A Modular Approach to Task and Motion Planning with an Extensible Planner-Independent Interface Layer

Siddharth Srivastava Eugene Fang Lorenzo Riano Rohan Chitnis Stuart Russell Pieter Abbeel

***Abstract*— The need for combined task and motion planning in robotics is well understood. Solving this problem typically requires special purpose implementations of task planning and/or motion planning algorithms. We propose a new approach for task and motion planning using off-the-shelf task and motion planners. This approach makes no assumptions on the implementation of task and motion planners, and only requires that failures in refining a high level action into a motion plan be identifiable and expressible in the form of logical predicates over possibly continuous variables. We evaluate the approach and illustrate its robustness through a number of experiments using a state-of-the-art robotics simulator and a real-world implementation with the PR2 robot. These experiments show the system accomplishing a diverse set of challenging tasks such as picking objects from cluttered environments and laying out a table for dinner.**

I. Introduction

In order to achieve high level goals like laying out a table, robots need to be able to carry out high level task planning in conjunction with low level motion planning. Task planning is needed to determine long term strategies such as whether or not to use a tray to transport multiple objects, and motion planning is required for computing the actual movements that the robot should carry out. However, combining task and motion planners is a hard problem because task planning descriptions typically ignore the geometric preconditions of physical actions. In reality, even high level actions such as picking up an object have continuous arguments and geometric preconditions and effects. As a result, naive approaches for combining off-the-shelf task planners and motion planners result in task plans that have no feasible motion plans.

We present an approach for addressing this fundamental problem using an abstract representation of the true geometric preconditions in high level task descriptions. We illustrate the fundamental problem and our solution using a simple example set in $\mathcal{R}^2$ (Fig. 1(L)). The robot has been deliberately trivialized in this example to focus on the fundamental aspects of the problem. In this problem, a gripper can pick a block if it is in a small region around the block and aligned with one of its sides. It can also place a block that is currently held by moving to a region near the target location and releasing it. The goal is to pick b_1. A discrete planning problem specification for this model would include two actions *pick* and *place*, each of which have simple preconditions and effects: if the gripper is empty and $pick(b_1)$ is applied, the gripper holds b_1; if the gripper is holding b_1 and $place(b_1, S)$ is applied, the gripper no longer holds b_1 and b_1 is placed in the region S. However,

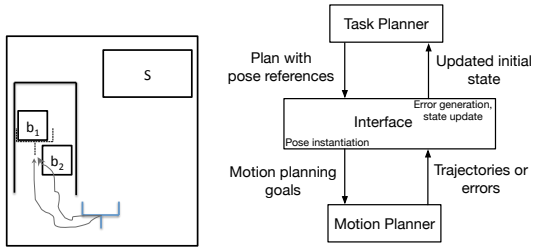


Fig. 1: (L) Running example in $\mathcal{R}^2$: the gripper needs to pick b_1 after moving to the dotted pose (R) Outline of our approach.

this description is clearly inadequate because b_2 obstructs all trajectories to b_1 in the state depicted in Fig. 1, and $pick(b_1)$ cannot be executed.

A true representation for this domain has to take into account the geometric locations of objects and the gripper in $\mathcal{R}^2$. The *pick* action's true arguments are *initPose, targetPose, blockPose, traj* denoting the gripper's initial pose, the target pose where picking should be done, the pose of the block and the trajectory along which the gripper should move to get from *initPose* to *targetPose*. The preconditions for picking b_1 are: gripper and b_1 are in their corresponding poses; $traj$ is a motion plan between *initialPose* and *targetPose*, which is a gripping pose for b_1; and there is no obstruction in *traj*. The last condition is false for all trajectories to grasping poses for b_1 in the state illustrated in Fig. 1.

Task planning domain descriptions require actions with discrete arguments and thus cannot express such preconditions or effects accurately. Discretization is also infeasible for such domains: with 10 sampled points along each axis, 5 objects, and considering only Manhattan paths that don't loop over themselves, we would need to precompute the truth values of close to $50,000$ predicates determining whether or not an object obstructs a particular trajectory between any two poses in the initial state alone. A similar discretization for one arm and the base for a PR2 robot would require the precomputation of close to 10^{11} facts. Even if such precomputation could be carried out, the resulting sizes of problem instances make task planning infeasible.

Our Approach Intuitively, our solution to this problem is to use the true action descriptions, but to replace the continuous variables in action arguments with those that range over symbols that are references to continuous values such as "grasping pose for b_1". The use of pose references results in small, relatively easier task planning problems. This allows us to use an off-the-shelf task planner to obtain a plan of the form "execute pick with a target pose which is a grasping pose for b_1 and has a feasible motion plan",

where "grasping pose for b_1" is a pose name, or reference rather than a value. An instantiation of the pose references in a plan with real values is feasible iff there is an error-free motion plan corresponding to each action in the plan. For each action in the task plan, an interface layer uses an off-the-shelf motion planner to select a feasible instantiation of all the pose references in the plan. This process constitutes the *refining* of a task plan into a motion plan and can fail only if the geometric precondition for an action was actually false when it was attempted. Such preconditions are not limited to the absence of obstructions, and could capture arbitrary physical properties.

If the interface layer is unable to find a feasible instantiation of the pose references in a plan (e.g. picking b_1 in the state shown in Fig. 1) it selects an infeasible instantiation of the pose references, identifies the first action for which this was infeasible, and computes the failed preconditions corresponding to this infeasibility in terms of the pose references, e.g. "b_2 obstructs the grasping pose for b_1 from the gripper." This information is used to correct the high level state where the failed action was attempted. The interface layer then reinvokes the task planner to obtain a plan such as "execute pick with a target pose from where b_2 can be grasped; execute place with a target pose from where releasing b_2 will place it on S; execute pick with a target pose from where b_1 can be grasped." The process of plan refinement and generation then continues with the new plan.

If at any stage the task planning problem turns out to be unsolvable, the interface layer computes the failed preconditions for another instantiation of pose references. This interaction continues until either the problem is solved or a resource limit is reached.

The role of the interface layer in our approach is to take a high level plan and (a) construct a sequence of motion planning goals and incrementally invoke an arbitrary motion planner with those goals, and (b) if a refinement fails, use an arbitrary task planner to efficiently generate plans resolving the infeasibilities identified at the motion planning level. We assume that the reason for infeasibility of refinement can be identified. In this way, our approach can be used with arbitrary task and motion planners and results in a system that easily incorporates advances in both of these active areas of research.

We being with a description of the formal task and motion planning problems (Sec. II), followed by our representational formulation (Sec. III). Sec. IV describes our key algorithms. We describe several experiments conducted using a state of the art robotics simulator and a system validation on the PR2 robot (Sec. V) and conclude with a discussion of related work (Sec. VI).

II. Background

A. Task Planning

We assume full observability and deterministic action effects. The formal language PDDL [2] defines a task planning problem as a tuple $\langle A, s_0, g\rangle$, where A is a set of parameterized propositional actions, s_0 is an initial state of the domain and g, a set of propositions, is the goal condition. For clarity, we will describe both preconditions and effects of actions as conjunctive lists of literals in first-order logic, and use quantifiers as necessary for brevity. The discrete *pick* action could be represented in this representation as follows:

pick(b_1, gripper)
precon *Empty(gripper)*
effect *InGripper(b_1), ¬Empty(gripper)*

Let $a(s)$ denote the result of applying action a on state s, obtained by modifying s to make the positive (negative) literals in the effects of a true (false). A solution to a task planning problem is a sequence of grounded actions $a_0, \ldots, a_n$ such that s_0 satisfies the preconditions of a_0, $s_{i+1} = a_i(s_i)$ satisfies a_{i+1}'s preconditions for $i = 1, \ldots, n-1$ and s_{n+1} satisfies g.

B. Motion Planning

A motion planning problem is defined as a tuple $\langle C, f, p_0, p_t\rangle$, where C is the space of possible configurations or poses of a robot, f is a boolean function that determines whether or not a pose is in collision and $p_0, p_t \in C$ are the initial and final poses. A collision-free solution to a motion planning problem is a trajectory in C from p_0 to p_t such that f doesn't hold for any pose in the trajectory. Motion planning algorithms use a variety of approaches for representing C and f efficiently. A solution that allows collisions only with the movable objects in a given environment may be obtained by invoking a motion planner after modifying f to be false for all collisions with movable obstructions.

III. Abstract Formulation Using Pose References

Although high level specifications like *pick* shown above capture the logical preconditions of physical actions they cannot be used in real pick and place tasks. A true representation of the *pick* action can be written using predicates *IsGP*, *IsMP* and *Obstructs* capturing geometric conditions: *IsGP(p, o)* holds iff p is a pose at which o can be grasped; *IsMP($traj, p_1, p_2$)* holds iff $traj$ is a motion plan from p_1 to p_2; *Obstructs($obj', traj, obj_1$)* holds iff obj' is one of the objects obstructing a pickup of obj_1 along the motion plan $traj$.

pick2D(obj, gripper, $pose_1$, $pose_2$, $traj$)
precon *Empty(gripper), At($gripper, pose_1$), IsGP($pose_2$, obj), IsMP($traj, pose_1, pose_2$), ∀obj′¬ Obstructs(obj′, traj, obj_1)*
effect *In($obj_1, gripper$), ¬Empty(gripper), At($gripper, pose_2$)*

The precondition enforces a collision-free trajectory. Unfortunately the continuous pose variables in this specification lead to an infinite branching factor and cannot be used with an off-the-shelf task planner. As noted in the introduction, discretization would be infeasible in such a domain.

We use an abstract representation where continuous variables are replaced by ones that range over finite sets of symbols that are references to continuous values. The initial state contains a finite set of facts over these references.

Our approach can be used with any approach for generating such an abstraction. In this paper, we use an approach derived from a form of quantifier elimination, as described in recent work [3]. We summarize this technique here using the running example. The pose variables range over pose references of the form $initPose$, gp_obj_i, $pdp\_obj_i\_S$ for each object obj_i. Intuitively these references denote the gripper's initial pose, a grasping pose (gp) for obj_i and a put-down pose (pdp) for placing obj_i in surface S. This is captured by including in the initial state, facts relating the pose references to the properties they are intended to satisfy: *IsGP*(gp_obj_i, obj_i), *IsPDP*($pdp\_obj_i\_S, obj_i, S$), *IsMP*($traj\_pose_1\_pose_2$, $pose_1$, $pose_2$), where $pose_1$ and $pose_2$ range over the introduced pose references. Thus, we use the *pick2D* specification defined above, but with variables ranging over references replacing the continuous variables.

We use this approach to complete our description of the running example's high level domain specification. The preconditions of place2D require two new predicates: *IsPDL*($tloc, S$) indicates that $tloc$ is a put-down location in S and PDObstructs(obj', $traj$, obj, $tloc$) indicates that obj' obstructs the trajectory $traj$ for placing obj at $tloc$.

place2D($obj, gripper, pose_1, pose_2, traj, tloc$)
- *precon* *In*($obj, gripper$), *At*($gripper, pose_1$), *IsPDP*($pose_2, obj, tloc$),*IsMP*($traj, pose_1, pose_2$), *IsPDL*($tloc, S$) $\forall obj' \neg$ *PDObstructs*(obj', $traj$, obj, $tloc$)
- *effect* $\neg$ *In*($obj, gripper$), *At*($obj, tloc$), *Empty*($gripper$), *At*($gripper$, $pose_2$), $\forall obj', traj' \neg$ *Obstructs*(obj, $traj'$, obj')

In this simple example, we assert that once an object is placed in the region S, it does not obstruct any pickup trajectories. Further representational optimization is possible. Some of the introduced references do not contribute any functionality in the high level specification and optimizations are possible to further simplify the high level specification. For instance, the $traj$ argument of *pick2D* doesn't occur in its effects. It is used in *IsMP*, which is not changed by any high level action and in *Obstructs*, which is changed by *place* for all trajectories, so that the effect is independent of $traj$. In other words, high level solutions are not affected by this argument. The domain specification can be optimized to remove such arguments and predicates. High level plans can then be post processed to reintroduce these arguments prior to refinement.

The approach outlined above easily extends to real robots like the PR2. The only difference is that we can now use different geometric conditions to capture base poses for grasping (*IsBPFG*) and gripper poses for grasping (*IsGPFG*), and for put-down (*IsBPFPD, IsGPFPD*). A base pose for grasping is a pose from which there is a collision-free IK solution to a gripper grasping pose if all movable objects are removed. A significant point of difference in this model is that when an object is picked up, it no longer obstructs any trajectories. Further, the predicate *IsLFPD* determines whether or not a location is one where objects can be placed. This can be true of all locations on surfaces that can support objects. Fig. 2 shows a specification of three actions for pick and place tasks, assuming for clarity that base movement is always possible (pr2Move action) and that stacking is not allowed.

pr2Pick(obj_1 *gripper*, $pose_1$, $pose_2$, $traj$)
- *precon* *Empty(gripper), RobotAt*($pose_1$), *IsBPFG*($pose_1$, obj),*IsGPFG*($pose_2$, obj), *IsMP*($traj$, $pose_1$, $pose_2$), $\forall$ obj' $\neg$ *Obstructs*(obj', $traj$, obj_1)
- *effect* *In*($obj_1, gripper$), $\neg$*Empty*(*gripper*), $\forall obj', traj'$ $\neg$*Obstructs*($obj_1, traj', obj'$), $\forall obj', traj', tloc'$ $\neg$*PDObstructs*($obj_1, traj', obj', tloc'$)

pr2PutDown($obj, gripper, pose_1, pose_2, traj, targetLoc$)
- *precon* *In*($obj, gripper$) $\wedge$ *RobotAt*($pose_1$), *IsBPFPD*($pose_1, obj, targetLoc$), *IsGPFPD*($pose_2, obj, targetLoc$) *IsMP*($traj, pose_1, pose_2$), *IsLFPD*($targetLoc, obj$) $\forall obj' \neg$*PDObstructs*(obj', $traj$, obj, $tloc$)
- *effect* $\neg$ *In*($obj, gripper$), *At*($obj, targetLoc$)

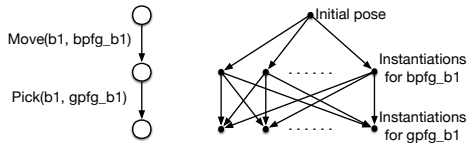
pr2Move($pose_1, pose_2, traj$)
- *precon* *RobotAt*($pose_1$), *IsMP*($traj$, $pose_1$, $pose_2$)
- *effect* $\neg$*RobotAt*($pose_1$), *RobotAt*($pose_2$)

Fig. 2: Action specifications for a robot with an articulated manipulator.

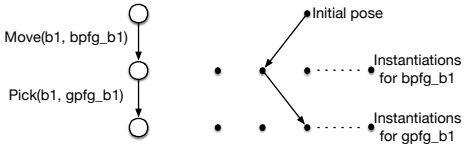
This technique leads to immense efficiency in problem representation compared to discretization (discretized values or sampled poses, which will be used later in the paper are not enumerated at the high level in this formulation). This makes it feasible to use a task planner to solve the high level planning problem. However, the high level descriptions of actions whose true effects depend on reasoning with continuous variables are sound but not complete: the stated effects are guaranteed to take place, but the true effects of an action may include any number of changes on the truth values of predicates whose arguments originally had continuous domains. E.g., it is not possible to specify which trajectories get obstructed as a result of applying a place action with a set of pose references for arguments. This incompleteness is a consequence of using an abstract representation suitable for discrete planners, and our approach is designed to compensate for it.

As noted in Section II-A, the initial state for a planning problem is defined using the ground atoms which are true in the initial state. However, the truth values of ground atoms over references like *Obstructs*($obj_{10}, traj\_pose_1\_pose_2, obj_{17}$) are not known initially. Default truth values are assumed for such atoms in the initial state. Domain-specific choices for these defaults can also be obtained automatically in a manner that facilitates completeness guarantees when possible [3].

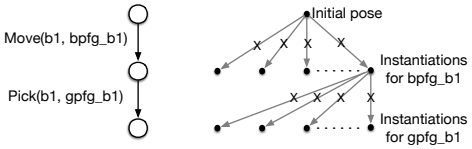
Conditional Costs The PDDL framework also allows actions to be associated with costs. The approach presented above applies to actions whose costs depend on a finite



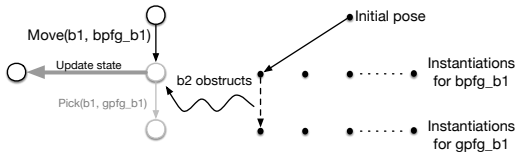
(a) On the left we show a task plan with pose references. On the right we show the search space of possible instantiations of these references. The initial pose has a unique instantiation as that is the robot's current pose. In order to refine the task plan into a motion plan, the interface layer needs to find instantiations for all pose references, such that there is an error-free motion plan between each successive pair of poses.



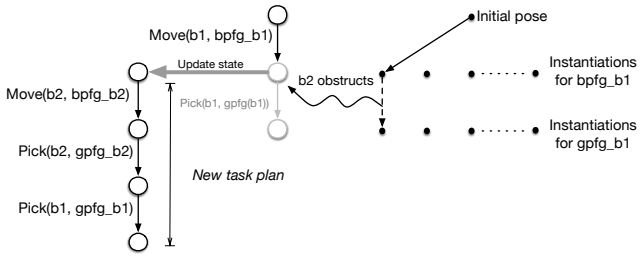
(b) Scenario 1: The interface layer finds a set of pose instantiations for which there is an error-free motion plan. This completes the refinement process.



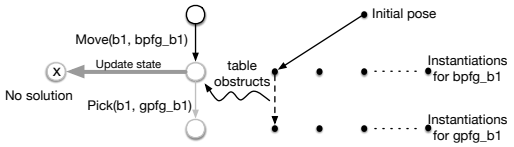
(c) Scenario 2: The interface layer completes backtracking search finds no complete instantiation of poses with an error-free motion plan. The original task plan cannot be refined into a motion plan.



(d) Scenario 2a: The interface layer selects the first possible pose instantiation and the error-free motion plan prefix for this instantiation. It uses the failed preconditions for the action that did not have an error-free motion plan to update the high level state.



(e) Scenario 2b: The interface layer invokes a task planner to compute a new solution plan for the updated state. Refinement continues with the new task plan.



(f) Scenario 3: The first partial motion plan results in an updated state that has no solution at the task planning level. The interface layer continues to select the next available partial plan and invoke the task planner with errors corresponding to it until it obtains one for which the high level state is solvable.

Fig. 3: Illustration of the interface layer's refinement process. Action arguments have been abbreviated.

number of geometric predicates over possibly continuous action arguments. Such an action can be formulated as an action with conditional costs, or as a set of actions with each action having a particular combination of the geometric preconditions and the cost associated with them. The approach outlined above applies seamlessly to such actions in either formulation.

IV. Task and Motion Planning

We illustrate our approach through a detailed example using the specification in Fig. 2. Fig. 3a shows an initial task plan that is obtained using a task planner and the search space for instantiations of the pose references used in it. Scenario 1 captures a situation where the interface layer finds instantiations that correspond to an error-free motion plan, thus solving the problem (Fig. 3b). Scenario 2 captures a situation where the interface layer is unable to find an instantiation of poses corresponding to an error-free motion plan (Fig. 3c). In this case it identifies partial solutions and attempts to extend them using a task planner. In scenario 2a this succeeds with the first partial motion plan (Figs. 3d). The interface layer generates logical facts capturing reasons for the failure of motion planning and uses them to update the high level state where this failure occurred. The high level then produces a new plan to solve the updated state (3e). In scenario 3, the updated state is found to be unsolvable and the interface layer continues to search for a partial motion plan that whose infeasibility corresponds to a solvable task planning problem.

In the remainder of this section we describe two algorithms that constitute the interface layer. Alg. 1 describes the outer loop of refinement and regeneration of high level plans that continues until a resource limit (e.g. time) is reached. Alg. 2 describes the process of refining high level plans into trajectories representing motion plans.

A. Overall Algorithm For the Interface

Alg. 1 proceeds by first invoking a classical planner with the given initial state of the task and motion planning problem to get *HLPlan*. In each iteration of the `while` loop, *TryRefine* (Alg. 2) is first called in line 5 in the error-free mode, which searches for a feasible instantiation of the pose references used in HLPlan. If this fails, the second call to *TryRefine* (line 8) is made in the partial trajectory

Algorithm 1: TaskAndMotionPlanningInterface

```
   Input: State, InitialPose, HLPlan, Step, PartialTraj
1  if HLPlan not created then
2  |  HLPlan ← callClassicalPlanner(State)
3  |  Step ← 0; PartialTraj ← None; StartPose ← InitialPose
   end
4  while resource limit not reached do
5  |  if TryRefine(StartPose, HLPlan, Step, PartialTraj,
   |  ErrFreeMode) succeeds then
   |  else
6  |  |
   |  end
   |  return refinement
7  |  repeat
8  |  |  (PartialTraj, EndPose, FailStep, FailCause)
   |  |        ← TryRefine(StartPose, HLPlan, Step,
   |  |              PartialTraj, PartialTrajMode)
9  |  |  State ← stateUpdate(State, FailCause, FailStep)
10 |  |  NewPlan ← callClassicalPlanner(State)
11 |  |  if NewPlan was obtained then
12 |  |  |  HLPlan ← HLPlan[0:FailStep] + NewPlan
13 |  |  |  StartPose ← EndPose; Step ← FailStep
   |  |  end
   |  until NewPlan obtained or MaxTrajCount reached
14 |  if MaxTrajCount reached then
15 |  |  Clear all learned facts from initial state
16 |  |  Reset PoseGenerator with new random seed
17 |  |  Reset Step, PartialTraj, StartPose to initial values
   |  end
   end
```

mode. In this mode, every invocation of *TryRefine* yields: (a) *PartialTraj*, a partial trajectory constituting an error-free refinement of a prefix of *HLPlan*, (b) *EndPose*, denoting the pose where this trajectory ends (c) *FailStep*, denoting the first action in HLPlan that could not be refined into a motion plan, and (d) *FailCause*, the failed preconditions of this action. *FailStep* and *FailCause* are used to update the high level state by applying the effects of actions in *HLPlan* until *FailStep* on the state for which HLPlan was obtained. If *FailStep* is the first step in the plan this leads to a modification of the initial state. In line 10 an arbitrary task planner is invoked with this new state. If the planner determines that this state is unsolvable, execution continues into the next iteration where *TryRefine* returns the next partial trajectory for the same *HLPlan* and the errors corresponding to it. If on the other hand the state was solvable and *NewPlan* was obtained execution continues with an invocation of *TryRefine* with the updated plan in error-free mode (line 11). If an upper limit on the number of attempted refinements for *HLPlan* is reached (line 14) the refinement process starts over from the first action in the available plan after resetting the *PoseGenerator* used by *TryRefine* to instantiate pose references, and removing facts corresponding to the old pose reference instantiations.

B. Refining Task Plans into Motion Plans

We assume wlog that all HLPlans are zero-indexed lists, and start with an initial action that has no effect.

1) The TryRefine Algorithm: TryRefine (Alg. 2) implements a backtracking search over sets of possible instantiations for each pose reference used in the high level plan. Starting with the input *InitialPose*, in each iteration of the loop, *TryRefine* invokes *PoseGenerator* to get a possible target pose for the next action. This is described in the next section. ActionNum maintains the current action index, for which a motion plan has been found. The bookkeeping function TargetPose() maps an action to the target pose currently being considered for that action. We first discuss lines 11-15. As long as another target pose for the next action is available, an arbitrary motion planner is called with it in line 11. If motion planning succeeds in the error-free mode, the iteration proceeds to the action after next. If not, then the algorithm proceeds differently in error-free and partial-trajectory modes. In error-free mode it simply obtains the another target pose for the next action. In partial trajectory mode it yields the reasons for motion planning failure (14). The `yield` keyword fixes the algorithm's flow of control so that the next invocation of TryRefine resumes from the statement after `yield`. The failure can result from (a) obstructions in motion planning, which are obtained as described in section II-B and (b) general geometric preconditions of actions, e.g. stackability constraints for trays, which are determined by dedicated modules or physics simulators. These errors are converted into logical facts in terms of pose references (independent of geometric values) and returned via the yield statement.

Lines 7-10 capture backtracking, which is carried out when the pose generator has exhausted all possible instantiations of the next action's references.

These algorithms can be optimized further. One optimization we use in practice is to only try motion planning in the error-free mode if an IK solution exists.

2) Pose Generators: The *PoseGenerator* for an action is intended to iterate over those values for pose references which satisfy the plan-independent geometric preconditions of that action. These are the geometric preconditions that are not affected by actions, such as *IsBPFG, IsGPFG* etc. In practice, the PoseGenerator iterates over a set of finite but randomly sampled values that are only likely to satisfy these properties. The random seed for generating these values is reset when *MaxTrajCount* is reached in Alg. 1.

More specifically, *PoseGenerator* generates (a) an instantiation of the pose references used in the action's arguments and (b) a target pose corresponding to each such instantiation. We also allow the pose generator to generate a tuple of target poses (waypoints) if needed, for multi-trajectory actions. The *GetMotionPlan* call in *TryRefine* succeeds for such a tuple of waypoints only if it can find a motion plan between each successive pair of waypoints. We discuss a few specific examples of pose generators below.

PoseGenerator for *pr2Pickup* The pickup pose generator instantiates the pose references $bpfg\_obj_i$ and $gpfg\_obj_i$, which need to satisfy the geometric properties *IsBPFG* and *IsGPFG*. For $bpfg\_obj_i$ it samples base poses oriented towards obj_i in an annulus around the object. For $gpfg\_obj_i$,

Algorithm 2: TryRefine

```
   Input: InitialPose, HLPlan, Step, TrajPrefix, Mode
1  ActionNum ← Step − 1; Traj ← TrajPrefix
2  Initialize target pose generators
3  Target pose list for HLPlan[ActionNum] ← {InitialPose}
4  while Step − 1 ≤ ActionNum ≤ length(HLPlan) do
5      axn ← HLPlan[ActionNum]
6      nextAxn ← HLPlan[ActionNum+1]
7      if TargetPose(nextAxn) is defined then
           /* backtrack                                   */
8          TargetPose(nextAxn)
               ← PoseGenerator(nextAxn).resetAndGetFirst()
9          TargetPose(axn)
               ←PoseGenerator(axn).getNext()
10         ActionNum−−; Traj ← Traj.delSuffixFor(axn)
       else
11         if GetMotionPlan(TargetPose(axn),
           TargetPose(nextAxn)) succeeds then
12             Traj ← Traj + ComputedPath; ActionNum++
           else
13             if Mode = PartialTrajMode then
14                 yield (TargetPose(axn), Traj, ActionNum+1,
                       GetMPErrors(TargetPose(axn),
                                   TargetPose(nextAxn)))
               end
15             TargetPose(nextAxn)
                   ← PoseGenerator(nextAxn).getNext()
           end
       end
16     if ActionNum = length(HLPlan)+1 then
17         return Traj
       end
   end
```

we need to generate poses from where closing the gripper will result in a stable grasp of the object. We assume that such poses are known for each object class (e.g. bowl, can, tray etc.). Computing such poses, or regions in the object's frame where they occur, is an independent problem and can be solved using state-of-the-art approaches for grasping. We used an approach where every grasp pose corresponds to a pregrasp pose and a raise pose that the gripper must move to, after it closes around the object. The pose generator is responsible for generating possible values for all of the intermediate poses as waypoints.

PoseGenerator for *pr2PutDown* The pose generator for put-down generates values for pose references of the form $tloc$ (a possible pose reference for $targetLoc$), $bpfpd\_obj_i\_tloc$, and $gpfpd\_obj_i\_tloc$ to satisfy the properties *IsBPFPD* and *IsGPFPD*. $tloc$ values are sampled locations on supporting surfaces within a certain radius of the current base pose; values for $bpfpd\_obj_i\_tloc$ are obtained by sampling base poses in an annulus around $tloc$ and oriented towards it. Gripper put-down poses of the form $gpfpd\_obj_i\_tloc$ are sampled by computing possible grasping poses assuming the object was at $tloc$.

C. Completeness

We now present a sufficient condition under which our approach is guaranteed to terminate and find a solution if one exists. This result follows directly from prior work [3] which stated the result for a non-backtracking version of Alg. 2 that carried out an online execution. In comparison, the algorithms presented here are more general.

Definition 1: A set of actions is *uniform* wrt a goal g and a set of predicates R if for every $r \in R$,

1) Occurrences of r in action preconditions and goal are either always positive, or always negative
2) Actions can only add atoms using r in the form (positive or negative) used in preconditions and the goal g

Theorem 1: Let $P = \langle A, s_0, g \rangle$ be a planning problem that does not have dead-end states and A a set of actions that is uniform wrt the g and the geometric predicates used in the domain. Let G be the pose generator for the pose references used in s_0.

If the combination of G and the motion planner is probabilistically complete (has a non-zero probability of finding a feasible pose instantiation when there exists one), then Alg. 1 will eventually terminate with a task and motion plan solving P if there exists one.

Intuitively, termination is guaranteed because under the premises, every time a state update takes place, missing geometric facts are added to the state. Under the uniform condition, these facts can only be removed by actions but not added again. A monotonicity argument leads to the termination result. We refer the reader to [3] for details of this proof. We note that these sufficient conditions are not necessary: our empirical evaluation shows the algorithm succeeding in a number of tasks that do not satisfy these conditions.

The uniform property used in this result plays a role similar to *simple domains* defined by Bonet and Geffner [4]. Neither categorization is more general than the other. In contrast to simple domains, uniformity is less restrictive in not requiring invariants over initially unknown facts, while simple domains are less restrictive in not enforcing all occurrences of unknown predicates to be of the same polarity.

V. Empirical Evaluation

[[SS:*Make source code available*]]

We implemented the proposed approach using the OpenRAVE simulator [1]. We evaluated our system with the PR2 robot in three different domains. In all of our exeriments we used Trajopt (multi-init mode), which is a state-of-the-art motion planner that uses sequential convex optimization to compute collision avoiding trajectories [5]. For every motion planning query, Trajop returns a trajectory with a cost. A wrapper script determined collisions (if any) along the returned trajectory. We used two task planners, FF [6] and the IPC 2011 version of FD [7] in `seq-opt-lmcut` mode, which makes it a cost-optimal planner. FD was not appropriate for the first two tasks described below since they used negative preconditions and FD has known performance issues with negative preconditions. Domain compilations for eliminating negative preconditions are possible but lead to hundreds of facts in the initial problem specifications, adversely affecting planner performance. Since our system

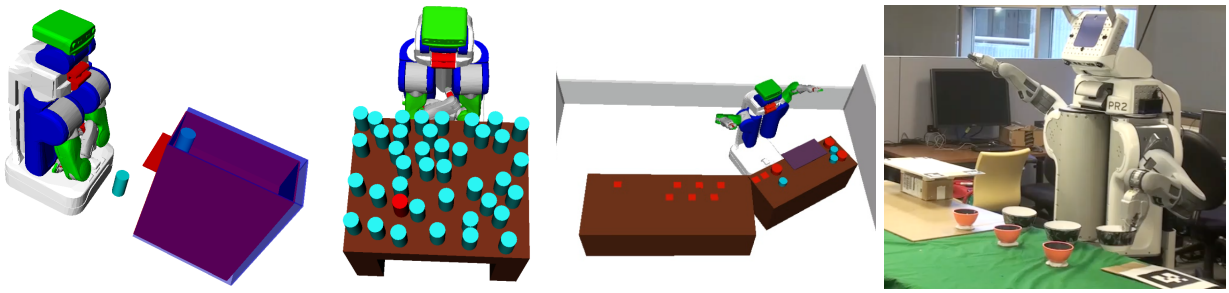


Fig. 4: Test domains from L to R: random initial states for the drawer world, cluttered table with 40 objects where the red object denotes the target object, and dinner layout. The rightmost image shows the real PR2 completing the dinner layout.

can work with any classical planner, we used FF for tasks where costs were not a concern. All the problems used an ambidexterous version of the PR2 actions shown in Fig. 2 with task-specific actions such as placing items on a tray and opening a drawer.

A. Object in a Drawer

In this domain the robot needs to open a drawer that has an object inside it. An object in front of the drawer may prevent a complete opening of the drawer. Typically opening the drawer would be modeled as a single high level action. However, the geometry of object placements determines (a) where the robot should place itself to open the drawer while avoiding collisions with the outer object and the open drawer and (b) whether the inner object is deep enough to make plans that don't move the outer object infeasible. Since the position for placing the outer object has to be chosen and can add obstruction facts, this domain does not satisfy the uniform condition required in Thm. 1.

We modeled this domain by including an open-drawer action, which takes as its argument the current robot pose and the amount by which to pull the drawer. The pose generator for this action generates random bounded values for the pull-distance. We experimented with various random placements of objects in this domain. The system displayed interesting behavior, where it would choose to position the robot so as to open the drawer minimally and access the inner object when possible. If necessary, it would pick up and replace the outer object in a position that it determined, through the search for possible instantiations of put down locations, was going to be collision free when the robot opened the drawer.

B. Cluttered Table

In this task, the robot needs to pick up a target object from a cluttered table. There is no designated free space for placing objects, so the planning process also needs to find places to put down the removed obstructions. In order to make the problem more challenging, we restricted pickups to only use side-grasps. As a result, the robot's thick grippers create several obstructions. Further, many of the pose instantiations lead to cyclic obstructions, which make the corresponding task planning problems unsolvable. As a result, this task does not satisfy the premises of Thm. 1.

C. Laying Out a Table for Dinner

The goal of this task is to lay out a dinner table with some items initially on a different table. A tray is available, but not necessary for the task. We modeled a scenario where the initial location of objects was far from the target location by asserting that these locations were in different rooms and associating a high cost to all moves across rooms. Thus, a cost-optimal plan for this task will opt to use the tray if more than two objects need to be transported. The geometric properties in this domain were stackability and relative positions of objects (see below) rather than obstructions. Stackability was determined by a module in the interface layer, using object diameters. The test scenarios consisted of varying numbers of pairs of two classes of objects (cups and bowls) placed at random locations on the table. Objects had random names to prevent the task planner from favoring any particular stacking order. The initial task planner specification allowed all objects to be stacked on each other. Optimal task planning is hard in this domain, as the number of reachable states exceeds 2.5 million with just six objects.

We used FD as the planner for this task, since plan cost was an important consideration. The output plans appropriately used the tray to transport items. However, we observed several inefficient movements when the robot picked objects on its left with its right hand (and vice versa) as the task planner had no way of determining relative positions and chose hands arbitrarily. We made two modifications to address this, both of which increase the complexity of the task planning problem by increasing its branching factor. We used the conditional cost formulation (Sec. III) to penalize actions which accessed an object or a location on the right (left) with the left (right) hand. We used geometric predicates *OnLeft(pose)* and *OnRight(pose)* to model the conditional costs. We also added a *Handoff* action in the domain, which allowed the robot to transfer objects from one hand to the other. The resulting behavior, though not guaranteed to be optimal, showed the system intelligently determining which hand to use for a particular grasp or putdown and whether or not a handoff should be done.

D. Real World Validation

For the real world experiments we used ROS packages for detecting object and table poses (`ar_track_alvar`) and for SLAM (`hector_slam`). A video of the PR2 laying out the table using this system is available at [[SS:*video*]].

VI. Related Work

[[SS:*workshop paper*]] This work is related to, and builds upon a vast history of research in planning and robotics. The field of discrete planning has made several advances in scope and scalability, mainly through the development of efficient methods for computing heuristic functions automatically from a domain definition [8], [6], [9]. Various researchers have investigated the problem of combining task and motion planning. However, to the best of our knowledge, very few approaches address it by combining off-the-shelf task and motion planners; the only existing approaches that utilize off-the-shelf task planners use task plans minimally, typically as one of the inputs in a heuristic function for guiding search in a configuration space. These approaches do not address the fundamental problems of the task planning description being incomplete and do not attempt to (a) represent geometric information in a form that task planners can use or (b) correct the task planner's representation with information gained through geometric reasoning. In contrast, our approach attempts to refine task plans into motion planning solutions and to provide the task planner with corrected geometric information when necessary.

Alami et al. [10] describe a system architecture that uses a translation module for converting high-level actions into continuous trajectories. Volpe et al. [11] also use a similar module, referred to as "time-line" interaction. However, these approaches do not discuss specific methods for compiling away continuous parameters of operators in order to facilitate task planning. Cambon et al. [12] propose an approach that bears similarity to ours in its use of location references. However, their approach requires the motion planner to use probabilistic roadmaps (PRMs) [13] and requires one roadmap per movable object. The role of the task plan is also limited in this approach: only its length is used, as one of the inputs in a heuristic function for search. However, their approach inherits the PRM guarantee of probabilistic completeness. Plaku et al. [14] develop a more efficient approach for incorporating dynamic feasibility and also allow non-deterministic actions. They also make minimal use of the task plan: only the first action is used to bias a low-level search process. Hauser [15] observes almost the same problems that motivated our work, in tasks like pick-and-place, but does not address the problem of combining off-the-shelf task and motion planners. Wolfe et al. [16] use angelic hierarchical planning to define a hierarchy of high-level actions over primitive actions. Our approach could also be viewed as using an angelic interpretation: the pose references in the high level specification are assumed to have *a* value that satisfies the preconditions, and the interface layer attempts to find such values. Kaelbling et al. [17] also combine task and motion planning using an input hierarchy and a specifically designed regression-based planner that is equipped to be able to utilize task-specific reasoning components and regress useful geometric properties.

Grasping objects in a cluttered environment is still an open problem in robotics. Dogar et al. [18] present an approach for replacing pick-and-place in cluttered environments with push-grasps. This is an interesting approach for dealing with obstructions while grasping, and would be promising as a primitive action in our overall approach. Approaches have also been developed for motion planning in the presence of movable objects (e.g. [19]), but they do not address the general problem of combining task and motion planning.

The notion of reinvoking task planners is related to replanning for partially observable or non-deterministic environments [20], [21]. However, the focus of this paper is on the substantially different problem of providing the task planner with information gained through geometric reasoning. An alternate representation of the problem of dealing with large numbers of relevant facts in the inital state would be to treat them as initially unknown and use a partially observable formulation with non-deterministic "sensing" actions [22]. However, this is a much harder problem in general [23], and contingent solutions typically do not exist for all possible combinations of truth values of geometric predicates. Approaches for planning modulo theories (PMT) [24] and planning with semantic attachments [25] address related problems high level planning in hybrid domains. However, these approaches use an extended planning language and require appropriately extended planners.

VII. Conclusions

We presented a novel approach for combined task and motion planning that is able to solve non-trivial robot planning problems without using task-specific heuristics or any hierarchical knowledge beyond the primitive PDDL actions. Our system works with off-the-shelf task and motion planners, and will therefore scale automatically with advances in either field. It supports actions with geometric preconditions without making assumptions about the implementation of the task planner. We also presented a sufficient, but not necessary condition for completeness. We demonstrated that our system works in several non-trivial, randomly generated tasks where this condition is not met and also validated it in the real world with a PR2 robot.

Acknowledgments

We thank Malte Helmert and Joerg Hoffmann for providing versions of their planners with verbose outputs that were helpful in early versions of our implementation. We also thank Ankush Gupta for help in working with the real PR2 and John Schulman for help in setting up Trajopt. This work was supported by the NSF under Grant IIS-0904672.

References

[1] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010.

[2] M. Fox and D. Long, "PDDL2.1: an extension to PDDL for expressing temporal planning domains," *J. Artif. Int. Res.*, vol. 20, no. 1, pp. 61–124, 2003.

[3] S. Srivastava, L. Riano, S. Russell, and P. Abbeel, "Using classical planners for tasks with continuous operators in robotics," in *ICAPS Workshop on Planning and Robotics*, 2013.

[4] B. Bonet and H. Geffner, "Planning under partial observability by classical replanning: Theory and experiments," in *IJCAI*, 2011, pp. 1936–1941.

[5] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization," in *Proceedings of Robotics: Science and Systems*, 2013.

[6] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[7] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[8] B. Bonet and H. Geffner, "Planning as heuristic search," *Artif. Intell.*, vol. 129, no. 1-2, pp. 5–33, 2001.

[9] M. Helmert, P. Haslum, and J. Hoffmann, "Flexible abstraction heuristics for optimal sequential planning," in *Proc. of the 17th International Conference on Automated Planning and Scheduling*, 2007, pp. 176–183.

[10] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal Of Robotics Research*, vol. 17, pp. 315–337, 1998.

[11] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *Proc. of IEEE Aerospace Conference*, 2001, pp. 121–132.

[12] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *The International Journal of Robotics Research*, vol. 28, no. 1, pp. 104–126, 2009.

[13] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, 1996.

[14] E. Plaku and G. D. Hager, "Sampling-based motion and symbolic action planning with geometric and differential constraints," in *IEEE International Conference on Robotics and Automation*, Anchorage, AK, 2010, pp. 5002–5008.

[15] K. Hauser, "Task planning with continuous actions and nondeterministic motion planning queries," in *Proc. of AAAI Workshop on Bridging the Gap between Task and Motion Planning*, 2010.

[16] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *Proc. of the International Conference on Automated Planning and Scheduling*, 2010, pp. 254–258.

[17] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *Proc. IEEE International Conference on Robotics and Automation*, 2011, pp. 1470–1477.

[18] M. Dogar and S. Srinivasa, "A framework for push-grasping in clutter," *Robotics: Science and Systems VII*, 2011.

[19] M. Levihn, J. Scholz, and M. Stilman, "Hierarchical decision theoretic planning for navigation among movable obstacles," in *Workshop on the Algorithmic Foundations of Robotics*, June 2012, pp. 19–35.

[20] K. Talamadupula, J. Benton, P. W. Schermerhorn, S. Kambhampati, and M. Scheutz, "Integrating a closed world planner with an open world robot: A case study," in *Proc. of AAAI*, 2010.

[21] S. W. Yoon, A. Fern, and R. Givan, "FF-replan: A baseline for probabilistic planning," in *Proc. of the International Conference on Automated Planning and Scheduling*, 2007, pp. 352–.

[22] B. Bonet and H. Geffner, "Planning with incomplete information as heuristic search in belief space," in *Proc. of the 6th International Conference on Artificial Intelligence Planning and Scheduling*, 2000, pp. 52–61.

[23] J. Rintanen, "Complexity of planning with partial observability," in *Proc. of the 14th International Conference on Automated Planning and Scheduling*, 2004, pp. 345–354.

[24] P. Gregory, D. Long, M. Fox, and J. C. Beck, "Planning modulo theories: Extending the planning paradigm," in *ICAPS*, 2012.

[25] A. Hertle, C. Dornhege, T. Keller, and B. Nebel, "Planning with semantic attachments: An object-oriented view," in *ECAI*, 2012, pp. 402–407.