MoveIt Task Constructor

Introspectable Task Specification and Planning*

Michael Görner, Robert Haschke

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Technical Aspects of Multimodal Systems

May 26, 2020

*published as Görner et al., 2019, ICRA.

Outline

Outline

Manipulation Actions

It is almost trivial to execute robot arm motions from A to B.

But "straight-forward" manipulation actions remain difficult to implement, even with a simple end-effector:

- Grasp known objects
- Pour from bottle
- Place object
- Press buttons
- ...

If scene geometry is quasi-fixed, implementations are easy but do not generalize well.

Context

Task: Perform "standard" manipulation in autonomous robotics.

- *Action sequence* is known, e.g.,
  - Collision-free transit motion near expected manipulation
  - Approach
  - Grasp control
  - Lift-off
- *Action parameters* are unknown
  - How "near"?
  - What approach vector?
  - What grasp?

If multiple actions are chained, the parameters are interdependent.
E.g., how a container is grasped restricts how to pour from it.

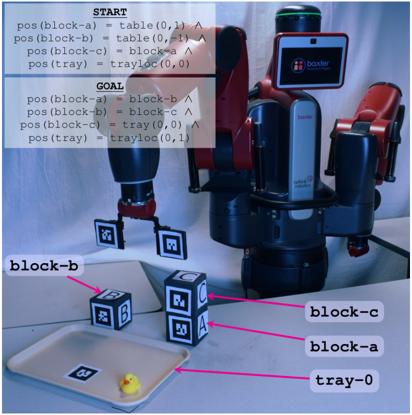
Outline

Related Work

Addresses part of the domain of **T**ask and **M**otion **P**lanning.

Whereas *Task Planning* infers whole action sequences, we usually already know what type of motion to perform.

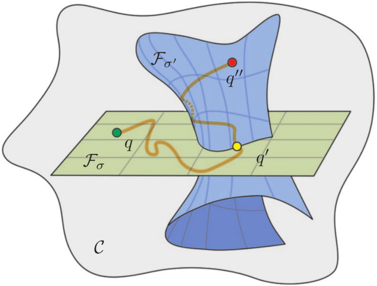
Related Work - TMP



Dantam et al., 2016

- Interface are symbolic goals
- Action parameterization part of the implementation
- Hard to predict the concrete action the robot will take
- Results are restricted to simulation or simple repetitive motions

Related Work - Manifold Motion Planning



Hauser and Latombe, 2010

- PRM planning in intersecting manifolds
- Idea: Sample from each manifold *and each intersection*
- "Multi-Modal" Planning
- Highly abstract formalization
- Ignores almost the whole problem

Outline

Design Goals

Define a framework to specify manipulation actions

- Robot-agnostic
  - Not restricted to specific kinematics/end-effector
- Introspectable
  - "IK not found" is no useful feedback
  - Nor is "optimization result has cost X"
- Meticulous control over trajectory processing
  - Do not hide anything
  - The engineer knows best what the behavior has to look like
- Full control over execution
  - Manipulation is more than sending a trajectory
  - Account for world changes, use dedicated controllers

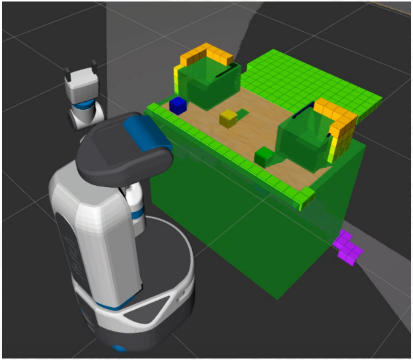
Outline

World Representation

- Motion generation needs a model of *the robot*.
- Manipulation planning needs a model of *the environment*

MoveIt's `PlanningScene`

- The robot state
  - Positions
  - Dynamics information
  - Collision Geometry
- Objects with shapes & types
- Attachment information
- Octomaps for sensor-based collision checking
- Allowed collisions



Motion Planning

Figure: Abstract sketch of motion planning problem

Motion Planning

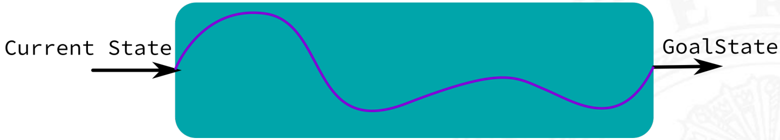


Figure: Abstract sketch of motion planning problem

Motion Planning

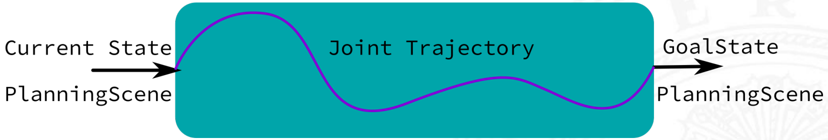
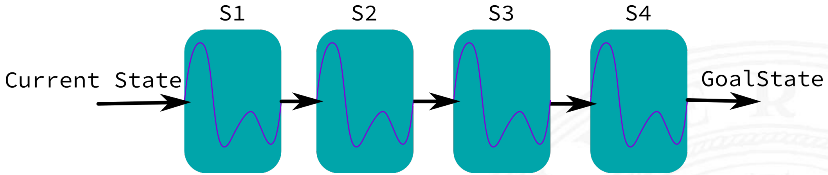


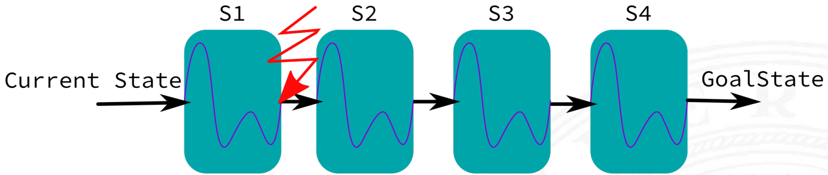
Figure: Abstract sketch of motion planning problem

Manipulation Planning



- Of course plans can be concatenated
- Greedy form of sequence planning

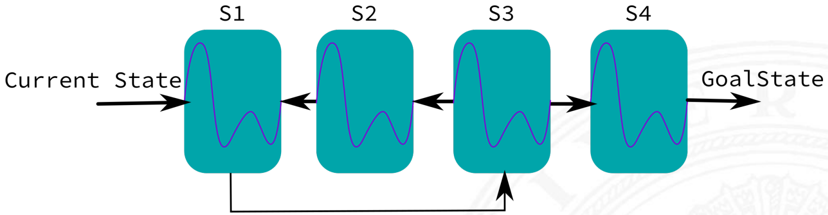
Manipulation Planning



- Of course plans can be concatenated
- Greedy form of sequence planning

- What if S2 depends on S3?
- E.g., a pre-approach position depends on the approach
- Early commitment will usually fail

Manipulation Planning

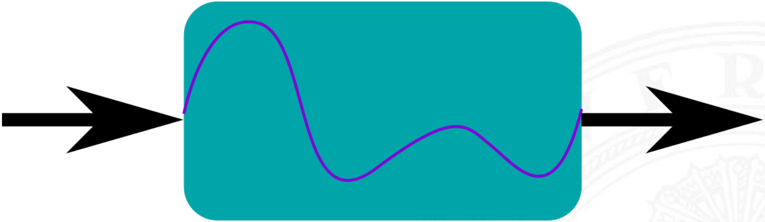


- Inverted inference
- Decouples planning order and execution order
- Requires more book keeping
- Retains isolated planning stages

This is the main idea of the Task Constructor system.

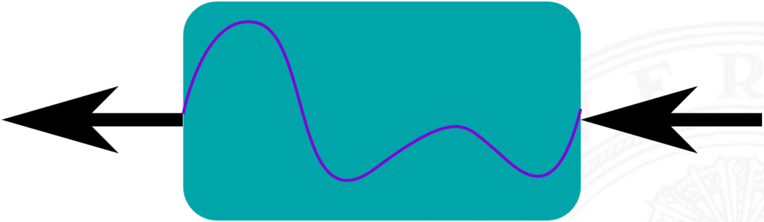
Outline

Stage Types - Forward Propagator



- **Assumes** a start scene
- **Yields** solution trajectories and end scenes

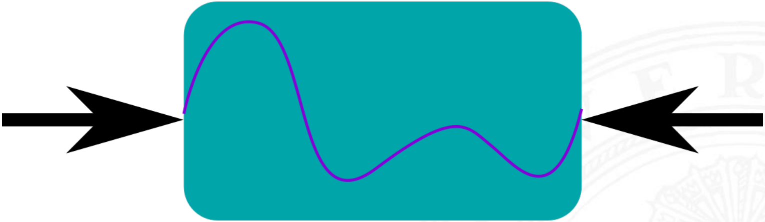
Stage Types - Backward Propagator



- **Assumes** an end scene
- **Yields** solution trajectories and start scenes

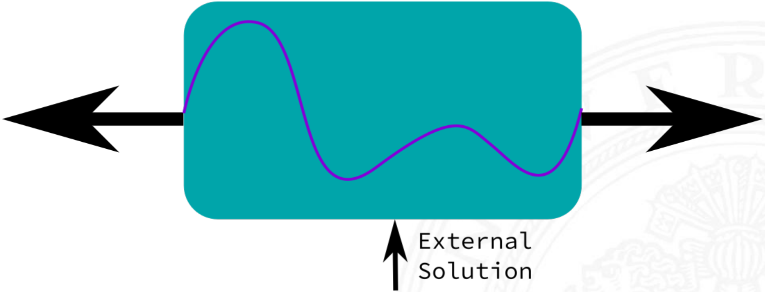
- `PropagatingEitherWay` subsumes both Forward and Backward

Stage Types - Connector



- **Assumes** a start and an end scene
- **Yields** solution trajectories

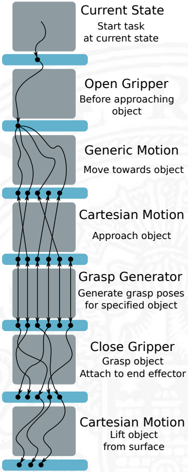
Stage Types - (Monitoring) Generator



- **Assumes** External Solution (optional)
- **Yields** start states, end states and trajectories

Cover a solution space

- Stages generate arbitrarily many local solutions
- Many solutions turn out infeasible in other stages
- Allows branching and ranking of alternative action parameters
- Sucessful solutions connect through the whole task



Standard Stages

Many planning stages can be reused

- Fetch the current scene from the system
  `CurrentState` - Generator
- Generic motion plan (possibly constraint)
  `MoveTo` - PropagatingEitherWay
- Relative motion
  `MoveRelative` - PropagatingEitherWay
- Changes in the Planning Scene
  `ModifyPlanningScene` - PropagatingEitherWay
- ...

Properties

Stages can be configured via declared *Properties*

For example, `MoveTo` defines these properties:

- Group
  - Which joints to move
- Goal
  - Cartesian goal specification
  - Or joint space goal
- IK frame
  - Frame to move to the goal
- Path constraints
  - To respect during motion

Specification vs. Implementation

MoveTo and MoveRelative specify a motion.
They do not define how to generate such a motion.

The request can be solved by

- Joint space interpolation
- Cartesian trajectory generation
- Any solver supported through MoveIt (e.g., OMPL)
- Your own trajectory generator

Outline

Subsolutions

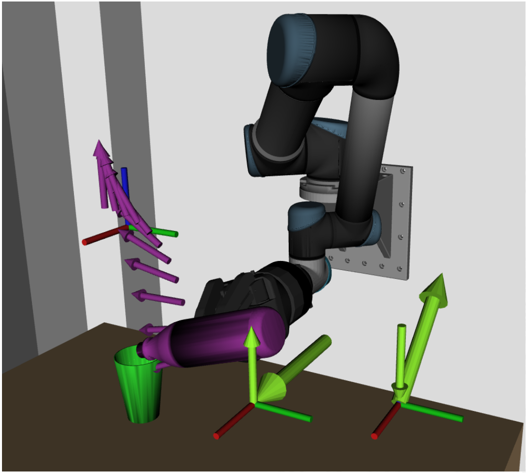
Local solutions comprise

- A *start* and *end* state
- A trajectory connecting them (might be empty)

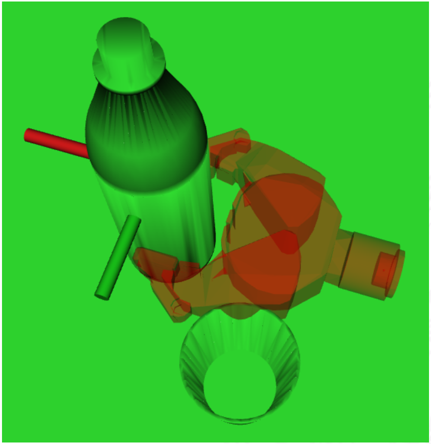
Locally forwarded states comprise:

- A `PlanningScene` world representation
- Task-characteristic visual markers
- Comments (optional)
  - Facilitate visual introspection
- Properties
  - Configure stages from partial solutions, e.g., which hand to use
- A cost
  - Rank solution among other candidates

Marker Introspection

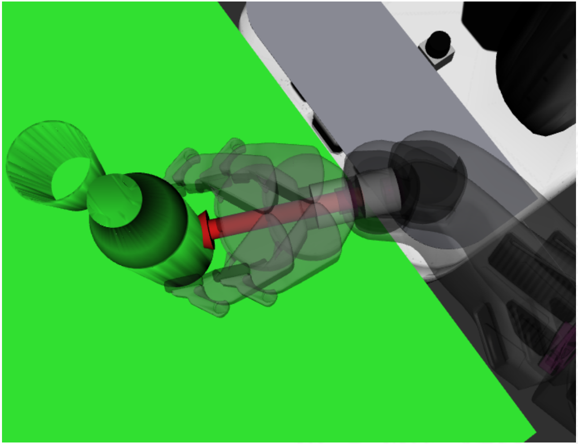


Example Issues - End-Effector in Collision



Comment: eef in collision: glass - l_gripper_l_finger_link

Example Issues - Partially Infeasible Cartesian motion



Comment: min_distance not reached (0.0471 < 0.07)

Custom Stages

```
class MyStage : public PropagatingForward {
public:
   MyStage(string name);

   void computeForward(const InterfaceState& from) override
   {
      ...
      SubTrajectory solution(trajectory, cost, comment);
      solution.markers().push_back(marker);
      sendForward(from, move(end_scene), move(solution));
   };
};
```

- Example stage: `PourInto`

Outline

Containers

To reuse stage sequences and encapsulate them, they can be aggregated in containers.

The simplest way to do so was already introduced:

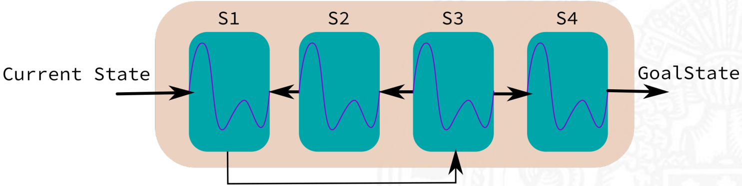


Figure: Serial Container

Wrapper

Modify a generated solution and pass it on.

- Apply post-processing
  - Smooth result
- Enforce additional constraints
  - Reject trajectories which spill liquid from container
- Compute solutions based on child input
  - Compute inverse kinematics for a target (property)
  - Duplicate solution with minor modifications

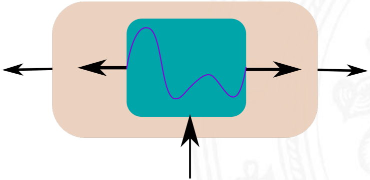
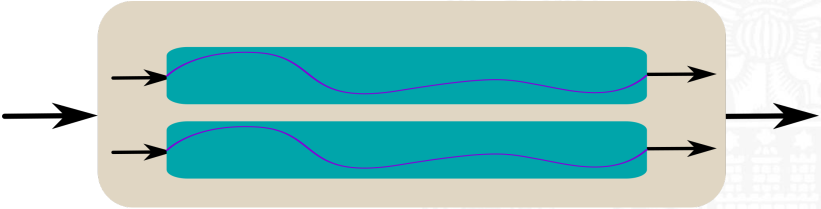


Figure: Generator Wrapper

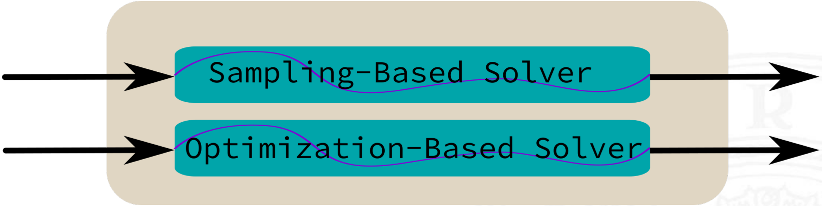
Parallel Containers

Parallel ordering is allowed as well.
Multiple interpretations are useful:

- Equally-ranked alternative solutions
- Fallback ordering
- Separated aspects of the same solution

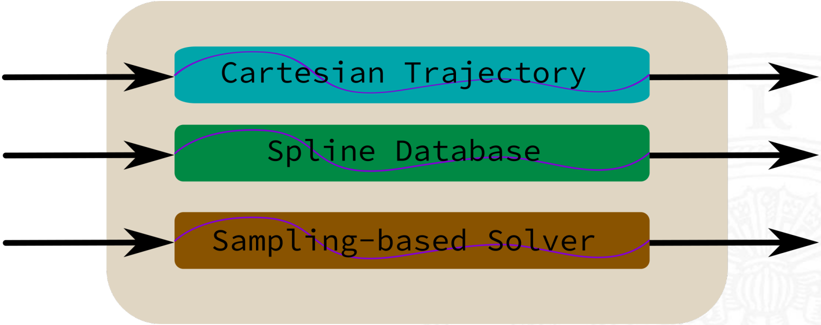


Parallel Containers - Alternatives



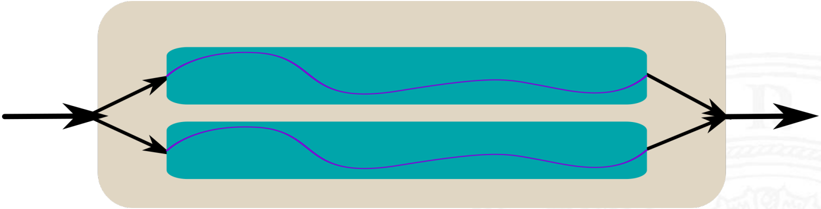
- Treat solutions as equal alternatives
- Specify orthogonal planners
- Specify different action modes, e.g., tripod or power grasp

Parallel Containers - Fallback



- Only attempt alternatives if better ones fail
- Discrete preferences
- Last-resort alternatives

Parallel Containers - Merger



- Plan trajectories for multiple joint groups independently
- Task setup must ensure disjunct solution spaces
- Otherwise trajectories will fail during merge

Cost Terms

- Support pluggable cost functions to rate local solutions
  - ConstantCost
  - PathLengthCost
  - LinkMotionCost
  - ClearanceCost
  - ...
- Additionally, containers can *transform* aggregated costs
  - Weigh alternatives against each other
  - Cap costs of successful solutions
  - ...

Downside:

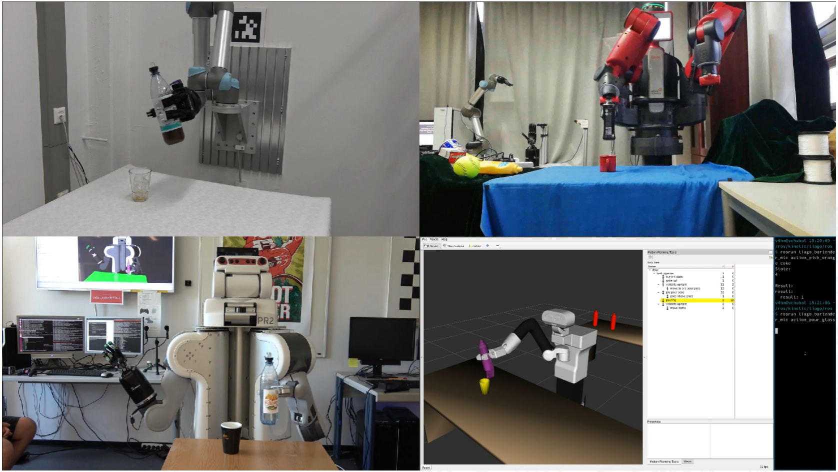
- As stages do not know the cost terms for solutions they generate, they cannot improve on them.

Outline

Pouring

- pouring task
  - ↕ current state
  - ↓ pick bottle
    - ↑ . . .
    - ↕ grasp
    - ↓ . . .
  - ⛓ move to pouring start
  - ↕ compute ik
    - ↕ bottle above glass
  - ↓ pouring
  - ↓ place bottle
    - ⛓ move to place
    - ↑ set down bottle
    - ↕ compute ik
      - ↕ bottle place location
    - ↓ release bottle
      - ↓ open gripper
      - ↓ detach object
      - ↓ forbid object collision
    - ↓ retreat gripper

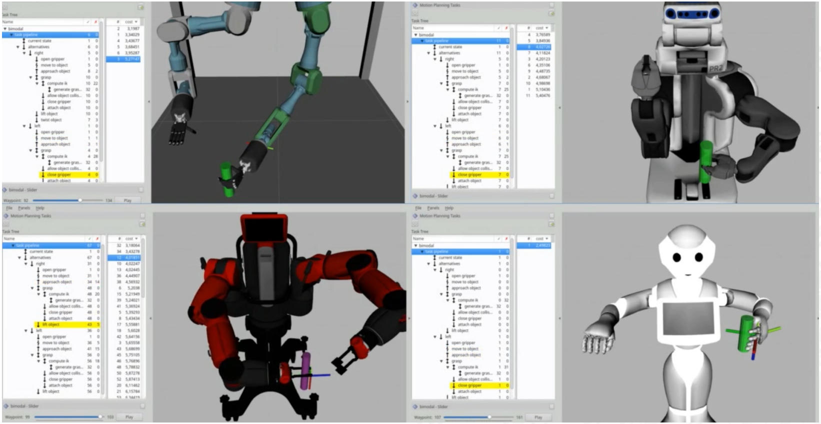
Pouring



Bimodal Pick

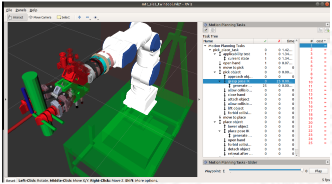
- bimodal pick task
  - ↕ current state
  - ↓ alternatives
    - ↓ right pick
      - ↓ open gripper
      - move to object
      - ↑ approach object
      - ↕ grasp
        - ↕ compute ik
          - ↕ generate grasp pose
        - ↓ permit object collision
        - ↓ close gripper
        - ↓ attach object
      - ↓ lift object
    - ↓ left pick
      - ↓ . . .

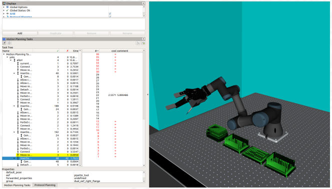
Bimodal Pick



Other Projects

Wang et al., 2020





Outline

Trajectory Blending

An annoying side-effect of kinematic path planning in stages is that motions stop between segments.

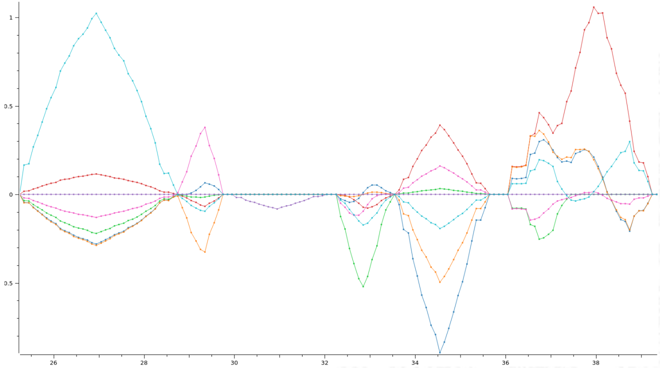


Figure: Velocity profile of task execution

Trajectory Blending

An annoying side-effect of kinematic path planning in stages is that motions stop between segments.

Approaches to blending:

- Position-based blending

$$\tau_b(t) = \tau_1(t) + \alpha(t) \cdot (\tau_2(t) - \tau_1(t))$$

- Quintic interpolation of candidate blend borders
- Limit-aware Trajectory Generation (reflexxes, TOTG, . . .)

All these approaches require additional feasibility-checking.

- Have path planners consider dynamics information in scenes

Drastically increases complexity

Scheduling

Current State

- The system plans single-threaded
- Simple round-robin scheduling of stages
- In stages, rank jobs depending on aggregated cost

Envisaged

- At least multi-threaded by stage
- Centralized worker scheduling
- Jobs to refine local solutions

Optimizer Backend

Instead of solving all stages as black boxes, they could also generate constraints for the trajectory segment.

- Compare approaches for complex trajectories at a sizable overhead
- Post-processing/refinement of task solutions

Learning Stages

The Task Constructor approach does not presume anything about the internal workings of a stage.

Ideal system for *including* a central learning/adaptive component, e.g., pluck a string.

Industry Feedback

Received quite a lot of industry feedback asking for more features.

- Python support
- Task serialization
- Integrated execution
- Script hooks (for suction control)
- Adaptive trajectories
- Optional early commitment
- Error recovery
- Iterative processes
- ...

Dantam, N. T., Kingston, Z. K., Chaudhuri, S., & Kavraki, L. E. (2016). Incremental Task and Motion Planning: A Constraint-Based Approach.. In *Robotics: Science and systems*. Ann Arbor, MI, USA.

Görner, M., Haschke, R., Ritter, H., & Zhang, J. (2019). MoveIt! Task Constructor for task-level motion planning. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE.

Hauser, K., & Latombe, J.-C. (2010). Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, *29*(7), 897–915.

Wang, Y., Ajaykumar, G., & Huang, C.-M. (2020). See What I See: Enabling User-Centric Robotic Assistance Using First-Person Demonstrations. In *Proceedings of the 15th annual ACM/IEEE international conference on Human-Robot Interaction*. ACM.

Thank You for Listening. Questions? Ideas?

Michael Görner, Robert Haschke
goerner@informatik.uni-hamburg.de

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Technical Aspects of Multimodal Systems