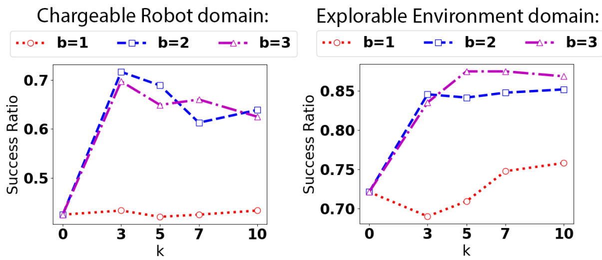

Figure 4: Success ratio (# of successful jobs/ total # of jobs) for various values of b and k in domains with dead ends.

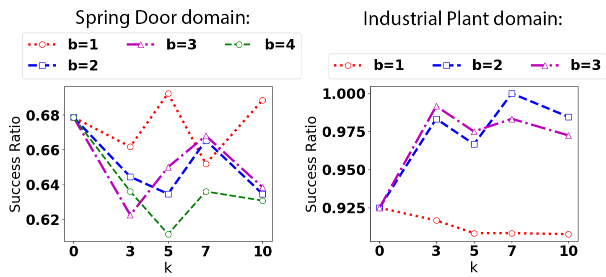

Figure 5: Success ratio (# of successful jobs/ total # of jobs) for various values of b and k in domains *without* dead ends.

ence ordering for the methods of the SD domain, the methods appearing earlier are better suited to handle the events in our problems whereas the methods appearing later produce plans that have lower cost but less robust to unexpected events. In the IP domain, we observe that success ratio increases with increase in b and k.

Retry ratio. Figures 6 and 7 shows the *retry ratio*, i.e., the number of times that RAE had to call the Retry procedure, divided by the total number of jobs to accomplish.

The Retry procedure is called when there is an execution failure in the method instance m that RAE choses for a task τ. Retry tries another applicable method instance for τ that it hasn't tried already. This is significantly different from backtracking since the failed method m has already been partially executed; it has changed the current state. In real-world execution there is no way to backtrack to a previous 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.

The retry ratio generally decreases from $b = 1$ to $b = 2$ and 3. This is because higher values of b and k make RAEplan examine a larger number of alternative plans before choosing one, thus increasing the chance that it finds a better method for each task. Hence, planning is important in order to reduce the number of retries. The reason the retry ratio increases from $b = 2$ to 3 for some points in IP and EE is that for a reasonable number of test cases, the third method in the preference ordering for the tasks appears to be more efficient but when executed, it is leading to a large number of retries, increasing the retry ratio.

In summary, for all the domains, planning with RAEplan clearly outperforms purely reactive RAE.

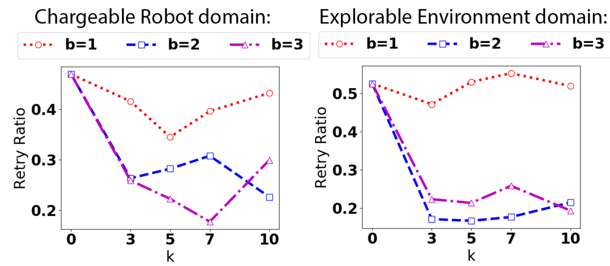

Figure 6: Retry ratio (# of retries / total # of jobs) for various values of b and k in domains with dead ends.

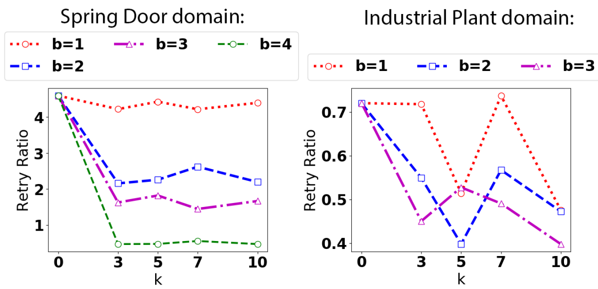

Figure 7: Retry ratio (# of retries / total # of jobs) for various values of b and k in domains *without* dead ends.

7 Concluding Remarks

We have proposed a novel acting and planning system for integrating acting and planning using the actor's operational models. Our experimentation covers different interesting aspects of realistic domains, like dynamicity, and the need for run-time sensing, information gathering, collaborative and concurrent tasks (see Figure 1). We have shown the difference between domains with dead ends, and domains without dead ends through three different performance metrics: efficiency (reciprocal of the cost), which is the optimization criteria of RAEplan, success ratio and retry ratio. Acting purely reactively in the domains with dead ends can be costly and risky. The homogeneous and sound integration of acting and planning provided by RAE and RAEplan is of great benefit for all the domains. This is reflected via a higher efficiency.

The retry ratio measures the execution effectiveness. Performing many retries is not desirable, since this has a high cost and faces the uncertainty of execution. We have shown that both in domains with dead ends and without, the retry ratio significantly diminishes with RAEplan.

Finally we have devised a novel, and we believe realistic and practical way, to measure the performance of RAE and similar systems. While most often the experimental evaluation of systems addressing acting and planning is simply performed on the sole planning functionality, we devised an *efficiency* measure to assess the overall performance to plan and act, including failure cases. This criteria takes into account that the cost to execute commands in the real world, which is usually much larger than the computation cost.

We have shown that the integration of acting and planning reduces cost significantly. Future work will include more elaborate experiments, with more domains and test cases. We also plan to test with different heuristics, compare RAE-plan with other approaches cited in the related work, and finally do testing in the physical world with actual robots.

Acknowledgements. We thank our anonymous AAAI reviewers for their excellent feedback and comments.

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*.

de Silva, L.; Meneguzzi, F.; and Logan, B. 2018. An Operational Semantics for a Fragment of PRS. In *IJCAI*.

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.

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*.

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

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

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.

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.

Pollack, M. E., and Horty, J. F. 1999. There's more to life than making plans: Plan management in dynamic, multiagent environments. *AI Mag.* 20(4):1–14.

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*.

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.