# Efficiently Combining Task and Motion Planning Using Geometric Constraints

Fabien Lagriffoul, Dimitar Dimitrov, Julien Bidot, Alessandro Saffiotti and Lars Karlsson
AASS Cognitive Robotic Systems Lab
Örebro University, S-70182 Örebro, Sweden
<name>.<surname>@oru.se

***Abstract*— We propose a constraint-based approach to address a class of problems encountered in Combined Task and Motion Planning (CTAMP), which we call *kinematically constrained problems*. CTAMP is a hybrid planning process in which task planning and geometric reasoning are interleaved. During this process, symbolic action sequences generated by a task planner are geometrically evaluated. This geometric evaluation is a search problem per se, which we refer to as *geometric backtrack search*. In kinematically constrained problems, a significant computational effort is spent on geometric backtrack search, which impairs search at the task-level. At the basis of our approach to address this problem, is the introduction of an intermediate layer between task planning and geometric reasoning. A set of constraints is automatically generated from the symbolic action sequences to evaluate, and combined with a set of constraints derived from the kinematic model of the robot. The resulting constraint network is then used to prune the search space during geometric backtrack search. We present experimental evidence that our approach significantly reduces the complexity of geometric backtrack search on various types of problem.**

## I. Introduction

In order to be autonomous in performing everyday-life tasks such as setting or cleaning a table, preparing coffee, etc., humanoid robotic assistants must be able to do complex object manipulation. Performing such tasks autonomously involves different levels of reasoning. High-level reasoning is needed to decompose the task into a sequence of sub-tasks which achieves the goal. At this level, reasoning is done about causal and precedence relationships between actions (task planning). Low-level reasoning is needed to plan for the physical execution of each sub-task (motion planning). Combining these two levels of reasoning leads to a new search problem that we call *geometric backtrack search*. The search space in geometric backtracking can be very large, and therefore we concentrate on this issue in this paper. We contribute a constraint-based approach to reducing that search space. We begin by motivating the need for geometric backtrack search through a concrete example with the robotic platform Justin (Ott et al., 2006).

Consider for instance the task of stacking three cups on a dish rack with the same orientation. The task can be carried out in different ways, but imagine for example that the task planner currently is evaluating the geometric instantiation of the following symbolic action sequence: `<pick cup3, place cup3 rack, pick cup2, stack cup2 cup3, pick cup1, stack cup1 cup2>`. In Fig. 1, one can see that the left arm of the robot has almost reached full extension. Stacking the first two cups is possible, but placing the last cup on top of the pile with the required orientation may fail because of the *kinematic constraints* of the robot. The choice of the position and orientation of the cup at the bottom of the pile is crucial for the success of the last action. If the first cup is placed too far from the robot, or with an inappropriate orientation, the sequence of actions may not be feasible. In this case, the symbolic plan is in principle feasible, but the geometric instantiation chosen for the first action leads to a failure. In order to find a solution to such problems, alternative geometric instances for the pose of the first cup must be tried out until a solution is found. In the present work, this is achieved by performing *geometric backtrack search*, a process in which all the combinations of geometric instances (up to some spatial resolution) are systematically considered. We call *kinematically constrained problems* the problems in which the cause for geometric backtracking is the violation of the kinematic constraints of the robot.

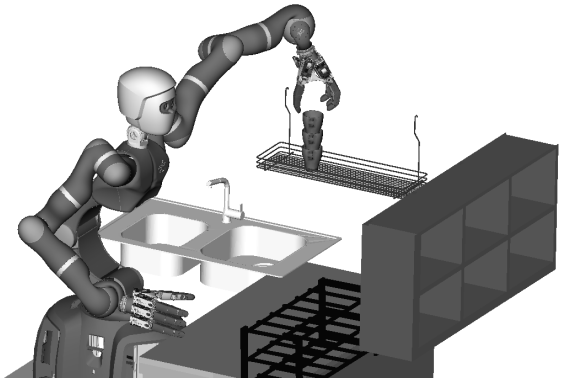

Fig. 1. Simulation of the DLR humanoid two-arm system JUSTIN: stacking the last cup is not possible due to kinematic constraints.

An important issue to consider in CTAMP is the case where a symbolic action sequence has to be *rejected*, because there is no way to instantiate it geometrically. This occurs often in CTAMP, because the task planner cannot reason about geometric aspects, therefore it often generates action sequences which are logically valid, but geometrically infeasible. Ideally, the low-level (geometric)

reasoning should reject infeasible action sequences as fast as possible, so that the high-level (task planner) could carry on searching for alternative action sequences, until a valid plan is found. Unfortunately, proving that a search problem has no solution is often more difficult than finding *a* solution, because it requires to explore the search space *exhaustively*. The approach proposed in this paper addresses this issue by allowing, in many cases, to bypass exhaustive search with a mere consistency check.

This paper does *not* present a new scheme for combining task and motion planning, but it describes a constraint-based approach for pruning the search space explored during geometric backtrack search. A constraint network is automatically built from the symbolic action sequence to be evaluated and from the geometric characteristics of the robot. This constraint network is then used during geometric backtrack search to remove inconsistent actions or inconsistent sets of actions, thus reducing the search effort. In some cases, the constraint network allows us to *reject* an action sequence for which no consistent geometric instantiation exists, hence avoiding to perform exhaustive geometric search on that sequence and speeding up the whole search process. The approach presented in this paper makes several simplifying assumptions:

- constraints are expressed at grasp and release positions only, not for the motions in-between;
- at grasp and release positions, the gripper(s) of the robot and the manipulated object(s) are constrained to be aligned about an a priori given finite set of reference axes.
- although the constraints used for pruning are expressed as continuous ranges, the space of geometric choices for the robot's actions is discretized; and
- all the reported experiments were conducted in simulation, assuming exact knowledge of objects poses and perfect execution of the robot's motions; simulation is acceptable here, since the goal of the experiments is to evaluate the effect of constraints on the planning process.

The rest of this paper is organized as follows. In the next section, we provide the reader with basic notions in task and motion planning, and present some of the related literature. In sections III and IV, an overview of the approach is presented, and the notation is introduced. Sections V and VI describe in detail how the constraints are generated, and which algorithms have been developed. The time-complexity analysis of these algorithms is given in section VII. Finally, we report on the experimental evaluation of the proposed approach in section VIII.

## II. Background and related work

### A. Task planning and motion planning

The scope of this paper is neither task planning nor motion planning, hence, we limit ourselves to informal definitions of task planning and motion planning, as a background for reading the next sections, and provide references for further reading.

*1) Task planning:* Task planning has to do with causal reasoning, i.e., reasoning about the preconditions and effects of the different actions an agent can take in order to reach a certain goal. These actions are an abstraction of the real-world actions, hence the geometric aspects are not represented. Symbols are used to represent predicates (relations) and constants of the application domain. A *symbolic state* can be represented by a conjunction of atomic statements, e.g.,

```
on cup1 table
empty right_hand
grasped cup2 left_hand
...
```

An *operator* is a triplet $\langle A(p_1, \ldots, p_n), pre, \mathit{eff} \rangle$ where

- $A$ is an action symbol;
- $p_1, \ldots, p_n$ are the parameters of the action;
- $pre$ is a logical expression called *precondition*;
- $\mathit{eff}$ is a set of literals called *effects*.

A *planning domain* is a set of *operators*. A *planning problem* is defined by a *planning domain*, an initial state, and a goal state which are described using *predicates* and a set of symbols called *constants*. A *plan* is a sequence of instantiated operators which transforms the initial state into the goal state. We illustrate these terms with an example:

- a planning domain:

action: `PICK ?hand ?grasp_type ?obj`
pre: `empty ?hand`
eff: `grasped ?obj ?hand, not(empty ?hand)`

action: `PLACE ?hand ?obj ?location`
pre: `grasped ?obj ?hand`
eff: `empty ?hand, at ?obj ?location`

- a planning problem:

initial state: `at cup1 sink, grasped cup2 left_hand, empty right_hand`
goal state: `at cup1 table`
constants: `left_hand, right_hand, cup1, cup2, table, sink`

- a plan:

```
PICK right_hand cup1
PLACE right_hand cup1 table
```

or, using the left hand:

```
PLACE left_hand cup2 sink
PICK left_hand cup1
PLACE left_hand cup1 table
```

Many techniques exist for task planning (Nau et al., 2004), some of which go beyond causal reasoning, e.g., taking into account temporal constraints or resource usage. Some planners can deal with numerical values and, by assigning costs to actions, solve problems involving distances and/or robot motions, but no task planner can deal

with the fine-grained geometrical representations which are needed for solving robot manipulation tasks. In order to do this, a task planner should call a specialized reasoner (motion planner) which assesses the feasibility of symbolic actions using its own algorithms and representations. Next, we describe the motion planning problem.

*2) Motion planning:* The motion planning problem consists in finding a collision-free path (a sequence of configurations) between a given initial configuration and a desired final configuration (see LaValle (2006) for an comprehensive review of motion planning techniques). A *configuration* generally represents the pose of one or several rigid bodies in space. It can represent the poses of a fleet of vehicles, as well as the angular values of the individual joints in a robotic manipulator. In problems where objects can be manipulated, a configuration can be defined as the pose of the robot together with the poses of the manipulable objects. In the present work, a humanoid robot is used, and the term *configuration* denotes the poses of all objects to be manipulated and the angular values of joints of both manipulators of the robot. The motion planning algorithms we use operate in the joint space of each manipulator.

Motion planners cannot do task planning, since they operate in the configuration space of the robot. However, manipulation planning algorithms (for instance Siméon (2004)) can produce plans in which the robot picks, places, and re-grasps several times an object until it can place it in the desired pose. However, task planning is needed for solving tasks where some actions have non-geometric effects, e.g., charging the battery of the robot, calling an elevator, switching on the light, taking a picture, etc. An example of such task is given in section VIII: filling a glass (experiments 3 and 6). This task cannot be specified only in terms of an initial and final configuration, otherwise the liquid may be poured on the hand that holds the glass, or the liquid may be poured on the table before the glass is set in a correct position. The order in which the steps are performed matters, therefore the sequence of actions needs to be computed using task planning techniques. For this reason, planners which can reason on both levels have been developed. We describe some of them in the next section.

### B. Related work

Recently, several CTAMP approaches were proposed. A common type of approach consists of a task planner steering the search, while a geometric reasoner is called to geometrically instantiate the symbolic actions. In SAHTN (Wolfe et al., 2010) for example, a hierarchical planning approach is used, and the geometric solutions to similar subproblems are cached in order to avoid redundant geometric computations. In Kaelbling and Lozano-Pérez (2011), a hierarchical approach is also used. Thanks to geometric suggesters, the planner can commit early on geometric choices, which reduces the search space and allow them to address large problem instances. In Karlsson et al. (2012), the focus is on reconsidering previous geometric choices, through a systematic geometric backtracking process. In Dornhege et al. (2009), an extension of PDDL[1] is proposed to handle calls to external reasoners, hence keeping the soundness and completeness properties of existing planners (under strict assumptions on these external reasoners). In Guitton and Farges (2009) the symbolic operators description is also augmented, and geometric constraints are proposed as a generic interface between symbolic and geometric levels. A central issue in CTAMP (which is not addressed in this article) is how to use information from the geometric level to guide the search at the task level. This issue is addressed in de Silva et al. (2013), where a set of geometric predicates (e.g., reachability or visibility of objects) are dynamically computed from the geometric state during task planning. This allows them to deal with indirect effects of actions, and opens for potentially richer domains. The approach of Srivastava et al. (2013) also deals with this issue. The domains of continuous variables are represented by a finite set of *Skolem symbols*. Through re-planning, these symbols are used to provide feedback to the task planner why a particular action failed.

In another type of approach, the planner works mainly on a motion planning problem, while the task planner is used as a heuristic. In Plaku and Hager (2010) for instance, SamplSGD consists of a motion planner handling differential constraints guided by a task planner. A utility function provides a loose interaction between the symbolic and the geometric levels. Similarly in aSyMov (Cambon et al., 2009), the task planner FF (Hoffmann and Nebel, 2001) guides a manipulation planner based on the composition of several probabilistic roadmaps (PRMs), which can deal with complex problems involving the cooperation of heterogeneous robots. Another less common type of approach states the symbolic planning problem in terms of logic programming. In Choi and Amir (2009), a sampling-based motion graph is used to build an action theory, from which an abstract solution plan is extracted. In the same vein, the action language $C+$ (Erdem et al., 2011) or ASP programs (Aker et al., 2012) are used to encode the planning problem into a logic program. The failures detected at the geometric level are fed back in the form of logical constraints to the causal reasoner to find alternative plans that are free of these failures. The advantage of this approach is the expressiveness of the formalisms which can handle concurrency, non-deterministic effects, ramifications, etc.

Note that these CTAMP approaches are in general incomplete. The motion planners they employ are incomplete, and as a large number of motion planning problems must be solved, a limited amount of time must be assigned to each. Often, continuous geometric choices such as target poses for objects have to be discretized. In addition, a number of approaches, such as Dornhege et al. (2009), do not support reconsideration of geometric choices of previous actions. Finally, several approaches employ HTN task planners which may be incomplete depending on how

[1] Planning Domain Definition Language

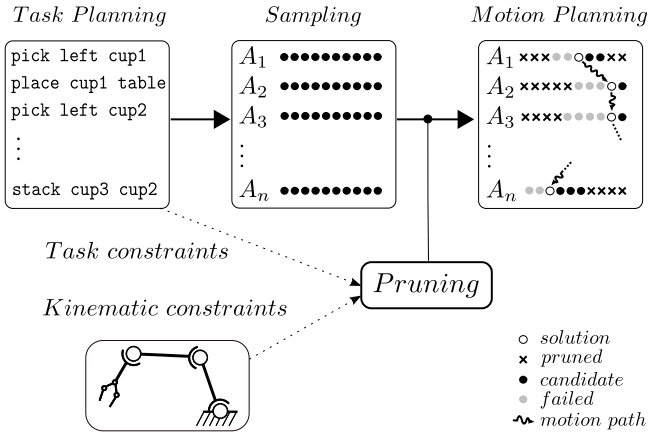


Fig. 2. Geometric instances of symbolic actions $(A_1, A_2, \ldots, A_n)$ are sampled. Task constraints (i) and kinematic constraints (ii) are used to prune out some geometric instances, which reduces geometric backtracking.

the HTN methods are defined.

The difficulty of geometric reasoning was early pointed out in Lozano-Perez et al. (1989) which showed that a simple task such as pick-and-place is not trivial. Because of the kinematic constraints, the task often needs to be decomposed into a sequence of re-grasping operations, for which a specialized reasoner is needed, e.g., Tournassoud et al. (1987); Cho et al. (2003). A similar issue is addressed in the work of Pandey et al. (2012) on grasp-placement selection under various task constraints. Different sampling-based planning techniques have been developed to solve problems that are more complex than mere motion planning; for example, manipulation planning (Siméon et al., 2004), multi-modal motion planning (Hauser et al., 2007), or planning among movable obstacles (Wilfong, 1988; Stilman and Kuffner, 2008). These techniques combine motion planning with discrete choices in order to explore high-dimensional configuration spaces more efficiently. However, they cannot address problems in which actions with non-geometric effects are involved.

The work presented in this paper is the continuation of the work initiated in Lagriffoul et al. (2012). Here, we describe in more details the overall architecture and the interface with the task planner (see next section). A strong limitation of the previous work was that it could only deal with rotations around the vertical axis. The present work shows that our scheme can be extended to an arbitrary number of axes, which makes it applicable to a wider range of problems. We also expose a study of the time complexity of the algorithms.

## III. Overview of our approach

A high-level view of our planning architecture is depicted in Fig. 2. A task planner outputs a sequence of symbolic actions. Each symbolic action can be instantiated in different ways at the geometric level, e.g., for the action `place cup1 table`, different positions and orientations on the table can be considered (more details can be found in Karlsson et al. (2012)). Sampling is required to generate a set of candidate geometric instances, from which we search for one which:

- is collision free;
- has an inverse kinematic solution;
- is reachable from the previous geometric instance (a collision-free path exists).

In the introduction, we have identified *geometric backtrack search* as necessary in order to ensure the completeness of this process. We focus on kinematically constrained problems, which require extensive search in the space of geometric instances. In order to reduce this search space, geometric instances are pruned out using a filtering technique based on linear programming. Linear constraints are automatically generated (i) from the symbolic sequence of actions (task constraints) and (ii) from the geometric model of the robot (kinematic constraints). These constraints are *conservative*, i.e., they can safely be used for pruning out infeasible sampled geometric instances. In this way, only the geometric instances that are within the feasible set have to be checked against inverse kinematics, collision detection, and motion planning.

The present work focuses on pruning the geometric search space using constraints generated from a sequence of symbolic actions. Note that this technique can be applied on partial plans as well, i.e., during task planning. Doing so entails additional considerations though, such as posting and rolling back constraints during symbolic backtracking, which would make the analyses proposed in sections VII and VIII tedious and less precise. We refer the reader interested in this issue to Bidot et al. (2013), where we report on our work on tightly integrating constraint reasoning and task planning. Next, we describe in more details some requirements put on the planning domain, and illustrate through an example how symbolic and geometric levels interact.

### A. Requirements on the planning domain

In this section, we describe how we implemented the planning domain, so that the symbolic plan generated by the task planner contain enough information to be converted into a set of constraints. This is achieved by using *coarse geometric representations*. For instance, we use the symbols `Ox1`, `Oy1`, `Oz1` to represent the fact that an object has its main axis aligned with respectively the $x, y, z$ axes of the world frame, and `Ox2`, `Oy2`, `Oz2` denote alignment with the opposite axes $-x, -y, -z$ (e.g., upside-down is represented by `Oz2`). The grasp type (side, top, bottom) and the manipulator used (left, right) are also represented at the symbolic level, hence decided by the task planner. However, the exact orientation of the TCP relative to the object during grasping, or position on the location during placing, are determined by the geometric reasoner, using a deterministic uniform sampling procedure based on Van der Corput sequences (Kuipers and Niederreiter, 1974). We define a minimal set of parameters for the actions:

A `pick`-like action (`pick`, `pick-regrasp`) must be at least parametrized by:

- *side*: `left_hand`, `right_hand`;

- *grasp_type*: `side`, `top`, `bottom`, ...
- current *coarse_object_orientation*: `Ox1`, `Ox2`, `Oy1`, ...
- *object*.

A `place`-like action (`place`, `place-regrasp`, `stack`) must be at least parametrized by:

- *side*: `left_hand`, `right_hand`;
- *grasp_type*: `side`, `top`, `bottom`, ...
- *target_location*;
- target *coarse_object_orientation*: `Ox1`, `Ox2`, `Oy1`, ...
- *object*.

To illustrate our action parametrization scheme with an example, consider the sequence of actions depicted in Fig. 3, where the robot grasps a cup with the left manipulator, re-grasps it with the right manipulator, and places it on the tray located on the table:

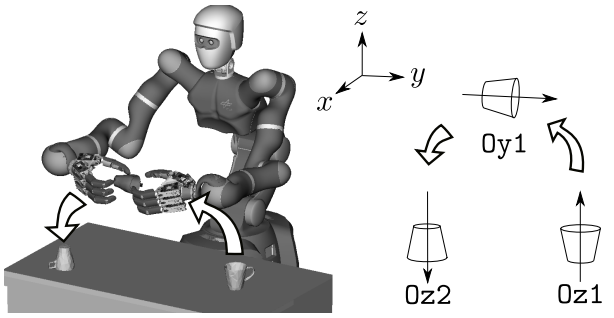


Fig. 3. Example of action sequence, with the corresponding coarse object orientations used by the task planner.

The corresponding symbolic action sequence is:

```
pick left_hand top Oz1 cup1
place_regrasp left_hand top Oy1 cup1
pick_regrasp right_hand bottom Oy1 cup1
place right_hand bottom tray Oz2 cup1
```

In other words, the task planner is not aware of the exact orientation of objects: it only knows if an object is aligned with the $x, y$, or $z$ axis, and in which direction. Similarly for grasps, the task planner reasons about the type of grasp without knowing the exact orientation of the TCP relative to the object. The exact angular values are handled at the constraint level (see section V). Next, we explain how these coarse geometric representations are used by the task planner for high-level geometric reasoning, and how they are used in the process of converting a symbolic action sequence into a set of linear constraints for quantitative geometric reasoning.

### B. Task-motion planning interaction

In order to understand the relevance of geometric evaluation (instantiation / rejection) of symbolic action sequences, it is helpful to see this problem in the context of CTAMP, i.e., in the task-motion planning loop. We roughly sketch an example showing why the symbolic level requires to repeatedly perform such evaluations. Consider for instance that you assign a robot the following task: place cup1 on the tray, upside-down. A solution to achieve this task is the plan depicted in Fig. 3 (left). But imagine that the cup was already upside-down, then this plan does not work, instead, the table has to be used as a temporary location to re-grasp the cup (because the tray is not reachable by the left manipulator):

```
pick left_hand bottom Oz2 cup1
place left_hand bottom Oz2 cup1 table
move_away left_hand
pick right_hand bottom Oz2 cup1
place right_hand bottom tray Oz2 cup1.
```

If the cup is already upside-down, and the tray is reachable by the left arm, then two actions are sufficient:

```
pick left_hand bottom Oz2 cup1
place left_hand bottom Oz2 cup1 tray
```

If the tray is reachable by the left arm, not reachable by the right arm, and the cup is initially in upright position, then the right arm and the table have to be used in order to flip the cup:

```
pick left_hand top Oz1 cup1
place_regrasp left_hand top Oy1 cup1
pick_regrasp right_hand bottom Oy1 cup1
place right_hand bottom table Oz2 cup1
pick left_hand bottom Oz2 cup1
place left_hand bottom Oz2 cup1 tray
```

Similarly, we could think of 4 symmetric plans (interchanging the `left_hand` and `right_hand` parameters), meaning that at the symbolic level, there are 8 candidate plans to solve this simple problem. This example gives an insight about the complexity of planning at the task level. Essentially, the number of candidate plans grows exponentially with the number of objects and the number of actions. For the task of moving 3 cups, several thousands of plans are possible, because the actions can be interleaved in different ways, and the order in which cups are manipulated can be changed. Hence, without a technique to evaluate these action sequences efficiently at the geometric level, search at the task level becomes intractable.

## IV. Notation

### A. Representing the poses of rigid bodies

For collision detection and motion planning, the pose of a rigid body is generally represented using a homogeneous transformation matrix, in which the orientation is represented by a rotation matrix. However, we want to model our problem with linear constraints only (see next section) in order to take advantage of linear programming techniques. We represent the orientation of a rigid body as the rotation of a reference frame about a reference axis. By constraining the rotation in this way, the orientation can be expressed using one angular value, on which constraints can be more easily formulated. Issues with wraparound of angles remain though, which we address in section VI-D. Translations are represented as usual.

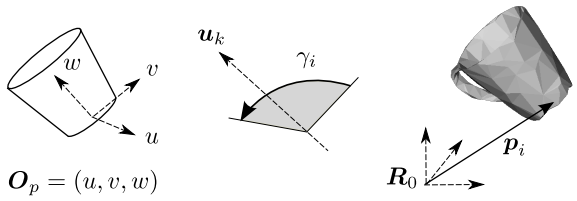


Fig. 4. The different components of the hybrid pose representation, from left to right: the object template frame $\mathbf{O}_p$, the reference axis and angle of rotation, and the translation in the world frame $\mathbf{p}_i$

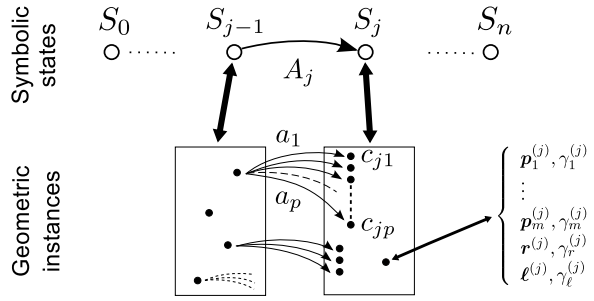


Fig. 5. Representation of symbolic states and stages of operation. Depending on how a symbolic action is performed, a symbolic state corresponds to many geometric instances.

For conciseness of notation, in some cases, we use $\boldsymbol{x} = (x_1, \ldots, x_n)$ to denote the elements of a column vector $\boldsymbol{x}$. All coordinates are expressed in the world frame. The pose of a body $o_i$ will be noted $(\boldsymbol{p}_i, \boldsymbol{O}_p, \boldsymbol{u}_k, \gamma_i)$, where $\boldsymbol{p}_i = (x_i, y_i, z_i) \in \mathbb{R}^3$ represents the translation of the $i^{th}$ body. $\boldsymbol{O}_p$ represents an orthonormal body-fixed frame, which we define as *object template frame*, $\boldsymbol{u}_k \in \mathbb{R}^3$ a unit vector which we define as *reference axis*, and $\gamma_i \in \mathbb{R}$ an angle of rotation around the axis $\boldsymbol{u}_k$.

The pose of the body in the world frame is parametrized by rotating the frame $\boldsymbol{O}_p$ about $\boldsymbol{u}_k$ with the angle $\gamma_i$, and translating it by $\boldsymbol{p}_i$ (see Fig. 4). $\boldsymbol{O}_p$ belongs to a predefined set of body-fixed frames, and $\boldsymbol{u}_k$ is chosen among a predefined set of axes. Object template frames correspond to orientations of interest for an object class, i.e., stable resting orientations, or orientations used during re-grasping operations. Each template frame is associated to a reference axis about which it can be rotated, but we use different indexes because different template frames can be associated to the same axis.

The same representation is used for the poses of the TCPs of the robot. In our setup, the robotic platform Justin has two TCPs, left and right, which we represent by $(\boldsymbol{\ell}, \boldsymbol{R}_p, \boldsymbol{u}_k, \gamma_\ell)$ and $(\boldsymbol{r}, \boldsymbol{R}_p, \boldsymbol{u}_k, \gamma_r)$ respectively. Similarly, *TCP template frames* correspond to orientations of interest for the TCPs, i.e., orientations used for pick, place, and re-grasping operations, for various types of grasps. For clarity in the rest of this article, we will simply represent the pose of an object $o_i$ by $(\boldsymbol{p}_i, \gamma_i)$, and the pose of right and left TCPs respectively by $(\boldsymbol{r}, \gamma_r)$ and $(\boldsymbol{\ell}, \gamma_\ell)$.

The limitation of this representation is that all possible orientations cannot be represented, since finite sets of template frames and reference axes are used. However, many man-made objects have a default stable upright position, which can naturally be used as a template frame associated to the $z$ axis. Hence, the limitation of this representation is not really for pick and place operations, but rather for re-grasping operations, for which limiting the set of reference axes may compromise the feasibility of re-grasping. Although this representation is limited, it allows us to formulate linear constraints on the orientations of TCPs and objects, and speed up the planning process. This choice is therefore a trade-off between geometric resolution and planning performance, which is discussed in the conclusion.

### B. Representing actions and states

Let $\langle A_1, \ldots, A_n \rangle$ be a sequence of symbolic actions, e.g., `pick left bottom Oz2 cup1`, `place left bottom Oz2 cup1 table`, etc. At the geometric level, a symbolic action $A_j$ can be performed in various ways, e.g., a `pick` action can be performed with different orientations of the TCP, a `place` action can result into different positions/orientations for the object. We denote the geometric instantiation of a symbolic action $A_j$ by $a_k, k \in \{1, \ldots, p\}$, where $p$ depends on the type of action and the resolution used for sampling (see section VII-A). We denote by $S_j$ the symbolic state resulting from applying the symbolic action $A_j$ on the previous symbolic state (see Fig. 5). We consider $m$ rigid objects. The $i^{th}$ object is denoted by $o_i$, $i \in \{1, \ldots, m\}$. The position of object $o_i$ in state $S_j$ (i.e., after action $A_j$ has been completed) is denoted by $\boldsymbol{p}_i^{(j)}$, and its orientation, by $\gamma_i^{(j)}$:

$$\left(\boldsymbol{p}_i^{(j-1)}, \gamma_i^{(j-1)}\right) \xrightarrow{A_j} \left(\boldsymbol{p}_i^{(j)}, \gamma_i^{(j)}\right).$$

Hence, a sequence of $n$ actions $\langle A_1, \ldots, A_n \rangle$ corresponds to a sequence of poses for all $m$ objects:

$$\langle A_1, \ldots, A_n \rangle \rightarrow \left\langle \begin{array}{c} (\boldsymbol{p}_1^{(0)}, \gamma_1^{(0)}, \ldots, \boldsymbol{p}_1^{(n)}, \gamma_1^{(n)}) \\ \vdots \\ (\boldsymbol{p}_m^{(0)}, \gamma_m^{(0)}, \ldots, \boldsymbol{p}_m^{(n)}, \gamma_m^{(n)}) \end{array} \right\rangle.$$

Note that $(\boldsymbol{p}_i^{(j-1)}, \gamma_i^{(j-1)}) = (\boldsymbol{p}_i^{(j)}, \gamma_i^{(j)})$ when object $o_i$ is not manipulated during action $A_j$. We define a geometric instance, or *configuration* as a set of values representing the poses of all objects, and the configurations of both manipulators (denoted by $\boldsymbol{c}_\ell$ and $\boldsymbol{c}_r$):

$$c = \{\boldsymbol{p}_1, \gamma_1, \ldots, \boldsymbol{p}_m, \gamma_m, \boldsymbol{c}_\ell, \boldsymbol{c}_r\}. \quad (1)$$

Symbolic actions on objects are meant to be applied using the robotic system Justin (Ott et al., 2006). At the $j^{th}$ symbolic state, the pose of the TCP of the right (resp. left) manipulator of Justin will be denoted by $(\boldsymbol{r}^{(j)}, \gamma_r^{(j)})$ (resp. $(\boldsymbol{\ell}^{(j)}, \gamma_\ell^{(j)})$). For example, consider the following sequence of `grasp` ($G$) and `place` ($P$) actions:

| states | $S_o$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ |
|---|---|---|---|---|---|---|---|
| actions | | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
| object $o_1$ | | $G_1^{(1)}$ | $P_1^{(2)}$ | · | · | · | · |
| object $o_2$ | | · | · | $G_2^{(3)}$ | · | $P_2^{(5)}$ | · |
| object $o_3$ | | · | · | · | $G_3^{(4)}$ | · | $P_3^{(6)}$ |
| R. hand | | ✓ | ✓ | ✓ | · | ✓ | · |
| L. hand | | · | · | · | ✓ | · | ✓ |

As a result of the first symbolic action $A_1 = G_1^{(1)}$, $o_1$ is grasped by the right hand, resulting in the first symbolic state. Note that $(\boldsymbol{p}_1^{(1)}, \gamma_1^{(1)}) = (\boldsymbol{p}_1^{(0)}, \gamma_1^{(0)})$ as the object is not yet moved. The second action $A_2 = P_1^{(2)}$ places $o_1$ at $(\boldsymbol{p}_1^{(2)}, \gamma_1^{(2)}) \neq (\boldsymbol{p}_1^{(1)}, \gamma_1^{(1)})$ since the object has been moved. During the remaining stages $o_1$ is not acted upon, and hence its pose remains unchanged. At the $4^{th}$ symbolic state, Justin has grasped $o_2$ and $o_3$ in its right and left hand, respectively, and placed them, during actions $A_5$ and $A_6$, at positions $\boldsymbol{p}_2^{(5)}$ and $\boldsymbol{p}_3^{(6)}$, with orientation $\gamma_2^{(5)}$ and $\gamma_3^{(6)}$, respectively.

## V. Generating the constraints

This section describes the constraints which are generated from the symbolic plan (task constraints) and from the geometric model of the robot (kinematic constraints). We emphasize here that these constraints only refers to what is happening *at grasp and release positions*, not during the motions in between.

### A. Grasp Constraints $\mathcal{C}_G$

Grasp constraints represent the possible relative positions of the TCP with respect to the object when the object is grasped (or released). In case of a top-grasp (or bottom-grasp) the TCP remains in a small region situated roughly above (or below) the object (see Fig. 6). In case of a side-grasp, the TCP is situated along a circle centered on the $z$ axis of the object, which can be bounded by a square region using linear constraints. Note that alternative grasp types can be handled, provided that the grasp constraints associated with each object of interest define a polyhedral region. Being inside such a region does not guarantee a feasible grasp (there may be collisions between the fingers and the object). However, being *outside* guarantees that the grasp cannot be executed, and allow us to safely prune out such configurations.

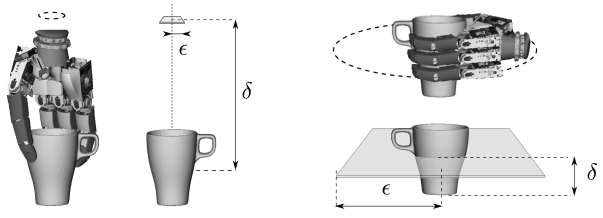


Fig. 6. Grasp constraints for a top-grasp (left) and a side-grasp (right).

Using the notation in Fig. 4, grasp constraints can be formulated using two parameters:

- $\delta$, the distance between the TCP and the plane $(u, v)$;
- $\epsilon$, the orthogonal distance between the TCP and $w$.

The grasp constraint in the frame of the object at state $s_j$ can be written as a linear inequality (for the right TCP):

$$\boldsymbol{A}\boldsymbol{r}^{(j)} \leq \boldsymbol{b}^{(j)}$$

where

$$\boldsymbol{A} = \begin{bmatrix} I \\ -I \end{bmatrix} \quad \boldsymbol{b}^{(j)} = (\epsilon, \epsilon, \delta, \epsilon, \epsilon, -\delta).$$

In order to express this constraint in the world frame, it has to be transformed according to the translation $\boldsymbol{p}_i^{(j)}$ and the rotation matrix $\boldsymbol{Q}_i^{(j)}$ of the object $o_i$:

$$\boldsymbol{A}\boldsymbol{Q}_i^T(\boldsymbol{r}^{(j)} - \boldsymbol{p}_i^{(j)}) \leq \boldsymbol{b}^{(j)}.$$

Considering Fig. 6, we observe that the polyhedral region representing the grasp constraint does not depend on the exact orientation of the object, but rather on the direction of its main axis. Hence, we can replace $\boldsymbol{Q}_i^{(j)}$ by the constant $\boldsymbol{O}_p^{(j)}$ (the object template frame) without changing the constraint. The constraint can then be written:

$$\boldsymbol{A}\boldsymbol{O}_p^{(j)T}(\boldsymbol{r}^{(j)} - \boldsymbol{p}_i^{(j)}) \leq \boldsymbol{b}^{(j)}.$$

In the case of non-cylindrical grasps, for example, we can deal with grasp constraints related to an object, if they define a polyhedral region for the TCP around the object. Consider for instance grasping a cup by the handle: one could use the same template frame as for a side-grasp (albeit using different finger configurations) and a larger $\epsilon$ value, although the constraint would be relaxed compared to a top-grasp, because the corresponding polyhedral region is larger. However, if the grasp-frame cannot be represented as a rotation of one of the TCP template frames about one of the reference axes, *kinematic constraints* cannot be formulated, because no associated kinematic map can be used (see section V-E).

### B. Transfer Constraints $\mathcal{C}_T$

Transfer constraints occur each time a `place`-like action is executed. When an object is transferred between state $s_j$ and state $s_q$, the object template frame and the angle of rotation (possibly) change. A new grasp constraint is written for state $s_q$ in order to represent this:

$$\boldsymbol{A}\boldsymbol{O}_p^{(q)T}(\boldsymbol{r}^{(q)} - \boldsymbol{p}_i^{(q)}) \leq \boldsymbol{b}^{(q)}.$$

In addition to this, we can express the fact that the object has rotated by the same amount as the TCP:

$$\gamma_r^{(q)} - \gamma_r^{(j)} = \gamma_i^{(q)} - \gamma_i^{(j)}, \tag{2}$$

where $j$ denotes the state index during which $o_i$ was grasped prior to its release during state $s_q$. In order for this relation to remain consistent even when the axis of rotation changes between state $j$ and state $q$, object template frames and TCP template frames must be constructed consistently, meaning that the same rotation is applied to all template frames when

changing axis. For instance, when changing from the axis $(0, 0, 1)$ to the axis $(0, 1, 0)$, one always uses the rotation of angle $\pi/2$ about axis $(0, 0, 1)$, although other rotations are possible.

### C. Placement Constraints $\mathcal{C}_P$

This constraint represents the set of all possible relative stable positions of a manipulable object at/in a fixed location. This constraint applies to the translation part of the object pose $\boldsymbol{p}_i$. We assume a flat surface, but the formulation can be easily generalized to a volume.

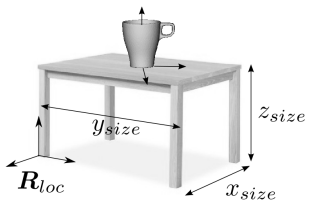


Fig. 7. Placement constraint for the action `place right top table Oz1 cup1`

Assuming that the sizes of objects are known, we can formulate the following constraint in the frame of the location:

$$\boldsymbol{A}\boldsymbol{p}_i^{(j)} \leq \boldsymbol{c}_{loc}$$

where

$$\boldsymbol{c}_{loc} = (0, y_{size}, z_{size}, x_{size}, 0, -z_{size}).$$

The values of the parameters used in $\boldsymbol{c}_{loc}$ may change according to the object template frame used. For instance, if the cup is upside-down, $z_{size}$ will be automatically replaced by $z_{size} + cup.z_{size}$. In order to get this constraint expressed in the world frame, it has to be transformed according to the translation $\boldsymbol{t}_{loc}$ and the rotation matrix $\boldsymbol{R}_{loc}$ of the location:

$$\boldsymbol{A}\boldsymbol{R}_{loc}^T(\boldsymbol{p}^{(j)} - \boldsymbol{t}_{loc}) \leq \boldsymbol{c}_{loc}.$$

### D. Stack Constraints $\mathcal{C}_S$

A stack constraint is similar to a placement constraint, except that the region where the object is allowed to be in is reduced to a single point. For some classes of objects, a constraint on the orientation is added because the objects need to have the same orientation in order to be stacked:

$$\gamma_1^{(q)} = \gamma_2^{(j)}, \tag{3}$$

where $q$ denotes the last state index during which $o_1$ was manipulated, and $j$ is the state index during which $o_2$ is stacked on $o_1$. Specifying constraints involving equality of angular values introduces a problem related the periodicity of angles. Hence, equation (2) is not sufficient to represent all the values that satisfy the stack constraint. This problem is addressed in detail in section VI-D .

### E. Kinematic Constraints $\mathcal{C}_K$

These constraints are the core of our approach. They are important for manipulation tasks because they express the relationship between the position of the TCP $\boldsymbol{r}$ in the workspace and its possible range of rotation. This relationship is non-linear and complex to compute. We approximate it conservatively using linear constraints. In order to find a linear approximation of these constraints, we compute a set of *kinematic maps* off-line, using a similar procedure to Zacharias et al. (2007). The workspace of the robot is discretized into a 3-dimensional grid, and for each cell, the existence of an inverse kinematic (IK) solution is tested for all possible rotations of a TCP template frame around its associated reference axis with 0.1 rads discrete steps (see Fig. 8).

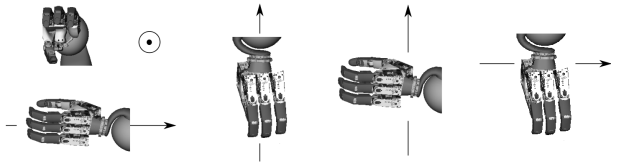

Fig. 8. Examples of left TCP template frames with their associated reference axis.

From this data, we build two maps $\gamma^{min}$ and $\gamma^{max}$, which respectively associate the position $\boldsymbol{r}$ of the TCP to a lower and upper bound for $\gamma_r$. This procedure is repeated for each pair (TCP template frames, reference axis), so that each such pair is associated with two maps:

$$\gamma^{min}(\boldsymbol{R}_p, \boldsymbol{u}_k) : (r_x, r_y, r_z) \mapsto \gamma_r^{min}$$
$$\gamma^{max}(\boldsymbol{R}_p, \boldsymbol{u}_k) : (r_x, r_y, r_z) \mapsto \gamma_r^{max}.$$

These maps tell us that if $\gamma_r \in \left[\gamma_r^{min}, \gamma_r^{max}\right]$, then $\boldsymbol{R}_p$ rotated by $\gamma_r$ about $\boldsymbol{u}_k$ *may* accept an IK solution, and that if $\gamma_r \notin \left[\gamma_r^{min}, \gamma_r^{max}\right]$, then we know *for sure* that no IK solution exists. In order to extract linear constraints from kinematic maps, we define two functions:

$$h_{max}(\underline{\boldsymbol{r}}^{(j)}, \overline{\boldsymbol{r}}^{(j)}) \rightarrow (\boldsymbol{n}_{ub}^{(j)}, m_{ub}^{(j)})$$
$$h_{min}(\underline{\boldsymbol{r}}^{(j)}, \overline{\boldsymbol{r}}^{(j)}) \rightarrow (\boldsymbol{n}_{lb}^{(j)}, m_{lb}^{(j)}),$$

where $\underline{\boldsymbol{r}}^{(j)}$ and $\overline{\boldsymbol{r}}^{(j)}$ are respectively a lower and upper bound for the variables $r_x^{(j)}, r_y^{(j)}$ and $r_z^{(j)}$, i.e., a region of space for which we want to approximate these constraints. This linear approximation remains correct only for small regions of space, but it is sufficient to approximate the kinematic constraints at grasps and release positions, because the position of the TCP is then bounded by a grasp constraint. The bounds ($\underline{r_x}$ and $\overline{r_x}$ in Fig. 9) are used to select a subset of points in $\gamma_{max}$ and $\gamma_{min}$, from which a linear regression is used in order to identify the unit normals $(\boldsymbol{n}_{ub}^{(j)}, \boldsymbol{n}_{lb}^{(j)})$ and offsets $(m_{ub}^{(j)}, m_{lb}^{(j)})$ of two bounding hyperplanes (see Fig. 9). Then, these parameters are used to formulate a constraint which gives the range of possible rotation of the TCP during an action $A_j$:

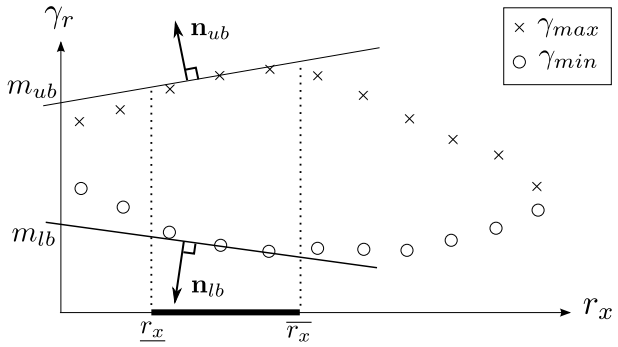


Fig. 9. Schematic 2-d view of the 4-dimensional linear outer approximations of a kinematic map by the functions $h_{min}$ and $h_{max}$

$$\begin{bmatrix} \boldsymbol{n}_{lb}^{(j)\mathrm{T}} & m_{lb} \end{bmatrix} \begin{bmatrix} \boldsymbol{r}^{(j)} \\ 1 \end{bmatrix} \;\leq\; \gamma_r^{(j)} \;\leq\; \begin{bmatrix} \boldsymbol{n}_{ub}^{(j)\mathrm{T}} & m_{ub} \end{bmatrix} \begin{bmatrix} \boldsymbol{r}^{(j)} \\ 1 \end{bmatrix},$$

which we compactly denote by:

$$f_{min}^{(j)}(\boldsymbol{r}^{(j)}) \;\leq\; \gamma_r^{(j)} \;\leq\; f_{max}^{(j)}(\boldsymbol{r}^{(j)}).$$

Note that we assume that the map is smooth between the sample points of the grid. If this is not the case, some solutions may be ruled out. However, for the experiments conducted in this paper, the maps were built with 5 cm spatial resolution and 0.1 rads angular resolution, and no such cases were observed.

### F. Other types of constraints

So far, we have introduced five types of constraints which are used in most common manipulation tasks. We could think of other constraints for other types of action like screwing / unscrewing, opening a door, or turning a knob. Similarly for the placement constraints and the stack constraints introduced in the previous sections, we presented one possible way of modeling them, but one could think of a more elaborate model in which, e.g., the object and its location are not aligned along the same axis. The only requirement is that these constraints should model the *result* of the action, not what happens *during* the action. Complex actions can be modeled more precisely by splitting them into sub-actions. Pouring a liquid into a recipient is an example of action which requires such splitting. A brief description is given in section VII-C.

### G. Example of constraint generation

An important feature of our scheme is that all the constraints described above are generated *automatically* from a given sequence of symbolic actions (c.f. Fig. 2). Next, we show how this can be done through an example. Consider for instance the following two-actions sequence:

```
pick right_hand top Oz1 cup1
place right_hand top table Oz1 cup1
```

This action sequence results in three states: the initial state $s_0$, the state resulting from the first action $s_1$, and the state resulting from the second action $s_2$. Six variables are automatically created which represent the poses of objects and manipulators. In this example, the pose of a cup is denoted by $\boldsymbol{p}_{cup}^{(j)}, \gamma_{cup}^{(j)}$, and the pose of a right manipulator by $\boldsymbol{r}^{(j)}, \gamma_r^{(j)}$, where $j$ is the index of the state. After the `pick` action, only the manipulator has moved, so no variables are created for the cup. After the `place` action, both the manipulator and the object have moved, so two sets of variables are created. All the constraints of the problem are formulated with these variables:

| state | variables | | | |
|---|---|---|---|---|
| $s_0$ | $\boldsymbol{p}_{cup}^{(0)}$ | $\gamma_{cup}^{(0)}$ | | |
| $s_1$ | | | $\boldsymbol{r}^{(1)}$ | $\gamma_r^{(1)}$ |
| $s_2$ | $\boldsymbol{p}_{cup}^{(2)}$ | $\gamma_{cup}^{(2)}$ | $\boldsymbol{r}^{(2)}$ | $\gamma_r^{(2)}$ |

The `pick` action leads to the creation of a grasp constraint and a kinematic constraint:

$$\begin{aligned} &\boldsymbol{A}\boldsymbol{O}_p^{(1)T}(\boldsymbol{r}^{(1)} - \boldsymbol{p}_{cup}^{(0)}) \leq \boldsymbol{b}^{(1)} \\ &f_{min}^{(1)}(\boldsymbol{r}^{(1)}) \;\leq\; \gamma_r^{(1)} \;\leq\; f_{max}^{(1)}(\boldsymbol{r}^{(1)}), \end{aligned}$$

where $\boldsymbol{A}$ is constant, the object template frame $\boldsymbol{O}_p^{(1)}$ is known from the symbolic parameters `Oz1` and `cup1`, and $\boldsymbol{b}^{(1)}$ which parametrizes the grasp region is known from the symbolic parameters `right_hand`, `top`, and `cup1`. $f_{min}$ and $f_{max}$ are the linear approximations of the kinematic constraints extracted from a kinematic map (see previous subsection). Which map to use is determined by the parameters `right_hand`, `top`, and `Oz1`.

Similarly, the `place` action automatically generates a transfer constraint (grasp constraint + angular constraint, see equation (2)), a placement constraint, and again, a kinematic constraint:

$$\begin{aligned} &\boldsymbol{A}\boldsymbol{O}_p^{(2)T}(\boldsymbol{r}^{(2)} - \boldsymbol{p}_{cup}^{(2)}) \leq \boldsymbol{b}^{(2)} \\ &\gamma_r^{(2)} - \gamma_r^{(1)} = \gamma_{cup}^{(2)} - \gamma_{cup}^{(0)} \\ &\boldsymbol{A}\boldsymbol{R}_{table}^{T}(\boldsymbol{p}_{cup}^{(2)} - \boldsymbol{t}_{table}) \leq \boldsymbol{c}_{table} \\ &f_{min}^{(2)}(\boldsymbol{r}^{(2)}) \;\leq\; \gamma_r^{(2)} \;\leq\; f_{max}^{(2)}(\boldsymbol{r}^{(2)}). \end{aligned}$$

Like for the `pick` action, $\boldsymbol{O}_p^{(2)}$ and $\boldsymbol{b}^{(2)}$ are known from the symbolic parameters of the `place` action, and $\boldsymbol{R}_{table}$, $\boldsymbol{t}_{table}$, and $\boldsymbol{c}_{table}$ are constants.

We have shown how a pick-place sequence is automatically converted into six linear constraints on 16 variables ($\boldsymbol{p}, \boldsymbol{r} \in \mathbb{R}^3$). These constraints delimit how the two symbolic actions can be geometrically instantiated. In the next section, we explain how these constraints are used to prune the search space of geometric instantiations.

## VI. Search procedure

The geometric constraints of the problem are formulated as a set of linear inequalities and equalities. The kinematic constraints $\mathcal{C}_K$ have been formulated in terms of lower and upper bounds on the actual capabilities of the manipulator. Consequently, the set of constraints is *conservative*, i.e., if

a solution exists, it must belong to the feasible set defined by the constraints. Conversely, if the constraints result in an empty feasible set, the problem has no solution, which implies that one can safely prune search at this point. Note, however, that

- we still have to search the feasible set for a sequence of geometric instances which solves the problem;
- we still have to do motion planning to find collision-free paths connecting grasp and release positions.

For these reasons, instead of searching for a single solution, we use the constraints to tighten the bounds of a set of *intervals* which contain all the solutions to the problem. We define the vector of all the variables of the problem:

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_N),$$

to which we associate a domain

$$\mathcal{D} = \langle [\underline{v_1}, \overline{v_1}], [\underline{v_2}, \overline{v_2}], \ldots, [\underline{v_N}, \overline{v_N}] \rangle,$$

where each variable $v_i$ has associated bounds $[\underline{v_i}, \overline{v_i}]$. The set of all constraints of the problem

$$\mathcal{C} = \{\mathcal{C}_G^{(j)}, \mathcal{C}_T^{(j)}, \mathcal{C}_P^{(j)}, \mathcal{C}_K^{(j)}, \mathcal{C}_S^{(j)}\},\ j \in \{1, \ldots, n\}$$

can be expressed as

$$\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d} \tag{4}$$

$$\boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}, \tag{5}$$

where (4) and (5) represent respectively the inequality and equality constraints of our problem.

### A. Narrowing intervals with the constraints

**Algorithm 1:** FilterDomain

Function **FilterDomain**$(\mathcal{D}, \mathcal{C})$

**input** : $\mathcal{D}$: a domain
$C$: a set of linear constraints

1 $\varepsilon$ = minimal domain reduction
2 $\mathcal{D}' = \mathcal{D}$
3 **repeat**
4 $\quad \mathcal{D} = \mathcal{D}'$
5 $\quad UpdateKinematicConstraints(C, \mathcal{D})$
6 $\quad$ **for** $i \leftarrow 1$ **to** $N$ **do**
7 $\quad\quad$ minimize $v_i$, subject to $\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d}, \boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}$
$\quad\quad\quad \boldsymbol{v}$
8 $\quad\quad \underline{v_i}' \leftarrow \max(\underline{v_i}, v_i^\star)$
9 $\quad\quad$ maximize $v_i$, subject to $\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d}, \boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}$
$\quad\quad\quad \boldsymbol{v}$
10 $\quad\quad \overline{v_i}' \leftarrow \min(\overline{v_i}, v_i^\star)$
11 **until** $Dist(\mathcal{D}, \mathcal{D}') \leq \varepsilon$ **or** $\mathcal{D}' = \emptyset$
12 **return** $\mathcal{D}'$

The algorithm presented in this section was inspired by the constraint satisfaction literature and bound propagation literature (Davis, 1987; Lhomme, 1993; Jaulin, 2000). Since our problem is formulated with linear constraints and is convex (see equations (4) and (5)), we can solve it using a global filtering algorithm (adapted from Lebbah et al. (2002)). The bounds of the intervals are found by solving several linear programs (LP) in order to find the minimum (resp. maximum) value $v_i^\star$ of each variable $v_i$. $v_i^\star$ is then used to update the lower (resp. upper) bound of $v_i$ (lines 8 and 10). The values are updated in a temporary copy of the domain $\mathcal{D}' = \langle [\underline{v_1}', \overline{v_1}'], [\underline{v_2}', \overline{v_2}'], \ldots, [\underline{v_N}', \overline{v_N}'] \rangle$, which is used to measure how much the intervals have shrunk after each iteration. This is done by the $Dist$ function (line 11), which returns the average of the differences between upper and lower bounds in $\mathcal{D}$ and $\mathcal{D}'$. The process is repeated until the domain does not change more than a predefined $\varepsilon$ value. The result is a domain in which the intervals are narrowed down with respect to the constraints, or $\emptyset$ if an inconsistency is detected during the resolution of an LP.

We have modified the original algorithm (Lebbah et al., 2002) by adding the function $UpdateKinematicConstraints(C, \mathcal{D})$ in the main loop (line 5). Indeed, after each iteration, the intervals may shrink. If the intervals representing the TCP positions are reduced, it is meaningful to refine the kinematic constraints using the functions $h_{min}$ and $h_{max}$ in order to get a tighter linear approximation of the real problem.

Refining the kinematic constraints while filtering the domains is a very efficient process. Let us illustrate this with a numerical example for a pick action. Initially, the problem consists of four variables representing the position of the object `cup` located at $(0.6, 0.25, 0.1)$ with orientation $0$ rad.

$$\boldsymbol{v} = (x_{cup}^{(0)},\ y_{cup}^{(0)},\ z_{cup}^{(0)},\ \gamma_{cup}^{(0)})$$
$$\mathcal{D} = \langle [0.6, 0.6],\ [0.25, 0.25],\ [0.1, 0.1],\ [0, 0] \rangle.$$

The lower bounds are equal to the upper bounds because the values of the variables are determined. The pick action leads us to the creation of 4 new variables for the TCP, which are initially assigned arbitrarily large intervals:

$$\boldsymbol{v} = (x_{cup}^{(0)},\ y_{cup}^{(0)},\ z_{cup}^{(0)},\ \gamma_{cup}^{(0)},\ r_x^{(1)},\ r_y^{(1)},\ r_z^{(1)},\ \gamma_r^{(1)})$$
$$\mathcal{D} = \langle [0.6, 0.6],\ [0.25, 0.25],\ [0.10, 0.10],\ [0, 0],$$
$$[-10, 10],\ [-10, 10],\ [-10, 10],\ [-10, 10] \rangle.$$

The pick action also generates grasp constraints $\mathcal{C}_G$ and kinematic constraints $\mathcal{C}_K$. In this example, we assume a top-grasp and an object in an upright position. The grasp constraint is:

$$r_x^{(1)} = x_{cup}^{(0)}$$
$$r_y^{(1)} = y_{cup}^{(0)}$$
$$r_z^{(1)} = z_{cup}^{(0)} + 0.34.$$

At this stage, the kinematic constraint is inaccurate, because the domain of $\boldsymbol{r}^{(1)}$ is arbitrarily large. Therefore, the best approximation is given by 2 hyperplanes with constant values equal to the max value of the map $\gamma^{max}$ and the min

value of the map $\gamma^{min}$. In this example, we assume these values to be respectively $2\pi$ and $-2\pi$, hence the kinematic constraint is:

$$0x^{(1)} + 0y^{(1)} + 0z^{(1)} - 2\pi \leq \gamma_r^{(1)} \leq$$
$$0x^{(1)} + 0y^{(1)} + 0z^{(1)} + 2\pi.$$

After the first iteration in $FilterDomain$, the grasp constraints have propagated the values of the position of the cup to the position of the TCP, and the kinematic constraints have updated the bounds of the variable $\gamma_r^{(1)}$, hence $\mathcal{D}$ becomes:

$$\mathcal{D} = \langle [0.6, 0.6],\ [0.25, 0.25],\ [0.10, 0.10],\ [0, 0],$$
$$[0.6, 0.6],\ [0.25, 0.25],\ [0.44, 0.44],\ [-2\pi, 2\pi] \rangle.$$

In the second iteration, the domain of the TCP has been reduced (to a single point), so the function $UpdateKinematicConstraints$ provides a tighter approximation of the kinematic constraints:

$$3.3x^{(1)} - 2.5y^{(1)} + 0.1z^{(1)} - 2.1 \leq \gamma_r^{(1)} \leq$$
$$4.6x^{(1)} - 1.1y^{(1)} + 2.9z^{(1)} - 1.4,$$

which results by propagation in tighter bounds for the variable $\gamma_r^{(1)}$. Finally, $\mathcal{D}$ becomes:

$$\mathcal{D} = \langle [0.6, 0.6],\ [0.25, 0.25],\ [0.10, 0.10],\ [0, 0],$$
$$[0.6, 0.6],\ [0.25, 0.25],\ [0.44, 0.44],\ [-0.7, 2.36] \rangle.$$

Hence, in order to pick the cup, the orientation of the top-grasp must be chosen between $-0.70$ and $2.36$ radians.

This constraint propagation process is interesting for longer sequences of actions, because it allows us to propagate the consequences of early choices until the final actions. It could for instance solve the stacking problem described in the introduction, by giving an approximation of a region on the dish rack which is appropriate for placing the first cup, as well as bounds for its orientation.

### *B. Narrowing down intervals during search*

In order to find a solution, we use a basic depth-first-search algorithm, endowed with a pruning step (see algorithm 2: $SearchAndFilter$ (SAF)). When a geometric instance is chosen (lines 4 and 5 in Algorithm 2), we check if the variables representing it belong to their respective intervals (line 6): this is the first level of pruning. But after an action has been chosen (e.g., to place the cup at position $(0.7, -0.25, 0.1)$ with $\gamma = \pi/2$), the variables representing this choice are assigned fixed values, so the corresponding intervals can be reduced to single points (i.e., for a variable $v_i$, $\underline{v_i} = \overline{v_i}$). Then, we can filter the domains *again*, i.e., we propagate this choice to other variables through the constraints. The other intervals will be shrunk accordingly, which reduces even more the possibilities for further actions. This process is repeated each time an action is instantiated, so that intervals are shrunk as the search progresses.

**Algorithm 2:** SearchAndFilter

Function **SearchAndFilter**$(c_1, Seq, \mathcal{D}, \mathcal{C})$

**input** : $c_1$: a configuration
$Seq$: a sequence of symbolic actions
$\mathcal{D}$: a domain
$\mathcal{C}$: a set of linear constraints

```
1  if Seq = ⟨⟩ then return c_1
2  Action = Seq.head
3  Rest = Seq.tail
4  foreach a_k ∈ geometricInstanceOf (Action) do
5      c_2 = getSuccessorConf(c_1, a_k)
6      if c_2 ∈ D then
7          D' = assignValues(D, c_2)
8          D' = FilterDomain(D', C)
9          if D' ≠ ∅ then
10             feasible = pathPlanning(c_1, c_2)
11             if feasible then
12                 s = SearchAndFilter(c_2, Rest, D')
13                 if s ≠ false then
14                     return ⟨c_2, s⟩
15 return false
```

Algorithm 2 is initially called with the initial configuration, the sequence of symbolic actions, and the initial domain filtered according to the constraints of the problem. An action $a_k$ is chosen among the possible geometric instances of the symbolic action $Action$. $c_2$ is the result of applying $a_k$ to $c_1$. If this configuration belongs to the domain, we apply the strategy described above that assigns the values to the domain and filters it again (lines 7 and 8). If no inconsistency appears, the motion planning algorithm is called to check if a collision-free path exists to reach $c_2$. If a path exists, the function is recursively called on $c_2$ with the remaining actions and the shrunk domain $\mathcal{D}'$, otherwise the next action $a_{k+1}$ is tried. Note that when the configuration $c_2$ is computed, a choice is possible among the possible inverse kinematic solutions of the manipulator. We choose the configuration which is the closest to $c_1$ (with euclidean distance), in order to ease RRT search, but we do not backtrack on this choice, which may result in loss of solutions. If all the actions fail, the function returns $false$ to the calling function via the return statement line 15. If a final configuration is reached (line 1), the solution is incrementally built (line 14) and returned to the main calling function. The result is a list of configurations and paths which are used to execute the final plan (paths are smoothed after a plan is found, see end of section VIII-B).

Note that using a conservative filtering algorithm does not imply that Algorithm 2 is complete. Completeness is lost at different decision points: (i) the choice of a geometric instance $a_k$ for an action is subject to a fixed resolution, (ii) the choice of an inverse kinematic solution is greedy, and (iii) the RRT path planner is itself incomplete.

### C. Detecting inconsistency and pruning

One of the main problems of geometric backtrack search is when no geometric instantiation of the action sequence exists. This happens often during task planning, because no geometric information is used. For instance, the task planner may try a sequence in which the right arm of the robot should grasp an object situated on the left side. If this sequence is in fact infeasible, all the space of configurations has to be explored in order to assess it, which may be computationally expensive. The only solution to avoid this is to impose a time limit on the backtracking process. Unfortunately by doing this, some solutions are lost as the time limit may be exceeded in cases in which the problem *is* feasible.

With our approach however, inconsistency can be detected *before* entering the backtracking procedure, while we filter the initial domain according to the constraints of the problem. This is more efficient, since no search is required. Inconsistency can also be exploited *during* search. Suppose for instance that the problem is initially consistent, and at some point in the search, a configuration is chosen that makes the problem inconsistent. This will be detected during filtering (lines 8 and 9 in Algorithm 2). In this case, we can stop exploring this branch of the search tree, and directly try the next configuration.

### D. Dealing with the periodicity of angles

The periodicity of angular values introduces a problem for constraints involving angular values, namely constraints on the final orientations of objects, and stack constraints.

*1) Final orientation constraint:* In some problems, such as setting the table, we may impose a constraint on the final orientations of some objects (e.g., the cutlery). To do this, we do not need to add a new constraint to the problem, but we simply set the bounds of the variable representing the orientation of the object to the desired final value. For an object $o_i$, we simply modify the domain as follow:

$$\left[\gamma_i^{(f_i)}, \gamma_i^{(f_i)}\right] = [\Gamma_{goal}, \Gamma_{goal}],$$

where $f_i$ is the index of the last state when $o_i$ was moved, and $\Gamma_{goal} \in [-\pi, \pi]$ is the desired orientation. The problem is that after several manipulations, the object can be rotated by more than one turn, and the variable $\gamma_i^{(f_i)}$ can reach a value out of the interval $[-\pi, \pi]$ . In this case, it may happen that the constraint network is inconsistent, while $\gamma_i^{(f_i)}$ is a valid solution modulo $2\pi$, which means that some solutions may be lost. Suppose for instance, a cup initially oriented with angle $\gamma_{cup}^{(0)} = 2.5$. Then, this object is picked and placed with a constraint on its final orientation $\gamma_{cup}^{(2)} = 2.5$. The transfer constraint is

$$\gamma_r^{(2)} - \gamma_r^{(1)} = \gamma_{cup}^{(2)} - \gamma_{cup}^{(0)}$$

Suppose that $\gamma_r^{(2)}$ and $\gamma_r^{(1)}$ are constant:

$$\gamma_r^{(2)} = 1.3 + 2\pi$$
$$\gamma_r^{(1)} = 1.3$$

In this example, a solution exists modulo $2\pi$, but the system is not consistent. One may argue that the wraparound problem can be easily fixed by normalizing all values to the range $[-\pi, \pi]$, but this is not possible because $\gamma_r^{(2)}$ and $\gamma_r^{(1)}$ are not constant values: they are variables bounded by two linear functions of the position of the TCP in space (see the kinematic constraints, section V-E), for instance:

$$ax^{(2)} + by^{(2)} + cz^{(2)} + d \leq \gamma_r^{(2)} \leq$$
$$ex^{(2)} + fy^{(2)} + gz^{(2)} + h$$

Hence, the variables $\gamma_r^{(2)}$ and $\gamma_r^{(1)}$ depend on the values computed in the kinematic maps, which happen to be out of the range $[-\pi, \pi]$ by construction. Unfortunately, the maps cannot be normalized because that would break their continuity, therefore making the linear approximations inaccurate.

To fix this problem, we make the following assumption: *In its final pose, an object can be at most rotated by one extra-turn*, i.e., we only deal with orientations in the interval $[-3\pi, 3\pi]$, which is sufficient in most cases. Then, we fix the problem by solving three problems instead of one, which covers all the cases defined by our assumption. This is done by creating two copies $\mathcal{D}_1$ and $\mathcal{D}_2$ of the original domain $\mathcal{D}$ and modifying the bounds of the variable $\gamma_i^{(f_i)}$:

$$\mathcal{D}_1 = \langle [\underline{v_1}, \overline{v_1}], \ldots, [\Gamma_{goal} - 2\pi, \Gamma_{goal} - 2\pi], \ldots, [\underline{v_N}, \overline{v_N}] \rangle$$
$$\mathcal{D} = \langle [\underline{v_1}, \overline{v_1}], \ldots, [\Gamma_{goal}, \Gamma_{goal}], \ldots\ldots\ldots\ldots, [\underline{v_N}, \overline{v_N}] \rangle$$
$$\mathcal{D}_2 = \langle [\underline{v_1}, \overline{v_1}], \ldots, [\Gamma_{goal} + 2\pi, \Gamma_{goal} + 2\pi], \ldots, [\underline{v_N}, \overline{v_N}] \rangle,$$

To deal with several domains, Algorithm 2 was not modified, but we replaced the function *filterDomain()* (line 8) by the function *FilterDomainList()* which is called with a list of domains instead of a single domain as input parameter, and returns a list of filtered domains. Inconsistent domains are removed from the list (see Algorithm 3).

**Algorithm 3:** FilterDomainList

Function **FilterDomainList**($\mathcal{L}, \mathcal{C}$)

**input** : $\mathcal{L}$: a list of domains
$\mathcal{C}$: a set of linear constraints

1 $\mathcal{L}' = \{\emptyset\}$
2 **foreach** $\mathcal{D} \in \mathcal{L}$ **do**
3 $\quad \mathcal{D}' = filterDomain(\mathcal{D}, \mathcal{C})$
4 $\quad$ **if** $\mathcal{D}' \neq \emptyset$ **then**
5 $\quad\quad \mathcal{L}' \leftarrow \mathcal{L}' \cup \mathcal{D}'$
6 **return** $\mathcal{L}'$

*2) Stack constraints:* A similar problem arises with the stack constraints (3):

$$\gamma_1^{(q)} = \gamma_2^{(j)}.$$

Since our assumption allows for orientations in the interval $[-3\pi, 3\pi]$, it can be the case where one of the objects, say object $o_1$, is oriented with a value in $[-3\pi, -\pi]$, $[-\pi, \pi]$, or $[\pi, 3\pi]$. At the constraint level, variables are not instantiated, so we have to consider the three possibilities. Hence, the constraint is modified as follow:

$$\gamma_1^{(q)} = \gamma_2^{(j)} + k2\pi,$$

and introduce a new variable $k$ in the domain. Then, we create two copies of the existing domains (as we did for final orientations constraints), and assign the values $-1, 0$, and $1$ in each copy. Note that, in theory, the same problem can occur for transfer constraints, when extreme regions of the kinematic maps are used for the linear approximation. But this rarely occurs in practice, and when it does, it is often the case that the resulting extra-turn is "absorbed" by the final orientation constraint or by a stack constraint with the technique described above (provided that the variable has not left the range $[-3\pi, 3\pi]$).

In terms of complexity, it is important to see that each such constraint does not increase the number of domains by 3, but *multiplies* it by 3, which can quickly lead to a large number of domains. In practice however, many of the domains created in this way are not consistent, so that they are eliminated by the filtering process.

## VII. Analysis of the search space

In this section, we analyze the time complexity of our approach, by analyzing the complexity of its individuals components, namely Algorithm 1 and Algorithm 2. Then, we qualitatively analyze how our pruning scheme takes advantage of the structure of the problem.

### A. Branching factor

The main algorithm (Algorithm 2) is a depth-first search algorithm with pruning. The amount of pruning cannot be easily analyzed because it depends on each problem instance. But the branching factor of the search tree can be directly determined: it depends on the resolution used for discretizing actions.

| action | variables | branching factor |
|---|---|---|
| `pick` `stack` | $\theta$ | 10 |
| `place` | $x, y, \theta$ | 500 |
| `regrasp` | $x, y, z, \theta$ | 2000 |

TABLE I

| | test | duration |
|---|---|---|
| 1 | consistency | $0 - 4$ ms |
| 2 | IK | $< 1$ ms |
| 3 | collision | $< 1$ ms |
| 4 | path | $\approx 0.2$ s |

TABLE II

Table I gives an order of magnitude of the resolutions used for different actions. For instance, `pick/stack` actions are only sampled according to the orientation $\theta$ of the TCP/object. Assuming that a `place` action occurs on a flat surface, three parameters are needed to sample the position $(x, y)$ and the orientation $\theta$ of the object. A `regrasp` action is similar, but a volume is sampled instead. In order to instantiate an action, the four tests described in Table II are done in sequence before validating the configuration. This occurs when the function $getSuccessorConf()$ is called (line 5 in Algorithm 2). If one of the tests fails, another configuration is sampled and evaluated. If none of the samples can be validated, geometric backtracking occurs.

### B. Complexity of Algorithm 2

The time complexity of Algorithm 2 results from the interaction of three components:

(i) Algorithm 2, which is exponential with the depth of the search (the number of actions);
(ii) the number of domains, which may grow exponentially because of the periodicity of angles;
(iii) Algorithm 1, which calls $2N$ times the simplex algorithm, where $N$ is the number of variables.

The simplex algorithm's worst case complexity is exponential with the number of variables. Hence, the worst-case complexity of our approach is

$$\mathcal{O}(\underbrace{b^n}_{(i)} \cdot \underbrace{3^{(n_f+n_s)}}_{(ii)} \cdot \underbrace{N \cdot exp(N)}_{(iii)}),$$

where $n$ is the total number of actions, $n_f$ is the number of final orientation constraints, $n_s$ is the number of `stack` actions, $b$ is an upper bound on the branching factor, and $N$ is the number of variables. However, studying the worst-case complexity of our algorithm does not give a fair insight into its performance in practice because:

(i) the depth-first-search algorithm may find a solution with few backtracks if pruning is efficient;
(ii) the multiplication of the number of domains is only for specific tasks, and is balanced by the filtering procedure;
(iii) the simplex algorithm behaves polynomially in most practical cases (Megiddo, 1987).

In section VII, we empirically show that in practice, the computational cost of the simplex algorithm grows *linearly* with the number of variables. We also observe that the cost of components (i) and (ii) grows exponentially with the number of actions. However, we show with three different tasks involving the same number of actions that the critical parameter affecting our algorithm is the *structure* of the problem rather than the number of actions. Next, we introduce some concepts used to characterize this structure.

### C. Problem structure

We base our analysis on the complexity analysis of constraint satisfaction problems (CSPs), in which the problem structure is characterized by the topology of its constraint network (Dechter, 2006). A constraint network can be represented by its primal-constraint graph or its dual-constraint

graph (see Fig. 10). In the former, variables are represented by nodes and arcs associate any two nodes residing in the same constraint. In the latter, each node contains the variables associated to a constraint, and arcs are labeled by the variables that any two constraints share.

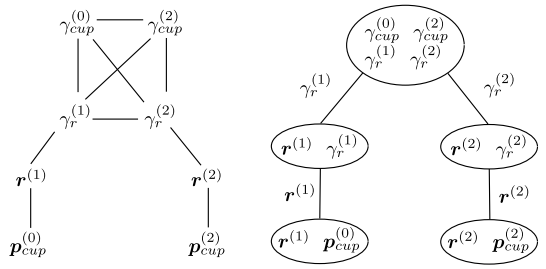

Fig. 10. The primal-constraint graph (left) and dual-constraint graph (right) for the pick-and-place task (see section III.F). The node on the top (right) represents the transfer constraint, then the two kinematic constraints, and at the bottom the two grasp constraints (we do not show placement constraints because they do not change the topology of the graph).

This graphical representation gives an insight of the geometric dependencies between actions. Consider for instance a sequence of actions where an object is grasped with one arm, handed over to the other arm, and placed somewhere, as illustrated in Fig. 3. This sequence results in the dual-constraint graph shown in Fig. 11.

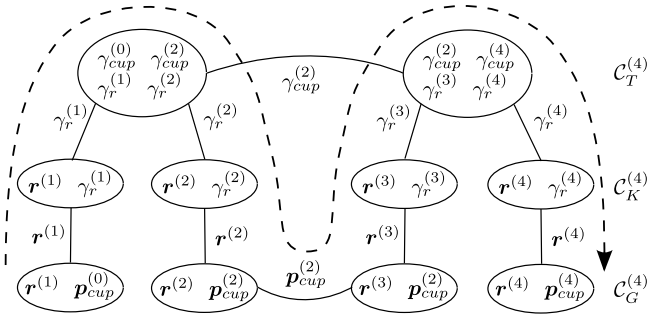

Fig. 11. Dual-constraint graph the task `pick-place-regrasp-place` (see illustration Fig. 3.) The dashed arrow roughly represents the order of instantiation.

The graph consists of two "`pick-place`" sub-graphs (see Fig. 10), joined through the variables $\boldsymbol{p}_{cup}^{(2)}$ and $\gamma_{cup}^{(2)}$. $\gamma_{cup}^{(2)}$ is shared by the two transfer constraints, denoting the fact that the same object is rotated two times. $\boldsymbol{p}_{cup}^{(2)}$ is shared by the two grasp constraints, meaning that the intermediate position chosen for re-grasping must satisfy the grasp constraints for both arms. Recall that Algorithm 2 is a depth-first-search algorithm, hence the variables are instantiated chronologically (see dashed arrow in Fig. 11). Imagine that a final position for the object was imposed, and that the first two actions have been instantiated. Then, the variables $\boldsymbol{p}_{cup}^{(2)}$, $\gamma_{cup}^{(2)}$, $\boldsymbol{p}_{cup}^{(4)}$, and $\gamma_{cup}^{(4)}$ are assigned a value. The problem can then be solved without backtracking, by trying all possible instances of the second pick action (parametrized by $\gamma_r^{(3)}$). Indeed, if the variable $\gamma_r^{(3)}$ is instantiated, $\gamma_r^{(4)}$ is automatically found by the propagation of the transfer constraint $\mathcal{C}_T^{(4)}$. Once $\gamma_r^{(4)}$ is instantiated, a value for $\boldsymbol{r}^{(4)}$ is easily found or rejected because it is involved in two constraints ($\mathcal{C}_K^{(4)}$ and $\mathcal{C}_G^{(4)}$) in which all the remaining variables are instantiated.

This example shows how action dependencies can be exploited by our algorithm, i.e., by using previous instantiations in order to reduce the choice for instantiating the remaining variables. This is only possible if the problem structure allows for it. Note that by knowing the dependencies between variables, one could decide on which variables to backtrack in priority (technique known as *variable ordering* in the CSP literature, see for instance Smith (1995)). This technique is not effective here, because backtracking in a non-chronological order implies to recompute all the subsequent motion paths, which may be invalidated by changing the configuration of obstacles. In the next section, we evaluate our approach on three problems which have different structures: pick-and-place, filling a glass, and stacking cups. Next, we give the dual-constraint graphs corresponding to these tasks.

*1) Pick-and-place (Fig. 12):* The task consists in moving three cups from the table to a dish rack with the left arm (Fig. 12).

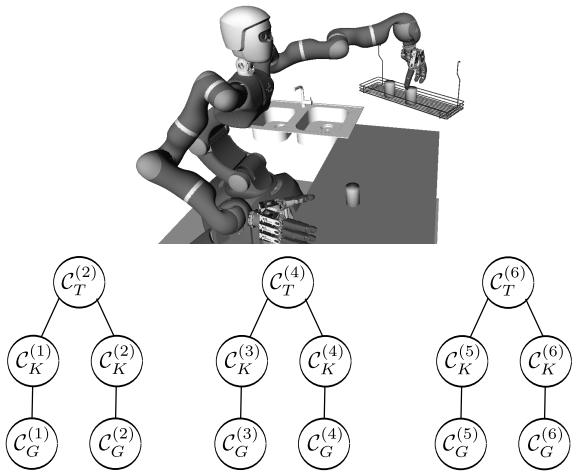

Fig. 12. Dual-constraint graph for a pick-and-place task with three objects.

The graph consists of three disconnected `pick-place` sub-graphs (The variables are not explicitly represented for clarity). There are local dependencies within each `pick-place` pair, but the sub-graphs do not share any variables because a different object is manipulated each time.

*2) Filling a glass (Fig. 13):* The task consists in 6 actions: (1) picking a glass with the left arm, (2) placing it on the table, (3) moving the left arm away, (4) picking a bottle with the right arm, (5) placing the bottle in a vertical position from which pouring can be done, and (6) rotating by 90 degrees the right TCP about a predefined axis (the x-axis in this case, see Fig. 12).

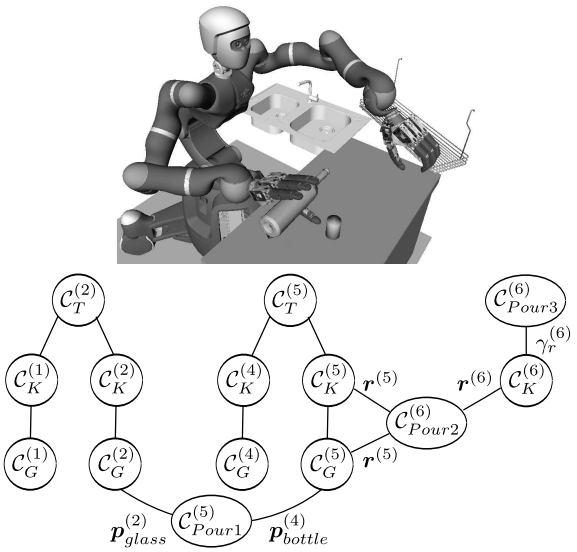


Fig. 13. Dual-constraint graph for the fill-glass task.

We observe two `pick-place` sub-graphs (one for the glass, one for the bottle), with a dependency between them due to the constraints generated by the `pour` action. In a nutshell: $\mathcal{C}_{Pour1}^{(5)}$ imposes a constraint on the relative position of the glass and the bottle before pouring (the bottle is vertical), $\mathcal{C}_{Pour2}^{(6)}$ says that the position of the TCP does not change during pouring, and $\mathcal{C}_{Pour3}^{(6)}$ says that the TCP reaches a certain orientation at the end of pouring (the bottle is horizontal). The difficulty of this task is to find a position for the glass which is reachable by the left arm *and* from which the right arm can pour in.

*3) Stacking cups (Fig. 14):* This task consists in moving a cup on the table, then stacking the two other cups on it such that all the cups have the same orientation.

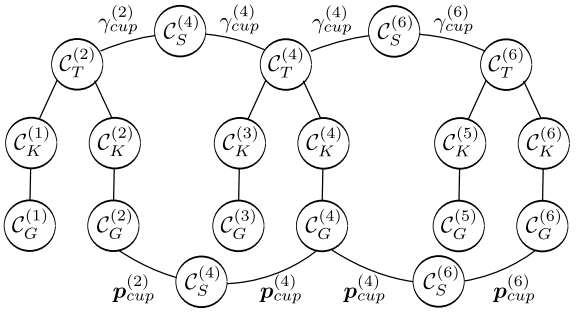


Fig. 14. Dual-constraint graph for a stacking task with three objects.

We observe three `pick-place` sub-graphs, connected by the stack constraints ($\mathcal{C}_S$) restricting the positions ($\boldsymbol{p}_{cup}^{(2)}$, $\boldsymbol{p}_{cup}^{(4)}$, $\boldsymbol{p}_{cup}^{(6)}$) and orientations ($\gamma_r^{(2)}$, $\gamma_r^{(4)}$, $\gamma_r^{(6)}$) of the three cups. Since these are only equality constraints (see section V-D), when the pose of one cup is instantiated, the pose of the other cups is exactly determined by the stack constraints. Hence, this task is highly constrained, but after instantiating the pose of the first cup, we might be able to detect a violation of kinematic constraints for placing the last cup, thanks to the constraints $\mathcal{C}_K^{(6)}$ and $\mathcal{C}_G^{(6)}$.

These three tasks display constraint networks of increasing intricacy. Since our approach takes advantage of such dependencies to reduce the search space, we expect Algorithm 2 to perform increasingly better on these three tasks. For the "Stacking cups" task however, the reduction of the search space is balanced by the presence of 2 `stack` actions, which causes a multiplication of the number of domains.

## VIII. Experimental evaluation

We use a simulation environment provided by DLR[2] for the robotic platform Justin. Justin is a humanoid robot with two arms with 7 DoF each, and two dexterous hands. The robot is situated in front of a table, on which are placed objects that can be manipulated. The algorithms are implemented in Java, and all the experiments are conducted on a computer with a dual-core processor (Intel Core i7, 2.66 GHz).

### A. Evaluation of Algorithm 1

The linear programs were solved with Gurobi (Gurobi Optimization, Inc., 2013). We measured the average filtering time for linear programs of increasing size. The number of variables was increased by generating linear programs from action sequences of increasing size, i.e., `<pick>`, `<pick, place>`, `<pick, place, pick>`,... until 10 actions (five cups manipulated). We also compared between two kind of tasks: Stacking cups and Pick-and-Place (see Fig. 15). The number variables increases by 4 for a `pick` action, 8 for a `place` action, and 9 for a stack action, which explains that the curve representing the stacking task gets further on the right. The number of variables starts at value 28, because 7 manipulable objects where present in the scene, and four variables are used to represent their initial positions. The number of constraints is roughly proportional to the number of actions. By way of example, a sequence of 5 pick-and-place actions lead to 84 variables and 115 constraints.

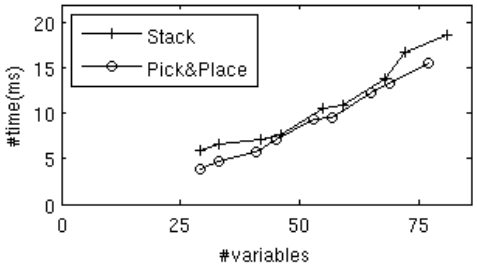


Fig. 15. Average filtering time as a function of the number of variables.

We observe that the filtering time increases linearly with the number of variables. In case of stacking, filtering takes slightly more time, which is due to the use of a larger number of equality constraints of type (3) compared to a `place` action. Note that these results cannot be generalized

[2]Deutches Zentrum für Luft und Raumfahrt (German Aerospace Center)

to problems involving constraints different from the ones used in this work, although it is known that the simplex algorithm is efficient for most distributions of problems.

### B. Evaluation of Algorithm 2

We evaluated the algorithm SearchAndFilter (SAF) by comparing it to a plain depth-first-search (DFS) procedure, i.e., SAF without the filtering process. In this way, we make sure that the differences observed in the results are only due to the filtering process. We ran SAF on the three tasks introduced in the previous section: pick-and-place, filling a glass, and stacking cups. These scenarios present an increasing difficulty in terms of kinematic dependencies, but they are not challenging in terms of collisions i.e., the workspace is not cluttered with occluding objects. Indeed, our scheme is not helpful for pruning out configurations that are infeasible owing to collisions, hence no gain would be observed on this type of scenarios.

We present the results for cases where the problem is feasible (experiments 1,2,3), and when the problem has no solution (experiments 4,5,6) in order to evaluate the capacity for SAF to *reject* a given symbolic sequence. For all experiments, we have measured the number of geometric configurations explored (#config), and the search time (time). The horizontal axis represents the runs, sorted by increasing number of configurations explored by the DFS algorithm. Hence, the horizontal axis represents the complexity of the problem measured ex post facto by DFS. Around 100 runs were conducted, and for each run, the initial positions and orientations of objects were randomized. The results for SAF were sorted accordingly to DFS, so that two points with the same abscissa correspond to the same initial conditions. Motion paths were computed using a standard rapidly exploring random tree (RRT) algorithm. In order to save time during search, raw trajectories are computed to assess the feasibility of the motions. The paths are smoothed afterward, when a solution has been found for the whole sequence. The time given in the results does not take path smoothing into account.

We also define two concepts which are useful to interpret the results:

- *the overhead* is a constant pre-computing time, during which the symbolic action sequence is converted into a linear program, and Algorithm 3 is called once in order to eliminate inconsistent domains and compute the initial bounds of all variables;
- *the filtering effort* is due to the fact that Algorithm 3 is called at each step of Algorithm 2. Hence, it depends on the number of variables, the number of domains, and the number of configurations explored.

- Experiment 1 : pick-and-place

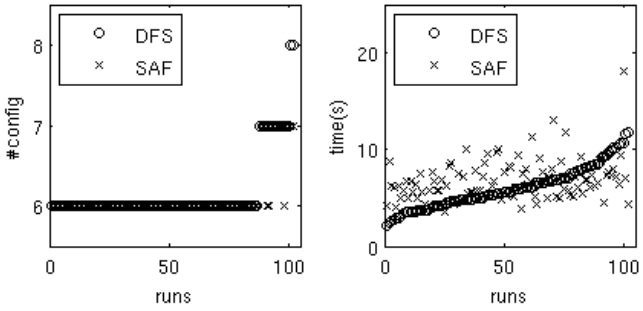


Fig. 16. Results for the "Pick-and-place" task

The results for this task are shown on Fig 16. The graph on the left shows that in 90% of the cases, both algorithms solved the task by exploring 6 configurations, which means that no backtracking was required (since the task contains 6 actions). In the remaining cases, 1 or 2 backtracks occurred, and 7 or 8 configurations were explored. The reason is that if the first cups are placed too much "on the right" of the rack, then the remaining space for placing the third cup is hard to reach (see Fig. 12), which causes RRT to fail. But overall, this task is simple because actions are kinematically independent (see the dual-constraint graph, Fig. 12), and the space on the rack is sufficient to place the three cups. A solution is found by picking each cup and sampling the space on the rack until a free position is found to place them.

The average time to solve the task is 6 s for DFS, and 6.2 s for SAF. This difference is explained by the *overhead* which is 0.2 s. The *overhead* is low because the sequence of actions contains only `place` actions, which does not imply the creation of new domains. The filtering effort is not significant here because only few configurations are explored. It is fair to say that our approach has no benefit for this type of problem, because it introduces a overhead without reducing the search space.

- Experiment 2 : Stacking cups

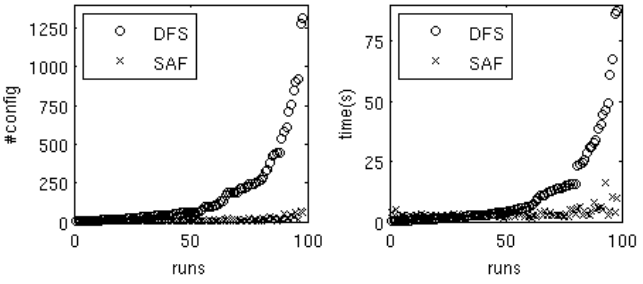


Fig. 17. Results for the "Stacking cups" task

The results for this task are shown on Fig 17. The contrast between DFS and SAF is striking on this task. DFS needs to explore on average 500 configurations to find a solution, whereas SAF finds a solution by exploring fewer than 10 configurations in 80% of the cases. The dual-constraint graph for this task (Fig. 14) shows strong dependencies

between all variables. Once the position of the first cup is instantiated, the stack constraints (which are equality constraints) determine *the exact point* where the two other cups should be. The grasp constraints reduce the domains of the variables representing the TCPs to small regions above these points (see Fig. 6). The kinematic constraints can be accurately determined on these small regions. Finally, the transfer constraints on orientations (2) determine *exactly* what should be the orientation of the TCP during the `pick` actions in order to be able rotate the cups by the required amount. The constraints are so tight that once the position of the first cup is instantiated, the instantiation of the remaining variables is almost completely determined by the constraints.

Since the task contains two `stack` actions, 9 domains are created, but after elimination of the inconsistent ones, this number drops to 5 on average. This results in an *overhead* of 0.8 s on average. The filtering effort is not much affected though, because the amount of configurations explored is drastically reduced. DFS is faster than SAF on easy problem instances (left part of the chart), because it has no *overhead*, even though more configurations are explored. However, when more search is required (right part of the chart), spending time on computing constraints pays off.

- Experiment 3 : Filling a glass

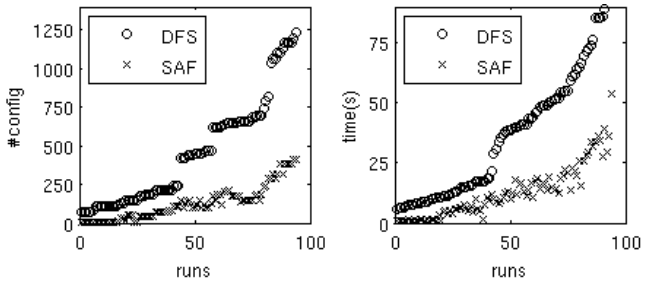


Fig. 18. Results for the "Filling a glass" task

The results for this task are shown on Fig 18. The analysis of the execution shows that the filtering prunes many instances of action 2 (placing the glass on the table). It makes sense since the variables of action 2 are subject to the constraint $\mathcal{C}^{(6)}_{Pour1}$ (see Fig. 13), but it is not obvious why this constraint can allow pruning so early, since the position of the bottle is not instantiated yet. The reason is that the position of the bottle is already very constrained by the interaction of the kinematic constraints and the constraints of the `pour` action, i.e., the region of space where the right TCP can rotate by 90 degrees (action 6) around the x-axis is limited (the algorithm "knows" this region through the kinematic constraints, which express a linear relationship between position and orientation bounds for the TCP). Hence, it is possible to prune out inconsistent positions for the glass even if the position of the bottle is not instantiated yet.

Thanks to pruning, SAF explores 3.6 times fewer configurations than DFS on average, but spends 2.9 less time only, because of the filtering effort. The simplest problem instance requires DFS to explore 74 configurations, vs. 5 configurations for SAF. Hence, even with the *overhead* ($\sim$ 0.25 s), SAF performs better on simple problem instances.

- Experiment 4 : rejecting pick-and-place

For this experiment, the randomization of the initial positions of the cups was biased, so that some of the cups are not reachable by the left arm. Hence, Fig. 19 shows results for problem instances which *cannot* be instantiated. Proving that a sequence of actions cannot be instantiated is often harder than finding a solution, because the search space has to be completely explored. For this reason, difficult problem instances take a long time to be rejected. Hence, we set a cutoff value on the number of configurations explored, which explains the plateau (3) in the first graph.

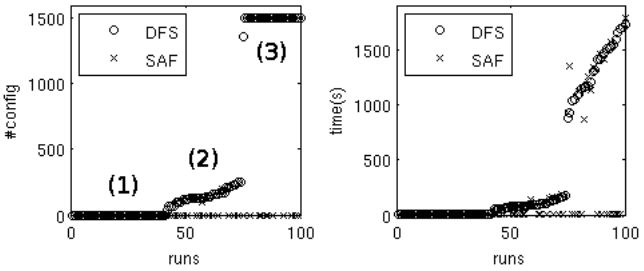


Fig. 19. Results for rejecting the "pick-and-place" task

Three types of problem instances emerge from the results Fig. 19. In type (1) no search is required because the first cup is not reachable. DFS tries all possible instances for the first `pick` action, checks if an IK solution exists, and all of them fail. This takes 20 ms on average. SAF reaches to the same result by detecting that the first grasp is not feasible thanks to the violation of the kinematic constraints generated by the first `pick` action. It is not visible on the figure, but it is faster (4 ms on average) than DFS.

In type (2), the second cup cannot be reached. In this case, the `pick` action on the second cup fails, which causes DFS to backtrack on 2 levels, i.e., it tries to pick and place the first cup in all possible ways, and each time re-attempts to pick the second cup. The same happens for type (3), but with the third cup. This causes DFS to backtrack on 4 levels, which is computationally expensive and explains why the cutoff value is always reached.

We also observe on types (2) and (3) that in 50% of the cases, SAF does not explore any configuration and rejects the sequence in 200 ms on average. Again, this is because when a cup is not reachable (for kinematic reasons or because it is too far), SAF detects it during the generation of the linear program, which is inconsistent with respect to some kinematic constraint. In this case, the sequence is rejected without the need for search. However, since the kinematic constraints are an *approximation* of the reality (they overestimate the actual capabilities of the manipulator), this does not work in the 50% remaining cases.

Hence, it happens that the linear program is consistent even though the cup is not graspable. This typically occurs when a cup is located close to the border of the kinematic map (overestimation of the reachable distance). It also occurs if the position of the cup, combined with the presence of obstacles, impose to grasp the cup with an angle which is close to one of the bounds computed in the kinematic maps (overestimation of the reachable orientation). Then SAF performs like DFS, because the task is not constrained (see Experiment 1).

- Experiment 5 : rejecting stacking cups

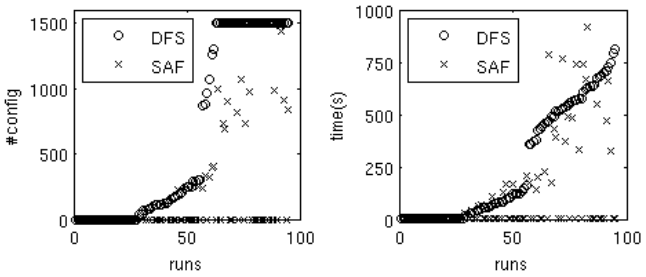


Fig. 20. Results for rejecting the "stacking cups" task

The results in Experiment 5 can be interpreted in the same way as Experiment 4, except that in the third type of problem instances, SAF manages to complete the search before reaching the cutoff value. This confirms that filtering is more efficient on constrained tasks, for instantiating as well as for rejecting an action sequence.

It is interesting to report that experiment 5 sporadically produces some problem instances which are the worst possible case for DFS: all the cups are easily graspable, but they cannot be stacked together, wherever the pile is started, whatever the orientation of the first cup is. This arises from the kinematic constraints, together with a peculiar initial configuration. In this case, DFS needs to perform an exhaustive search, whereas SAF detects the inconsistency during the generation of the linear program.

- Experiment 6 : rejecting filling a glass

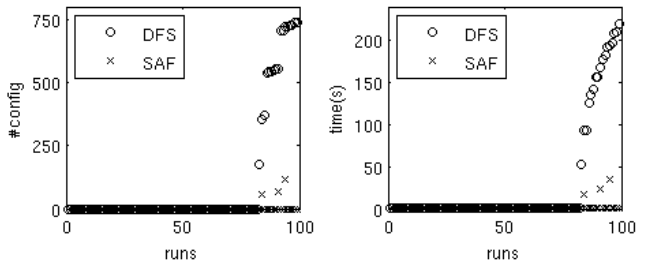


Fig. 21. Results for rejecting the "filling glass" task

In this experiment, two behaviors are typically observed. In the first one (the plateau), the sequence is rejected because the cup cannot be grasped, and in the second one because the bottle cannot be grasped. The first one occurs more often because the cup is grasped using a top-grasp, whereas the bottle is grasped by the side, which offers less possibilities for grasping when the bottle is ill-placed. Like in experiments 4 and 5, SAF detects that the first grasp is not feasible thanks to the violation of kinematic constraints. It is not visible on the figure, but it is faster (4 ms on average) than DFS (20 ms on average), as explained before.

With the second behavior, if the bottle cannot be grasped, DFS needs to backtrack on all previous actions which takes a long time. DFS always performs better: by early detecting inconsistency (30 ms on average) in 65% of these cases, or by doing a smaller amount of search thanks to filtering in the remaining cases.

### C. Summary

The experiments led to several interesting results. For instantiating a sequence of actions, our approach always performs better on average compared to a depth-first-search algorithm, if the task is sufficiently constrained (experiments 2 and 3). The more the task is constrained, the more efficient the filtering is. However, Experiment 2 suggests that when the number of domains increases, the *overhead* causes a lower performance for simple problem instances. Experiment 1 shows that problems with a loose constraint network do not benefit from our scheme.

Another interesting result concerns the rejection of action sequences. On these problems, our algorithm consistently performs better on all types of tasks by orders of magnitude. Obviously, by checking the consistency of the problem before starting an exhaustive exploration of the search space, a lot of time is saved. But experiments 4 and 5 show that even when no inconsistency is detected, our approach significantly reduces the search space. This can allow us to use lower cutoff values, hence loosing fewer potential solutions.

## IX. Conclusion

We have pointed out an inherent difficulty of CTAMP problems, which is the necessity to consider multiple ways of instantiating symbolic action sequences at the geometric level. *Geometric backtrack search* is one way of doing that. In kinematically constrained problems, *geometric backtrack search* becomes the bottleneck of the planning process. We have proposed a constraint-based approach in order to address this issue. A set of linear constraints is automatically generated from the symbolic action sequence, and from the geometric characteristics of the robot. We proposed an algorithm which uses these constraints to shrink a set of intervals bounding the variables of the problem, hence reducing the space of geometric configurations. Our experiments show that this scheme allows us to reach a solution faster when there is one, and sometimes to avoid exhaustive search when the problem is infeasible.

One limitation of the approach is to work with predefined grasps, because it prevents us from manipulating unknown objects. A possible way to deal with this would be to

"generalize" the grasp constraints from known objects to similar unknown objects. The same limitation exists with axes. Working with a pre-defined set of axes is necessary in order to generate the kinematic maps off-line. In principle, increasing the number of axes does not affect the performance of the algorithm (it simply increases the number of kinematic maps). However, it increases the complexity at the task planning level by increasing the number of possible actions. Hence, a trade-off has to be found. But the problem remains that with a finite set of axes, some actions are geometrically out of reach. A simple way to address both limitations together would be to make a hybrid system, in which actions are instantiated using the existing set of axes when it is possible, while some actions are instantiated using arbitrary grasps and arbitrary axes when it is not. Kinematic constraints could be formulated in the former case, but not in the latter.

An advantage of the proposed approach is that it is conservative, and always performs at least as well as the reference algorithm (in the worst case), with a negligible overhead. This is an advantage compared to heuristic-based approaches, which often perform the best on some problem instances, but the worst on other problem instances. Another advantage of this approach is that given a light modification of the symbolic domain (introduction of discrete axes), it can easily be plugged-in to many approaches to CTAMP.

## ACKNOWLEDGMENTS

We would like to thank in particular Florian Schmidt and Daniel Leidner from the Robotics and Mechatronics Center of DLR, for their advice, and for the functionalities they developed in Justin's simulator, which made this work possible.

## FUNDING

This work was partially supported by EU FP7 project "Generalizing Robot Manipulation Tasks" (GeRT) [contract number 248273].

## REFERENCES

E. Aker, V. Patoglu, and E. Erdem, "Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution," *Intelligent Service Robotics*, vol. 5, no. 4, pp. 275–291, 2012.

J. Bidot, L. Karlsson, F. Lagriffoul, and A. Saffiotti, "Geometric backtracking for combined task and path planning in robotic systems," University of Orebro - Cognitive Robotic Systems Lab., Tech. Rep., december 2013, ftp://aass.oru.se/pub/asaffio/robot/tr1301.pdf.

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.

K. Cho, M. Kim, and J.-B. Song, "Complete and rapid regrasp planning with look-up table," *Journal of Intelligent Robotics and Systems*, vol. 36, no. 4, pp. 371–387, Apr. 2003.

J. Choi and E. Amir, "Combining planning and motion planning," in *Proceedings of Int. Conf. on Robotics and Automation (ICRA)*, 2009, pp. 238–244.

E. Davis, "Constraint propagation with interval labels," *Artificial Intelligence*, vol. 32, no. 3, pp. 281–331, 1987.

L. de Silva, A. K. Pandey, M. Gharbi, and R. Alami, "Towards combining HTN planning and geometric task planning," in *Proc of the RSS workshop on Combined Robot Motion Planning and AI Planning for Practical Applications*, 2013, arXiv:1307.1482.

R. Dechter, "Tractable structures for constraint satisfaction problems," in *Handbook of Constraint Programming, part I, chapter 7*, 2006, pp. 209–244.

C. Dornhege, P. Eyerich, T. Keller, S. Trg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2009, pp. 114–121.

E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, "Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation," in *Proceedings of Int. Conf. on Robotics and Automation (ICRA)*, 2011, pp. 4575–4581.

J. Guitton and J.-L. Farges, "Taking into account geometric constraints for task-oriented motion planning," in *ICAPS Workshop on Bridging the gap Between Task And Motion Planning, BTAMP'09*, 2009.

Gurobi Optimization, Inc., "Gurobi optimizer reference manual," 2013. [Online]. Available: http://www.gurobi.com

K. Hauser, V. Ng-Thow-Hing, and H. H. González-Baños, "Multi-modal motion planning for a humanoid robot manipulation task." in *Proc of the Int Symp on Robotic Research*, 2007, pp. 307–317.

J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, no. 1, pp. 253–302, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622394.1622404

L. Jaulin, "Interval constraint propagation with application to bounded-error estimation," *Automatica*, vol. 36, no. 10, pp. 1547–1552, 2000.

L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now." in *Proceedings of Int. Conf. on Robotics and Automation (ICRA)*, 2011, pp. 1470–1477.

L. Karlsson, J. Bidot, F. Lagriffoul, A. Saffiotti, U. Hillenbrand, and F. Schmidt, "Combining task and path planning for a humanoid two-arm robotic system," in *TAMPRA: ICAPS Workshop on Combining Task and Motion Planning for Real-World Applications*, 2012.

L. Kuipers and H. Niederreiter, *Uniform distribution of sequences*. New York: Wiley, 1974.

F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson, "Constraint propagation on interval bounds for dealing with geometric backtracking," in *Proc. of the Int. Conf. on Intelligent Robots and Systems (IROS)*, 2012, pp. 957–

964.

S. LaValle, *Planning Algorithms*. Cambridge, UK: Cambridge University Press, 2006.

Y. Lebbah, M. Rueher, and C. Michel, "A global filtering algorithm for handling systems of quadratic equations and inequations," in *Proceedings of the 8th Int. Conf. on Principles and Practice of Constraint Programming*, ser. CP '02, 2002, pp. 109–123.

O. Lhomme, "Consistency techniques for numeric csps," in *Proceedings of the 13th international joint conference on Artifical intelligence*, 1993, pp. 232–238.

T. Lozano-Perez, J. Jones, E. Mazer, and P. O'Donnell, "Task-level planning of pick-and-place robot motions," *Computer*, vol. 22, no. 3, pp. 21–29, 1989.

N. Megiddo, "On the complexity of linear programming," in *Advances in economic theory: Fifth world congress*, 1987, pp. 225–268.

D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

C. Ott, O. Eiberger, W. Friedl, B. Bäuml, U. Hillenbr, C. Borst, A. Albu-schäffer, B. Brunner, H. Hirschmüller, S. Kielhöfer, R. Konietschke, T. Wimböck, F. Zacharias, and G. Hirzinger, "A humanoid two-arm system for dexterous manipulation," in *Proceedings of Int. Conf. on Humanoid Robots (Humanoids)*, 2006, pp. 276–283.

A. Pandey, J.-P. Saut, D. Sidobre, and R. Alami, "Towards planning human-robot interactive manipulation tasks: Task dependent and human oriented autonomous selection of grasp and placement," in *International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, june 2012, pp. 1371–1376.

E. Plaku and G. Hager, "Sampling-based motion planning with symbolic, geometric, and differential constraints," in *Proceedings of Int. Conf. on Robotics and Automation (ICRA)*, 2010, pp. 5002–5008.

T. Siméon, "Manipulation planning with probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 729–746, 2004.

T. Siméon, J.-P. Laumond, J. Cortes, and A. Sahbani, "Manipulation Planning with Probabilistic Roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 729–746, Aug. 2004.

B. M. Smith, "A tutorial on constraint programming," University of Leeds, Tech. Rep., 1995.

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.

M. Stilman and J. Kuffner, "Planning among movable obstacles with artificial constraints," in *Algorithmic Foundation of Robotics VII*, ser. Springer Tracts in Advanced Robotics, 2008, vol. 47, pp. 119–135.

P. Tournassoud, T. Lozano-Perez, and E. Mazer, "Regrasping," in *Proc. of Int. Conf. on Robotics and Automation*, 1987, pp. 1924–1928.

G. Wilfong, "Motion planning in the presence of movable obstacles," in *Proceedings of the fourth annual symposium on Computational geometry*, ser. SCG '88, 1988, pp. 279–288.

J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2010, pp. 254–258.

F. Zacharias, C. Borst, and G. Hirzinger, "Capturing robot workspace structure: representing robot capabilities," in *Proc. of the Int. Conf. on Intelligent Robots and Systems*, 2007, pp. 3229–3236.