Constraint-based Action Specification for bridging the gap between Symbolic and Control-theoretic Action Execution

Ingo Kresse and Michael Beetz
Intelligent Autonomous Systems Group
Department of Computer Science
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
{kresse, beetz}@cs.tum.edu

Abstract—In this paper we propose a bridge between a symbolic reasoning system and a task function based controller. We suggest to use modular position- and force constraints, which are represented as action-object-object triples on the symbolic side and as task function parameters on the controller side. This description is a considerably more fine-grained interface than what has been seen in high-level planners before. It can preserve the 'null space' of the task and make it available to the control level. We demonstrate how a symbolic description can be translated to a control-level description that is executable on the robot. We describe the relation to existing robot knowledge bases and indicate information sources for generating constraints on the symbolic side. On the control side we then show how our approach outperforms a traditional controller, by exploiting the tasks null space, leading to a significantly extended work space.

I. INTRODUCTION

Over the last decades a number of control paradigms have been developed for specifying the activities to be performed by autonomous robots. These paradigms include behavior-based control [1], emergent and developmental approaches [2], control-theoretic [3, 4] and symbolic methods [5]. Within this spectrum of control paradigms the control-theoretic as well as the symbolic approaches aim at engineering high-performance task execution of complex real-world tasks. However, both approaches differ substantially with respect to their level of abstraction and the conceptual apparatus they use for action specification.

Symbolic approaches enable the programmers to specify tasks in terms of objects that are to be manipulated and used, the desired effects and undesired side effects. They allow to specify actions such as “push the spatula under the pancake in order to flip the pancake.” As a consequence, the robot can reason about what it is doing, how, and why. They are, however, unable to specify fine-grained motions or reactions to deal with disturbances and are therefore not able to ensure stable, controllable, and optimal movement performance.

Control engineering approaches have complementary strengths and weaknesses. They specify actions as fine-grained movements by specifying coordinate systems or frames and geometric-, motion-, and force constraints on how frames should move with respect to each other. In a number of control frameworks, such as iTaSC [4], the programmer can then ensure that the frames move in a stable, controllable and optimal way. The conceptual apparatus that these frameworks provide does however not allow us to talk about the effects of movements on object states.

The need for integrating the two paradigms has been recognized for a long time and researchers have proposed a number of software architectures that featured three layers of abstraction, often called 3T architectures. The basic idea behind 3T architectures is simple. The architectures use symbolic action specification and execution at the top layer and control engineering methods at the bottom layer. As both paradigms do not fit properly together, the researchers have introduced an intermediate layer for bridging the differences. This intermediate layer is typically implemented by a reactive situated task management system, which provided the means
for triggering task execution, for making task execution sensible to the situational context, for managing concurrent task execution, and for achieving robustness in the presence of task plan failure.

3-layer architectures [6] constitute a shallow integration that enabled researchers to keep their symbolic and control engineering approaches without changing them. For obvious reasons, this shallow integration turned out to be of limited use. As the abstract symbolic layer does not have the notions of concurrency, situatedness, failure and failure recovery, there is no way how this layer can understand the behavior generated at the lowest level and thereby diagnose the weaknesses in its intended course of action. In addition, the abstract layer does not provide the control structures needed for making the behavior flexible, robust, and concurrent.

A promising alternative is to couple the symbolic and the control layer directly by means of shared data structures that specify movements as first-class objects. As constraint-based specifications are successful representations both in artificial intelligence and in control engineering, we use constraint-based representations as an interlingua.

For bridging the gap between the symbolic side and the control side, we propose a constraint based movement specification that is fine-grained, modular and transparent. It avoids the overspecification of tasks, and thus retains the null space that can be exploited by the robots control level. This leads to more dexterity and an increased effective work space.

The rest of the paper is structured as follows: In Section II, we give an overview of our system and introduce a constraint specification which is general enough to represent common household tasks and transparent enough to be reasoned about by a high level controller. Section III explains our implementation in detail. In Section IV we apply our approach on an everyday manipulation task, namely pancake flipping with a spatula and execute this task on a robot. This demonstration is evaluated in Section V before we conclude in Section VI.

II. SYSTEM OVERVIEW

In this section we introduce the three main components of our system: A constraint based movement specification for our low level controller, a high level action specification for our symbolic planner and we sketch how the high level representation can be mapped onto the controllers specification.

The symbolic planner is dealing with abstract representations like identities of objects, types, properties, qualitative spatial relations like ‘on’, or ‘towards’ and the effects that are to be achieved.

Such a description can be generated e.g. from web instructions and translated into logic-based action specifications [7]. The system can then decompose that high-level task description into fine-grained action specifications based on its knowledge about the hierarchical structure of actions [8, 9]. As a result, it has described the task as detailed symbolic movement descriptions, such as

```
(perform (an action
  (type push)
```

This fine-grained symbolic representation needs to be translated into numeric constraints that can be understood by a motion controller.

A complementary method is to generate positions for objects based on constraints like 'free' or 'stable' using a physics engine [10]. However, our focus lies on generating executable robot motions rather than only positions.

Given a description like the one above, we are interested in three symbols: An action type and two objects whose relative pose or force is described. Some instructions in the planner may not contain all of these symbols. In this case we assume that they can be completed on the symbolic level using either context information or default values. If such a completion is not possible, we simply ignore the instruction.

Since one such constraint is only a partial description of one movement, the controller must be able to execute 1) a set of such constraints simultaneously and 2) a sequence of such constraint sets.

In order to fulfill these requirements we use a controller based on the task function approach. This allows us to specify a robotic task in an arbitrary (possibly low-dimensional) coordinate system and keep a notion of the task’s null space. This nullspace can then be exploited by the robot for secondary tasks like obstacle avoidance or joint limit avoidance.

For our tasks, we span a 6-DOF virtual linkage between two objects and control the virtual joints over time. A virtual joint can also be “switched off”, to indicate that its position is not important for the current task and can be exploited by the secondary constraints.

This controller deals with coordinates and trajectories, and has little knowledge about the symbols of the high level planner. On this virtual linkage level we represent a constraint as a triple of positions, forces and weights where positions ∈ R^{2×6} are six ranges of allowed positions, forces ∈ R^{2×6} are six ranges of allowed forces and weights ∈ R^6 are six booleans values describing whether a joint is important at the moment.

In order to translate a constraint from the high level into virtual linkage terms, the following tasks must be solved:

- extract from the action type whether it is a position- or a force constraint
- associate object symbols with perceived objects in the scene.
- extract from the action symbol a) the linkage joint to use and b) the desired joint position/force or ranges of position/force.
The interpretation of the action type is solved by a simple mapping. The association of object symbols to the scene is assumed to be done by the perception system and has been studied before [11].

In order to find the linkage joint we semantically annotate the joints of the virtual linkage. The first three axes of the linkage, which form cylinder coordinates, are labeled “angle”, “distance” and “height”, while the last three axes can labeled “roll”, “pitch” and “yaw”.

We use a map associating relation symbols with these axis names and a comparison operator (smaller, bigger, etc.). After looking up the current positions of both object parts, the ranges for the desired positions and forces are generated.

III. IMPLEMENTATION

For the sake of simplicity we concentrate on tasks with one object and either one hand or tool. However, the underlying control system iTaSC has been shown to support multi-robot scenarios [4], so an extension to two-arm tasks are possible.

A. Control System

For defining the virtual linkage task we first have to define coordinate systems for the tool (or hand) and the object. It is placed in the center of the object or tool, onto symmetry axes (if present) and is aligned with important plane- or line segments.

Next, we span one of two common linkages: Either a combination of cylinder coordinates and roll-pitch-yaw (for objects that have an axis of symmetry), otherwise we use cartesian coordinates with roll-pitch-yaw. These coordinate systems uniquely cover the six degrees of freedom in the transform between the two objects. Each of these axes is labeled with a name for linking with the higher level constraints: We use ‘angle’, ‘distance’ and ‘height’ for the first three axes that form cylinder coordinates and ‘roll’, ‘pitch’ and ‘yaw’ for the last three axes which define the orientation.

Several of these chains can be combined and executed simultaneously. If some constraints are conflicting, then a hierarchy can be built in order to execute one constraint in the null space of the other.

While being useful for our tasks, these chains are not the only possibilities to specify constraints. It is e.g. possible to only specify the alignment of two line segments as a constraint instead of using a 6-dof chain. For our validation task, however, one 6-dof chain was versatile enough.

We then control this virtual linkage by specifying desired ranges for its axes. This approach is motivated by the fact that many symbolic constraints map to inequality expressions, e.g. “keep the spatula above the oven” implies that the height of the spatula should be greater than the height of the oven. Such inequalities can be solved elegantly with more specialized methods as in [12]. However, a simpler approach was sufficient for our scenario.

On this low (virtual linkage) level we represent a constraint as follows:

\[
\text{constraint}_i = (\text{positions}, \text{forces}, \text{weights})
\]

where \(\text{positions} \in \mathbb{R}^{2 \times 6}\) are six ranges of allowed positions, \(\text{forces} \in \mathbb{R}^{2 \times 6}\) are six ranges of allowed forces and \(\text{weights} \in \mathbb{R}^6\) are six booleans values describing whether a joint is important at the moment.

For every task axis with the allowed range \([y_{lo}, y_{hi}]\) we implement the position constraints as a P-controller with a ‘dead zone’ inside the allowed range. Additionally, we lower the weight \(w\) of the constraint so that lower level tasks can exploit that axis:

\[
\dot{y}_p = \begin{cases} 
K_p(y_{hi} - m - y) : & y > y_{hi} - m \\
K_p(y_{lo} - m - y) : & y < y_{lo} + m \\
0 & \text{else}
\end{cases}
\]

\[
w = \begin{cases} 
\frac{1}{m}(y - y_{hi}) + 1 & : y_{lo} < y < y_{lo} + m \\
\frac{1}{m}(y_{lo} - y) + 1 & : y_{hi} - m < y < y_{hi} \\
1 & : \text{otherwise}
\end{cases}
\]

Here, \(y\) is the current task angle, \(\dot{y}_p\) is the desired speed of the task angle, \(K_p\) is a plant-dependent control coefficient and \(m\) is a margin by which the controller shall push inside the allowed range. This is visualized in Figure 3.

Using the task’s Jacobian matrix (i.e. the derivative of every task angle w.r.t. movements around every joint axis of the robot) the controller can transform the desired task velocities into robot joint velocities.

Since our robot has impedance control, we can apply forces by offsetting the position. The required offset is computed using the controllers spring constant \(K_F\).

\[
\Delta y_f = K_F(F_{\text{desired}} - F_{\text{measured}})
\]

For applying a force on a given object, we propose the following approach: The position range has priority over force and limits the arm movements. When the actual task angle
is inside the allowed range, then an offset is added to the desired task angle until either the desired force is reached or the allowed position range is left.

In order to exploit the null space of the task we use a simple joint limit avoidance task. The joint limits are only avoided when the joint angle is within a distance of $d$ from the limit, the maximum repelling velocity is $h$ and the function is continuous and differentiable over the whole joint range.

A piecewise quadratic function which is depicted in Figure 4 is chosen to fulfill these requirements:

$$
\dot{q}_a = \begin{cases} 
\frac{h}{d^2}(q_{lo} - q)^2 & : q < q_{lo} + d \\
-\frac{h}{d^2}(q_{hi} - q)^2 & : q > q_{hi} - d \\
0 & : \text{otherwise}
\end{cases}
$$

On the high level side we obtain a triple $\langle\text{action}, \text{object1}, \text{object2}\rangle$, where object2 can be empty and is completed with the object 'world', representing merely the vector of gravity.

We then perform a synonym lookup on the action words in order to simply the next step.

Then we map the action to a set of $\langle\text{axis}, \text{comparison}, \text{margin}\rangle$ triples, where $\text{axis}$ is a name, $\text{comparison}$ is one of $\langle >, <, = \rangle$ and $\text{margin}$ is a real number. This expresses that the first object’s value of $\text{axis}$ should be higher/lower/equal to the second object’s value by a certain margin.

In order to compute the desired range we take advantage of one more piece of knowledge that is commonly available to a robot: The sizes of the objects involved. For instance, if two objects have an extension in the vertical direction and and one object should be above the other, then the sizes need to be examined in order to compute the correct distance. This can be computed by simple interval arithmetics.

The resulting intervals from these primitive constraint triples are then stored in the six ranges and weights so they can be executed by the controller.

This intermediate layer – still mostly symbolic – expresses knowledge that is hard to find in natural language resources. It is thus specified by hand, but it could be an interesting learning task using annotated tracking data from human demonstration.

The low level specification allows to perform a simple check whether two constraints $c_1$ and $c_2$ are compatible:

$$
\text{compatible}_i(c_1, c_2) = \text{pos}_i(c_1) \cap \text{pos}_i(c_2) \neq \emptyset \\
\forall \text{weight}_i(c_1) = 0 \\
\forall \text{weight}_i(c_2) = 0 \forall i \in 1..6
$$

Similarly, two constraints can be merged by intersecting its position and force ranges and and-ing their weights.

When the reference frames of the two constraints differ, then this test can be computed probabilistically: For random poses, the constraint Jacobians $J_1$ and $J_2$ are computed and its rank is computed. When the condition

$$
\text{rank}(J_1) + \text{rank}(J_1) = \text{rank}([J_1, J_2])
$$

holds then the constraints $c_1$ and $c_2$ are not conflicting. They may still interfere with each other (then the columns of $J_1$ need to be orthogonal with all columns of $J_2$). However, the rank condition implies that all the constraints are instantaneously solvable.

### IV. Example Task

We consider the example task of flipping a pancake with a spatula. Assuming that the robot has its tool already in the hand and is standing in front of an oven with the pancake on it. We specify the following symbolic constraints:
$c_p = ('point towards', 'spatula', 'oven')$
$c_h = ('keep horizontal', 'spatula')$
$c_n = ('move next to', 'spatula', 'pancake')$
$c_u = ('move under', 'spatula', 'pancake')$
$c_l = ('lift', 'spatula')$
$c_o = ('keep over', 'spatula', 'oven')$
$c_f = ('flip', 'spatula')$

These constraints are combined into four steps which need to be executed:

$s_1 = [c_p, c_h, c_n]$
$s_2 = [c_p, c_h, c_u]$
$s_3 = [c_h, c_o, c_l]$
$s_4 = [c_o, c_f]$

Each step is completed when its constraints are satisfied. We use the following relations to translate these constraints into virtual linkage coordinates:

'point towards' $\rightarrow$ \{(pitch', $\approx$, 0, 0.2)\}
'from left' $\rightarrow$ \{(angle', $\approx$, 0.5$\pi$, 1.0)\}
'move under' $\rightarrow$ \{(height', $\ll$, 0, 0),
(distance', $\approx$, 0, 0)\}
'move over' $\rightarrow$ \{(height', $\gg$, 0.15, 0),
(distance', $\ll$, radius(obj2), 0)\}
'move next to' $\rightarrow$ \{(height', $\approx$, 0, 0),
(distance', $\approx$, 0.03, 0)\}
'keep horizontal' $\rightarrow$ \{(roll', $\approx$, 0, 0.05),
(yaw', $\approx$, 0, 0.05)\}
'lift' $\rightarrow$ \{(height', $\gg$, 0.2, 0)\}
'flip' $\rightarrow$ \{(roll', $\approx$, 0, 0.15),
(yaw', $\approx$, 0.6$\pi$, 0.1)\}

These mappings specify a set of 4-tuples: A task axis name, a relation, a fixed offset, which is necessary for constraints like 'flip' and finally a tolerance. The numbers had to be completed manually. However, given the rest of this description, it may be possible to extract them from human demonstrations.

In this list of actions there are two classes: Actions that relate two objects with respect to each other and actions on one object. For the second class, the "world" could be filled in.

For some actions, the verb move could be replaced by keep. In our system this leads to the same constraint, but this distinction can be a hint to monitoring: Move expresses, that the constraint is violated at the beginning and holds at the end of the movement. Keep, on the other hand, expresses that the constraint should always hold. If the actual behaviour is different, then this is a hint that the execution is not running smooth.

Some actions are underspecified in this verbal form. The action turn, for instance, just specifies some rotation around the object center. With the context knowledge, that gravity is playing a role in this step, it can be inferred that a rotation around the direction of gravity has no effect but a rotation perpendicular to gravity has the maximum effect.

In Figure 5 the advantages of the constraint-based specification becomes apparent: It depicts several possible positions that are allowed by the approach constraint $s_2$. It is left up to the robot to instantaneously choose one that is currently reachable. This separates the task specification from the robots embodiment. Figure 6 shows the effect of combining constraints: The grey spatulas are instances of lift-poses, while the red spatulas show poses that are allowed when the 'above oven' constraint is taken away.

This specification was run on the robot TUM-ROSIE, which is equipped with two impedance-controlled lightweight arms from KUKA, two DLR-HIT hands from Schunk. These arms are mounted on a Mecanum-wheel omnidirectional base, together with a sensor head containing (amongst others) a high-resolution stereo camera pair and a kinect.

In order to successfully execute this task, an extra constraint had to be introduced, which assures that the spatula remains on one side of the oven. This was required to avoid a singularity of the RPY-angles which surfaced during flipping. We are currently considering more adequate angle representations.

V. EVALUATION

Allowing a robot to exploit a task’s null space can significantly increase its work space. In order to evaluate this advantage we tested the pancake-baking task in simulation.
We generated 30 random poses for the oven in a 0.3 x 0.3 x 0.4 meter space and execute the pancake flipping once with our constraint based controller and once using fixed relative poses between the oven and the spatula which were taken from one trial of the constraint based controller. The results are shown in Table I. The constraint based controller can reach more than twice as many poses than the pose based control. Only the flipping pose was posing problems for our approach: Being already a difficult pose to reach, we had to constrain it unnecessarily in order to avoid RPY singularities. A more specialized angle representation may lead to even more successes.

Figure 8 illustrates why the constraint based approach was more successful: When the pose-based controller has to stretch the robot arm in order to reach its sideways approach pose, the constraint based controller exploits the symmetry of the oven and chooses a more dextrous and convenient pose.

VI. CONCLUSION

We presented a modular way of specifying motions using constraints, which tightly links symbolic reasoning to control theoretic execution. We showed how these constraints can be combined, reasoned about, and monitored during execution. We demonstrated the execution of such constraints and show that they can significantly increase the robots dexterity.

REFERENCES


