2019 International Conference on Robotics and Automation (ICRA)
Palais des congres de Montreal, Montreal, Canada, May 20-24, 2019

MoveIt! Task Constructor for Task-Level Motion Planning

Michael Görner*, Robert Haschke*, Helge Ritter, Jianwei Zhang

***Abstract*— A lot of motion planning research in robotics focuses on efficient means to find trajectories between individual start and goal regions, but it remains challenging to specify and plan robotic manipulation actions which consist of *multiple interdependent* subtasks. The Task Constructor framework we present in this work provides a flexible and transparent way to define and plan such actions, enhancing the capabilities of the popular robotic manipulation framework *MoveIt!*.[1] Subproblems are solved in isolation in black-box planning stages and a common interface is used to pass solution hypotheses between stages. The framework enables the hierarchical organization of basic stages using *containers*, allowing for sequential as well as parallel compositions. The flexibility of the framework is illustrated in multiple scenarios performed on various robot platforms, including bimanual ones.**

I. Introduction

Motion planning for robot control traditionally considers the problem of finding a feasible trajectory between a start and a goal pose, where both are specified in either joint or Cartesian space. Standard robotic applications, however, are usually composed of multiple, interdependent sub-stages with varying characteristics and sub-goals. To find trajectories that satisfy all constraints, all steps need to be planned in advance to yield feasible, collision-free, and possibly cost-optimized paths.

Typical examples are pick-and-place tasks, that require (i) finding a set of feasible grasp and place poses, and (ii) planning a feasible path connecting the initial robot pose to a compatible candidate pose. The latter, in turn, involves approaching, lifting, and retracting – performing well-defined Cartesian motions during these critical phases. As there typically exist several grasp and place poses, any combination of them might be valid and should be considered for planning.

Such problems present various challenges: Individual planning stages are often strongly interrelated and cannot be considered independently from each other. For example, turning an object upside-down in a pick-and-place task renders a top grasp infeasible. Whereas some initial joint configuration might be adequate for the first part of a task, it could interfere with a second part due to inconvenient joint limits.

The present work proposes a framework to describe and plan composite tasks, where the high-level sequence of actions is fixed and known in advance, but the concrete realization needs to adapt to the environmental context.

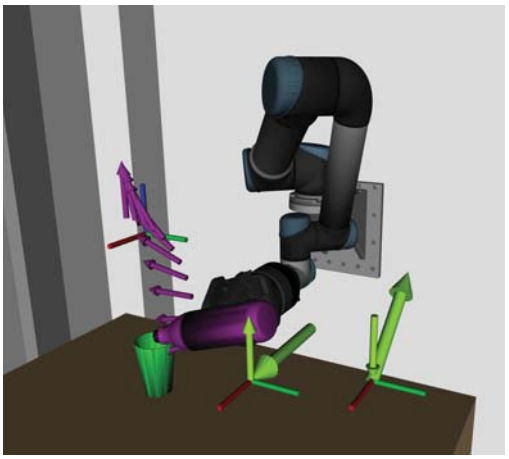


Fig. 1. Example task: a UR5 robot executes a task composed of (a) picking up a bottle from the table, (b) pouring liquid into a nearby glass, and (c) placing the bottle in a different location. Markers (arrows and frames) show key aspects of the task, including approach and lift directions during (a), bottle poses for (a) and (c), and the tip of the bottle during (b). Fig. 5 illustrates the associated task structure.

With this, we aim to fill a gap between high-level symbolic task planning and low-level, manipulation planning, thus contributing to the field of *Task and Motion Planning*.

Within the framework, *tasks* are described as hierarchical tree structures providing both sequential and parallel combinations of subtasks. The leaves of a task tree represent primitive planning stages, which are solved by arbitrary motion planners integrated within MoveIt!, thus providing the full power and flexibility of MoveIt! to model the characteristics of specific subproblems. To account for interdependencies, stages propagate the world state of their sub-solutions within the task tree. Efficient schedulers are proposed to first focus search on critical parts and cheap-to-compute stages of the task and thus retrieve cost-economical solutions as early as possible. Continuing planning can improve the quality of discovered solutions over time, taking into consideration all generated sub-solutions.

Additionally, the explicit factorization into distinct stages and world states facilitates error analysis: individual parts of the task can be investigated in isolation, and key aspects of individual stages can be visualized easily. Fig. 1 illustrates an example task with supporting visualizations.

II. Related Work

The scope of this work lies between two fields of research. On the one side, *manipulation planning* emphasizes the problem of trajectory planning with multiple kinematic and dynamic constraints [1], [2]. These approaches can cope with

* These authors contributed equally to this work.

This work was supported by the DFG Center of Excellence EXC 277, the DFG Transregional Research Centre CML, TRR-169, and has received funding from EU project SaraFun (grant 644938).

[1]The Task Constructor framework is publicly available at `https://github.com/ros-planning/moveit_task_constructor`

multiple constraints for a single task, but usually do not factorize efficiently into comprehensive subproblems.

On the other side, the symbolic *task planning* community has long realized that reasoning about robotic actions has to consider geometric constraints at planning level, forming the field of *task and motion planning* [3]–[6]. While these approaches demonstrate impressive show-cases, solving complex, puzzle-like scenarios, they are often too generic and very complicated to configure for concrete use cases. As a consequence of their task planning approach involving both, symbolic- and geometry-level planning simultaneously, these systems are vulnerable to small changes in parameterization, strongly depend on an accurate and consistent domain representation, and can exhibit behavior that is formally adequate, but surprising to humans. Interrelations between various subtasks either need to be modeled explicitly in the symbolic domain or many potential sub-solutions have to be rejected afterwards. As a consequence, the underlying backtracking-based search is inefficient and it is difficult to exploit the local structure of a specific problem.

Instead of solving such generic task problems, we focus on the common subproblem of finding feasible sequences of trajectories given that the high-level action sequence is already known in advance (with the notable exception of well-defined alternative pathways). The assumed action sequence could be compared to the *plan skeletons* defined in [7]. Whereas the authors propose methods to convert action sequences into discrete-space constraint satisfaction problems, we utilize traditional motion planning algorithms to find solutions in continuous space.

Another closely related work was presented in [8]. They present the roadmap-based planning algorithm *Multi-Modal-PRM* to generate trajectories across multiple different configuration manifolds. Whereas they emphasize the integrated planning procedure, relying on existing manifold specifications, our work focuses on these specifications instead. We explicitly treat the planning process of primitive stages as black-boxes, to enhance modularity and developer-insight through the explicit exchange of generated world states.

In MoveIt!, the composition of multiple planning steps is partially supported by the manipulation stack. However, this API is limited to sequential pick-and-place requests that are planned for in a greedy fashion, often resulting in poor-quality or no solutions at all.

The Descartes planning library [9] follows a strategy similar to this work, searching all possible paths in a graph formed by sets of consecutive goal states. In contrast to Descartes, which restricts itself to Cartesian paths through fixed waypoints, this framework allows for arbitrary motion planning stages as well as arbitrary complex hierarchies.

For shared robot control, *affordance templates* [10] provide the structure and visualization to implement single object-centered tasks, like turning a valve, through multiple Cartesian end-effector waypoints. While their implementation employs greedy forward-planning to solve for Cartesian trajectories, the constructed task specifications could be used as Cartesian planning stages within this work.

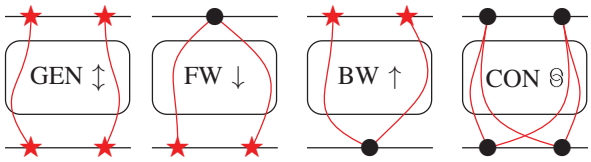


Fig. 2. Stage types distinguished by their interface: a) generators, b), c) forward and backward propagating stages, d) connecting stages. Black dots indicate input states, red stars indicate newly spawned states.

III. Task Constructor Framework

A. Task Description

Within this framework, tasks are composed in a hierarchical fashion from primitive planning *stages* that describe atomic motion planning problems that can be solved by existing motion planning frameworks like OpenRAVE [11], Klamp't [12] or MoveIt! [13]. These frameworks typically allow for motion planning from a single start to a goal configuration, which both are usually fully-specified in configuration space. Often they also permit to specify *goal regions*, both in configuration and Cartesian space, and appropriate state samplers are employed to yield discrete configuration-space states for planning.

Individual planning stages communicate their results via a common interface using a MoveIt! *planning scene* to describe the whole state of the environment relevant for motion planning. This comprises the shapes and poses of all objects, the collision environment, all robot joint states, and information about objects attached to the robot. This geometric/kinematic state description can be augmented by additional semantic information in terms of typed, named properties, forming the final *state* representation. Each stage then attempts to connect states from its start and end interfaces via one or more planned trajectories.

Container stages allow for hierarchical grouping of stages. Depending on the type of the container, solutions found by its children are converted to compound solutions and propagated up the task hierarchy (for more details, refer to section III-D).

B. Stage Interface Types

We distinguish stages based on their interface type (see Fig. 2). The traditional planning stage is the *connecting stage*, which takes a pair of start/end states and tries to connect them with a feasible solution trajectory. This type of planning stage often corresponds to *transit motions* that move the robot between different regions of interest. In this case, any combination of states from the start and end interfaces is considered for planning, realizing an *exhaustive* search. As such a planning stage only affects a small set of active joints usually, a pair of start and end states need to match w.r.t. all other aspects of the state representation. Particularly, all other joints as well as the number, pose, and attachment status of collision objects need to match.

The second type, *generator stages*, populate their start and end interfaces from scratch, without any explicit input

from adjacent stages. Usually, they define key aspects of an action, for example defining the initial robot state or a fixed goal state, which subsequently can serve as input for adjacent stages. Another example is a grasp generator, which provides pairs of pre- and final grasp poses, computing their corresponding robot poses based on inverse kinematics. In this case, generated start and end states usually differ and are connected by a non-trivial joint trajectory (provided by the grasp planner) to accomplish actual grasping.

The most common type of stages are *propagators*, which read an input state from either its start or end interface, plan to fulfill any predefined goal or action, and finally spawn one (or more) new state(s) at the opposite interface together with a trajectory connecting both states.

Note that propagation can act in both directions, from start to end as well as from end to start. For this reason, it is important to distinguish the *temporal* from the *logical* flow. The temporal flow is always from a start to an end interface and defines the temporal evolution of a solution trajectory. However, the logical (program) flow defines the state information flow during planning and is determined by the propagation direction of individual stages. Backward propagation allows for planning a relative motion to reach a given end state from a yet unknown start state. A typical example is the Cartesian approach phase before grasping: Here the final grasp pose is given, and a linear approach motion to the pre-grasp pose needs to be found, whose extent is only coarsely specified within a range of several centimeters. Corresponding solutions are planned in reverse direction, from the end towards the start state. Finally, the solution is reversed to yield a trajectory properly evolving in time from start to end.

Obviously, the interface types of stages constrain how they can be sequenced: A stage *spawning* new states along one direction (forward / backward) should be followed/preceded by a stage that *reads* from the shared interface and vice versa. Otherwise, the logical information flow would be broken. The framework provides mechanisms for automatic derivation and validation of the connectivity of a specified Task prior to any planning and thus can reject wrongly configured task trees already at configuration time.

C. Available Primitive Stages

The Task Constructor library provides a connecting stage and two basic propagating stages, which all are driven by individual planner instances. We decided to decouple the planning from the stage implementation to increase modularity and facilitate code reuse. While stages *specify* a subtask, i.e., which robot states to connect, planners perform the actual work to find a feasible trajectory between these two states. Hence, planners can be reused in different stages. Two basic planning instances are provided: (i) MoveIt's planning pipeline offering wrappers for OMPL [14], CHOMP [15], and STOMP [16]; and (ii) a Cartesian path generator based on straight-line Cartesian interpolation and validation.

The two propagating stages allow for (i) absolute and (ii) relative goal pose specification, either in joint or Cartesian space. While in the former case, the goal pose is specified in an absolute fashion w.r.t. a known reference frame, the latter case permits specifying relative motions of a specific end-effector link. In the general case, a twist motion (translation direction and rotation axis) is specified w.r.t. an arbitrary known reference frame and finally applied to the given end-effector link. This representation makes it possible, for example, to specify a linear approach or lifting motion relative to the object or a global reference frame.

Generator stages provided are: (i) the *current state* stage fetching the current planning scene state from MoveIt!'s `move_group` node, and (ii) the *fixed state* stage allowing to specify an arbitrary, predefined goal state.

In some cases, the sequential information within the task pipeline is too restrictive to specify a task: Particularly, generator stages might depend on the outcome of another, *non-neighboring* stage, thus necessitating a short-cut connection within the task pipeline. For example, to place an object after usage at the original pick location, the corresponding place-pose generator needs access to the original pick pose. To allow for such short-cuts, generators can subscribe to solutions found by another stage.

D. Available Containers

As mentioned before, container stages are used to *hierarchically* compose stages into a tree. Each container encapsulates and groups a set of children stages performing some semantically coherent subtask, e.g., grasping or placing. Children stages can easily inherit properties from their parent, thus reducing the configuration overhead. Two main types are distinguished: serial and parallel containers.

Serial containers organize their children into a linear sequence of subtasks which need to be performed in the specified order to accomplish the overall task of the container. Accordingly, a solution of a serial container connects a state from the start interface of the first child stage to the end interface of the last child via a *fully-connected*, multi-stage trajectory path.

In a sequential pipeline, generator-type stages play a particularly important role: They generate *seed* states, which subsequently are extended (in both directions) via propagating stages to form longer partial solution paths. Finally, connecting stages are responsible for linking individual partial solution paths to yield a fully-connected solution ranging from the very beginning to the very end of the pipeline.

Note that in general there can be multiple paths connecting a single pair of start-end states and there can be multiple solutions corresponding to different pairs of start-end states. Hence, it becomes essential to rank all found solutions according to a task-specific cost function (see Sec. III-E).

Parallel containers allow for planning of several alternative solutions, e.g., grasping with the left or right arm. Each solution found by its children directly contributes to the shared pool of solutions of the container. Different types of parallel containers are distinguished, depending on the planning strategy for children:

(i) *Alternatives*: Consider all children in parallel. All generated solutions become solutions of the container.

(ii) *Fallbacks*: Consider children sequentially, only proceeding to the next child if the previous one has vainly searched its solution space. Only solutions found by the first successful child constitute the solutions of the container.

(iii) *Independent Components*: consider all children in parallel. In contrast to (i), children generate solutions for *disjoint* sets of robot joints (e.g., arm and hand), which are subsequently *merged* into a single coherent trajectory performing all sub-solutions in parallel. Obviously, such a merge might fail and explicit constraint checks (including collision checking) are required for final validation. This divide-and-conquer approach is particularly useful, if the planning spaces of individual children are truly independent, as for example in approaching an object for bimanual grasping. In this case, the motion of both arms can be planned independently in lower-dimensional configuration spaces. To enforce independence, one may introduce additional constraints, e.g., a plane separating the Cartesian workspaces of both arms. This task-specific knowledge needs to be provided with the task specification.

E. Scheduling

The proposed framework exhaustively enumerates all possible solution paths connecting individual interface states, which obviously suffers from combinatorial explosion. Thus, scheduling heuristics are applied to focus the search on promising solution paths first.

To this end, solutions have an associated cost that is computed in a task-specific fashion by user-defined cost functions. Potential functions include, among others, length of trajectory, amount of Cartesian or joint motion, minimum or average clearance. Serial container stages accumulate the costs of all sub-solutions of a full path and only report the minimal-cost path for any pair of start-end states. In a similar fashion, parallel containers only report minimal-cost solutions of their children. Each stage, and particularly the root stage of the task tree, can then rank their solutions according to this cost and stop planning when an acceptable overall solution is found.

Each stage ranks all its incoming interface states according to (i) the length and (ii) cost of the associated partial solution. The former criterion biases the search to depth-first (in contrast to breadth-first), which ensures finding full solutions as soon as possible. If a partial solution fails to extend at either end, this failure propagates to the other end, and the corresponding interface states are removed from the interfaces of the associated stages as there is no benefit in continuing work on that particular solution.

Additionally, containers handle the scheduling of their children stages. Again the serial container plays the most crucial role for this. Generators need to be scheduled first in order to generate seed states, which subsequently are extended via propagating stages, and finally connected to full solution paths. Obviously, scheduling of connecting stages should be postponed as long as possible, because their pair-wise combination of start-end states leads to a combinatorial explosion of the search space and thus their planning attempts should be limited to the most promising candidate pairs only.

On top of these heuristics, there is room for further optimization. For example, one could try to balance the expected computation time vs. the expected connection success (or reduction in overall trajectory cost) by ranking stages according to the ratio of these values. To yield estimates for them, one could consider heuristic measures (e.g., joint or Cartesian-space distance of states), or maintain statistics over previous stage executions. To yield higher diversity and randomization, the actual ranking can be based on the Boltzmann distribution of the computed performance rank.

F. Execution

The main contribution of this work lies in modeling and planning manipulation tasks. Nonetheless, a solution should be executed on the actual robot, eventually. Traditionally, planning research simply forwards the final solution trajectory to a low-level controller. To this end, the proposed framework provides utilities to access planned task solutions, such that the user can decide whether to execute, for example, (i) the first valid solution, (ii) the first solution below some cost threshold, or (iii) the best trajectory found within a given amount of time or after exhaustively searching the full solution space. Modifications to the world state performed as part of the task, e.g., attaching or releasing an object, are performed in the same fashion as trajectories are executed, thus ensuring a consistent world representation throughout task execution.

Given the modularity of the task pipeline, several improvements are possible. Assuming feasible trajectories for the whole task will be found eventually, initial stages (or groups of stages) could commit early to a particular *partial* solution and forward it for execution *before* a full solution trajectory is found. As a consequence, this strategy can noticeably reduce the perceived planning time as the robot can start to move early. This is particularly useful when initial stages only yield a single canonical solution, but can also be used to significantly prune the search space, assuming full solutions will be available for most early sub-solutions.

To handle failures during task execution (e.g., due to dynamical changes in the environment, or because an early executed partial solution eventually turns out to be incompatible with later planning stages), a recovery strategy is essential. Again, the modular structure of the task pipeline can be exploited for intelligent recovery, dependent on the failed sub-stage. Potential strategies might replan from the reached stage, or partially revert sub-solutions to continue planning from a well-defined state.

In the future, it should also be possible to specify different execution controllers (or parameterizations) for individual stages (or groups of stages) to account for different control needs. For example, an approach stage might employ visual servoing to account for perception inaccuracies, and a grasp stage should use a compliant motion strategy until contact is

established and subsequently switch to force-controlled grasp stabilization. As long as the motion of these reactive, sensor-driven controllers remains within specified bounds to the planned trajectory, subsequent stages can connect seamlessly.

Finally, solution segments found by individual planning stages can be post-processed to yield a globally smooth solution trajectory. This requires local modifications at the transition between consecutive segments as they might have discontinuous velocity or acceleration profiles. To this end, acceleration-aware trajectory generation [17] can be applied to splice sub-trajectories smoothly within position bounds. The resulting trajectory segments might only replace the original solutions if they satisfy collision checks and other constraints.

G. Introspection

As pointed out in [13], a key element for the success and acceptance of a software package is its transparency and ease of use. Although MoveIt! comes with its own implementation of a manipulation pipeline, its major drawback is its intransparency: the provided pick and place stages are black boxes that do not allow for inspection of their inner workings.

Hence, essential elements of the presented software package are pipeline validation, error reporting, and introspection. Stages can publish both successful and failed solution attempts and augment them with arbitrary visual markers or comments, thus providing useful hints for failure analysis. This information, together with the status of the overall planning progress of the pipeline (number of successful and failed solution attempts per stage) is regularly published.

In rviz, the user can monitor the status of the task pipeline and interactively navigate individual solutions of all stages, inspecting their associated markers and comments. In the future, it is also planned to provide an interactive GUI to configure, validate, execute, and finally save a planning pipeline directly in rviz.

IV. Applications

In the following, we describe two typical manipulation tasks and showcase involved planning stages. The first task considers picking up an object. The second task demonstrates a pouring task, which involves picking up a bottle, approaching a glass, performing the pouring, and placing the bottle back on the table. The accompanying video shows that the very same task pipeline can be employed on various robots.

A. Bi-Modal Object Picking

As picking up an object is a common subtask for many manipulation tasks, a dedicated stage is provided for this. To apply this stage to a specific robot, only some key properties need to be configured, namely the end-effector to use, the name of the object to grasp, and the intended approach and retract directions. The actual grasping is planned by another generic stage, the grasp stage, which is provided as a configurable child stage to the pick template.

In the example shown in Fig. 3, we consider dual-handed robots, which can use either their left or right hand for grasping. Consequently, the pipeline comprises two alternative

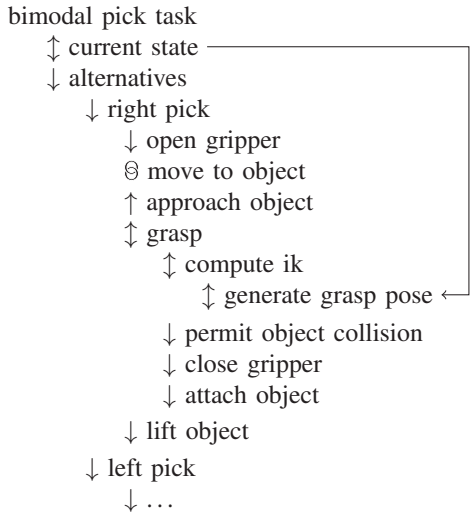


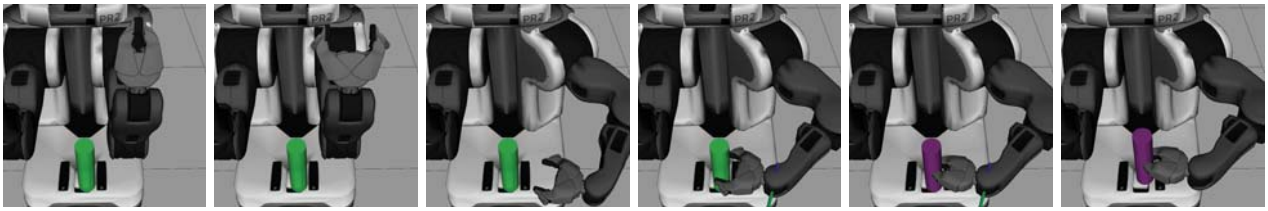
Fig. 3. Bi-modal Pick Task: Left / Right arm is chosen by parallel container *alternatives*. The propagation direction of planning states and solution monitoring are indicated by arrows. Hierarchy is indicated by indentation.

pick stages (*left* and *right*), configured to use the respective end-effector. The *alternatives* parallel container follows the *current state* generator, which fetches the current planning scene state from MoveIt!.

Planning for the pick stages starts with the *grasp* generator stage and proceeds in both directions: The *approach* stage realizes a Cartesian, straight-line approach motion, starting from a pre-grasp posture and is planned *backwards* to find a safe starting pose for grasping (see Fig. 4). On the opposite side, the *lift* stage starts from the grasped object state and realizes the Cartesian lifting motion in a *forward* fashion.

The grasp stage, in our simple scenarios, samples collision-free pre-grasp poses relative to the object at hand, computes the inverse kinematics to yield a joint-state pose suitable for use in the interface state, and finally performs grasping by closing the gripper. To this end, first collision detection between end-effector and object is turned off to allow the end-effector to contact or penetrate the object when actuating the grasp pose. For real-world execution, the *close gripper* stage obviously requires a force-controlled or compliant controller to avoid squashing the object. Finally, the object is attached to the end-effector link, such that further planning knows about the grasped-object state. These helper subtasks, which only modify the planning scene state, but do not actually perform any planning, are realized by utility stages, which permit to change allowed collisions as well as to attach and detach collision objects.

The sampling of pre-grasp poses, in our examples, considers a pre-defined open-gripper posture for the end-effector and proposes Cartesian poses of the end-effector relative to object-specific grasp frames. We sample grasp frames T_w^g by rotating the object frame about its z-axis in steps of 0.2 rad, resulting in 32 grasp frame samples. The end-effector is placed relative to these grasp frames by applying



initial | gripper opened | pre approach | pre grasp | post grasp | lifted

Fig. 4. Temporally ordered sequence of planning scene states of the pick task shown in Fig. 3.

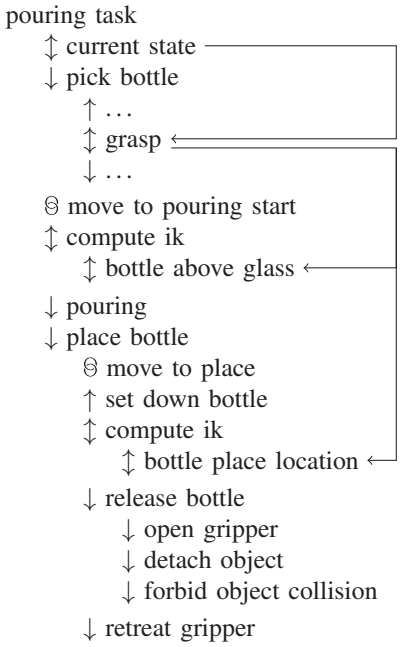


Fig. 5. Pouring Task: The manipulator picks a bottle, performs constrained pouring motions and places it back on the table. All stages (except pouring) are provided by the framework.

the inverse of a fixed tool-to-grasp transform T_t^g: $T_w^t = T_w^g \cdot T_g^t$. The resulting transform T_w^t is used as the target for inverse kinematics. Before applying inverse kinematics sampling, the IK stage validates the feasibility of the targeted pose, i.e., whether placing the end-effector at the target is collision-free. If not, IK sampling is skipped and a failure is reported immediately. While the first solution on all studied robots is found within a fraction of a second, the planning time for exhaustive search clearly varies between all studied robots and is dominated by the number of sampling-based planning attempts (in stage *move to object*), which in turn is determined by the number of solutions found by the *grasp* stage. While Pepper, having only a 5-DoF arm, finds a single feasible grasp pose only (< 1 s), the Baxter robot finds more than 60 solutions (≈ 45 s).

B. Pouring Task

The second described application demonstrates the use of the task pipeline with a custom stage, using the example of pouring into a glass. Its specification is shown in Fig. 5, its execution on a UR5 robot is illustrated in Fig 1 and the accompanying video. While the scenario requires a custom *pouring* stage, all other stages are realized with suitably parameterized standard stages provided by the planning framework. The task reuses the previously described *pick* container to pick up the bottle. A similar container *place* provides the generic subtask to compute place motion sequences, given a generator for feasible place poses.

The *pouring* stage is implemented as tilting the tip of an attached object (the bottle) in a pouring motion over another object (the glass) for a specific period of time. A Cartesian planner solves the path along object-centric waypoints.

The four generator stages involved in this task are interrelated: the two last ones, *bottle above glass* and *place location*, depend on the grasp pose chosen in the *pick* stage. To this end, they monitor the solutions generated by the *grasp* stage and produce matching solutions.

Lastly, moving the bottle over the glass and moving it towards its place location are transit motions that have to account for an additional path constraint, namely keeping the bottle upright to avoid spilling of the liquid. This constraint is specified as part of the stage description and is passed on to the underlying trajectory planner. To accelerate planning with the constraint, we make use of configuration space approximations [18] implemented for OMPL-based solvers.

In our experiments, using sequential planning, the task produces its first full solution after 15.6s on average.

V. Summary

We presented a modular and flexible planning system to fill the gap between high-level, symbolic task planning and low-level motion planning for robotic manipulation. Given a concrete task plan composed of individually characterized sub-stages, our system can yield combined trajectories that achieve the whole task. Failures can be readily analyzed by visualization and isolation of problematic stages. A number of generic planning stages are already in place and were employed to demonstrate the potential of the framework for use on multiple robotic platforms. The Task Constructor is meant to enhance the functionality of the MoveIt! framework and replace its previous, severely limited pick-and-place pipeline. The open-source software library is under continuous development and various extensions were outlined directly within the corresponding sections.

REFERENCES

[1] M. Stilman, "Global manipulation planning in robot joint space with task constraints," *IEEE Transactions on Robotics*, vol. 26, no. 3, pp. 576–584, 2010.

[2] D. Berenson, S. S. Srinivasa, D. Ferguson, and J. J. Kuffner, "Manipulation planning on constraint manifolds," in *2009 IEEE International Conference on Robotics and Automation*, May 2009, pp. 625–632.

[3] J. Bidot, L. Karlsson, F. Lagriffoul, and A. Saffiotti, "Geometric backtracking for combined task and motion planning in robotic systems," *Artificial Intelligence*, vol. 247, pp. 229 – 265, 2017, special Issue on AI and Robotics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437021500051X

[4] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 639–646.

[5] J. Ferrer-Mestres, G. Francès, and H. Geffner, "Combined task and motion planning as classical ai planning," *arXiv preprint arXiv:1706.06927*, 2017.

[6] F. Gravot, S. Cambon, and R. Alami, "aSyMov: a planner that deals with intricate symbolic and geometric problems," in *Robotics Research. The Eleventh International Symposium*. Springer, 2005, pp. 100–110.

[7] T. Lozano-Pérez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 3684–3691.

[8] K. Hauser and J.-C. Latombe, "Multi-modal motion planning in non-expansive spaces," *The International Journal of Robotics Research*, vol. 29, no. 7, pp. 897–915, 2010.

[9] S. Edwards, R. Madaan, and J. Meyer, "Descartes planning library for semi-constrained cartesian trajectories," ROSCon, 2015. [Online]. Available: http://wiki.ros.org/descartes

[10] S. Hart, P. Dinh, and K. Hambuchen, "The affordance template ROS package for robot task programming," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE, 2015, pp. 6227–6234.

[11] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010. [Online]. Available: http://www.programmingvision.com/rosen_diankov_thesis.pdf

[12] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes," in *Robotics Research*. Springer, 2016, pp. 357–373.

[13] D. Coleman, I. A. Şucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a MoveIt! case study," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 3–16, May 2014. [Online]. Available: http://moveit.ros.org

[14] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[15] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE, 2009, pp. 489–494.

[16] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 4569–4574.

[17] T. Kröger, *On-Line Trajectory Generation in Robotic Systems: Basic Concepts for Instantaneous Reactions to Unforeseen (Sensor) Events*. Springer, 2010, vol. 58.

[18] I. A. Şucan and S. Chitta, "Motion planning with constraints using configuration space approximations," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 1904–1910.