MoveIt! Task Constructor for task-level motion planning

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

***Abstract*— While a lot of motion planning research in robotics focuses on efficient means to find trajectories between individual start and goal regions, 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 plan such actions, enhancing the capabilities of the popular robotic manipulation framework *MoveIt!*.[1] Whereas subproblems are still solved in individual planning stages, a common interface makes it possible 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. In order 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.

A typical example 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 with compatible candidate poses. This in turn involves approaching, lifting, and retracting the end-effector – 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. Eventually, an exhaustive search can yield *optimal* task solutions w.r.t. some cost function.

Problems like these 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 could be optimal for the first part of a task, it might 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 be adapted to the environmental context. With this, we aim to fill a gap between high-level, symbolic task planning and low-level, manipulation planning. *Tasks* are described as hierarchical tree structures providing both sequential and parallel combinations of subtasks. The leaves of a task tree represent primitive stages that can be solved by arbitrary MoveIt! planners, providing the full power and flexibility of the framework. 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 in an exhaustive fashion guarantees finding cost-optimal solutions within the configuration space spanned by the task description.

Additionally, the explicit factorization into well-defined 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.

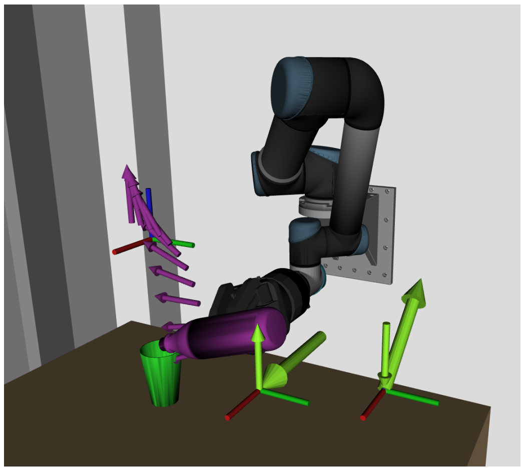

Fig. 1. Example task: a UR5 robot executes a task composed of (a) picking up a bottle from the table, (b) pouring liquid into a nearby glass, and (c) placing the bottle in a different location. Markers show key aspects of the task, including approach and lift directions during (a), bottle poses for (a)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

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

* 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`

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 complex 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 parameterization issues, strongly depend on an accurate and consistent domain representation, and can exhibit behavior that is hard to predict. Interrelations between various subtasks either need to be modeled explicitly in the symbolic domain or many potential sub-solutions will be rejected afterwards. As a consequence, the underlying backtracking-based search is inefficient and early-reported plans are often highly suboptimal.

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 can 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.

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 [8] follows a similar strategy as this work, namely doing an exhaustive search over all possible paths in a graph formed by sets of consecutive goal states. In contrast to Descartes, which restricts itself to Cartesian path validation through fixed waypoints, the proposed framework allows for arbitrary motion planning stages as well as arbitrary complex hierarchies.

For shared robot control, *affordance templates* [9] 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, using a subset of capabilities of the Task Constructor.

III. Task Constructor Framework

A. Task Description

In MoveIt! Task Constructor, 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 [10], Klamp't [11] or MoveIt! [12]. 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.

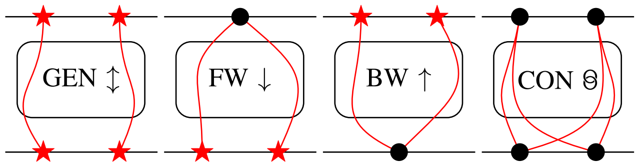

Fig. 2. Stage types distinguished by their interface: a) generators, b, c) forward and backward propagating stages, d) connecting stages. Black dots indicate input states, red stars indicate newly spawned states.

In line with this approach, we chose to represent the communication interface between individual planning stages by a MoveIt! *planning scene* that describes the whole state of the environment relevant for motion planning. This comprises the shape and pose of all objects, 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 solution 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. Primitive Stage Types

We distinguish a number of basic types of primitive stages based on their interface type (see Fig. 2). The classical planning stage is the *connecting stage*, which takes a pair of states from its start and end interface 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 of the start and end interfaces is considered for planning, realizing an exhaustive search through solution paths. As a primitive planning stage will only affect a small set of active joints, 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*, initially populate stage interfaces by generating new states into both, their start and end interfaces simultaneously. These states can be determined without prior start or end postures and usually define key aspects of an action. Examples are stages that generate the current robot state or a predefined goal state, which then can serve as input for adjacent stages. Another example are grasp generators, which provide various pairs of pre- and final grasp poses, computing their corresponding robot poses based on inverse kinematics. In this case, start and end states might differ (if pre- and final grasp poses do), and will be connected by a non-trivial joint trajectory to accomplish actual grasping.

The most common stage type are *propagators*, which read an input state from either its start or end interface, plan to fulfill defined goals or actions, and finally generate one (or more) new state(s) at the opposite interface together with a solution 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. Backwards propagation allows for planning a relative motion to reach a given end state from a yet unknown start state. A common example is the Cartesian approach phase before grasping: Here the final grasp pose at the object is given, and a linear approach motion to the pre-grasp pose needs to be found, whose distance is only coarsely specified within a range of several centimeters. Corresponding solutions can be planned in reverse direction, taking the end state as input and planning towards the start state. Finally, a found solution is simply reversed to yield a trajectory properly evolving in time from start to end.

C. Basic 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 [13], CHOMP [14], and STOMP [15]; 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 makes it possible for example to specify a linear approach or lifting motion relative to the object or a global reference frame.

Two basic generator stages are provided to define seeds for a planning pipeline: The *current state* stage fetches the current planning scene state from MoveIt!'s move-group node, and the *fixed state* stage permits to specify an arbitrary, predefined goal state.

Sometimes, the sequential information flow through the stage interfaces is not sufficient to specify a task: 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 exactly at the original pick location, the corresponding place-pose generator needs to access the original pick pose. To allow for such short-cuts, a special generator class is provided that receives a notification for every generated solution of another, monitored stage and subsequently can generate its own solutions dependent on the former ones.

D. Containers

As mentioned before, container stages are used to create a complex hierarchical task 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 executed one after the other in 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, generators 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 to link individual partial solution paths to yield a fully-connected solution ranging from the very beginning to the very end of the pipeline.

Obviously, based on the interface type of stages there exist some restrictions how they can be sequenced: A stage writing new states along one direction (forward / backward) should be followed / preceded by a stage that reads from the corresponding shared interface and vice versa. Otherwise, the logical information flow would be broken. Containers provide automatic validation of the connectivity of their children prior to any planning and thus can reject wrongly configured task trees during the configuration phase.

Note that in general there can be multiple paths connecting a single pair of start-end states of a serial container and there can be multiple solutions corresponding to different pairs of start-end states. Hence, it becomes important 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 left or right arm. Each solution found by its children directly contributes to the common pool of solutions of the container. Different types of parallel containers are distinguished, depending on the planning strategy for children: (i) planning all stages in parallel and returning the best solution according to some cost function; or (ii) planning stages one after the other, only progressing to the next child if the previous one exhaustively searched its solution space and failed. In the latter case, later children stages can be considered as *fallback alternatives*, which are only taken into account if prior options failed. Returning to the example of grasping with left or right arm, this either means (i) to consider both alternatives to be a-priori equivalent and finally choose based on overall costs, or (ii) to prioritize grasping with one arm and only falling back on the other, if that fails in any regard.

We have also implemented a **merging container**, which combines solutions of different children (involving disjoint sets of joints) into a single trajectory and – if it satisfies all collision and explicit constraints – propagates it as a feasible solution. This finally permits executing all sub-solution trajectories in parallel. This divide-and-conquer approach is particularly useful, if the planning spaces of individual children are truly independent of each other, 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, but subsequently merged and executed in parallel. To enforce independence during planning, one may introduce additional constraints, e.g. a plane separating the Cartesian work spaces of the left and right arm. However, this task-specific knowledge needs to be provided by the task programmer or a higher-level symbolic planner.

E. Scheduling

The proposed task planning pipeline achieves completeness by exhaustively enumerating all possible solution paths connecting individual interface states. Obviously, this suffers from combinatorial explosion and strategies to focus search on promising solution paths are required in order to yield cost-economical solutions as early as possible during the planning process.

To this end, all individual primitive solutions have an associated cost that is computed in a task-specific fashion using user-defined callback functions. Potential cost 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 between 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 do early stopping if a high-quality solution is found.

At a second scheduling level, stages rank their input interface states to decide which state is most promising to proceed with. This ranking not only considers (accumulated) costs of incoming / outgoing solution paths, but primarily also the *length* of partial solution paths. This only effects children of serial containers, whose goal is to find a fully-connected path. (Solutions of a single stage, primitive or container, always have length 1.) Obviously, longer partial solution paths should be preferred over shorter ones as they indicate more progress of the overall planning process. Solution paths of equal length are ranked by their accumulated costs. If, in a serial container, a partial solution path fails to extend at either end, this failure is propagated to the other end, and the corresponding interface state is removed from the interface of the associated stage as there is no benefit in continuing work on that particular solution.

Finally, at the highest level, containers decide which stages to schedule next for planning. Again the serial container plays the most important role for this. Indeed, 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, execution 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. Hence, only those partial solutions are considered that have maximal length, i.e. they reach to the start/end interface or the previous/next connecting stage in the sequence. This optimization is crucial to restrict the computational effort of the planning process, efficiently avoiding costly planning steps for candidate solutions that later would turn out infeasible.

Even though this seems to define a canonical execution order, there is room for further optimizations. 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 state configurations), or maintain statistics over previous stage executions. To yield higher diversity and randomization, actual ranking can be performed based on the Boltzmann distribution of the computed performance rank. These optimizations are not yet implemented, but planned for the very near future.

F. Execution

The main contribution of this work lies in modeling and planning manipulation tasks. Nonetheless, eventually a solution should be executed on the actual robot. 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 (i) the first valid solution, (ii) the best one found within some time frame, or (iii) the globally optimal trajectory after exhaustively searching the complete configuration space. Modifications to the world state performed during the task (e.g. attaching or releasing an object) can be accessed and applied in the same fashion as trajectories, to ensure a consistent representation during execution.

However, given the modular 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 to a particular *partial* solution and forward it for early execution *before* a full solution trajectory is found. This particularly makes sense if initial stages only yield a single canonical solution, but also will be useful if close-to-optimal sub-solutions become available. As a consequence, this strategy will noticeably reduce the perceived planning time as the robot starts to move early. Similarly to the selection process for full solution, the task programmer will define, when and which partial solution to choose for early execution. Aside from early execution, this approach will significantly prune the search space as further planning for early-committed stages can be dropped (or paused).

Obviously, a partial initial solution could eventually turn out to be incompatible to later planning stages. In this case a recovery strategy is needed and planning needs to continue also for early stages. Note that such a failure during execution can occur also in general, e.g. due to dynamical changes in the environment or due to deviations in path execution. Hence, an error recovery strategy is useful in any case. Again, the modular structure of the task pipeline will be useful to device meaningful recovery strategies – dependent on the failed sub-stage. Potential strategies could include replanning from the reached state, or partially reverting particular solution trajectories to continue planning from a well-defined stage, etc.

In the future, it should be also possible to specify different execution controllers (and parameterizations) for individual stages (or groups of stages) to account for different control needs. For example an approach stage should 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 should be post-processed to yield a globally smooth solution trajectory. This will require local modifications at the transition between consecutive segments as they might have discontinuous velocity or acceleration profiles. To support this, acceleration-aware trajectory generation [16] could be applied to splice sub-trajectories smoothly within position bounds. Obviously, the resulting trajectory segments have to be re-evaluated for collisions and other constraints.

G. Introspection

As pointed out in [12], a key element for the success and acceptance of a software package to new users is its transparency and ease of use. Although MoveIt! comes with its own implementation of a manipulation pipeline, its major drawback is its in-transparency: the provided pick and place stages are black boxes that do not allow for inspection of their inner workings. This renders debugging in case of failures a nightmare.

Hence, important elements of the presented software package are pipeline validation, error reporting, and introspection. To support failure analysis, stages can not only publish successful solutions, but also failed ones. While this alone

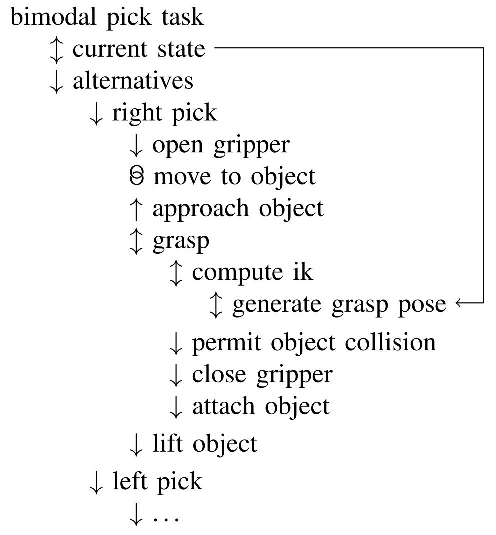

Fig. 3. Bi-modal Pick Task: Left / Right arm is chosen by parallel container *alternatives*. The propagation direction of planning states and solution monitoring are indicated by arrows. Hierarchy is indicated by indentation.

is not yet very useful, it allows us to monitor the number of failed planning attempts. However, the key feature is the possibility to augment a solution trajectory with arbitrary rviz markers and a comment providing useful hints why a planning attempt failed. 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 and visualized by a corresponding rviz plugin.

In rviz, the user can monitor the status of the task pipeline and interactively navigate individual solutions of all stages. Solution trajectories and associated markers are displayed and thus facilitate debugging. 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 pipeline can be employed to realize object grasping with various robots and resulting trajectories can be executed on physical robots.

A. Bi-Modal Object Picking

As picking up an object is a very common subtask for many manipulation tasks, a dedicated generic stage is provided for this. To apply this generic stage to a specific robot, only a few properties need to be configured, namely the end-effector to use, the name of the object to grasp, as well as the Cartesian 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 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 object* stage realizes a Cartesian, straight-line approach motion, starting from the desired pre-grasp posture and planning *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 a utility stage, which permits to modify the allowed collision matrix as well as attaching and detaching collision objects.

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. In our case, 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 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. However, 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 can be skipped and a failure 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 ($\approx$ 45 s).

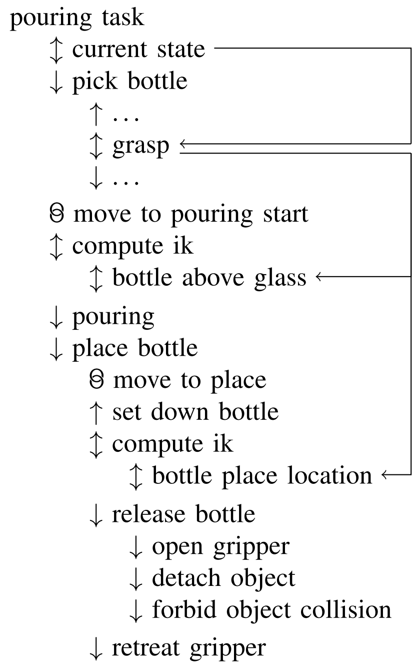

Fig. 5. Pouring Task: The manipulator picks a bottle, performs constrained pouring motions and places it back on the table.

B. Pouring Task

This application demonstrates the use of the task pipeline for custom applications, using the example of pouring into a glass. Execution on a UR5 robot is illustrated in Fig 1 and the accompanying video. While the scenario requires a specific pouring stage, most other stages are realized with suitably parameterized standard stages and provide a robust context for the central *pouring* component. The task reuses the previously described *pick* container to pick up the bottle. A similar container *place* provides a generic stage to compute place motion sequences, given a generator for feasible place poses.

The *pouring* stage is implemented as tilting the tip of a specific attached object in a pouring motion over another object in the scene for a specific period of time. The path is solved by a Cartesian planner along multiple object-centric waypoints, thereby providing some level of abstraction.

Although the example contains four generator stages, these are not at all independent. Instead, the two last ones, *bottle above glass* and *place location*, depend on the grasp pose chosen in the pick stage. To this end, both stages monitor the resulting solutions generated by the *grasp* stage and produce corresponding solutions.

Lastly, moving the bottle over the glass and moving it towards its place location are both transit motions that have to account for an additional path constraint, keeping the bottle upright to avoid spilling of the liquid. The constraint can be specified in these sub-stages and is passed on to the implementing trajectory planner. To accelerate planning with the constraint, we make use of configuration space approximations [17] implemented for OMPL-based solvers.

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

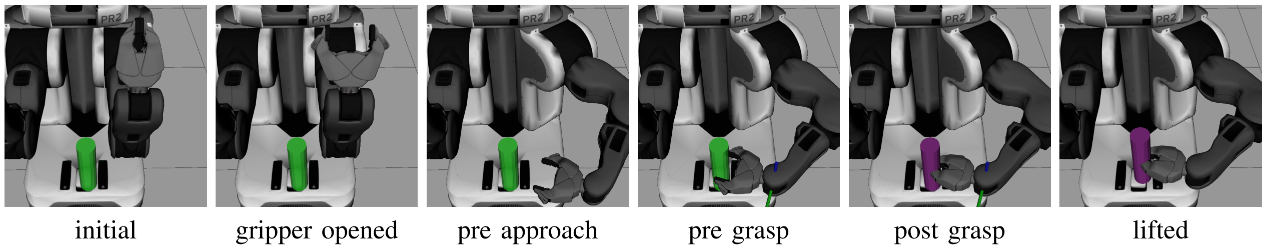

Fig. 4. Temporally ordered sequence of planning scene states of the pick task shown in Fig. 3.

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

[9] 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.

[10] 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

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

[12] 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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.