# Using Operational Models to Integrate Acting and Planning

Sunandita Patra, Malik Ghallab, Dana Nau, Paolo Traverso

# Using Operational Models to Integrate Acting and Planning

**Sunandita Patra**[1], **Malik Ghallab**[2], **Dana Nau**[3], **Paolo Traverso**[4]
[1,3]University of Maryland, College Park, USA [2]LAAS-CNRS, France, [4]FBK-ICT, Trento, Italy
[1]patras@umd.edu, [2]malik@laas.fr, [3]nau@cs.umd.edu, [4]traverso@fbk.eu
Workshop on Hybrid Planning, ICAPS 2018

## Abstract

A significant problem for integrating acting and planning is how to maintain consistency between the planner's *descriptive* action models, which abstractly describe *what* the actions do, and the actor's *operational* models, which tell *how* to perform the actions with rich control structures for closed-loop online decision-making. Operational models allow for dealing with a variety of contexts, and responding to unexpected outcomes and events in a dynamically changing environment.

To circumvent the consistency problem, we use the actor's operational models both for acting and for planning. Our acting-and-planning algorithm, APE, uses hierarchical operational models inspired from those in the well-known PRS system. But unlike the reactive PRS algorithm, APE chooses its course of action using a planner that does Monte Carlo sampling over simulated executions of APE's operational models.

Our experiments with this approach show significant benefits in the success rates of the acting system, in particular for domains with dead ends.

## Introduction

The integration of acting and planning is a long-standing AI problem that has been discussed by many authors. For example, (Pollack and Horty 1999) argue that despite successful progress to go beyond the restricted assumptions of classical planning (e.g., handle uncertainty, partial observability, or exogenous events), in most realistic applications just making plans is not enough. Their argument still holds. Planning, as a search over predicted state changes, uses *descriptive models* of actions (*what* might happen). Acting, as an adaptation and reaction to an unfolding context, requires *operational models* of actions (*how* to do things) with rich control structures for closed-loop online decision-making.

A recent survey shows that most approaches to integrating acting and planning seek to combine descriptive and operational representations, using the former for planning and the latter for acting (Ingrand and Ghallab 2017). This has several drawbacks in particular for the development and consistency verification of the models. To ensure consistency, it is highly desirable to have a single representation for both acting and planning. But if this representation were a descriptive one, it wouldn't provide sufficient functionality. Instead, the planner needs to be capable of reasoning directly with the actor's operational models.

In this paper, we provide an integrated acting-and-planning system, APE (Acting and Planning Engine). APE's operational representation language and its acting algorithm are inspired by the well-known PRS system (Ingrand et al. 1996). The operational model is hierarchical: a collection of refinement methods offers 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 *commands* (including sensing commands), which are sent to an execution platform in order to execute them in the real world, and *subtasks*, which need to be refined recursively. APE's acting engine is based on an expressive, general-purpose operational language with rich control structures for closed-loop online decision-making.

To integrate acting and planning, APE extends the reactive PRS-like acting algorithm to include a planner, APE-plan. At each point where APE needs to decide how to refine a task, subtask, or event, APE-plan does Monte Carlo rollouts with a subset of the applicable refinement methods. At each point where a refinement method contains a command to the execution platform, APE-plan takes samples of its possible outcomes using a predictive model of what each command will do.

We have implemented APE and APE-plan and have done preliminary empirical assessments of them on four domains. The results show significant benefits in the success rates of the acting system, in particular for domains with dead ends.

The related work is described in the following section. Then we briefly summarize the operational model. APE and APE-plan are presented in the following section. We present our benchmark domains and experimental results. Finally, we discuss the results and provide conclusions.

## Related Work

To the best of our knowledge, no previous approach has proposed the integration of acting and planning by looking directly within the language of a true operational model like that of APE. Our approach is based on the operational representation language and RAE algorithm in (Ghallab, Nau, and Traverso 2016, Chapter 3), which in turn were inspired

by PRS (Ingrand et al. 1996). RAE operates purely reactively. If it needs to choose among several refinement methods that are eligible for a given task or event, it makes the choice without any attempt to plan ahead. The approach has been extended with some planning capabilities in PropicePlan (Despouys and Ingrand 1999) and SeRPE (Ghallab, Nau, and Traverso 2016). The two systems model commands with 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 APE 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 provide refinement capabilities. 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 propose here. Most of the mentioned systems do not reason about alternative refinements.

Finite State Automata (FSA) and Petri Nets have also been used as representations for acting models, e.g.,(Verma et al. 2005; Wang et al. 1991), again without planning capability. For example, the ROS execution system SMACH (Bohren et al. 2011), implements an automata-based approach, where each state of a hierarchical state machine corresponds to the execution of a command. However, the semantics of constructs available in SMACH is limited for reasoning on goals and states, and there is no planning.

The Reactive Model-based Programming Language (RMPL) (Ingham, Ragno, and Williams 2001) is an object-oriented language that allows a domain to be structured through an object hierarchy with subclasses and multiple inheritance. It combines a system model with a control model, using state-based, procedural control and temporal representations. 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 Temporal Plan Networks (TPN)(Williams and Abramson 2001), an extension of Simple Temporal Networks with symbolic constraints and decision nodes. Temporal reasoning consists in finding a path, i.e., a plan, in the TPN that meets the constraints. The execution of generated plans allows for online choices (Conrad, Shah, and Williams 2009). TPNs are extended with error recovery, temporal flexibility, and conditional execution based on the state of the world (Effinger, Williams, and Hofmann 2010). Primitive tasks are specified with distributions of their likely durations. A probabilistic sampling algorithm finds an execution guaranteed to succeed with a given probability. Probabilistic TPN are introduced in (Santana and Williams 2014) with the notions of weak and strong consistency. (Levine and Williams 2014) add the notion of uncertainty to TPNs 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. It is a quite comprehensive CSP-based approach for temporal planning and acting; it provides refinement, instantiation, time, nondeterminism, a plan repair. Our approach does not handle time; it focuses instead on decomposition into communicating asynchronous components.

Behavior trees (BT) (Colledanchise 2017; Colledanchise and Ögren 2017) aim at integrating acting and planning within a hierarchical representation. Similarly to our framework, a BT can reactively respond to contingent events that were not predicted. The authors propose a mechanism to synthesize a BT that has a desired behavior. The construction of the tree refines the acting process by mapping the descriptive model of actions onto an operational model. Our approach is different since APE provides the rich and general control constructs of a programming language and we do planning directly within the operational model, rather than through a 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 or 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 used in APE. Moreover, these approaches integrate acting and planning without exploiting the hierarchical approach based on refinement methods described in this paper.

Our framework has some similarities with HTN (see, e.g., (Nau et al. 1999)), since tasks can be refined with different methods. However, our methods are significantly different from HTN ones since our methods are programs that can encode rich control constructs rather than simple sequences of primitive tasks. This is what allows us to provide a framework for acting and planning.

(Bucchiarone et al. 2013) propose a hierarchical representation framework that includes abstract actions and that can interleave acting and planning for composing web services. However this work focus on distributed processes, which are represented as state transition systems, and does not allow for refinement methods.

Finally, a wide literature on probabilistic planning and Monte Carlo tree search refers to simulated execution, e.g., (Feldman and Domshlak 2013; Feldman and Domshlak 2014; Kocsis and Szepesvári 2006; James, Konidaris, and Rosman 2017) and sampling outcomes of action models e.g., the RFF algorithm (Teichteil-Königsbuch, Infantes, and Kuter 2008), FF-replan (Yoon, Fern, and Givan 2007) and hindsight optimization (Yoon et al. 2008). Beyond the fact that all these works are based on a probabilistic MDP framework, the main conceptual and practical difference with our work is that they consider just a descriptive model, i.e., abstract actions on finite MDPs. Their focus is therefore entirely on planning, and do not allow for an integration of acting and planning. Most of the papers refer to doing the planning online – but they are doing 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.

## Operational Models

Our formalism for operational models of actions is based on the one in (Ghallab, Nau, and Traverso 2016, Chapter 3). It has features that allow for dealing with a dynamic environment which has other actors and exogenous events. The main ingredients are *tasks*, *events*, *commands*, *refinement methods*, and *state variables*. Some of the state variables are *observable*, i.e., the execution platform will automatically keep them up-to-date through sensing operations. We illustrate this representation through the following 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. In this domain, let*

- $R = \{g_1, g_2, a_1, a_2\}$ *be the set of robots,*
- $L = \{base, z_1, z_2, z_3, z_4\}$ *be the set of locations,*
- survey$(r, l)$ *be a command performed by robot $r$ in location $l$ that surveys $l$ and collects data*
- loc$(r) \in L$ *and* data$(r) \in [0, 100]$ *be observable state variables that contain the robot $r$'s current location and the amount of data it has collected.*

*Let* explore$(r, l)$ *be a task for robot $r \in R$ to reach location $l \in L$ and perform the command* survey$(r, l)$*. In order to survey, the robot needs some equipment that might either be available or in use by another robot. Robot $r$ should collect the equipment, 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: get-Equipment(r, 'survey')
        moveTo(r, l)
        if loc(r) = l then:
            Execute command survey(r, l)
            if data(r) = 100 then:
                depositData(r)
            return success
        else return failure
```

*Above,* get-Equipment$(r,$ *'survey'*$)$*,* moveTo$(r, l)$ *and* depositData$(r)$ *are subtasks which need to be further refined via suitable refinement methods. Only UAVs have the ability to fly. So, there can be different possible refinement methods for the task* moveTo$(r, l)$ *based on whether $r$ can fly or not.*

*Each robot can hold a limited amount of charge and is rechargeable. Depending on what locations it needs to survey, it might need to recharge by going to the base where the charger is located. Different ways of doing this can be captured by multiple refinement methods for the task* doActivities$(r,$ *locList*$)$*. Here are two of them:*

```
m1-doActivities(r, locList)
  task: doActivities(r, locList)
  body: for l in locList do:
            explore(r, l)
            moveTo(r, 'base')
            if loc(r) = 'base':
                recharge(r)
            else return failure
        return success
```

```
m2-doActivities(r, locList)
  task: doActivities(r, locList)
  body: for l in locList do:
            explore(r, l)
            moveTo(r, 'base')
            if loc(r) = 'base':
                recharge(r)
            else return failure
        return success
```

Note that 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. This procedure can include any of the usual programming constructs, e.g., if-then-else, loops and so forth.

The above example illustrates tasks and refinement methods. Let us give the robots a method for reacting to an event.

**Example 2.** *Suppose that an 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$(r) \in \{$T, F$\}$ *which indicates whether the robot $r$ is engaged in handling an alien. Here is a refinement method for this event:*

```
m-handleAlien(r, l)
  event: alienSpotted(l)
  body: if alien-handling(r) = F then:
            alien-handling(r) ← T
            moveToAlien(r, l)
            Execute command negotiate-with-alien(r, l)
            alien-handling(r) ← F
            return success
        else return failure
```

*This method 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.*

## APE and APE-plan

Algorithm 1, APE (Acting and Planning Engine), is based loosely on the RAE (Refinement Acting Engine) algorithm in (Ghallab, Nau, and Traverso 2016, Chapter 3). APE's 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 APE's refinement methods. For each such job $\tau$, APE creates a *refinement stack* analogous to a computer program's execution stack. *Agenda* is the set of all current refinement stacks.

In the second inner loop (line (4)), for each refinement stack in *Agenda*, APE *progresses* the topmost *stack element* by one step. The stack element includes (among other things) a task or event $\tau$ and the method instance $m$ that APE has chosen to use for $\tau$. The body of $m$ is a program, and progressing the stack element (the Progress subroutine) means executing the next step in this program. This may involve monitoring the status of a currently executing command (line (6)), following a control structure such as a loop or if-then-else (line (7)), executing an assignment statement, sending a command to the execution platform, or handling a subtask $\tau'$ by pushing a new stack element onto the stack

```
APE( )
  Agenda ← empty list
  loop
    for each new task or event τ in the input stream, do                (1)
      s ← current state
      M ← {applicable method instances for τ in state s}
      T ← APE-plan(M, s, τ)                                             (2)
      if T = failed then output("failed to address", τ)
      else do
        m ← the method instance at the top of T                         (3)
        stack ← a new, empty refinement stack
        push (τ, m, nil, ∅) onto stack
        insert stack into Agenda
    for each stack ∈ Agenda do                                          (4)
      Progress(stack)
      if stack is empty then remove it from Agenda                      (5)

Progress(stack)
  (τ, m, step, tried) ← top(stack)
  if step ≠ nil then  // i.e., if we have started executing m
    // step is the current step of m
    if type(step) = command then                                        (6)
      // step is running on the execution platform
      case execution-status(step):
        still-running: return
        failed:        Retry(stack); return
        successful:    pass     // continue to next line
  if there are no more steps in m then pop(stack); return
  step ← next step of m after accounting for the effects
         of control statements (loops, if-then-else, etc.)              (7)
  case type(step):
    assignment: update s according to step; return
    command:
        send step to the execution platform; return                     (8)
    task: pass          // continue to next line
  τ' ← step;  s ← current state                                         (9)
  M' ← {applicable method instances for τ' in state s}
  T' ← APE-plan(M', s, τ')                                              (10)
  if T' = failed then Retry(stack); return
  m' ← the method instance at the top of T'                             (11)
  push (τ', m', nil, ∅) onto stack                                      (12)

Retry(stack)
  (τ, m, step, tried) ← pop(stack)
  add m to tried   // the things we tried that didn't work
  s ← current state
  M ← {applicable method instances for τ in state s}
  T ← APE-plan(M \ tried, s, τ)                                         (13)
  if T ≠ failed then
    m' ← the method instance at the top of T                            (14)
    push (τ, m', nil, tried) onto stack
  else if stack is empty then
    output("failed to accomplish", τ)
    remove stack from Agenda
  else Retry(stack)
```

Algorithm 1: APE, the Acting and Planning Engine.

(line (12)). A method succeeds in accomplishing a task when it returns without any failure.

Whenever APE creates a stack element for a task $\tau$, it must choose (lines (3), (11), and (14)) a method instance $m$ for $\tau$. In order to make an informed choice of $m$, APE calls (lines (2), (10), and (13)) a planner, APE-plan, that returns a plan for accomplishing $\tau$. The returned plan, $T$, will begin with a method $m$ to use for $\tau$. If $m$ contains subtasks, then $T$ must include methods for accomplishing them (and so forth recursively), so $T$ is a tree with $m$ at the root.

Once APE has selected $m$, it ignores the rest of $T$. Thus in line (9), where $m$ has a subtask $\tau'$, APE doesn't use the method that $T$ used for $\tau'$. Instead, in line (11), APE calls APE-plan to get a new plan $T'$ for $\tau'$. This is a receding-horizon search analogous to how a game-playing program might call an alpha-beta game-tree search at every move.[1]

The pseudocode of APE-plan is given in the appendix. It is a modified version of the APE pseudocode that incorporates these main modifications:

1. Each call to APE-plan returns a *refinement tree* $T$ whose root node contains a method instance $m$ to use for $\tau$. The children of this node include a refinement tree (or terminal node) for each subtask (or command, respectively) that APE-plan produced during its Monte Carlo rollout of $m$.
2. In lines (2), (10), and (13), APE-plan calls itself recursively on a set $M' \subseteq M$ that contains the first $b$ members of $M$ a list of method instances ordered according to some domain-specific preference order (with $M' = M$ if $|M| < b$), where $b$ is a parameter called the *search breadth*.This produces a set of refinement trees. If the set is nonempty, then APE-plan chooses one that optimizes cost, time or any other user-specified objective function. If the set is empty, then APE-plan returns the first method instance from $M'$ if $|M'| >= 1$; otherwise it returns failed. See Figures 5 and 6 in the appendix for more details.
3. Each call to Retry is replaced with an expression that just returns failed. While APE needs to retry in the real world with respect to the real actual state, APE-plan considers that a failure is simply a dead end for that particular sequence of choices.
4. In line (8) (the case where step is a command), instead of sending *step* to the actor's execution platform, APE-plan invokes a predictive model of what the execution platform would do. Such a predictive model may be any piece of code capable of making such a prediction, e.g., a deterministic, nondeterministic, or probabilistic state-transition model, or a simulator of some kind. Since different calls to the predictive model may produce different results, APE-plan calls it $b'$ times, where $b'$ is a parameter called the *sample breadth*. From the $b'$ trial runs, APE-plan gets an estimate of *step*'s expected time, cost, and probability of leading to success. See Figures 8 and 10 in the appendix for more details.

[1](Ghallab, Nau, and Traverso 2016) describes a "lazy lookahead" in which an actor keeps using its current plan until an unexpected outcome or event makes the plan incorrect, and a "concurrent lookahead" in which the acting and planning procedures run concurrently. We tried implementing these for APE, but in our experimental domains they did not make much difference in APE's performance.

5. Finally, APE-plan has a *search depth* parameter $d$. When APE calls APE-plan, APE-plan continues planning either to completion or depth $d$, whichever comes earlier. Such a parameter can be useful in real-time environments where there may not be enough time to plan all the way to completion.

## 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 some additional tasks and refinement methods. This domain has dead ends because a robot may run of charge in an isolated location.

The Chargeable Robot Domain (CR) consists of several robots moving around to collect objects of interest. The robots can hold a limited amount of charge and are rechargeable. To move from one location to another, the robots use Dijkstra's shortest path algorithm. The robots don't know where objects are unless a sensing action is performed in the object's location. They have to search for an object before collecting it. Also, the robot may or may not carry the charger with it. The environment is dynamic due to emergency events as in Example 2. A task reaches a dead end when a robot, which is far away from the charger, has run out of charge.

The Spring Door domain (SD) has several robots are 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 carry an object and hold a door simultaneously. So, whenever it needs to move through a spring door, it needs to ask for help from another robot. Any robot which is free can act as the helper. The environment is dynamic because the the type of door is unknown to the robot. But, there are no dead ends.

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 complex, like, paint two objects, assemble them together, and pack the resulting object. Once the order is done, the final 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.

These four domains have different properties, summarized in Figure 1. CR includes a model for the sensing action where the robot can sense a location and identify objects in that location. SD models a situation where robots need to collaborate with each other. They 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 like the SD domain. 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 subtasks (movement of objects to the required locations) is handled inside the refinement methods. CR and EE are domains that can represent dead-ends, whereas SD and IP do not have dead-ends.

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

Figure 1: Properties of our domains

### Experiments and Analysis

The objective of our experiments was to examine how APE's performance might depend on the amount of planning that we told APE to do. For this purpose, we created a suite of test problems. Each test problem included 1 to 4 jobs to accomplish, and for each job, there was a randomly chosen time point at which it would arrive in APE's input stream.

The amount of planning done by APE-plan depends on its search breadth $b$, sample breadth $b'$, and search depth $d$. We used $b' = 1$ (one outcome for each command), and $d = \infty$ (planning always proceeded to completion), and five different search breadths, $b = 0, 1, 2, 3, 4$. Since APE tries $b$ alternative refinement methods for each task or subtask, the number of alternative plans examined by APE is exponential in $b$. As a special case, $b = 0$ means running APE in a purely reactive way without any planning at all. Our objective function for the experiments is the number of commands in the plan.

In the CR, EE, SD and IP domains, our test suites consisted of 60, 54, 60, and 84 problems, with the numbers of jobs to accomplish being 114, 126, 84 and 276, respectively. In our experiments we used simulated versions of the four environments, running on a 2.6 GHz Intel Core i5 processor.

**Success ratio.** Figure 2 shows APE's *success ratio*, the proportion of jobs that it successfully accomplished in each domain. For the two domains with dead ends (CR and EE), the success ratio generally increases as we increase the value of $b$. In the CR domain, the success ratio makes a big jump from $b = 1$ to $b = 2$ and then remains nearly the same for $b = 2, 3, 4$. This is because for most of the CR tasks, the second method in the preference ordering (in our experiments, this order is decided by the domains' author)turned out to be the best one, so higher value of $b$ did not help much. In contrast, in the EE domain, the success ratio continued to improve significantly for $b = 3$ and $b = 4$.

In the domains with no dead ends, $b$ didn't make very much difference in the success ratio. In the IP domain, $b$ made almost no difference at all. In the SD domain, the success ratio even decreased slightly from $b = 1$ to $b = 4$. This is because in our preference ordering for the tasks of the SD domain, the methods appearing earlier are better suited to handle the events in our problems whereas the methods ap-

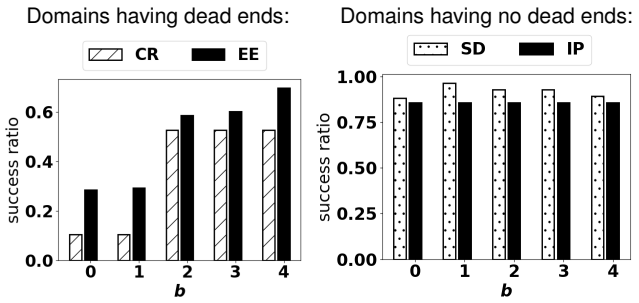


Figure 2: Success ratio (number of successful jobs / total number of jobs) for different values of search breadth $b$.

pearing later produce plans that are shorter but less robust to unexpected events. These experiments confirm the expectation that planning is critical in domains where the actor may get stuck in dead ends. It also has benefits in acting costs (the *retry ratio* and *speed to success* measurements described below).

**Retry ratio.** Figure 3 shows the *retry ratio*, i.e., the number of times that APE had to call the Retry procedure, divided by the total number of jobs to accomplish. The Retry procedure is called when there is a failure in the method instance $m$ that APE chose for some task $\tau$ (see Algorithm 1). Retry works by trying to use another applicable method instance for $\tau$ that it hasn't tried already. Although this is a little like backtracking, a critical difference is that since the method $m$ has already been partially executed, it has changed the current state, and in real-world execution (unlike planning), 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.

In all four of the domains, the retry ratio decreases slightly from $b = 0$ (purely reactive APE) to $b = 1$, and it generally decreases more as $b$ increases. This is because higher values of $b$ make APE-plan examine a larger number of alternative plans before choosing one, thus increasing the chance that it finds a better method for each task. In the CR domain, the big decrease in retry ratio from $b = 1$ to $b = 2$ corresponds to the increase in success ratio observed in Figure 2. The same is true for the EE domain at $b = 2$ and $b = 4$. Since the retry ratio decreases with increasing $b$ in all four domains, this means that the integration of acting and planning in APE is important in order to reduce the number of retries.

**Speed to success.** An acting-and-planning system's performance cannot be measured only with respect to the time to plan; it must also include the *time to success*, i.e., the total amount of time required for both planning and acting. Acting is in general much more expensive, resource demanding, and time consuming than planning; and unexpected outcomes and events may necessitate additional acting and planning.

For a successful job, the time to success is finite, but for a failed job it is effectively infinite. To make all of the numbers finite so that they can be averaged, we use the reciprocal

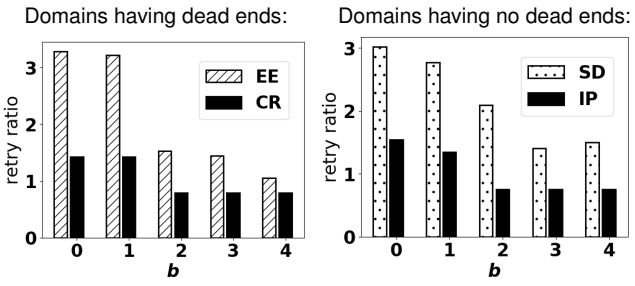


Figure 3: Retry ratio (number of retries / total number of jobs) for different values of search breadth $b$.

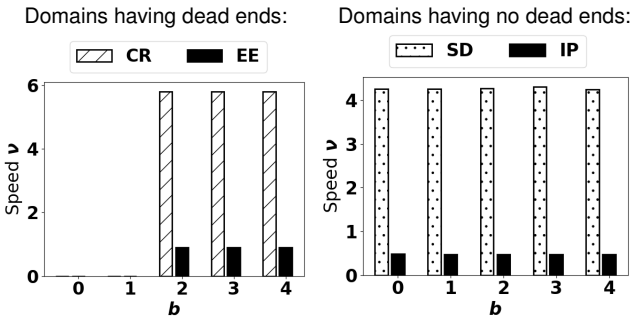


Figure 4: Speed to success $\nu$ averaged over all of the jobs, for different values of search breadth $b$.

amount, the *speed to success*, which we define as follows:

$$\nu = \begin{cases} 0 & \text{if the job isn't successful,} \\ \alpha/(t_p + t_a + n_c t_c) & \text{if the job is successful,} \end{cases}$$

where $\alpha$ is a scaling factor (we used $\alpha = 10,000$, otherwise all of our numbers would be very small), $t_p$ and $t_a$ are APE-plan's and APE's total computation time, $n_c$ is the number of commands sent to the execution platform, and $t_c$ the average amount of time needed to perform a command. In our experiments we used $t_c = 250$ seconds. The higher the average value of $\nu$, the better the performance.

Figure 4 shows how the average value of $\nu$ depends on $b$. In the domains with dead-ends (CR and EE), there is a huge improvement in $\nu$ from $b = 1$ (where $\nu$ is nearly 0) to $b = 2$. This corresponds to a larger number of successful jobs in less time. As we increase $b$ further, we only see slight change in $\nu$ for all the domains even though the success ratio and retry ratio improve (Figures 2 and 3). This is because of the extra time overhead of running APE-plan with higher $b$.

In summary, for domains with dead ends, planning with APE-plan outperforms purely reactive APE. The same occurs to some extent in the domains without dead ends, but there the effect is less pronounced thanks to the good domain specific heuristics in our experiments.

## Concluding Remarks

We have proposed a novel algorithm APE 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: the success ratio, retry ratio and speed to success. We saw that acting purely reactively in the domains with dead ends can be costly and dangerous. The homogenous and sound integration of acting and planning provided by APE is of great benefit for domains with dead ends which is reflected through a higher success ratio. In most of the cases, the success ratio increases with increase in the parameter, search breadth, $b$ of APE-plan. In the case of safely explorable domains, APE manages to have a similar ratio of success for all values of $b$.

Our second measure, the retry ratio, counts the number of retries of the same task done by APE before succeeding. 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 APE-plan, demonstrating the benefits of using APE-plan also in safely explorable domains.

Finally we have devised a novel, and we believe realistic and practical way, to measure the performance of APE 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 a *speed to success* measure to assess the overall time to plan and act, including failure cases. It takes into account that the time to execute commands in the real world are usually much longer than the actor's computation time. We have shown that, in general, the integration of APE-plan reduces time significantly in the case of domains with dead ends, while there is not such significant decrease in performance in the case of safely explorable domains.

Future work will include more elaborate experiments, with more domains and test cases, and different settings of APE-plan's search breadth, search depth, and sample breadth parameters. We also plan to test with different heuristics, compare APE with other approaches cited in the related work, and finally do testing in the physical world with actual robots.

## Appendix

In this section, we describe the pseudocode of APE-plan, the planner of our acting-and-planning engine, APE. $b$, $b'$ and $d$ are global variables representing the search breadth, sample breadth and search depth respectively (described in the main paper). The main procedure of APE-plan is shown in Figure 5. APE-plan receives as input a task $\tau$ to be planned for, a set of methods $M$ and the current state $s$. APE-plan returns a refinement tree $T$ for $\tau$. It starts by creating a refinement tree with a single node $n$ labeled $\tau$ and calls a sub-routine APE-plan-Task which builds a complete refinement tree for $n$.

APE-plan has three main sub-procedures: APE-plan-Task, APE-plan-Method and APE-plan-Command. APE-plan-Task looks at $b$ method instances for refining a task $\tau$. It calls

```
APE-plan (M, s, τ)
  n ← new tree node
  label(n) ← τ
  T_0 ← tree with only one node n
  (T, v) ← APE-plan-Task(s, T_0, n, M, 0)
  if v ≠ failure then
    return (T, v)
  else:
    B ← { Applicable method instances for τ in M
          ordered according to a preference ordering }
    if B ≠ ∅ then
      n ← Create new node
      label(n) ← B[1]
      T ← tree with only one node n as the root
      return (T, 0)
    else:
      return null, failure
```

Figure 5: The pseudocode of the planner used by APE

APE-plan-Method for each of the $b$ method instances and returns the tree with the most optimal *value*. Every refinement tree has a value based on probability and cost. Once APE-plan-Task has chosen a method instance $m$ for $\tau$, it re-labels the node $n$ from $\tau$ to $m$, in the current refinement tree $T$. Then it simulates the steps in $m$ one by one by calling the sub-routine APE-plan-Method.

```
APE-plan-Task (s, T, n, M, d_curr)
  τ ← label(n)
  B ← { Applicable method instances for τ in M ordered
        according to a preference ordering }
  if |B| < b then
    B' ← B
  else:
    B' ← B[1...b]
  U, V ← empty dictionaries
  for each m ∈ B' do
    label(n) ← m
    U[m], V[m] ← APE-plan-Method(
                          s, T, n, M, d_curr + 1)
  m_opt ← arg-optimal_m{V[m]}
  return (U[m_opt], V[m_opt])
```

Figure 6: The pseudocode for APE-plan-Task

APE-plan-Method first checks whether the search has reached the maximum depth. If it has reached the maximum depth, APE-plan-Method makes an heuristic estimate of the cost and predicts the next state after going through the steps present inside the method. Otherwise, it creates a new node in the current refinement tree $T$ labeled with the first step in the method. If the step is a task, then APE-plan-Task is called for the task. If the step is a command, then APE-plan-Method calls the sub-routine APE-plan-Command.

APE-plan-Command first calls the sub-routine SampleCommandOutcomes. SampleCommandOutcomes samples $b'$ outcomes of the command *com* in the current state $s$. The sampling is done from a probability distribution specified by the domain's author. SampleCommandOutcomes re-

```
APE-plan-Method (s, T, n, M, d_curr)
  m ← label(n)
  if d_curr = d then
      s', cost' ← HeuristicEstimate(s, m)
      n', d' ← NextStep (s', T, n, d_curr)
  else:
      step ← first step in m
      n' ← new tree node
      label(n') ← step
      Add n' as a child of n
      d' ← d_curr
      cost' ← 0
      s' ← s
  case type(label(n')):
      task: T', v' ← APE-plan-Task(s', T, n', M, d')
      command: T', v' ← APE-plan-Command(
                                  s', T, n', M, d')
      end: T' ← T; v' ← 0
  return (T', v' + cost')
```

Figure 7: The pseudocode for APE-plan-Method

```
APE-plan-Command (s, T, n, M, d_curr)
  res ← SampleCommandOutcomes (s, label(n))
  value ← 0
  for (s', v, p) in res do
      n', d' ← NextStep (s', T, n, d_curr)
      case type(label(n')):
          command:
              T_s', v_s' ← APE-plan-Command(
                                  s', T, n', M, d_curr)
          task:
              T_s', v_s' ← APE-plan-Task(s', T, n', M, d_curr)
          end:
              T_s' ← T; v_s' ← 0
      value ← value + (p * (v + v_s'))
  return T, value
```

Figure 8: The pseudocode for APE-plan-Command

turns a set consisting of three tuples of the form $(s', v, p)$, where $s'$ is a predicted state after performing command *com*, and $v$ and $p$ are the cost and probabilities of reaching that state estimated from the sampling. We need the next state $s'$ to build the remaining portion of the refinement tree $T$ starting from the state $s'$. The cost $v$ contributes to the expected value of $T$ with probability $p$. Now, after getting this list of three tuples from SampleCommandOutcomes, APE-plan-Command calls the NextStep sub-routine.

NextStep (shown in Figure 9) takes as input the current refinement tree $T$ and the current node $n$ being explored. If $n$ refers to some task or command in the middle of a refinement method $m$, then NextStep creates a new node labeled with the next step inside $m$. The depth of $n_{next}$ will be same as $n$. Otherwise, if $n$ is the last step of $m$, it continues to loop and travel towards the root of the refinement tree until it finds the root or a method that has not been fully simulated yet. It returns `end` when $T$ is completely refined or a node labeled with the next step in $T$ according to $s$ and its depth.

```
NextStep (s, T, n, d_curr)
  d_next ← d_curr
  while(True)
      n_old ← n
      n ← parent(n_old) in T
      m ← label(n)
      step ← next step in m after label(n_old) depending on s
      if step is not the last step of m then
          n_next ← new tree node
          label(n_next) ← step
          break
      else
          d_next ← d_next − 1
          if d_next = 0 then
              n_next ← new tree node
              label(n_next) ← end
              break
          else
              continue
  return n_next, d_next
```

Figure 9: The sub-routine NextStep

After APE-plan-Command gets a new node $n'$ and its depth from NextStep, it calls APE-plan-Command or APE-plan-Task depending on the label of $n'$. It does this for every $s'$ in *res* and estimates a value for $T$ from these runs.

```
SampleCommandOutcomes (s, com)
  S ← φ
  Cost, Count ← empty dictionaries
  loop b' times:
      s' ← Sample(s, com)
      S ← S ∪ {s'}
      if s' in Count:
          Count[s'] ← 1
          Cost[s'] ← cost_{s,m[i]}(s')
      else:
          Count[s'] ← Count[s'] + 1
  normalize(Count)
  res ← φ
  for s' ∈ S do
      res ← res ∪ {(s', Cost[s'], Count[s'])}
  return res
```

Figure 10: The sub-routine SampleCommandOutcomes

## References


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

[Bohren et al. 2011] 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 et al. 2013] 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 et al. 2012] 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 and Ögren 2017] 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 2017] Colledanchise, M. 2017. *Behavior Trees in Robotics*. Ph.D. Dissertation, KTH, Stockholm, Sweden.

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

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

[Doherty, Kvarnström, and Heintz 2009] 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, Williams, and Hofmann 2010] 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 and Domshlak 2013] Feldman, Z., and Domshlak, C. 2013. Monte-carlo planning: Theoretically fast convergence meets practical efficiency. In *UAI*.

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

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

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

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

[Hähnel, Burgard, and Lakemeyer 1998] 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, Ragno, and Williams 2001] 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 and Ghallab 2017] Ingrand, F., and Ghallab, M. 2017. Deliberation for Autonomous Robots: A Survey. *Artificial Intelligence* 247:10–44.

[Ingrand et al. 1996] 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, Konidaris, and Rosman 2017] James, S.; Konidaris, G.; and Rosman, B. 2017. An analysis of monte carlo tree search. In *AAAI*, 3576–3582.

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

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

[Muscettola et al. 1998] 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.

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

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

[Pollack and Horty 1999] 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 and Williams 2014] Santana, P. H. R. Q. A., and Williams, B. C. 2014. Chance-constrained consistency for probabilistic temporal plan networks. In *ICAPS*.

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

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

[Teichteil-Königsbuch, Infantes, and Kuter 2008] 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 et al. 2005] 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 et al. 1991] 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 and Abramson 2001] Williams, B. C., and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*.

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

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