ABSTRACT

Title of Dissertation:	ACTING, PLANNING, AND LEARNING, USING HIERARCHICAL OPERATIONAL MODELS
	Sunandita Patra Doctor of Philosophy, 2020
Dissertation Directed by:	Professor Dana Nau Department of Computer Science

The most common representation formalisms for planning are *descriptive models* that abstractly describe *what* the actions do and are tailored for efficiently computing the next state(s) in a state-transition system. However, real-world acting requires *operational models* that describe *how* to do things, with rich control structures for closed-loop online decision-making in a dynamic environment. Use of a different action model for planning than the one used for acting causes problems with combining acting and planning, in particular for the development and consistency verification of the different models.

As an alternative, this dissertation defines and implements an integrated acting-and-planning system in which both planning and acting use the same operational models, which are written in a general-purpose hierarchical task-oriented language offering rich control structures. The acting component, called Reactive Acting Engine (RAE), is inspired by the well-known PRS system, except that instead of being purely reactive, it can get advice from a planner. The dissertation also describes three planning algorithms which plan by doing several Monte Carlo rollouts in the space of operational models. The best of these three planners, Plan-with-UPOM uses a UCT-like Monte Carlo Tree Search procedure called UPOM (*UCT Procedure for Operational Models*), whose rollouts are simulated executions of the actor's operational models. The dissertation also presents learning strategies for use with RAE and UPOM that acquire from online acting experiences and/or simulated planning results, a mapping from decision contexts to method instances as well as a heuristic function to guide UPOM. The experimental results show that Plan-with-UPOM and the learning strategies significantly improve the acting efficiency and robustness of RAE. It can be proved that UPOM converges asymptotically by mapping its search space to an MDP. The dissertation also describes a real-world prototype of RAE and Plan-with-UPOM to defend software-defined networks, a relatively new network management architecture, against incoming attacks.

Acting, Planning, and Learning Using Hierarchical Operational Models

by

Sunandita Patra

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Dedication

To all the teachers who have taught me.

Acknowledgments

I would like to take this opportunity to thank everyone who have made this dissertation possible and because of whom my graduate experience has been extraordinary.

First and foremost I'd like to thank my advisor, Professor Dana Nau for giving me an invaluable opportunity to work on extremely interesting and challenging projects over the past five years. He has always made himself available for help and advice. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank my collaborators, Dr. Malik Ghallab, and Paolo Traverso. Without their amazing research ideas, expertise, and guidance this dissertation would have been a distant dream. Thanks are due to Professor Jordan Boyd-Graber, Professor Tom Goldstein and Professor Pamela Abshire for agreeing to serve on my dissertation committee and for sparing their invaluable time reviewing the manuscript.

My collaborators at The Naval Research Labs have enriched my graduate life in many ways and deserve a special mention. Dr. Myong Kang, Dr. Anya Kim and Alex Velazquez have helped me develop a real-world application of the algorithms presented in the dissertation. I would also like to acknowledge financial support that I received from their research grant.

My interactions with other students, mainly Amit Kumar, James Mason, Ruoxi Li, and Soham De have been very productive and encouraging.

Thanks are due to the extremely friendly and efficient Staff members at the Department of Computer Science, University of Maryland, who were always available to help me with administrative queries and procedures.

I owe my deepest thanks to my family - my mother and father who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them.

My housemates and friends have been a crucial factor in my finishing smoothly. I'd like to express my gratitude to all of them for their friendship and support.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents v

List of Tables viii

List of Figures ix

List of Abbreviations xii

Chapter 1: Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions of the Dissertation . . . 4
- 1.3 Thesis Organization . . . 6

Chapter 2: Related Work 8
- 2.1 Acting Systems . . . 8
- 2.2 Systems that integrate acting and planning . . . 9
- 2.3 Planning Algorithms . . . 13
- 2.4 Planning and Learning . . . 14
- 2.5 Hierarchical Reinforcement Learning . . . 14

Chapter 3: Acting and Planning Algorithms 16
- 3.1 Formalism: Hierarchical Operational Models . . . 16
- 3.2 The actor, RAE . . . 30
- 3.3 Planner, APEplan . . . 35
- 3.4 Planner, RAEplan . . . 38
  - 3.4.1 Properties of RAEplan . . . 42
- 3.5 UPOM, a UCT-like search procedure . . . 48
  - 3.5.1 Utility criteria and optimal approach . . . 50
  - 3.5.2 A planning algorithm based on UCT . . . 54
- 3.6 Learning for RAE and UPOM . . . 62
  - 3.6.1 Learning to choose methods (Learnπ) . . . 63
  - 3.6.2 Learning to choose method instances (Learnπ_i) . . . 65
  - 3.6.3 Learning a heuristic evaluation function (LearnH) . . . 67

3.6.4 Incremental online learning 69
3.7 Properties of Plan-with-UPOM 69
3.7.1 Mapping UPOM's Search Space to an MDP 71
3.7.2 Search Space for Refinement Planning 72
3.8 Summary 81

Chapter 4: Implementation and Experimental Evaluation 82
4.1 Evaluation of APEplan 82
4.1.1 Domains 82
4.1.2 Assessment of APEplan's parameters 85
4.2 Evaluation of RAEplan 90
4.2.1 Domains 90
4.2.2 Assessment of RAEplan's parameters 92
4.3 Evaluation of Plan-with-UPOM 98
4.3.1 Domains 98
4.3.2 Assessment of UPOM's parameters 99
4.3.3 Comparison of the two utility functions 101
4.3.4 Retry ratio 102
4.3.5 Efficiency across domains 102
4.3.6 Success ratio across domains 103
4.3.7 Depth and Heuristics 105
4.3.8 Measured *vs* expected efficiency 107
4.4 Comparison of RAEplan and UPOM 109
4.5 Assessment of UPOM's learning strategies 111
4.5.1 Learning Method Instances 115
4.6 Summary 116

Chapter 5: Real-world prototype of RAE and Plan-with-UPOM: Defense against SDN attacks 118
5.1 Software Defined Networks (SDNs) 118
5.2 AIRMAN Architecture 120
5.3 Attack recovery using RAE and Plan-with-UPOM 121
5.3.1 Integration of AIRMAN SecurityManager and RAE 126
5.3.2 State Definition for SDN 126
5.3.3 Utility function optimized: CostEffectiveness 128
5.3.4 Communication between SecurityManager and RAE 130
5.3.5 Example of task invocation workflow 132
5.3.6 Domain Definition for SDN 134
5.3.7 Environment for SDN 134
5.3.8 Action Model 135
5.4 Experimental Evaluation 136
5.5 Summary 139

Chapter 6: Conclusion 140
6.1 Looking Ahead: Limitations and Future Directions 142

6.1.1 Retrial in RAE 143
6.1.2 Planning for multiple tasks at once 144
6.1.3 Concurrency 144
6.1.4 Learning operational models 145
6.1.5 Benchmarking 146

Appendix A: Description of APEplan 149

Appendix B: Descriptions of Experimental Domains 154
B.1 Fetch domain 154
B.2 Explore domain 162
B.3 Navigate domain 177
B.4 Rescue domain 186
B.5 Delivery domain 197
B.6 Attack recovery in Software-defined networks (SDN) 209

Bibliography 231

List of Tables

4.1 Properties of the four test domains of APEplan. 85
4.2 Features of the five test domains of RAE + UPOM 99
4.3 The size of the training set, number of input features and outputs, and the number of training epochs for three different learning procedures: Learnπ-1, Learnπ-2, and LearnH. We note LM-1 = Learnπ-1, LM-2 = Learnπ-2, and LH = LearnH. 113
4.4 The size of the training set, number of input features and outputs for learning method parameters in Learnπ_i. 115

A.1 The pseudocode of APEplan and APE-plan-task, a sub-routine of APE-plan. APEplan is the planner used by RAE. 150
A.2 The pseudocode for APE-plan-method. *pt = APE-plan-task, pc = APE-plan-command. 151
A.3 The pseudocode for NextStep and SampleCommandOutcomes. *pt = APE-plan-task, pc = APE-plan-command. 152

List of Figures

1.1 A schematic diagram showing the interaction between the actor and planner and how they usually interact in general. 3

3.1 (a) Architecture of an actor reacting to events and tasks through an execution platform; (b) Integration of refinement acting, planning and learning. 17
3.2 A possible refinement tree for the task explore(r, l) corresponding to a successfully accomplished task. 23
3.3 A refinement tree for the task explore(r, l) where r failed to reach location l. 23
3.4 A refinement tree for the task explore(r, l) where r surveyed location l successfully but does not need to deposit the data. 23
3.5 A refinement tree for the task explore(r, l) where m1-getTool(r) is used to refine the task getTool(r). 25
3.6 A refinement tree for the task explore(r, l) where m2-getTool(r) is used to refine the task getTool(r). 25
3.7 A refinement tree for the task explore(r, l) where the move subtasks are refined. 26
3.8 A refinement tree, with three types of nodes: *disjunction* for a task over possible methods, *sequence* for a method over all its steps, and *sampling* for an action over its possible outcomes. A rollout can be, for example, the sequence of nodes marked 1 (a sample of a_1), 2 (first step of m_1), $\ldots, j$ (subsequent refinements), $j + 1$ (next step of m_1), $\ldots, n$ (a sample of a_2), $n + 1$ (first step of m_2), etc. 37
3.9 The sub-task getTool(r) can be refined using two different refinement methods leading to two possible refinement trees for the task explore(r, l). 58
3.10 A table informally showing how the Q-values and N counts are updated for every task/subtask after each rollout (a call to UPOM) for the search tree shown in Figure 3.9. 59
3.11 A schematic diagram for the Learnπ procedure. 64
3.12 A schematic diagram for the LearnH procedure. 67

4.1 Success ratio (number of successful jobs/ total number of jobs) for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) 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. . . 87
4.2 Retry ratio (number of retries / total number of jobs) for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) 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. . . 88
4.3 Speed to success ν averaged over all of the jobs, for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) 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. 90
4.4 Efficiency E averaged over all of the jobs, for various values of b and k of RAEplan in domains with dead ends. 94
4.5 Efficiency E averaged over all of the jobs, for various values of b and k of RAEplan in domains *without* dead ends. 94
4.6 Success ratio (# of successful jobs/ total # of jobs) for various values of b and k of RAEplan in domains with dead ends. 95
4.7 Success ratio (# of successful jobs/ total # of jobs) for various values of b and k of RAEplan in domains *without* dead ends. 95
4.8 Retry ratio (# of retries / total # of jobs) for various values of b and k of RAEplan in domains with dead ends. 97
4.9 Retry ratio (# of retries / total # of jobs) for various values of b and k of RAEplan in domains *without* dead ends. 97
4.10 Efficiency and success ratio for two different utility functions of Plan-with-UPOM (orange is expected success ratio and gray is expected efficiency) averaged over all five domains, with $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 101
4.11 Retry ratio (# of retries / total # of jobs) averaged over all five domains, for Plan-with-UPOM with $d_{max} = \infty$. 103
4.12 Measured efficiency of RAE with Plan-with-UPOM for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 104
4.13 Measured efficiency averaged over only test cases with one root task, in Fetch and Deliver domains with Plan-with-UPOM's parameter, $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). . . 105
4.14 Measured success ratio (# of successful jobs/ total # of jobs) for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 106

4.15 Measured success ratio averaged over only test cases with one root task, in Fetch and Deliver domains with $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 107
4.16 Measured efficiency with limited depth and three different heuristic functions. The utility function optimized is expected efficiency (relative values with respect to the base case of U for $n_{ro} = 0$). 108
4.17 Absolute difference between measured and expected efficiency, as a function of the refinement deepness (0 is the root task), for various number of rollouts. 109
4.18 Average computation time in seconds for a single run of a problem, for RAE with and without the planners. 110
4.19 Measured efficiency for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 110
4.20 Measured success ratio for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$). 112
4.21 Training and validation results for Learnπ and LearnH, averaged over all domains. 113
4.22 The cross hatched blue bars show the performance of RAE with Learnπ_i (learning method instances) for the two domains, Nav and Deliver, which have methods with parameters not in tasks (relative values with respect to the base case of U for $n_{ro} = 0$). 116

5.1 AIRS Architecture for SDN 119
5.2 AIRMAN architecture 120
5.3 A refinement tree for the event packetIn-flooding(id) using the refinement method m1_ctrl_clearstate_besteffort(id) 124
5.4 A refinement tree for the event packetIn-flooding(id) using the refinement method m2_ctrl_clearstate_fallback(id) 125
5.5 A partial refinement tree for the event packetIn-flooding(id) using the refinement method m3_ctrl_mitigate_pktinflood(id). fix-switch(s_1) is a sub-task that should further be refined. 126
5.6 Estimated time for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM. 137
5.7 Efficiency (reciprocal of estimated cost) for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM. 138
5.8 Retry ratio for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM. 139

List of Abbreviations

RAE	Refinement Acting Engine
PRS	Procedural Reasoning System
APEplan	Acting-and-Planning-Engine Planner
RAEplan	Refinement-Acting-Engine Planner
UCT	Upper Confidence Bound on Trees
UPOM	UCT Planner for Operational Models
RMPL	Reactive Model Based Planner
BT	Behavior Trees
HTN	Hierarchical Task Network
HPN	Hierarchical Planning in the Now
MDP	Markov Decision Process
SeRPE	Sequential Refinement Planning Engine
SDN	Software-Defined Networking
AIRS	Autonomous Intelligent Resilient Security
AIRMAN	AIRS Management System
PDDL	Planning Domain Definition Language
PPDDL	Probabilistic Planning Domain Definition Language
STRIPS	Standford Research Institute Problem Solver

Chapter 1: Introduction

1.1 Motivation

Numerous knowledge representations have been proposed for describing and reasoning about actions. However, for the purpose of AI planning, the dominant representation is the one inherited from the early STRIPS system and formalized in various versions of the PDDL description language, and representations for planning under uncertainty, such as PPDDL. Such *descriptive models* of actions are tailored to efficiently compute the next states in a state transition system. This works well for classical planning algorithms that assume a static world and no concurrency or uncertainty. However, for acting in the real world, the classical planning assumptions tend to be very restrictive and are almost always violated. The planning capabilities, that the descriptive models (which *describe what* the actions do) provide to a planner in dynamic real-world scenarios, are quite limited. In particular,

- It cannot reason about ongoing activities, for e.g., an agent might want to act differently depending on whether a particular command is currently running or not. Descriptive models generally assume that actions always happen instantaneously.

- It cannot react and adapt to an unfolding context or exogenous events happening in the environment. Consider the following scenario: today is Friday and you need to prepare for Monday's lecture sometime in the weekend. You decide to do it on Sunday morning. Meanwhile, on Saturday morning, your friends called you asking you to join a game on Sunday evening. You are not sure whether you will be done preparing for your lecture by Sunday morning, and you also don't want to promise your friend and then back out.

- It provides very little help for plan management and makes it complicated. For an actor, just coming up with plans is not enough. It also needs to monitor that the plan is being executed correctly and re-plan whenever required. If the action models that are executed are different from the ones used for planning, plan monitoring can also become challenging.

As argued above and by many authors, e.g., [1], plans are needed for acting deliberately, but they are not sufficient for realistic applications. Acting requires *operational models* that describe *how* to do things, with rich control structures for closed-loop online decision-making. Acting also requires some mechanism for plan management.

Most approaches for the integration of planning and acting seek to combine descriptive representations for the planner and operational representations for the actor [2]. A schematic diagram showing this approach is shown in Figure 1.1. However, this decomposition has several drawbacks. First, it fails to take into account the highly interconnected reasoning that is required between planning and delibera-

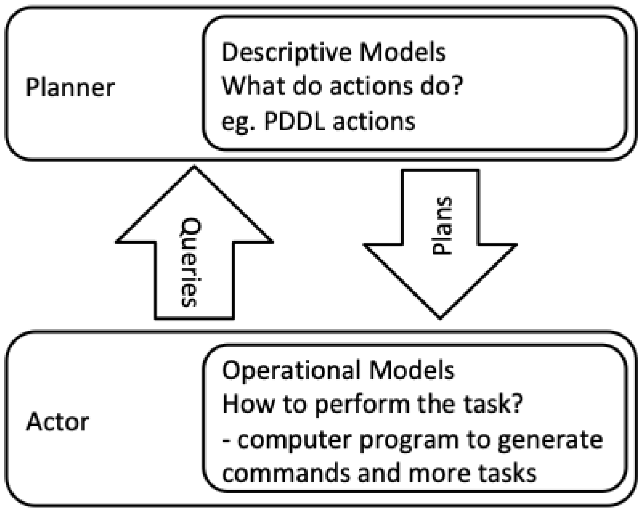


Figure 1.1: A schematic diagram showing the interaction between the actor and planner and how they usually interact in general.

tive acting in most practical settings. Second, in several applications, the mapping between descriptive and operational models is complex. A guarantee of the consistency of this mapping is required in safety-critical applications, such as self-driving cars [3], collaborative robots working directly with humans [4], or virtual coaching systems to help patients with chronic diseases [5]. However, to verify the consistency between the two different models is usually difficult (e.g., see the work on formal verification of operational models such as PRS-like procedures, using model checking and theorem proving [6, 7]). Finally, modeling is always a costly bottleneck; reducing the corresponding efforts is beneficial in most applications.

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

1.2 Contributions of the Dissertation

The author has developed integrated planning and acting algorithms in which both planning and acting use the actor's operational models. To her knowledge, no previous approach has proposed the integration of planning and acting directly within the language of an operational model.

The acting component used in this work is RAE, taken from [8, Chapter 3], which, in turn, is inspired by the well-known PRS system [9]. It uses a hierarchical task-oriented operational representation with an expressive, general-purpose language offering rich control structures for closed-loop online decision-making. A collection of refinement methods describes alternative ways to handle *tasks* and react to *events*. Each method has a *body* that can be any complex algorithm. In addition to the usual programming constructs, the body may contain *subtasks*, which need to be refined recursively, and sensory-motor *commands*, which query and change the world non-deterministically. Commands are simulated when planning and performed by an execution platform in the real world when acting.

The actor, RAE can perform multiple tasks in parallel. Rather than behaving purely reactively like PRS, RAE can interact with a planner. To decide how to refine tasks or events, the author has developed three refinement planning algorithms, APEplan, RAEplan and UPOM, which do three different kinds of Monte Carlo rollouts. When a refinement method contains a command, the planners take samples of its possible outcomes, using either a domain-dependent generative simulator, when available, or a probability distribution of its possible outcomes.

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

The author has implemented and evaluated the approach described above, and the results show significant benefits in five simulated domains. We have developed three new metrics to measure the performance of systems that integrate planning and acting: efficiency, retry-ratio and success-ratio.

In addition the dissertation includes a proof of UPOM's asymptotic convergence to optimal choices.

In summary, the contributions of this thesis are the following:

- Acting and planning algorithms where both the actor and the planner use operational models of actions. Our actor is called RAE and the planners are called APEplan, RAEplan and UPOM.
- Learning strategies to integrate UPOM with learning; learn the "best" refinement method instances for tasks and learn a heuristic evaluation function to guide UPOM's search.
- Implementation and experimental evaluation of our algorithms. We have implemented and tested it on five simulated domains.
- A comprehensive way to evaluate the performance of our approach or any other algorithm that integrates acting and planning using three performance metrics: efficiency, retry ratio and success ratio. The algorithms performed well in the evaluation process.
- A real-world prototype of RAE and UPOM to defend software-defined networks against incoming attacks.

1.3 Thesis Organization

In Chapter 2, the related work is discussed mainly with respect to five areas: acting algorithms, planning algorithms, algorithms that integrate acting and planning, systems integrating planning and learning, and hierarchical reinforcement learning approaches. In Chapter 3, the hierarchical operational model representation is described with some examples. The acting algorithm, RAE and planning algorithms, APEplan, RAEplan and UPOM are also presented with the learning strategies. The convergence of UPOM to an MDP is proved. In Chapter 4, our implemention of the algorithms on five simulated domains is discussed and the experimental results are presented. In Chapter 5, a real-world prototype of RAE and UPOM is described. Chapter 6 concludes the dissertation.

Chapter 2: Related Work

This chapter discusses the different areas of work related to the problem of integrating acting, planning, and learning. There has been work in developing purely reactive acting systems, without any planning capabilities. Some works integrate acting with operational models and planning with descriptive models. Our refinement planning algorithms are in some ways similar to hierarchical task network planners and Monte Carlo Tree Search, because they take into account hierarchy in the task network and does several Monte Carlo rollouts respectively. Finally, since we develop learning strategies for refinement planning, this entails comparison with approaches that integrate planning and learning, and hierarchical reinforcement learning. In this chapter, all of the above are discussed in details.

2.1 Acting Systems

Our acting algorithm and operational models are based on the RAE algorithm [8, Chapter 3], which in turn is based on PRS. If RAE and PRS need to choose among several eligible refinement methods for a given task or event, they make the choice without trying to plan ahead. This approach has been extended with some planning capabilities in PropicePlan [10] and SeRPE [8]. Unlike our approach, those systems model commands as classical planning operators; they both require the action models and the refinement methods to satisfy classical planning assumptions of deterministic, fully observable and static environments, which are not acceptable assumptions for most acting systems. This makes the planning limited in many aspects such as, handling other agents, an unfolding context or exogenous events.

Various acting approaches similar to PRS and RAE have been proposed, e.g., [11, 12, 13, 14, 15, 16]. Some of these have refinement capabilities and hierarchical models, e.g., [17, 18, 19]. While such systems offer expressive acting environments, e.g., with real time handling primitives, none of them provide the ability to plan with the operational models used for acting, and thus cannot integrate acting and planning as we do. Most of these systems do not reason about alternative refinements.

2.2 Systems that integrate acting and planning

[20, 21, 22] propose a way to do online planning and acting, but their notion of "online" is different from ours. In [20], the old plan is executed repeatedly in a loop while the planner synthesizes a new plan (which the authors say can take a large amount of time), and the new plan isn't installed until planning has been finished. In our planning algorithms, hierarchical task refinement is used to do the planning quickly, and RAE waits until the planner returns.

RMPL. The Reactive Model-based Programming Language (RMPL) [23] is a comprehensive CSP-based approach for temporal planning and acting which combines a system model with a control model. The system model specifies nominal as well as failure state transitions with hierarchical constraints. The control model uses standard reactive programming constructs. RMPL programs are transformed into an extension of Simple Temporal Networks with symbolic constraints and decision nodes [24, 25]. Planning consists in finding a path in the network that meets the constraints. RMPL has been extended with error recovery, temporal flexibility, and conditional execution based on the state of the world [26]. Probabilistic RMPL are introduced in [27, 28] with the notions of weak and strong consistency, as well as uncertainty for contingent decisions taken by the environment or another agent. The acting system adapts the execution to observations and predictions based on the plan. RMPL and subsequent developments have been illustrated with a service robot which observes and assists a human. Our approach does not handle time; it focuses instead on hierarchical decomposition with Monte Carlo rollout and sampling.

Behavior trees. Behavior trees (BT) [29, 30] can also respond reactively to contingent events that were not predicted. Planning synthesizes a BT that has a desired behavior. Building the tree refines the acting process by mapping the descriptive action model onto an operational model. Our approach is different since RAE provides the rich and general control constructs of a programming language and plans directly within the operational model, not by mapping from the descriptive to an operational model. Moreover, the BT approach does not allow for refinement methods, which are a rather natural and practical way to specify different possible refinements of tasks.

Robotics. There has been a lot of work in robotics to integrate planning and execution. They propose various techniques and strategies to handle the inconsistency issues that arise when execution and planning are done with different models. [31] shows how HTN planning can be used in robotics. [32] and [33] integrates task and motion planning for robotics. The approach of [34] addresses a problem similar to ours but specific to robot navigation. Several methods for performing a navigation task and its subtasks are available, each with strong and weak points depending on the context. The problem of choosing a best method instance for starting or pursuing a task in a given context is stated as a receding horizon planning in an MDP for which a model-explicit RL technique is proposed. Our approach is not limited to navigation tasks; it allows for richer hierarchical refinement models and is combined with a powerful Monte-Carlo tree search technique.

The Hierarchical Planning in the Now (HPN) of [35] is designed for integrating task and motion planning and acting in robotics. Task planning in HPN relies on a goal regression hierarchized according to the level of fluents in an operator preconditions. The regression is pursued until the preconditions of the considered action (at some hierarchical level) are met by current world state, at which point acting starts. Geometric reasoning is performed at the planning level (i) to test ground fluents through procedural attachement (for truth, entailment, contradiction), and (ii) to focus the search on a few suggested branches corresponding to geometric bindings of relevant operators using heuristics called geometric suggesters. It is also performed at the acting level to plan feasible motions for the primitives to be executed. HPN is correct but not complete; however when primitive actions are reversible, interleaved planning and acting is complete. HPN has been extended in a comprehensive system for handling geometric uncertainty [36].

The integration of task and motion planning problem is also addressed in [37], which uses an HTN approach. Motion primitives are assessed with a specific solver through sampling for cost and feasibility. An algorithm called SAHTN extends the usual HTN search with a bookkeeping mechanism to cache previously computed motions. In comparison to this work as well as to HPN, our approach does not integrate specific constructs for motion planning. However, it is more generic regarding the integration of planning and acting.

Approaches based on temporal logics and situation calculus [38, 39, 40, 41] specify acting and planning knowledge through high-level descriptive models and not through operational models like in RAE. Moreover, these approaches integrate acting and planning without exploiting the hierarchical refinement approach described here.

Web services. The hierarchical representation framework of [42] includes abstract actions to interleave acting and planning for composing web services—but it focuses on distributed processes, which are represented as state transition systems, not operational models. It does not allow for refinement methods.

2.3 Planning Algorithms

The representation of our operational models has hierarchy. Also, we sample the outcomes of commands. So, the relevant planning approaches for our work are hierarchical planning algorithms and planners that do Monte Carlo rollouts with nondeterministic actions.

HTNs (Hierarchical Task Networks). Our methods are significantly different from those used in HTNs [43]: to allow for the operational models needed for acting, we use rich control constructs rather than simple sequences of primitives.

Planning with Monte Carlo rollouts. A wide literature on MDP-based probabilistic planning and Monte Carlo tree search refers to simulated execution, e.g., [44, 45, 46, 47] and sampling outcomes of action models e.g., RFF [48], FF-replan [49] and hindsight optimization [50]. The main conceptual and practical difference with our work is that these approaches use descriptive models, i.e., abstract actions on finite MDPs. Although most of the papers refer to doing the planning online, they do the planning using descriptive models rather than operational models. There is no notion of integration of acting and planning, hence no notion of how to maintain consistency between the planner's descriptive models and the actor's operational models. Moreover, they have no notion of hierarchy and refinement methods.

2.4 Planning and Learning

Learning HTN methods has also been investigated. HTN-MAKER [51] learns methods given a set of actions, a set of solutions to classical planning problems, and a collection of annotated tasks. This is extended for nondeterministic domains in [52]. [53] integrates HTN with Reinforcement Learning (RL), and estimates the expected values of the learned methods by performing Monte Carlo updates. At this stage, we do not learn the methods but only how to chose the appropriate one. We predict that learning refinement methods in an environment with nondeterminism, partial observability, and exogenous events is likely to be much more challenging than learning HTN methods in a classical environment.

2.5 Hierarchical Reinforcement Learning

Our approach shares some similarities with the work on planning by reinforcement learning (RL) [54, 55, 56, 57, 58], since we learn by acting in a (simulated) environment. However, most of the works on RL learn policies that map states to actions to be executed, and learning is performed in a descriptive model. We learn how to select refinement method instances in an operational model that allows for programming control constructs. This main difference holds also with works on hierarchical reinforcement learning, see, e.g., [59], [60], [61]. Works on user-guided learning, see e.g., [62], [63], use model based RL to learn relational models, and the learner is integrated in a robot for planning with exogenous events. Even if relational models are then mapped to execution platforms, the main difference with our work still holds: Learning is performed in a descriptive model. [64] uses RL for user-guided learning directly in the specific case of robot motion primitives.

The UCT algorithm (Upper Confidence bound applied to Trees) [46] does several Monte Carlo rollouts in an MDP, in order to come up with an approximately optimal policy. Our best refinement planning algorithm relies on a procedure called UPOM that does an UCT-style search in the space of hierarchical operational models.

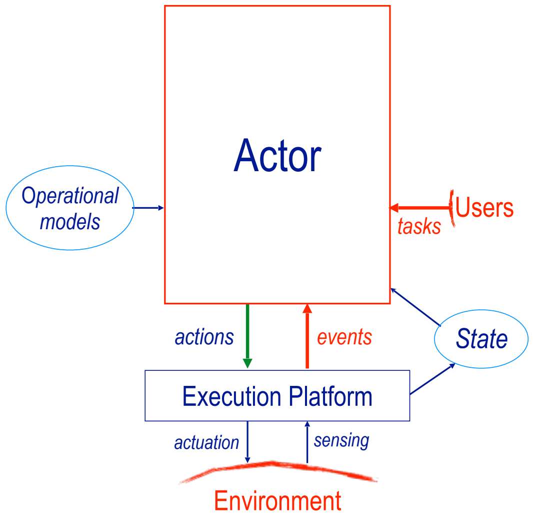
Learning planning domain models has been investigated along several approaches. In probabilistic planning, for example learning approaches in [65] and [66] learn a POMDP domain model through interactions with the environment, in order to plan by reinforcement learning or by sampling methods. In these cases, no integration with operational models and hierarchical refinements is provided.

Chapter 3: Acting and Planning Algorithms

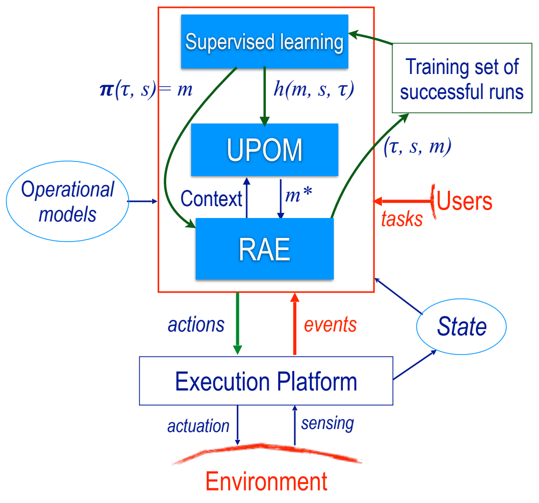
In this chapter, we describe the hierarchical operational model formalism and the acting engine RAE from [8, Chapter 3]. We define the components of the representation and illustrate it with several examples. Then, we describe our three refinement planning algorithms: APEplan, RAEplan, and Plan-with-UPOM, a planner based on a UCT-like procedure UPOM. For RAEplan, we prove its soundness, completeness and optimality under a certain set of assumptions. For UPOM, we map it to an MDP, to prove that it makes optimal choices.

3.1 Formalism: Hierarchical Operational Models

The usual *preconditions-effects* representation of actions in AI planning research is tailored for the efficient exploration of a state-transition system. It does not describe *how* to perform an action in a particular context, or how to react to dynamic events. For that, we will use a representation based on the one described in [8, Chapter 3], which has been designed for acting and reacting in a dynamic environment. It provides a hierarchical representation of tasks through alternative refinement methods and primitive actions. This representation is called *operational* since it allows an actor to perform the tasks requested by users and to react to events. The actor perceives the current state of the world and interacts with the environment for sensing and actuation through an execution platform (see Figures 3.1(a) for the general architecture, and 3.1(b) for the integration of planning, learning and refinement acting explained in subsequent sections). Let us describe the main ingredients.



(a)



(b)

Figure 3.1: (a) Architecture of an actor reacting to events and tasks through an execution platform; (b) Integration of refinement acting, planning and learning.

States. We rely on a parameterized state variable representation, i.e., a finite collection of mappings from typed sets of objects of the planning domain into some range, such as door-status(d) $\in$ {closed, open, cracked, unknown} which describes the status of a door d. Let X be a finite set of state variables; variable $x \in X$ takes values from the set Range(x), assumed at this stage to be finite.

A state is a total assignment of values to state variables. The world state ξ is updated through observation by the execution platform, reflecting the dynamics of the external world. For the purpose of the planning lookahead, ξ may be simplified into an abstract state $s \in S$ that gets updated from ξ each time the actor calls the planner. Both ξ and s are defined with the same set X of state variables. In general, s is a domain dependent abstraction of ξ, in which some state variables are ignored or range over sparser ranges. A given world state ξ is mapped to a single abstract state; an abstract state s may correspond to a subset of world sates.

To provide a convenient notation for handling partial knowledge, we extend the range of values of every state variable to include a special symbol, unknown, which is the default value of any state variable that has not been set or updated to another value.

It is also convenient to have a distinct set of variables that we call *internal* variables. Internal variables are updated by assignment statements inside methods. An assignment statement is of the form $x \leftarrow \mathit{expr}$, where *expr* may be either a ground value in Range(x), or a computational expression that returns a ground value in Range(x). Such an expression may include, for example, calls to specialized software packages.

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

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

Actions. An action is a primitive function with instantiated parameters that can be executed by the execution platform through sensory motor commands. It has *nondeterministic* effects. For the purpose of planning, we do not represent actions with formal templates, as usually done with descriptive models. Instead, we assume to have a generative nondeterministic sampling simulator, denoted Sample. A call to Sample(a, s) returns a state s' randomly drawn among the possible states resulting from the execution of a in s. Sample can be implemented simply through a probability distribution of the effects of a (see Section 3.5).

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

Refinement Methods. A refinement method is a triple of the form

$$(role, precondition, body).$$

The first field, either a task or an event, is its *role*; it tells what the method is about. When the *precondition* holds in the current state, the method is *applicable* for addressing the task or event by running a program given in the method's *body*. This program refines the task or event into a sequence of subtasks, actions, and assignments. It may use recursions and iteration loops.

Refinement methods are specified as parameterized templates with a name and list of parameters $\textit{method-name}(\textit{param}_1, \ldots, \textit{param}_k)$. An instance of a method is given by the substitution of its parameters by values that come from the arguments of the task the method is for and other state variables.

A method instance is applicable for a task or event τ if its role matches that of τ, and its preconditions are satisfied by the current values of the state variables. A method may have several applicable instances for a current state, task, and event. This will be illustrated in Example 3. An applicable instance of a method, if executed, addresses a task or an event by refining it, in a context dependent manner, into subtasks, actions, and possibly state updates, as specified in its body.

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

The following is an example of robots exploring partially known environments.

Example 1. *Consider several robots (UGVs and UAVs) moving around in a partially known terrain, performing operations such as data gathering, processing, screen-*

ing and monitoring. This domain is specified with the following:

- *a set of robots,* $R = \{g_1, g_2, a_1, a_2\}$,
- *a set of locations,* $L = \{base, z_1, z_2, z_3, z_4\}$,
- *a set of tools,* $\mathsf{TOOLS} = \{e_1, e_2, e_3\}$,
- $\mathsf{loc}(r) \in L$ *and* $\mathsf{data}(r) \in [0, 100]$, *for* $r \in R$, *are observable state variables that gives the current location and the amount of data the robot* r *has collected,*
- $\mathsf{status}(e) \in \{\mathrm{free}, \mathrm{busy}\}$ *is an observable state variable that says whether the tool* e *is free or being used,*
- $\mathsf{survey}(r, l)$ *is a command performed by robot* r *in location* l *that surveys* l *and collects data.*

Let $\mathsf{explore}(r, l)$ *be a task for robot* r *to reach location* l *and perform the command* $\mathsf{survey}(r, l)$*. In order to survey, the robot needs some tool that might be in use by another robot. Robot* r *should collect the tool, then move to the location* l *and execute the command* $\mathsf{survey}(r, l)$*. 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* $\mathsf{depositData}$*(r). Here is a refinement method to do this.*

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

A partial refinement tree for the task explore(r, l) *refined using* m1-explore(r, l) *is shown in Figure 3.2. This tree corresponds to the case where the subtasks* getTool(r) *and* moveTo(r, l) *succeed and* data(r) *is 100. A refinement tree for the case where the subtask* moveTo(r, l) *fails to change the location of r to l is shown in Figure 3.3. Figure 3.4 shows a refinement tree for the case where r surveys location l successfully but doesn't need to deposit the data.*

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

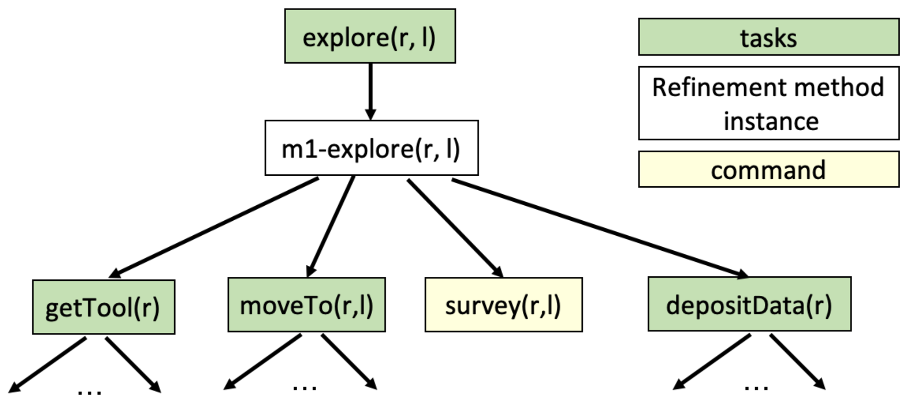


Figure 3.2: A possible refinement tree for the task explore(r, l) corresponding to a successfully accomplished task.

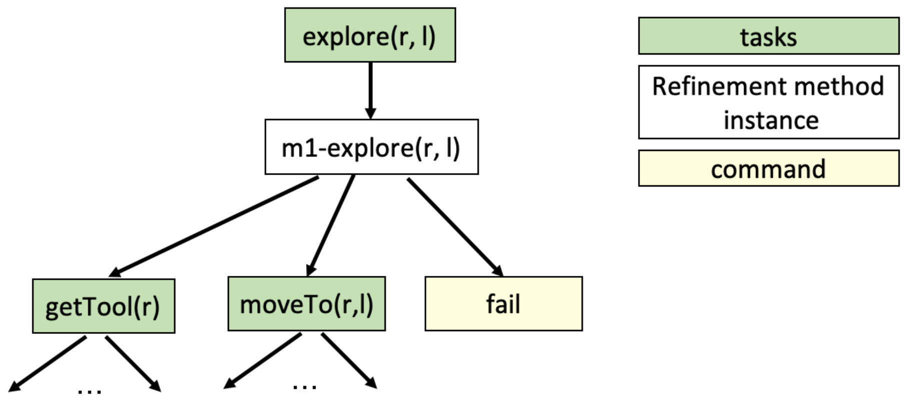


Figure 3.3: A refinement tree for the task explore(r, l) where r failed to reach location l.

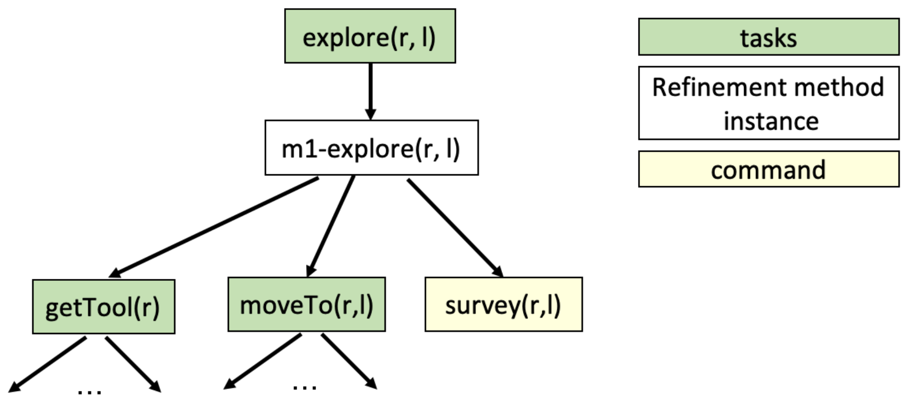


Figure 3.4: A refinement tree for the task explore(r, l) where r surveyed location l successfully but does not need to deposit the data.

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

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

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

A refinement tree for the task explore where m1-getTool(r) is used to refine getTool(r) is shown in Figure 3.5. A refinement tree for the task explore where m2-getTool(r) is used to refine getTool(r) is shown in Figure 3.6.

Expanding further from Figure 3.6 by refining the move sub-tasks, we get the refinement tree shown in Figure 3.7.

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

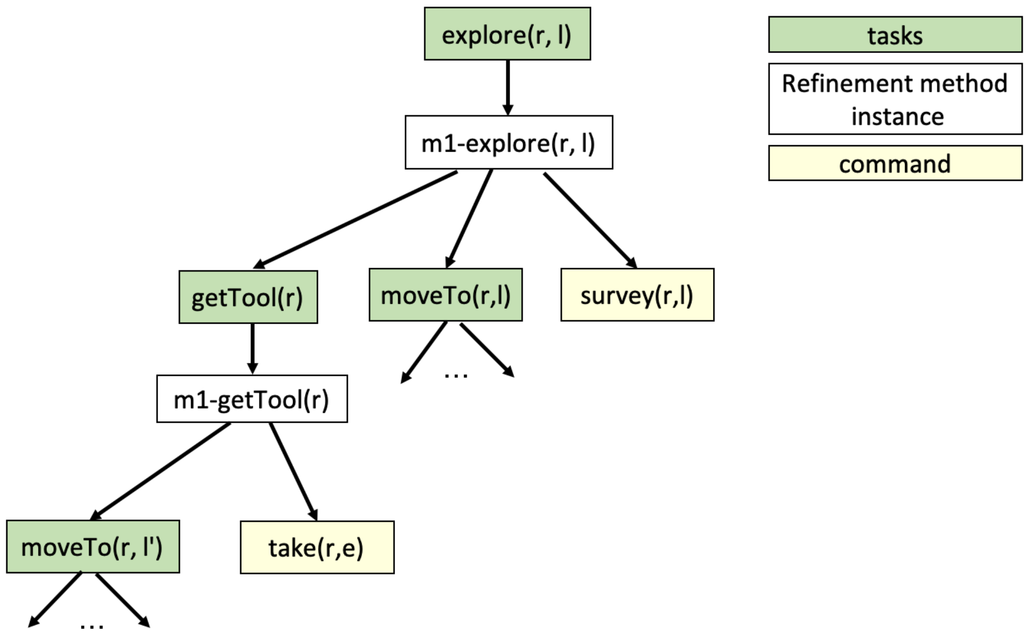


Figure 3.5: A refinement tree for the task explore(r, l) where m1-getTool(r) is used to refine the task getTool(r).

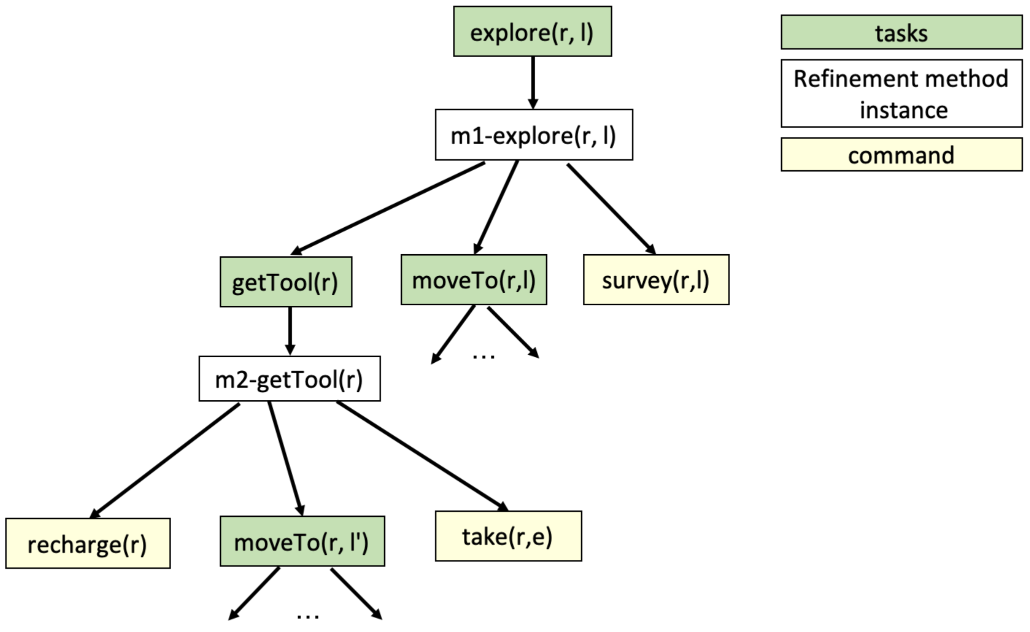


Figure 3.6: A refinement tree for the task explore(r, l) where m2-getTool(r) is used to refine the task getTool(r).

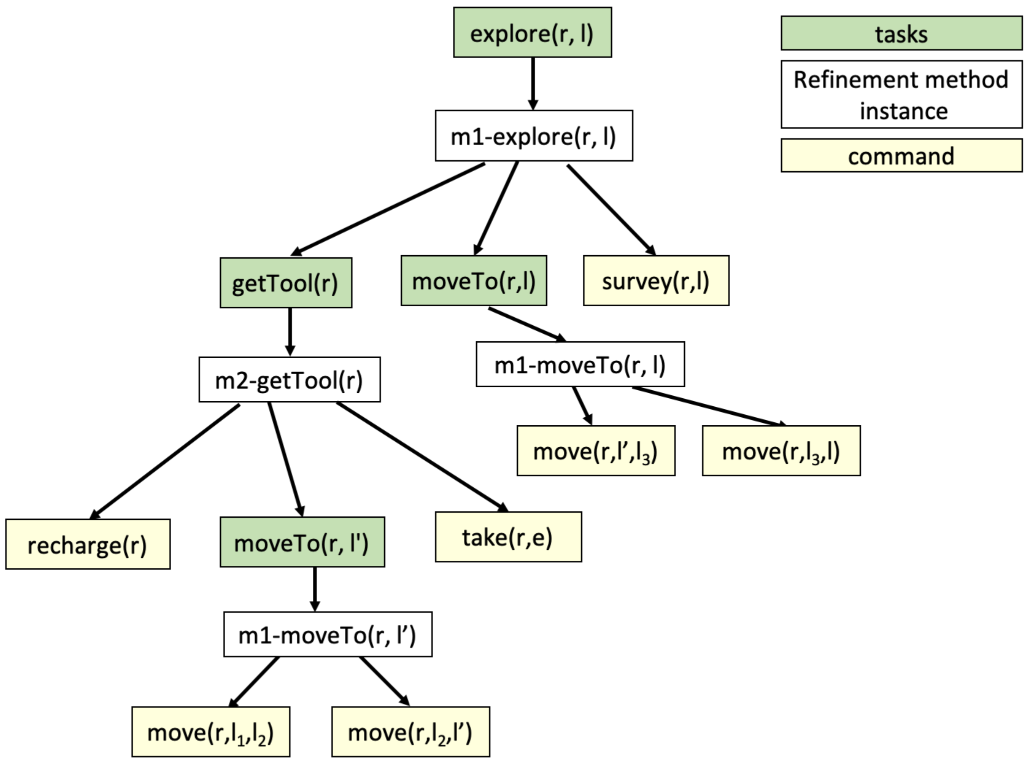


Figure 3.7: A refinement tree for the task explore(r, l) where the move subtasks are refined.

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

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

The following is an example of a simplified search-and-rescue domain to illustrate the representation.

Example 3. *Consider a set* R *of robots performing search and rescue operations in a partially mapped area. The robots have to find people needing help in some area and leave them a package of supplies (medication, food, water, etc.). This domain is specified with state variables such as* robotType$(r) \in \{UAV,\ UGV\}$, $r \in R$*, a finite* *set of robot names;* $\mathsf{hasSupply}(r) \in \{\top, \bot\}$*;* $\mathsf{loc}(r) \in L$*, a finite set of locations. A rigid relation* $\mathsf{adjacent} \subseteq L^2$ *gives the topology of the domain.*

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

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

```
m1-survey($l, r$)
   task: survey($l$)
    pre: robotType($r$) = $UAV$ and loc($r$) = $l$ and status($r$) = $free$
  body: for all $l'$ in neighbouring areas of $l$ do:
            moveTo($r, l'$)
            for $cam$ in cameras($r$):
               if $\textsc{DetectPerson}(r, cam) = \top$ then:
                  if hasSupply($r$) then rescue($r, l'$)
                  else $\textsc{TriggerAlarm}(r, l')$
```

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

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

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

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

- Ξ is the set of world states the actor may be in.
- $\mathcal{T}$ is the set of tasks and events the actor may have to deal with.
- $\mathcal{M}$ is the set of methods for handling tasks or events in $\mathcal{T}$, Applicable(ξ, τ) is the

set of method instances applicable to τ in state ξ.

- $\mathcal{A}$ is the set of actions the actor may perform. We let $\gamma(\xi, a)$ be the set of states that may be reached after performing action a in state ξ.

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

The deliberative acting problem can be stated as follows: given Σ and a task or event $\tau \in \mathcal{T}$, what is the "best" method instance $m \in \mathcal{M}$ to perform τ in a current state ξ [1]. The acting domain is $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$. Strictly speaking, the actor does not require a plan, i.e., an organized set of actions or a policy. It requires a selection procedure which designates for each task or subtask at hand the "best" method instance for pursuing the activity in the current context.

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

3.2 The actor, RAE

RAE (for Refinement Acting Engine) is adapted from [8, Chapter 3]. It maintains an *Agenda* consisting of a set of *refinement stacks*, one for each root task or event that needs to be addressed. A refinement stack stack is a LIFO list of tuples of

[1] Please note that the best refinement method instance for a task also depends on the refinement tree (see Figure 3.8).

the form $(\tau, m, i, \mathit{tried})$ where τ is an identifier for the task or event; m is a method instance to refine τ (set to *nil* if no method instance has been chosen yet); i is a pointer to a line in the body of m, initialized to 1 (first line in the body); and *tried* is a set of refinement method instances already tried for τ that failed to accomplish it. A stack stack is handled with the usual push, pop and top functions.

```
 1 RAE:
 2 $Agenda \leftarrow$ empty list
 3 while $True$ do
 5     for each new task or event $\tau$ to be addressed do
 7         observe current state $\xi$
 9         $m \leftarrow \mathsf{Select}(\xi, \tau, \langle(\tau, nil, 1, \emptyset)\rangle, d_{max}, n_{ro})$
11         if $m = \emptyset$ then output($\tau$, “failed”)
12         else  $Agenda \leftarrow Agenda \cup \{\langle(\tau, m, 1, \emptyset)\rangle\}$
13     end
15     for each stack $\in Agenda$ do
16         observe current state $\xi$
17         stack $\leftarrow$ Progress(stack, $\xi$)
19         if stack $= \emptyset$ then
20             $Agenda \leftarrow Agenda \setminus$ stack
21             output($\tau$, “succeeded”)
22         end
24         else if stack =failure then
25             $Agenda \leftarrow Agenda \setminus$ stack
26             output($\tau$, “failed”)
27         end
28     end
29 end
```

Algorithm 1: Refinement Acting Engine RAE

When RAE addresses a task τ, it must choose a method instance m for τ. This is performed by function Select (lines 9 of RAE, 24 of Progress, and 7 of Retry). Select takes five arguments: the current state ξ, task τ, and stack stack, and two control parameters d_{max}, n_{ro} which are needed only for planning. In purely reactive mode (without planning), Select returns the first applicable method instance, according to a pre-defined ordering, which has not already been tried (*tried* is given in stack). Note that this choice is with respect to the current world state ξ. Lines 7,22,5 in RAE, Progress and Retry respectively, specify to get an update of the world state from the execution platform. If Applicable$(\xi, \tau) \subseteq$ *tried*, then Select returns $\emptyset$, i.e., there is no applicable method instances for τ in ξ that has not already been tried, meaning a failure to address τ.

The first inner loop of RAE (line 5) reads each new root task or event τ to be addressed and adds to the *Agenda* its refinement stack, initialized to $\langle(\tau, m, 1, \emptyset)\rangle$, m being the method instance returned by Select, if there is one. The root task τ for this stack will remain at the bottom of stack until solved; the subtasks in which τ refines will be pushed onto stack along with the refinement. The second loop of RAE progresses by one step in the topmost method instance of each stack in the *Agenda*.

To progress a refinement stack stack, Progress (Algorithm 2) focuses on the tuple $(\tau, m, i, \mathit{tried})$ at the top of stack. If the current line $m[i]$ is an action already triggered, then the execution status of this action is checked. If the action $m[i]$ is still running, this stack has to wait, but RAE goes on for other pending stacks in the *Agenda*. If $m[i]$ failed, Retry examines alternative method instances. Otherwise the action $m[i]$ is done: RAE will proceed in the following iteration with the next step in method instance m, as defined by the function Next (Algorithm 3).

Next(stack, ξ) advances within the body of the topmost method instance m in stack as well as with respect to stack. If i is the last step in the body of m, the current tuple is removed from stack: method instance m has successfully addressed τ. If τ is a root task; Next and Progress return $\emptyset$, meaning that τ succeeded; its

```
1  Progress(stack, $\xi$):
2  $(\tau, m, i, tried) \leftarrow$ top(stack)
4  if $m[i]$ is an already triggered action then
5      case execution-status($m[i]$):
6          running:   return stack
8          failed:    return  Retry(stack)
9          done:      return Next(stack, $\xi$)
10 end
12 else if $m[i]$ is an assignement step then
13     update $\xi$ according to $m[i]$
14     return Next(stack, $\xi$)
15 end
16 else if $m[i]$ is an action $a$ then
17     trigger the execution of action $a$
18     return stack
19 end
20 else if $m[i]$ is a task $\tau'$ then
22     observe current state $\xi$
24     $m' \leftarrow$ Select($\xi, \tau'$, stack, $d_{max}, n_{ro}$)
25     if $m' = \emptyset$ then return  Retry(stack)
26     else return push(($\tau', m', 1, \emptyset$), stack)
27 end
```

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

```
1  Next (stack, $\xi$):
2  repeat
3      $(\tau, m, i, tried) \leftarrow$ top(stack)
4      pop(stack)
5      if stack = $\langle\rangle$ then  return $\langle\rangle$
6  until $i$ is not the last step of $m$
7  $j \leftarrow$ step following $i$ in $m$ depending on $\xi$
8  return push(($\tau, m, j, tried$), stack)
```

Algorithm 3: Next step in a method instance m for a given stack.

stack stack is removed from the *Agenda*. If i is not the last step in m, RAE proceeds to the next step in the body of m. This step j following i in m is defined with respect to the current state ξ and the control instruction in line i of m, if any.

Starting from line 12 in Progress, i points to the next line of m to be processed.

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

```
1  Retry(stack):
2  $(\tau, m, step, tried) \leftarrow$ pop(stack)
3  $tried \leftarrow tried \cup \{m\}$                                   // $m$ failed
5  observe current state $\xi$
7  $m' \leftarrow$ Select($\xi, \tau$, stack, $d_{max}, n_{ro}$)
9  if $m' \neq \emptyset$ then return push(($\tau, m', 1, tried$), stack)
10 else if stack $\neq \emptyset$ then return Retry(stack)
12 else return failure
```

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

Retry (Algorithm 4) adds the failed method instance m to the set of method instances that have been tried for τ and failed. It removes the corresponding tuple from stack. It retries refining τ with another method instance m' returned by Select which has not been already tried (line 9). If there is no such m' and if stack is not empty, Retry calls itself recursively on the topmost stack element, which is the one that generated τ as a subtask: retrial is performed one level up in the refinement tree. If stack stack is empty, then τ is the root task or event: RAE failed to accomplish τ.

RAE fails either (i) when there is no method instance applicable to the root task in the current state (line 11 of RAE), or (ii) when all applicable method instances have been tried and failed (line 27). A method instance fails either (i) when one of its actions fails (line 8 in Progress) or (ii) when all applicable method instances for one of its subtasks have been tried and failed (line 12 in Retry).

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

3.3 Planner, APEplan

The actor's problem is how to "best" perform a task τ in a current state ξ. In the purely reactive approach, RAE chooses a refinement method instance from a predefined order of refinement methods, without comparing alternative options in the current context. Another alternative is to call a planner everytime a task or sub-task needs to be refined. Our first refinement planning algorithm, for planning using hierarchical operational models, is called APEplan. APEplan optimized the number of commands in the refinement tree for the task. It does this optimization using a greedy approach. Since APEplan has several limitations and is not our best planning algorithm, the full pseudocode of APEplan is described in Appendix A. Here is a summary of it. APEplan is a modified version of the RAE pseudocode that incorporates the following modifications:

1. Each call to APEplan returns a *refinement tree* T (see Figure 3.8) for a task τ 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 APEplan produced during a Monte Carlo rollout of m.

2. In line 9 of RAE, line 24 of Progress, and line 7 of Retry, APEplan 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 APEplan chooses one that optimizes cost, time or any other user-specified objective function. If the set is empty, then APEplan 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 RAE 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 5 of Progress (the case where *step* is a command), instead of sending *step* to the actor's execution platform, APEplan 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, non-deterministic, or probabilistic state-transition model, or a simulator of some kind. Different calls to the predictive model may produce different results. However, APEplan simplifies and approximates this by calling the predictive model only once. Our planner, called RAEplan, described in the next section gets rids of this simplification and estimates the outcome by calling the predictive model several times and estimating some optimization criterion.

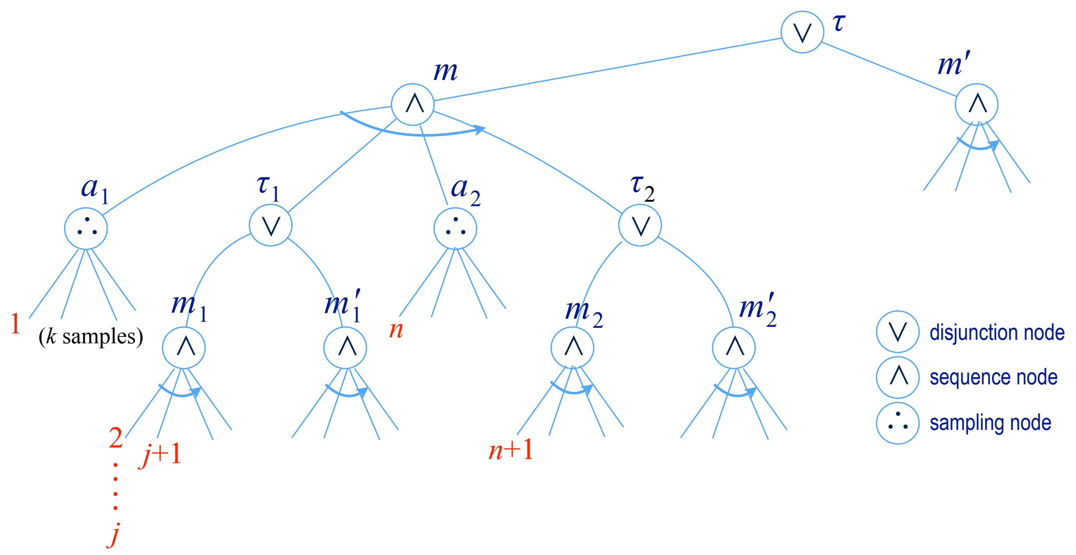


Figure 3.8: A refinement tree, with three types of nodes: *disjunction* for a task over possible methods, *sequence* for a method over all its steps, and *sampling* for an action over its possible outcomes. A rollout can be, for example, the sequence of nodes marked 1 (a sample of a_1), 2 (first step of m_1), $\ldots$, j (subsequent refinements), $j+1$ (next step of m_1), $\ldots$, n (a sample of a_2), $n+1$ (first step of m_2), etc.

3.4 Planner, RAEplan

RAEplan does a SLATE style [8, Chapter 6] recursive search to optimize a criterion called *efficiency* that is roughly reciprocal of the cost. We first consider the simple case where the simulated execution of method instance never fails; then we'll explain how to account for planning-time failures (which are distinct from running-time failures addressed by Retry) using efficiency. The reason it is difficult to evaluate planning time failures using cost is the following: Consider a task τ with two applicable refinement method instances, m_1 and m_2. Let's say, we simulate each of them n times. m_1 succeeds n_1 times and m_2 succeeds n_2 times. With $n_1 > n_2$, one would prefer to use m_1 over m_2. But with the expected cost of both m_1 and m_2 being infinite (a failed simulated execution equals infinite cost), there is no way to distinguish between them using expected cost.

Now, let us first discuss the case where there are no planning time failures. We choose a refinement method that has a refinement tree with a minimum expected cost for accomplishing a task τ (along with the remaining partially accomplished tasks in the current refinement stack).

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

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

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

$$EV_{s' \in S'} \left[\{cost(s, i, s') + C^*(s', \textsf{next}(s', R_p))\} \right], \tag{3.1}$$

  where S' is the set of outcomes of command i in s and EV stands for expected value.

- If i is an assignment statement, then $C^*(s, R_p) = C^*(s', \textsf{next}(s', R_p))$, where s' is the state produced from s by performing the assignment statement.

- If i is a task, then $C^*(s, R_p)$ recursively optimizes over the candidate method instances for i. That is

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

  where $M' = Candidates(i, s)$.

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

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

Next, let us see how to account for planning failures. Note that C^* cannot handle failures because the cost of a failed command is ∞, resulting in an expected value of ∞ in equation 3.1 for all commands with at least one possibility of failure. In order to overcome this, we introduce the *efficiency* criteria, $\nu = 1/cost$, to measure the efficiency of a plan. RAEplan maximizes efficiency instead of minimizing cost.

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

$$1/(c_1 + c_2) = e_1 e_2/(e_1 + e_2). \tag{3.2}$$

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

$$\begin{aligned} e_1 \bullet e_2 = e_2 \text{ if } e_1 = \infty, \text{ else} \\ e_1 \text{ if } e_2 = \infty, \text{ else } e_1 e_2/(e_1 + e_2). \end{aligned} \tag{3.3}$$

If τ_1 (or τ_2) fails, then c_1 is ∞, $e_1 = 0$. Thus $e_1 \bullet e_2 = 0$, meaning that τ fails with this decomposition. Note that formula 3.3 is associative.

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

Additional details are on the next page, in the *Experiments and Analysis* subsection.

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

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

$$\tfrac{1}{k} \textstyle\sum_{s' \in S'} \tfrac{1}{\mathit{cost}(s,i,s')} \bullet E^*_{b,k}(s', \mathsf{next}(s', R_p)), \tag{3.4}$$

  where S' is a random sample of k outcomes of command i in state s, with duplicates allowed. Since S' has the probability distributions of the outcomes of the commands, it converges asymptotically to the expected value of E^*.

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

- If i is a task, then $E^*_{b,k}(s, R_p)$ recursively optimizes over the candidate method instances for i. That is:

$$E^*_{b,k}(s, R_p) = \max_{m \in M'} E^*_{b,k}(s, (i, m, \mathsf{nil}, \emptyset).R_p), \tag{3.5}$$

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

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

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

where $a.R_p$ is a refinement stack with a pushed on $\mathit{top}(R_p)$. The larger the values of b and k in $E^*_{b,k}(s, R_p)$, the more plans RAEplan will examine. It can be proved that when $b = \max_{\tau,s}\{|\mathit{Candidates}(\tau, s)|\}$ (call it b_{max}) and $k \to \infty$, the method instance returned by RAEplan converges to one with the maximum expected efficiency. We now outline the proof. It is by induction on the number of remaining push operations in R_p. In the base case, the number of remaining push operations in R_p is 1. This has to be a command, because if it were a task, then it would further refine into more commands, resulting in more push operations. The maximum expected efficiency for a command is just its expected value. The induction hypothesis is that for any stack R_p with n remaining push operations, $E^*_{b_{max},\infty}$ gives the maximum expected efficiency. In the inductive step, we show that equations 3.4 and 3.5 converge to the maximum expected efficiency for any R_p with $n + 1$ remaining push operations.

3.4.1 Properties of RAEplan

Now, we present the theoretical results related to the algorithms RAE and RAEplan. The theorems rely on the following assumptions:

1. We take infinite number of samples for every command. This is important to obtain the exact probability of each outcome

2. We look at all candidate method instances for every task

3. No dynamic events happen in the environment

4. The predictive models of actions accurately model the real world (the state transition probabilities are correct)

Theorem 1 (Soundness/Correctness of RAEplan)**.** *The refinement method chosen by RAEplan for refining a task τ will be successful in accomplishing τ in the real world.*

Proof. This theorem follows from the assumptions 3 and 4 above. RAEplan simulates the refinement of the task τ in a simulator that accurately models the real world. It does so by looking at all the applicable refinement methods for τ and simulating them step by step. Each step corresponds to a push or pop operation in the current refinement stack R_p for τ. So, if all the remaining actions (consisting of sub-tasks and commands) in R are successfully simulated by RAEplan, and there are no dynamic events that change the state externally (Assumption 3), the method chosen by RAEplan will successfully accomplish τ in the real world.

□

Corollary 1 (Correctness of RAE)**.** *Because RAEplan returns the method with the maximum expected efficiency every time it is called, RAE accomplishes the task with maximum possible efficiency.*

Theorem 2 (Completeness of RAEplan)**.** *If there exists a method for successfully accomplishing a task τ, then RAEplan will find it.*

Proof. This follows from the assumptions 1 and 2 above. Let m be the method that will succeed in accomplishing τ. From assumption 2, RAEplan will look at m and simulate it step by step. From assumptions 1 and 4, simulation of the commands and sub-tasks will be accurate and they will correctly model what happens in the real world. So, RAEplan will succeed in accomplishing τ the simulation of m, and return it. One could argue that RAEplan may succeed in simulating some method m' other than m. But from Theorem 1 it follows that m' will also succeed in accomplishing τ in the real world.

□

Theorem 3 (Optimality of RAEplan)**.** *RAEplan (τ) will return a method which has the maximum expected efficiency for accomplishing the task τ.*

Proof. Let the current refinement stack for τ be R_p. This corresponds to a partially built/executed refinement tree which has remaining sub-tasks and commands that need to be accomplished later.

RAEplan continues to search for the solution by doing a series of push/pop operations on the refinement stack R_p until remaining of R_p is accomplished. This happens through simulation and not in the real world.

We prove the optimality of RAEplan by induction on the number of remaining push operations. We can ignore the pop operations because it doesn't change the efficiency and they have a one-to-one correspondence with the push operations.

Basis. In the base case, the number of remaining push operations for the stack is 1. This has to be a command, because if it were a task, then it would further refine into more commands, resulting in more push operations.

For a command, the efficiency is

$$E^*_{b,k}(s, R_p) = \tfrac{1}{k} \textstyle\sum_{s' \in S'} \tfrac{1}{cost(s,i,s')} \bullet E^*_{b,k}(s', \mathsf{next}(s', R_p)). \tag{3.6}$$

Thus, in the current state s,

$$E(s, R) = \lim_{k \to \infty} \frac{1}{k} \sum_{s' \in S'} \frac{1}{cost(s, c, s')} \bullet E^*(s', \langle\rangle)$$

$$= \lim_{k \to \infty} \frac{1}{k} \sum_{s' \in S'} \frac{1}{cost(s, c, s')} \bullet \infty$$

$$= \lim_{k \to \infty} \frac{1}{k} \sum_{s' \in S'} \frac{1}{cost(s, c, s')}$$

$$= \text{Expected efficiency of the command } c$$

The maximum expected efficiency for a command is just its expected value. Hence, the theorem is true in the base case.

Induction Hypothesis. Our formula for calculating the efficiency of a stack gives the maximum expected efficiency for all refinement stacks with less than n remaining push operations.

Induction. Consider the case when a refinement stack R_p has $n + 1$ remaining push operations.

Case 1: The next push operation is a task.

The efficiency is calculated using the formula:

$$E^*_{b,k}(s, R_p) = \max_{m \in M'} E^*_{b,k}(s, (i, m, \mathsf{nil}, \emptyset).R_p), \tag{3.7}$$

$E^*_{b,k}(s, (i, m', \mathsf{nil}, \emptyset).R_p)$ is the maximum expected efficiency from the induction hypothesis because it has $\leq n$ remaining push operations. In the current step, we look at all possible candidates and take the maximum which will the maximum expected efficiency for the refinement stack R_p.

Case 2: The next push operation is a command.

The efficiency is calculated using equation 3.6 which is as follows.

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

$E^*_{b,k}(s', \mathsf{next}(R_p))$ is the maximum expected efficiency from the induction hypothesis because it has $\leq n$ remaining push operations. In the current step, we look at k outcomes of the command and take the average. With $k \to \infty$, this will give the maximum expected efficiency for the refinement stack R_p.

$\square$

Theorem 4. *Time complexity of* RAEplan *is*

$$O\left(|Cand|.b^{\frac{q^{h+1}-1}{q-1}}.k^{\frac{c(q^h-1)}{q-1}}\right)$$

where

Cand is the set of applicable methods for τ*,*

h *is the maximum possible height of the refinement tree for* τ*,*

q *is the maximum number of sub-tasks within a method,*

c *is the maximum number of commands within a method.*

Note: This assumes that there are no cycles and assumptions (1) and (2) do not hold.

Proof. One way of looking at what RAEplan does is that in order to choose a suitable refinement method for a task τ, RAEplan simulates several refinement trees for it. It looks at a particular refinement tree exactly once and goes through all its nodes one by one. Let T be the set of all such refinement trees. Let $T = \{t_1, t_2, ..., t_n\}$ with $n = |T|$. Thus, time complexity of RAEplan is $O(\sum_{i=1}^{n} |t_i|)$ where $|t_i|$ is the number of nodes in the refinement tree t_i. Looking at one node of a tree has $O(1)$ running time. So, if we can count all the nodes of the trees in T, we can estimate the running time of RAEplan. Each of the task nodes at any level i can have q sub-tasks and c commands.

Thus, the maximum number of task nodes is

$$1 + q + q^2 + ... + q^h = \frac{q^{h+1}-1}{q-1},$$

and the maximum number of command nodes is

$$c + cq + cq^2 + ... + cq^{h-1} = c.\frac{q^h - 1}{q - 1}$$

In each of the task nodes, we look at b possible choices (except the root node where we look at all applicable candidates). In each of the command nodes, we look at k possible outcomes. So, the running time of RAEplan is

$$O\left(|Cand|.b^{\frac{q^{h+1}-1}{q-1}}.k^{\frac{c(q^h-1)}{q-1}}\right).$$

$\square$

3.5 UPOM, a UCT-like search procedure

The actor's problem was informally defined as how to "best" perform a task τ in a current state ξ. In the purely reactive approach, RAE, chooses a refinement method instance from a predefined order of refinement methods, without comparing alternative options in the current context. RAEplan suggests a refinement method instance by following a SLATE style sampling strategy and choosing a method instance with the highest expected efficiency. In this section, we define a general utility function to assess and compare methods in Applicable(ξ, τ) to select the best one; and a planner based on a procedure that does UCT-style sampling and returns the method instance with the highest expected utility.

The utility function might, in principle, be used by an exact optimization procedure for finding the optimal method instance for a task. We propose a more efficient Monte Carlo Tree Search approach, called Plan-with-UPOM, for finding an approximately optimal method instance. Plan-with-UPOM relies on a function, called UPOM, inspired from the Upper Confidence bounds search applied to Trees (UCT) procedure. UPOM (*UCT Procedure for Operational Models*) has parameters d for rollout depth and n_{ro} for number of rollouts. It relies on a heuristic function h for estimating the criterion at the end of the rollouts when $d < \infty$.

Plan-with-UPOM runs multiple simulations using the methods in $\mathcal{M}$ and a *generative sampling model* of actions. This model is defined as a function Sample: $S \times \mathcal{A} \rightarrow S$. Sample$(s, a)$ returns a state s' randomly drawn from $\gamma(s, a)$, with $\gamma : S \times A \rightarrow (2^S \cup \{\mathsf{failed}\})$. The transition function γ is augmented with the token failed to account for possible failures of a. We assume, as usual, that the sampling reflects the probability distribution of the action's real-world outcomes.

A simulation of a method m for a task τ during planning goes successively through the steps of m, as required by the control flow for the current context, and generates a sequence of *simulated states* $\langle s_0, \ldots, s_i, \ldots \rangle$, where initially s_0 corresponds to the current real-world state ξ. The utility function is computed along such a sequence, taking into account the *deterministic* refinements of methods and the *nondeterministic* outcomes of actions (see Figure 3.8). In simulation during planning, we do not Retry, as in RAE, but we take into account possible failures. We assume the simulations to be fast enough with respect to the real-world dynamics. Hence, we do not consider possible changes in ξ during a simulation. These changes, if any, are dealt with at the acting level.

3.5.1 Utility criteria and optimal approach

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

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

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

$$v_e(s, a, s') = \begin{cases} 0 & \text{if } s' = \text{“}\mathsf{failed}\text{”}, \\ 1/\mathsf{Cost}(s, a, s') & \text{otherwise.} \end{cases} \tag{3.8}$$

The cumulative efficiency value of two successive actions whose values are $v_{e1} = 1/c_1$ and $v_{e2} = 1/c_2$ is denoted: $v_{e1} \oplus v_{e2} = 1/(c_1 + c_2) = v_{e1} \times v_{e2}/(v_{e1} + v_{e2})$.

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

$$v_s(s, a, s') = \begin{cases} 0 & \text{if } s' = \text{ ``}\mathsf{failed}\text{''}, \\ \\ 1 & \text{otherwise.} \end{cases} \tag{3.9}$$

The cumulative success ratio value of two successive actions in a method whose values are v_{e1} and v_{e2} is $v_{e1} \oplus v_{e2} = v_{e1} \times v_{e2}$.

For both value functions v_e and v_s, the operator $\oplus$ is associative, which is needed for combining successive steps. We use $\mathbb{I}$ to denote the identity element for operation $\oplus$, i.e., $x \oplus \mathbb{I} = x$; $\mathbb{I}$ is ∞ for v_e and 1 for v_s. Note that if either of two actions in a method m fails, their combined value is 0, since m also fails.

Let us now define a utility function for methods using either v_e or v_s. In order to compute the expected utility of a method m we need to consider possible *traces* of the execution of m for a task τ. In RAE, an execution trace was conveniently represented though the evolution of stack for the task τ. In planning, we similarly use stack as a LIFO list of tuples $(\tau, m, i, \textit{tried})$, as defined in RAE.[2] For a given simulation of m for τ, stack is initialized as a copy of the current stack in RAE.We

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

progress in the simulation of m step by step using the function next (Algorithm 3), pushing in stack a new tuple when a step requires a refinement into a subtask.

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

$$U(m, s, \mathsf{stack}) = \begin{cases} U(m, s', \mathsf{next}(\mathsf{stack}, s)) & \text{if } m[i] \text{ is an assignment,} \\ v(s, a, s') \oplus U(m, s', \mathsf{next}(\mathsf{stack}, s)) & \text{if } m[i] \text{ is an action } a, \\ U(m', s, \mathsf{push}((\tau', m', 1, \emptyset), \mathsf{next}(\mathsf{stack}, s)) & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if } \mathsf{stack} = \emptyset, \end{cases} \tag{3.10}$$

Here, v is either v_e or v_s. An assignment step changes the state from s to s' but does not change the utility U. An action a changes the state nondeterministically to s'; the utility is the combined value of a and the utility of the remaining step. A refinement step does not change the state; it is addressed in this particular simulation by refining τ into τ' with m'. The function next moves to the following step, and to the empty stack at the end of every simulated execution.

From Equation 3.10 we derive the *maximal expected utility* of m for τ by maximizing recursively over all possible refinements in m and averaging over all possible outcomes of actions, including failures:

$$U^*(m, s, \mathsf{stack}) = \begin{cases} U^*(m, s', \mathsf{next}(\mathsf{stack}, s)) & \text{if } m[i] \text{ is an assignment,} \\ \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a) \times [v(s,a,s') \oplus U^*(m, s', \mathsf{next}(\mathsf{stack}, s))] & \\ & \text{if } m[i] \text{ is an action } a, \\ \max_{m' \in \mathsf{Applicable}(s,\tau')} U^*(m', s, \mathsf{push}((\tau', m', 1), \mathsf{next}(\mathsf{stack}, s)) & \\ & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if } \mathsf{stack} = \emptyset. \end{cases} \tag{3.11}$$

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

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

$$m^*_{\tau,s} = \operatorname{argmax}_{m \in \mathsf{Applicable}(s,\tau)} U^*(m, s, \langle(\tau, m, 1, \emptyset)\rangle) \tag{3.12}$$

It is possible to implement Equation 3.11 directly as a recursive backtracking optimization algorithm and to make the planning algorithm return $m^*_{\tau,s}$, as defined above. However, this would be too computationally demanding and not practical for an online planner. We propose instead to seek an approximately optimal method with an anytime controllable procedure using a Monte Carlo Tree Search algorithm in the space of operational models.

3.5.2 A planning algorithm based on UCT

To find an approximation $\tilde{m}$ of m^*, we propose a progressive deepening Monte Carlo Tree Search procedure with n_{ro} rollouts, down to a depth d_{max} in the refinement tree of a task τ (see Figure 3.8). The basic ideas are the following:

- at an action node of the search tree, we average over the value of the corresponding n_{ro} rollouts;
- at a task node, we choose the refinement method instance with the highest expected utility;
- starting from $d = d_{\max}$,we decrease d for a refinement step and an action step, but not in an assignment step;
- we take a heuristic estimate of the utility of the remaining refinements at the tip of a rollout, i.e., at $d = 0$;
- we stop a rollout at a failure of an action or a refinement, and return a value $U_{\mathsf{Failure}} = 0$; we also stop when the stack is empty and return $U_{\mathsf{Success}} = \mathbb{I}$.

This is detailed in algorithms 5 and 6. Select is called by RAE with five parameters: ξ, τ, and stack, and the control parameters, d_{max} the maximum rollout depth, and n_{ro} the number of UCT rollouts. Recall that on a new root task τ, RAE calls Select with $\sigma = \langle(\tau, nil, 1, \emptyset)\rangle$. Select returns $\tilde{m}$, an approximately optimal method for τ, or $\emptyset$ if no method is found, i.e., if there is no applicable method for τ in ξ, but of those already tried by RAE for this task. Select uses a copy of RAE's current stack stack, and a simulation state s, which is an abstraction of the current execu-

```
1  Plan-with-UPOM($\xi, \tau$, stack, $d_{max}, n_{ro}$):
2  $(\tau, m, i, tried) \leftarrow$ top(stack)
3  $M \leftarrow$ Applicable$(\xi, \tau) \setminus tried$
4  if $M = \emptyset$ then return $\emptyset$
5  if $|M = \{m\}| = 1$ then return $m$
6  $s \leftarrow$ Abstract$(\xi)$ ; $\sigma \leftarrow$ copy of stack; $d \leftarrow 0$
8   $\tilde{m} \leftarrow \text{argmax}_{m \in M} h(\tau, m, s)$
10  repeat
11      $d \leftarrow d + 1$
13      for  $n_{ro}$ times do
14          UPOM ($s$, push($(\tau, nil, 1, \emptyset)$, stack), $d$)
15      end
16      $\tilde{m} \leftarrow \text{argmax}_{m \in M} Q_{\text{stack},s}(m)$
17 until $d = d_{max}$ or search time is over
18 return $\tilde{m}$
```
Algorithm 5: Plan-with-UPOM is a progressive deepening procedure using UPOM for finding an approximately optimal method instance.

tion state ξ (e.g., in Example 1, l can be a precise metric location for acting and topological reference for planning). It initializes $\tilde{m}$ with a heuristic estimates (line 8). It performs a succession of simulations at progressively deeper refinement levels using the function UPOM to evaluate the utility of a candidate method instance. The progressive deepening loop (line 10) is pursued until reaching the maximum rollout depth, or until the actor interrupts the search because of time limit or any other reason, at which point the current $\tilde{m}$ is returned and will be tried by RAE. Select is an *anytime* procedure: it returns a solution whenever interrupted.

UPOM (Algorithm 6) takes as arguments a simulation state s, a stack stack, and the rollout depth d. It performs one rollout over recursive calls for a method m and its refinements. On the first call of a rollout, $m = nil$, meaning that no method has yet been chosen. A method m_c is chosen among untried methods (line 20). If all methods have been tried, m_c is chosen (line 23) according to a tradeoff

```
1  $\underline{\mathsf{UPOM}(s, \mathsf{stack}, d)}$:
2  if $\mathsf{stack} = \langle\rangle$ then return $U_{\mathsf{Success}}$
3  $(\tau, m, i, \mathit{tried}) \leftarrow \mathsf{top}(\mathsf{stack})$
5  if $d = 0$ then return $h(\tau, m, s)$
6  if $m = \mathit{nil}$ or $m[i]$ is a task $\tau'$ then
7      if $m = \mathit{nil}$ then $\tau' \leftarrow \tau$
8      if $N_{\mathsf{stack},s}(\tau')$ is not initialized yet then
10         $M' \leftarrow \mathsf{Applicable}(s, \tau') \setminus \mathit{tried}$
11         if $M' = 0$ then return $U_{\mathsf{Failure}}$
12         $N_{\mathsf{stack},s}(\tau') \leftarrow 0$
13         for $m' \in M'$ do
14             $N_{\mathsf{stack},s}(m') \leftarrow 0$ ; $Q_{\mathsf{stack},s}(m') \leftarrow 0$
15         end
16     end
17     $\mathit{Untried} \leftarrow \{m' \in M' | N_{\mathsf{stack},s}(m') = 0\}$
18     if $\mathit{Untried} \neq \emptyset$ then
20         $m_c \leftarrow$ random selection from $\mathit{Untried}$
21     end
23     else
           $m_c \leftarrow \mathrm{argmax}_{m \in M'} \{Q_{\mathsf{stack},s}(m) + C \times [\log N_{\mathsf{stack},s}(\tau) / N_{\mathsf{stack},s}(m)]^{1/2}\}$

25     $\lambda \leftarrow \mathsf{UPOM}(s, \mathsf{push}((\tau', m_c, 1, \emptyset), \mathsf{next}(\mathsf{stack}, s)), d - 1)$
27     $Q_{\mathsf{stack},s}(m_c) \leftarrow [N_{\mathsf{stack},s}(m_c) \times Q_{\mathsf{stack},s}(m_c) + \lambda] / [1 + N_{\mathsf{stack},s}(m_c)]$
28     $N_{\mathsf{stack},s}(m_c) \leftarrow N_{\mathsf{stack},s}(m_c) + 1$
29     return $\lambda$
30 end
31 if $m[i]$ is an assignment then
32     $s' \leftarrow$ state $s$ updated according to $m[i]$
33     return $\mathsf{UPOM}(s', \mathsf{next}(\mathsf{stack}, s'), d)$
34 end
35 if $m[i]$ is an action $a$ then
37     $s' \leftarrow \mathsf{Sample}(s, a)$
38     if $s' = \mathit{failed}$ then return $U_{\mathsf{Failure}}$
40     else   return $v(s, a, s') \oplus \mathsf{UPOM}(s', \mathsf{next}(\mathsf{stack}, s'), d - 1)$
41 end
```

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

between exploration and exploitation. The constant $C > 0$ fixes this tradeoff for the exploration less sampled methods (high C) versus the exploitation or more promising ones (low C).

$Q_{\sigma,s}(m)$, a global data structure, approximates $U^*(m, s, \sigma)$: it combines the value of a sampled action with the utility of the remaining part of a rollout (line 40), and it updates Q by averaging over previous rollouts (line 27). The value function v (line 40) is either v_e or v_s depending on the chosen utility function, efficiency or success ratio. For both function, $U_{\mathsf{Success}} = \mathbb{I}$ and $U_{\mathsf{Failure}} = 0$.

Now, let us see how Plan-with-UPOM can be used to refine the task explore(r, l) in Example 1. Since the task explore(r, l) has only one applicable refinement method, RAE will choose m1-explore(r, l) to refine explore(r, l). Once RAE encounters the subtask getTool(r), it is faced with two choices, either m1-getTool(r) or m2-getTool(r) to refine getTool(r). The search tree for this is shown in Figure 3.9. RAE calls Plan-with-UPOM online to refine getTool(r).

Figure 3.10 informally shows how the Q-values and the N counts are updated at each task node of the search tree, one rollout at a time, for the first three rollouts. For simplicity, assume that Plan-with-UPOM is minimizing cost. Say, in the first rollout (first call to UPOM), m1-getTool(r) is chosen to refine getTool(r) and the cost of the rollout is found to be 10. The Q-values of the corresponding visited nodes are updated to be 10. In the second rollout, since m2-getTool(r) hasn't been explored yet, UPOM will choose m2-getTool(r) to refine getTool(r). The cost of the rollout is computed to be 5, and the Q and N values are updated accordingly after the rollout is done. For the third rollout, UPOM chooses m2-getTool(r) using the UCB1 formula and computes the cost to be 6. After three rollouts, the Q-values vector for getTool(r) is $[10, 5.5]$ suggesting that the expected cost for m1-getTool(r) is 10 and for m2-getTool(r) is 5.5. So, Plan-with-UPOM will suggest m1-getTool(r) since the objective is to minimize cost.

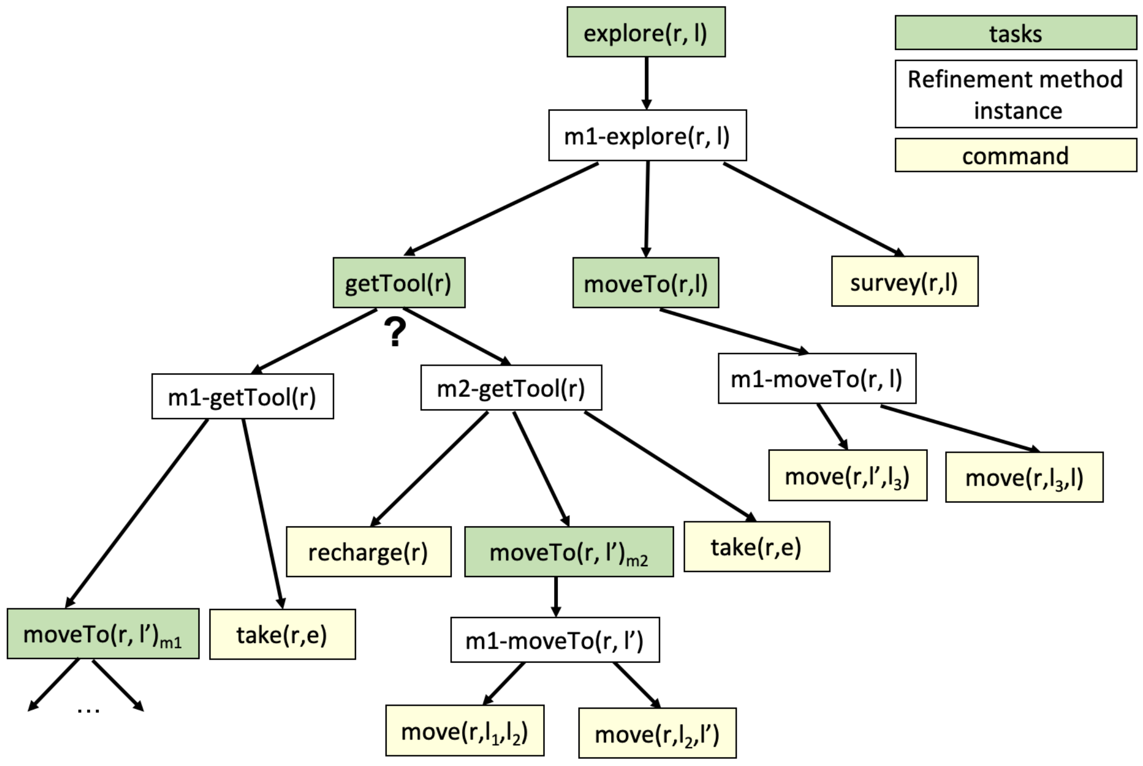
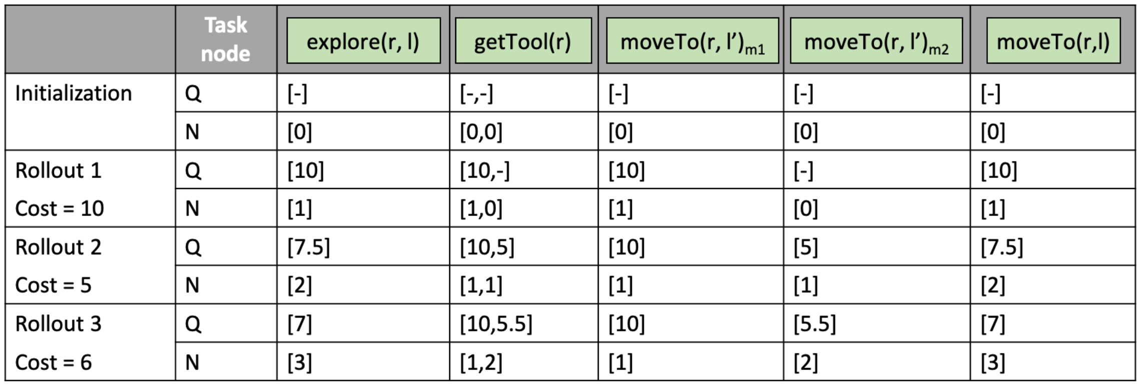


Figure 3.9: The sub-task getTool(r) can be refined using two different refinement methods leading to two possible refinement trees for the task explore(r, l).

A significant difference between the pseudocode in Algorithm 6 and Equation 3.11 is the restriction of Applicable to methods that have not been tried before by RAE for the same task. This is a conservative strategy, because at this point the actor has no means for distinguishing failures of tried methods that require retrials from those that don't. We'll come back to a retrial strategy in Chapter 6.

Another difference shows up in the initialization of stack in Select. This is



	Task node	explore(r, l)	getTool(r)	moveTo(r, l')$_{m1}$	moveTo(r, l')$_{m2}$	moveTo(r,l)
Initialization	Q	[-]	[-,-]	[-]	[-]	[-]
	N	[0]	[0,0]	[0]	[0]	[0]
Rollout 1	Q	[10]	[10,-]	[10]	[-]	[10]
Cost = 10	N	[1]	[1,0]	[1]	[0]	[1]
Rollout 2	Q	[7.5]	[10,5]	[10]	[5]	[7.5]
Cost = 5	N	[2]	[1,1]	[1]	[1]	[2]
Rollout 3	Q	[7]	[10,5.5]	[10]	[5.5]	[7]
Cost = 6	N	[3]	[1,2]	[1]	[2]	[3]

Figure 3.10: A table informally showing how the Q-values and N counts are updated for every task/subtask after each rollout (a call to UPOM) for the search tree shown in Figure 3.9.

explained by going back to how Select is used by RAE. At a root task τ, when Select is called the first time (line 9 of RAE), stack $= \langle(\tau, nil, 1, \emptyset)\rangle$. If RAE proceeds for τ with a method m returned by Select, at the next refinement call of RAE, e.g., for τ_1 (see Figure 3.8) Select needs to consider the utility of the methods for τ_1, but also their impact on the remaining steps in m, here on a_2 and τ_2. In other words, the actor requires the best method for τ_1 in the context of its current execution state, taking into account the remaining steps of the method m it is executing. This best method for τ_1 may be different from that given by Equation 3.12. The need to keep track of previously tried methods and pending tasks explains why stack is taken as a copy of the current stack in RAE for the root task at hand. However, this does not lead to reconsider previously made choices of methods the actor is currently executing, e.g., in Figure 3.8 m' is not reassessed. Note that UPOM does not pursue a rollout at an internal refinement node with the method maximizing the current utility evaluation Q, but with the best method according to the UCT exploration/exploitation tradeoff (line 23).

The two control parameters d_{max} and n_{ro} are dependent because of the following reason. The rational of UCT is that exploration should leave no untried alternatives down to the depth searched. For that, we should make sure that $n_{ro} > \mu$, where $\mu = \sum_{\tau_i} \max_s |Applicable(s, \tau_i)|$ over all subtasks τ_i, down to a refinement depth of the root task. But μ increases with d_{max}. In our experiments we keep a large constant n_{ro} and increase d in the progressive deepening loop until the max depth d_{max}. An alternative control of Select can be the following:

- for a given d, pursue the rollouts (line 13) until there are K successive exploitation rollouts, i.e., for which $Untried = \emptyset$, for some constant K;[3]
- pursue the progressive deepening loop (line 10) until no subtask is left unrefined for the K exploitation rollouts or until the search time is over.

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

Finally, let us discuss the important issue of the depth cutoff strategy. Two options may be considered: *(i)* d is the number of steps of a rollout (as in MDP algorithms), or *(ii)* d is the refinement depth of a rollout. The pseudocode in Algorithm 6 takes the former option: d decreases at every recursive call, for an action step as well as for a task refinement step. The advantage is that the cutoff at $d = 0$ stops the current evaluation. The disadvantage is that the root method, and possibly its refinements, are only partially evaluated. For example in Figure 3.8, if $j > d_{max}$, steps a_2 and τ_2 of m will never be considered; similarly for the remaining

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

steps in m_1: rollouts will go in deep refinements and never assess all the steps of evaluated methods. The value returned by UPOM can be arbitrarily far from U^*. The other issue of this strategy is that the heuristic estimate has to take into account remaining refinements lower down the cutoff point as well as remaining steps higher up in the refinement tree, i.e., what remains to be evaluated in stack.

In Option *(ii)*, where d is the refinement depth of a rollout, d decreases at a task refinement step only, not at an action step. The advantage is to allow each rollout to go through all the steps of every developed method. Furthermore, the heuristic estimate at a cutoff is focused in this case on a subtask and its applicable methods, whose simulation will not be started (nondeveloped methods). The disadvantage is that one needs an estimate of the state following the achievement of a task with a nondeveloped method in order to pursue the sibling steps. In Figure 3.8 with $d = 1$ for example, τ_1 will not be refined; a_2 and remaining steps of m will be based on an estimated state following the achievement of τ_1. The definition of a default state change following a task is domain dependent and might not be easily specified in general.

The modifications needed in UPOM to implement this option *(ii)* are the following:

- In order to be able to go back to higher levels of d when the simulation is pursued in parent methods after a cutoff, it is convenient to maintain d as part of the simulation stack: a fifth term d is added in every tuple of stack.

- The arguments of UPOM are modified according to the previous point.

- Line 5 in UPOM has to pursue the evaluation higher up in stack:

  if $d = 0$ **then** return $h(\tau, m, s) \oplus \mathsf{UPOM}(g(s, \tau, m), pop(\mathsf{stack}), b, k)$, where $g(s, \tau, m)$ is a default state after the achievement of τ with m in s.

For our experimental results (see Chapter 4), we have implemented a mixture of the two options: we take d as the refinement depth of a rollout (decreasing d at a task refinement step only), but we stop the evaluation when reaching $d = 0$, taking heuristic estimates for the remaining steps of pending methods. This has the disadvantage of a partial evaluation, but its advantages are to allow easily defined heuristic and not require a following state estimate.

3.6 Learning for RAE and UPOM

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

The classical techniques for domain independent heuristics in planning do not work for operational refinement models. Specifying by hand efficient domain-specific heuristics is not an acceptable solution. However, it is possible to learn such heuristics automatically by running UPOM offline in simulation over numerous cases. For this work we relied on a neural network approach, using both linear and rectified linear unit (ReLU) layers. However, we suspect that other learning approaches, e.g., SVMs, might have provided comparable results.

We developed two learning procedures to guide RAE and UPOM (Figure 3.1(b))

- Learnπ, learns a policy which maps a context defined by a task τ, a state s, and a stack σ, to a refinement method m in this context, to be chosen by RAE when no planning can be performed.
- LearnH, learns a heuristic evaluation function to be used by UPOM.

3.6.1 Learning to choose methods (Learnπ)

In a first approach, Learnπ learns a mapping from contexts to partially instantiated methods. A parameter of a method instance can inherit its value from the task at hand. However, different instances of a method may be applicable in a given state to the same task. This is illustrated in Example 1 by method m1-survey(l, r) where l is inherited from the task, but r can be instantiated as any robot such that status(r) = *free*. Learnπ simplifies the learning by abstracting all these applicable method instances to a single class. To use the learn policy, RAE chooses randomly among all applicable instances of the learned method for the context at hand. Learnπ learning procedure consists of the following four steps, which are schematically depicted in Figure 3.11.

Step 1: Data generation. Training is performed on a set of data records of the form $r = ((s, \tau), m)$, where s is a state, τ is a task to be refined and m is a method for τ. Data records are obtained by making RAE call the planner offline with randomly generated tasks. Each call returns a method instance m. We tested two approaches (the results of the tests are in Section 4.5):

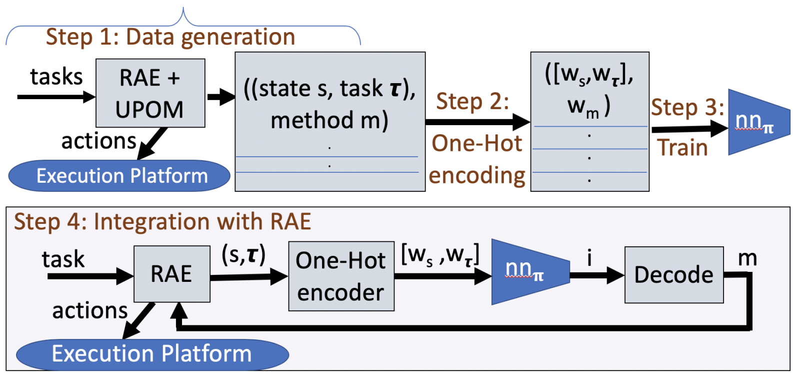


Figure 3.11: A schematic diagram for the Learnπ procedure.

- Learnπ-1 adds $r = ((s, \tau), m)$ to the training set if RAE succeeds with m in accomplishing τ while acting in a dynamic environment.
- Learnπ-2 adds r to the training set irrespective of whether m succeeded during acting.

Step 2: Encoding. The data records are encoded according to the usual requirements of neural net approaches. Given a record $r = ((s, \tau), m)$, we encode (s, τ) into an input-feature vector and encode m into an output label, with the refinement stack σ omitted from the encoding for the sake of simplicity.[4] Thus the encoding is

$$((s, \tau), m) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau], w_m), \tag{3.13}$$

with w_s, w_τ and w_m being One-Hot representations of s, τ, and m. The encoding uses an N-dimensional One-Hot vector representation of each state variable, with N being the maximum range of any state variable. Thus if every $s \in S$ has V state-variables, then s's representation w_s is $V \times N$ dimensional. Note that some

[4]Technically, the choice of m depends partly on σ. However, since σ is a program execution stack, including it would greatly increase the input feature vector's complexity, and the neural network's size and complexity.

information may be lost in this step due to discretization.

Step 3: Training. Our multi-layer perceptron (MLP) nn_π consists of two linear layers separated by a ReLU layer to account for non-linearity in our training data. To learn and classify $[w_s, w_\tau]$ by refinement methods, we used a SGD (Stochastic Gradient Descent) optimizer and the Cross Entropy loss function. The output of nn_π is a vector of size $|\mathcal{M}|$ where $\mathcal{M}$ is the set of all refinement methods in a domain. Each dimension in the output represents the degree to which a specific method is optimal in accomplishing τ.

Step 4: Integration in RAE. RAE uses the trained network nn_π to choose a refinement method whenever a task or sub-task needs to be refined. Instead of calling the planner, RAE encodes (s, τ) into $[w_s, w_\tau]$ using Equation 3.13. Then, m is chosen as

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

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

3.6.2 Learning to choose method instances (Learnπ_i)

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

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

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

$$((s, v_\tau), b) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_{v_\tau}], w_b), \tag{3.14}$$

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

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

Step 4: Integration in RAE. After RAE has chosen a refinement method m for task τ, we have RAE use the trained network $nn_{v_{ui}}$ to choose a value for each uninstantiated parameter v_{ui}. RAE encodes (s, v_τ) into $[w_s, w_{v_\tau}]$ using Equation 3.14. Then, the value for v_{ui}, b is chosen as

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

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

3.6.3 Learning a heuristic evaluation function (LearnH)

The LearnH procedure tries to learn an estimate of the utility u of accomplishing a task τ with a method m in state s. One difficulty with this is that u is a real number. In principle, an MLP could learn the u values using either regression or classification. To our knowledge, there is no rule to choose between the two; the best approach depends on the data distribution. Further, regression can be converted into classification when the range of the target values is finite. In our case, we don't need an exact utility value. We only need to compare candidate method instances. Experimentally, we observed that classification performed better than regression. We divided the range of utility values into K intervals. By studying the range and distribution of utility values, we chose K and the range of each interval such that the intervals contained approximately equal numbers of data records. LearnH learns to predict $interval(u)$, i.e., the interval in which u lies. The steps of LearnH are as follows (see Figure 3.12):

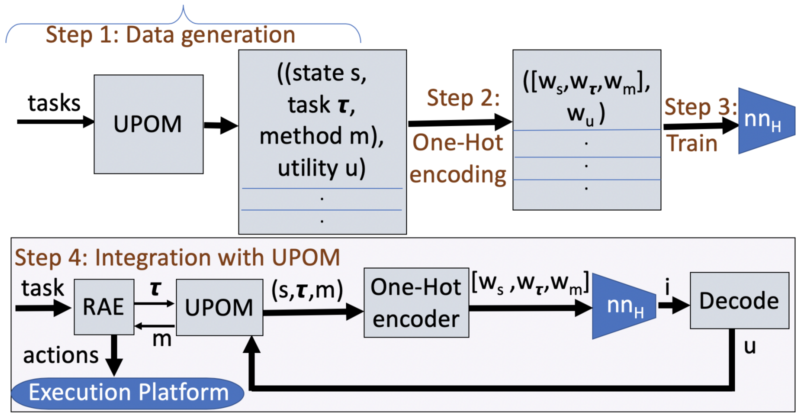


Figure 3.12: A schematic diagram for the LearnH procedure.

Step 1: Data generation. We generate data records in a similar way as in the Learnπ procedure, with the difference that each record r is of the form $((s, \tau, m), u)$ where u is the estimated utility value calculated by UPOM.

Step 2: Encoding. In a record $r = ((s, \tau, m), u)$, we encode (s, τ, m) into an input-feature vector using N-dimensional One-Hot vector representation, omitting σ for the same reasons as before. If $interval(u)$ is as described above, then the encoding is

$$((s, \tau, m), interval(u)) \overset{\text{Encoding}}{\longmapsto} ([w_s, w_\tau, w_m], w_u) \tag{3.15}$$

with w_s, w_τ, w_m and w_u being One-Hot representations of s, τ, m and $interval(u)$.

Step 3: Training. LearnH's MLP nn_H is the same as Learnπ's, except for the output layer. nn_H has a vector of size K as output where K is the number of intervals into which the utility values are split. Each dimension in the output of nn_H represents the degree to which the estimated utility lies in that interval.

Step 4: Integration in RAE. RAE calls the planner with a limited rollout length d, giving UPOM the following heuristic function to estimate a rollout's remaining utility:

$$h(\tau, m, s) \leftarrow Decode(\text{argmax}_i(nn_H([w_s, w_\tau, w_m])[i])),$$

where $[w_s, w_\tau, w_m]$ is the encoding of (τ, m, s) using Equation 3.15, and $Decode$ is a one-one mapping from a utility interval to its mid-point. Before the progressive deepening loop over calls to UPOM, Select initializes $\tilde{m}$ in line 8 according to this heuristic h.

3.6.4 Incremental online learning

RAE, in combination with UPOM, supports incremental online learning (although online learning has not been experimented with). The initialization can be performed either (i) without a heuristic by running RAE+UPOM online with $d_{max} = \infty$, or (ii) with an initial heuristic obtained from offline learning on simulated data. The online acting, planning and incremental learning is performed as follows:

- Augment the training set by recording successful methods and U values; train the models using Learnπ and LearnH with Z records, and then switch RAE to use either Learnπ alone when no search time is available, or UPOM with current heuristic h and finite d_{max} when planning time available.
- Repeat the above steps every X runs (or on idle periods) using the most recent Z training records (for Z about a few thousands) to improve the learning on both LearnH and Learnπ.

3.7 Properties of Plan-with-UPOM

In this section, we derive the time and space complexities of Plan-with-UPOM and show how UPOM can be mapped to an MDP. The proof of mapping UPOM to an MDP was done in collaboration with Dana Nau, Malib Ghallab, Paolo Traverso and James Mason. The credit for the details of the mapping proof goes to the dissertation advisor, Dana Nau.

Theorem 5. *Assuming that commands can be simulated in $O(1)$ time, the time complexity of* Plan-with-UPOM (s, σ, d) *is* $O(n_{ro}d \times \max_{\tau,s'} |Applicable(\tau, s')| + f(\sigma, s))$, *where* $f(\sigma, s)$ *is the time complexity of calculating next steps in the current refinement stack* σ *in state* s.

Proof. Plan-with-UPOM does n_{ro} rollouts. Each rollout has length d and looks at d steps in the current refinement stack σ. Let us first calculate the time complexity of a single step of a rollout. A step of a rollout takes a different amount of time depending on whether it is a task, an assignment or a command. We assume that a command can be simulated in $O(1)$ time. Same is true for an assignment step. For a task τ, Plan-with-UPOM looks at the set of applicable refinment methods for τ in current state s, $Applicable(\tau, s)$. So, the time complexity of a step is $max_{\tau,s'}|Applicable(\tau, s')|$ in the worst case. The only thing remaining in this analysis is the time required to calculate the next step which depends on the refinment stack and the procedural code present inside the refinement methods. We name this function as $f(\sigma, s)$. $f(\sigma, s)$ is the total time required for calculating next steps Next(σ, s) in n_{ro} rollouts. Therefore, the time complexity of Plan-with-UPOM is

$$O(n_{ro}d \times \max_{\tau,s'} |Applicable(\tau, s')| + f(\sigma, s)).$$

□

Remark 1. *The time complexity of* Plan-with-UPOM *is linear wrt the number of* *rollouts, n_{ro} and the rollout length, d.*

Corollary 2. *The space complexity of* Plan-with-UPOM (s, σ, d) *is*

$$O(n_{ro}d \times \max_{\tau,s'} |Applicable(\tau, s')|).$$

Proof. Plan-with-UPOM saves some metadata in every node it expands. Based on this meta-data, it chooses the refinement method instance with the highest expected utility (highest Q-value). With n_{ro} rollouts each of length d, the maximum number of unique nodes created is $O(n_{ro}d)$. Each node stores Q-values for the applicable method instances for the current task to be refined. The maximum number of applicable method instances is $\max_{\tau,s'} |Applicable(\tau, s')|$. So, the space complexity of Plan-with-UPOM is $O(n_{ro}d \times \max_{\tau,s'} |Applicable(\tau, s')|)$. □

3.7.1 Mapping UPOM's Search Space to an MDP

We demonstrate the asymptotic convergence of UPOM towards an optimal method on static domains, i.e., domains without exogenous events. UPOM is based on UCT, which is demonstrated to converge on a finite horizon MDP with a probability of not finding the optimal action at the root node that goes to zero at a polynomial rate as the number of rollouts grows to infinity (Theorem 6 of [46]).

To simplify the mapping, we first consider UPOM with an additive utility function, and show how to map the search space of UPOM into an MDP. We then discuss how this can be extended to the efficiency and success ratio utility functions defined in 3.5, since the UCT algorithm is not restricted to the additive case; it still converges as long as the utility function is monotonic.

3.7.2 Search Space for Refinement Planning

Let $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, \mathcal{A})$ be an acting domain, as specified at the end of Section 3.1. Throughout this proof, we will assume that Σ is static.

Recall from 3.5 that the space searched by UPOM is a simulated version of Σ. To talk about this formally, let's say that a *refinement planning domain* is a tuple $\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$, where S is the set of states (recall that these are abstractions of states in Ξ), and $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are the same as in Σ. Recall from Section 3.1 that Ξ (and thus S), $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{A}$ are all finite, and that every sequence of steps generated by the methods in $\mathcal{M}$ is finite. [5]

For $s \in S$ and $a \in \mathcal{A}$, we let $\gamma(s, a) \subseteq S$ be the set of all states that may be produced by simulating a's execution in s. For each $s' \in \gamma(s, a)$, we let $T(s, a, s')$ be the probability that state s' will be produced if we simulate a's execution in state s.

Recall from Section 3.2 that a *refinement stack* is a LIFO stack in which each element is a tuple $(\tau, m, i, \mathit{tried})$, where τ is a task, m is a method, i is an instruction pointer that points to the i'th line of m's body (which is a computer program), and *tried* is the set of methods previously tried for τ. We will call the tuple $(\tau, m, i, \mathit{tried})$ a *stack frame*, and we will let $m[i]$ denote the i'th line of the body of m.

We now can define a *refinement planning problem* to be a tuple $\Pi =$

[5]One way to enforce such a restriction would be as follows. For each iteration loop, one could require it to have a loop counter that will terminate it after a finite number of iterations. For recursions, one could use a *level mapping* (e.g., see [67, 68]) that assigns to each task t a positive integer $\ell(t)$, and require that for every method m whose task is t and every task t' that appears in the body of m, $\ell(t') < \ell(t)$. However, in most problem domains it is straightforward to write a set of methods that don't necessarily satisfy this property but still don't produce infinite recursion.

(Φ, s_0, σ_0, U), where s_0 is the initial state, σ_0 is the initial refinement stack, and U is a utility function.

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

$$\rho = \langle(\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n)\rangle \tag{3.16}$$

satisfying the following properties:

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

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

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

$$\sigma_0 = \langle(\tau_0, m_0, 1, \varnothing)\rangle, \tag{3.17}$$

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

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

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

such that σ_0 satisfies Equation 3.17 and $\sigma \in \{\sigma_0, \ldots, \sigma_n\}$. We let $\mathcal{R}(\Phi)$ be the set of all refinement stacks that are reachable in Φ. Since every sequence of steps generated by the methods in $\mathcal{M}$ is finite, it follows that $\mathcal{R}(\Phi)$ is also finite.

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

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

$$\begin{aligned} t_\rho = \{(s_j, m_j[i_j], s_{j+1}) \mid (\sigma_j, s_j) &\text{ and } (\sigma_{j+1}, s_{j+1}) \text{ are members of } \rho, \\ &(\tau_j, m_j, i_j, \mathit{tried}_j) = \mathsf{top}(\sigma_j), \text{ and } m_j[i_j] \text{ is an action}\}. \end{aligned} \tag{3.18}$$

Thus if U is additive, then

$$U(\rho) = \begin{cases} \sum_{(s,a,s') \in t_\rho} R(s'), & \text{if } U \text{ is the sum of rewards,} \\ \sum_{(s,a,s') \in t_\rho} C(s,a,s'), & \text{if } U \text{ is the sum of costs.} \end{cases} \tag{3.19}$$

Defining the MDP

We want to define an MDP Ψ such that choosing among methods in Φ corresponds to choosing among actions in Ψ. The easiest way to do this is to let all of Φ's actions and methods be actions in Ψ. Based loosely on the notation in [69], we will write Ψ as

$$\Psi = (S^\Psi, \mathcal{A}^\Psi, s_0^\Psi, S_g^\Psi, \gamma^\Psi, T^\Psi, U^\Psi) \tag{3.20}$$

where

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

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

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

$$\text{Applicable}^\Psi((\sigma, s)) = \begin{cases} \mathsf{Instances}(\mathcal{M}, m[i], s), & \text{if } m[i] \text{ is a task,} \\ \{m[i]\}, & \text{if } m[i] \text{ is an action.} \end{cases} \tag{3.21}$$

Thus if $a \in \text{Applicable}^\Psi((\sigma, s))$, then there are two cases for what $\gamma^\Psi(s, a)$ and $T^\Psi(s, a, s')$ might be:

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

$$\gamma((\sigma, s), a) = \{(\phi + \sigma, s)\};$$
$$T^\Psi[(\sigma, s), a, (\phi + \sigma, s)] = 1;$$
$$T^\Psi[(\sigma, s), a, (\sigma', s')] = 0, \quad \text{if } (\sigma', s') \neq (\phi + \sigma, s).$$

- Case 2: $m[i]$ is an action in $\mathcal{A}$, and $a = m[i]$. Then a's possible outcomes in Ψ correspond one-to-one to its possible outcomes in Φ. More specifically, if γ is the state-transition function for Φ (see Section 3.1), then

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

and

$$T^\Psi((\sigma, s), a, (\sigma', s'))) = \begin{cases} T(s, a, s'), & \text{if } (\sigma', s') \in \gamma^\Psi((\sigma, s), a), \\ 0, & \text{otherwise.} \end{cases}$$

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

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

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

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

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

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

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

If U is additive, then so is U^Ψ. In this case, Ψ satisfies the definition of an MDP with initial state (see [69]).

Mapping UPOM's Search to an Equivalent UCT Search

Let

$$\Pi = (\Phi, s_0, \sigma_0, U) \tag{3.22}$$

be a refinement planning problem, where

$$\Phi = (S, \mathcal{T}, \mathcal{M}, \mathcal{A}). \tag{3.23}$$

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

$$\rho = \langle(\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n)\rangle, \tag{3.24}$$

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

$$p = p_1 \times \ldots \times p_n, \tag{3.25}$$

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

We want to show that UPOM's search of Π corresponds to an equivalent UCT search of Ψ. Theorem 6 accomplishes this in the case where the utility function U is additive. After the theorem, we discuss the case where U is not additive.

Theorem 6. *Let Π, Φ, ρ and p be as in Equations 3.22–3.25, and let U be additive. Let UPOM's metadata for Π be as described above. Let $\Psi = (S^\Psi, \mathcal{A}^\Psi, \gamma^\Psi, T^\Psi, U^\Psi)$ be the MDP corresponding to Π. If UCT searches Ψ using the same metadata that*

UPOM used, then the probability that UCT generates the rollout

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

is the same probability $p = p_1 \times \ldots \times p_n$ as in Equation 3.25.

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

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

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

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

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

- Case 1: m_1 is a method in Φ. As shown in Algorithm 6, $\mathsf{UPOM}(\Phi, s_0, \sigma_0, U)$

chooses m_1 using the same UCB-style computation that a UCT search in Ψ would use at (σ_0, s_0). Thus, omitting the details about how to compute p_1 from the metadata, it follows that if $\mathsf{UPOM}(\Phi, s_0, \sigma_0, U)$ chooses m_1 with probability p_1, then so does the UCT search.

- Case 2: m_1 is an action in Φ. Then UPOM's computation (in line 40 through the end of Algorithm 6) is *not* a UCT-style computation, but this does not matter, because there is only one possible choice, namely m_1. In this case, UPOM's probability of choosing m_1 is $p_1 = 1$, and the same is true for the UCT search.

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

This concludes the sketch of the proof. □

Generalizing beyond MDPs If the utility function U is not additive, Equation 3.20 produces a probabilistic planning problem that looks similar to an MDP, the only difference being that the utility function U^Ψ is not additive. Furthermore, Theorem 6 still holds even when U is not additive, if we modify the proof to remove the claim that Ψ is an MDP.

We note that the UCT algorithm [46] is not restricted to the case where U^Ψ is additive; it will still converge as long as U^Ψ is monotonic. If U is monotonic, then so is U^Ψ. In this case it follows that UCT—and thus UPOM—will converge to an optimal solution. In particular, UPOM will converge to an optimal solution when using the *efficiency* and *success ratio* utility functions in Section 3.5.1.

3.8 Summary

In this chapter, we described our hierarchical operational model formalism, the acting algorithm RAE, and three refinement planning algorithms: APEplan; RAEplan, a SLATE-like planner; and Plan-with-UPOM, a planner based on a UCT-like procedure UPOM. We proved theoretical properties of RAEplan and UPOM under certain sets of assumptions. We also showed how RAE and UPOM can be integrated with learning via three different learning strategies, Learnπ, LearnH and Learnπ_i. Learnπ helps to choose the best refinement method, LearnH learns to estimate a heuristic evaluation function, and Learnπ_i learns refinement method instances. Recall that a refinement method instance is a method with values assigned to all uninstantiated parameters.

Chapter 4: Implementation and Experimental Evaluation

In this chapter, we describe the experimental setup and evaluation of our three refinement planning algorithms: APEplan, RAEplan, and UPOM when run in combination with RAE. For each of them, we first describe the simulated test domains. The domains have various properties which include sensing, agent collaboration, dead ends, dynamic events and concurrent tasks. The performance is evaluated by varying the parameters of the planning algorithms and measuring three performance metrics, called efficiency, success-ratio and retry ratio. For UPOM, the performance is also evaluated for the integration of three learning strategies: Learnπ, LearnH and Learnπ_i.

4.1 Evaluation of APEplan

4.1.1 Domains

We have implemented and tested APEplan on four simulated 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 RAE is sufficient for achieving the tasks, but not efficiently. One of our domains illustrates sensing (or information gathering) actions, three involve (centrally controlled) collaboration between actors. All domains have dynamic events and concurrent tasks (see Table 4.1).

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 soil samples, etc. 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 consists of several robots. They 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, 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. 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 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, 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 together, 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 get damaged and need repair before being used again. But there are no dead ends.

These four domains have different properties, as summarized in Table 4.1. 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 sub tasks (movement of objects to the required locations) is handled inside the re-

Table 4.1: Properties of the four test domains of APEplan.

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

finement methods. The Chargeable Robot domain and the Explorable Environment domain are domains that have dead ends, whereas the Spring Door domain and the Industrial Plant do not have them.

4.1.2 Assessment of APEplan's parameters

The objective of our experiments was to examine how RAE'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 RAE'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 we used simulated versions of the four environments, that ran on a 2.6 GHz Intel Core i5 processor.

The amount of planning done by APEplan 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 RAE 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 RAE in a purely reactive way with no planning at all. Our objective function for the experiments is the number of commands in the plan.

Hypothesis 1. *Increasing the value of RAE'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.*

Success ratio. Figure 4.1 plots *success ratio*, the proportion of jobs that RAE successfully accomplished in each domain. For the two domains with dead ends (Chargeable Robot domain and Explorable Environment domain), 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 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 domain, it made almost no difference at all. In the Spring Door domain 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

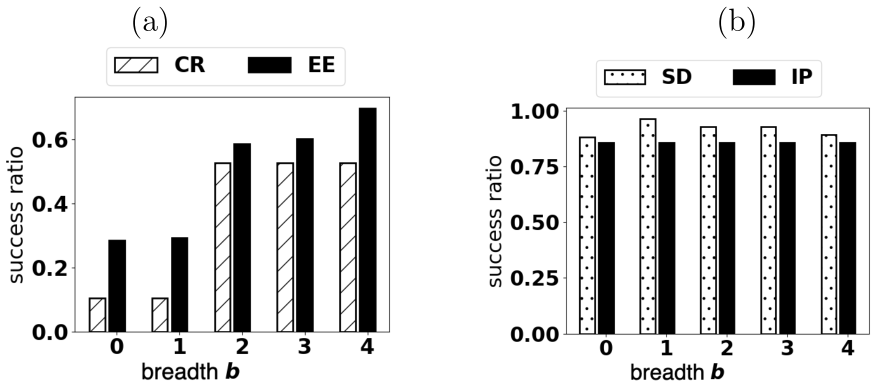


Figure 4.1: Success ratio (number of successful jobs/ total number of jobs) for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) and (b) domains having no dead ends (Spring Door domain and Industrial Plant domain). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

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 4.2 plots results for a second measure *retry ratio*, or the number of times that RAE 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 RAE chose for some task τ (see Algorithm 1). Retry works by trying to use another applicable method instance for τ that it has not tried already. Although thisis 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

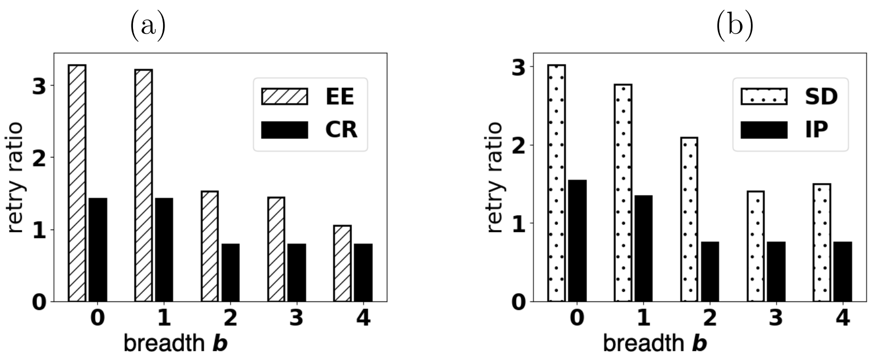


Figure 4.2: Retry ratio (number of retries / total number of jobs) for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) and (b) domains having no dead ends (Spring Door domain and Industrial Plant domain). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

expense.

In all four domains, the retry ratio decreased slightly from $b = 0$ (purely reactive RAE) to $b = 1$, and it generally decreased more as b increased. This is because higher values of b made APEplan 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 Chargeable Robot domain domain, the large decrease in retry ratio from $b = 1$ to $b = 2$ corresponds to the increase in success ratio observed in Figure 4.1. The same is true for the Explorable Environment domain 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 RAE 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 APEplan's and RAE'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 4.3 shows how the average value of ν depends on b. In the domains with dead ends (Chargeable Robot domain and Explorable Environment domain), 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 4.1 and 4.2). This is because of the extra time overhead of running APEplan with higher search breadth.

In summary, for domains with dead ends, planning with APEplan outperforms the purely reactive version of RAE. The same results occur in the domains without

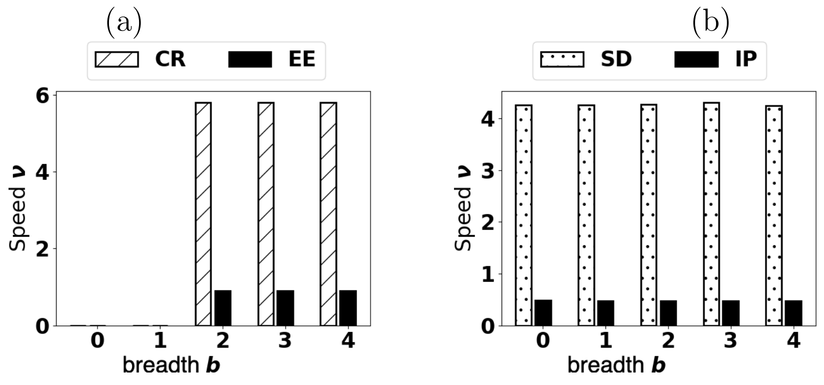


Figure 4.3: Speed to success ν averaged over all of the jobs, for different values of search breadth b of APEplan for (a) domains having dead ends (Chargeable Robot domain and Explorable Environment domain) and (b) domains having no dead ends (Spring Door domain and Industrial Plant domain). CR = Chargeable Robot, EE = Explorable Environment, SD = Spring Door, IP = Industrial Plant.

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.

4.2 Evaluation of RAEplan

4.2.1 Domains

Our test domains for evaluating RAEplan are similar to the ones used for evaluating APEplan with some feature enhancements.

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

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

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

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

Table 4.1 summarizes the different properties of these domains. CR includes a model of a *sensing* action that a robot can use to identify what objects are at a given location. SD models a situation where robots need to collaborate, and can ask for help from each other. EE models a combination of robots with different capabilities (UGVs and UAVs) whereas in the other three domains all robots have same capabilities. It also models collaboration. In the IP domain, the allocation of tasks among the robots is hidden from the user. The user just specifies their orders; the delegation of the sub-tasks (movement of objects to the required locations) is handled inside the refinement methods. CR and EE can have dead-ends, whereas SD and IP do not have dead-ends.

4.2.2 Assessment of RAEplan's parameters

To examine how RAE's performance might depend on the amount of planning with RAEplan, we created a suite of test problems for the four domains described in Section 4.1.1. Each test problem consists of a job to accomplish, that arrives at a randomly chosen time point in RAE's input stream. For each such time point, we chose a random value and held it fixed throughout the experiments.

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

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

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

Efficiency. Figures 4.4 and 4.5 show how the average efficiency E depends on b and k. We see that efficiency increases with increase in b and k as expected. This is true for all four domains. In the CR domain, efficiency increases considerably as we move from $b = 1$ to $b = 2$, then (specifically when $k = 3$ and 5) decreases slightly as we move to $b = 3$. This is possibly because the commands present in the third method require more sampling to make accurate predictions. Indeed, with more samples, $k = 7$ and 10, $b = 3$ has better efficiency than $b = 2$. In the EE domain, we see that the efficiency improves up to $k = 5$ and then remains stable, indicating that 5 samples are enough for this domain. In the domains without dead ends (SD and IP), we see a gradual increase in efficiency with k. In Figure 2, the large increase in efficiency between $b = 1$ and $b = 2$ (as opposed to a more uniform increase) is because RAEplan explores methods according to a preference ordering specified by the domain's author. For many of the problems in our test suite, the 2^{nd} method in the preference ordering turned out to be the one with the largest expected efficiency. These experiments confirm our expectation that efficiency improves with b and k.

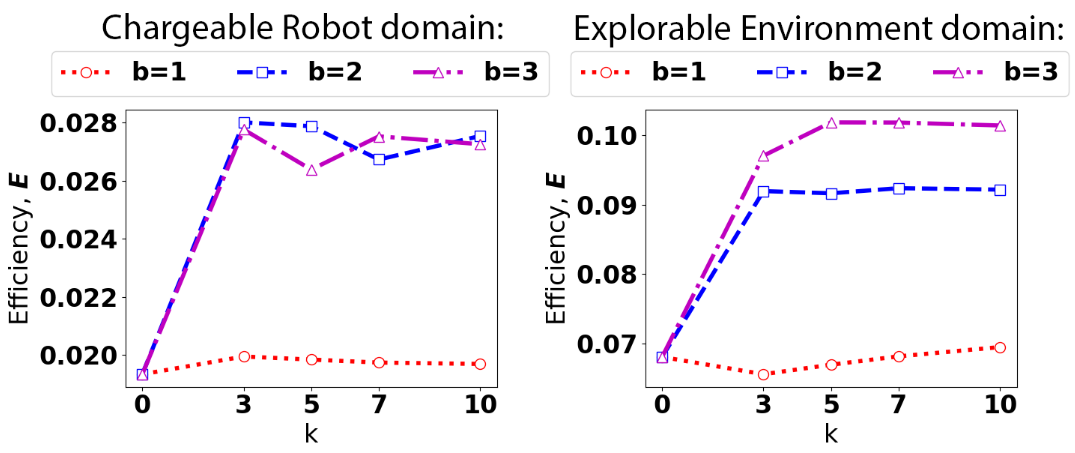


Figure 4.4: Efficiency E averaged over all of the jobs, for various values of b and k of RAEplan in domains with dead ends.

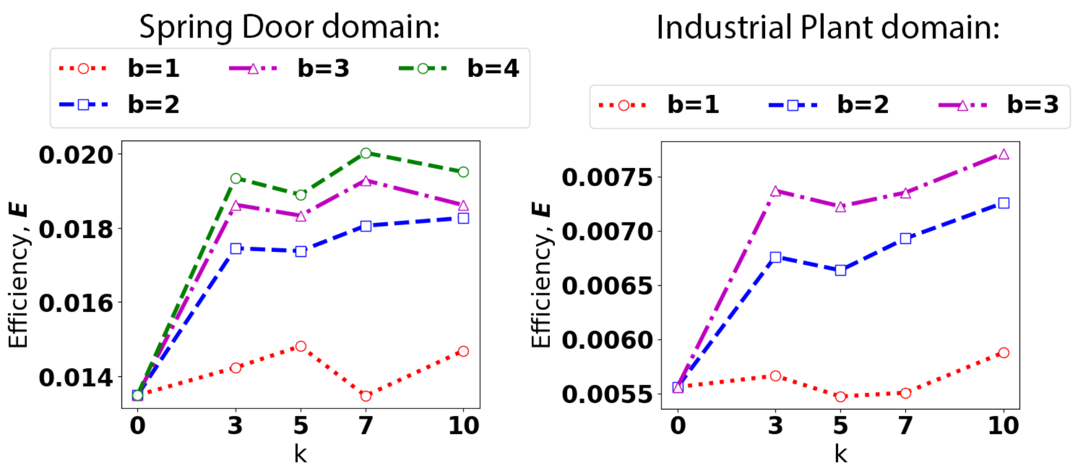


Figure 4.5: Efficiency E averaged over all of the jobs, for various values of b and k of RAEplan in domains *without* dead ends.

Success ratio. We wanted to assess how robust RAE was with and without planning. Figures 4.6 and 4.7 show RAE's *success ratio*, i.e., the proportion of jobs successfully accomplished in each domain. For the domains with dead ends (CR and EE), the success ratio increases as b increases. However, in the CR domain, there is some decrease after $k = 3$ because we are optimizing efficiency, not robustness. Formulating an explicit robustness criterion is non-trivial and will require further work. For the success ratio experiments, when we say we're not optimizing

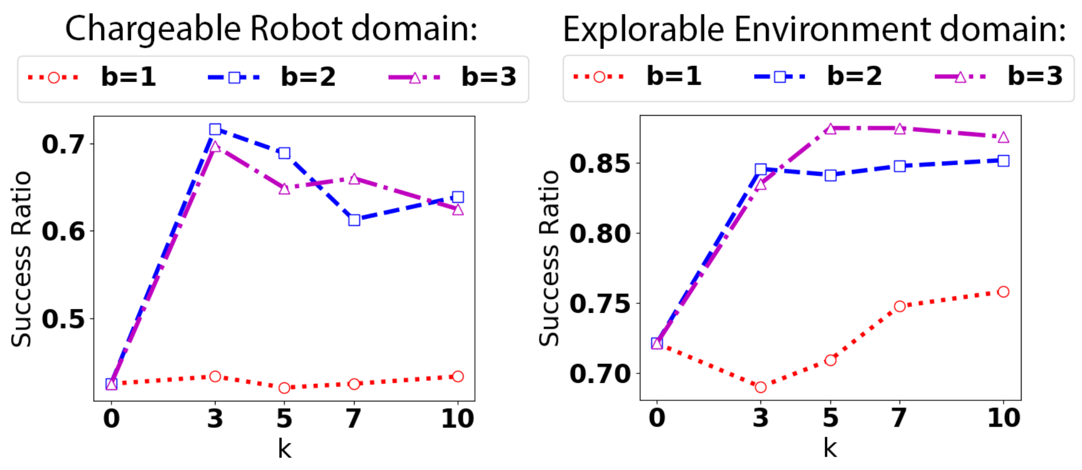


Figure 4.6: Success ratio (# of successful jobs/ total # of jobs) for various values of b and k of RAEplan in domains with dead ends.

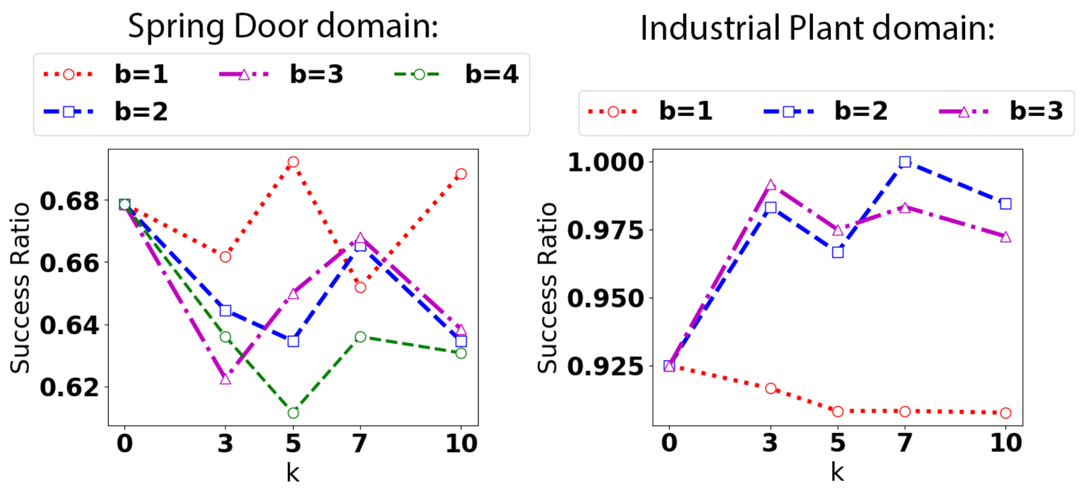


Figure 4.7: Success ratio (# of successful jobs/ total # of jobs) for various values of b and k of RAEplan in domains *without* dead ends.

robustness, we mean we're not optimizing a specific criterion that leads to better recovery if an unexpected event causes failure. RAEplan looks for the most efficient plan. In our efficiency formula in Eqs. (2,3), a plan with a high risk of failure will have low efficiency, but so will a high-cost plan that always succeeds.

In the SD domain, b or k didn't make very much difference in the success ratio. In fact, for some values of b and k, the success ratio decreases. This is because in our preference ordering for the methods of the SD domain, the methods appearing earlier are better suited to handle the events in our problems whereas the methods appearing later produce plans that have lower cost but less robust to unexpected events. In the IP domain, we observe that success ratio increases with increase in b and k.

Retry ratio. Figures 4.8 and 4.9 shows the *retry ratio*, i.e., the number of times that RAE had to call the Retry procedure, divided by the total number of jobs to accomplish.

The Retry procedure is called when there is an execution failure in the method instance m that RAE choses for a task τ. Retry tries another applicable method instance for τ that it hasn't tried already. This is significantly different from backtracking since the failed method m has already been partially executed; it has changed the current state. In real-world execution there is no way to backtrack to a previous state. In many application domains it is important to minimize the total number of retries, since recovery from failure may incur significant, unbudgeted amounts of time and expense.

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

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

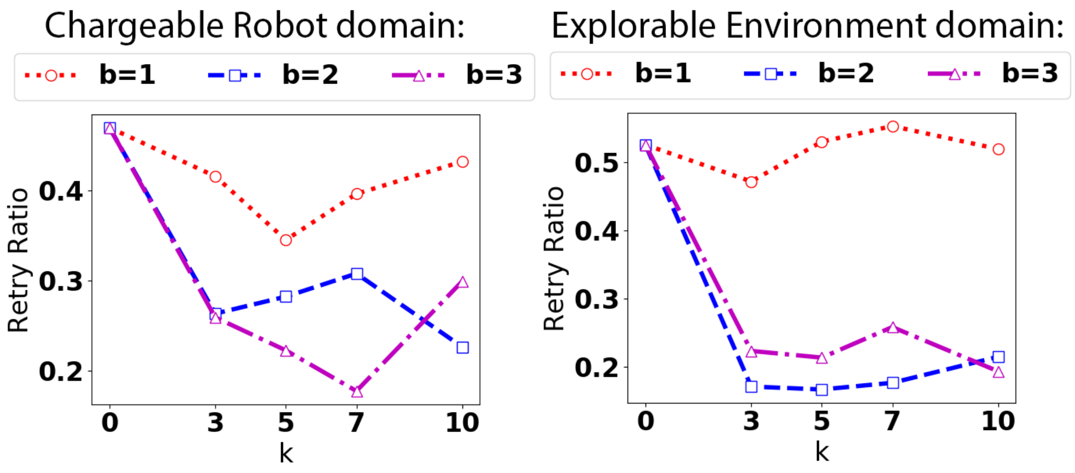


Figure 4.8: Retry ratio (# of retries / total # of jobs) for various values of b and k of RAEplan in domains with dead ends.

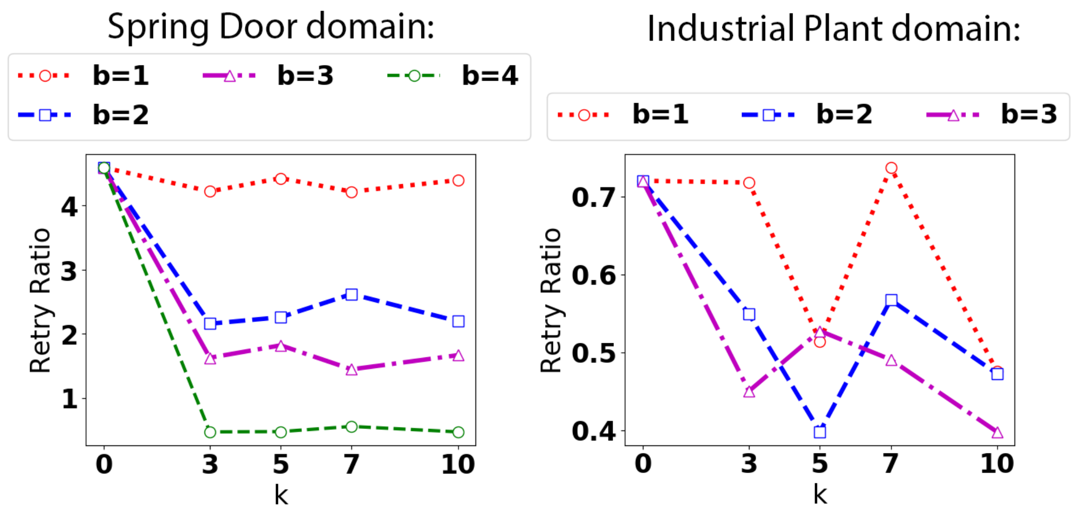


Figure 4.9: Retry ratio (# of retries / total # of jobs) for various values of b and k of RAEplan in domains *without* dead ends.

4.3 Evaluation of Plan-with-UPOM

4.3.1 Domains

We have implemented and tested our framework on five domains which illustrate service and exploration robotics scenarios with aerial and ground robots.

The S&R domain extends the search and rescue setting of Example 3 with several UAVs surveying a partially mapped area and finding injured people in need of help. UGVs gather supplies, such as medicines, and go to rescue the localized persons. Exogenous events are weather conditions and debris in paths.

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

In the Fetch domain, several robots are collecting objects of interest. The robots are rechargeable and may carry the charger with them. They can't know where objects are, unless they do a sensing action at the object's location. They must search for an object before collecting it. A task reaches a dead end if a robot is far away from the charger and runs out of charge. While collecting objects, robots may have to attend to some emergency events happening in certain locations.

The Nav domain has several robots trying to move objects from one room to another in an environment with a mixture of spring doors (which close unless they're held open) and ordinary doors. A robot can't simultaneously carry an object and hold a spring door open, so it must ask for help from another robot. A free robot can be the helper. The type of each door isn't known to the robots in advance.

The Deliver domain was developed by James Mason as a part of his Undergraduate Honors project at University of Maryland. It has several robots in a shipping warehouse that must co-operatively package incoming orders, i.e., lists of items of different types and weights to deliver to customers. Items for a single order have be placed in a machine, which packs them together; packages have to be placed in the shipping doc. To process multiple orders concurrently, items can be moved to a pallet before transfer to a machine. Robots have limited capacities.

S&R, Explore, Nav and Fetch have sensing actions. S&R, Explore, Fetch and Deliver can have dead-ends. The features of these domains are in Table 4.2. Please recall from Section 3.1 that $\mathcal{M}$ is the set of refinement methods, and $\mathcal{M}_i$ is the set of refinement method instances. The full descriptions of the operational models are in Appendix B.

Domain	$\|\mathcal{T}\|$	$\|\mathcal{M}\|$	$\|\mathcal{M}_i\|$	$\|\mathcal{A}\|$	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
S&R	8	16	16	14	✓	✓	✓	✓	✓
Explore	9	17	17	14	✓	✓	✓	✓	✓
Fetch	7	10	10	9	✓	✓	✓	–	✓
Nav	6	9	15	10	✓	–	✓	✓	✓
Deliver	6	6	50	9	✓	✓	–	✓	✓

Table 4.2: Features of the five test domains of RAE + UPOM

4.3.2 Assessment of UPOM's parameters

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

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

Note that the measured efficiency takes into account the execution context with concurrent tasks and exogenous events; hence it is different for the corresponding utility function optimized in UPOM (i.e., the expected efficiency of Equation 3.11); similarly for the success ratio. We used a 2.8 GHz Intel Ivy Bridge processor. The cut-off time for a run was set to 30 minutes.

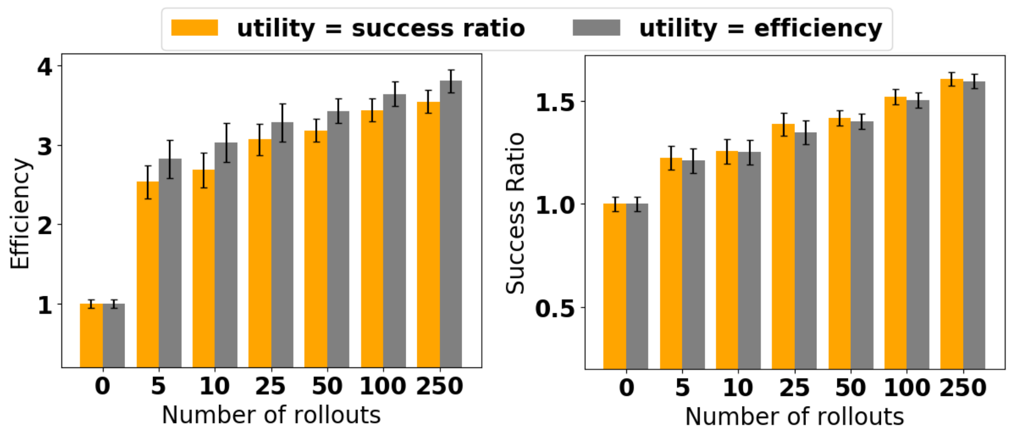


Figure 4.10: Efficiency and success ratio for two different utility functions of Plan-with-UPOM (orange is expected success ratio and gray is expected efficiency) averaged over all five domains, with $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

4.3.3 Comparison of the two utility functions

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

Figure 4.10 shows the measured efficiency and success ratio of RAE for the two utility functions, averaged over all domains. Each data point is the average of 10^4 runs, with the error bars showing 95% confidence interval; we plot relative values with respect the base case of U for $n_{ro} = 0$. As expected, the measured efficiency is higher when the optimized utility function of UPOM is the expected efficiency. Similarly for the success ratio. However, optimizing one criteria has also a good effect on the other one, since the two are not independent. We also observe that 5 rollouts have already a significant effect on the efficiency, with slight improvements as UPOM does more rollouts. In contrast, the success-ratio increases smoothly from no planning to planning with 250 rollouts. This can be due to the difference between the two criteria: a task that succeeds in its first attempt and a task that succeeds after several retries of RAE have both a success-ratio of 1, but the efficiency in the latter case is lower. This point is analyzed next.

4.3.4 Retry ratio

Figure 4.11 shows the *retry ratio*, i.e., the number of calls to Retry, divided by the total number of tasks. Recall that when a chosen method fails, Retry tries another applicable method instance that hasn't been tried already. The retry ratio measures the execution effectiveness. Performing many retries is not desirable, since this has a high cost and faces the uncertainty of execution. We observe that the retry ratio drops sharply from purely reactive RAE to calling UPOM with 5 rollouts. From then onwards, until 250 rollouts, the retry ratio continues to decrease gradually. The behavior is similar in all domains, so we have combined the results together to show the average values in a single plot.

4.3.5 Efficiency across domains

In Figure 4.12 we detail for each domain the measured efficiency of RAE when the utility of UPOM was set to expected efficiency, for varying n_{ro} and $d_{max} = \infty$. Each data point is the average of 2500 runs. We observe that the efficiency generally

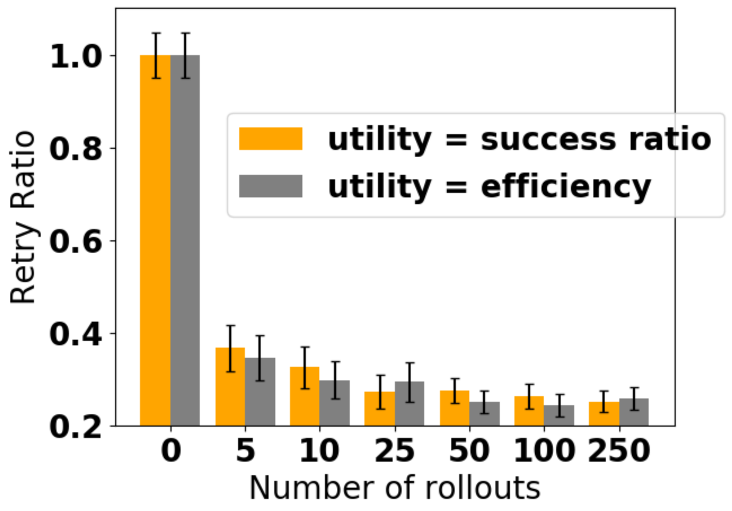


Figure 4.11: Retry ratio (# of retries / total # of jobs) averaged over all five domains, for Plan-with-UPOM with $d_{max} = \infty$.

improves with the number of rollouts. However, there is not much improvement with increase in n_{ro} in the Fetch domain, and in the Deliver domain, the efficiency drops slightly when $n_{ro} = 250$. We conjectured that this can be due to concurrent interfering tasks. Hence, we measured for Fetch and Deliver domains the efficiency for test cases with only one root task; the results in Figure 4.13 confirmed this conjecture.

4.3.6 Success ratio across domains

Figure 4.14 shows for each domain the measured success ratio of RAE when the utility of UPOM was set to expected success ratio, for varying n_{ro} and $d_{max} = \infty$. The success-ratio generally increases with increase in the number of rollouts. Again, a slight drop is observed in the Deliver domain. Figure 4.15 shows that for test cases with only one root task the success-ratio improves in the Fetch domain, and remains constant in the Deliver domain. The success ratio remains 1 in the Deliver domain

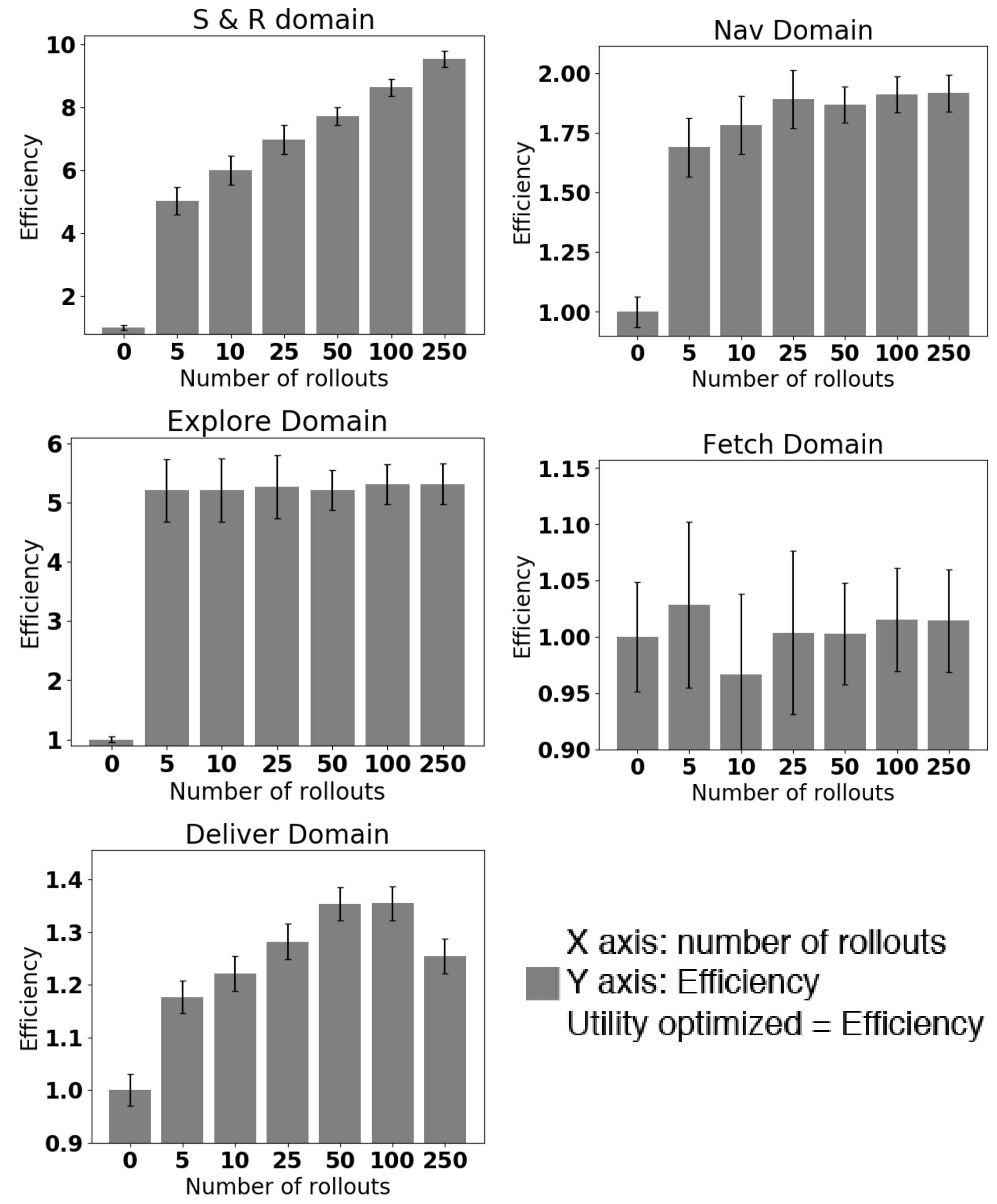


Figure 4.12: Measured efficiency of RAE with Plan-with-UPOM for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

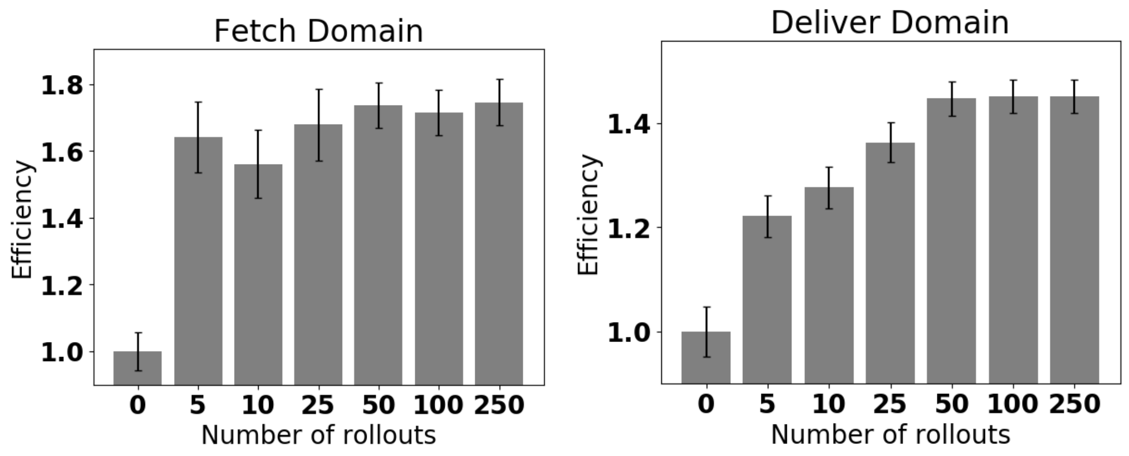


Figure 4.13: Measured efficiency averaged over only test cases with one root task, in Fetch and Deliver domains with Plan-with-UPOM's parameter, $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

because all test cases with one root task succeed eventually, with or without retries. In the domains with dead ends, the improvement in success ratio is more substantial than domains without dead ends because planning is more critical for cases where one bad choice of refinement method can lead to permanent failure.

4.3.7 Depth and Heuristics

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

- $h0$ always returns ∞,
- hD a hand written domain specific heuristic; and
- hLearnH the heuristic function learned by the LearnH procedure (Section 3.6.3).

The results, in Figure 4.16, show that the efficiency generally increases with

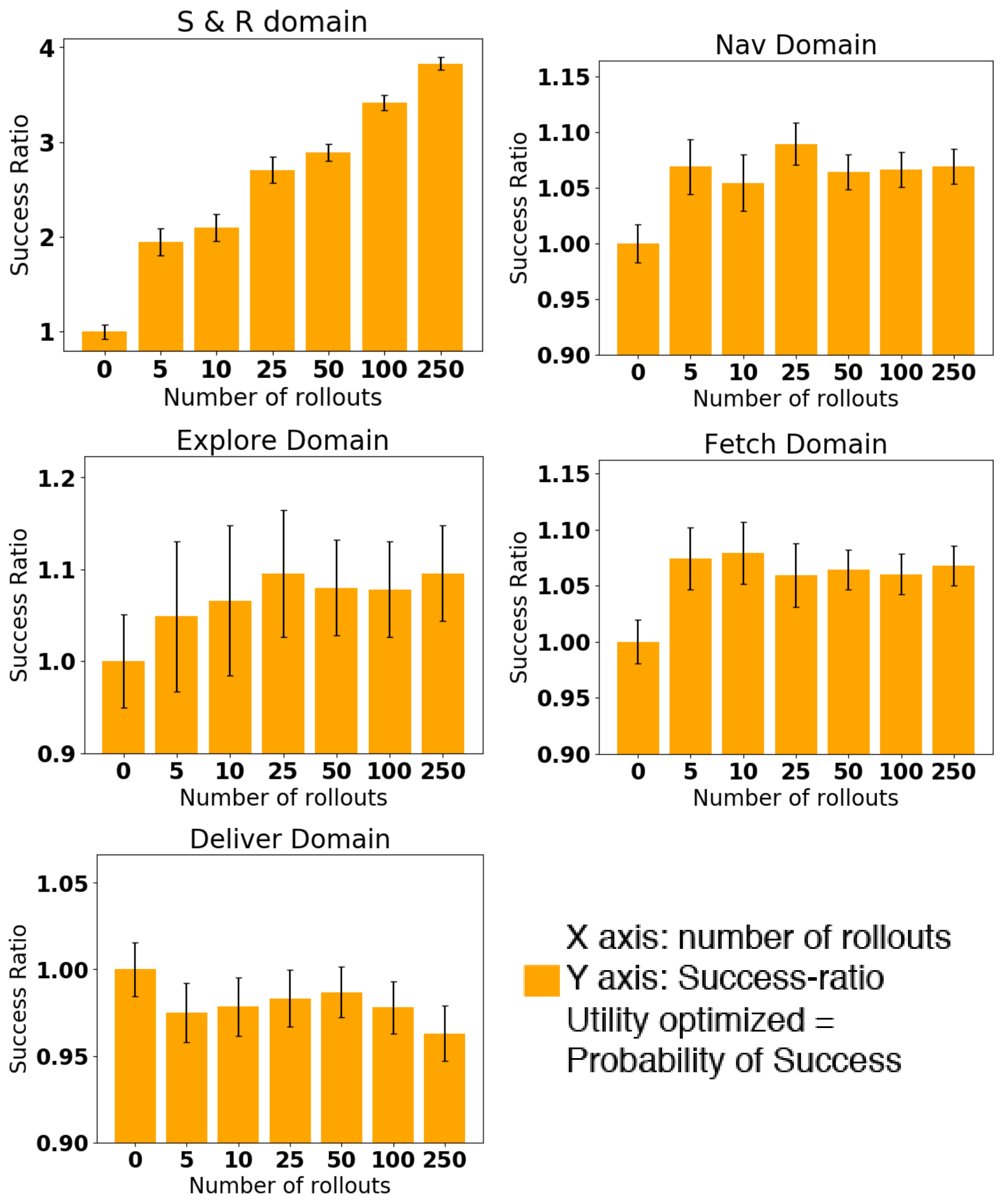


Figure 4.14: Measured success ratio (# of successful jobs/ total # of jobs) for $n_{ro} \in [0, 250]$ and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

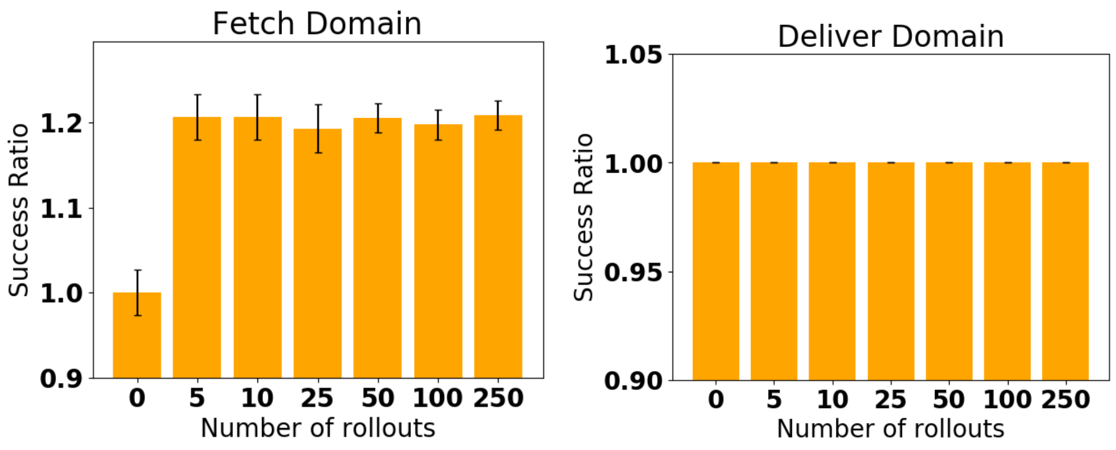


Figure 4.15: Measured success ratio averaged over only test cases with one root task, in Fetch and Deliver domains with $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

depth across all domains. In the Nav domain, the hLearnH performs better than $h0$ and hD with 95% confidence at depths 2 and 3. In the Explore domain, hLearnH performs better than $h0$ and hD at depth 1 with 95% confidence. The same is true for Fetch at depth 2. In the Deliver domain, the learned heuristic performs better than the others with 95% confidence for all depths >= 1. The performance difference between the three different heuristics are due to the properties of the domain, how the refinement methods are designed and how much of it is learnable by the LearnH procedure.

4.3.8 Measured *vs* expected efficiency

We already discussed how the measured efficiency of RAE is different from the expected one computed in UPOM. The difference between the two is given in Figure 4.17 with respect to the refinement deepness. A root task is refined recursively into sub-tasks. For each sub-task, RAE calls UPOM to choose a method. We

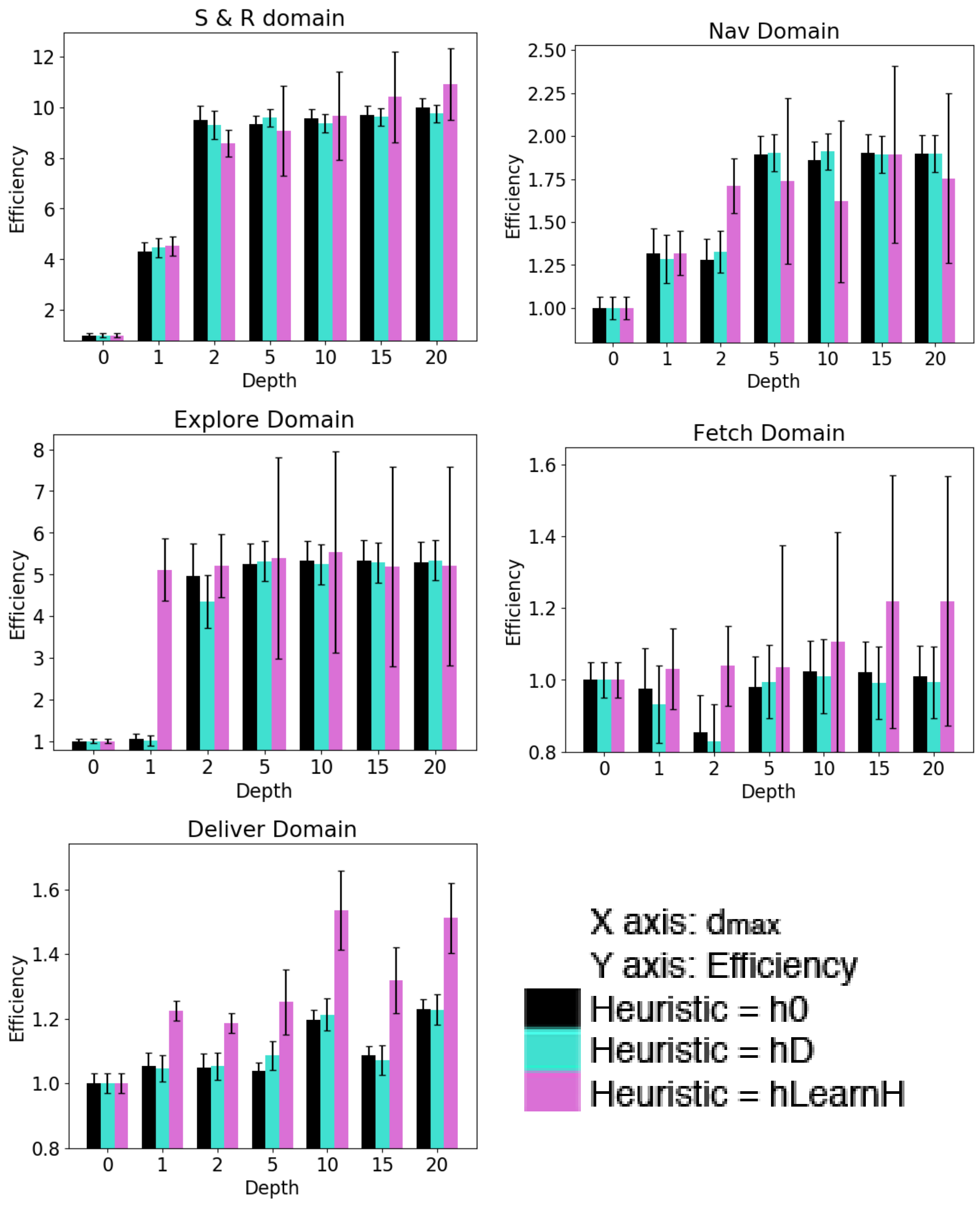


Figure 4.16: Measured efficiency with limited depth and three different heuristic functions. The utility function optimized is expected efficiency (relative values with respect to the base case of U for $n_{ro} = 0$).

note that the difference between the measured efficiency and the expected efficiency decreases as RAE makes progress towards accomplishing the root task.

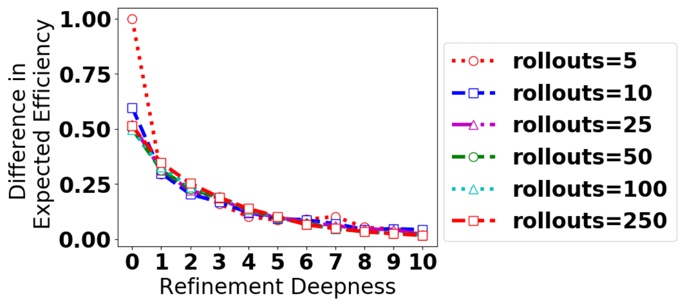


Figure 4.17: Absolute difference between measured and expected efficiency, as a function of the refinement deepness (0 is the root task), for various number of rollouts.

4.4 Comparison of RAEplan and UPOM

We discuss here RAE with UPOM *vs* RAEplan. We didn't compare UPOM with any non-hierarchical planning algorithms because it would be very difficult to perform a fair comparison, as discussed in [70].

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

Figure 4.18 shows the computation time for a single run of a problem (one or two root tasks), averaged across all domains and problems, i.e., over 10^4 runs. RAE with UPOM runs more than twice as fast as RAE with RAEplan. Note that the computation time of RAE alone is negligible, since it is designed to be a fast reactive system, without search. However, in physical experiments, the total time includes sensing and actuation time, hence the planning overhead would not appear as significant as it is here.

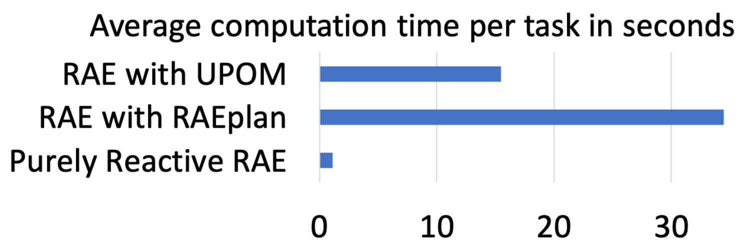


Figure 4.18: Average computation time in seconds for a single run of a problem, for RAE with and without the planners.

Efficiency. Figure 4.19 gives the measured efficiency for the five domains, with the 95% confidence intervals. It shows in all domains that RAE with UPOM is more efficient than purely reactive RAE and RAE with RAEplan.

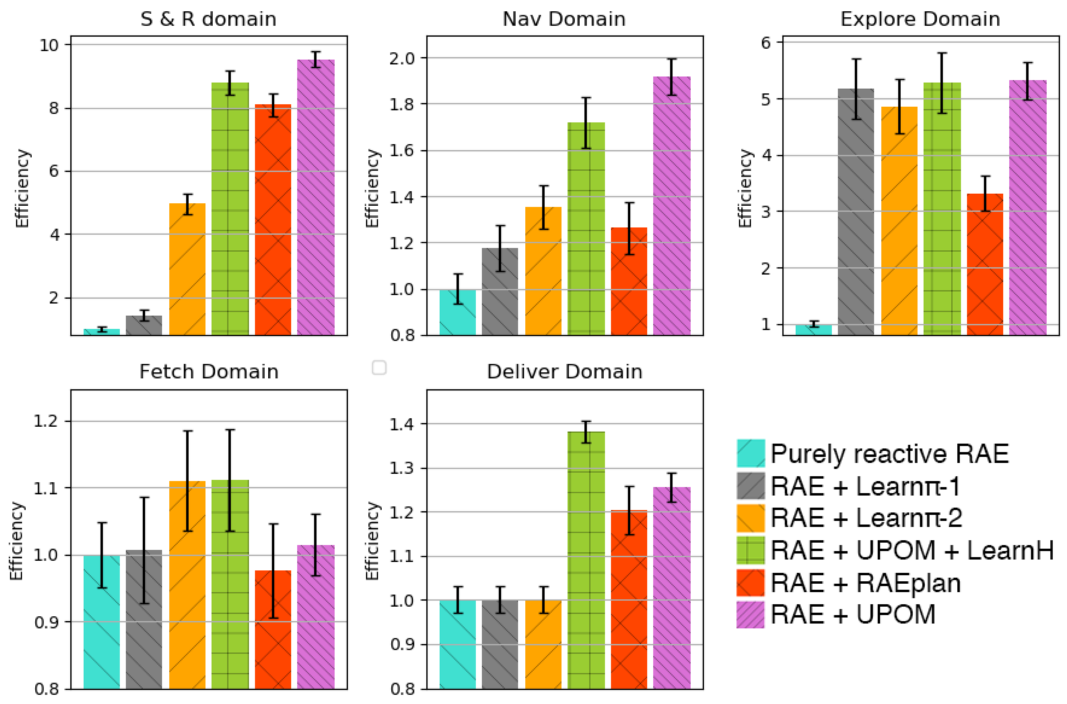


Figure 4.19: Measured efficiency for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

Success ratio. Figure 4.20 shows RAE's success ratio both with and without the planners. We observe that planning with UPOM outperforms purely reactive RAE in S&R and Fetch with 95% confidence, and Explore and Nav with 85% confidence. Also, UPOM outperforms RAEplan in Fetch and Nav domains with a 95% confidence, and Explore domain with 85% confidence. In the S&R domain, the success ratio is similar for RAEplan and UPOM.

Asymptotically, UPOM and RAEplan should have near-equivalent efficiency and success ratio metrics. They differ because neither are able to traverse the entire search space due to computational constraints. Our experiments on simulated environments suggest that UPOM is more effective than RAEplan when called online with real-time constraints.

4.5 Assessment of UPOM's learning strategies

For training purposes, we synthesized data records for each domain by randomly generating root tasks and then running RAE with UPOM. The number of randomly generated tasks in S&R, Nav, Explore, Fetch, and Deliver domains are 96, 132, 189, 123, and 100 respectively. We save the data records according to the Learnπ-1, Learnπ-2, Learnπ_i and LearnH procedures, and encode them using the One-Hot schema. We divide the training set randomly into two parts: 80% for training and 20% for validation to avoid overfitting on the training data.

The training and validation losses decrease and the accuracy increases with increase in the number of training epochs (see Figure 4.21).

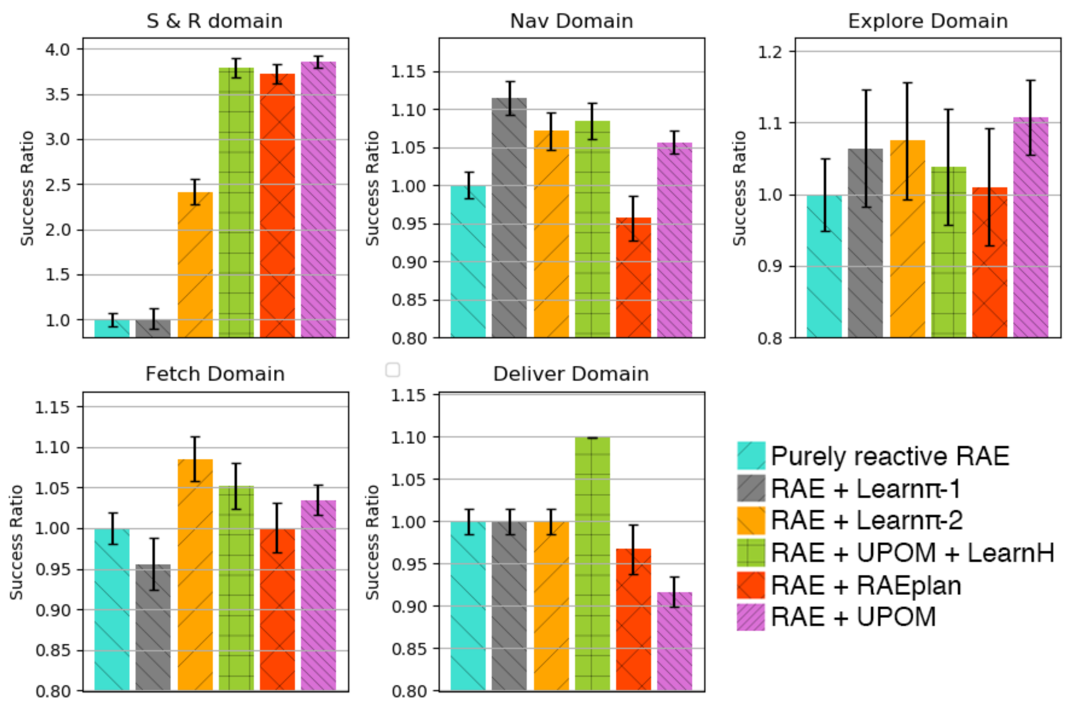


Figure 4.20: Measured success ratio for each domain with purely reactive RAE, RAE with RAEplan, RAE with the policies learned by Learnπ without planning, RAE with UPOM, the heuristic learned by LearnH and $d_{max} = 5$, and RAE with UPOM and $d_{max} = \infty$ (relative values with respect to the base case of U for $n_{ro} = 0$).

The accuracy of Learnπ is measured by checking whether the refinement method instance returned by UPOM matches the template predicted by the MLP nn_π, whereas the accuracy of LearnH is measured by checking whether the efficiency estimated by UPOM lies in the interval predicted by nn_H. We chose the learning rate to be in the range $[10^{-3}, 10^{-1}]$. Learning rate is a scaling factor that controls how weights are updated in each training epoch via backpropagation.

Table 4.3 summarizes the training set size, the number of input features and outputs after data records are encoded using the One-Hot schema, number of training epochs for the three different learning procedures. In the LearnH learning procedure, we define the number of output intervals K from the training data such that

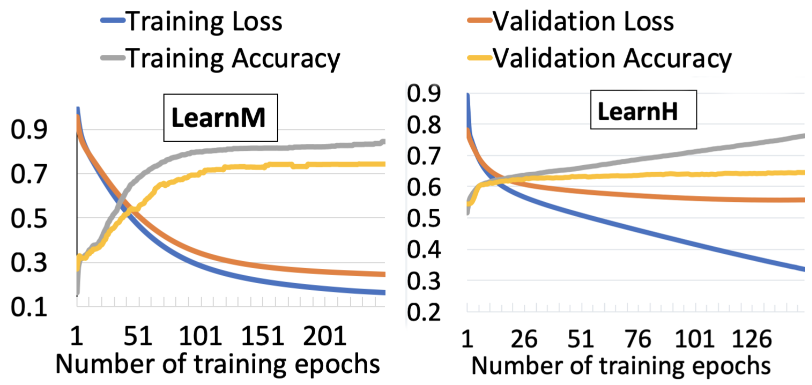


Figure 4.21: Training and validation results for Learnπ and LearnH, averaged over all domains.

Domain	Training Set Size			#(input features)		Training epochs		#(outputs)	
	LM-1	LM-2	LH	LM-1 and -2	LH	LM-1 and -2	LH	LM-1 and -2	LH
S&R	250	634	3542	330	401	225	250	16	10
Nav	1686	5331	16251	126	144	750	150	9	75
Explore	2391	6883	10503	182	204	1000	250	17	200
Fetch	262	508	1084	97	104	430	250	10	100
Deliver	-	-	2001	-	627	-	250	-	10

Table 4.3: The size of the training set, number of input features and outputs, and the number of training epochs for three different learning procedures: Learnπ-1, Learnπ-2, and LearnH. We note LM-1 = Learnπ-1, LM-2 = Learnπ-2, and LH = LearnH.

each interval has an approximately equal number of data records. The final validation accuracies for Learnπ are 65%, 91%, 66% and 78% in the domains Fetch, Explore, S&R and Nav respectively. The final validation accuracies for LearnH are similar but slightly lower. The accuracy values may possibly improve with more training data and encoding the refinement stacks as part of the input feature vectors.

To test the learning procedures we measured the efficiency and success ratio of RAE with the policies learned by Learnπ-1 and Learnπ-2 without planning, and RAE with UPOM and the heuristic learned by LearnH. We use the same test suite as in our experiments with RAE using RAEplan and UPOM, and do 20 runs for each test problem. When using UPOM with LearnH, we set d_{max} to 5 and n_{ro} to 50, which has about 88% less computation time compared to using UPOM with infinite d_{max} and $n_{ro} = 1000$. Since the learning happens offline, there is almost no computational overhead when RAE uses the learned models for online acting.

Efficiency. Figure 4.19 shows that RAE with UPOM + LearnH is more efficient than both purely reactive RAE and RAE with RAEplan in three domains (Explore, S&R and Nav) with 95% confidence, and in the Fetch domain with 90% confidence. The efficiency of RAE with Learnπ-1 and Learnπ-2 lies in between RAE with RAEplan and RAE with UPOM + LearnH, except in the S&R domain, where they perform worse than RAE with RAEplan but better than purely reactive RAE. This is possibly because the refinement stack plays a major role in the resulting efficiency in the S&R domain.

Success ratio. In these last experiments, UPOM optimizes for the efficiency, not the success ratio. It is however interesting to see how we perform for this criteria even when it is not the chosen utility function. In Figure 4.20, we observe that RAE with UPOM + LearnH outperforms purely reactive RAE and RAE with RAEplan in three domains (Fetch, Nav and S&R) with 95% confidence in terms of success ratio. In Explore, there is only slight improvement in success-ratio possibly because of high level of non-determinism in the domain's design.

In most cases, we observe that RAE does better with Learnπ-2 than with Learnπ-1. Recall that the training set for Learnπ-2 is created with all method instances returned by UPOM regardless of whether they succeed while acting or not, whereas Learnπ-1 leaves out the methods that don't. This makes Learnπ-1's training set much smaller. In our simulated environments, the acting failures due to random exogenous events don't have a learnable pattern, and a smaller training set makes Learnπ-1's performance worse.

4.5.1 Learning Method Instances

Two of our simulated domains, Nav and Deliver, have refinement methods with parameters that are not inherited from the task at hand. For these domains, Learnπ-1 and Learnπ-2 give only partially instantiated methods, while Learnπ_i is more discriminate. To test its benefit, we trained a MLP for each parameter not specified in the task. The size of the training set, number of input features and number of outputs are summarized in Table 4.4.

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

Table 4.4: The size of the training set, number of input features and outputs for learning method parameters in Learnπ_i.

Figure 4.22 compares the efficiency of RAE with Learnπ_i vs purely reactive RAE and RAE with RAEplan, Learnπ-1, Learnπ-2, LearnH, and UPOM. In the Deliver domain, RAE with Learnπ_i is better than purely reactive RAE as well as RAE with Learnπ-1 or Learnπ-2 with 95% confidence. In the Nav domain, RAE with Learnπ_i also outperforms Learnπ-1 and purely reactive RAE with 95% confidence, but not Learnπ-

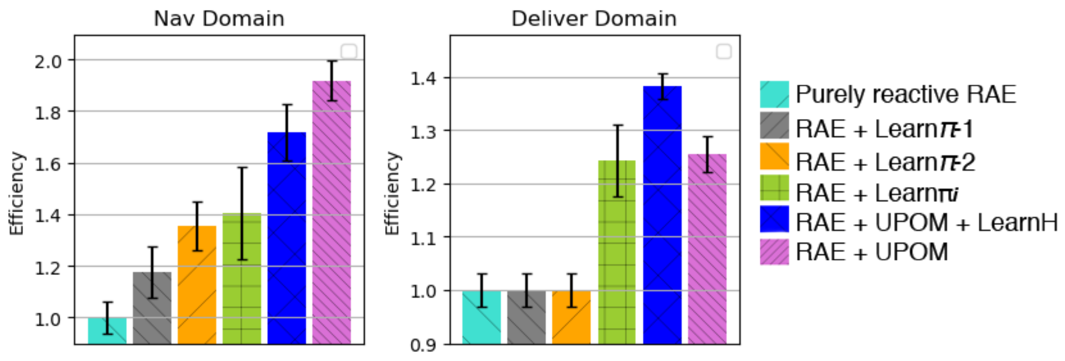


Figure 4.22: The cross hatched blue bars show the performance of RAE with Learnπ_i (learning method instances) for the two domains, Nav and Deliver, which have methods with parameters not in tasks (relative values with respect to the base case of U for $n_{ro} = 0$).

2. The performance benefit is significant in the Deliver domain because refinement methods have several uninstantiated parameters.

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

4.6 Summary

In this chapter, we described the experimental setup and evaluation of our three refinement planning algorithms, APEplan, RAEplan, and UPOM, each integrated with RAE. Using RAE with refinement planners always proved to be more beneficial than purely reacting acting in terms of efficiency, retry ratio and success-ratio. We measured RAE's efficiency, success ratio and retry ratio, and discussed their relationships with respect to the planner's utility function, maximizing either the expected efficiency or the expected success ratio. For UPOM, integration with learning strategies performed better than purely reactive RAE with 95% confidence across all five test domains.

Chapter 5: Real-world prototype of RAE and Plan-with-UPOM: Defense against SDN attacks

In this chapter, we describe a real-world prototype of RAE and Plan-with-UPOM to defend software-defined networks against incoming attacks. First, we give a brief overview of Software defined networks (SDNs). We describe the AIRMAN system developed by our collaborators at NRL to monitor, manage and detect attacks happening to an SDN. Then, we describe how RAE and Plan-with-UPOM are integrated with AIRMAN to defend against attacks by planning. Finally, we present some experimental results.

5.1 Software Defined Networks (SDNs)

Software-defined networking is a relatively new network management approach that enables dynamic, modular, programmatically efficient network configuration in order to improve network performance and to simplify monitoring. In general, network management architectures have two layers:

1. the data layer, where the traffic flows, and network packets are forwarded;
2. the control layer, which manages the packet routing process.

In traditional network architectures, these two layers are highly coupled and the control is decentralized, which leads to several complexities. SDN architectures addresses these issues by decoupling the two layers and having a centralized control layer with a set of controllers. However, this change in design comes with its own drawbacks when it comes to security. To address the security challenges, our collaborators at NRL have developed a system called Autonomous Intelligent Resilient Security (AIRS) shown in Figure 5.1. RAE and Plan-with-UPOM are adapted to communicate with AIRMAN in order to defend the SDNs against incoming attacks.

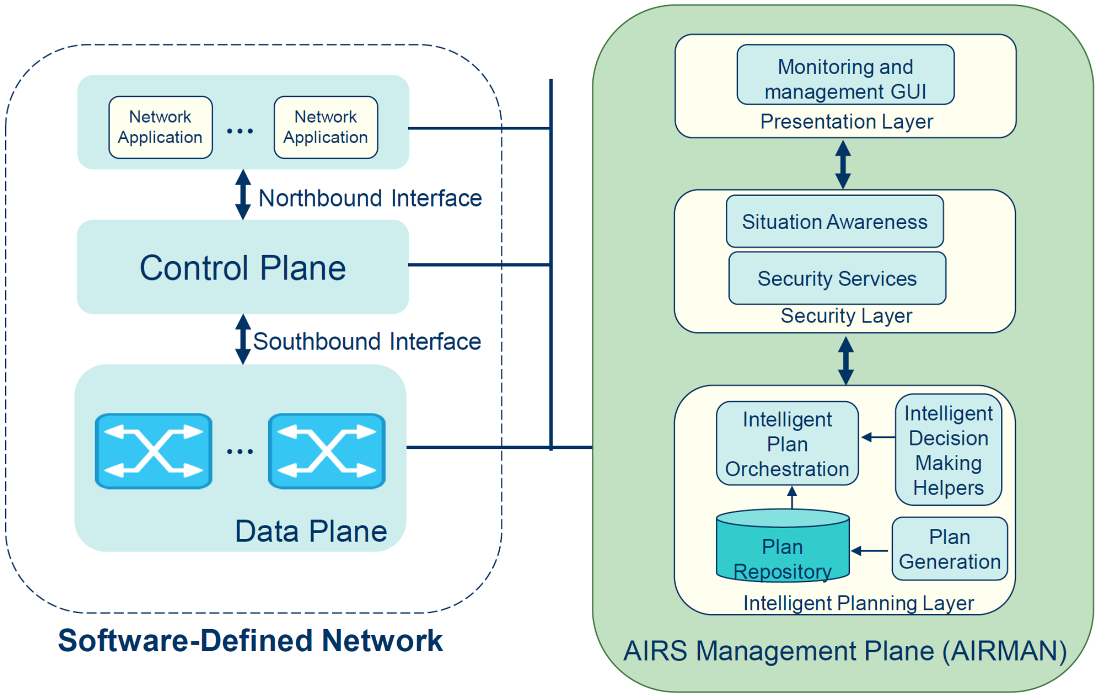


Figure 5.1: AIRS Architecture for SDN

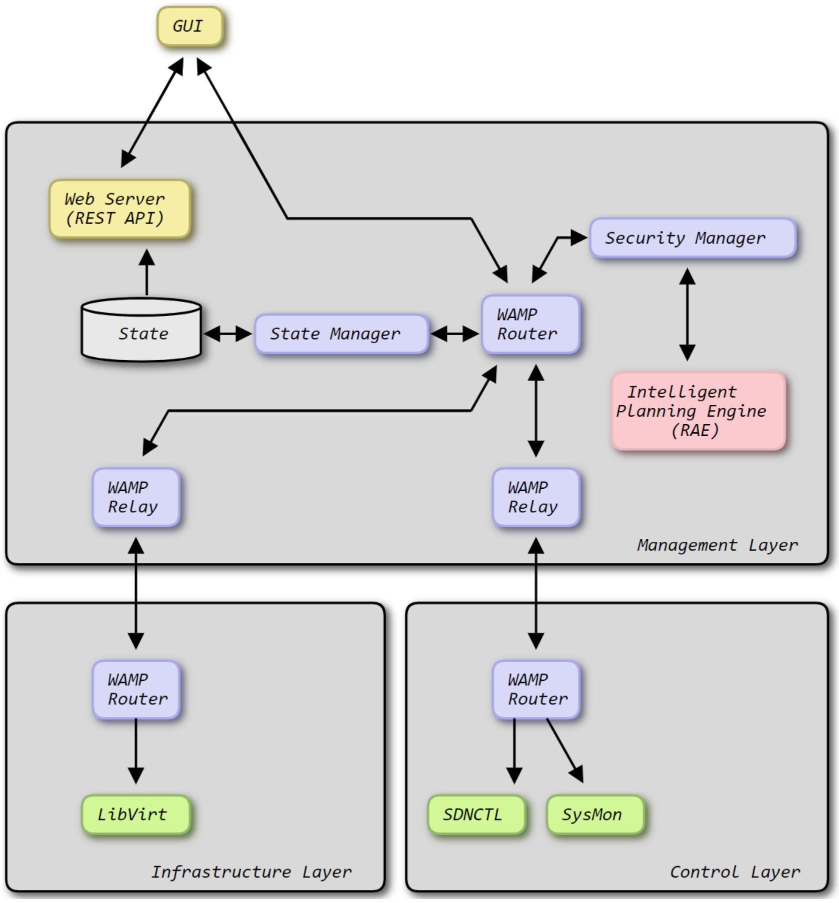


Figure 5.2: AIRMAN architecture

5.2 AIRMAN Architecture

Figure 5.2 shows the architecture of the AIRS Management Plane (AIRMAN). The central WAMP router enables communication between the SecurityManager and other AIRMAN components via a mixture of publish-subscribe messaging (Pub-Sub) and remote procedure calls (RPC).

RAE, the actor, is integrated directly with the SecurityManager as a Python module, and they communicate with each other asynchronously using a set of shared queues between the processes AIRMAN and RAE. RAE gets all of its information about the system's state directly from the SecurityManager and relies on the SecurityManager to carry out the planned actions on the system and provide feedback. The Security Manager is a part of the larger AIRMAN system (AIRMAN monitors the whole Software-defined network) that interacts with RAE. It handles the details related to the Software-defined network which, if included in RAE, might make RAE too complex for general purposes. Its job is mainly to push tasks into RAE's task queue with the right parameters and state, and review the commands that RAE suggests. It has an interface for a human expert to review what RAE is suggesting for safety purposes.

Whenever RAE has planned a command for the SecurityManager to execute, it puts the command in a command execution queue. The SecurityManager continuously monitors the command execution queue for incoming commands. Once it receives a command, it executes it, and sends back to RAE the information about whether the command succeeded or failed and the next state. We describe the communication in more detail in Section 5.3.4.

5.3 Attack recovery using RAE and Plan-with-UPOM

Software-defined networks (SDN) is prone to several different kinds of known and unknown attacks. We develop a method of automatically defending against, and, recovering from, attacks on SDNs using RAE and Plan-with-UPOM. The AIRS management plane (AIRMAN) has a security layer and an intelligent planning layer (Figure 5.1). We describe how the AIRMAN SecurityManager, Refinement Acting

Engine (RAE), and planner (Plan-with-UPOM), interact with each other and work together to defend against, and, recover from, attacks to an SDN.

SDNs are vulnerable to various different kinds of attacks and failures, such as:

1. packet overflow: too many packets arrive at the host;
2. packet underflow: expected packets don't arrive;
3. switch malfunction: swich starts showing unexpected behavior due to an internal error or some attack from outside;
4. controller malfunction: the health and trust values of the controller go down.

A defense system for SDNs continuously monitors the SDN and ensures that it is behaving as expected. To accomplish this, the defense system must:

1. detect than an attack has occured on the SDN.
2. diagnose what kind of attack has occured. An attack may be detected when the network reaches some inconsistent state, or shows irregular behavior.
3. come up with an online plan to recover from the damage this attack has caused. The recovery process may be different depending on current state of the system and the nature of the attack. The planner may replan when necessary.

Let us now look at an example of an attack and refinement methods to recover from it. In a PACKET_IN flooding attack, one or more malicious hosts continuously send traffic to an unknown (and possibly randomized) destination address. Every time a switch receives a packet that it does not know what to do with, it sends an OpenFlow PACKET_IN request to the controller. Since the controller does not know the location of the destination address, it instructs the switch to flood the packet to all output ports. Over time, if the volume of packets is large enough, the flood of PACKET_IN requests eats up control plane bandwidth and can cause denial of service of the controller. It can also consume resources on the switch that an offending host is connected to, and side-effects can be felt network-wide as long as the controller remains unresponsive. Here are three refinement methods to recover from PACKET_IN flooding.

The first method, m1_ctrl_clearstate_besteffort(id), simply clears the host table in the controller, which may be enough if the attacker has stopped sending new packets and the only lingering cause of resource exhaustion is an inflated host table in the controller. A refinement tree for packetIn-flooding(id) using m1_ctrl_clearstate_besteffort(id) is shown in Figure 5.3.

```
m1_ctrl_clearstate_besteffort($id$)
  event: packetIn-flooding($id$)
    pre: None
   body:
         if not is_component_type($id$, 'CTRL'): fail
         else: clear_ctrl_state_besteffort($id$)
```

The second method, m2_ctrl_clearstate_fallback(id), also clears the host table in the controller, but does so in a higher assurance way, which may take longer but is

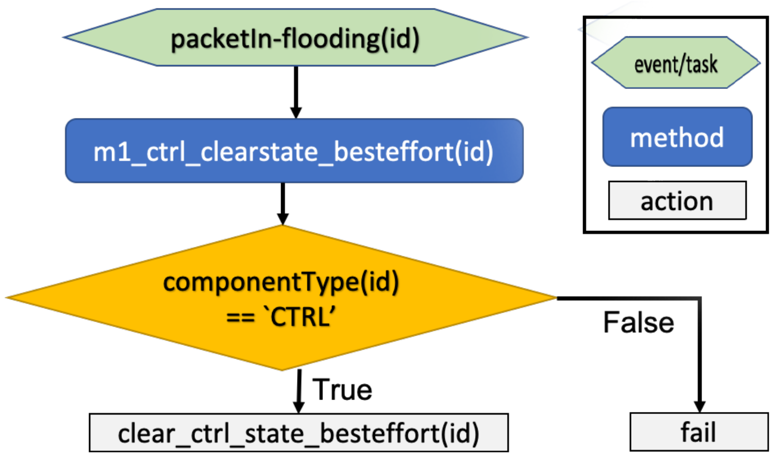

packetIn-flooding(id)
m1_ctrl_clearstate_besteffort(id)
componentType(id) == `CTRL'
False
True
clear_ctrl_state_besteffort(id)
fail
event/task
method
action

Figure 5.3: A refinement tree for the event packetIn-flooding(id) using the refinement method m1_ctrl_clearstate_besteffort(id)

less likely to fail. A refinement tree using m2_ctrl_clearstate_fallback(id) is shown in Figure 5.4.

```
m2_ctrl_clearstate_fallback(id)
  event: packetIn-flooding(id)
         if not is_component_type(id, 'CTRL'):
             fail
         else:
             clear_ctrl_state_fallback(id)
```

The third method, m3_ctrl_mitigate_pktinflood(id), searches for switches which have been marked as unhealthy by the SecurityManager of AIRMAN, moves all critical hosts away from each such switch to a newly added switch before attempting to fix the old switch, and finally clears the host table in the controller. A refinement tree using m3_ctrl_mitigate_pktinflood(id) and with one unhealthy switch s_1 is shown in Figure 5.5.

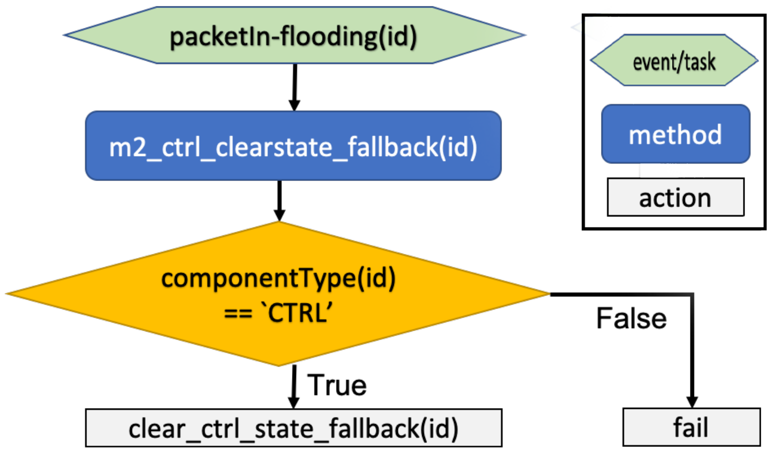

packetIn-flooding(id)
m2_ctrl_clearstate_fallback(id)
componentType(id) == `CTRL'
True
False
clear_ctrl_state_fallback(id)
fail
event/task
method
action

Figure 5.4: A refinement tree for the event packetIn-flooding(id) using the refinement method m2_ctrl_clearstate_fallback(id)

```
m3_ctrl_mitigate_pktinflood$(id)$
 event: packetIn-flooding$(id)$
 body:
     if is_component_type$(id) \neq$ 'CTRL': fail
     for $s\_id$ in state.components: # Detect the attack's source
       if is_component_type($s\_id$, 'SWITCH') and !is_component_healthy($s\_id$):
         if is_component_critical($s\_id$): # Move critical hosts away
           add_switch ($s\_id$) # Add new switch
           move_critical_hosts($s\_id$, $s\_id$ + '-new')
         fix_switch($s\_id$) # Fix unhealthy switch
     clear_ctrl_state_besteffort$(id)$ # Clear controller state
     if not is_component_healthy$(id)$: fail
```

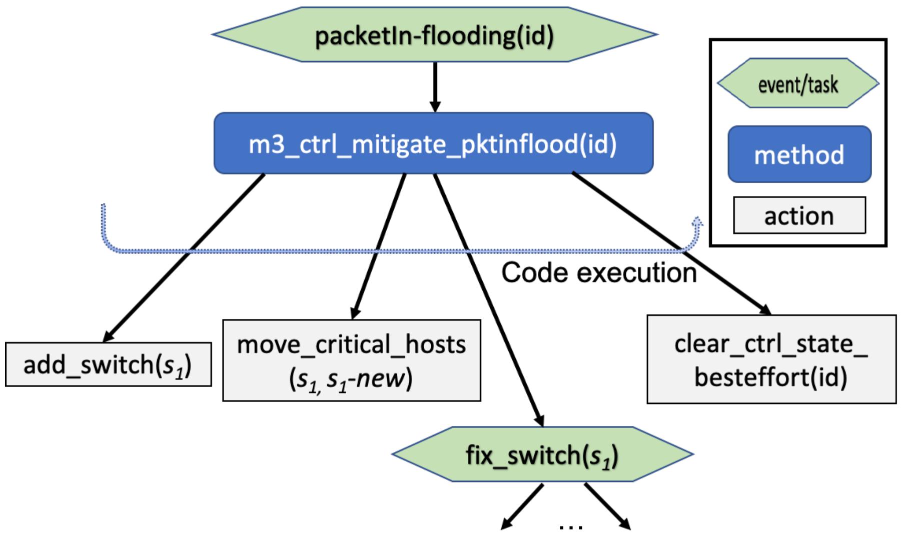

packetIn-flooding(id)
m3_ctrl_mitigate_pktinflood(id)
event/task
method
action
Code execution
add_switch(s_1)
move_critical_hosts (s_1, s_1-new)
clear_ctrl_state_ besteffort(id)
fix_switch(s_1)
...

Figure 5.5: A partial refinement tree for the event packetIn-flooding(id) using the refinement method m3_ctrl_mitigate_pktinflood(id). fix-switch(s_1) is a sub-task that should further be refined.

5.3.1 Integration of AIRMAN SecurityManager and RAE

The SecurityManager has options in its configuration file that control whether the planner will be used and how verbose its log messages should be. If the planner is enabled, the SecurityManager initializes RAE with its state and planning parameters and receives references to the shared queues that will be used for communication.

5.3.2 State Definition for SDN

The state consists of two top-level Python dictionaries: `components` and `stats`.

We mainly deal with two types of components: controllers and switches. The `components` dictionary maps from component IDs to a nested dictionary containing keys that map to properties of the component (`id`, `type`, `critical`, etc.):

The `stats` dictionary maps from component IDs to a nested dictionary containing the keys `health`, `cpu_perc_ewma`, and potentially a number of other keys for statistics that may depend on the component's type (e.g., a switch will have `flow_table_size`). Each of these maps to another dictionary containing the key's `value`, the current value of this statistic, and `thresh_exceeded_fn`, a function that evaluates to `true` if the value exceeds the configured threshold (a numeric value in valid range of the state variable chosen by a human expert) for that statistic.

A variety of scenarios could require planning, such as:

- a specific malicious event is detected (e.g., an intrusion detection system on a controller or switch generates an alert and that is reported to the SecurityManager);
- some kind of manual or automated diagnosis leads to suspicion of a specific attack;
- the system exhibits symptoms that are outside of the normal healthy bounds.

In the latter case (such as, an abnormal symptom in a component leads to degraded "health" metric), typically, the following sequence of events cause the SecurityManager to submit an "attack" event to RAE or an "attack recovery" task to RAE:

1. A component sends a `sys_stats` message with a particular attribute (e.g., CPU utilization percentage) that has exceeded its configured threshold.

2. SecurityManager has been tracking statistics for that component's attribute (e.g., the exponential weighted moving average (EWMA) of the CPU utilization percentage) and observes that the attribute has exceeded a configured threshold.

3. The component's health is reduced and falls below the configured action threshold.

4. SecurityManager updates the shared state and submits a task to RAE to fix this component's symptoms.

We have modeled the Security domain such that every incoming attack to the Software-defined network corresponds to a recovery task. For example, if the Security Manager detects an attack to a controller, it will put the task fix_controller in RAE's task queue. In the language of events, it can also put the event, controller_attacked in the RAE's task queue. It will be handled in the same way.

5.3.3 Utility function optimized: CostEffectiveness

Plan-with-UPOM can optimize different utility functions, such as the acting efficiency (reciprocal of the cost) or the probability of success. For this domain, we focus on optimizing the cost-effectiveness of methods, which is a linear combination of efficiency and probability of success. For the Software-defined network domain, we define cost-effectiveness to be a utility function with the following properties:

1. The value is 0 if the action ultimately leads to failure.

2. The value is inversely proportional to the cost; that is, a more expensive action corresponds to a lower value.

If we are interested in defending and recovering an IT system from cyberattacks, we need to select actions that maintain or return the system to a healthy state, and do so with minimal resources spent.

If the task at hand is to repair a given component (switch or controller), we are not interested in actions that get us close but ultimately fail at completing the task, no matter how much cheaper those actions are compared to the alternatives. We would rather select a more expensive course of actions that succeeds at the recovery task. Having said that, we do want to minimize cost. For example, if a switch's flow table is corrupted, rebooting the switch can be effective but would cause considerable downtime, whereas flushing and repopulating the flow table could be a cheaper (faster) way to achieve the result.

Our motivation for defining our utility function, cost-effectiveness, in this way is to guide Plan-with-UPOM in its Monte Carlo tree search towards effective-but-cheap solutions, while ignoring ineffective solutions and de-prioritizing expensive ones.

CostEffectiveness. Let a method m for a task τ have two sub-tasks, τ_1 and τ_2, with cost c_1 and c_2, and probability of success, p_1 and p_2 respectively. The cost-effectiveness of τ_1 is $u_1 = 1/c_1 + p_1$ and the cost-effectiveness of τ_2 is $u_2 = 1/c_2 + p_2$.

The cost-effectiveness of accomplishing both tasks is

$$1/(c_1 + c_2) + p_1 p_2 \tag{5.1}$$

If $c_1 = 0$ and $p_1 = 1$, the cost-effectiveness for both tasks is u_2; likewise for $c_2 = 0$ and $p_2 = 1$. Thus, the incremental cost-effectiveness composition is:

$$\begin{aligned} u_1 \oplus u_2 = u_2 &\text{ if } u_1 = \infty, \text{ else} && (5.2)\\ u_1 &\text{ if } u_2 = \infty, \text{ else} \\ 0 &\text{ if } u_1 = 0 \text{ or } u_2 = 0\text{, else} && (5.3)\\ &\frac{(u_1-1)(u_2-1)}{u_1+u_2-2} + 1. && (5.4) \end{aligned}$$

If τ_1 (or τ_2) fails, then c_1 is ∞, $u_1 = 0$. Thus $u_1 \oplus u_2 = 0$, meaning that τ fails with method m. Note that formula 5.2 is associative. When using cost-effectiveness as a utility function, we denote $U(\mathsf{Success}) = \infty$ and $U(\mathsf{Failure}) = 0$.

5.3.4 Communication between SecurityManager and RAE

Communication between the SecurityManager and RAE occurs using three shared queues:

- **Task queue**: After the SecurityManager detects an attack, it puts an “attack recovery” task on the task queue. The task stays in the queue until RAE reads from it.

- **Command execution queue**: After planning using Plan-with-UPOM, RAE sends commands (atomic actions to be executed) to the SecurityManager by putting them in the command execution queue one by one. The SecurityManager reads the command from the queue.

- **Command status queue**: After executing a command, SecurityManager puts the information about whether a command succeeded or failed and next state of the SDN in the command status queue. RAE reads this information and updates the state accordingly.

The command execution queue and the command status queue work together in a back-and-forth manner. RAE plans for an attack recovery task using Plan-with-UPOM. Plan-with-UPOM returns the first command in the plan. RAE puts this command in the command execution queue. The SecurityManager continuously monitors the command execution queue for incoming commands. Once it sees a command, it pops the command from the queue, and executes the command on the target system, the SDN. Once the command is done executing, the SecurityManager puts the next status, and the information about whether the command has succeeded or failed, on the command status queue. RAE receives this information and again replans (calls Plan-with-UPOM with the updated state) or calls Retry depending on whether the command has succeeded or failed respectively. It then sends the next command to execute, to the SecurityManager.

5.3.5 Example of task invocation workflow

For example, suppose the SecurityManager sends a task `fix_component`(`switch1`) to RAE by putting it on the task queue.

RAE removes this task from the task queue and chooses an applicable refinement method instance for the task. A task can have several refinement methods, each of which accomplishes the task in a different way. For the given component ID (`switch1`) and the component's type (which can be looked up by ID in the state, see Section 5.3.2), the only refinement method that can be applied is `fix_switch`. Each method has its own arguments. The arguments can come from the task or be assigned via planning. `fix_component` has two refinement methods: `fix_ctrl` and `fix_switch`. The first thing that the `fix_ctrl` refinement method does is to check the component's type and fail since it is not a controller.

The `fix_switch` refinement method does a similar check and sees that the referenced component's type is "SWITCH". Therefore, it refines into further tasks and actions that apply to a switch. Which tasks/actions apply depends on the current state of the component and the system as a whole.

Within a refinement method, if more information is needed than is available in the state, then a probing action is used to request this information from the SecurityManager. For example, if a switch component is misbehaving and the course of action depends on whether its flow table is over-filled, but the size of its flow table was not included in the state, then the `get_switch_flowtable_size` action can be requested. When this command returns successfully, then flow of control can continue and the missing value will be included in the updated state.

When a refinement method needs to call a probing action for more information, it calls `rae.do_command(commandName, commandParameters)`. RAE puts the command on the execution queue, from where the SecurityManager will read it. After the command is executed, the SecurityManager puts the result and the next state information in the command status queue. RAE then calls Plan-with-UPOM or Retry depending on whether the command has succeeded or failed respectively.

Ultimately, a task gets refined into one or more atomic actions (commands). However, RAE does not pass all of them to the SecurityManager at once. Rather, it requests them one at a time, considers the success/failure of each, and calls Plan-with-UPOM or Retry after each (taking into account the updated state it receives from the SecurityManager after each command) in case a better result is possible. For each command that RAE wants to be executed on the SDN, RAE puts the name of the command, as well as any parameters, on the execution queue:

The SecurityManager has an ongoing process that continually checks the execution queue. When a command is available, it reads the command name and any parameters from the queue.

It then looks up the command by name and dispatches it as appropriate. Some commands need to run on the component itself, while others need to run on the underlying platform (e.g., the virtual machine monitor (VMM)) that the component is running on. In either case, the SecurityManager notifies RAE of success or failure (along with a copy of the updated state) via the status queue.

.

5.3.6 Domain Definition for SDN

The AIRS SDN domain is defined in terms of tasks, refinement methods, mappings from task to refinement method(s), atomic actions (commands), and costs of actions. All of these constituents are defined in a domain file and encoded by domain experts in the language of operational models using the framework and constructs that RAE provides. This includes the tasks, events, refinement methods and actions described in Section 3.1. Each predictive model of an action has a counterpart on the execution platform of the SecurityManager, so that when an action is passed by RAE to the SecurityManager, it can actually be carried out on the software-defined network.

RAE provides the ability to define a cost for a method directly. In case a method has a non zero cost assigned to it, this cost is used in addition to the sum of the individual command costs.

5.3.7 Environment for SDN

The actions are nondeterministic and the predictive models used by Plan-with-UPOM sample their outcome from a probability distribution. During runtime, these actions may be called with various different parameters and the empirical success rate may deviate from these expected probabilities. The probability values are assigned by the human expert. They may also be adjusted using empirical data by doing something as follows. For example, suppose while recovering from attacks, we execute the command `restart_vm`, inside a refinement method instance, a number of times for two different components, `ctrl1` and `switch1`. For `ctrl1`, it works 10 out of 10 times, but for `switch1` it only works 2 out of 5 times. Over time, one can learn that the probability of success for `restart_vm` with param `ctrl1` may be higher than 95%, closer to 100%, while the probability of success for `restart_vm` with param `switch1` is likely lower than 95%, closer to 40%.

5.3.8 Action Model

In the SecurityManager, there is a mapping from command names to Python functions that carry out the action on the execution platform.

In the domain definition, we declare the command in the operational model. This assigns the name, any parameters (e.g., component ID), and code that modifies the state and return success or failure, to model the effect that the command is expected to have on the system. Pre conditions are encoded in the operation model (either in a command function or refinement method) by checking the state and returning failure if the command does not apply to the current state.

In the environment definition, the domain expert assigns an estimated probability to each command. Other strategies, such as, learning from history, or using a simulator, may also be used to guess how likely a command (potentially with specific parameters) is to succeed.

5.4 Experimental Evaluation

In order to test the our SDN attack recovery system (AIRMAN with RAE and Plan-with-UPOM), we modelled attacks to the SDN as events and recovery procedures as refinement methods to recover from the attacks. We have 11 tasks, 28 refinement methods and 13 commands. Full descriptions of the operational models are in Section B.6. We randomly generated a test suite of 50 problems. Each problem has a initial state that is decided by randomly assigning values to the state variables, ensuring that they correspond to a valid state. The number of controllers in a problem ranged from one to four and the number of switches ranged from 16 to 64. One to three switches or controllers were randomly chosen to be attacked. We configured Plan-with-UPOM to optimize cost-effectiveness. Each test problem was run 50 times to account for nondeterministic outcomes of commands. In our experiments, the probability distribution of the commands' outcomes are chosen by a human expert. We ran the tests on a simulated Software-Defined network running on a 2.8 GHz Intel Ivy Bridge processor.

In order to measure the performance of AIRMAN, we measure four different metrics: the estimated time for attack recovery, the efficiency and retry-ratio, and the cost-effectiveness. We discuss each of them as follow.

Estimated time for attack recovery. Figure 5.6 shows how the estimated time for attack recovery changes as we give more time to the refinement planner, Plan-with-UPOM. We observe that purely reactive RAE (with no planning, i.e., 0 time given to Plan-with-UPOM) is able to help the SDN recover from attacks in ∼11 seconds. Further, when doing refinement planning with Plan-with-UPOM, we observe ∼32% decrease in the estimated time for recovery. The error bars in the plot show 95% confidence intervals.

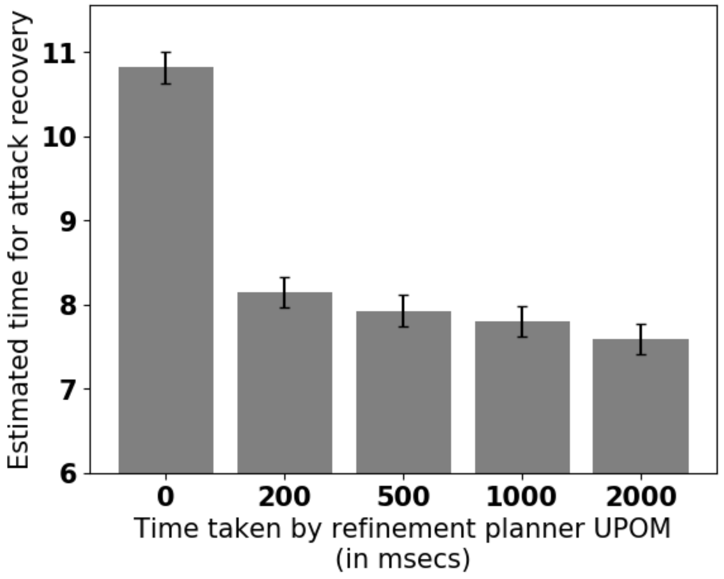


Figure 5.6: Estimated time for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM.

Efficiency. Figure 5.7 shows how the efficiency improves as Plan-with-UPOM is given more time to do more rollouts. Efficiency is the reciprocal of the estimated cost for attack recovery. In our experiments, the cost of the commands are chosen by human experts and they roughly correspond to the duration of the commands. The efficiency measures the reciprocal of the cost instead of measuring the cost value directly to account for failed commands which have infinite cost.

Retry ratio. If a method for a task τ fails during execution, RAE looks at the list of untried methods for τ and chooses one among them. Each such choice is called a Retry. The higher the number of retries, the higher is the execution time. Retry ratio measures the number of retries including all sub-tasks divided the total number

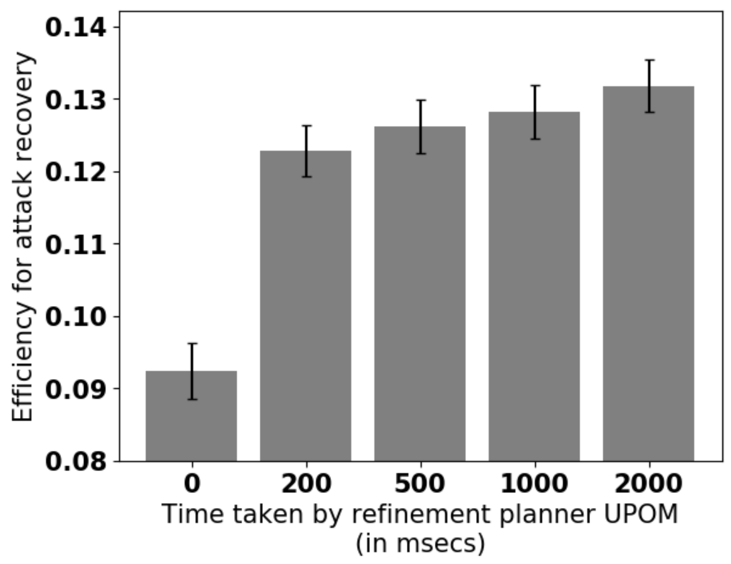


Figure 5.7: Efficiency (reciprocal of estimated cost) for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM.

of incoming tasks in RAE. Figures 5.8 shows the retry ratio for the experiments with the randomly generated test suite. From the plot, we can conclude that planning with a time limit of 200 msecs for Plan-with-UPOM outperforms purely reactive RAE with 95% confidence. Planning with a time limit of 2 seconds for Plan-with-UPOM outperforms planning with a time limit of 200 msecs with 95% confidence.

CostEffectiveness. The cost-effectiveness (Equation 3.3) is a linear combination of the efficiency and the probability of success. The cost-effectiveness remains the same with and without using Plan-with-UPOM because the domain has no dead-ends. A dead-end is a situation/state from which the actor cannot recover from. In SDNs, one can always restart the network if nothing works making dead ends impossible. A reboot can recover from any state that the network can possibly reach.

In summary, we are able to automate attack recovery in SDNs using AIRMAN and the refinement acting engine, RAE. Planning with Plan-with-UPOM further im-

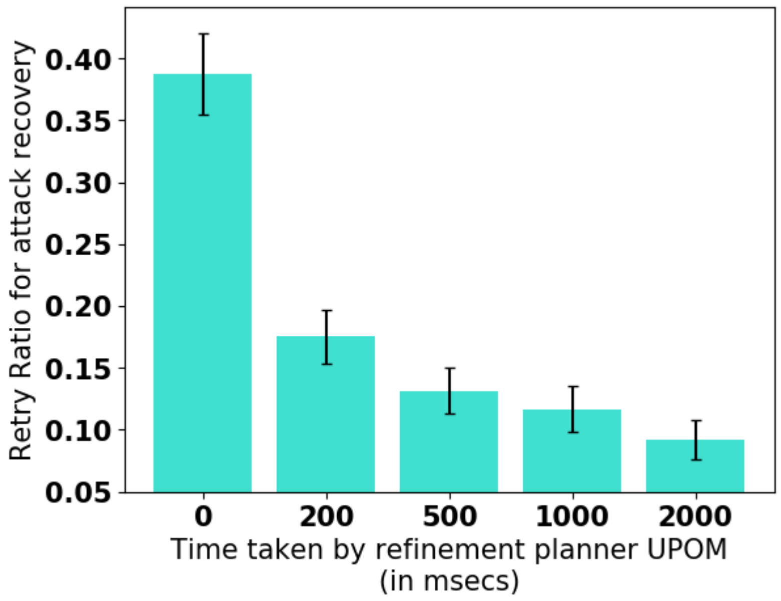


Figure 5.8: Retry ratio for attack recovery in AIRMAN using the Refinement Acting Engine, RAE and refinement planner, Plan-with-UPOM.

proves the performance in terms of estimated recovery time, efficiency and retry-ratio with 95% confidence.

5.5 Summary

In this chapter, we saw how RAE, Plan-with-UPOM and the language of hierarchical operational models can be used to defend software-defined networks against incoming attacks. An attack to the SDN corresponds to a recovery task, and there may be multiple applicable refinement methods to recover from it. RAE and Plan-with-UPOM suggests to the SecurityManager of AIRMAN the best way to proceed. Our experiments show that integrating AIRMAN, RAE and Plan-with-UPOM improves the estimate time, efficiency, and retry-ratio of attack recovery.

Chapter 6: Conclusion

In this dissertation, the author has presented a novel set of algorithms for integrating acting and planning using hierarchical operational models. The APEplan and RAEplan algorithms use a SLATE-style sampling strategy. The algorithm Plan-with-UPOM, a planning algorithm that outperforms both APEplan and RAEplan, uses a search strategy inspired by UCT, but is adapted to operate in a more complicated search space. Plan-with-UPOM provides near-optimal choices with respect to an arbitrary utility function and converges asymptotically. Plan-with-UPOM has been used successfully with two distinct utility functions, favoring respectively efficiency and robustness. Plan-with-UPOM outperforms APEplan and RAEplan because UCT search explores more promising refinement methods with the help of *Upper Confidence Bound (UCB)* formula and estimated utility values. In contrast, SLATE strategy explores all refinement methods uniformly. Further, Plan-with-UPOM is more flexible because its time complexity is linear with respect to its parameters, n_{ro} and d, whereas, RAEplan's time complexity is exponential with respect to its parameters, b and k. Plan-with-UPOM integrates RAE with UPOM by doing a receding horizon, anytime progressive deepening.

Finally, Learnπ, Learnπ_i, and LearnH are learning strategies that can be used to improve RAE's performance. Learnπ learns a mapping from a task in a given context to a good method, Learnπ_i learns values of uninstantiated method parameters, and LearnH learns domain-specific heuristic function.

The author has implemented these algorithms and has test tested them on five simulated domains. We have devised a novel, and we believe realistic and practical way, to measure the performance of RAE and similar systems. The domains were designed to reflect interesting aspects of real-world domains, e.g., dynamicity, the need for run-time sensing, information gathering, collaborative and concurrent tasks. The results show how performance is affected by the existence or non-existence of dead-ends using three different performance metrics: efficiency (reciprocal of the cost), which is the optimization criteria of RAEplan and Plan-with-UPOM, success ratio and retry ratio. Acting purely reactively in the domains with dead ends can be costly and risky. The integration of acting and planning provided by RAE and a planning algorithm is of great benefit for all the domains, which is illustrated by a higher efficiency. In all of the domains, the efficiency generally increases with increase in the parameters, b and k of RAEplan, and n_{ro} and d of Plan-with-UPOM.

The retry ratio measures the number of times Retry is called per incoming task. 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 when RAE uses one of the planners. While most often the experimental evaluation of systems addressing acting and planning is simply performed on the sole planning functionality, we devised an *efficiency* measure to assess the overall performance to plan and act, including failure cases. This measure takes into account the cost to execute commands in the real world, which is usually much larger than the computation cost.

We have shown that the integration of acting and planning reduces the cost significantly and improves success ratio and retry ratio.

Our results show that Learnπ improves the performance of reactive RAE with respect to the three measures; RAE with UPOM and LearnH or with UPOM at unbounded depth improve significantly all the performance measures. Thanks to learning, the computational overhead remains acceptable for online procedure, since in this case a small number of rollouts bring already a good benefit.

An open-source code for the implementation of RAE, APEplan, RAEplan and Plan-with-UPOM and the test domains are available online.[1]

We developed a real-world prototype of RAE and Plan-with-UPOM by integrating it with a software-defined networking architecture, called AIRMAN, to defend against incoming attacks. Our prototype of RAE and UPOM communicates with AIRMAN via shared queues between Python processes. Our experimental results show that integrating AIRMAN, RAE and Plan-with-UPOM improves the estimated time, efficiency and retry-ratio of attack recovery with 95% confidence.

6.1 Looking Ahead: Limitations and Future Directions

While covering a range of refinement acting, planning and learning algorithms, we left a few pending issues and assumptions, whose discussion can be of help to the reader for using and deploying this material in a practical application.

[1] `https://bitbucket.org/sunandita/rae/`

6.1.1 Retrial in RAE

As mentioned earlier, Retry is not a backtracking procedure. Since RAE interacts with a dynamic world, Retry cannot go back to a previous state. It selects a method instance among those applicable in the *current* world state, except for those that have been tried before and failed. This restriction, of never repeating a previously tried method instance, may not always be necessary, since the same method instance that failed at some point may succeed later on. A full analysis of the conditions responsible for failures to make sure that they no longer hold can be complicated. One way to accomplish this is as follows. For example, consider the case for methods that are vulnerable to noisy sensing and execution contexts, and merit to be retried. This can be done by extending their parameters with arguments not needed for the logic of the method but that characterize the context (e.g., the pose of a sensor that may have changed between trials), while bounding the number of retrials.

Another future direction is allowing Retrial of actions. In RAE a method fails when one of its actions fails. But actions being non deterministic, it can be worthwhile retrying an action as assessed by its expected utility. Retrial of actions may be implemented after a full analysis and the computation of an optimal MDP policy[2], or simply with an ad-hoc loop on the execution-status of the actions that merit retrials. Furthermore, the body of a method being any procedure, complex

[2]This can be done with a sequence of dummy states s_{fail_i} such that the effects of action a in s include $s'_{fail_1} \in \gamma(s,a), \ldots, s'_{fail_{i+1}} \in \gamma(s'_{fail_i}, a)$; two actions are applicable to each s'_{fail_i}: a and stop-with-failure.

retrial loop can be specified. For example, a difficult grasp action in robotics may need several sequences of ⟨move, sense, grasp⟩ before succeeding or renouncing to the corresponding method.

6.1.2 Planning for multiple tasks at once

RAE supports multiple tasks running in parallel. Each task has its own refinement stack, and RAE calls a planner separately for each stack. However, all the stacks share a common state and indirectly affect each other. If we can take other active tasks into consideration while planning for a specific task, we may come up with more efficient plans. There are several ways to include the effect of tasks in other stacks, e.g.,

1. we come with ways to simulate the other tasks in the environment that the planners use. The simulation of tasks may follow the refinement tree that RAE would create;

2. we plan for all the current tasks together, i.e., instead of simulating only one stack, the planner may simulate the steps of all the tasks in RAE's *Agenda* and come up with a centralized plan.

6.1.3 Concurrency

The main loop of RAE progresses concurrently over several stacks in the *Agenda*, one for each top level task. All the domains we experimented with have several active stacks at once and involve concurrent tasks. However, in our current implementation possible conflicts and needed synchronizations are managed through the control statements and constraint checks inside the body of the refinement methods. To ease this process, it may be possible to enrich the body of methods with temporal and synchronization constructs, such as those used in TCA and TDL [13, 71], and rely on the execution-status of actions to handle waits. Since both UPOM and learning rely on simulated execution of the methods, they can support such extensions as long as Sample is able to simulate the duration of actions. More research would be needed to integrate to RAE and Plan-with-UPOM, extensions permitting the formal verification of concurrency property (liveness, deadlocks), e.g., as in the Petri-net based reactive system ASPiC [72].

Note that it is possible to extend RAE with refinement into concurrent subtasks (see [8, Sect. 3.2.4]).

6.1.4 Learning operational models

The Learnπ and LearnH procedures improve the decision making of RAE, with or without planning. But they don't learn how to construct refinement methods. They are also of help to a domain author, who does not need to design a minimal set of methods associated with a preference ordering. However, assistance in acquiring operational models, which are more detailed than the abstract descriptive models of planning (and which are always needed for acting), would be highly desirable. Let us discuss a few points about this important issue of future work.

Actions and methods, the two main components of operational models, would probably demand different learning techniques. Execution models of actions are domain dependent. For example, in robotics several approaches have been studied, e.g., [73, 74, 75, 76]. They usually rely on *Reinforcement Learning*(RL), possibly supervised and/or with inverse RL (see survey [77]). Other techniques for learning actions as low level skills can also be relevant, e.g., [78, 79]. These techniques would provide the procedure Sample, a corner stone in our approach: Sample(s, a) returns a state s' randomly drawn from $\gamma(s, a)$ according to the distribution of the outcomes of a in s. UPOM needs Sample (line 37 in Algorithm 6) to simulate the execution of methods in a rollout. Note that many application areas benefit from a domain simulator which can be very useful for learning action models and synthesizing the command's outcome sampling function, Sample.

Learning refinement methods have been addressed for HTN descriptive models, e.g., [51, 52, 53, 79, 80]. Our refinement methods for operational models can be significantly more complex. Possible investigation avenues for synthesizing these methods are: program synthesis techniques [81, 82], partial programming and RL [83, 84, 85], learning from the demonstrations of a tutor [86].

6.1.5 Benchmarking

The algorithms, RAE, APEplan, RAEplan and Plan-with-UPOM, were evaluated on five simulated domains having various different properties, including, concurrent tasks, dead ends, sensory actions and exogenous events. They performed well with respect to three different metrics: efficiency, retry ratio and success ratio. However, we don't have any emperical comparison of our refinement planning algorithms with the approaches we discuss in related work (Chapter 2) that integrated acting and planning (Section 2.2), or do hierarchical reinforcement learning (Section 2.5). Doing such a comparison is non-trivial and requires significant work because of the following challenges:

- Different approaches use different formalisms and languages to represent the test experimental domains. In order to do a comparison with some algorithm, our domains would need to be rewritten in the language that the algorithm supports. For example, to compare Plan-with-UPOM with planning for behavior trees [29], we would need to rewrite our refinement methods as behavior trees.

- In order to experiment with RAE and our refinement planners on other test domains, an expert would need to define tasks, commands and refinement methods. The performance of RAE and Plan-with-UPOM would depend heavily on how good these refinement methods are. If the refinement methods are too good, one might need very little planning and RAE will be successful really fast. On the other hand, if the refinement methods are bad without any control contructs, then reduces Plan-with-UPOM to the standard UCT search.

- Every approach to integrate planning and acting comes with its own set of assumptions, biases and scenarios it is meant for. If one or more test domains is written by the writer of one of the algorithms in the mix, it gives an unfair advantage to their algorithm. This is argued in [70] and the article suggests having the test domains designed by an independent third party.

- For hierarchical acting and planning, there are mainly two ways to represent the objective: tasks and goals. Task is an activity to be accomplished by the actor, and a goal is a final state that should be reached. Depending on a domain's properties and requirements, users can choose between task-based and goal-based approaches. Our hierarchical operational models are task-based. So, in order to compare with goal-based approaches, each task would need a corresponding goal and vice-versa. This mapping may be difficult in dynamic scenarios that involve closed-loop online decision making, because a task may correspond to an infinite number of possible goal states.

The International planning competition (IPC) for HTN planning tries to address some of the above challenges by enforcing some restrictions on the domain properties and the hierarchical task networks, e.g., all preconditions and effects can only contain literals, negated literals, conjunctions, and universal quantifiers, all actions have unit-cost, and methods may not contain any state constraits except in the preconditions. Domains may be either totally ordered, partially ordered or non-recursive, each enforcing a certain set of restrictions on the resulting task network. Similar constrains could be imposed on the hierarchical operational models to do a comparison with other approaches.

Appendix A: Description of APEplan

The main procedure of APEplan is shown in Table A.1. 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.

APEplan has three main sub procedures: APE-plan-task, APE-plan-method, and APE-plan-command. The first looks at b method instances for refining a task τ, calling APE-plan-method for each of the b method instances and returning the tree with the best *value*. Every refinement tree has a value based on probability and cost. Once APE-plan-task has chosen a method instance m for τ, it re labels the node n from τ to m in the current refinement tree T, then simulates the steps in m one by one by calling APE-plan-method.

This subroutine first checks whether the search has reached the maximum depth. If so, then it makes heuristic estimate of the cost and predicts the next state after going through the steps of 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

Table A.1: The pseudocode of APEplan and APE-plan-task, a sub-routine of APEplan. APEplan is the planner used by RAE.

```
APEplan $(M, s, \tau)$:
$n \leftarrow$ 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 $(U[m_{opt}], V[m_{opt}])$
```

is a task, then APE-plan-task is called for the task. If the step is a command, then it instead calls on APE-plan-command.

The APE-plan-command subroutine first calls SampleCommandOutcomes, which samples b' outcomes of the command com in the current state s. Samples are taken

Table A.2: 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 $T, value$
```

Table A.3: The pseudocode for NextStep and SampleCommandOutcomes. *pt = APE-plan-task, pc = APE-plan-command.

```
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 $res$
```

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.

The NextStep subroutine shown in Figure A.2 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.

Appendix B: Descriptions of Experimental Domains

B.1 Fetch domain

```
# the commands in the Fetch domain
declare_commands([put, take, perceive, charge, move, moveToEmergency,
↪  addressEmergency, wait, fail])

declare_task('search', 'r', 'o') # task and its parameters
declare_task('fetch', 'r', 'o')
declare_task('recharge', 'r', 'c')
declare_task('moveTo', 'r', 'l')
declare_task('emergency', 'r', 'l', 'i')
declare_task('nonEmergencyMove', 'r', 'l1', 'l2', 'dist')

# the refinement methods for the tasks
declare_methods('search', Search_Method1, Search_Method2)
declare_methods('fetch', Fetch_Method1, Fetch_Method2)
declare_methods('recharge', Recharge_Method1, Recharge_Method2,
↪  Recharge_Method3)
declare_methods('moveTo', MoveTo_Method1)
declare_methods('emergency', Emergency_Method1)
declare_methods('nonEmergencyMove', NonEmergencyMove_Method1)

# the commands
def take(r, o):
    state.load.AcquireLock(r)
    if state.load[r] == NIL:
        state.pos.AcquireLock(o)
        if state.loc[r] == state.pos[o]:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('take', start)
            ↪  == False):
                pass
            res = Sense('take')
            if res == SUCCESS:
                Simulate("Robot %s has picked up object %s\n" %(r,
                ↪  o))
                state.pos[o] = r
                state.load[r] = o
            else:
                Simulate("Non-deterministic event has made the take
                 ↪  command fail\n")
        else:
            Simulate("Robot %s is not at object %s's location\n" %(r,
             ↪  o))
            res = FAILURE
        state.pos.ReleaseLock(o)
    else:
        Simulate("Robot %s is not free to take anything\n" %r)
        res = FAILURE
    state.load.ReleaseLock(r)
    return res

def put(r, o):
    state.pos.AcquireLock(o)
    if state.pos[o] == r:
        start = globalTimer.GetTime()
        state.loc.AcquireLock(r)
        state.load.AcquireLock(r)
        while(globalTimer.IsCommandExecutionOver('put', start) ==
         ↪  False):
            pass
        res = Sense('put')
        if res == SUCCESS:
            Simulate("Robot %s has put object %s at location %d\n"
             ↪  %(r,o,state.loc[r]))
            state.pos[o] = state.loc[r]
            state.load[r] = NIL
        else:
            Simulate("Robot %s has failed to put %s because of some
             ↪  internal error")
        state.loc.ReleaseLock(r)
        state.load.ReleaseLock(r)
    else:
        Simulate("Object %s is not with robot %s\n" %(o,r))
        res = FAILURE
    state.pos.ReleaseLock(o)
    return res

def charge(r, c):
    state.loc.AcquireLock(r)
    state.pos.AcquireLock(c)
    if state.loc[r] == state.pos[c] or state.pos[c] == r:
        state.charge.AcquireLock(r)
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('charge', start) ==
         ↪  False):
            pass
        res = Sense('charge')
        if res == SUCCESS:
            state.charge[r] = 4
            Simulate("Robot %s is fully charged\n" %r)
        else:
            Simulate("Charging of robot %s failed due to some
             ↪  internal error.\n" %r)
        state.charge.ReleaseLock(r)
    else:
        Simulate("Robot %s is not in the charger's location or it
         ↪  doesn't have the charger with it\n" %r)
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.pos.ReleaseLock(c)
    return res

def moveToEmergency(r, l1, l2, dist):
    state.loc.AcquireLock(r)
    state.charge.AcquireLock(r)
    if l1 == l2:
        Simulate("Robot %s is already at location %s\n" %(r, l2))
        res = SUCCESS
    elif state.loc[r] == l1 and state.charge[r] >= dist:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('move', start) ==
         ↪  False):
           pass
        res = Sense('moveToEmergency')
        if res == SUCCESS:
            Simulate("Robot %s has moved from %d to %d\n" %(r, l1,
             ↪  l2))
            state.loc[r] = l2
            state.charge[r] = state.charge[r] - dist
        else:
            Simulate("Moving failed due to some internal error\n")
    elif state.loc[r] != l1 and state.charge[r] >= dist:
        Simulate("Robot %s is not in location %d\n" %(r, l1))
        res = FAILURE
    elif state.loc[r] == l1 and state.charge[r] < dist:
        Simulate("Robot %s does not have enough charge to move :(\n"
         ↪  %r)
        state.charge[r] = 0 # should we do this?
        res = FAILURE
    else:
        Simulate("Robot %s is not at location %s and it doesn't have
         ↪  enough charge!\n" %(r, l1))
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.charge.ReleaseLock(r)
    if res == FAILURE:
        state.emergencyHandling.AcquireLock(r)
        state.emergencyHandling[r] = False
        state.emergencyHandling.ReleaseLock(r)
    return res

def perceive(l):
    state.view.AcquireLock(l)
    if state.view[l] == False:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('perceive', start)
         ↪  == False):
            pass
        Sense('perceive')
        for c in state.containers[l]:
            state.pos.AcquireLock(c)
            state.pos[c] = l
            state.pos.ReleaseLock(c)
        state.view[l] = True
        Simulate("Perceived location %d\n" %l)
    else:
        Simulate("Already perceived\n")
    state.view.ReleaseLock(l)
    return SUCCESS

def move(r, l1, l2, dist):
    state.emergencyHandling.AcquireLock(r)
    if state.emergencyHandling[r] == False:
        state.loc.AcquireLock(r)
        state.charge.AcquireLock(r)
        if l1 == l2:
            Simulate("Robot %s is already at location %s\n" %(r, l2))
            res = SUCCESS
        elif state.loc[r] == l1 and (state.charge[r] >= dist or
         ↪  state.load[r] == 'c1'):
```

```
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('move', start)
             ↪  == False):
               pass
            res = Sense('move')
            if res == SUCCESS:
                Simulate("Robot %s has moved from %d to %d\n" %(r,
                 ↪  l1, l2))
                state.loc[r] = l2
                if state.load[r] != 'c1':
                    state.charge[r] = state.charge[r] - dist
            else:
                Simulate("Robot %s failed to move due to some
                 ↪  internal failure\n" %r)
        elif state.loc[r] != l1 and state.charge[r] >= dist:
            Simulate("Robot %s is not in location %d\n" %(r, l1))
            res = FAILURE
        elif state.loc[r] == l1 and state.charge[r] < dist:
            Simulate("Robot %s does not have enough charge to move
             ↪  :(\n" %r)
            state.charge[r] = 0 # should we do this?
            res = FAILURE
        else:
            Simulate("Robot %s is not at location %s and it doesn't
             ↪  have enough charge!\n" %(r, l1))
            res = FAILURE
        state.loc.ReleaseLock(r)
        state.charge.ReleaseLock(r)
    else:
        Simulate("Robot is addressing emergency so it cannot
         ↪  move.\n")
        res = FAILURE
    state.emergencyHandling.ReleaseLock(r)
    return res

def addressEmergency(r, l, i):
    state.loc.AcquireLock(r)
    state.emergencyHandling.AcquireLock(r)
    if state.loc[r] == l:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('addressEmergency',
         ↪  start) == False):
            pass
        res = Sense('addressEmergency')
        if res == SUCCESS:
            Simulate("Robot %s has addressed emergency %d\n" %(r, i))
        else:
            Simulate("Robot %s has failed to address emergency due to
             ↪  some internal error \n" %r)
    else:
        Simulate("Robot %s has failed to address emergency %d\n" %(r,
         ↪  i))
        res = FAILURE
    state.emergencyHandling[r] = False
    state.loc.ReleaseLock(r)
    state.emergencyHandling.ReleaseLock(r)
    return res

def wait(r):
    while(state.emergencyHandling[r] == True):
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('wait', start) ==
         ↪  False):
            pass
        Simulate("Robot %s is waiting for emergency to be over\n" %r)
        Sense('wait')
    return SUCCESS

# the refinement methods
def Recharge_Method3(r, c):
    """ Robot r charges and carries the charger with it """
    if state.loc[r] != state.pos[c] and state.pos[c] != r:
        if state.pos[c] in rv.LOCATIONS:
            do_task('moveTo', r, state.pos[c])
        else:
            robot = state.pos[c]
            do_command(put, robot, c)
            do_task('moveTo', r, state.pos[c])
    do_command(charge, r, c)
    do_command(take, r, c)

def Recharge_Method2(r, c):
    """ Robot r charges and does not carry the charger with it """
    if state.loc[r] != state.pos[c] and state.pos[c] != r:
        if state.pos[c] in rv.LOCATIONS:
            do_task('moveTo', r, state.pos[c])
        else:
            robot = state.pos[c]
            do_command(put, robot, c)
            do_task('moveTo', r, state.pos[c])
    do_command(charge, r, c)

def Recharge_Method1(r, c):
    """ When the charger is with another robot and that robot takes
    ↪  the charger back """
    robot = NIL
    if state.loc[r] != state.pos[c] and state.pos[c] != r:
        if state.pos[c] in rv.LOCATIONS:
            do_task('moveTo', r, state.pos[c])
        else:
            robot = state.pos[c]
            do_command(put, robot, c)
            do_task('moveTo', r, state.pos[c])
    do_command(charge, r, c)
    if robot != NIL:
        do_command(take, robot, c)

def Search_Method1(r, o):
    if state.pos[o] == UNK:
        toBePerceived = NIL
        for l in rv.LOCATIONS:
            if state.view[l] == False:
                toBePerceived = l
                break

        if toBePerceived != NIL:
            do_task('moveTo', r, toBePerceived)
            do_command(perceive, toBePerceived)
            if state.pos[o] == toBePerceived:
                if state.load[r] != NIL:
                    do_command(put, r, state.load[r])
                do_command(take, r, o)
            else:
                do_task('search', r, o)
        else:
            Simulate("Failed to search %s" %o)
            do_command(fail)
    else:
        Simulate("Position of %s is already known\n" %o)

def Search_Method2(r, o):
    if state.pos[o] == UNK:
        toBePerceived = NIL
        for l in rv.LOCATIONS:
            if state.view[l] == False:
                toBePerceived = l
                break

        if toBePerceived != NIL:
            do_task('recharge', r, 'c1') # is this allowed?
            do_task('moveTo', r, toBePerceived)
            do_command(perceive, toBePerceived)
            if state.pos[o] == toBePerceived:
                if state.load[r] != NIL:
                    do_command(put, r, state.load[r])
                do_command(take, r, o)
            else:
                do_task('search', r, o)
        else:
            Simulate("Failed to search %s" %o)
            do_command(fail)
    else:
        Simulate("Position of %s is already known\n" %o)

def Fetch_Method1(r, o):
    pos_o = state.pos[o]
    if pos_o == UNK:
        do_task('search', r, o)
    else:
        if state.loc[r] != pos_o:
            do_task('moveTo', r, pos_o)
        if state.load[r] != NIL:
            do_command(put, r, state.load[r])
        do_command(take, r, o)

def Fetch_Method2(r, o):
    pos_o = state.pos[o]
    if pos_o == UNK:
        do_task('search', r, o)
    else:
        if state.loc[r] != pos_o:
            do_task('recharge', r, 'c1')
            do_task('moveTo', r, pos_o)
        if state.load[r] != NIL:
            do_command(put, r, state.load[r])
        do_command(take, r, o)

def Emergency_Method1(r, l, i):
    if state.emergencyHandling[r] == False:
        state.emergencyHandling[r] = True
        load_r = state.load[r]
        if load_r != NIL:
            do_command(put, r, load_r)
        l1 = state.loc[r]
        dist = CR_GETDISTANCE(l1, l)
        do_command(moveToEmergency, r, l1, l, dist)
        do_command(addressEmergency, r, l, i)
    else:
        Simulate("%r is already busy handling another emergency\n"
         ↪  %r)
        do_command(fail)

def NonEmergencyMove_Method1(r, l1, l2, dist):
    if state.emergencyHandling[r] == False:
        do_command(move, r, l1, l2, dist)
    else:
        do_command(wait, r)
        do_command(move, r, l1, l2, dist)

def MoveTo_Method1(r, l):
    x = state.loc[r]
    dist = CR_GETDISTANCE(x, l)
    if state.charge[r] >= dist or state.load[r] == 'c1':
        do_task('nonEmergencyMove', r, x, l, dist)
    else:
        state.charge[r] = 0
        Simulate("Robot %s does not have enough charge to move from
         ↪  %d to %d\n" %(r, x, l))
        do_command(fail)
```

B.2 Explore domain

```
declare_commands([
    survey,
    monitor,
    screen,
    sample,
    process,
    charge,
    move,
    put,
    take,
    fly,
    deposit,
    transferData,
    handleAlien,
    fail])

declare_task('explore', 'r', 'activity', 'l')
declare_task('getEquipment', 'r', 'activity')
declare_task('flyTo', 'r', 'l')
declare_task('moveTo', 'r', 'l')
declare_task('recharge', 'r')
declare_task('depositData', 'r')
declare_task('doActivities', 'r', 'actList')
declare_task('handleEmergency', 'r', 'l')

declare_methods('explore',
    Explore_Method1)

declare_methods('getEquipment',
    GetEquipment_Method1,
    GetEquipment_Method2,
    GetEquipment_Method3)

declare_methods('moveTo',
    MoveTo_Method1)

declare_methods('flyTo',
    FlyTo_Method1,
    FlyTo_Method2)

declare_methods('recharge',
    Recharge_Method1,
    Recharge_Method2)

declare_methods('depositData',
    DepositData_Method1,
    DepositData_Method2)

declare_methods('doActivities',
    DoActivities_Method1,
    DoActivities_Method2,
    DoActivities_Method3)

declare_methods('handleEmergency',
    HandleEmergency_Method2,
    HandleEmergency_Method1,
    HandleEmergency_Method3)
```

```
# commands
def survey(r, l):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.data.AcquireLock(r)
    e = state.load[r]
    if e not in rv.EQUIPMENTTYPE:
        Simulate("%s does not have any equipment\n" %r)
        res = FAILURE
    elif state.loc[r] == l and rv.EQUIPMENTTYPE[e] == 'survey' and
     ↪  state.data[r] < 4:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('survey', start) ==
         ↪  False):
            pass
        res = Sense('survey')
        if res == SUCCESS:
            state.data[r] += 1
            Simulate("%s has surveyed the location %s\n" %(r, l))
        else:
            Simulate("%s has failed to do survey %s due to an
             ↪  internal error.\n" %(r,l))
    elif state.loc[r] != l:
        Simulate("%s is not in location %s\n" %(r, l))
        res = FAILURE
    elif rv.EQUIPMENTTYPE[e] != 'survey':
        Simulate("%s is not the right equipment for survey\n" %e)
        res = FAILURE
    elif state.data[r] == 4:
        Simulate("%s cannot store any more data\n" %r)
        res = FAILURE
    state.load.ReleaseLock(r)
    state.loc.ReleaseLock(r)
    state.data.ReleaseLock(r)
    return res

def monitor(r, l):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.data.AcquireLock(r)
    e = state.load[r]
    if e not in rv.EQUIPMENTTYPE:
        Simulate("%s does not have any equipment\n" %r)
        res = FAILURE
    elif state.loc[r] == l and rv.EQUIPMENTTYPE[e] == 'monitor' and r
     ↪  != 'UAV' and state.data[r] < 4:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('monitor', start) ==
         ↪  False):
            pass
        res = Sense('monitor')
        if res == SUCCESS:
            Simulate("%s has monitored the location\n" %r)
            state.data[r] += 1
        else:
            Simulate("Monitoring has failed due to some internal
             ↪  error\n")
    elif state.loc[r] != l:
        Simulate("%s is not in location %s\n" %(r, l))
        res = FAILURE
    elif rv.EQUIPMENTTYPE[e] != 'monitor':
        Simulate("%s is not the right equipment for monitor\n" %e)
        res = FAILURE
    elif r == 'UAV':
        Simulate("UAV cannot monitor\n")
        res = FAILURE
    elif state.data[r] == 4:
        Simulate("%s cannot store any more data\n" %r)
        res = FAILURE
    state.load.ReleaseLock(r)
    state.loc.ReleaseLock(r)
    state.data.ReleaseLock(r)
    return res

def screen(r, l):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.data.AcquireLock(r)
    e = state.load[r]
    if e not in rv.EQUIPMENTTYPE:
        Simulate("%s does not have any equipment\n" %r)
        res = FAILURE
    elif state.loc[r] == l and rv.EQUIPMENTTYPE[e] == 'screen' and r
     ↪  != 'UAV' and state.data[r] < 4:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('screen', start) ==
         ↪  False):
            pass
        res = Sense('screen')
```

```
            if res == SUCCESS:
                Simulate("%s has screened the location\n" %r)
                state.data[r] += 1
            else:
                Simulate("Screening failed due to some internal error\n")
        elif state.loc[r] != l:
            Simulate("%s is not in location %s\n" %(r, l))
            res = FAILURE
        elif rv.EQUIPMENTTYPE[e] != 'screen':
            Simulate("%s is not the right equipment for screening\n" %e)
            res = FAILURE
        elif r == 'UAV':
            Simulate("UAV cannot do screening\n")
            res = FAILURE
        elif state.data[r] == 4:
            Simulate("%s cannot store any more data\n" %r)
            res = FAILURE
        state.load.ReleaseLock(r)
        state.loc.ReleaseLock(r)
        state.data.ReleaseLock(r)
        return res

    def sample(r, l):
        state.load.AcquireLock(r)
        state.loc.AcquireLock(r)
        state.data.AcquireLock(r)
        e = state.load[r]
        if e not in rv.EQUIPMENTTYPE:
            Simulate("%s does not have any equipment\n" %r)
            res = FAILURE
        elif state.loc[r] == l and rv.EQUIPMENTTYPE[e] == 'sample' and r
         ↪  != 'UAV' and state.data[r] < 4:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('sample', start) ==
             ↪  False):
                pass
            res = Sense('sample')
            if res == SUCCESS:
                Simulate("%s has sampled the location\n" %r)
                state.data[r] += 1
            else:
                Simulate("Sampling failed due to internal error\n")
        elif state.loc[r] != l:
            Simulate("%s is not in location %s\n" %(r, l))
            res = FAILURE
        elif rv.EQUIPMENTTYPE[e] != 'sample':
            Simulate("%s is not the right equipment for sampling\n" %e)
            res = FAILURE
        elif r == 'UAV':
            Simulate("UAV cannot do sampling\n")
            res = FAILURE
        elif state.data[r] == 4:
            Simulate("%s cannot store any more data\n" %r)
            res = FAILURE
        state.load.ReleaseLock(r)
        state.loc.ReleaseLock(r)
        state.data.ReleaseLock(r)
        return res

    def process(r, l):
        state.load.AcquireLock(r)
        state.loc.AcquireLock(r)
        state.data.AcquireLock(r)
        e = state.load[r]
        if e not in rv.EQUIPMENTTYPE:
            Simulate("%s does not have any equipment\n" %r)
            res = FAILURE
        elif state.loc[r] == l and rv.EQUIPMENTTYPE[e] == 'process' and r
         ↪  != 'UAV' and state.data[r] < 4:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('process', start) ==
             ↪  False):
                pass
            res = Sense('process')
            if res == SUCCESS:
                Simulate("%s has processed the location\n" %r)
                state.data[r] += 1
            else:
                Simulate("Processing failed due to an internal error\n")
        elif state.loc[r] != l:
            Simulate("%s is not in location %s\n" %(r, l))
            res = FAILURE
        elif rv.EQUIPMENTTYPE[e] != 'process':
            Simulate("%s is not the right equipment for process\n" %e)
            res = FAILURE
        elif r == 'UAV':
            Simulate("UAV cannot do processing\n")
            res = FAILURE
        elif state.data[r] == 4:
            Simulate("%s cannot store any more data\n" %r)
```

```
            res = FAILURE
        state.load.ReleaseLock(r)
        state.loc.ReleaseLock(r)
        state.data.ReleaseLock(r)
        return res

def alienSpotted(l):
    Simulate("An alien is spotted in location %s \n" %l)
    return SUCCESS

def handleAlien(r, l):
    if state.loc[r] == l:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('handleAlien',
         ↪  start) == False):
            pass
        Simulate("Robot %s is negotiating with alien.\n" %r)
        res = SUCCESS
    else:
        Simulate("Robot %s is not in alien's location\n" %r)
        res = FAILURE
    return res

def charge(r, c):
    state.loc.AcquireLock(r)
    state.pos.AcquireLock(c)
    if state.loc[r] == state.pos[c] or state.pos[c] == r:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('charge', start) ==
         ↪  False):
            pass
        res = Sense('charge')
        if res == SUCCESS:
            state.charge.AcquireLock(r)
            state.charge[r] = 100
            Simulate("%s is fully charged\n" %r)
            state.charge.ReleaseLock(r)
        else:
            Simulate("Charging failed due to some internal error.\n")
    else:
        Simulate("%s is not in the charger's location or it doesn't
         ↪  have the charger with it\n" %r)
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.pos.ReleaseLock(c)
    return res

def move(r, l1, l2):
    state.loc.AcquireLock(r)
    state.charge.AcquireLock(r)
    dist = EE_GETDISTANCE(l1, l2)
    if l1 == l2:
        Simulate("%s is already at location %s\n" %(r, l2))
        res = SUCCESS
    elif state.loc[r] == l1 and state.charge[r] >= dist:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('move', start) ==
         ↪  False):
            pass
        res = Sense('move')
        if res == SUCCESS:
            Simulate("%s has moved from %s to %s\n" %(r, l1, l2))
            state.loc[r] = l2
            state.charge[r] = state.charge[r] - dist
        else:
            Simulate("Move failed due to an internal error.\n")
    elif state.loc[r] != l1 and state.charge[r] >= dist:
        Simulate("%s is not in location %s\n" %(r, l1))
        res = FAILURE
    elif state.loc[r] == l1 and state.charge[r] < dist:
        Simulate("%s does not have any charge to move :(\n" %r)
        res = FAILURE
    else:
        Simulate("%s is not at location %s and it doesn't have enough
         ↪  charge!\n" %(r, l1))
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.charge.ReleaseLock(r)
    return res

def fly(r, l1, l2):
    state.loc.AcquireLock(r)
    state.charge.AcquireLock(r)

    dist = EE_GETDISTANCE(l1, l2)/2
    if r != 'UAV':
        Simulate("%s cannot fly\n" %r)
        res = FAILURE
    elif l1 == l2:
        Simulate("%s is already at location %s\n" %(r, l2))
```

```
            res = SUCCESS
        elif state.loc[r] == l1 and state.charge[r] >= dist:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('fly', start) ==
             ↪  False):
                pass
            res = SenseFly()
            if res == SUCCESS:
                Simulate("%s has flied from %s to %s\n" %(r, l1, l2))
                state.loc[r] = l2
                state.charge[r] = state.charge[r] - dist
            else:
                Simulate("Flying failed due to an internal error.\n")
        elif state.loc[r] != l1 and state.charge[r] >= dist:
            Simulate("%s is not in location %s\n" %(r, l1))
            res = FAILURE
        elif state.loc[r] == l1 and state.charge[r] < dist:
            Simulate("%s does not have any charge to fly :( charge = %d
             ↪  but distance = %d \n" %(r,state.charge[r], dist))
            state.charge[r] = 0
            res = FAILURE
        else:
            Simulate("%s is not at location %s and it doesn't have enough
             ↪  charge!\n" %(r, l1))
            res = FAILURE
        state.loc.ReleaseLock(r)
        state.charge.ReleaseLock(r)
        return res

    def take(r, o):
        state.load.AcquireLock(r)
        if state.load[r] == NIL:
            state.loc.AcquireLock(r)
            state.pos.AcquireLock(o)
            if state.loc[r] == state.pos[o]:
                start = globalTimer.GetTime()
                while(globalTimer.IsCommandExecutionOver('take', start)
                 ↪  == False):
                    pass
                res = Sense('take')
                if res == SUCCESS:
                    Simulate("%s has picked up %s\n" %(r, o))
                    state.pos[o] = r
                    state.load[r] = o
                else:
```

```
                Simulate("Take failed due to an internal failure.\n")
        else:
            Simulate("%s is not at %s's location\n" %(r, o))
            res = FAILURE
        state.loc.ReleaseLock(r)
        state.pos.ReleaseLock(o)
    else:
        Simulate("%s is not free to take anything\n" %r)
        res = FAILURE
    state.load.ReleaseLock(r)
    return res

def put(r, o):
    state.loc.AcquireLock(r)
    state.pos.AcquireLock(o)
    if state.pos[o] == r:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('put', start) ==
         ↪  False):
            pass
        res = Sense('put')
        if res == SUCCESS:
            state.pos[o] = state.loc[r]
            state.load[r] = NIL
            Simulate("%s has put %s at location %s\n"
             ↪  %(r,o,state.loc[r]))
        else:
            Simulate("put failed due to an internal error.\n")
    else:
        Simulate("%s is not with %s\n" %(o,r))
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.pos.ReleaseLock(o)
    return res

def deposit(r):
    state.loc.AcquireLock(r)
    state.data.AcquireLock(r)
    if state.loc[r] == 'base':
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('deposit', start) ==
         ↪  False):
            pass
        res = Sense('deposit')
        if res == SUCCESS:
            Simulate("%s has deposited data in the base\n" %r)
            state.data[r] = 0
        else:
            Simulate("Deposit failed due to an internal error.\n")
    else:
        Simulate("%s is not in base, so it cannot deposit data.\n"
         ↪  %r)
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.data.ReleaseLock(r)
    return res

def transferData(r1, r2):
    state.loc.AcquireLock(r1)
    state.loc.AcquireLock(r2)
    state.data.AcquireLock(r1)
    state.data.AcquireLock(r2)

    if state.loc[r1] != state.loc[r2]:
        Simulate("%s and %s are not in same location.\n" %(r1, r2))
        res = FAILURE
    elif state.data[r2] + state.data[r1] <= 4:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('transferData',
         ↪  start) == False):
            pass
        res = Sense('transferData')
        if res == SUCCESS:
            Simulate("%s transfered data to %s\n" %(r1, r2))
            state.data[r2] += state.data[r1]
            state.data[r1] = 0
        else:
            Simulate("Transfer data failed due to an internal
             ↪  error.\n")

    elif state.data[r2] < 4:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('transferData',
         ↪  start) == False):
            pass
        res = Sense('transferData')
        if res == SUCCESS:
            t = 4 - state.data[r2]
            state.data[r2] = 4
            state.data[r1] -= t
            Simulate("%s transfered data to %s\n" %(r1, r2))
        else:
            Simulate("Transfer data failed due to an internal
             ↪  error.\n")
    else:
        Simulate("There is no space in %s to get data\n" %r2)
        res = FAILURE

    state.loc.ReleaseLock(r1)
    state.loc.ReleaseLock(r2)
    state.data.ReleaseLock(r1)
    state.data.ReleaseLock(r2)
    return res

# refinement methods
def Explore_Method1(r, activity, l):
    do_task('getEquipment', r, activity)
    do_task('moveTo', r, l)
    if activity == 'survey':
        do_command(survey, r, l)
    elif activity == 'monitor':
        do_command(monitor, r, l)
    elif activity == 'screen':
        do_command(screen, r, l)
    elif activity == 'sample':
        do_command(sample, r, l)
    elif activity == 'process':
        do_command(process, r, l)

def GetEquipment_Method1(r, activity):
    """ When the equipment is at a particular location and r does
     ↪  not carry any load"""
    e = rv.EQUIPMENT[activity]
    if state.load[r] != e and state.pos[e] in rv.LOCATIONS:
        do_task('moveTo', r, state.pos[e])
        do_command(take, r, e)
    else:
        do_command(fail)

def GetEquipment_Method2(r, activity):
    """ When r is already carrying some load and equipment is at a
     ↪  particular location"""
    e = rv.EQUIPMENT[activity]
    if state.load[r] != e and state.pos[e] in rv.LOCATIONS:
        do_task('moveTo', r, state.pos[e])
        state.load.AcquireLock(r)
        if state.load[r] != NIL:
            o = state.load[r]
            state.load.ReleaseLock(r)
            do_command(put, r, o)
        else:
            state.load.ReleaseLock(r)
            do_command(fail)
        do_command(take, r, e)
    else:
        do_command(fail)

def GetEquipment_Method3(r, activity):
    """ When the equipment is with another robot """
    r1 = r
    e = rv.EQUIPMENT[activity]
    if state.load[r1] != e:
        loc = state.pos[e]
        if loc not in rv.LOCATIONS:
            r2 = loc
            do_task('moveTo', r, state.loc[r2])
            state.load.AcquireLock(r1)
            if state.load[r1] != NIL:
                x = state.load[r1]
                state.load.ReleaseLock(r1)
                do_command(put, r1, x)
            else:
                state.load.ReleaseLock(r1)
            do_command(put, r2, e)
            do_command(take, r1, e)
        else:
            do_command(fail)
    else:
        do_command(fail)

def MoveTo_MethodHelper(r, l):
    path = EE_GETPATH(state.loc[r], l)
    if path == {}:
        Simulate("%s is already at location %s \n" %(r, l))
    else:
        lTemp = state.loc[r]
        if lTemp not in path:
            Simulate("%s is out of its path to %s\n" %(r, l))
            do_command(fail)
        else:
            while(lTemp != l):
                lNext = path[lTemp]
                do_command(move, r, lTemp, lNext)
                if lNext != state.loc[r]:
                    Simulate("%s is out of its path to %s\n" %(r, l))
                    do_command(fail)
                else:
                    lTemp = lNext

def MoveTo_Method1(r, l):
    if l not in rv.LOCATIONS:
        Simulate("%s is trying to go to an invalid location\n" %r)
        do_command(fail)
    else:
        MoveTo_MethodHelper(r, l)

def FlyTo_Method1(r, l):
    if r == 'UAV':
        do_command(fly, r, state.loc[r], l)
    else:
        Simulate("%s is not a UAV. So, it cannot fly\n" %r)
        do_command(fail)

def FlyTo_Method2(r, l):
    dist = EE_GETDISTANCE(state.loc[r], l)
    if r == 'UAV':
        do_task('recharge', r)
        do_command(fly, r, state.loc[r], l)
    else:
        Simulate("%s is not a UAV. So, it cannot fly\n" %r)
        do_command(fail)

def DepositData_Method1(r):
    if state.data[r] > 0:
        do_task('moveTo', r, 'base')
        do_command(deposit, r)
    else:
        Simulate("%s has no data to deposit.\n" %r)
        do_command(fail)

def DepositData_Method2(r):
    if state.data[r] > 0:
        if r != 'UAV':
            do_task('flyTo', 'UAV', state.loc[r])
            do_command(transferData, r, 'UAV')
        do_task('flyTo', 'UAV', 'base')
        do_command(deposit, 'UAV')
    else:
        Simulate("%s has no data to deposit.\n" %r)
        do_command(fail)

def Recharge_Method1(r):
    c = 'c1'
    l1 = state.loc[r]
    if state.pos[c] != l1 and state.pos[c] != r:
        if state.pos[c] in rv.LOCATIONS:
            l2 = state.pos[c]
        else:
            r2 = state.pos[c]
            l2 = state.loc[r2]
        dist = EE_GETDISTANCE(l1, l2)
        if state.charge[r] >= dist:
            do_task('moveTo', r, l2)
            do_command(charge, r, c)
        else:
            Simulate("%s is stranded without any possibility of
             ↪  charging\n" %r)
            do_command(fail)
    else:
        do_command(charge, r, c)

def Recharge_Method2(r):
    c = 'c1'
    l1 = state.loc[r]
    if state.pos[c] != l1 and state.pos[c] != r:
        if state.pos[c] in rv.LOCATIONS:
            l2 = state.pos[c]
        else:
            r2 = state.pos[c]
            l2 = state.loc[r2]
        dist = EE_GETDISTANCE(l1, l2)
        if state.charge[r] >= dist:
            do_task('moveTo', r, l2)
            do_command(charge, r, c)
            do_command(take, r, c)
        else:
            Simulate("%s is stranded without any possibility of
             ↪  charging\n" %r)
            do_command(fail)
    else:
        do_command(charge, r, c)
        do_command(take, r, c)

def DoActivities_Method1(r, actList):
    for act in actList:
        do_task('explore', r, act[0], act[1])
    do_task('depositData', r)

def DoActivities_Method3(r, actList):
    for act in actList:
        do_task('explore', r, act[0], act[1])
        do_task('depositData', r)
        do_task('recharge', r)

def DoActivities_Method2(r, actList):
    for act in actList:
        do_task('explore', r, act[0], act[1])
        do_task('recharge', r)
    do_task('depositData', r)

def HandleEmergency_Method1(r, l):
    do_task('recharge', r)
    do_task('moveTo', r, l)
    do_command(handleAlien, r, l)

def HandleEmergency_Method2(r, l):
    do_task('moveTo', r, l)
    do_command(handleAlien, r, l)

def HandleEmergency_Method3(r, l):
    do_task('flyTo', r, l)
    do_command(handleAlien, r, l)
```

B.3 Navigate domain

```
declare_commands([
    holdDoor,
    passDoor,
    releaseDoor,
    move,
    put,
    take,
    unlatch1,
    unlatch2,
    helpRobot,
    fail],)

declare_task('fetch', 'r', 'o', 'l')
declare_task('moveTo', 'r', 'l')
declare_task('moveThroughDoorway', 'r', 'd', 'l')
declare_task('unlatch', 'r', 'd')
declare_task('collision', 'r')

declare_methods('fetch', Fetch_Method1)
declare_methods('moveTo', MoveTo_Method1)
declare_methods('moveThroughDoorway',
    MoveThroughDoorway_Method1,
    MoveThroughDoorway_Method3,
    MoveThroughDoorway_Method4,
    MoveThroughDoorway_Method2) # has multiple method instances
declare_methods('unlatch', Unlatch_Method1, Unlatch_Method2)
declare_methods('collision', Recover_Method1) # has multiple
↪   instances

# commands
def helpRobot(r1, r2):
    if state.loc[r1] == state.loc[r2]:
        Simulate("%s is helping %s \n" %(r1, r2))
        return SUCCESS
    else:
        return FAILURE

def unlatch1(r, d):
    state.load.AcquireLock(r)
    state.doorStatus.AcquireLock(d)
    if state.doorStatus[d] != 'closed':
        Simulate("Door %s is already open\n" %d)
        res = SUCCESS
    elif state.load[r] == NIL:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('unlatch1', start)
        ↪   == False):
                pass
        res = Sense('unlatch1')
        if res == SUCCESS:
            Simulate("Robot %s has opened door %s\n" %(r, d))
            state.doorStatus[d] = 'opened'
        else:
            Simulate("Unlatching has failed due to an internal
             ↪  error\n")
    else:
        Simulate("Robot %s is not free to open door %s\n" %(r, d))
        res = FAILURE
    state.load.ReleaseLock(r)
    state.doorStatus.ReleaseLock(d)
    return res

def unlatch2(r, d):
    state.load.AcquireLock(r)
    state.doorStatus.AcquireLock(d)
    if state.doorStatus[d] != 'closed': # status can be closed,
     ↪  opened or held
        Simulate("Door %s is already open\n" %d)
        res = SUCCESS
    elif state.load[r] == NIL:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('unlatch2', start)
         ↪  == False):
            pass
        res = Sense('unlatch2')
        if res == SUCCESS:
            Simulate("Robot %s has opened door %s\n" %(r, d))
            state.doorStatus[d] = 'opened'
        else:
            Simulate("Unlatching has failed due to an internal
             ↪  error\n")
    else:
        Simulate("Robot %s is not free to open door %s\n" %(r, d))
        res = FAILURE
    state.load.ReleaseLock(r)
    state.doorStatus.ReleaseLock(d)
    return res

def passDoor(r, d, l):
    state.doorStatus.AcquireLock(d)
    state.loc.AcquireLock(r)
    if state.doorStatus[d] == 'opened' or state.doorStatus[d] ==
     ↪  'held':
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('passDoor', start)
         ↪  == False):
                pass
        res = Sense('passDoor', d)
        if res == SUCCESS:
            state.loc[r] = l
            Simulate("Robot %s has passed the door %s\n" %(r, d))
        else:
            Simulate("Robot %s is not able to pass door %s\n" %(r,
             ↪  d))
        state.doorType[d] = rv.DOORTYPES[d]
    else:
        Simulate("Robot %s is not able to pass door %s\n" %(r, d))
        res = FAILURE
    state.loc.ReleaseLock(r)
    state.doorStatus.ReleaseLock(d)
    return res

# should always be followed by releaseDoor
def holdDoor(r, d):
    state.doorStatus.AcquireLock(d)
    state.load.AcquireLock(r)
    if state.doorStatus[d] != 'closed' and state.load[r] == NIL:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('holdDoor', start)
         ↪  == False):
            pass
        Simulate("Robot %s is holding the door %s\n" %(r, d))
        state.load[r] = 'H'
        state.doorStatus[d] = 'held'
        res = SUCCESS
    elif state.doorStatus[d] == 'closed':
        Simulate("Door %s is closed and cannot be held by %s\n" %(d,
         ↪  r))
        res = FAILURE
    elif state.load[r] != NIL:
        Simulate("Robot %s is not free to hold the door %s\n" %(r,
         ↪  d))
        res = FAILURE
    state.doorStatus.ReleaseLock(d)
    state.load.ReleaseLock(r)
    return res

def releaseDoor(r, d):
    if state.doorStatus[d] != 'held':
        return SUCCESS
    elif state.doorStatus[d] == 'held' and state.load[r] == 'H':
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('releaseDoor',
         ↪  start) == False):
                pass
        Simulate("Robot %s has released the the door %s\n" %(r, d))
        state.doorStatus[d] = 'closed'
        state.load[r] = NIL
    else:
        Simulate("Robot %s is not holding door %s\n" %(r, d))
    return SUCCESS

def move(r, l1, l2):
    state.loc.AcquireLock(r)
    if l1 == l2:
        Simulate("Robot %s is already at location %s\n" %(r, l2))
        res = SUCCESS
    elif state.loc[r] == l1:
        if (l1, l2) in rv.DOORLOCATIONS or (l2, l1) in
         ↪  rv.DOORLOCATIONS:
            Simulate("Robot %s cannot move. There is a door between
             ↪  %s and %s \n" %(r, l1, l2))
            res = FAILURE
        else:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('move', start)
             ↪  == False):
                pass
            res = Sense('move')
            if res == SUCCESS:
                Simulate("Robot %s has moved from %d to %d\n" %(r,
                 ↪  l1, l2))
                state.loc[r] = l2
            else:
                Simulate("Move has failed due to some internal
                 ↪  failure.\n")
    else:
        Simulate("Invalid move by robot %s\n" %r)
        res = FAILURE
    state.loc.ReleaseLock(r)
    return res

def put(r, o):
    state.pos.AcquireLock(o)
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    if state.pos[o] == r:
        start = globalTimer.GetTime()
        while(globalTimer.IsCommandExecutionOver('put', start) ==
         ↪  False):
                pass
        res = Sense('put')
        if res == SUCCESS:
            state.pos[o] = state.loc[r]
            state.load[r] = NIL
            Simulate("Robot %s has put object %s at location %d\n"
             ↪  %(r,o,state.loc[r]))
        else:
            Simulate("put has failed due to some internal
             ↪  failure.\n")
    else:
        Simulate("Object %s is not with robot %s\n" %(o,r))
        res = FAILURE
    state.pos.ReleaseLock(o)
    state.load.ReleaseLock(r)
    state.loc.ReleaseLock(r)
    return res

def take(r, o):
    state.pos.AcquireLock(o)
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    if state.load[r] == NIL:
        if state.loc[r] == state.pos[o]:
            start = globalTimer.GetTime()
            while(globalTimer.IsCommandExecutionOver('take', start)
             ↪  == False):
                    pass
            res = Sense('take')
            if res == SUCCESS:
                Simulate("Robot %s has picked up object %s\n" %(r,
                 ↪  o))
                state.pos[o] = r
                state.load[r] = o
            else:
                Simulate("take failed due to some internal error.\n")
        elif state.loc[r] != state.pos[o]:
            Simulate("Robot %s is not at object %s's location\n" %(r,
             ↪  o))
            res = FAILURE
    else:
        Simulate("Robot %s is not free to take anything\n" %r)
            res = FAILURE
        state.pos.ReleaseLock(o)
        state.load.ReleaseLock(r)
        state.loc.ReleaseLock(r)
        return res

# refinement methods
def MoveThroughDoorway_Method3(r, d, l):
    """ For a robot passing a spring door without any load """
    if state.load[r] == NIL and (state.doorType[d] == 'spring' or
    ↪  state.doorType[d] == UNK):
        do_task('unlatch', r, d)
        do_command(holdDoor, r, d)
        do_command(passDoor, r, d, l)
        do_command(releaseDoor, r, d)
    else:
        do_command(fail)

def MoveThroughDoorway_Method2(r, d, l, r2):
    """ For a robot passing a spring door with a load """
    if state.load[r] != NIL and (state.doorType[d] == 'spring' or
    ↪  state.doorType[d] == UNK):
        state.status.AcquireLock(r2)
        if state.status[r2] == 'free':
            state.status[r2] = 'busy'
            state.status.ReleaseLock(r2)
        else:
            state.status.ReleaseLock(r2)
            do_command(fail)

        obj = state.load[r2]
        if obj != NIL:
            if obj != 'H':
                do_command(put, r2, state.load[r2])
            else:
                do_command(fail)
        do_task('moveTo', r2, state.loc[r])
        do_task('unlatch', r2, d)
        do_command(holdDoor, r2, d)
        do_command(passDoor, r, d, l)
        do_command(releaseDoor, r2, d)
        state.status[r2] = 'free'
    else:
        do_command(fail)
```

```
MoveThroughDoorway_Method2.parameters = "[(r2,) for r2 in rv.ROBOTS
↪  if r2 != r and state.status[r2] == 'free']"

def MoveThroughDoorway_Method4(r, d, l):
    """ For a robot passing a normal door with a load """
    if state.load[r] != NIL and (state.doorType[d] == 'ordinary' or
    ↪  state.doorType[d] == UNK):
        obj = state.load[r]
        if obj != 'H':
            do_command(put, r, obj)
        else:
            Simulate("%r is holding another door\n" %r)
            do_command(fail)
        do_task('unlatch', r, d)
        do_command(take, r, obj)
        do_command(passDoor, r, d, l)
    else:
        do_command(fail)

def MoveThroughDoorway_Method1(r, d, l):
    """ For a robot passing a normal door without a load """
    if state.load[r] == NIL and (state.doorType[d] == 'ordinary' or
    ↪  state.doorType[d] == UNK):
        do_task('unlatch', r, d)
        do_command(passDoor, r, d, l)
    else:
        do_command(fail)

def MoveTo_Method1(r, l):
    x = state.loc[r]
    if l in rv.LOCATIONS:
        path = SD_GETPATH(x, l)
        if path == None:
            Simulate("Unsolvable problem. No path exists.\n")
            do_command(fail)
        if path == {}:
            Simulate("Robot %s is already at location %s \n" %(r, l))
        else:
            lTemp = x
            lNext = path[lTemp]
            while(lTemp != l):
                lNext = path[lTemp]
                if (lTemp, lNext) in rv.DOORLOCATIONS or (lNext,
                ↪  lTemp) in rv.DOORLOCATIONS:
                    d = SD_GETDOOR(lTemp, lNext)
                    do_task('moveThroughDoorway', r, d, lNext)
                else:
                    do_command(move, r, lTemp, lNext)
                if lNext != state.loc[r]:
                    do_command(fail)
                else:
                    lTemp = lNext
    elif l in rv.ROBOTS: # maybe not used?
        loc = state.loc[l]
        do_task('moveTo', r, loc)
    else:
        Simulate("Robot %s going to invalid location.\n" %(r))
        do_command(fail)

def Fetch_Method1(r, o, l):
    state.status.AcquireLock(r)
    if state.status[r] == 'free':
        state.status[r] = 'busy'
        state.status.ReleaseLock(r)
    else:
        state.status.ReleaseLock(r)
        do_command(fail)

    do_task('moveTo', r, state.pos[o])
    do_command(take, r, o)
    do_task('moveTo', r, l)
    state.status[r] = 'free'

def Recover_Method1(r, r2): # multiple instances
    state.status.AcquireLock(r2)

    if state.status[r2] == 'busy':
        state.status.ReleaseLock(r2)
        do_command(fail)
    else:
        state.status[r2] = 'busy'
        state.status.ReleaseLock(r2)
        do_task('moveTo', r2, state.loc[r])
        do_command(helpRobot, r2, r)
        state.status[r2] = 'free'
        Simulate("Robot %s is helping %s to recover from collision\n"
         ↪  %(r2, r))
Recover_Method1.parameters = "[(r2,) for r2 in rv.ROBOTS if r2 != r
↪  and state.status[r2] == 'free']"
```

```
def Unlatch_Method1(r, d):
    do_command(unlatch1, r, d)

def Unlatch_Method2(r, d):
    do_command(unlatch2, r, d)
```

B.4 Rescue domain

```
declare_commands([
    moveEuclidean,
    moveCurved,
    moveManhattan,
    fly,
    giveSupportToPerson,
    clearLocation,
    inspectLocation,
    inspectPerson,
    transfer,
    replenishSupplies,
    captureImage,
    changeAltitude,
    deadEnd,
    fail
    ])

declare_task('moveTo', 'r', 'l')
declare_task('rescue', 'r', 'p')
declare_task('helpPerson', 'r', 'p')
declare_task('getSupplies', 'r')
declare_task('survey', 'r', 'l')
declare_task('getRobot')
declare_task('adjustAltitude', 'r')

declare_methods('moveTo',
    MoveTo_Method4,
    MoveTo_Method3,
    MoveTo_Method2,
    MoveTo_Method1,
    )

declare_methods('rescue',
    Rescue_Method1,
    Rescue_Method2,
    )
```

```
declare_methods('helpPerson',
    HelpPerson_Method2,
    HelpPerson_Method1,
    )

declare_methods('getSupplies',
    GetSupplies_Method2,
    GetSupplies_Method1,
    )

declare_methods('survey',
    Survey_Method1,
    Survey_Method2
    )

declare_methods('getRobot',
    GetRobot_Method1,
    GetRobot_Method2,
    )

declare_methods('adjustAltitude',
    AdjustAltitude_Method1,
    AdjustAltitude_Method2,
    )

# commands

def moveEuclidean(r, l1, l2, dist):
    (x1, y1) = l1
    (x2, y2) = l2
    xlow = min(x1, x2)
    xhigh = max(x1, x2)
    ylow = min(y1, y2)
    yhigh = max(y1, y2)
    for o in rv.OBSTACLES:
        (ox, oy) = o
        if ox >= xlow and ox <= xhigh and oy >= ylow and oy <= yhigh:
            if ox == x1 or x2 == x1:
                Simulate("%s cannot move in Euclidean path because of
                 ↪  obstacle\n" %r)
                return FAILURE
            elif abs((oy - y1)/(ox - x1) - (y2 - y1)/(x2 - x1)) <=
             ↪  0.0001:
                Simulate("%s cannot move in Euclidean path because of
                 ↪  obstacle\n" %r)
                return FAILURE

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

def moveCurved(r, l1, l2, dist):
    (x1, y1) = l1
    (x2, y2) = l2
    centrex = (x1 + x2)/2
    centrey = (y1 + y2)/2
    for o in rv.OBSTACLES:
        (ox, oy) = o
        r2 = (x2 - centrex)*(x2 - centrex) + (y2 - centrey)*(y2 -
         ↪  centrey)
        ro = (ox - centrex)*(ox - centrex) + (oy - centrey)*(oy -
         ↪  centrey)
        if abs(r2 - ro) <= 0.0001:
            Simulate("%s cannot move in curved path because of
             ↪  obstacle\n" %r)
            return FAILURE

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

def moveManhattan(r, l1, l2, dist):
    (x1, y1) = l1
    (x2, y2) = l2
    xlow = min(x1, x2)
    xhigh = max(x1, x2)
    ylow = min(y1, y2)
    yhigh = max(y1, y2)
    for o in rv.OBSTACLES:
        (ox, oy) = o
        if abs(oy - y1) <= 0.0001 and ox >= xlow and ox <= xhigh:
            Simulate("%s cannot move in Manhattan path because of
             ↪  obstacle\n" %r)
            return FAILURE

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

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

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

def inspectPerson(r, p):
    Simulate("Robot %s is inspecting person %s \n" %(r, p))
    state.status[p] = state.realStatus[p]
    return SUCCESS

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

def inspectLocation(r, l):
    Simulate("Robot %s is inspecting location %s \n" %(r, str(l)))
    state.status[l] = state.realStatus[l]
    return SUCCESS

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

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

    state.hasMedicine.ReleaseLock(r)
    return res

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

def captureImage(r, camera, l):
    img = Sense('captureImage', r, camera, l)

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

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

def SR_GETDISTANCE_Euclidean(l0, l1):
    (x0, y0) = l0
    (x1, y1) = l1
    return math.sqrt((x1 - x0)*(x1 - x0) + (y1 - y0)*(y1-y0))

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

def SR_GETDISTANCE_Manhattan(l0, l1):
    (x1, y1) = l0
    (x2, y2) = l1
    return abs(x2 - x1) + abs(y2 - y1)

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

def SR_GETDISTANCE_Curved(l0, l1):
    diameter = SR_GETDISTANCE_Euclidean(l0, l1)
    return math.pi * diameter / 2

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

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

def Rescue_Method1(r, p):
    if state.robotType[r] != 'uav':
        if state.hasMedicine[r] == 0:
            do_task('getSupplies', r)
        do_task('helpPerson', r, p)
    else:
        do_command(fail)

def Rescue_Method2(r, p):
    if state.robotType[r] == 'uav':
        do_task('getRobot')
    r2 = state.newRobot[1]
    if r2 != None:
        if state.hasMedicine[r2] == 0:
            do_task('getSupplies', r2)
        do_task('helpPerson', r2, p)
        state.status[r2] = 'free'
    else:
        Simulate("No robot is free to help person %s\n" %p)
        do_command(fail)

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

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

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

    else:
        do_command(fail)

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

def CheckReal(l):
    p = state.realPerson[l]
    if p != None:
        if state.realStatus[p] == 'injured' or state.realStatus[p] ==
         ↪  'dead' or state.realStatus[l] == 'hasDebri':
            Simulate("Person in location %s failed to be saved.\n"
             ↪  %str(l))
            do_command(deadEnd, p)
            do_command(fail)

def Survey_Method1(r, l):
    if state.robotType[r] != 'uav':
        do_command(fail)

    do_task('adjustAltitude', r)

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

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

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

    CheckReal(l)

def Survey_Method2(r, l):
    if state.robotType[r] != 'uav':
        do_command(fail)

    do_task('adjustAltitude', r)

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

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

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

    CheckReal(l)

def GetRobot_Method1():
    dist = float("inf")
    robot = None
    for r in rv.WHEELEDROBOTS:
        if state.status[r] == 'free':
            if SR_GETDISTANCE_Euclidean(state.loc[r], (1,1)) < dist:
                robot = r
                dist = SR_GETDISTANCE_Euclidean(state.loc[r], (1,1))
    if robot == None:
        do_command(fail)
    else:
        state.status[robot] = 'busy'
        state.newRobot[1] = robot   # Check if this can cause any
        ↪  regression with assignment statements

def GetRobot_Method2():
    state.newRobot[1] = rv.WHEELEDROBOTS[0]
    state.status[rv.WHEELEDROBOTS[0]] = 'busy'

def AdjustAltitude_Method1(r):
    if state.altitude[r] == 'high':
        do_command(changeAltitude, r, 'low')

def AdjustAltitude_Method2(r):
    if state.altitude[r] == 'low':
        do_command(changeAltitude, r, 'high')
```

B.5 Delivery domain

```
declare_commands([fail, wrap, pickup, acquireRobot, loadMachine,
↪  moveRobot, freeRobot, putdown, wait])

# Declare tasks
declare_task('orderStart', 'orderList')
declare_task('order', 'orderList')
declare_task('pickupAndLoad', 'orderName', 'o', 'm')
declare_task('unloadAndDeliver', 'm', 'package')
declare_task('moveToPallet', 'o', 'p')
declare_task('redoer', 'command')

declare_methods('orderStart', OrderStart_Method1)
declare_methods('order', Order_Method1, Order_Method2)
declare_methods('pickupAndLoad', PickupAndLoad_Method1)
declare_methods('unloadAndDeliver', UnloadAndDeliver_Method1)
declare_methods('moveToPallet', MoveToPallet_Method1)
declare_methods('redoer', Redoer)

def wait():
    if GLOBALS.GetPlanningMode() == True:
        return SUCCESS
    else:
        t1 = time()
        while(time() - t1 < 3):
            pass
        return SUCCESS

def Redoer(command, *args):
    i = 0
    while i < 3:
        if i > 0:
            Simulate("--Redoing command-- %s\n" % command)

        state.var1.AcquireLock('redoId')
        localRedoId = state.var1['redoId']
```

```
            state.var1['redoId'] += 1
            state.var1.ReleaseLock('redoId')

            state.shouldRedo[localRedoId] = False
            do_command(command, localRedoId, *args)

            if i > 0:
                Simulate("--Finished redo-- %s\n" % command)

            if not state.shouldRedo.pop(localRedoId):
                break
            i += 1

        if i >= 3:
            do_command(fail)

    # this is a function for narrowing down possibilities for obj lists
    # e.g. reduces combos for problem 4 task 1 from 10 to 4
    def MakeFocusedObjList():
        combos = (itertools.combinations([k for (k,v) in
         ↪  state.OBJECTS.items() if v == True],
         ↪  state.var1['inputLength']))
        focus = []

        orderList = state.var1['input']

        for objList in combos:
            works = True
            for i,objType in enumerate(orderList):
                # verify correct type
                if objList[i] not in state.OBJ_CLASS[objType]:
                    works = False
            if works:
                focus.append(objList)

        random.shuffle(focus)

        return focus

    # this is a dummy task so we can set the length
    # of the order
    def OrderStart_Method1(orderList):
        state.var1['inputLength'] = len(orderList)
        state.var1['input'] = orderList
        state.var1['focusObjList'] = MakeFocusedObjList
        do_task('order', orderList)

    def Order_Method1(orderList, m, objList):
        if len(orderList) != len(objList):
            Simulate("wrong length objList %s\n" % str(objList))
            do_command(fail)

        # make sure order is of correct type, reserve objects
        for i,objType in enumerate(orderList):
            # verify correct type
            if objList[i] not in state.OBJ_CLASS[objType]:
                Simulate("wrong type %s\n" % str(objList))
                do_command(fail)

            if state.OBJECTS[objList[i]] == False:
                Simulate("obj already used %s\n" % str(objList[i]))
                do_command(fail)

            Simulate("Reserving obj %s\n" % str(objList[i]))
            state.OBJECTS[objList[i]] = False

        # get unique ID for order name
        state.var1.AcquireLock('redoId')
        id = state.var1['redoId']
        state.var1['redoId'] += 1
        state.var1.ReleaseLock('redoId')

        Simulate("This is order ID: " + str(id) + "\n")

        for i, objType in enumerate(orderList):
            # move to object, pick it up, load in machine
            do_task('pickupAndLoad', frozenset([id] + [objList]),
             ↪  objList[i], m)

        do_task('redoer', wrap, frozenset([id] + [objList]), m, objList)
        package = state.var1['temp1']

        do_task('unloadAndDeliver', m, package)

    Order_Method1.parameters = "[(m, objList,) for m in rv.MACHINES for
     ↪  objList in state.var1['focusObjList']()]"

    def Order_Method2(orderList, m, objList, p):
        # wait if needed
        if len(orderList) != len(objList):
            Simulate("wrong length %s\n" % str(objList))
            do_command(fail)

        # make sure order is of correct type, reserve objects
        for i,objType in enumerate(orderList):
            # verify correct type
            if objList[i] not in state.OBJ_CLASS[objType]:
                Simulate("wrong type %s\n" % str(objList))
                do_command(fail)

            if state.OBJECTS[objList[i]] == False:
                Simulate("obj already used %s\n" % str(objList))
                do_command(fail)

            state.OBJECTS[objList[i]] = False

        # move objects to the pallet
        for i, objType in enumerate(orderList):
            # move to object, pick it up, place on pallet
            do_task('moveToPallet', objList[i], p)

        # get unique ID for order name
        state.var1.AcquireLock('redoId')
        id = state.var1['redoId']
        state.var1['redoId'] += 1
        state.var1.ReleaseLock('redoId')

        # move objects to machine
        for i,objType in enumerate(orderList):
            # move to object, pick it up, load in machine
            do_task('pickupAndLoad', frozenset([id] + [objList]),
             ↪  objList[i], m)

        do_task('redoer', wrap, frozenset([id] + [objList]), m, objList)
        package = state.var1['temp1']

        do_task('unloadAndDeliver', m, package)

    Order_Method2.parameters = "[(m, objList, p) for m in rv.MACHINES for
     ↪  objList in state.var1['focusObjList']()" \
                               "for p in rv.PALLETS]"

    # for free r
    def PickupAndLoad_Method1(orderName, o, m, r):
        # wait for robot if needed
        i = 0
        while state.busy[r] == True and i < 5:
            do_command(wait)
            i += 1

        if state.busy[r] == True:
            do_command(fail)

        # acquire robot
        do_task('redoer', acquireRobot, r)

        # move to object
        if state.loc[o] in rv.ROBOTS:
            do_task('redoer', putdown, state.loc[o], o)
        dist = OF_GETDISTANCE_GROUND(state.loc[r], state.loc[o])
        do_task('redoer', moveRobot, r, state.loc[r], state.loc[o], dist)

        # pick up object
        do_task('redoer', pickup, r, o)

        # move to machine
        dist = OF_GETDISTANCE_GROUND(state.loc[r], state.loc[m])
        do_task('redoer', moveRobot, r, state.loc[r], state.loc[m], dist)

        # wait if needed
        i = 0
        while state.busy[m] != False and state.busy[m] != orderName and i
         ↪  < 5:
            do_command(wait)
            i += 1

        if state.busy[m] != False and state.busy[m] != orderName:
            do_command(fail)

        # load machine
        do_task('redoer', loadMachine, orderName, r, m, o)

        do_task('redoer', freeRobot, r)
    PickupAndLoad_Method1.parameters = "[(r,) for r in rv.ROBOTS]"

    # unload a package from the machine, move package to shipping doc
    # for free r
    def UnloadAndDeliver_Method1(m, package, r):
        # wait for robot if needed
        i = 0
        while state.busy[r] == True and i < 5:
            do_command(wait)
            i += 1

        if state.busy[r] == True:
            do_command(fail)

        do_task('redoer', acquireRobot, r)

        dist = OF_GETDISTANCE_GROUND(state.loc[r], state.loc[m])
        do_task('redoer', moveRobot, r, state.loc[r], state.loc[m], dist)

        do_task('redoer', pickup, r, package)

        doc = rv.SHIPPING_DOC[rv.ROBOTS[r]]

        dist = OF_GETDISTANCE_GROUND(state.loc[r], doc)
        do_task('redoer', moveRobot, r, state.loc[r], doc, dist)

        do_task('redoer', putdown, r, package)

        do_task('redoer', freeRobot, r)

        Simulate("Package %s has been delivered\n" % package)
    UnloadAndDeliver_Method1.parameters = "[(r,) for r in rv.ROBOTS]"

    # for free r
    def MoveToPallet_Method1(o, p, r):
        # wait for robot if needed
        i = 0
        while state.busy[r] == True and i < 5:
            do_command(wait)
            i += 1

        if state.busy[r] == True:
            do_command(fail)

        do_task('redoer', acquireRobot, r)

        dist = OF_GETDISTANCE_GROUND(state.loc[r], state.loc[o])
        do_task('redoer', moveRobot, r, state.loc[r], state.loc[o], dist)

        do_task('redoer', pickup, r, o)

        dist = OF_GETDISTANCE_GROUND(state.loc[r], state.loc[p])
        do_task('redoer', moveRobot, r, state.loc[r], state.loc[p], dist)

        do_task('redoer', putdown, r, o)
        Simulate("Object %s was placed on pallet %s (goes with earlier
         ↪  msg)\n" % (o, p))

        do_task('redoer', freeRobot, r)
    MoveToPallet_Method1.parameters = "[(r,) for r in rv.ROBOTS]"

    def moveRobot(redoId, r, l1, l2, dist):
        state.loc.AcquireLock(r)
        state.shouldRedo.AcquireLock(redoId)

        if l1 == l2:
            Simulate("Robot %s is already at location %s\n" %(r, l2))
            res = SUCCESS
            state.shouldRedo[redoId] = False

        elif state.loc[r] == l1:
            start = globalTimer.GetTime()
            while (globalTimer.IsCommandExecutionOver('moveRobot', start,
             ↪  redoId, r, l1, l2, dist) == False):
                pass
            res = Sense('moveRobot')

            if res == SUCCESS:
                Simulate("Robot %s has moved from %s to %s\n" %(r, l1,
                 ↪  l2))
                state.loc[r] = l2
                state.shouldRedo[redoId] = False
            else:
                Simulate("Robot %s failed to move due to some internal
                 ↪  failure\n" %r)
                state.shouldRedo[redoId] = True
                res = SUCCESS
        else:
            Simulate("Robot %s is not in location %d\n" %(r, l1))
            res = FAILURE
            state.shouldRedo[redoId] = False

        state.shouldRedo.ReleaseLock(redoId)
        state.loc.ReleaseLock(r)

        return res
```

```
def pickup(redoId, r, item):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.loc.AcquireLock(item)
    state.shouldRedo.AcquireLock(redoId)

    if r not in rv.ROBOTS or item not in state.OBJECTS:
        Simulate("Passed argument %s or %s isn't a robot or object\n"
         ↪  % (r, item))
        res = FAILURE
        state.shouldRedo[redoId] = False
    elif state.load[r] != NIL:
        Simulate("Robot %s is already carrying an object\n" % r)
        res = FAILURE
        state.shouldRedo[redoId] = False
    elif state.loc[r] != state.loc[item]:
        Simulate("Robot %s and item %s are in different locations\n"
         ↪  % (r, item))
        res = FAILURE
        state.shouldRedo[redoId] = False
    elif state.OBJ_WEIGHT[item] > rv.ROBOT_CAPACITY[r]:
        start = globalTimer.GetTime()
        while (globalTimer.IsCommandExecutionOver('pickup', start,
         ↪  redoId, r, item) == False):
            pass

        Simulate("Item %s is too heavy for robot %s to pick up\n" %
         ↪  (item, r))
        res = FAILURE
        state.shouldRedo[redoId] = False
    else:
        start = globalTimer.GetTime()
        while (globalTimer.IsCommandExecutionOver('pickup', start,
         ↪  redoId, r, item) == False):
            pass
        res = Sense('pickup')

        if res == SUCCESS:
            Simulate("Robot %s picked up %s\n" % (r, item))
            state.loc[item] = r
            state.load[r] = item
            state.shouldRedo[redoId] = False
        else:
            Simulate("Robot %s dropped item %s\n" % (r, item))
            res = SUCCESS
            state.shouldRedo[redoId] = True

    state.shouldRedo.ReleaseLock(redoId)
    state.loc.ReleaseLock(item)
    state.loc.ReleaseLock(r)
    state.load.ReleaseLock(r)

    return res

def putdown(redoId, r, item):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.loc.AcquireLock(item)
    state.shouldRedo.AcquireLock(redoId)

    if state.load[r] != item:
        Simulate("Robot %s is not carrying the object %s\n" % (r,
        ↪  item))
        res = FAILURE
        state.shouldRedo[redoId] = False
    else:
        start = globalTimer.GetTime()
        while (globalTimer.IsCommandExecutionOver('putdown', start,
        ↪  redoId, r, item) == False):
            pass
        res = Sense('putdown')

        if res == SUCCESS:
            Simulate("Robot %s put down %s at loc %s\n" % (r, item,
            ↪  state.loc[r]))
            state.loc[item] = state.loc[r]
            state.load[r] = NIL
            state.shouldRedo[redoId] = False
        else:
            Simulate("Robot %s failed to put down %s\n" % (r, item))
            res = SUCCESS
            state.shouldRedo[redoId] = True

    state.shouldRedo.ReleaseLock(redoId)
    state.loc.ReleaseLock(item)
    state.loc.ReleaseLock(r)
    state.load.ReleaseLock(r)

    return res
```

```
def loadMachine(redoId, orderName, r, m, item):
    state.load.AcquireLock(r)
    state.loc.AcquireLock(r)
    state.loc.AcquireLock(m)
    state.loc.AcquireLock(item)
    state.busy.AcquireLock(m)
    state.shouldRedo.AcquireLock(redoId)

    if state.loc[r] != state.loc[m]:
        Simulate("Robot %s isn't at machine %s" % (r, m))
        res = FAILURE
        state.shouldRedo[redoId] = False
    elif state.busy[m] != orderName and state.busy[m] != False:
        Simulate("Robot %s can't load machine %s, it is working on a
         ↪  different order\n" %(r, m,))
        res = FAILURE
        state.shouldRedo[redoId] = False
    else:
        start = globalTimer.GetTime()
        while (globalTimer.IsCommandExecutionOver('loadMachine',
         ↪  start, redoId, orderName, r, m, item) == False):
            pass

        res = Sense('loadMachine')

        if res == SUCCESS:
            Simulate("Robot %s loaded machine %s with item %s\n" %
             ↪  (r, m, item))
            state.load[r] = NIL
            state.loc[item] = m

            state.busy[m] = orderName
            state.shouldRedo[redoId] = False
        else:
            Simulate("Robot %s failed to load machine %s\n" % (r, m))
            res = SUCCESS
            state.shouldRedo[redoId] = True

    state.shouldRedo.ReleaseLock(redoId)
    state.busy.ReleaseLock(m)
    state.loc.ReleaseLock(item)
    state.loc.ReleaseLock(m)
    state.loc.ReleaseLock(r)
    state.load.ReleaseLock(r)
    return res

# to be not busy anymore, or drop object
def acquireRobot(redoId, r):
    state.busy.AcquireLock(r)
    state.load.AcquireLock(r)
    state.shouldRedo.AcquireLock(redoId)

    state.shouldRedo[redoId] = False

    if state.busy[r] == True:
        Simulate("Robot %s is busy\n" % r)
        res = FAILURE
    elif state.load[r] != NIL:
        Simulate("Robot %s is carrying an object\n" % r)
        res = FAILURE
    else:
        Simulate("Robot %s is acquired for a new task\n" % r)
        state.busy[r] = True
        res = SUCCESS

    state.shouldRedo.ReleaseLock(redoId)
    state.load.ReleaseLock(r)
    state.busy.ReleaseLock(r)

    return res

def freeRobot(redoId, r):
    state.busy.AcquireLock(r)
    state.load.AcquireLock(r)
    state.shouldRedo.AcquireLock(redoId)

    state.shouldRedo[redoId] = False

    if state.load[r] != NIL:
        Simulate("Robot %s is carrying an object\n" % r)
        res = FAILURE
    else:
        Simulate("Robot %s is now free\n" % r)
        state.busy[r] = False
        res = SUCCESS

    state.shouldRedo.ReleaseLock(redoId)
    state.load.ReleaseLock(r)
    state.busy.ReleaseLock(r)
    return res

def wrap(redoId, orderName, m, objList):
    state.loc.AcquireLock(m)
    state.busy.AcquireLock(m)
    state.numUses.AcquireLock(m)
    state.shouldRedo.AcquireLock(redoId)

    weight = 0

    for obj in objList:
        weight += state.OBJ_WEIGHT[obj]
        if state.loc[obj] != m:
            Simulate("Machine %s not loaded with item %s\n" % (m,
             ↪  obj))
            res = FAILURE
            state.shouldRedo[redoId] = False

            state.shouldRedo.ReleaseLock(redoId)
            state.numUses.ReleaseLock(m)
            state.busy.ReleaseLock(m)
            state.loc.ReleaseLock(m)

            return res

    if state.busy[m] != orderName:
        Simulate("Machine %s is busy with a diffrent order\n" % m)
        res = FAILURE
        state.shouldRedo[redoId] = False
    else:
        start = globalTimer.GetTime()
        while globalTimer.IsCommandExecutionOver('wrap', start,
         ↪  redoId, orderName, m, objList) == False:
            pass

        state.numUses[m] += 1
        res = SenseWrap(state.numUses[m])

        if res == SUCCESS:
            packageName = "Pack#" + str(orderName)

            state.OBJECTS[packageName] = True
            state.loc[packageName] = state.loc[m]
            state.OBJ_WEIGHT[packageName] = weight
            state.var1.AcquireLock('temp1')
            state.var1['temp1'] = packageName
            state.var1.ReleaseLock('temp1')

            Simulate("Machine %s wrapped package %s\n" % (m,
             ↪  packageName))
            state.busy[m] = False
            state.shouldRedo[redoId] = False

        else:
            Simulate("Machine %s jammed. Failed to wrap\n" % m)
            state.shouldRedo[redoId] = True
            # set res to success for return
            res = SUCCESS

    state.shouldRedo.ReleaseLock(redoId)
    state.numUses.ReleaseLock(m)
    state.busy.ReleaseLock(m)
    state.loc.ReleaseLock(m)

    return res
```

B.6 Attack recovery in Software-defined networks (SDN)

```
declare_commands([
    restart_vm,
    add_vcpu,
    increase_mem,
    kill_top_proc,
    apply_update,
    add_switch,
    move_critical_hosts,
    clear_ctrl_state_besteffort,
    clear_ctrl_state_fallback,
    reinstall_ctrl_besteffort,
    reinstall_ctrl_fallback,
    clear_switch_state_besteffort,
    clear_switch_state_fallback,
    disconnect_reconnect_switch_port,
    disconnect_switch_port,
    succeed,
    unsure,
    fail
])

declare_task('fix_sdn', 'config')
declare_task('handle_event', 'event', 'config')
declare_task('fix_component', 'component_id', 'config')
declare_task('fix_low_resources', 'component_id', 'config')
declare_task('try_generic_fix', 'component_id', 'config')
declare_task('fix_sdn_controller', 'component_id', 'config')
declare_task('fix_switch', 'component_id', 'config')
declare_task('shrink_ctrl_hosttable', 'component_id')
declare_task('alleviate_ctrl_cpu', 'component_id')
declare_task('restore_ctrl_health', 'component_id')
declare_task('shrink_switch_flowtable', 'component_id')
declare_task('alleviate_switch_cpu', 'component_id')
declare_task('restore_switch_health', 'component_id')

declare_methods(
    'fix_sdn',
    m_fix_sdn
)

declare_methods(
    'handle_event',
    m_handle_event
)

declare_methods(
    'fix_component',
    m_fix_vm,
    m_fix_software
)
declare_methods(
    'fix_low_resources',
    m_add_vcpu,
    m_increase_mem
)
declare_methods(
    'try_generic_fix',
    m_software_update,
    m_software_reinstall
)
declare_methods(
    'fix_sdn_controller',
    m_ctrl_clearstate_besteffort,
    m_ctrl_mitigate_pktinflood,
    m_fix_sdn_controller_fallback
)
declare_methods(
    'fix_switch',
    m_fix_switch
)

declare_methods(
    'shrink_ctrl_hosttable',
    m_ctrl_clearstate_besteffort,
    m_ctrl_clearstate_fallback,
    m_ctrl_reinstall_besteffort,
    m_ctrl_reinstall_fallback
)
declare_methods(
    'alleviate_ctrl_cpu',
    m_component_kill_top_proc,
    m_ctrl_reinstall_besteffort,
    m_ctrl_reinstall_fallback,
    m_component_restartvm
)
declare_methods(
    'restore_ctrl_health',
    m_ctrl_reinstall_besteffort,
    m_ctrl_reinstall_fallback,
    m_component_restartvm
)

declare_methods(
    'shrink_switch_flowtable',
    m_switch_clearstate_besteffort,
    m_switch_clearstate_fallback
)
declare_methods(
    'alleviate_switch_cpu',
    m_switch_discon_recon_txport,
    m_component_kill_top_proc,
    m_switch_clearstate_besteffort,
    m_switch_clearstate_fallback,
    m_switch_disconnect_txport
)
declare_methods(
    'restore_switch_health',
    m_switch_discon_recon_txport,
    m_switch_clearstate_besteffort,
    m_switch_clearstate_fallback,
    m_switch_disconnect_txport
)

def is_component_type(component_id, comp_type):
    """Check whether given component is of the given type (e.g.,
    ↪   ``CTRL`` or ``SWITCH``)."""

    # Component types should be defined in state
    if not hasattr(state, 'components'):
        return False
    if component_id not in state.components:
        return False
    if 'type' not in state.components[component_id]:
        return False

    # Check whether component type matches
    if state.components[component_id]['type'] == comp_type:
        return True
    else:
        return False

def is_component_critical(component_id):
    """Check whether given component is critical."""

    # Component types should be defined in state
    if not hasattr(state, 'components'):
        return False
    if component_id not in state.components:
        return False
    if 'critical' not in state.components[component_id]:
        return False

    # Check whether component type matches
    return (state.components[component_id]['critical'] is True)

def get_component_stat(component_id, stat_key):
    """Returns the :class:`dict` for the given statistic, or
    ↪   ``None`` if it doesn't exist."""

    if hasattr(state, 'stats') and component_id in state.stats:
        if stat_key in state.stats[component_id]:
            return state.stats[component_id][stat_key]
    return None
```

```
def is_component_healthy(component_id):
    """Check whether the given component's health value is above
    ↪  the healthy threshold."""

    health_stat = get_component_stat(component_id, 'health')
    if health_stat is not None:
        health_val = health_stat['value']
        health_thresh_fn = health_stat['thresh_exceeded_fn']
        if not health_thresh_fn(health_val):
            return True
    return False

# Commands

def restart_vm(component_id):
    """Restart a component virtual machine."""

    # Sense success vs. failure
    res = Sense('restart_vm')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "restart_vm"')
        return FAILURE

    # Restarting takes some time, but can fix some problems, so
    ↪  increase health
    health_stat = get_component_stat(component_id, 'health')
    if health_stat is not None:
        cur_health = health_stat['value']
        new_health = min(1.0, (cur_health + 0.1) * 2)
        health_stat['value'] = new_health

    # CPU utilization should reset after restarting
    cpu_stat = get_component_stat(component_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cpu_stat['value'] = 0.0

    # Memory utilization should reset after restarting
    mem_stat = get_component_stat(component_id, 'mem_perc_ewma')
    if mem_stat is not None:
        mem_stat['value'] = 0.0

    # Host table size should reset after restarting
    hosttable_stat = get_component_stat(component_id,
    ↪  'host_table_size')
    if hosttable_stat is not None:
        hosttable_stat['value'] = 0

    # Flow table size should reset after restarting
    flowtable_stat = get_component_stat(component_id,
    ↪   'flow_table_size')
    if flowtable_stat is not None:
        flowtable_stat['value'] = 0

    # Done
    return SUCCESS

def add_vcpu(component_id):
    """Add VCPU to component virtual machine, thus increasing
    ↪   component's VCPU count by one."""

    # Sense success vs. failure
    res = Sense('add_vcpu')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "add_vcpu"')
        return FAILURE

    # CPU utilization should decrease
    cpu_stat = get_component_stat(component_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cpu_val = cpu_stat['value']
        cpu_stat['value'] = cpu_val / 2.0

    # Health should increase after increasing CPU
    health_stat = get_component_stat(component_id, 'health')
    if health_stat is not None:
        cur_health = health_stat['value']
        new_health = min(1.0, cur_health + 0.1)
        health_stat['value'] = new_health
    return SUCCESS

def increase_mem(component_id):
    """Increase memory of component virtual machine."""

    # Sense success vs. failure
    res = Sense('increase_mem')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "increase_mem"')
        return FAILURE

    # Memory utilization should decrease
    mem_stat = get_component_stat(component_id, 'mem_perc_ewma')
    if mem_stat is not None:
        mem_val = mem_stat['value']
        mem_stat['value'] = mem_val / 2.0

    # Health should increase after increasing memory
    health_stat = get_component_stat(component_id, 'health')
    if health_stat is not None:
        cur_health = health_stat['value']
        new_health = min(1.0, cur_health + 0.1)
        health_stat['value'] = new_health
    return SUCCESS

def kill_top_proc(component_id):
    """Kill top CPU-consuming process in a component virtual
    ↪   machine."""

    # Sense success vs. failure
    res = Sense('kill_top_proc')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "kill_top_proc"')
        return FAILURE

    # CPU utilization should decrease if CPU-hungry process is
    ↪   stopped
    cpu_stat = get_component_stat(component_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cur_cpu = cpu_stat['value']
        new_cpu = max(0.0, (cur_cpu - 50.0) / 2)
        cpu_stat['value'] = new_cpu

    # Health should increase after CPU-hungry process is stopped
    health_stat = get_component_stat(component_id, 'health')
    if health_stat is not None:
        cur_health = health_stat['value']
        new_health = min(1.0, (cur_health + 0.1) * 2)
        health_stat['value'] = new_health
    return SUCCESS

def apply_update(component_id, software):
    """Apply updates to the given software package in the component
    ↪   virtual machine."""

    # Sense success vs. failure
    res = Sense('apply_update')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "apply_update"')
        return FAILURE
    return SUCCESS

def add_switch(component_id):
    """Add a new switch to the SDN, copying connectivity/links of
    ↪  the given switch."""

    # Sense success vs. failure
    res = Sense('add_switch')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "add_switch"')
        return FAILURE

    new_id = component_id + '-new'
    state.components[new_id] = state.components[component_id]
    state.components[new_id]['id'] = new_id

    state.stats[new_id] = state.stats[component_id]

    # Reset health
    health_stat = get_component_stat(new_id, 'health')
    if health_stat is not None:
        health_stat['value'] = 1.0

    # Reset CPU utilization
    cpu_stat = get_component_stat(new_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cpu_stat['value'] = 0.0

    # Reset memory utilization
    mem_stat = get_component_stat(new_id, 'mem_perc_ewma')
    if mem_stat is not None:
        mem_stat['value'] = 0.0

    # Reset flow table size
    flowtable_stat = get_component_stat(new_id, 'flow_table_size')
    if flowtable_stat is not None:
        flowtable_stat['value'] = 0

    return SUCCESS

def move_critical_hosts(old_switch_id, new_switch_id):
    """Move critical hosts from one switch to another."""
    # Sense success vs. failure
    res = Sense('move_critical_hosts')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "move_critical_hosts"')
        return FAILURE
    return SUCCESS

def clear_ctrl_state_besteffort(component_id):
    """Clear the SDN controller state (including host table), if
     ↪  possible."""

    # Sense success vs. failure
    res = Sense('clear_ctrl_state_besteffort')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "clear_ctrl_state_besteffort"')
        return FAILURE

    stat = get_component_stat(component_id, 'host_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def clear_ctrl_state_fallback(component_id):
    """Clear the SDN controller state (including host table) in a
     ↪  more robust way."""

    # Sense success vs. failure
    res = Sense('clear_ctrl_state_fallback')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "clear_ctrl_state_fallback"')
        return FAILURE

    stat = get_component_stat(component_id, 'host_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def reinstall_ctrl_besteffort(component_id):
    """Reinstall the SDN controller software, if possible."""

    # Sense success vs. failure
    res = Sense('reinstall_ctrl_besteffort')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "reinstall_ctrl_besteffort"')
        return FAILURE

    stat = get_component_stat(component_id, 'host_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def reinstall_ctrl_fallback(component_id):
    """Reinstall the SDN controller software in a more robust
     ↪  way."""

    # Sense success vs. failure
    res = Sense('reinstall_ctrl_fallback')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "reinstall_ctrl_fallback"')
        return FAILURE

    stat = get_component_stat(component_id, 'host_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def clear_switch_state_besteffort(component_id):
    """Clear the switch state (including flow table), if
     ↪  possible."""

    # Sense success vs. failure
    res = Sense('clear_switch_state_besteffort')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "clear_switch_state_besteffort"')
        return FAILURE

    stat = get_component_stat(component_id, 'flow_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def clear_switch_state_fallback(component_id):
    """Clear the switch state (including flow table) in a more
     ↪  robust way."""
    # Sense success vs. failure
    res = Sense('clear_switch_state_fallback')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "clear_switch_state_fallback"')
        return FAILURE

    stat = get_component_stat(component_id, 'flow_table_size')
    if stat is not None:
        stat['value'] = 0
    return SUCCESS

def disconnect_reconnect_switch_port(component_id):
    """Disconnect and then reconnect switch port with most
     ↪  transmitted traffic."""

    # Sense success vs. failure
    res = Sense('disconnect_reconnect_switch_port')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "disconnect_reconnect_switch_port"')
        return FAILURE

    cpu_stat = get_component_stat(component_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cur_cpu = cpu_stat['value']
        new_cpu = max(0.0, (cur_cpu - 50.0) / 2)
        cpu_stat['value'] = new_cpu
    return SUCCESS

def disconnect_switch_port(component_id):
    """Disconnect switch port with most transmitted traffic."""

    # Sense success vs. failure
    res = Sense('disconnect_switch_port')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for
         ↪  "disconnect_switch_port"')
        return FAILURE

    cpu_stat = get_component_stat(component_id, 'cpu_perc_ewma')
    if cpu_stat is not None:
        cur_cpu = cpu_stat['value']
        new_cpu = max(0.0, (cur_cpu - 50.0) / 2)
        cpu_stat['value'] = new_cpu
    return SUCCESS

def unsure():
    """Add some cost within refinement method."""

    # Sense success vs. failure
    res = Sense('unsure')
    if res == FAILURE:
        log_err('Sense() returned FAILURE for "unsure"')
        return FAILURE
    return SUCCESS

# Methods
def m_fix_sdn(config):
    """Method to fix all symptoms in the SDN by checking each
    ↪  component.

    Checks the health of each component. For any component with
↪   health below the critical threshold,
    delegates to ``fix_component``.
    """

    if not isinstance(config, dict) or 'health_critical_thresh' not
    ↪  in config:
        log_err('could not find "health_critical_thresh" in config')
        do_command(fail)
    else:
        log_info('will check health for ' +
        ↪  str(len(state.components.keys())) + ' components')
        for component_id in state.components:
            if component_id not in state.stats or 'health' not in
            ↪  state.stats[component_id]:
                log_err('could not find "health" in state.stats["' +
                ↪  component_id + '"]')
                do_command(fail)
            else:
                health_obj = state.stats[component_id]['health']
                if 'value' not in health_obj or 'thresh_exceeded_fn'
                ↪  not in health_obj:
                    log_err('could not find "value" or
                    ↪  "thresh_exceeded_fn" in state.stats["'
                            + component_id + '"]["health"]')
                    do_command(fail)
                else:
                    # Check for low health
                    value = health_obj['value']
                    thresh_exceeded_fn =
                     ↪  health_obj['thresh_exceeded_fn']
                    if thresh_exceeded_fn(value):
                        log_info('threshold exceeded for stat
                         ↪  "health": ' + component_id)
                        log_info('adding new task "fix_component" for
                         ↪  "' + component_id + '"')
                        do_task('fix_component', component_id,
                         ↪  config)

                        # Check new health
                        if not is_component_healthy(component_id):
                            log_err('failed to restore component
                             ↪  health: ' + component_id)
                            do_task(fail)


def m_handle_event(event, config):
    """Method to handle a specific anomaly/event."""

    # Handle event types based on their source
    if 'source' not in event:
        log_err('could not find "source" in event')
        do_command(fail)
    else:
        if event['source'] == 'sysmon':
            # SysMon detects events related to high resource
             ↪  consumption
            low_resource_components = []

            # Add component that triggered this event
            component_id = event['component_id']
            low_resource_components.append(component_id)

            # Address symptoms of each affected component
            for component in low_resource_components:
                log_info('adding new task "fix_component" for "' +
                 ↪  component + '"')
                do_task('fix_component', component, config)

            # Check whether affected component is now healthy
            if not is_component_healthy(component_id):
                log_err('failed to restore component health: ' +
                 ↪  component_id)
                do_task(fail)

        # Unhandled event source
        else:
            log_err('unhandled event source "' + event['source'] +
             ↪  '"')
            do_command(fail)

def m_fix_vm(component_id, config):
    """Method to fix symptoms at the virtual machine level."""

    do_fix_low_resources = False
    for stat_key in ['cpu_perc_ewma', 'mem_perc_ewma']:
        stat_obj = get_component_stat(component_id, stat_key)
        if stat_obj is not None:
            cur_val = stat_obj['value']
            thresh_exceeded_fn = stat_obj['thresh_exceeded_fn']
            if thresh_exceeded_fn(cur_val):
                do_fix_low_resources = True
                break
    if do_fix_low_resources is True:
        do_task('fix_low_resources', component_id, config)
    else:
        do_command(restart_vm, component_id)

def m_fix_software(component_id, config):
    """Method to fix symptoms at the software/process level."""

    do_fix_generic = False
    do_fix_sdnctrl = False
    do_fix_switch = False
    if is_component_type(component_id, 'CTRL'):
        do_fix_sdnctrl = True
    elif is_component_type(component_id, 'SWITCH'):
        do_fix_switch = True
    else:
        do_fix_generic = True

    if do_fix_generic is True:
        do_task('try_generic_fix', component_id, config)
    elif do_fix_sdnctrl is True:
        do_task('fix_sdn_controller', component_id, config)
    elif do_fix_switch is True:
        do_task('fix_switch', component_id, config)
    else:
        do_command(fail)

def m_software_update(component_id, config):
    """Method to apply updates to a software package."""

    do_command(apply_update, component_id)

def m_software_reinstall(component_id, config):
    """Method to reinstall a software package."""

    if is_component_type(component_id, 'CTRL'):
        do_command(reinstall_ctrl_besteffort, component_id)
    else:
        do_command(fail)

def m_ctrl_mitigate_pktinflood(component_id, config):
    """Method to mitigate an SDN PACKET_IN flooding attack on a
    ↪  controller."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
        ↪  controller')
        do_command(fail)

    # Detect which switches are the source of attack
    for switch_id in state.components:
        if is_component_type(switch_id, 'SWITCH') and not
        ↪  is_component_healthy(switch_id):

            # Move critical hosts away from unhealthy switch
            if is_component_critical(switch_id):

                # Add new switch
                do_command(add_switch, switch_id)

                # Move critical hosts from unhealthy switches
                do_command(move_critical_hosts, switch_id, switch_id
                ↪  + '-new')

            # Fix unhealthy switch
            do_task('fix_switch', switch_id)

    # Clear controller state
    do_command(clear_ctrl_state_besteffort, component_id)

    # Check whether controller is now healthy
    if not is_component_healthy(component_id):
        log_err('failed to restore component health: ' +
         ↪  component_id)
        do_task(fail)

def m_fix_sdn_controller_fallback(component_id, config):
    """Method to fix symptoms for a controller."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
         ↪  controller')
        do_command(fail)
    elif component_id not in state.stats:
        log_err('could not find "' + component_id + '" in
         ↪  state.stats')
        do_command(fail)
    else:
        # Check stats and determine what needs to be fixed
        stat_obj = state.stats[component_id]
        do_shrink_hosttable = False
        do_alleviate_cpu = False
        do_restore_health = False
        if 'host_table_size' in stat_obj:
            if ('value' not in stat_obj['host_table_size']
                    or 'thresh_exceeded_fn' not in
                     ↪  stat_obj['host_table_size']):
                log_err('could not find "value" or
                 ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["host_table_size"]')
                do_command(fail)
            else:
                # Check for inflated host table
                value = stat_obj['host_table_size']['value']
                thresh_exceeded_fn =
                 ↪  stat_obj['host_table_size']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat
                     ↪  "host_table_size"')
                    do_shrink_hosttable = True
        if 'cpu_perc_ewma' in stat_obj:
            if ('value' not in stat_obj['cpu_perc_ewma']
                    or 'thresh_exceeded_fn' not in
                     ↪  stat_obj['cpu_perc_ewma']):
                log_err('could not find "value" or
                 ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["cpu_perc_ewma"]')
                do_command(fail)
            else:
                # Check for elevated CPU stat
                value = stat_obj['cpu_perc_ewma']['value']
                thresh_exceeded_fn =
                 ↪  stat_obj['cpu_perc_ewma']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat
                     ↪  "cpu_perc_ewma": ' + component_id)
                    do_alleviate_cpu = True
        if 'health' in stat_obj:
            if ('value' not in stat_obj['health']
                    or 'thresh_exceeded_fn' not in
                     ↪  stat_obj['health']):
                log_err('could not find "value" or
                 ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["health"]')
                do_command(fail)
            else:
                # Check for low health
                value = stat_obj['health']['value']
                thresh_exceeded_fn =
                 ↪  stat_obj['health']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat "health": '
                     ↪  + component_id)
                    do_restore_health = True

        if do_shrink_hosttable:
            # Fix problem with inflated host table
            log_info('adding new task "shrink_ctrl_hosttable" for "'
             ↪  + component_id + '"')
            do_task('shrink_ctrl_hosttable', component_id)
        elif do_alleviate_cpu:
            # Alleviate elevated CPU stat
            log_info('adding new task "alleviate_ctrl_cpu" for "' +
             ↪  component_id + '"')
            do_task('alleviate_ctrl_cpu', component_id)
        elif do_restore_health:
            # Restore low health (often also fixes CPU
            ↪  over-utilization)
            log_info('adding new task "restore_ctrl_health" for "' +
            ↪  component_id + '"')
            do_task('restore_ctrl_health', component_id)
        else:
            # No problem could be identified from stats
            log_info('no task to add for "' + component_id + '"')
            # For now, fail
            # log_err('could not figure out how to fix controller
            ↪  "' + component_id + '"')
            # do_command(fail)

def m_fix_switch(component_id, config):
    """Method to fix symptoms for a switch."""

    if not is_component_type(component_id, 'SWITCH'):
        log_err('component "' + component_id + '" is not a switch')
        do_command(fail)
    elif component_id not in state.stats:
        log_err('could not find "' + component_id + '" in
        ↪  state.stats')
        do_command(fail)
    else:
        # Check stats and determine what needs to be fixed
        stat_obj = state.stats[component_id]
        do_shrink_flowtable = False
        do_alleviate_cpu = False
        do_restore_health = False
        if 'flow_table_size' in stat_obj:
            if ('value' not in stat_obj['flow_table_size']
                    or 'thresh_exceeded_fn' not in
                    ↪  stat_obj['flow_table_size']):
                log_err('could not find "value" or
                ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["flow_table_size"]')
                do_command(fail)
            else:
                # Check for inflated flow table
                value = stat_obj['flow_table_size']['value']
                thresh_exceeded_fn =
                ↪  stat_obj['flow_table_size']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat
                    ↪  "flow_table_size"')
                    do_shrink_flowtable = True
        if 'cpu_perc_ewma' in stat_obj:
            if ('value' not in stat_obj['cpu_perc_ewma']
                    or 'thresh_exceeded_fn' not in
                    ↪  stat_obj['cpu_perc_ewma']):
                log_err('could not find "value" or
                ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["cpu_perc_ewma"]')
                do_command(fail)
            else:
                # Check for elevated CPU stat
                value = stat_obj['cpu_perc_ewma']['value']
                thresh_exceeded_fn =
                ↪  stat_obj['cpu_perc_ewma']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat
                    ↪  "cpu_perc_ewma": ' + component_id)
                    do_alleviate_cpu = True
        if 'health' in stat_obj:
            if ('value' not in stat_obj['health']
                    or 'thresh_exceeded_fn' not in
                    ↪  stat_obj['health']):
                log_err('could not find "value" or
                ↪  "thresh_exceeded_fn" in state.stats["'
                        + component_id + '"]["health"]')
                do_command(fail)
            else:
                # Check for low health
                value = stat_obj['health']['value']
                thresh_exceeded_fn =
                ↪  stat_obj['health']['thresh_exceeded_fn']
                if thresh_exceeded_fn(value):
                    log_info('threshold exceeded for stat "health"')
                    do_restore_health = True
        if do_shrink_flowtable:
            # Fix problem with inflated flow table
            log_info('adding new task "shrink_switch_flowtable" for
            ↪  "' + component_id + '"')
            do_task('shrink_switch_flowtable', component_id)
        elif do_alleviate_cpu:
            # Alleviate elevated CPU stat
            log_info('adding new task "alleviate_switch_cpu" for "' +
            ↪  component_id + '"')
            do_task('alleviate_switch_cpu', component_id)
        elif do_restore_health:
```

```
            # Restore low health (often also fixes CPU
            ↪  over-utilization)
            log_info('adding new task "restore_switch_health" for "'
            ↪  + component_id + '"')
            do_task('restore_switch_health', component_id)
        else:
            # No problem could be identified from stats
            log_err('could not figure out how to fix switch "' +
            ↪  component_id + '"')
            do_command(fail)

def m_add_vcpu(component_id):
    """Method to add a VCPU to a component virtual machine."""

    do_command(add_vcpu, component_id)

def m_increase_mem(component_id):
    """Method to increase memory in a component virtual machine."""

    do_command(increase_mem, component_id)

def m_ctrl_clearstate_besteffort(component_id):
    """Method to clear controller state (best effort)."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
        ↪  controller')
        do_command(fail)
    else:
        do_command(clear_ctrl_state_besteffort, component_id)

def m_ctrl_clearstate_fallback(component_id):
    """Method to clear controller state (fallback)."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
        ↪  controller')
        do_command(fail)
    else:
        do_command(clear_ctrl_state_fallback, component_id)

def m_ctrl_reinstall_besteffort(component_id):
    """Method to reinstall controller software (best effort)."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
         ↪  controller')
        do_command(fail)
    else:
        do_command(reinstall_ctrl_besteffort, component_id)

def m_ctrl_reinstall_fallback(component_id):
    """Method to reinstall controller software (fallback)."""

    if not is_component_type(component_id, 'CTRL'):
        log_err('component "' + component_id + '" is not a
         ↪  controller')
        do_command(fail)
    else:
        do_command(reinstall_ctrl_fallback, component_id)

def m_component_restartvm(component_id):
    """Method to restart the virtual machine of a component."""

    do_command(restart_vm, component_id)

def m_component_kill_top_proc(component_id):
    """Method to kill the top CPU-consuming process in a component
     ↪  virtual machine."""

    do_command(kill_top_proc, component_id)

def m_switch_clearstate_besteffort(component_id):
    """Method to clear switch state (best effort)."""

    if not is_component_type(component_id, 'SWITCH'):
        log_err('component "' + component_id + '" is not a switch')
        do_command(fail)
    else:
        do_command(clear_switch_state_besteffort, component_id)

def m_switch_clearstate_fallback(component_id):
    """Method to clear switch state (fallback)."""

    if not is_component_type(component_id, 'SWITCH'):
        log_err('component "' + component_id + '" is not a switch')
        do_command(fail)
    else:
        do_command(clear_switch_state_fallback, component_id)
```

```
def m_switch_discon_recon_txport(component_id):
    """Method to disconnect and then reconnect switch port with
    ↪  most transmitted traffic."""

    if not is_component_type(component_id, 'SWITCH'):
        log_err('component "' + component_id + '" is not a switch')
        do_command(fail)
    else:
        do_command(disconnect_reconnect_switch_port, component_id)

def m_switch_disconnect_txport(component_id):
    """Method to disconnect switch port with most transmitted
    ↪  traffic."""

    if not is_component_type(component_id, 'SWITCH'):
        log_err('component "' + component_id + '" is not a switch')
        do_command(fail)
    else:
        do_command(disconnect_switch_port, component_id)
```

Bibliography

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

[2] Félix Ingrand and Malik Ghallab. Deliberation for Autonomous Robots: A Survey. *Artificial Intelligence*, 247:10–44, 2017.

[3] Mikael Henaff, Alfredo Canziani, and Yann LeCun. Model-predictive policy learning with uncertainty regularization for driving in dense traffic. *arXiv preprint arXiv:1901.02705*, 2019.

[4] Manuela M Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *IJCAI*, page 4423, 2015.

[5] Neesha Ramchandani. Virtual coaching to enhace diabetes care. *Diabetes Technology and Therapeutics*, 21(2):S2–48–S2–51, 2019.

[6] Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. An Operational Semantics for a Fragment of PRS. In *IJCAI*, 2018.

[7] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[8] Malik Ghallab, Dana S Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[9] Félix Ingrand, Raja Chatilla, Rachid Alami, and Frederic Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, pages 43–49, 1996.

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

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

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

[13] Reid Simmons and David Apfelbaum. A task description language for robot control. In *IROS*, pages 1931–1937, 1998.

[14] Michael Beetz and Drew McDermott. Improving robot plans during their execution. In *AIPS*, 1994.

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

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

[17] Vandi Verma, Tara Estlin, Ari K Jónsson, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *i-SAIRAS*, 2005.

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

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

[20] David J Musliner, Michael JS Pelican, Robert P Goldman, Kurt D Krebsbach, and Edmund H Durfee. The evolution of circa, a theory-based ai architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, volume 1205, 2008.

[21] Robert P Goldman, Daniel Bryce, Michael JS Pelican, David J Musliner, and Kyungmin Bae. A hybrid architecture for correct-by-construction hybrid planning and control. In *NASA Formal Methods Symposium*, pages 388–394. Springer, 2016.

[22] Robert P Goldman. A semantics for htn methods. In *ICAPS*, 2009.

[23] Michel D Ingham, Robert J Ragno, and Brian C Williams. A reactive model-based programming language for robotic space explorers. In *i-SAIRAS*, 2001.

[24] Brian C Williams and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*, 2001.

[25] P.R. Conrad, J.A. Shah, and Brian C Williams. Flexible execution of plans with choice. In *ICAPS*, 2009.

[26] Robert Effinger, Brian Williams, and Andreas Hofmann. Dynamic execution of temporally and spatially flexible reactive programs. In *AAAI Wksp. on Bridging the Gap between Task and Motion Planning*, pages 1–8, 2010.

[27] Pedro Henrique Rodrigues Quemel Assis Santana and Brian Charles Williams. Chance-constrained consistency for probabilistic temporal plan networks. In *ICAPS*, November 2014.

[28] Steven James Levine and Brian Charles Williams. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*, November 2014.

[29] Michele Colledanchise. *Behavior Trees in Robotics*. PhD thesis, KTH, Stockholm, Sweden, 2017.

[30] Michele Colledanchise and Petter Ögren. 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, 2017.

[31] Raphaël Lallement, Lavindra De Silva, and Rachid Alami. Hatp: An htn planner for robotics. *arXiv preprint arXiv:1405.5345*, 2014.

[32] Caelan Reed Garrett, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research*, 37(1):104–136, 2018.

[33] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Stripstream: Integrating symbolic planners and blackbox samplers. *arXiv preprint arXiv:1802.08705*, 2018.

[34] Benoit Morisset and Malik Ghallab. Learning how to combine sensory-motor functions into a robust behavior. *Artificial Intelligence*, 172(4-5):392–412, March 2008.

[35] Leslie Pack Kaelbling and T. Lozano-Perez. Hierarchical task and motion planning in the now. In *ICRA*, pages 1470–1477, 2011.

[36] Leslie Pack Kaelbling and Tomas Lozano-Perez. Integrated task and motion planning in belief space. *Intl. J. Robotics Research*, 32:1194–1227, 2013.

[37] Jason Wolfe and B Marthi. Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling*, pages 254–257, 2010.

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

[39] D Hähnel, Wolfram Burgard, and Gerhard Lakemeyer. GOLEX – bridging the gap between logic (GOLOG) and a real robot. In *KI*, pages 165–176. Springer, 1998.

[40] Jens Claßen, Gabriele Röger, Gerhard Lakemeyer, and Bernhard Nebel. Platas—integrating planning and the action language Golog. *KI-Künstliche Intelligenz*, 26(1):61–67, 2012.

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

[42] Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Raman Kazhamiakin. Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*, pages 571–578, 2013.

[43] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *IJCAI*, pages 968–973, 1999.

[44] Zohar Feldman and Carmel Domshlak. Monte-carlo planning: Theoretically fast convergence meets practical efficiency. In *UAI*, 2013.

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

[46] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 6, pages 282–293, 2006.

[47] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. In *AAAI*, pages 3576–3582, 2017.

[48] F. Teichteil-Königsbuch, Guillaume Infantes, and Ugur Kuter. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *ICAPS*, 2008.

[49] Sung Wook Yoon, Alan Fern, and Robert Givan. Ff-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, pages 352–359, 2007.

[50] Sung Wook Yoon, Alan Fern, Robert Givan, and Subbarao Kambhampati. Probabilistic planning via determinization in hindsight. In *AAAI*, pages 1010–1016, 2008.

[51] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. HTN-MAKER: learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956, 2008.

[52] Chad Hogg, Ugur Kuter, and Héctor Muñoz-Avila. Learning hierarchical task networks for nondeterministic planning domains. In *IJCAI*, pages 1708–1714, 2009.

[53] Chad Hogg, Ugur Kuter, and Héctor Muñoz-Avila. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In *AAAI*, 2010.

[54] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *JAIR*, 4:237–285, 1996.

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

[56] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, 2013.

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

[58] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *CoRR*, abs/1609.05518, 2016.

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

[60] R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

[61] M. R. K. Ryan. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *ICML*, 2002.

[62] David Martínez Martínez, Guillem Alenyà, and Carme Torras. Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, 247:295–312, 2017.

[63] David Martínez Martínez, Guillem Alenyà, Tony Ribeiro, Katsumi Inoue, and Carme Torras. Relational reinforcement learning for planning with exogenous effects. *J. Mach. Learn. Res.*, 18:78:1–78:44, 2017.

[64] Aleksandar Jevtic, Adrià Colomé, Guillem Alenyà, and Carme Torras. Robot motion adaptation through user intervention and reinforcement learning. *Pattern Recognition Letters*, 105:67–75, 2018.

[65] Stéphane Ross, Joelle Pineau, Brahim Chaib-draa, and Pierre Kreitmann. A bayesian approach for learning and planning in partially observable Markov decision processes. *J. Machine Learning Research*, 12:1729–1770, 2011.

[66] Sammie Katt, Frans A. Oliehoek, and Christopher Amato. Learning in pomdps with Monte Carlo tree search. In *ICML*, 2017.

[67] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

[68] Pascal Hitzler and Matthias Wendt. A uniform approach to logic programming semantics. *Theory and Practice of Logic Programming*, 5(1-2):93–121, 2005.

[69] Andrey Kolobov Mausam. Planning with markov decision processes: an ai perspective. *Morgan & Claypool Publishers*, 2012.

[70] Subbarao Kambhampati. Are we comparing Dana and Fahiem or SHOP and TLPlan? A critique of the knowledge-based planning track at ICP. http://rakaposhi.eas.asu.edu/kbplan.pdf, 2003.

[71] Reid Simmons. Structured control for autonomous robots. *IEEE Trans. Robotics and Automation*, 10(1):34–43, 1994.

[72] Charles Lesire and Franck Pommereau. Aspic: an acting system based on skill petri net composition. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6952–6958. IEEE, 2018.

[73] Jan Peters, Jens Kober, and Duy Nguyen-Tuong. Policy Learning–a unified perspective with applications in robotics. *Recent Advances in Reinforcement Learning*, pages 220–228, 2008.

[74] Jens Kober. *Learning Motor Skills: From Algorithms to Robot Experiments.* PhD thesis, Darmstadt University, April 2012.

[75] Todd Hester and Peter Stone. TEXPLORE: Real-Time Sample-Efficient Reinforcement Learning for Robots. In *AAAI Spring Symposium*, pages 1–6, February 2012.

[76] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142, 2013.

[77] Jens Kober, J. A. Bagnell, and Jan Peters. Reinforcement Learning in Robotics: A Survey. *International Journal of Robotics Research*, August 2013.

[78] Michele Colledanchise, Ramviyas Parasuraman, and Petter Ogren. Learning of Behavior Trees for Autonomous Agents. *arXiv.org*, pages 1–8, April 2015.

[79] Qi Zhang, Jian Yao, Quanjun Yin, and Yabing Zha. Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution. *Applied Sciences*, 8(7):1077, July 2018.

[80] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Héctor Muñoz-Avila. Learning HTN method preconditions and action models from partial observations. In *IJCAI*, pages 1804–1810, 2009.

[81] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[82] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[83] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI*, 2002.

[84] Bhaskara Mannar Marthi, Stuart J Russell, David Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *AAAI*, page 1652, 2005.

[85] Christopher Simpkins, Sooraj Bhat, Charles Isbell, Jr., and Michael Mateas. Towards adaptive programming: integrating reinforcement learning into a programming language. In *ACM SIGPLAN Conf. on Object-Oriented Progr. Syst., Lang., and Applications (OOPSLA)*, pages 603–614. ACM, 2008.

[86] B D Argall, S Chernova, Manuela M veloso, and B Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.