A domain for the evaluation of RAEplan

James Mason

Abstract

The planning community has recently seen a surge in the development of integrated planning-and-acting systems. One system is RAEplan [10], which plans over *operational* models, which describe how to perform an action, rather than *descriptive* models, which describe what an action does. RAEplan is the first-known system to plan on operational models, which can be represented with a general-purpose programming language such as Python. RAEplan's development presents a new problem: the need for domains to test the effectiveness of operational model-based systems. In this paper we present the Order Fulfillment (OF) domain, a domain that requires online planning over concurrent tasks in a non-deterministic environment with robot collaboration.

1 Introduction

AI planning has traditionally used *descriptive* models for planning that describe what an action does and what state an actor will transition to, rather than how that action is to be performed [4]. As the planning community shifts its focus to the integration of planning and acting, descriptive models are often inadequate for modeling action execution, struggling to represent continuous actions and unexpected changes to the world state which may occur during execution [10]. This prompted the development of RAEplan [10], a planner that plans over *operational* models, which describe how an action is to be executed.

The acting side of RAEplan is RAE [4], which is inspired by the PRS system [7]. RAE uses a hierarchal, task-based language for acting. It supports typical programming language features such as loops, variables, and functions. Our implementation of RAE supports full Python code. RAE can arbitrarily choose from different options to accomplish tasks while reacting to dynamic events from the environment.

In RAE, *refinement methods* describe alternative approaches to accomplishing the same *task*. The refinement methods may contain arbitrary code, and may contain recursive *subtasks* or non-deterministic *commands*, which the actor can use to change the world state. These methods and commands are manually programed.

The planning side of the system, RAEplan, plans by performing Monte Carlo rollouts over these same refinement methods. RAEplan executes these commands and methods in a simulated world. This simulator is domain-dependent. We allow refinement methods to contain free variables, which RAEplan will assign based on some constraints. Each fully assigned refinement method is called a *method instance*. RAEplan looks at all or some of the method instances, and tells RAE the method instance it should execute.

We need new domains to evaluate this new kind of planner. These domains should be non-trivial and contain some combination of non-deterministic actions, dead ends, concurrent tasking, robot collaboration, and environment sensing. The goal of these testing domains is not to be realistic, but to provide a challenge to RAEplan and to demonstrate the types of challenges it can plan over.

We initially decided to evaluate RAEplan on the Robocup Logistics League Competition. After several months of effort, we were unable to make progress due to problems we encountered with setting up and connecting to the simulation. We instead decided to create our own domain.

We present the Order Fulfillment (OF) domain, a domain that requires online planning over concurrent tasks in a non-deterministic environment with robot collaboration. We also present several variations of the domain that provide dead ends, the need for sensing commands, and dynamic events. These variations can be mixed and matched to display various properties as desired. We also present limitations of RAEplan that we discovered during the creation of the domain.

This paper is structured as follows. In Section 2 we describe the OF domain and how you can create a task to redo commands with RAE. In Section 3 we describe several variations of OF and the problem with concurrent actions in RAEplan. In Section 4 we describe three bugs we found with RAEplan's implementation. In Section 5 we discus related work. In Section 6 we give our conclusions.

2 The Order Fulfillment Domain

The Order Fulfillment (OF) domain has several robots in a shipping warehouse that must co-operatively package incoming orders. An order contains a list of items that a customer wants delivered. The list contains some number of object classes, with duplicates allowed. An object of each class should be placed together in a machine,

which packs the objects together into a box. Only the items for a single order can be in a machine at the same time. The task is completed when the appropriate package is placed in the shipping doc.

It should be noted that this domain is not representative of how shipping warehouses work and it is not intended to be. This is a toy domain designed to be challenging for RAEplan.

Planning is necessary in order to complete the problem efficiently. A planner must decide which machine should be used for a particular order and which robot to use for a given task. Further complexity is added to the domain by allowing objects to be moved to a pallet instead of directly to a machine. This allows multiple orders to be processed simultaneously, and requires a planner to identify which orders to complete before others.

The OF domain is described by state variables including: robot $r \in R$, the set of robots; object $obj \in O$, the set of objects; machine $m \in M$, the set of machines; $loc(x) = L$ for $x \in R \cup O \cup M$ where L is the set of locations; and $type(obj) = objClass$ for $obj \in O$ and objClass, the type of object. The robots can execute commands such as $MoveRobot(r, loc_1, loc_2)$, $Pickup(r, obj)$, $Putdown(r, obj)$, $LoadMachine(r, obj, m)$, and $UnloadMachine(r, p, m)$. Unlike classical planning, these commands have durations that are modeled by random variables — another factor that must be considered while planning.

The description of methods for the top level *Order* task are given below. The task itself takes one argument, orderList, a list of the object types to be packaged. The first method moves each object in the order and places it in the machine one by one. The second method first stores the objects on a pallet to wait for a machine to become available.

```
// for free variables m and objList
Order_Method1(orderList, m, objList):
  wait() // may want to wait

  for i = 1 to len(orderList):
    if type(objList[i]) != orderList[i]: Fail()

    // move the object and load in machine
    // for free r in robots
    task: pickupAndLoad(r, objList[i], m)

  command: package(m, objects)
```

```
  // unload machine and move package to loading doc
  // for free r in robots
  task: unloadAndDeliver(r, m, package)
```

```
// for free variables m and objList
Order_Method2(orderList, m, objList):
  // here we move objects to the pallet
  // in more complicated implementations, we can
  // move a subset of items instead of all

  for i = 1 to len(orderList):
    if type(objList[i]) != orderList[i]: Fail()

    // move the object and place on pallet.
    // for free r in robots, p in pallets
    task: moveToPallet(r, objList[i], p)

  wait() // may want to wait

  // move objects from pallet to machine
  for each obj in objList:
    // for free r in robots
    task: pickupAndLoad(r, obj, m)

  command: package(m, objects)

  // unload machine and move package to loading doc
  // for free r in robots
  task: unloadAndDeliver(r, m, package)
```

Commands in this domain are non-deterministic and may not perform correctly. For example, the *Pickup* command may fail and have the object fall to the ground. When commands fail, RAE will try to switch to an alternative *method*. This behavior is often undesirable in the OF domain, where redoing the same *command* will typically result in the desired outcome. To solve this, we created the *Redoer* task. The *Redoer* task is a wrapper for a command that will try to redo the command up to n times in the case of a failure. This addition to the domain improves the success rate of OF problems.

```
Redoer(command, args):
  i = 0

  while(i < n):
    static redoID += 1
    id = redoID
    doCommand: command(args, id)
    if shouldRedo[id] == FALSE:
      return SUCCESS
    i += 1

  return FAILURE
```

Since RAE will not directly return the result of a command, the result is passed back through a state variable which relays if the command failed. *Redoer* can be further modified to describe the type of failure, which can be used to stop earlier if the command can never be completed in the current state.

3 Variations and Alternative Designs

We have considered several variations of the order fulfillment domain. Several of these variations can be used together so that OF displays various properties. These variations on OF have also revealed several issues with RAEplan's implementation.

3.1 Combining the logistics domain with a smart factory domain

When creating OF, we thought it would be best to start from a domain that is already used in the community. The logistics domain (one version described in [5]) is a simple planning domain where actors must deliver packages to various locations. We thought it would be interesting to combine this domain with a smart factory domain that would package the orders. The domain could also include multiple factories so that a planner would need to decide which factory a particular package should be processed in.

This domain is interesting from a planning perspective because there are many ways to perform the same task. Dynamic events such as traffic or bad weather are natural additions to the domain. The problem with this domain is that the costs of traveling to deliver packages are much higher than the costs of actions inside the

factory. This makes the choices within the factory fairly irrelevant when it comes to the overall picture, and the domain is therefore ineffective. We decided to remove the logistics challenges and focus on the smart factory because we want a domain that allows RAE to change the environment instead of simply traversing it.

3.2 Concurrent actions for the same task

Time in RAE is represented as a number $t \in \mathbb{N}$. Every top-level task, represented as a stack, is advanced by one step as t increases. One feature of RAE is the `concurrent` operator [4], which splits the method into k branches to be processed in parallel, all of which must be resolved before the original method is resolved. This allows multiple commands for the same method to occur in the same time step.

The `concurrent` operator could be useful for the *Order* task, allowing multiple robots to move objects to a packaging machine simultaneously. This method can better utilize a factory's resources if there are more robots than orders.

While good in theory, the `concurrent` operator is hard to use in practice. [4] describes the operator as "[splitting the stack] into k substacks." This ignores the issue of how the substacks must maintain some level of co-dependence. If each substack has its own set of state variables, the substacks can become inconsistent with one another. If all of the substacks share the same state variables, then you can't plan over each individual substack.

Due to these implementation problems, we have not implemented the `concurrent` operator in our version of RAE. Since we lack the operator, this method for *Order* is similarly unimplemented. Future work should be done to implement this important feature of RAE.

3.3 Sensing Commands

RAEplan is capable of planning without full knowledge of the domain. RAE can use sensing commands to learn information about its environment, such as if an object is present at a location. This capacity can be demonstrated in OF by requiring RAE to use a database to find the object. We decided not to use this feature because it doesn't seem realistic for our domain; a smart factory should be able to keep track of the objects within it without making lots of database queries.

3.4 Machine Failure and Repair

In the real world, things go wrong. Dynamic events occur. RAE is capable of responding to dynamic events, so it is desirable to include this feature in the domain. The most realistic event that can occur in a smart factory is a machine failing.

The Industrial Plant (IP) domain [10], a domain similar to ours, can have machines dynamically break. When a machine breaks, it can not be used until it has been repaired. This idea is good, but we decided to try something different. We wanted to test if RAEplan would take preventive measures in order to reduce the chance of a failure.

We set the chance that a machine could fail during use to be a function of the number of times the machine has been used, with more uses increasing the chance of it failing. Instead of in IP, if the machine fails it could be used again without repair, just with a higher chance of another failure. If fixed by a robot, the counter for the number of machine uses would be reduced.

We found experimentally that this approach is ineffective at prompting RAEplan to decide to take the time to repair the machine. It viewed the cost of the machine failing to be less than the cost of repairing the machine unless the cost of repair was near 0. Perhaps if our experiments were over a larger number of orders RAEplan would repair the machine, but for a small number of orders RAE rarely considers the option. We therefore suggest using the IP approach to machine failure, as our approach was found to be ineffective.

3.5 Movement Restrictions

How robots move is largely ignored in our implementation of OF. When executing a *Move* command, our implementation maintains the robot in its original position until the command has finished, when it is then said to be at the final destination. This isn't a problem since a robot can't affect other robots, but it is not realistic.

A more interesting version of the problem stipulates that only one robot can be in a particular location at any given time. We would then have to move the robot by one location each time step. It could also create possible dead ends, a property that OF doesn't have. We believe that this would be a good addition to the domain but we didn't implement it.

4 Bugs with RAEplan

During the development of OF, we discovered several bugs with RAEplan's implementation.

The earliest bug encountered is that when planning over a non-trivial number of commands, RAEplan reaches Python's maximum recursion depth, even though the depth of the refinement tree was small. We worked with the developers of RAEplan and discovered that they were using a recursive implementation of Monte Carlo rollouts that repeatedly rebuilt the tree while performing rollouts, which takes exponential space. The developers of RAEplan have sense changed their Monte Carlo rollout implementation so that the tree can be reused and the results stored in linear space.

The original RAEplan implementation was missing a key feature: the ability to use uninstantiated variables. Without using uninstantiated variables, the only way for RAEplan to search over various options is to create tasks that will instantiate them for you (e.g. *ChooseRobot*). This means that RAEplan could only plan over a small subset of method instances, equal in size to the number of combinations of methods for choosing how to instantiate the free variables. After bringing this to their attention, the developers of RAEplan have sense implemented this feature, allowing RAEplan to search over a much larger search space.

While testing RAEplan with Bash scripts, we also discovered that the RAEplan system sometimes takes unexpectedly large amounts of memory. These tests are designed to stop searching if the tasks are not completed after a certain amount of time. We believe that RAEplan is orphaning child processes when it is unable to solve the problem. Our findings were shared with the developers of RAEplan, but they were unable to reproduce the bug. This issue is still being explored.

5 Related Work

To our knowledge, four other domains [10] for RAEplan have been created and a fifth is in development. The Explorable Environment (EE) domain has more tasks, refinement methods, and commands than OF does, and it has dead ends. The Chargeable Robot (CR) domain has dynamic events and dead ends. CR also uses sensing commands to identify the locations of objects. The Spring Door (SD) domain is a dynamic environment with lots of robot collaboration. The Industrial Plant (IP) domain is the most similar to OF. IP is based off of the smart factory environment as described in the RoboCup Logistics League competition. It requires orders to be processed in various machines, which may be damaged and require repair. The main

difference between OF and these domains is that OF has a much higher branching factor than they do, especially when OF is handling more orders than machines. This is especially true because these domains were designed before the uninstantiated variable feature was implemented, reducing the branching factor even further. These domains have static command durations. The command durations in OF are random variables. While the other domains have properties such as dead ends and sensing which OF doesn't have, these properties can be added to OF as described in Section 4.

There are few other domains created for planning and acting over operational models, as most algorithms plan over descriptive models. RAE [4] (without RAEplan) and PRS [7] act over operational models, but make choices without planning ahead. Due to this limitation, domains created for RAE and PRS don't take efficiency into consideration.

There have been several proposed approaches to the integration of planning and acting (e.g. PropicePlan [2], SeRPE [4]). These algorithms plan over descriptive models, not operational models. Unlike RAEplan, These approaches generally can't handle unexpected events.

RMPL [6] can handle many complex domains using constraint satisfaction techniques. It generates descriptive models based on programs written in RMPL. In order to plan [12] [1], RMPL is parsed into a temporal planning problem. RMPL has been extended to handle error recovery [3] and probabilistic models [11] [8]. Unlike RMPL, RAEplan doesn't use a temporal approach and can plan over commands with unknown time constraints.

HTN planning (e.g. SHOP [9]) uses a special case of RAE's refinement methods which it can plan over very quickly. HTN planning doesn't consider acting and can't express as much as RAE's general methods. This makes HTN domains undesirable for our purpose.

6 Conclusion

We have proposed a new domain for the evaluation of RAEplan and other similar systems, and presented several variations on the domain. We presented Redoer, a technique to redo commands with RAE in the case of unexpected failures. The domain will be used for testing in an upcoming journal paper on RAEplan.

We have also shown problems with RAE's `concurrent` operator. Concurrency is an important feature for robot collaboration, and future work should be done to resolve this issue.

We did not present data in this paper. Preliminary data we collected led us to discovering a bug with instantiated variables that is discussed in Section 4. We have sense had to make a large number of changes to RAEplan and the OF domain, and have been unable to collect adequate amounts of data before the submission deadline. Future work will involve collecting data on the handwritten problems we have created. We also plan to test on computer-generated problems for the domain.

References

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

[2] Despouys, O., and Ingrand, F. Propice-plan: Toward a unified framework for planning and execution. *ECP* (1999).

[3] Effinger, R., Williams, B., and Hofmann, A. Dynamic execution of temporally and spatially flexible reactive programs. *AAAI Wksp. on Bridging the Gap between Task and Motion Planning* (2010).

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

[5] Helmert, M. On the complexity of planning in transportation domains. *Proc. AAAI* (2013).

[6] Ingham, M. D., Ragno, R. J., and Williams, B. C. A reactive model-based programming language for robotic space explorers. *i-SAIRAS* (2001).

[7] Ingrand, F., Chatilla, R., Alami, R., and Robert, F. Prs: A high level supervision and control language for autonomous mobile robots. *ICRA* (1996), 43–49.

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

[9] Nau, D. S., Cao, Y., Lotem, A., and Muoz-Avila, H. Shop: Simple hierarchical ordered planner. *IJCAI* (1999).

[10] Patra, S., Traverso, P., Ghallab, M., and Nau, D. Acting and planning using operational models. *Proc. AAAI* (2019).

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

[12] Williams, B. C., and Abramson, M. Executing reactive, model-based programs through graph-based temporal planning. *IJCAI* (2001).