Combined Task and Motion Planning Through an Extensible Planner-Independent Interface Layer (Full Version)

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. Solutions to this problem have typically relied on special purpose, integrated implementations of task planning and motion planning algorithms. We propose a new approach that uses off-the-shelf task and motion planners and makes no assumptions about their implementation. Doing so enables our approach to directly build on, and benefit from, the vast literature and latest advances in task planning and motion planning. It uses a novel representational abstraction and requires only that failures in computing a motion plan for a high-level action be identifiable and expressible in the form of logical predicates at the task level. We evaluate the approach and illustrate its robustness through a number of experiments using a state-of-the-art robotics simulator and a PR2 robot. These experiments show the system accomplishing a diverse set of challenging tasks such as taking advantage of a tray when laying out a table for dinner and picking objects from cluttered environments where other objects need to be re-arranged before the target object can be reached.**

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 simple high-level actions such as picking up an object have continuous arguments, geometric preconditions and effects. As a result, combining task and motion planning by simply first generating a task plan and then relying on a motion planner to refine that task plan into a sequence of motions will often not work as there might not exist a feasible motion plan for the task plan.

The main contribution of this paper is an approach that provides an interface between task and motion planning (with fairly minimal assumptions on each planning layer), such that the task planner can effectively operate in an abstracted state space that ignores geometry. Geometric constraints discovered by the low-level motion planner are communicated to the task planner through our interface layer in the form of discrete predicates.

While our approach is generally applicable for interfacing task planners and motion planners, for concreteness, we introduce the main ideas through a tiny example in $\mathbb{R}^2$ (Fig. 2). In this problem, a gripper can pick up a block if it is in a small region around the block and aligned with one of its sides; it can place a block that it is currently holding by moving to a target location and releasing it. The goal is for block b_1 to end up in region S.

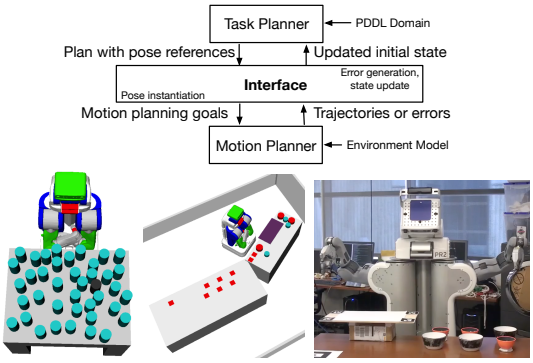


Fig. 1: Top: Outline of our approach. Bottom: Example test scenarios—(L) A cluttered table where the dark object has to be picked (there is no designated free space); (M) A dinner layout task where a tray is available but not necessary for transportation; (R) The PR2 starting a dinner layout.

A purely discrete planning problem specification for this model would include two actions *pick* and *place*, each of which has 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, this description is clearly inadequate because b_2 obstructs all trajectories to b_1 in the state depicted in Fig. 2, and $pick(b_1)$ cannot be executed.

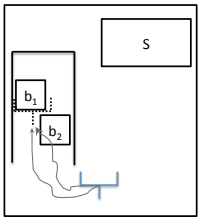


Fig. 2: Running example in $\mathcal{R}^2$: the gripper needs to pick b_1 after moving to the dotted pose.

An accurate representation for this domain has to take into account the geometric locations of objects and the gripper. 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 is at *initPose*; b_1 is at *blockPose*; *targetPose* is a valid gripping pose for b_1 in *blockPose*; $traj$ is a trajectory that starts at *initPose* and ends at *targetPose*; there is no obstruction in *traj*.

Task planning domain descriptions require actions with

discrete arguments and thus cannot directly handle this new representation. Using discretization to explicitly consider the continuous variables in the high-level task planner is impractical, as even crude discretized approximations of the domains of the continuous variables quickly lead to computationally impractical problems. [1]

The main idea behind our approach is to enable using the more accurate descriptions by replacing the continuous-valued arguments with symbolic references to functionally defined values—such as "a grasping pose for b_1." We can then continue to use an off-the-shelf task planner to produce plans of the form: "execute pick with a target pose which is a grasping pose for b_1 and has a feasible motion plan." Once the task plan has been generated, each of the symbols used in such plans needs to be refined into numbers (coordinates in this case). The interface layer sets possible values for them and asks the motion planner to compute a plan corresponding to them (this constitutes the plan *refinement* process); if it succeeds we are in the simplest case and the problem is solved. Often though, no such numbers will exist—i.e., the refinement will fail. In our running example, since b_2 is obstructing the gripper's path, a task plan of picking b_1 followed by placing it in the region S cannot be refined into a feasible motion plan. This will manifest as follows: the interface layer will run a search over possible instantiations of the continuous variables in the task plan. For each possible instantiation, it will ask the motion planner if a low-level plan exists. The motion planner would report that it didn't find a feasible plan for any of the instantiations. At this point the interface layer will pick an instantiation and extract from the motion planner's output what the obstructing objects are, and will update the task level state to include this information. In this particular case, it would add "b_2 obstructs all trajectories to the grasping pose for b_1." Now the task planner generates a new plan: "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." For this plan the interface layer will find an instantiation of the continuous variables for which the motion planner can find a feasible motion plan and we are done. Although our description of this example, followed a particular execution, the key point is that our approach effectively specifies a search problem and then runs a search over this space. In general more backtracking, and more communication through the interface layer is likely to happen.

An important observation, and, indeed, key challenge, is that the high-level task planner cannot compute or reason about the truth values of predicates involving the symbolic references to continuous variables. To handle this issue our approach initializes the truth values of such predicates to a set of default values.

At any stage, the refinement process can fail only if the geometric precondition for an action was actually false when it was attempted. Such preconditions often concern the absence of obstructions, but may also refer to torque limits, stability properties of assemblies, etc. This is a natural consequence of using a representation suitable for task planners, and our approach is designed to handle it: truth values of geometric properties are updated if needed, during the plan refinement and generation process.

In the next sections we formalize our algorithm and describe experiments on a number of tasks, including laying out a dinner table, which has millions of discrete states and picking objects while dealing with the replacement of several obstructions on a tightly cluttered table. Videos of all the experiments are available at:
`http://www.cs.berkeley.edu/~siddharth/icra14`.

II. Background

A. Task Planning

The formal language PDDL [1] defines a fully observable, deterministic 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, using quantifiers for brevity. The discrete *pick* action could be represented as follows:

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

A sequence of actions $a_0, \ldots, a_n$ executed beginning in s_0 generates a state sequence $s_1, \ldots, s_{n+1}$ where $s_{i+1} = a_i(s_i)$ is the result of executing a_i in s_i. The action sequence is a *solution* if s_i satisfies the preconditions of a_i for all i and s_{n+1} satisfies g.

B. Motion Planning

A motion planning problem is 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 motion plan* solving 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. Throughout this paper, we will use the term *motion plan* to denote a trajectory that may include collisions.

III. Abstract Formulation Using Pose References

Although high-level specifications like *pick* above capture the logical preconditions of physical actions they cannot be

[1] For example, in just a 2D world 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 $50,000$ "obstructs" predicates just for the initial state. A similar discretization for one arm and the base for a PR2 robot would require $\sim 10^{11}$ facts. Even if such precomputation is feasible, the resulting problems instances are too large for task planners to solve.

used in real pick and place tasks. A more complete 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 $traj$.

pick2D(obj, gripper, pose$_1$*, pose*$_2$*, traj)*
precon *Empty(gripper), At(*$gripper, pose_1$*),*
IsGP($pose_2$*, obj), IsMP(*$traj, pose_1, pose_2$*),*
$\forall obj' \neg$ *Obstructs(*obj'*, traj,* obj_1*)*
effect *In(*$obj_1, gripper$*),* $\neg$*Empty(gripper),*
At($gripper, pose_2$*)*

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 impractical.

We propose an abstract representation where continuous variables are replaced by ones that range over finite sets of symbols that are references to continuous values. These references are derived from a form quantifier elimination; we refer the reader to the appendix for details A. The initial state contains a finite set of facts linking the references to plan-independent geometric properties they have to satisfy. Continuing with the example, pose variables range over pose references such as $initPose$, 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. For these references, the initial state includes facts: *at(*$gripper, initPose$*), 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. The task planner can now use the *pick2D* specification defined above, but with variables ranging over references replacing the continuous variables. This leads to immense efficiency in problem representation (discretized values or sampled poses, which will be used later in the paper are not enumerated at the high-level in this formulation) and makes task planning practical.

Returning to the 2D example, 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 $tloc$. In this simple example, we assert that once an object is placed in the region S, it does not obstruct any pickup trajectories.

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'*)*

Further representational optimization is possible by removing action arguments and that do not contribute any functionality to the high-level specification. Such arguments can be reintroduced in task plans prior to refinement.

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. 3: Action specifications for robots with articulated manipulators.

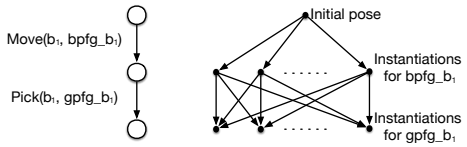
Our approach easily extends to real robots like the PR2 (e.g., Fig. 3). We can use different geometric conditions to capture base poses for grasping (*IsBPFG*) and gripper poses for grasping (*IsGPFG*), and for put-down (*IsBPFPD, IsGPFPD*). A base pose for grasping is a pose from which there is a collision-free 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.

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. We initialize such atoms in the initial state. Domain-specific initializations can also be generated automatically to facilitate completeness guarantees B.

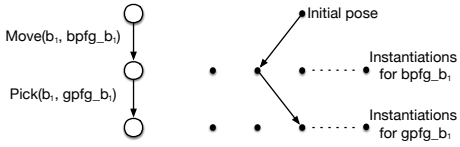
Conditional Costs The PDDL framework also allows actions to be associated with costs. The approach presented above applies to actions whose costs depend on a nite number of geometric predicates over possibly continuous action arguments. Such an action can be formulated as an action with conditional costs, 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

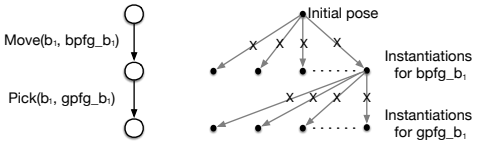
We illustrate our approach through a detailed example using the specification in Fig. 3. Fig. 4a shows an initial task plan obtained using a task planner and the search space for instantiations of the pose references used in it. In scenario 1 the interface layer finds instantiations that correspond to an error-free motion plan, thus solving the problem (Fig. 4b). In scenario 2 the interface layer is unable to find such an



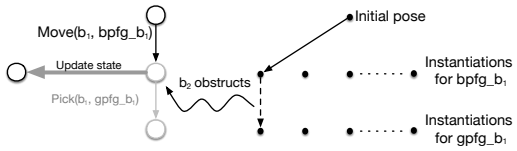
(a) On the left we show a task plan with pose references. On the right we show the search space of possible instantiations of these references. The initial pose has a unique instantiation as that is the robot's current pose. In order to refine the task plan into a motion plan, the interface layer needs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



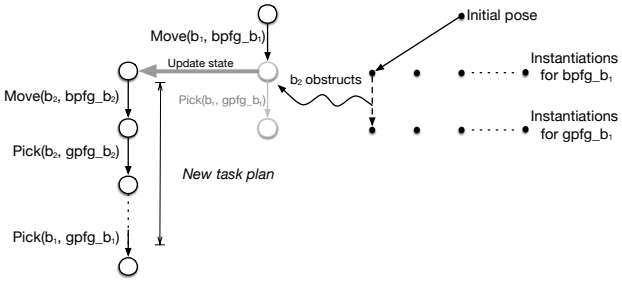
(b) Scenario 1: The interface layer finds a set of pose instantiations for which there is an error-free motion plan. This completes the refinement process.



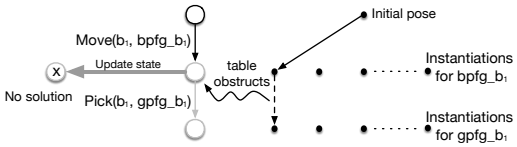
(c) Scenario 2: The interface layer completes backtracking search finds no complete instantiation of poses with an error-free motion plan. The original task plan cannot be refined into a motion plan.



(d) Scenario 2a: The interface layer selects the first possible pose instantiation and the error-free motion plan for the first step. It uses the failed preconditions for the second action 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 2c: The first partial motion plan results in an updated state that has no solution at the task planning level. The interface layer will continue 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. It will then proceed as in Scenario 2b.

Fig. 4: Illustration of the interface layer's refinement process. Action arguments have been abbreviated.

instantiation (Fig. 4c). It goes on to identify partial solutions and attempts to extend them using a task planner. In scenario 2a this succeeds with the first partial motion plan (Figs. 4d). The interface layer generates logical facts capturing reasons for the failure and updates the high-level state where this failure occurred. The task planner then produces a new plan to solve the updated state (4e). In scenario 2c, 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.

We now describe two algorithms that constitute the interface layer. Alg. 1 describes the outer loop of refinement and regeneration of task plans that continues until a resource limit (e.g. time) is reached. Alg. 2 describes the process of refining task 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 6 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 mode. In this mode, every invocation of *TryRefine* yields: (a) a partial trajectory constituting an error-free refinement of a prefix of *HLPlan*, (b) the pose where this trajectory ends (c) the first action in *HLPlan* that could not be refined into a motion plan, and (d) 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 a task planner is invoked with this new state. If 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.

Algorithm 1: Task and Motion Planning Algorithm

```
   Input: State, InitialPose
1  if HLPlan not created then
2      HLPlan ← callClassicalPlanner(State)
3      step ← 1; partialTraj ← None; pose_1 ← InitialPose
4  while resource limit not reached do
5      if TryRefine(pose_1, HLPlan, step, partialTraj,
       ErrFreeMode) succeeds then
6          return refinement
7      repeat
8          (partialTraj, pose_2, failStep, failCause)
                 ← TryRefine(pose_1, 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             pose_1 ← pose_2; step ← failStep
       until NewPlan obtained or MaxTrajCount reached
14     if MaxTrajCount reached then
15         Clear all learned facts from initial state
16         Reset PoseGenerators with new random seed
17         Reset step, partialTraj, pose_1 to initial values
```

Algorithm 2: TryRefine Subroutine

```
   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)
11     else if GetMotionPlan(TargetPose(axn),
       TargetPose(nextAxn)) succeeds then
12         Traj ← Traj + ComputedPath; ActionNum++
13     if Mode = PartialTrajMode then
14         yield (TargetPose(axn), Traj, ActionNum+1,
                   GetMPErrors(TargetPose(axn),
                               TargetPose(nextAxn)))
       end
15     TargetPose(nextAxn)
                 ← PoseGenerator(nextAxn).getNext()
16     if ActionNum = length(HLPlan)+1 then
17         return Traj
       end
   end
```

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 Subroutine: 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 function *TargetPose()* maps an action to the target pose currently being considered for that action. We first discuss lines 11-15. If 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. Otherwise, the algorithm proceeds differently in error-free and partial-trajectory modes. In error-free mode it obtains another target pose for the next action. In partial-trajectory mode it yields the reasons for 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`). This 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, which are determined by dedicated modules. 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.

In practice, we carried out motion planning in error-free mode only when an IK solution existed.

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 $gpfg\_obj_i$, we need poses at which closing the gripper will result in a stable grasp of the object. Computing such poses is an independent problem and can be solved using state-of-the-art approaches for grasping. We assume that such poses are known for each object class (e.g. bowl, can, tray etc.) and used an approach where every grasp pose corresponds to a

pre-grasp 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 instantiates 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 present a sufficient condition under which our approach is guaranteed to find a solution if one exists. These conditions are not necessary: our empirical evaluation shows the algorithm succeeding in a number of tasks that do not satisfy these conditions.

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 such that there are no reachable dead-end states w.r.t. g and A is 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 all the calls to the motion planner terminate, then Alg. 1 will find a sequence of motion plans solving P if one exists using the pose references captured by G.

Intuitively, termination follows because under the premises, every time a state update takes place, missing geometric facts are added to the state and can only be removed by actions but not added again. We refer the reader to the appendix B for the proof.

V. Empirical Evaluation

We implemented the proposed approach using the OpenRAVE simulator [2]. In all of our experiments 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 [3]. For every motion planning query, Trajopt returns a trajectory with a cost. A wrapper script determined collisions (if any) along the returned trajectory. We used two task planners, FF [4] and the IPC 2011 version of FD [5] 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 impractical as they lead to hundreds of facts in the initial problem specifications. Since our system can work with any classical planner, we used FF for tasks where costs were not a concern. All the problems used an ambidextrous version of the PR2 actions shown in Fig. 3 with task-specific actions such as placing items on a tray and opening a drawer. The source code, problem generation scripts for use in benchmarking and videos for all tasks are available to the community at the website for this paper [6]. All experiments were carried out on Intel Core i7-4770K machines with 16GB RAM, with two tests running in parallel at a time. All the success rates and times are summarized in Table I.

Problem	%Solved in 600s	Avg. Solution Time (s)
Drawer (O)	100	34
Drawer (N)	100	185
Clutter-15	100	32
Clutter-20	94	57
Clutter-25	90	69
Clutter-30	84	77
Clutter-35	67	71
Clutter-40	63	68
Dinner-2	100	63
Dinner-4	100	133
Dinner-6	100	209

TABLE I: Summary of the results. All numbers except for the cluttered table problem are from 10 randomly generated problems. Cluttered table problems showed greater variance and are averages of 100 randomly generated problems for each number of objects.

A. Object in a Drawer

In this domain the robot needs to open a drawer and retrieve an object inside it. An object in front of the drawer prevents its complete opening. The inner object's placement determines where the robot should place itself to avoid collisions with the outer object, and whether it is possible to retrieve the inner object without moving the outer object. This task illustrates the generality of our approach in going beyond pick and place, and grasping in cluttered environments. We modeled it using an open-drawer action, whose pose generator generates random bounded values for the pull-distance. In the solution plans, our system chose to position the robot so as to open the drawer and access the inner object without moving the obstruction when possible. The results show average solution times for situations where removing the obstruction was optional (O) and necessary (N).

B. Cluttered Table

In this task, the objective is to pick up a target object from a cluttered table. There is no designated free space for placing objects, so the planning process needs to find spots for placing obstructing objects. In order to make the problem more challenging, we restricted pickups to only use side-grasps. The robot's thick grippers create several obstructions and, many of the pose instantiations lead to cyclic obstructions. Since placing objects adds obstructions, this task does not satisfy the premises of Thm. 1. In addition to the summarized results in Table I, Fig. 7 shows a histogram of the solution times. To the best of our knowledge, no other approach has been shown to perform at this level

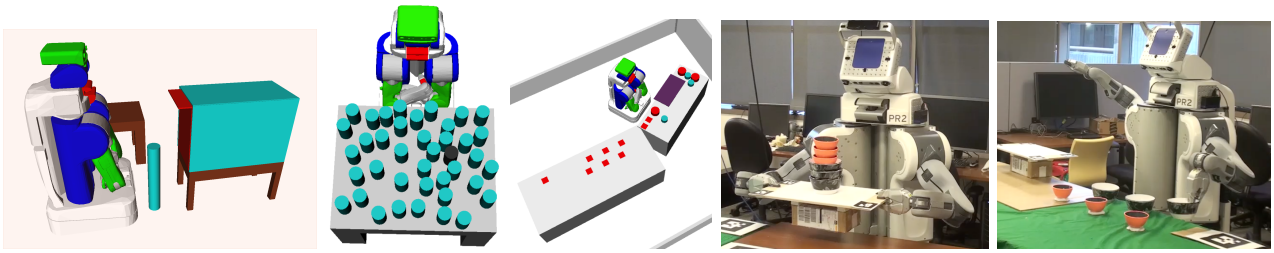


Fig. 5: Test domains from L to R: drawer domain, cluttered table with 40 objects where the dark object denotes the target object, and dinner layout. The rightmost images show the PR2 using the tray and completing the dinner layout.

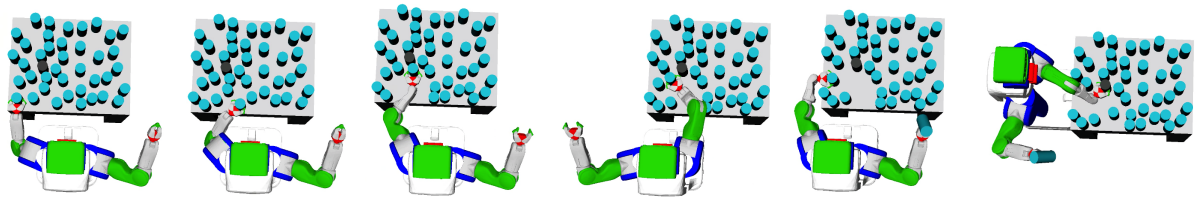


Fig. 6: Some of the steps executed while solving an instance of the 40 object cluttered table with the dark object as the target. Each snapshot shows a grasp being executed.

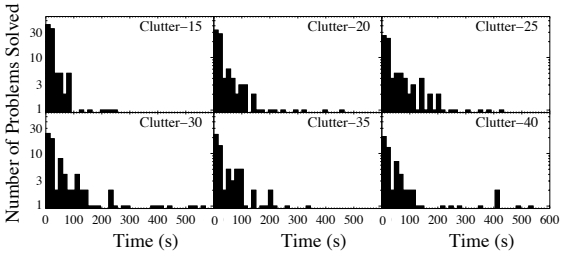


Fig. 7: Histograms of solution times for problems solved within 600s in the cluttered table domain. Y-axis is in log-scale.

on randomly generated constrained problems without using specialized geometric reasoning routines.

C. Laying Out a Table for Dinner

The goal of this task is to lay out a dinner table. A tray is available, but not necessary for transportation. 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 task-level moves across rooms. The geometric properties in this domain were stackability and relative positions of objects (see below). Stackability was determined using object diameters. The test scenarios had 2, 4 and 6 objects (cups and bowls with equal numbers of each), 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 3 million with just 6 objects. We used FD as the task planner since plan cost was a consideration.

Our system appropriately used the tray to transport items. It used inefficient movements when the robot picked objects on its left with its right hand (and vice versa) as the task planner 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 a 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 also added a *Handoff* action in the domain, which transfered an object from one hand to the other. The resulting behavior, though not guaranteed to be optimal, showed the system determining which hand to use for a particular grasp or putdown and whether or not a handoff should be done. To the best of our knowledge, no other approach has been shown to solve task and motion planning problems with such large high-level state spaces without using task-specific heuristics or knowledge beyond the set of primitive task-level actions.

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 this paper's website [6].

VI. Related Work

The closest related work was a preliminary version of the approach presented here [7]. It used a non-backtracking version of Alg. 2 and carried out online execution. Our approach builds upon the vast literature of related work in robotics and planning. 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 (e.g. [8]). Various researchers have investigated the problem of combining task and motion planning [9], [10], [11], [12]. However, to the best of our knowledge, very few approaches are able to utilize off-the-shelf task and motion planners and most rely on specially designed task and/or motion planning algorithms. Those that use off-the-shelf task planners make minimal use of task plans: 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 if necessary.

Cambon et al. [13] propose an approach that bears similarity to ours in using location references. The references in their approach however are not developed into a system for communicating geometric information to the task planner and correcting its domain definition. They are also not derived from a specification of action preconditions and effects. This approach requires the motion planner to use probabilistic roadmaps (PRMs) [14] with one roadmap per movable object, and per permutation of a movable object in each gripper for robots like the PR2. The role of the task plan is also limited to its length, which is one of the inputs in a heuristic function for search. However, their approach is probabilistically complete. Kaelbling et al. [15] 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. [16] 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. [17]), but they do not address the general problem of combining task and motion planning.

Reinvoking task planners relates to replanning for partially observable or non-deterministic environments [18], [19]. 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 for dealing with large sets of relevant facts in the initial state would be to treat them as initially unknown and use a partially observable planner with non-deterministic "sensing" actions [20]. Finding offline contingent solutions would be impractical in our setting, and they typically don't exist for all possible truth values of geometric predicates. Wolfe et al. [21] use angelic hierarchical planning to define a hierarchy of high-level actions over primitive actions. Our approach could be viewed as using an angelic interpretation: pose references in task plans are assumed to have *a* value that satisfies the preconditions, and the interface layer attempts to find such values. Approaches for planning modulo theories (PMT) [22] and planning with semantic attachments [23] address related problems high-level planning in hybrid domains. These approaches require an extended planning language and corresponding, 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 created a suite of problems that can be used as benchmarks for task and motion planning and also 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] M. Fox and D. Long, "PDDL2.1: an extension to PDDL for expressing temporal planning domains," *JAIR*, vol. 20, no. 1, pp. 61–124, 2003.
[2] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, 2010.
[3] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization," in *RSS*, 2013.
[4] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *JAIR*, vol. 14, pp. 253–302, 2001.
[5] M. Helmert, "The fast downward planning system," *JAIR*, vol. 26, pp. 191–246, 2006.
[6] "Source code and result videos." [Online]. Available: http://www.cs.berkeley.edu/~siddharth/icra14/
[7] 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.
[8] M. Helmert, P. Haslum, and J. Hoffmann, "Flexible abstraction heuristics for optimal sequential planning," in *ICAPS*, 2007, pp. 176–183.
[9] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *IJRR*, vol. 17, pp. 315–337, 1998.
[10] 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.
[11] E. Plaku and G. D. Hager, "Sampling-based motion and symbolic action planning with geometric and differential constraints," in *ICRA*, 2010, pp. 5002–5008.
[12] 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.
[13] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *IJRR*, vol. 28, pp. 104–126, 2009.
[14] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 566–580, 1996.
[15] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *ICRA*, 2011, pp. 1470–1477.
[16] M. Dogar and S. Srinivasa, "A framework for push-grasping in clutter," *RSS*, 2011.
[17] M. Levihn, J. Scholz, and M. Stilman, "Hierarchical decision theoretic planning for navigation among movable obstacles," in *WAFR*, 2012, pp. 19–35.
[18] 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 *AAAI*, 2010.

[19] S. W. Yoon, A. Fern, and R. Givan, "FF-replan: A baseline for probabilistic planning," in *ICAPS*, 2007.
[20] B. Bonet and H. Geffner, "Planning with incomplete information as heuristic search in belief space," in *ICAPS*, 2000, pp. 52–61.
[21] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *ICAPS*, 2010, pp. 254–258.
[22] P. Gregory, D. Long, M. Fox, and J. C. Beck, "Planning modulo theories: Extending the planning paradigm," in *ICAPS*, 2012.
[23] A. Hertle, C. Dornhege, T. Keller, and B. Nebel, "Planning with semantic attachments: An object-oriented view," in *ECAI*, 2012.
[24] H. J. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus," *Electronic Transactions on Artificial Intelligence*, vol. 2, pp. 159–178, 1998.
[25] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.
[26] B. Bonet and H. Geffner, "Planning under partial observability by classical replanning: Theory and experiments," in *IJCAI*, 2011, pp. 1936–1941.

APPENDIX

A. Abstract Formulation Through Quantifier Elimination

We introduce the central principle behind using pose referencs with an example. Consider an action like grasping in a pick and place domain. Its preconditions include real-valued vectors and preconditions require spatial reasoning. For clarity in this description, we consider a situation where objects are not stacked and assume that the target location where an object has to be placed is clear. The high-level grasp action can be specified as follows:

pick-simple(c, obj_1)
precon *IsBPFG*$(c, obj_1)\wedge$ *robotAt*(c)
$\wedge\forall obj_2\neg$ *obstructs*(c, obj_2, obj_1)
effect *in-gripper*(obj_1)

In this specification, the variable c represents robot configurations. In order to motivate our approach, consider the effect of this action in a framework like situation calculus [24] but using a timestep rather than a situation to represent the fluents:

$\forall t, obj_1 \forall c(IsBPFG(c, obj_1, t) \wedge robotAt(c, t) \wedge \forall obj_2 \neg obstructs(c, obj_2, obj_1, t) \rightarrow inGripper(obj_1, t+1))$

Note that this is not the complete *successor state axiom* for *in-gripper*, which will also have to include other actions that affect it and default conditions under which it doesn't change across timesteps. However, this implication is sufficient to illustrate the main representational device we will use. We first use the clearer, logically equivalent form:

$\forall t, obj_1(\exists c(IsBPFG(c, obj_1, t) \wedge robotAt(c, t) \wedge \forall obj_2 \neg obstructs(c, obj_2, obj_1, t)) \rightarrow inGripper(obj_1, t)$

which asserts more clearly that *any* value of c that satisfies the preconditions allows us to achieve the postcondition. We can now use the standard technique of Skolemization to replace occurrences of c with a function of obj_1, t:

$\forall t, obj_1((IsBPFG(gp(obj_1, t), obj_1, t)$
$\wedge\, robotAt(gp(obj_1), t)$
$\wedge\, \forall obj_2 \neg obstructs(gp(obj_1), obj_2, obj_1, t))$
$\rightarrow inGripper(obj_1, t+1)$

where the Skolem function $gp(x, t)$ is just a symbol of type *location* to the discrete planner. Intuitively, it represents a robot configuration corresponding to the grasping-pose of x. This representation will allow the discrete planner to treat the entire problem at a symbolic level, without the need for creating a problem-specific discretization. A potential problem however, is that the Skolem functions will depend on t, or the current step in the plan. In practice however, at the discrete level, the time argument in a Skolem function $f(\bar{x}, t)$ can be ignored as long as it is possible to recompute (or reinterpret) f when an action's precondition is violated by its existing interpretation during the execution (as is the case in our implementation). We therefore drop the t argument from the Skolem functions in the rest of this paper.

The equivalence with an existential form as described above can be used for each action effect as long as the continuous variable being Skolemized is not free in the subformula on the right of the implication. For instance, we can add the effect that grasping an object removes all obstructions that it had created, regardless of robot configurations. Therefore, to represent the grasp operator for a discrete planning problem, rather than using a discretized space of configurations, we only need to add symbols of the form $f(\bar{o})$ in the planning problem specification, for each object argument tuple $\bar{o}$ consisting of the original objects, or constants in the problem. Since many classical planners don't support functions, they can be reified as objects of the form $f\_\bar{o}$ with an associated set of always true relations, e.g. $is\_f(f\_o_i, o_i)$. The discrete description of pick-simple thus becomes:

pick-simple(ℓ, obj_1)*:*
precon *is_gp*$(\ell, obj_1)\wedge$ *IsBPFG*(ℓ, obj_1)
$\wedge$ *robotAt*(ℓ)
$\wedge\forall obj_2\neg$ *obstructs*(ℓ, obj_2, obj_1)
effect *in-gripper*(obj_1)
$\wedge\forall \ell_2, obj_3 \neg$*obstructs*$(\ell_2, obj_1, obj_3)$

Here ℓ ranges over the finite set of constant symbols of the form gp_obj_i where obj_i are the original constant symbols in the problem. In this way, regardless of how many samples are used in the lower level process for interpreting these symbol, the discrete planner has a limited problem size to work with while computing the high-level plan.

Finally, consider the only remaining case for an action effect, when a continuous variable occurs freely in the subformula on the right, e.g. $\forall x, c(\varphi_{precon}(x, c) \rightarrow \varphi_{effect}(x, c))$. In this case we don't perform Skolemization. The symbol used for c in this case will be an action argument, and must range over the original objects in the domain or those already introduced via Skolemization.

Although this exhausts the set of actions commonly used in PDDL benchmark problems, the accurate description of an action may involve side-effects on symbols not used as its arguments. E.g., the putDown(obj_1, loc_1) action may deterministically introduce obstructions between a number of robot configurations and other objects. We don't encode such side effects in the high-level planning problem specification; these facts are discovered and reported by the lower level when they become relevant.

To summarize, we transform the given planning domain with actions using continuous arguments by replacing occurrences of continuous variables with symbols represent-

ing Skolem function application terms. Every continuous variable or the symbol replacing one, of type τ gets the new type τ_{sym}. Planning problems over the modified domains are defined by adding finite set of constants for each such τ_{sym} in addition to the constants representing problems in the original domain. The added constants denote function application terms, e.g. *gp_obj17*, for each physical object and Skolem function application. This increases the size of the input only linearly in the number of original objects if the Skolem functions are unary. Note that the set of Skolem functions itself is fixed by the domain and does not vary across problem instances. The initial state of the problem is described using facts over the set of declared objects, e.g. "*is_gp(gp_obj17, obj17)*" denoting that the location name *gp_obj17* is a grasping pose for *obj17* and "*obstructs(gp_obj17, obj10, obj17)*", denoting that *obj10* obstructs *obj17* when the robot is at *gp_obj17*.

In this formulation, the size of the input problem specification is independent of the sampling-based discretization: we do not need to represent sampled points from the domains of continuous variables.

B. Formal Results

We now discuss the conditions under which our solution approach is complete. In doing so we show that under certain conditions, effective default assignments for atoms can be ontained easily. We use the notion of *probabilistically-complete* [25] to categorize sampling based algorithms that are guaranteed to find a solution with sufficiently many samples.

The following definition uses the concept of positive and negative occurrences of atoms in formulas. Intuitively an occurrence of an atom in a formula is negative (positive) if it is negated (non-negated) in the negated-normal-form of the formula. This notion of occurrence is sufficient for our purposes as we deal only with problems with finite universes and all quantifiers can be compiled into conjunctions or disjunctions. The following lemma follows from the definition of positive and negative occurrences:

Lemma 1: Suppose an atom $p(c)$ occurs only positively (negatively) in a ground formula φ. If s is an assignment under which φ is true then it must also be true under an assignment s' that makes $p(c)$ true (false) and is the same as s for all other atoms.

Definition 2: A planning domain 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

Let P_S be the set of predicates whose atoms may use pose references as one of their arguments, and let $D = \langle \mathcal{R}, \mathcal{A} \rangle$ be a planning domain that is uniform wrt a goal g and P_S. In the following, consider atoms over a fixed set of constants U. Let s_{part} be an assignment of truth values to atoms over $R \setminus P_S$. Let $s_{default}$ be an assignment of truth values to atoms over P_S, assigning the atoms of each positively (negatively) occurring predicate the truth value true (false).

Proposition 1: The state $s_{part} \cup s_{default}$ has a solution plan for a goal g iff there is some assignment s_0 of atoms over P_S such that $s_{part} \cup s_0$ has a solution plan for reaching g.

Proof: Suppose there is an assignment s_0 under which $s_{part} \cup s_0$ has a solution π and which assigns an atom $p(\bar{c}_1)$ true, while all occurrences of p in action preconditions and g are negative. Consider the assignment s'_0 which assigns $p(\bar{c}_1)$ false, but is otherwise the same as s_0.

We show that π solves $s_{part} \cup s'_0$ as well. Suppose not, and that $a(\bar{c}_2)$ is the first action in π whose preconditions are not satisfied when π is applied on $s_{part} \cup s'_0$. In this failed execution, $a(\bar{c}_2)$ must have been applied on a state s'_k that differs only on $p(\bar{c}_1)$ from the corresponding state $s_k$in the execution of π on s_0. This is because all preceding action applications succeded and have the same deterministic effects in both executions. Let the ground formula representing the preconditions of this application of $a(\bar{c}_2)$ be φ. By Lemma 1, φ must be satisfied by s'_k, and we get a contradiction: $a(\bar{c}_2)$ must be applicable on s'_k. The case for positive defaults is similar. ■

Thus, in planning problems that are uniform wrt to the set of predicates which use pose references, it is easy to obtain a default truth assignment for atoms over these predicates, under which the problem is solvable if there is any assignment to those atoms under which it is solvable. The following result provides sufficient conditions under which our approach is complete. Let D be a planning domain and U a set of typed constants. A dead-end for $\langle D, U \rangle$ wrt g is a state over $atoms(\mathcal{R}, U)$ which has no path to g.

We say that two low-level states are physically similar if they differ only on the interpretation of pose references and atoms using pose references. We use the symbol $[s]$ to denote the discrete representation of a low-level state s, and $[s]_{default}$ to represent the version of $[s]$ obtained by using the default assignment for atoms using pose references, and the assignments in $[s]$ for all other atoms.

The following presents sufficient conditions under which our approach solves any problem which is solvable under some evaluation of pose references.

We now present the main theoretical result. Intuitively, the result holds 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. 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.

Theorem 2: Let D be a planning domain, U a set of typed constants, g a goal, and P_S the set of predicates in D that use pose references, such that $\langle D, U \rangle$ does not have dead-ends wrt g and D is uniform wrt g and P_S. 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 then for any low-level state s, if there exists a physically similar state s' such that $[s']$ is solvable

by the high-level planner then the execution of Alg. 1 with inputs $[s]_{default}$ and D will produce a motion plan leading to the goal.

Proof: The proof follows from the following two points:
1. For any given interpretation of pose references represented by a low-level state s'_ℓ, Alg. 1 will eventually either discover the truth values of all atoms using pose references or identify the interpretation as unsolvable.

This is because in every non-final iteration of Alg. 1, line 7 must be executed and will return a non-empty assignment of atoms constituting a violated precondition. The entire set of atoms that violated preconditions are generated from is finite. Once they have been returned by the low level, they will never be generated again for this interpretation. This is because the atom is incorporated into the new discrete state accurately. Subsequent actions can only make it true (false) when it occurs positively (negatively) in preconditions. As a result, the discrete planning layer will never consider an atom that occurs positively to be true if its truth value was returned as false (true) by the low-level—unless an action set it to true in which case the low-level also considers it to be true.

If at any stage a discrete state s with default values for some atoms using pose references is unsolvable, then replacing them with any other truth values will not make the problem solvable. Thus, if at any stage the classical planner call in line 2 fails for a discrete state s with default values that are inconsistent with the low-level state for some atoms using pose references, the problem is indeed unsolvable under that interpretation.
2. As a result of probabilistic completeness, the low-level will discover all possible interpretations of pose references. ■

The uniform property used in this result plays a role similar to *simple domains* defined by Bonet and Geffner [26]. 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.