Learning Operators for Manipulation Planning

Chris Burbridge[1], Zeyn Saigol[1], Florian Schmidt[2], Christoph Borst[2] and Richard Dearden[1]

Abstract— **We describe a method for learning planning operators for manipulation tasks from hand-written programs to provide a high-level command interface to a robot manipulator that allows tasks to be specified simply as goals. This is made challenging by the fact that a manipulator is a hybrid system—any model of it consists of discrete variables such as "holding cup" and continuous variables such as the poses of objects and position of the robot. The approach relies on three novel techniques: the action learning from annotated code uses simulation to find PDDL action models corresponding to code fragments. To provide the geometric information needed we use supervised learning to produce a mapping from geometric to symbolic state. The mapping can also be used in reverse to produce a geometric state that makes a set of predicates true, thus allowing desired object positions to be generated during planning. Finally, during execution of the plan we use a partially observable Markov decision problem-based planner to repair the initial plan when unforeseen geometric constraints prevent actions from being executed.**

I. INTRODUCTION

The vast majority of robots are currently controlled through hand-written programs that are highly specialised to a single task. Where there is any flexibility or generalisaition it is typically only over the locations of the objects [6]. Hand coding every robot behaviour is clearly unsustainable as robots become more common and are operated by non-programmers. Instead we are interested in simply specifying goals for the robot and using a planner to generate a behaviour that achieves the goals. However, we should be able to leverage the large body of human-generated *example programs* for the robot that often exists. This work does this by finding, in a semi-automated way, useful fragments of the programs that can be treated as actions for a high-level planner. The planner can then be used to construct new programs to achieve desired states of the world. For example, we might want a robot to place a number of cups on a tray. Rather than creating a bespoke program for the task, we specify that each cup should be on the tray, and then let the planner generate a plan, which is then translated into a program for achieving the goal.

Planning actions are typically described in terms of their *preconditions*—the things that must be true before the action can be performed—and *effects*—the changes the action makes to the world state. Therefore, to define planning actions from the example programs we need to determine what these preconditions and effects are. We do this by searching for fragments of code that are common between the programs and then simulating the program execution to determine what is true before and after a fragment of code. The actions are represented using the PDDL planning domain definition language [5]. To give the reader a complete view of our approach, we describe the action learning in more detail in Section II, however we claim little novelty in this section, following [17] for the learning.

For efficient planning, we need to specify action preconditions and effects symbolically as a predicate such as `on(cup,tray)`. Unfortunately this symbolic state is rarely explicit in the robot programs. Continuing the example, a program to place a cup on a tray will typically specify where to place the cup (perhaps in tray coordinates). To extract the symbolic state from the programs we first need the geometric states that result from executing them. We then label these using a pre-defined set of relevant predicates, and then use a supervised learning technique to learn a mapping from geometry to predicates. Once we have a symbolic plan we also need to generate geometric states, so we choose as our learned representation a kernel density estimate as we can use that both forward and backward. The learning approach, along with the details of how we use the predicates, is in Section III.

To generate plans we use a standard classical planner, *Fast Downward* [7]. This produces symbolic plans which we then generate geometric states for using the backward direction of the predicate mapping. More details of the planner are in Section IV.

Since hand-written programs typically compute path plans on the fly, the generated programs do the same. However, this approach means that in some circumstances the planner can generate plans that are not executable because other objects either obstruct the geometric positions in the plan or prevent an arm motion path from being found. To overcome this we use a second planner to repair the plans at execution time. This planner treats various computations (finding a geometric state for a symbolic state, or running the RRT planner) as actions that provide information about the task, and minimises the time for the computations plus the time to execute the plan. We model this problem as a partially observable Markov decision problem (POMDP) [2] and solve it using ZMDP [16] to find a policy to resolve the problem with the original plan and allow it to be executed. This work is described in Section V.

To evaluate the approach, we have run it in simulation, on

This work was supported by the European Community's Seventh Framework Programme as part of the GeRT project, grant agreement no ICT-248273.

[1]School of Computer Science, University of Birmingham, Edgbaston, Birmingham, B15 2TT, U.K. {C.J.C.Burbridge, Z.A.Saigol, R.W.Dearden}@cs.bham.ac.uk

[2]Institute of Robotics and Mechatronics, German Aerospace Center (DLR), Wessling, Germany {florian.schmidt, Christoph.Borst}@dlr.de

Predicate	Meaning
(above X Y)	Object X is above object Y
(touching X Y)	Object X is touching object Y
(object X)	X is an object
(cup-type X)	X is a cup
(tray-type X)	X is a tray
({right|left}-can-grasp X)	The hand is in position to grasp
({right|left}-holding X)	The hand is holding object X

TABLE I: Symbolic state predicates in the planning domain.

DLR's Justin platform [6], and on a Kuka industrial arm. The results are described in Section VI. Finally we draw some conclusions and discuss future directions in Section VII.

II. ACTION LEARNING

Human written robot control programs consists of many 'actions', each action potentially spanning multiple lines of code. The first challenge is to autonomously dissect the program, extracting re-usable parametrised code fragments that perform some useful action. An action could be anything from a single program instruction that affects the internal state of the computer to the execution of an entire, possibly parametrised, example program.

We consider an action to be a section of a program that changes the symbolic description of the geometric environment. This strikes a compromise between creating hundreds of low-level actions that make planning difficult, and having very few overly specialised actions that are hard to re-use.

The second challenge is to describe the source code as a *planning operator* in PDDL, thereby allowing an automated planner to generate plans comprising of the code fragments. Learning of planning operators has been considered by many researchers [17], [18], [11], [19]. Our method for calculating the symbolic effects of actions closely follows [17]. However, we leverage information present in the example programs to make learning easier.

In this work two example programs have been used for action learning, one that places a cup on a tray and another that places a cup on another cup. Each is written as a Python script that makes calls to external perception and path planning modules and uses fixed hand-taught positions to grasp and place objects. Both example scripts can be executed either directly on the robot or in an OpenRAVE [3] based simulation.

The predicates available for symbolically describing the geometric state have been specified by hand to fit with the tasks of cup stacking, as summarised in Table I. In addition, we provide the learner with semantic knowledge that the `can-grasp` and `holding` predicates are mutually exclusive.

A. Discovering actions

In order to discover what actions a hand-written example carries out, the script is automatically stepped through in the robot simulation environment. After each line of code the current geometric state is recorded. The geometric state consists of the full 6D pose of all objects in the environment together with all joint angles of the robot.

The logged geometric states are then transformed into symbolic state descriptions through a set of mapping functions. These mapping functions are of the form $f(G) \mapsto s$ where G is the geometric state and s is a partial symbolic representation of G containing clauses of the predicate f that hold in G. The full symbolic representation of G is then the union of each symbolic predicate mapping.

Finally each symbolic state description is compared to the next. When they differ a new action is flagged and the effect of the action is recorded as what changed in the state.

B. Learning planning operators

To describe the actions in the form of PDDL operators, both the effects of the action and the preconditions of its application need to be ascertained. Furthermore, in order to be of more general use and allow planning of new tasks that don't feature in the example programs, it is necessary to be able to parametrise the actions. For example, an action that grasps an object `cup2` should be parametrised so as to be applicable to objects other than `cup2`.

The effects of the action can be directly recorded as the clauses that were added to or deleted from the symbolic state while discovering the action. It follows that the constants that appear in the clauses in the effects list can provide the first level of parametrisation for the action. For example, if an action was found to add the clause `(above cup tray)` to the symbolic state, then `cup` and `tray` are the objects concerned by the action and should be the action parameters.

Determining the preconditions of the action involves deciding what clauses of the symbolic state before the action are critical for the action. The decision will directly affect the generality of the learned action. On the one hand, all elements of the symbolic state before execution could be included as preconditions, leading to actions with artificially low applicability. On the other hand, no preconditions could be recorded leading to inapplicable actions being selected at possibly hazardous times.

Different levels of preconditions have been considered. At the top-level, clauses of the state that involve only the objects that appear in the effects list (the concerned objects) or no objects at all are added. At the next level, clauses involving the concerned objects along with another object (a secondary object) are added, and at the next level, clauses concerning secondary objects with another object are added, and so on. We found that a second level of preconditions suitably specified the actions without over restricting their application.

An example of a learned planning operator expressed in PDDL[1]:

```
(:action learned_act_6 :parameters ( ?param0 ?sub_param1 )
  :precondition (and (hand-holding ?param0)
    (object ?param0) (touching ?param0 ?sub_param1)
    (above ?param0 ?sub_param1) (type ?param0 cup-category)
    (touching ?sub_param1 ?param0)   )
  :effect (and (hand-empty) (can-grasp ?param0)
    ( forall (?ALL)( not (can-grasp ?ALL)))
    ( not (hand-holding ?param0))    )
)
```

[1]The universally quantified *forall* in the effects is automatically generated from the semantic information about the predicate.

Action	Example 1	Duplicate in example 2?
1	Ready to grasp(?p0)	Yes
2	Grasp(?p0)	Yes
3	Lift of (?p0,?p1)	Yes
4	Move above(?p0,?p1)	No
5	Place on(?p0,p1)	No
6	Release(?p0)	Yes
7	Move arm away(?p0)	Yes

TABLE II: The actions learned from the example programs.

C. Extracting source code

The Python source code that corresponds to the actions and allows constructed plans to be turned into robot code is extracted alongside the action discovery. However, at this point the source code extracted for the action is specific to the objects and scene that the example script ran in. With sufficient example programs to learn from, a more general action could be learned directly. At present, due to the small number of examples, we parametrise the script using annotations in comments in the source that indicate frame or string parameters. A *string annotation* indicates to replace a given string with the action parameter value. In the example programs, *frame*s are geometric locations retrieved from a database; a *frame annotation* indicates that a frame should be replaced by one generated during the planning process (as described in Section III).

During experiments we found that it was necessary to add a further source annotation to inform the action learner that some groups of lines should not be split between actions. Although this only occurred at one point in each example program, in future work we aim to include syntactical analysis of the code to avoid such annotations.

D. Results

Executing the described action learning on the two example scripts resulted in 7 actions being extracted for each example. Five of the actions were common between the two domains, resulting in 9 distinct actions overall.

The learned actions are summarised in Table II. Actions 3, 4 and 5 were similar but not identical between example programs. The reason for this was that example script 1 taught the robot that it was possible to place a cup on a tray, while example script 2 taught the possibility of placing a cup on another cup.

III. GEOMETRIC AND SYMBOLIC STATES

The learned actions in the previous section provide the robot with the basis for planning and carrying out new tasks that are beyond those demonstrated in the example scripts. However, they do not provide a complete solution as so far only the symbolic meaning of the programs has been considered. This means that although a symbolic plan could be generated that moves a cup "above" another cup, the robot would have no means of specifying the geometric configuration needed for the generated Python script. To do this, we learn a bi-directional mapping between geometric and symbolic state. The mapping allows us to generate a symbolic representation of a scene, but also to generate a geometric state consistent with a symbolic effect.

A. Related Work

The idea of learning a mapping from geometric state to symbolic state has been considered before in the literature. However, what sets our approach apart from others is the idea of using the learnt mapping in reverse, generating geometry from symbolic description to enable symbolic planning operators to be utilised to produce geometric effects.

Jakel et al. [8] present an approach to action learning and geometric constraint learning in the context of Programming by Demonstration. After segmenting logs of human demonstrated actions, constraints on the possible motion of the robot arm are learnt. However, constraints on object positions are not directly considered.

Recent work of Sjöö et al. [14], [15] has also addressed spatial relationships. In [14] they consider the geometric meaning of "on", and use hand built rules to map from geometric to symbolic state. Although they go on to use the rules to construct a probability function of the "on" predicate, the representation chosen does not generalise to other predicates and requires hand construction.

In [15], the same authors address the issue of hand building rules, presenting a system than learns to classify object relationships by generating samples in simulation. While classification of several relations is successful, using the learnt relation in reverse is not considered. Furthermore, the geometric state is described by a 93 dimensional feature vector comprising contact points, surface normals and object poses. The high dimensionality makes reversing the mapping impossible.

An approach similar to that in [15] is presented in [13]. Again, contact points are used, this time between segmented point clouds. However, the use of K-Nearest neighbours for classification again prevents using the classifier in reverse.

B. Our Approach

Our approach is to learn a model of the probability density function $p(C|G)$ where C is a symbolic state clause, G is a geometric state, and a probability of 1.0 indicates that C holds in the given G. Given the symbolic state $S = C_1 \wedge C_2 \wedge ... \wedge C_n$, assuming that probability of each clause is independent, the probability of it holding in a given geometric state can then be calculated as follows:

$$p(S|G) = \prod_{i=1}^{n} p(C_i|G). \quad (1)$$

Our aim is to learn a classifier capable of producing class probability estimates from a number of labelled positive and negative examples of (G, C) pairs. Most importantly however, we choose a classifier and state representation that allow optimising the probability and turning a given symbolic state back into a geometric state.

1) Geometric state representation: For a symbolic predicate of two objects A and B, the geometric state representation is:

$$G = \left[\begin{array}{ccccc} S_A & S_B & T_B^A & R_A & R_B \end{array} \right] \quad (2)$$

where S_o, R_o are the size and rotation of object o respectively, and T_B^A is the relative translation between objects A and B. Using the difference in position between the objects rather than absolute positions provides better generalisation from the examples. However, the same can not be done with the rotation as many predicates require specific orientations (`pouring` for example). We represent the rotations using quaternions, taking the geometric state vector to $17D$.

The size of the each object is expressed as the dimensions of its bounding box. Including this in the state allows us to learn just a single general distribution per predicate rather than, for example, needing to learn distributions for both `above(cup,tray)` and `above(cup,cup)`.

2) Probability model: It is important to have a smooth continuous representation of $p(C|G)$. This allows points of high probability in the multi-dimensional space to be located using numerical methods, which in turn allows finding geometric representations for a given symbolic state. Modelling the probability using Gaussian Kernel Density Estimation (KDE) provides such a smooth function.

We assume non-informative class priors, so given a set of p geometric states G^P that a predicate holds in, and a set of n geometric states G^N in which the predicate does not hold, the class probability is:

$$p(C|G) = \frac{P(G)}{P(G) + N(G)}, \quad (3)$$

where $P(G)$ and $N(G)$ are the multivariate KDEs of the positive and negative datasets respectively evaluated at G:

$$P(x) = \frac{1}{p}\sum_{i=1}^{p} e^{-||G_i^P - x||^2/h^2} \quad (4)$$

$$N(x) = \frac{1}{n}\sum_{i=1}^{n} e^{-||G_i^N - x||^2/h^2}, \quad (5)$$

where h is the kernel bandwidth. We use the FigTree Fast Gaussian Transform library [10] for efficiency.

3) Using the learnt mapping: The learnt mapping can now be used both to classify geometric states into symbolic states, and to generate geometric states that comply with desired symbolic states. Classification can be achieved by thresholding the class probability at some sensible value, depending upon how certain the classification needs to be.

Generating geometry for a single predicate goal is carried out by maximising the $17D$ probability function, keeping the 6 elements corresponding to object size constant. The smooth and continuous nature of the function allows maximisation to be carried out numerically using hill climbing from an initial estimate selected from G^P. We found the Broyden-Fletcher-Goldfarb-Shanno optimisation method to perform suitably well for this purpose.

When considering more than one predicate clause using (1), the dimensionality of the space increases and dependencies between objects have to be taken into account. When an object appears in multiple clauses, the size and orientation of the object only appear once in the geometric state. In addition to this, the translations between objects are never allowed to form a cycle. For example, if the state contains the translation between A and B, and between B and C, then there must be no translation between A and C. This is ensured by generating a graph of the predicate relations and only considering the translations for predicates that form part of the graph's spanning tree.

C. Results

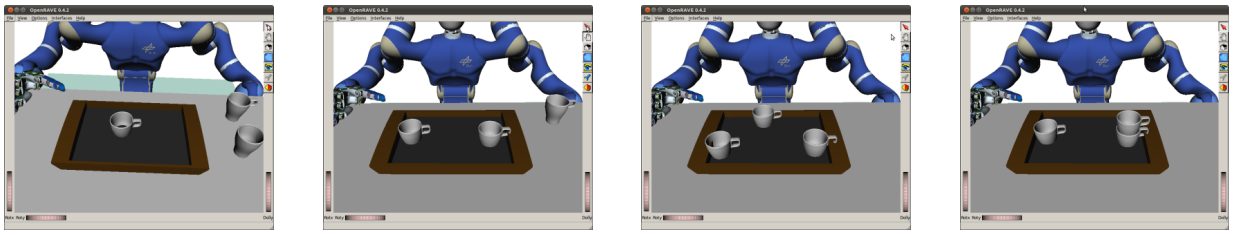
2000 training examples of geometric arrangements of a cup and a tray and of two cups were created. These were labelled with 1672 negative and 328 positive examples of the "above" predicate found, and 1872 negative and 128 positive examples of the "touching" predicate found. These samples were then used to learn distributions for each predicate. The result of using the learnt mapping to generate geometric states from symbolic states is demonstrated in Figure 1.

IV. PLANNING FRAMEWORK

As we said in the introduction, we can't rely on pre-built plans executing perfectly. Also, we would like the robot to at least start moving as early as possible. Therefore we perform initial planning fast and rely on execution-time repairs (described in Section V) to fix problems with the plan.

The planning task is made challenging by the mixture of geometric and symbolic state. To plan optimally for both of these we would need a planner where a geometric problem (e.g. a collision between objects) led to backtracking in the symbolic planning and vice versa. However, this is slow because geometric states and arm motion paths will be generated for states that are reached during search but never used in the final plan. In practice we find that for many tasks the geometric problem is straightforward, and even when it is not, it can often be fixed with only small changes to the plan. For this reason, we take a sequential approach: we generate a symbolic plan with no considerations of geometric constraints, then add the geometry using the symbolic-geometric mapping described in Section III.

Given a PDDL domain description, consisting of the learned planning operators, an initial state determined using the learned geometric to symbolic mapping, and a goal provided by the user, the high-level algorithm is given in Algorithm 1. We use Fast Downward [7] as the planner, using the FF heuristic. This generates a symbolic plan. We then simulate the plan symbolically to generate the sequence of states the robot will visit when executing the plan. Each of these states consists of a set of predicates that is true at that point in the plan. For each state, we then use the geometric to symbolic mapping in reverse to generate a geometric state in which the symbolic state predicates would be true with high probability. We do this state-by-state translation in reverse



(a) (b) (c) (d)

Fig. 1: Geometric states generated for given symbolic states. (a) `1 cup above and touching tray`; final probability=0.989. (b) `2 cups above and touching tray, not touching each other`; final probability=0.977. (c) `3 cups above and touching tray, not touching each other`; final probability=0.941 (d) `2 cups above and touching tray, 3rd cup above and touching 2nd`; final probability 0.887

order and when we search for a corresponding geometric state we only allow objects that have been touched by the robot up to that point in the plan to be moved.

Algorithm 1 High-level description of the planning approach.

start-state ← geometric-symbolic mapping applied to geometric state
plan ← symbolic planner applied to learned actions, goal, start-state
states ← the sequence of states visited by the plan
for $s \in$ states, starting with the final state **do**
    Find a geometric state that matches all geometric predicates in s with high probability
end for
Execute the plan

To illustrate the algorithm, consider an initial state with `cup1` and `cup2` on the table, and a goal state with `cup2` nested inside `cup1`. We represent this goal as `(above cup2 cup1)` ∧ `(touching cup2 cup1)`. Assuming we have actions to `pickup`, `putdown` and `stack` cups, the symbolic planner generates the simple plan `pickup(cup2),stack(cup2,cup1)`. We then simulate this to get the sequence of states:

```
{{(above cup1 table), (touching cup1 table),
(above cup2 table), (touching cup2 table)}, {(above
cup1 table), (touching cup1 table), (holding
cup2)}, {(above cup1 table), (touching cup1 table),
(above cup2 cup1), (touching cup2 cup1)}}
```

We then translate to geometric states starting with the last set of predicates.We cannot change the position of `cup1` since the plan doesn't involve the robot moving it, so we search for a position for `cup2` that satisfies all the predicates. This produces a position for the cup that is very close to the position of `cup1` but displaced vertically. We then continue backwards through the plan generating geometric positions; for this simple plan this is the only one that is required.

Once we have a plan with geometric states (we will refer to this as a *hybrid plan*) we execute it by using the same code the action was learned from, with the geometric positions filled in from the hybrid plan. Execution of actions which move the arm require a path planner to determine how the arm will get to a suitable position for manipulating the object. At run-time we use the RRT path planner from OpenRAVE to generate paths to the hand positions.

A limitation of this approach is that it may fail in domains where geometric constraints can produce 'dead-ends'.However, these domains can be detected a priori and an alternative planning approach taken.

V. EXECUTION

Execution monitoring is often required in real-world robotic systems [12], as it is almost impossible to generate plans that are robust to anything that may happen during execution. We take advantage of the fact that the programs we generate do path planning at run time (because the example programs do) by performing an execution-time repair whenever geometric (or other) considerations cause a plan to fail. This approach is sub-optimal in cases where geometric failures could be resolved by a different task-level plan, which a backtracking hybrid planner could produce (for example [1], [4]). However we believe that these cases are rare and the primary concern for most systems is simply successful execution; our framework can easily be extended to encode domain-specific fixes to common problems that are encountered when executing plans, making successful execution runs easier to achieve.

A. Framework

Hybrid plans are generated by the system outlined in Section IV and sent to the robot's control system via an execution monitor. The execution monitor checks for specific classes of failure, and invokes a repair module associated with that class when a failure is detected. The repair modules can choose from a set of hand-coded actions and they may need to perform a sequence of actions in order to repair the plan. Once the plan has been corrected, control is passed back to the execution system.

The repair options for each class of failure are encoded as a POMDP, which model stochastic state transition systems (MDPs) where the system state is not directly observable by the agent. Using a POMDP means a near-optimal repair plan can be generated given the current estimate of the system state and the uncertainty over the cause of the failure and the effects of potential repair actions.

A POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \mathcal{Z}, Z \rangle$, where $\mathcal{S}$ is a set of possible states, $\mathcal{A}$ is a set of actions available to the agent,

$T = \text{Prob}(s'|s,a)$ is a state transition function, $R(s,a)$ is a reward function, $\mathcal{Z}$ is a set of possible observations, and $Z = \text{Prob}(z|s,a)$ is an observation function. The agent's aim is to maximise its expected total discounted reward, using a discount factor γ, $0 \leq \gamma < 1$, at each timestep so future rewards are valued lower.

For our repair modules, the aim is to repair the plan as quickly as possible, so rewards are modelled by the negative of the expected time to perform each action. The solution of a POMDP is given in the form of a *policy*, which specifies the action to take for any estimated state (where the state estimates take the form of a *belief state*, a probability distribution over world states). The POMDPs generated are relatively small, of the order of dozens of states, and a full policy can be quickly generated using ZMDP [16].

B. Resolving Geometric Conflicts

The first class of plan failure considered is when path planning fails because there is no possible non-colliding configuration of the robot to achieve the target pose.

There are several options for plan repair given a geometric collision failure, depending on the exact cause.

1) If the target pose is out of reach for the robot, and the immediate goal is to grasp an object, the repair planner can search over the available actions for an action with the same pre- and post-conditions but different geometric properties, which we hope corresponds to a different type of grasp for that object. For example, the arm may not be long enough to grasp a cup from above, but a side grasp may still be possible.
2) If an obstacle is blocking the target pose, two options are available: the obstacle can be moved elsewhere, if it is moveable, or if the robot is putting down an object, a new target geometry can be generated which takes the obstacle into account.
3) Otherwise, if the target pose is out of reach or the obstacle cannot be moved, the repair planner must report failure.

The first component of the repair POMDP's state space is the result that running the RRT planner would produce, where we make the approximation that the RRT outcome is deterministic. In total, the state space is factored into four variables: `RRT-result` with domain {success, no-solution, movable-obstacle, immovable obstacle}, and three Boolean variables, `free-arms-exist` (tracking whether there is at least one arm not currently holding anything), `alternative-grasps-exist`, and `alternative-geometries-exist`.

There are five actions available to the repair planner: re-run the RRT path planner; try to find a new grasp; try to find a new geometry; move an obstacle; or report failure.

Observations are produced by running the RRT planner, trying to find an alternative grasp, or trying to find an alternative geometry. These observations provide a known value for the corresponding state component, but there may still be a probability spread over the subset of states that match that value.

Finally, the transition and reward functions are estimated from statistics collected by running the system. The reward function models the average time to perform an action, where the aim is to minimise this time. Note that moving an obstacle has much lower cost from states where `free-arms-exist` is true, as a free arm can be used. The repair module terminates when it either performs the "report failure" action, or the "RRT" action produces a valid path.

C. Idle-Time Optimisation by Obstacle Removal

The second class of failures addressed by the repair planner is motivated by the fact that RRT path planning in the high-dimensional configuration space of Justin can be a lengthy process, often taking 30 seconds or more. During this time the robot is idle, and in human-robot interaction terms this is highly undesirable—humans will often think the robot is broken. To address this, we take advantage of the sampling nature of RRT planning to define two versions of the planner: an RRT-long planner (with essentially unlimited samples), and an RRT-short planner where we set a limit on the number of samples that can be generated. In normal operation, we only call the RRT-short planner, which limits the time that can be spent on planning complex paths. This is useful because *impeding obstacles* can often be identified when the RRT search is truncated before a path is found. We define an impeding obstacle as an obstacle which forces complex paths to be generated, as opposed to a *blocking* obstacle which prevents any path at all from being generated. The principle of the idle-time optimisation repair is that an impeding obstacle can often be moved elsewhere relatively quickly, and then the RRT-short planner will be able to find a path to the original target pose.

To find impeding obstacles, some minor modifications to the OpenRAVE RRT planner were made. OpenRAVE uses a bi-directional RRT [9], where candidate paths are built incrementally by stochastically adding small (configuration-space) steps to a path. These small steps are biased towards the target configuration, rather than being a uniform random exploration of the search space. Candidate paths are discarded if the path would cause a collision. The modification keeps track of which objects caused path samples to fail and reports the most common object as the impeding obstacle. Note that RRT-short can also fail without there being an impeding obstacle.

The repair options, each a POMDP action, are to:

- Re-run RRT-short (which will only be worthwhile after performing some other corrective action)
- Run RRT-long
- Move an impeding obstacle out of the way
- Move the arm into a high position. The intention of this action is that the arm can easily and safely be moved in to a position high above the workspace, from where it is a much simpler path-planning problem to reach the desired pose.

The state space of the repair POMDP is defined by the result that running the short/long RRT planner would produce,

whether the arm is currently in the high position, and whether we know of an impeding obstacle or not. These variables give a POMDP with 24 states (as each RRT planner has three possible results, a path, an obstacle, or inconclusive). Given the actions, the observation function Z is deterministic. The remaining parts of the POMDP model are again estimated: the transition model based on how likely moving an obstacle or raising the arm would be to enable a path to be found by the RRT planner, and the reward function on how long a small set of trial actions took to perform in simulation.

Initial experiments have indicated that the robot's idle time before moving can be reduced from 30s to 15s, at the cost of slightly longer overall wall-clock time for the planning and execution of a given task. For scenarios where the robot is interacting with (or being observed by) humans, this makes it a worthwhile repair module.

D. Summary

The repair planner is at a proof-of-concept stage, and requires further evaluation to confirm that it is capable of handling a range of geometric failure conditions.

Future development work includes adding specialised modules to cope with other plan failures, as further experiments on the robot make it clear what kinds of failures are common and how they can be fixed. One candidate would be to reapir failed grasps.

An open issue is how to decide where to move an obstacle as the current mechanisms do not guarantee that the obstacle is no longer in the way once it has been moved, or that it will not obstruct some other future action, leading to more plan repairs later.

VI. EVALUATION

The four scenarios in this section demonstrate each of the elements of the system outlined in the previous sections, and show where and why the system can fail. To demonstrate the generality of the system, we use two different robots: a single 6 DOF Kuka industrial arm with a gripper (KR5-Sixx R850), and a two-armed research platform with multi fingered hands (DLR Justin). Object localisation is performed using a Kinect RGBD camera. The point cloud is segmented into clusters above the table plane, then each cluster centroid gives the location of the objects. The attached multi-media provides video clips of each scenario.

1) Scenario 1: Clear cups to tray: Fig. 2 shows a trial of the planning framework running on the Kuka robot. Given the goal of three cups being above and touching the tray and the fourth above and touching another, the symbolic planner created a plan with the actions learnt in Section II. Geometric states were then automatically generated using the learned symbolic-geometric mapping. The executed plan comprised of 24 steps, utilising actions learned from both example programs.

2) Scenario 2: Stack two cups: Fig. 3 illustrates a trial of the repair planner on the Justin robot. The task is to stack one cup on top of another, but the plan fails during execution. RRT-short planning is unable to find a path from the start configuration (Fig. 3a) to the pre-grasp location for the left-most cup, and reports the milk carton as an obstacle. The obstacle is therefore moved out of the way (Fig. 3b). RRT-short still fails, so the `move_arm_high` action is applied (Fig. 3c), and finally the robot is able to reach the left-hand cup easily and stack it on top of the right-hand cup (Fig. 3d).

3) Scenario 3: Two cups, occupied tray: This scenario is shown in Fig. 4a-c, and is similar to scenario 1, but this time the tray contains an object occupying part of it. During planning the object is not considered, and during execution no path is found to move one of the cups onto the tray. At this point the execution-time repair is invoked and finds the solution of moving the object off the tray making room for the original plan to continue execution.

4) Scenario 4: Two cups, blocked tray: This scenario repeats scenario 3, but this time there is no learned action to move the object blocking the tray (Fig. 4d). The execution monitor re-runs the symbolic-geometric mapping, but there is insufficient space to place both cups, leading to a failure. A planner that can backtrack between geometric and symbolic reasoning, or execution-time symbolic replanning would be needed to solve this scenario.

VII. CONCLUSIONS

We have demonstrated an approach to learning planning operators from programs. We show that with only a small amount of annotation of the programs, along with a learned mapping between geometric states and predicates, we can learn actions that can be applied to object poses different from the ones in the examples. These actions can then be used by a planner to solve novel tasks. We also demonstrate that by using a plan repair algorithm to make execution of the plans more robust we can avoid the need for a full hybrid planner for many problems. The approach is general enough that we have run it on two different manipulation platforms as well as in simulation.

REFERENCES

[1] Stephane Cambon, Rachid Alami, and Fabien Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009.

[2] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, pages 1023–1028, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[3] Rosen Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

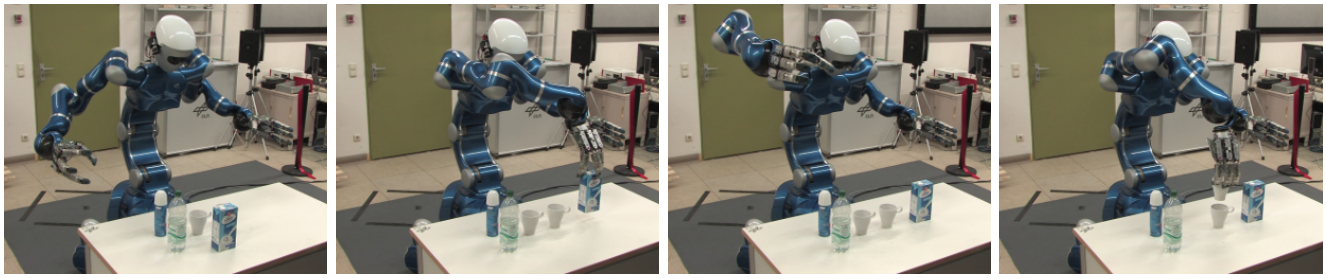
[4] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*, pages 114–121, 2009.

[5] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[6] M. Fuchs, C. Borst, P. R. Giordano, A. Baumann, J. Langwald E. Kramer, R. Gruber, N. Seitz, G. Plank, K. Kunze, F. Schmidt R. Burger, T. Wimboeck, and G. Hirzinger. Rollin' justin - design considerations and realization of a mobile platform for a humanoid upper body. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

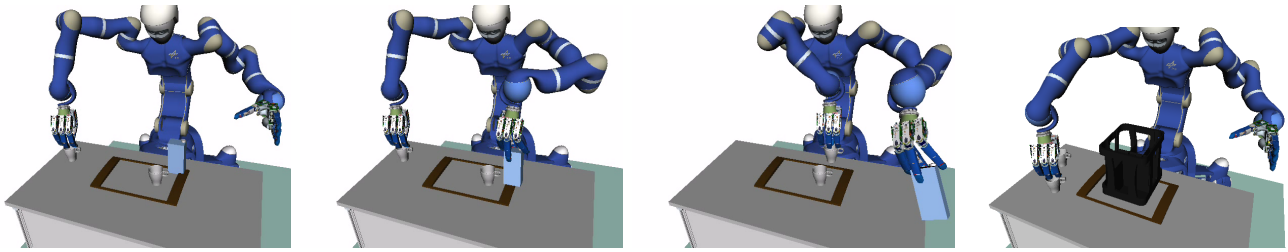
(a) (b) (c) (d)

Fig. 2: Scenario 1: (a) The initial configuration. (b) Ready to grasp a cup. (c) Cup above table. (d) Cup above another cup.



(a) Initial configuration (b) Removing an obstacle (c) A better start pose for planning (d) Stacking the cup

Fig. 3: Scenario 2: Repair plan execution run on Justin, where the aim is to stack the left-most cup on the other cup.



(a) (b) (c) (d)

Fig. 4: Scenarios 3 and (subfigure (d)) 4. In (a), the milk carton blocks placement of the second cup. Therefore the milk carton is removed using the left arm (b) so the cup can be placed (c). In scenario 4 (d), the system fails because the immovable obstacle does not leave space for both cups on the tray.

[7] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[8] R. Jäkel, S.R. Schmidt-Rohr, M. Lösch, A. Kasper, and R. Dillmann. Learning of generalized manipulation strategies in the context of programming by demonstration. In *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, pages 542–547. IEEE, 2010.

[9] Jr James J. Kuffner and Steven M. LaValle. RRT-connect: An efficient approach to single-query path planning. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-00)*, pages 995–1001, 2000.

[10] Vlad I. Morariu, Balaji Vasan Srinivasan, Vikas C. Raykar, Ramani Duraiswami, and Larry S. Davis. Automatic online tuning for fast Gaussian summation. In *Advances in Neural Information Processing Systems (NIPS)*, 2008.

[11] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning probabilistic relational planning rules. In Jana Koehler Shlomo Zilberstein and Sven Koenig, editors, *Proceedings of ICAPS'04*, pages 73–81, Whistler (Canada), June 2004. AAAI Press.

[12] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[13] B. Rosman and S. Ramamoorthy. Learning spatial relationships between objects. *The International Journal of Robotics Research*, 30(11):1328–1342, 2011.

[14] K. Sjoo, A. Aydemir, T. Morwald, K. Zhou, and P. Jensfelt. Mechanical support as a spatial abstraction for mobile robots. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4894–4900. IEEE, 2010.

[15] K. Sjoo and P. Jensfelt. Learning spatial relations from functional simulation. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1513–1519. IEEE, 2011.

[16] Trey Smith and Reid Simmons. Point-based POMDP algorithms: Improved analysis and implementation. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, 2005.

[17] Xuemei Wang. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 335–340, Chicago, IL, June 1994. AAAI Press, CA.

[18] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, 2007.

[19] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence Journal*, 174(18):1540–1569, December 2010.