Advances in Cognitive Systems 7 (2019) 175-194 Submitted 6/2018; published 12/2019

APE: An Acting and Planning Engine

Sunandita Patra PATRAS@UMD.EDU
Department of Computer Science and Institute for Systems Research, University of Maryland, College Park, Maryland 20742 USA
Malik Ghallab MALIK@LAAS.FR
Centre national de la recherche scientifique (CNRS), Toulouse, 31077 France
Dana Nau NAU@CS.UMD.EDU
Department of Computer Science and Institute for Systems Research, University of Maryland, College Park, Maryland 20742 USA
Paolo Traverso TRAVERSO@FBK.EU
Fondazione Bruno Kessler (FBK), 38123 Trento, Italy

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

1. Introduction

The integration of acting and planning is a long standing AI problem discussed by many authors. For example, Pollack and Horty (1999) argue that, despite progress 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 & Ghallab, 2017). This has several drawbacks in particular for the development and consistency verification of the models. For consistency, it is highly desirable to have a single rep-

© 2019 Cognitive Systems Foundation. All rights reserved.

resentation for both acting and planning. But if this representation were a descriptive one, it would not provide sufficient functionality. Instead, the planner must be capable of reasoning directly with the actor's operational models.

Furthermore, several analyzes stress the importance of operational models. For example, Dennett (1996)'s hierarchy of natural beings refers to his third level as *Popperian creatures*, which have models to simulate their own actions before safely carrying them. Most vertebrates appear to be at least at the Popperian level. It seems clear that these simulation models are operational, since they are used for acting. However, it is unclear whether the Popperian creatures also use or need abstract descriptive models. This may even be true for human, in our everyday actions. We devote significant effort to exhibit descriptive models only for specific actions and purposes, such as planning and optimization, because our operational models are hidden to us. If a designer needs to develop operational models for artificial actors, he or she might want to use them for planning as well.

In this paper, we provide an integrated acting-and-planning system, APE (Acting and Planning Engine). The 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 can to be refined recursively. APE's acting engine is based on an expressive, general-purpose operational language.

To integrate acting and planning, APE extends the reactive PRS-like acting algorithm to include a planner, APE-plan. At each point where it must 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, the module takes samples of its possible outcomes using a predictive model of what each command will do.

We have implemented APE and APE-plan and we have done preliminary empirical assessments of them on four domains. Section 2 reviews related work, Section 3 briefly summarizes the operational model, and Section 4 describes APE and APE-plan. We present our benchmark domains and experimental results in Section 5. In Section 6, we discuss the results and provide conclusions.

2. Related Work

Acting. APE's task-refinement capabilities are similar to those in the RAE algorithm described by Chapter 3 of Ghallab et al. (2016). However, 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. RAE was based on PRS (Ingrand et al., 1996), which was also purely reactive. Some other related approaches, some of which include refinement capabilities, include work by Firby (1987), Simmons (1992), Simmons and Apfelbaum (1998), Beetz and McDermott (1994), Muscettola et al. (1998), and Myers (1999). 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 and Petri nets, e.g., Verma et al. (2005), Wang et al. (1991) have also been used as representations for acting models but do not incorporate planning. For example, in Bohren et al. (2011), the ROS execution system SMACH 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 to reasoning on goals and states, and there is no planning.

Behavior trees (Colledanchise, 2017; Colledanchise & Ögren, 2017) are a technique for acting using a hierarchical representation that is a generalization of a decision tree. In most cases the behavior tree is not generated by the system, but is provided by a human author. However, Lim et al. (2010) and Perez et al. (2011) describe some techniques have also been developed for automated learning of behavior trees. Behavior trees impose several restrictions on the structure of the tree and properties of its nodes. Our operational representation is more general and allows any valid programming construct to be the body of refinement methods.

Planning. The PropicePlan (Despouys & Ingrand, 1999) and SeRPE (Ghallab et al., 2016) planning algorithms use PRS's and RAE's refinement methods. However, instead of using command simulations, they model commands as classical planning actions. Since this assumes that the environment is deterministic, fully observable, and static, it limits the usefulness of PropicePlan and SeRPE, as most acting environments in which one would want to use PRS and RAE do not satisfy those assumptions.

Another related technique is planning with hierarchical task networks or HTNs (Nau et al., 1999), in which tasks can be refined with different methods, where a method's body is a simple collection of subtasks rather than our rich control constructs. HTN planners use classical planning actions rather than command simulations. This enables HTN planning algorithms to be significantly simpler than ours, but it also makes them unsuited for integration with an acting system such as PropicePlan, RAE, or APE.

Many papers on probabilistic planning and Monte Carlo tree search refer to simulated execution and sampling outcomes of action models (e.g., Feldman & Domshlak, 2013; Feldman & Domshlak, 2014; Kocsis & Szepesvári, 2006; James et al., 2017; Teichteil-Königsbuch et al., 2008; Yoon et al., 2007; and Yoon et al., 2008). These use a probabilistic MDP framework with a descriptive action model (generally an explicit probability distribution), without the notions of hierarchy and refinement methods. Although most of the papers refer to online planning, this activity consists of applying the planning actions within the MDP model, rather than real-world acting. There is thus no notion of integration of acting and planning, nor of how to maintain consistency between the planner's descriptive models and the actor's operational models.

Integrated Acting and Planning. Bucchiarone et al. (2013) propose a hierarchical representation framework for composing Web services. This includes abstract actions and it can interleave acting and planning, but it focuses on distributed processes, which are represented as state transition systems, and does not allow for refinement methods.

Cashmore et al. (2015) describe a system called ROSPlan that integrates planning with the ROS architecture. They model tasks in PDDL2.1 and use a classical planner to generate solution using descriptive models instead of using operational models directly for planning. Langley et al. (2017)

describe a system that integrates planning with monitoring and execution over goal-based utilities. Their approach supports durative operators with numeric effects, but it does not handle hierarchical decomposition, conditional effects, or nondeterministic outcomes.

Ingham et al. (2001)'s Reactive Model-based Programming Language (RMPL) 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 the temporal plan networks (TPNs) (Kim et al., 2001) which are an extension of simple temporal networks with symbolic constraints and decision nodes. Temporal reasoning consists of finding a path (i.e., a plan) in the TPN that meets the constraints. The execution of generated plans allows for online choices (Conrad et al., 2009). Effinger et al. (2010) extends TPNs with error recovery, temporal flexibility, and conditional execution based on the state of the world. 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 TPNs are introduced in Santana and Williams (2014) with the notions of weak and strong consistency. Levin 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 and plan repair. Our approach does not handle time; it focuses instead on decomposition into communicating asynchronous components.

3. Operational Models

Our formalism for operational models of actions is based on the one described by Chapter 3 of Ghallab et al. (2016). Its primary elements 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. Let us consider some examples.

Example 1. *Suppose several robots (UAVs and UGVs) are exploring a partially known terrain, performing operations such as data gathering, processing, screening and monitoring. 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,*
- $\mathsf{survey}(r, l)$ *be a command performed by robot* r *in location* l *that surveys* l *and collects data,*
- $\mathsf{loc}(r) \in L$ *and* $\mathsf{data}(r) \in [0, 100]$ *be observable state variables that contain the robot* r*'s current location and the amount of data it has collected.*

Let $\mathsf{explore}(r, l)$ *be a task for robot* $r \in R$ *to reach location* $l \in L$ *and perform the command* $\mathsf{survey}(r, l)$. *In order to survey, the robot needs some equipment that might either be available or*

Table 1. A refinement method for the *explore* task.

```
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
```

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). A refinement method to do this is shown in Table 1.*

Inside the refinement method m1-explore*,* get-Equipment$(r,$ *'survey'*$)$*,* moveTo(r, l) *and* depositData(r) *are subtasks which must 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 must 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*$)$*. Two of them are shown in Table 2.*

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. Let us give the robot a method for reacting to an event.

Example 2. *Suppose that 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 on Table 3. 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.*

This example illustrates tasks, events, and refinement methods.

Table 2. Two refinement methods for the task *doActivities*$(r, \textit{locList})$.

```
m1-doActivities(r, locList)                 m2-doActivities(r, locList)
   task: doActivities(r, locList)              task: doActivities(r, locList)
  body: for l in locList do:                  body: for l in locList do:
           explore(r, l)                                explore(r, l)
         moveTo(r, 'base')                              moveTo(r, 'base')
         if loc(r) = 'base':                            if loc(r) = 'base':
           recharge(r)                                    recharge(r)
         else return failure                            else return failure
         return success                              return success
```

4. APE and APE-plan

APE (Acting and Planning Engine) shown in Table 4, is based loosely on the RAE (Refinement Acting Engine) algorithm in Ghallab et al. (2016, Chapter 3). The first inner loop (line 1) reads each new *job* (a 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 τ, 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 of APE), 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 τ and the method instance m that APE has chosen to use for τ. The body of m is a program; 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 1 of Progress), following a control structure such as a loop or if-then-else (line 2 of Progress), 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 7 of Progress). A method succeeds in accomplishing a task when it returns without failure.

In line 3 of APE, line 6 of Progress, and line 2 of Retry, APE chooses a method instance m for τ. In order to make an informed choice of m, each of these lines is preceded by a call to a planner, APE-plan, that returns a plan for accomplishing τ. The returned plan, T, will begin with a method instance m to use for τ. 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 4 of Progress, where m has a subtask τ', APE does not use the method that T used for τ'. Instead, in line 5 of Progress, APE calls APE-plan to get a new plan T' for τ'. This is analogous to how a game-playing program might call an alpha-beta game-tree search at every move.[1]

1. Ghallab et al. (2016) describe 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 implemented these for APE, but in our experimental domains they did not make much difference in performance.

Table 3. A refinement method for the event *handleAlien*(r, l).

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

The pseudocode of APE-plan is a modified version of the APE pseudocode that incorporates five modifications:

1. Each call to APE-plan returns a *refinement tree* T whose root node contains a method instance m to use for τ. The children of this node include a refinement tree or terminal node for each subtask or command, respectively, that APE-plan produced during a Monte Carlo rollout of m.
2. In line 2 of APE, line 5 of Progress, and line 1 of Retry, 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 for 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.
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 3 of Progress (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 for 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.
5. Finally, APE-plan has a *search depth* d. When APE calls APE-plan, it continues planning either to completion or to 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.

Above, we have described the algorithm of our actor, APE, and our planner, APE-plan. The full pseudocode of APE-plan is given the Appendix.

Table 4. The pseudocode for APE (Acting and Planning Engine).

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

```
Retry(stack):
($\tau, m$, step, tried) ← pop(stack)
add $m$ to tried // list of method instances already tried
$s$ ← current state
$M$ ← {applicable method instances for $\tau$ in state $s$}
1 $T$ ← APE-plan($M \setminus$ tried$, s, \tau$)
if $T \neq$ failed then
2   $m'$ ← the method instance at the top of $T$
    push ($\tau, m'$, nil, tried) onto stack
else
    if stack is empty then
        output("failed to accomplish", $\tau$)
        remove stack from Agenda
    else
        Retry(stack)
```

```
Progress(stack):
($\tau, m$, step, tried) ← top(stack)   // step is the current step of $m$, i.e., if
if step ≠ nil then                      // we have started executing $m$, then step is
1   if type(step) = command then        // 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;
2   step ← next step of $m$ after accounting for the effects of control statements (loops, if-then-else, etc.)
    case type(step):
        assignment: update $s$ according to step; return;
3       command:    send step to the execution platform; return;
        task:       pass                        // continue to next line
4   $\tau'$ ← step;  $s$ ← current state ; $M'$ ← {applicable method instances for $\tau'$ in state $s$};
5   $T'$ ← APE-plan($M', s, \tau'$)
    if $T'$ = failed then Retry(stack); return;end
6   $m'$ ← the method instance at the top of $T'$
7   push ($\tau', m'$, nil, $\emptyset$) onto stack
```

5. Experimental Evaluation

In this section, we describe our test domains and experiments. Section 5.1 summarizes the properties of these domains, which include sensing, agent collaboration, dead ends, dynamic events and concurrent tasks. We present an analysis of our results in Section 5.2 in terms of three different performance metrics: efficiency, retry ratio, and success ratio.

5.1 Domains

We have implemented and tested our framework on four domains. We designed them in such a way that they model the common issues that are encountered while integrating acting and planning. Broadly, there are two groups, domains with dead ends and ones without them. A domain with dead ends means that it is possible for the agent to reach a state from which it cannot recover. Without dead ends, a purely reactive system like APE is sufficient for achieving the tasks, but not efficiently. One of our domains illustrates sensing (or information gathering) actions, three involve (centrally controlled) collaboration among actors. All domains have dynamic events and concurrent tasks (see Table 5).

The Explorable Environment domain extends the UAVs and UGVs setting of Example 1 with some additional tasks and refinement methods. The agents explore a partially known terrain and perform different operations, such as, survey, monitor, gather data or, sample soil. In order to perform a particular operation, it may need some special equipment. Robots can carry a limited amount of charge and data. This domain has dead ends because a robot may run out of charge in an isolated location.

The Chargeable Robot domain includes several robots that move 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, they use Dijkstra's shortest path algorithm. The robots do not know where objects are unless a sensing action is performed in the object's location and they must search for an object before collecting it. The robot may or may not carry the charger with it. As in Example 2, the environment is dynamic due to emergency events. A task reaches a dead end when a robot has run out of charge when it is far away from the charger.

The Spring Door domain has several robots trying to move objects from one room to another in an environment with both spring doors and ordinary doors. Spring doors close themselves unless they are held. A robot cannot carry an object and hold a door simultaneously. Thus, whenever it wants to move through a spring door, it must ask for help from another robot. Any robot that 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 involves 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, and the like arrive, carrier robots transport the necessary objects to the required machine's location. An order can be complex, such as painting two objects, assembling them, and packing 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 be damaged and need repair before being used again, but there are no dead ends.

Table 5. Properties of the four domains.

Domain	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
Chargeable Robot	✓	✓	✓	–	✓
Explorable Environment	✓	✓	–	✓	✓
Spring Door	✓	–	–	✓	✓
Industrial Plant	✓	–	–	✓	✓

These four domains have different properties, as summarized in Table 5. The Chargeable Robot domain includes a model for the sensing action where the robot can sense a location and identify objects in that location. Spring Door domain models a situation where robots need to collaborate with each other. They can ask for help from each other. The Explorable Environment 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 Spring Door domain. In the Industrial Plant 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. The Chargeable Robot domain and the Explorable Environment domain have dead ends, whereas the Spring Door domain and the Industrial Plant do not have them.

5.2 Experiments and Analysis

The objective of our experiments was to examine how APE's performance depends on the amount of planning that we told it to do. For this purpose, we created a suite of test problems, each of which included one to four jobs to accomplish, with each job inserted into APE's input stream at a randomly chosen time point. In the Chargeable Robot domain, Explorable Environment domain, Spring Door domain, and Industrial Plant domain, 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. The experiments each used simulated versions of the four environments that ran on a 2.6 GHz Intel Core i5 processor.

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 is exponential in b. As a special case, $b = 0$ means running APE in a purely reactive way with no planning at all. The objective function for the experiments was the number of commands included in the plan.

Hypothesis 1. *Increasing the value of APE's search breadth will improve its performance on three different metrics: success ratio, retry ratio, and speed to success, with greater improvement in domains with dead ends.*

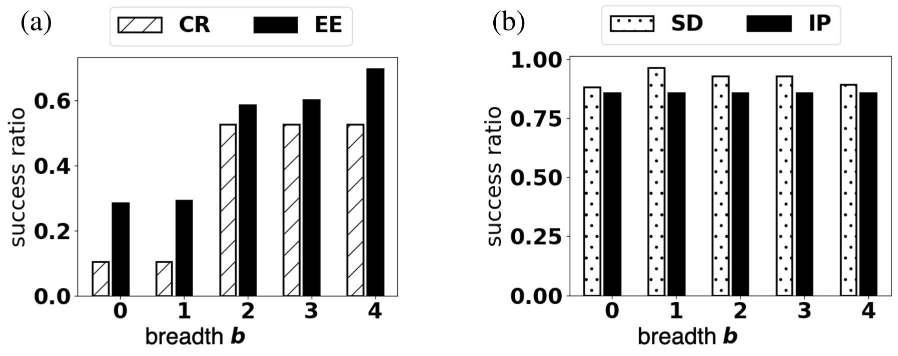

Figure 1. Success ratio (number of successful jobs /total number of jobs) for different values of search breadth b for (a) domains having dead ends (Chargeable Robot and Explorable Environment) and (b) domains having no dead ends (Spring Door and Industrial Plant). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

Success ratio. Figure 1 plots *success ratio*, the proportion of jobs that APE successfully accomplished in each domain. For the two domains with dead ends (Chargeable Robot and Explorable Environment), the ratio generally increases as the search breadth b increases. In the Chargeable Robot domain 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 tasks, the second method in the preference ordering (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 Explorable Environment domain, the success ratio continued to improve substantially for $b = 3$ and $b = 4$.

In the domains with no dead ends, the search breadth did not make much difference in the success ratio. In the Industrial Plant domain, it made almost no difference at all. In the Spring Door domain, the success ratio even decreased slightly from $b = 1$ to $b = 4$ because methods appearing earlier (in the preference ordering) are better suited to handle the events, whereas methods appearing later produce plans that are shorter but less robust to unexpected events. These experiments support the hypothesis that planning is beneficial in domains where the actor may get stuck in dead ends.

Retry ratio. Figure 2 plots results for a second measure *retry ratio*, or the number of times that APE had to call the Retry procedure divided by the total number of jobs to accomplish. Recall that the Retry procedure is called when there is a failure in the method instance m that APE chose for some task τ (see Algorithm 4). Retry works by trying to use another applicable method instance for τ that it has not tried already. Although this is similar 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 unbudgeted amounts of time and expense.

In all four domains, the retry ratio decreased slightly from $b = 0$ (purely reactive APE) to $b = 1$, and it generally decreased as b increased. This is because higher values of b made APE-plan examine

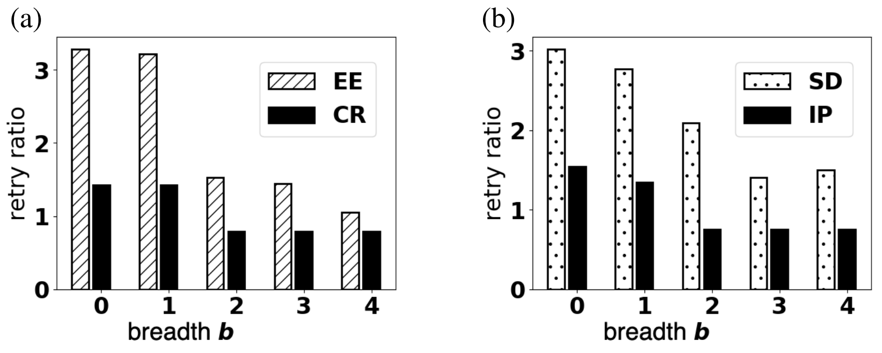

Figure 2. Retry ratio (number of retries / total number of jobs) for different values of search breadth b for (a) domains having dead ends (Chargeable Robot and Explorable Environment) and (b) domains having no dead ends (Spring Door and Industrial Plant). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

a larger number of alternative plans before choosing one, thus increasing the chance that it finds a better method for each task. In the Chargeable Robot domain, the large decrease in retry ratio from $b = 1$ to $b = 2$ corresponds to the increase in success ratio observed in Figure 1. The same is true for the Explorable Environment 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 total amount of time required for both planning and acting, which we refer to as *time to success*. 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 infinite. To average the outcomes, we use the reciprocal amount, the *speed to success*, which we define as:

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

where α is a scaling factor, 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 and we used $\alpha = 10,000$ to avoid very small numbers.

The higher the average value of ν, the better the performance. Figure 3 shows how the average value of ν depends on b. In the domains with dead ends (Chargeable Robot and Explorable Environment), there is a huge improvement in ν from $b = 1$ (where ν is nearly 0) to $b = 2$. This corresponds to more successful jobs in less time. As we increase b further, we only see slight change in ν for all the domains, even though the success ratio and retry ratio improve (Figures 1 and 2). This is because of the extra time overhead of running APE-plan with higher search breadth.

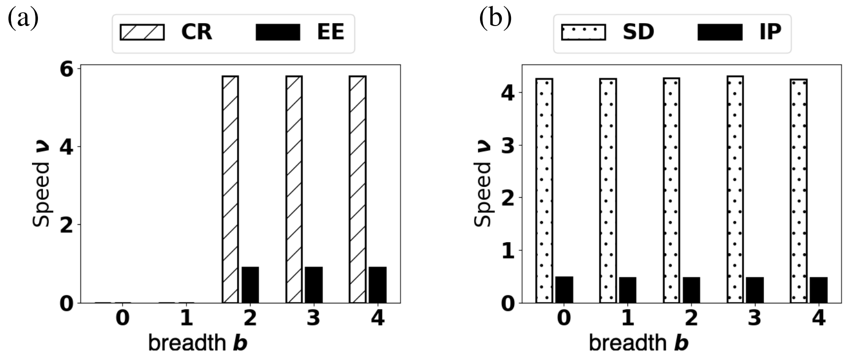

Figure 3. Speed to success ν averaged over all of the jobs, for different values of search breadth b for (a) domains having dead ends (Chargeable Robot and Explorable Environment and (b) domains having no dead ends (Spring Door domain and Industrial Plant domain). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

In summary, for domains with dead ends, planning with APE-plan outperforms the purely reactive version of APE. The same results occur in the domains without dead ends, but there the effect is less pronounced thanks to the domain specific heuristics in our experiments, which chooses good refinement methods early on.

6. Concluding Remarks

We have proposed a novel system APE that integrates acting and planning using the actor's operational models. Our experimentation address multiple aspects of realistic domains, including dynamicity and the need for run-time sensing, information gathering, collaboration, and concurrent tasks (see Table 5). We have shown the difference in performance on domains with and without dead ends on three different metrics: 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 produces benefit in domains with dead ends, which is reflected through a higher success ratio. In most cases, this measure increases with the search breadth of APE-plan. In the case of safely explorable domains, APE manages to have a similar ratio of success for all breadths.

Our second measure, the retry ratio, counts the number of retries of the same task done by APE before succeeding. Many retries is undesirable, since this has a high cost. We have shown that both in domains with dead ends and without, the retry ratio diminishes considerably with APE-plan, demonstrating its benefits even in safely explorable domains. Finally, we have devised a novel and practical way to measure the performance of APE and similar systems. While systems that both act and plan are usually evaluated only on their planning functionality, we devised a *speed to success* measure to assess the overall time to plan and act, including failures. This takes into account the fact that executing commands in the real world usually takes much longer than computation. We have shown that, in general, the integration of APE-plan reduces time significantly in the case of domains with dead ends, but we found no such decrease in performance for safely explorable domains.

Despite these encouraging results, there remains a need for additional research. When implementing APE-plan, we encountered problems involving APE-plan's *search breadth* and *sample breadth* parameters, b and b':

- The motivation for b is that there may be multiple method instances that are candidates for accomplishing a task τ. To find one that works as desired, it may be necessary to try several and b tells APE-plan how many to try. If τ occurs in the body of some method, m, then APE-plan should backtrack b times to the point in m's body where τ occurs and resume execution using a different candidate.
- The motivation for b' is that a command c may have multiple possible outcomes, which may vary probabilistically or depend on unknown conditions in the external environment, and APE may need to do different things depending on the outcome. Thus, it should sample multiple outcomes and generate plans for each one, with b' telling how many times to do so. If c appears in the body of a method instance m, then the system should backtrack b' times to the point where c occurs, sample c's outcome, and resume execution.

In both situations, it is necessary to backtrack and resume execution at a specific point in the body of a method m. That would be easy if APE-plan were an HTN planner like SHOP, because m's body would be a list of tasks that could be manipulated in the way desired. However, in APE-plan, m's body is Python code and backtracking to a specific point in m means backtracking over its execution. To make this possible, we would need to implement a modified Python interpreter running inside APE-plan. We made efforts to this end, but it turned out to be very difficult.

As a workaround for the search-breadth problem, our implementation of APE-plan tries b of the candidate method instances and chooses one that looks *locally* optimal for accomplishing τ, i.e., without regard to how this might affect what plan is needed for any subsequent tasks. Thus, although APE-plan's plan may be locally optimal, it generally will not be globally optimal. Since APE does not have a way to ensure the outcome of a command, we needed a different workaround for the sample-breadth problem. Our implementation requires $b' = 1$, i.e., it only samples one command outcome, namely the most probable one. If a command c has other less-probable outcomes, then APE-plan will not consider how to handle them.

Despite the above limitations, APE still performed well in our experiments. One reason is that, even if it sometimes makes errors, it is much better than having no planner at all. Another reason is that the system runs online. APE only uses a small part of the plan that APE-plan returns and calls it again from the next state of the world. If this state differs from that predicted, the next call will plan for the new state. Thus, APE is more resilient than it would be if APE-plan were operating offline. On the other hand, it is possible to produce better plans than the ones APE-plan produces, and these difficulties have prompted us to rethink how planning should occur in an environment where outcomes may be uncertain, information may be imperfect or incomplete, and the situation may change dynamically. We have designed a new system that does not have the deficiencies described above and we have begun implementation and testing. Our work-in-progress on that algorithm is presented in Patra et al. (2019).

Acknowledgements

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

References

Beetz, M., & McDermott, D. (1994). Improving robot plans during their execution. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems* (pp. 7–12). Chicago, IL: AAAI.

Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mösenlechner, L., Meeussen, W., & Holzer, S. (2011). Towards autonomous robotic butlers: Lessons learned with the PR2. *2011 IEEE International Conference on Robotics and Automation* (pp. 5568–5575). Shanghai, China: IEEE.

Bucchiarone, A., Marconi, A., Pistore, M., Traverso, P., Bertoli, P., & Kazhamiakin, R. (2013). Domain objects for continuous context-aware adaptation of service-based systems. *2013 IEEE International Conference on Web Services* (pp. 571–578). Santa Clara, CA: IEEE.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtos, N., & Carreras, M. (2015). ROSPlan: Planning in the robot operating system. *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling* (pp. 333–341). Jerusalem, Israel: AAAI Press.

Colledanchise, M. (2017). *Behavior trees in robotics*. Doctoral dissertation, Department of Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden.

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

Conrad, P., Shah, J., & Williams, B. C. (2009). Flexible execution of plans with choice. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling* (pp. 82–90). Thessaloniki, Greece: AAAI.

Dennett, D. (1996). *Kind of minds*. New York, NY: Perseus Books.

Despouys, O., & Ingrand, F. (1999). Propice-Plan: Toward a unified framework for planning and execution. *Recent Advances in AI Planning, 5th European Conference on Planning* (pp. 278–293). Durham, UK: Springer.

Effinger, R., Williams, B., & Hofmann, A. (2010). Dynamic execution of temporally and spatially flexible reactive programs. *Papers from the 2010 AAAI Workshop on Bridging the Gap Between Task and Motion Planning* (pp. 18–25). Atlanta, GA: AAAI Press.

Feldman, Z., & Domshlak, C. (2013). Monte-Carlo planning: Theoretically fast convergence meets practical efficiency. *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence* (pp. 212–220). Bellevue, WA: AUAI Press.

Feldman, Z., & Domshlak, C. (2014). Monte-Carlo tree search: To MC or to DP? *Proceedings of the Twenty-First European Conference on Artificial Intelligence* (pp. 321–326). Prague, Czech Republic: IOS Press.

Firby, R. J. (1987). An investigation into reactive planning in complex domains. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 202–206). Seattle, WA: Morgan Kaufmann.

Ghallab, M., Nau, D. S., & Traverso, P. (2016). *Automated planning and acting*. Cambridge, UK: Cambridge University Press.

Ingham, M. D., Ragno, R. J., & Williams, B. C. (2001). A reactive model-based programming language for robotic space explorers. *Proceedings of i-SAIRIS 2001: The Sixth International Symposium on Artificial Intelligence, Robotics, and Automation in Space* (pp. 487–493). Montreal, Canada.

Ingrand, F., Chatilla, R., Alami, R., & Robert, F. (1996). PRS: A high level supervision and control language for autonomous mobile robots. *Proceedings of the 1996 IEEE International Conference on Robotics and Automation* (pp. 43–49). Minneapolis, MN: IEEE.

Ingrand, F., & Ghallab, M. (2017). Deliberation for autonomous robots: A survey. *Artificial Intelligence*, *247*, 10–44.

James, S., Konidaris, G., & Rosman, B. (2017). An analysis of Monte Carlo tree search. *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (pp. 3576–3582). San Francisco, CA.

Kim, P., Williams, B. C., & Abramson, M. (2001). Executing reactive, model-based programs through graph-based temporal planning. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (pp. 487–493). Seattle, WA: Morgan Kaufmann.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *Proceedings of the Seventeenth European Conference on Machine Learning* (pp. 282–293). Berlin, Germany.

Langley, P., Choi, D., Barley, M., Meadows, B., & P. Katz, E. (2017). Generating, executing, and monitoring plans with goal-based utilities in continuous domains. *Proceedings of the Fifth Conference on Advances in Cognitive Systems* (pp. 1–12). Troy, NY.

Levine, S. J., & Williams, B. C. (2014). Concurrent plan recognition and execution for human-robot teams. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling* (pp. 490–498). Portsmouth, NH.

Lim, C.-U., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game DEFCON. *Applications of Evolutionary Computation, EvoApplications 2010* (pp. 100–110). Istanbul, Turkey: Springer.

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

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

Nau, D. S., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 968–975). Stockholm, Sweden.

Patra, S., Traverso, P., Ghallab, M., & Nau, D. (2019). Acting and planning using operational models. *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence* (pp. 7691–7698). Honolulu, HI: AAAI Press.

Perez, D., Nicolau, M., O'Neill, M., & Brabazon, A. (2011). Evolving behaviour trees for the Mario AI competition using grammatical evolution. *Applications of Evolutionary Computation - EvoApplications 2011* (pp. 123–132). Torino, Italy: Springer.

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

Santana, P. H. R. Q. A., & Williams, B. C. (2014). Chance-constrained consistency for probabilistic temporal plan networks. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling* (pp. 271–279). Portsmouth, NH.

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

Simmons, R., & Apfelbaum, D. (1998). A task description language for robot control. *Proceedings 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1931–1937). Victoria, BC, Canada.

Teichteil-Königsbuch, F., Infantes, G., & Kuter, U. (2008). RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. *The Uncertainty Part of the Sixth International Planning Competition 2008*. Sydney, Australia. From `http://ippc-2008.loria.fr/wiki/index.php/Results.html`.

Verma, V., Estlin, T., Jónsson, A. K., Pasareanu, C., Simmons, R., & Tso, K. (2005). Plan execution interchange language (PLEXIL) for executable plans and command sequences. *Proceedings of i-SAIRIS 2005: The Eighth International Symposium on Artificial Intelligence, Robotics and Automation in Space* (pp. 1–8). Munich, Germany.

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

Yoon, S. W., Fern, A., & Givan, R. (2007). FF-Replan: A baseline for probabilistic planning. *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling* (pp. 352–359). Providence, RI.

Yoon, S. W., Fern, A., Givan, R., & Kambhampati, S. (2008). Probabilistic planning via determinization in hindsight. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence* (pp. 1010–1016). Chicago, IL: AAAI Press.

Table 6. The pseudocode of APE-plan and APE-plan-task, a subroutine of APE-plan, which is the planner used by APE.

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

```
APE-plan-task $(s, T, n, M, d_{curr})$:
$\tau \leftarrow label(n)$
$B \leftarrow$ { Applicable method instances for
  $\tau$ in $M$ ordered according to a
  preference ordering }
if $|B| < b$ then
    $B' \leftarrow B$
else
    $B' \leftarrow B[1...b]$
    $U, V \leftarrow$ empty dictionaries
for each $m \in B'$ do
    $label(n) \leftarrow m$
    $U[m], V[m] \leftarrow$
      APE-plan-method$(s, T, n, M, d_{curr}+$
      $1)$
    $m_{opt} \leftarrow$ arg-optimal$_m\{V[m]\}$
return $(U[m_{opt}], V[m_{opt}])$
```

7. Appendix: Description of APE-plan

The main procedure of APE-plan is shown in Table 6. The parameters b, b' and d are global variables and denote the search breadth, sample breadth, and search depth, respectively. The system receives as input a task τ to be addressed, a set of methods M, and a current state s, for which it returns a refinement tree T for τ. It starts by creating a refinement tree with a single node n labeled τ and calls a sub routine APE-plan-task, which builds a complete refinement tree for n.

The APE-plan-command subroutine first calls SampleCommandOutcomes, which samples b' outcomes of the command *com* in the current state s. Samples are taken from a probability distribution specified by the domain's designer.The module returns a set consisting of three tuples of the form (s', v, p), where s' is a predicted state after performing command *com*, and where v and p are the cost and probabilities of reaching that state estimated from 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. After getting this list of three tuples from SampleCommandOutcomes, APE-plan-command calls on NextStep.

Table 7. The pseudocode for APE-plan-method. *pt = APE-plan-task, pc = APE-plan-command.

```
APE-plan-method $(s, T, n, M, d_{curr})$:
$m \leftarrow label(n)$
if $d_{curr} = d$ then
    $s', cost' \leftarrow$ HeuristicEstimate$(s, m)$
    $n', d' \leftarrow$ NextStep $(s', T, n, d_{curr})$
else
    $step \leftarrow$ first step in $m$
    $n' \leftarrow$ new tree node;
     $label(n') \leftarrow step$
    Add $n'$ as a child of $n$
    $d' \leftarrow d_{curr}$; $cost' \leftarrow 0$; $s' \leftarrow s$
case type$(label(n'))$:
   task: $T', v' \leftarrow$ pt*$(s', T, n', M, d')$
   command:
 $T', v' \leftarrow$ pc*$(s', T, n', M, d')$
   end: $T' \leftarrow T; v' \leftarrow 0$
return $(T', v' + cost')$
```

```
APE-plan-command $(s, T, n, M, d_{curr})$:
$c \leftarrow label(n)$
$res \leftarrow$ SampleCommandOutcomes $(s, c)$
$value \leftarrow 0$
for $(s', v, p)$ in $res$ do
    $n', d' \leftarrow$ NextStep $(s', T, n, d_{curr})$
    case type$(label(n'))$:
       task: $T_{s'}, v_{s'} \leftarrow$ pt*$(s', T, n', M, d_{curr})$
       command:
     $T_{s'}, v_{s'} \leftarrow$ pc*$(s', T, n', M, d_{curr})$
       end: $T_{s'} \leftarrow T; v_{s'} \leftarrow 0$
     $value \leftarrow value + (p * (v + v_{s'}))$
return $T, value$
```

```
NextStep $(s, T, n, d_{curr})$:
$d_{next} \leftarrow d_{curr}$
while True do
    $n_{old} \leftarrow n$
    $n \leftarrow parent(n_{old})$ in $T$
    $m \leftarrow label(n)$
    $step \leftarrow$ next step in $m$ after
     $label(n_{old})$ depending on $s$
    if step is not the last step of $m$ then
        $n_{next} \leftarrow$ new tree node
        $label(n_{next}) \leftarrow step$; break
    else
        $d_{next} \leftarrow d_{next} - 1$
        if $d_{next} = 0$ then
            $n_{next} \leftarrow$ new tree node
            $label(n_{next}) \leftarrow$ end; break
        else
            continue
return $n_{next}, d_{next}$
```

```
SampleCommandOutcomes $(s, com)$:
$S \leftarrow \phi$
$Cost, Count \leftarrow$ empty dictionaries
repeat
    $s' \leftarrow$ Sample$(s, com)$
    $S \leftarrow S \cup \{s'\}$
    if $s'$ in Count then
        $Count[s'] \leftarrow 1$
        $Cost[s'] \leftarrow cost_{s,m[i]}(s')$
    else
        $Count[s'] \leftarrow Count[s'] + 1$
until $b'$ samples are taken
normalize(Count)
$res \leftarrow \phi$
for $s' \in S$ do
    $res \leftarrow$
      $res \cup \{(s', Cost[s'], Count[s'])\}$
return $res$
```

The NextStep subroutine shown in Figure 7 takes as input the current refinement tree T and 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 of m, the depth of n_{next} being the 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. The function returns `end` when T is completely refined or a node labeled with the next step in T. The label depends on the current state s and the depth of T. After APE-plan-command gets a new node n' and its depth from NextStep, it calls either APE-plan-command or APE-plan-task, depending on the label of n'. The routine does this for every s' in *res* and estimates a value for T from these runs.