The Task-Motion Kit

By Neil T. Dantam, Swarat Chaudhuri, and Lydia E. Kavraki

An Open Source, General-Purpose Task and Motion Planning Framework

Robots require novel reasoning systems to achieve complex objectives in new environments. Daily activities in the physical world combine two types of reasoning: discrete and continuous. For example, to set the table in Figure 1, the robot must make discrete decisions about which and in what order to pick objects, and it must execute these decisions by computing continuous motions to reach objects or desired locations. Robotics has traditionally treated these issues in isolation. Reasoning about discrete events is referred to as *task planning*, while reasoning about and computing continuous motions is in the realm of motion planning.

However, several recent works have shown that separating task planning from motion planning, i.e., finding a series of actions that will later be executed through continuous motion, is problematic. For example, the next discrete action may specify picking an object, but there may be no continuous motion for the robot to bring its hand to a configuration that can actually grasp the object to pick it up. Instead, task-motion planning (TMP) tightly couples task planning and motion planning, producing a sequence of steps that can actually be executed by a real robot to bring the world from an initial to a final state. This article provides an introduction to TMP and discusses the implementation and use of an open-source TMP framework that is adaptable to new robots, scenarios, and algorithms.

TMP presents challenges both in algorithmic design and software engineering. Interaction between the discrete task component and the continuous motion component imposes requirements not faced by stand-alone task planners or motion planners. The planner may need to consider alternative task plans in an efficient way until finding one that can actually be executed by the robot at hand, whereas typical task planners generate only a single plan. In addition, actions where the robot grasps and rearranges objects will change the kinematics and configuration space in which the robot can move,

©ISTOCKPHOTO.COM/R_TYPE

Digital Object Identifier 10.1109/MRA.2018.2815081
Date of publication: 17 May 2018

whereas typical motion planners assume a fixed configuration space. Thus, we cannot expect to combine existing tools for isolated task planning and motion planning and produce frameworks that can consistently use high-level specifications of behavior to produce motion. Instead, we must handle the possible interactions of discrete and continuous components to identify task plans and executable motions.

TM Kit (TMKit) is an end-to-end system for probabilistically complete TMP and real-time execution. [Code and documentation are available at [21] under a permissive (BSD) license.] TMKit follows the high-level design shown in Figure 2 to implement the algorithm of [1] and [2] and at the same time provides a general framework to integrate multiple methods for task planning, motion planning, and TM interaction. Shared abstractions and data structures are fundamental aspects of TMKit that enable the coupling of task planning, motion planning, and real-time estimation and control. TMKit is modular and extensible, and we are adapting it to additional methods for TMP [3], [4]. Whenever appropriate, we employ widely used formats and protocols to promote compatibility. The resulting system generates real-time, collision-free robot motion from high-level specifications. To our knowledge, this is the first publicly available, general-purpose TMP framework. Sharing this project with the community will encourage the implementation of more TMP approaches and provide a valuable tool for the development and comparison of related techniques.

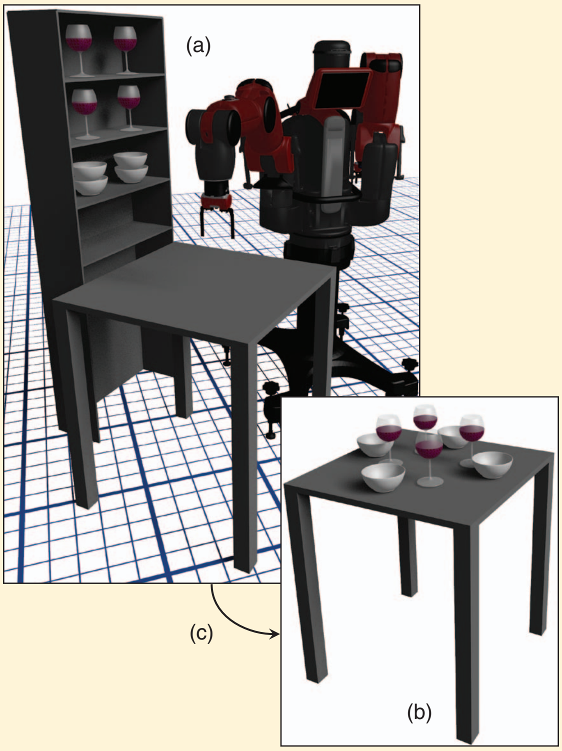


Figure 1. An example of a TMP problem: setting a table. The input for the TMP includes (a) the start state, (b) the goal state, and a set of allowable actions (e.g., pick, place, and so on). (c) TMP finds the output, which consists of a sequence of discrete actions (the task plan) and their corresponding continuous paths (or motion plans).

Background

Task Planning

Task planning identifies a sequence of discrete actions that change an initial state into a desired goal state or condition, given a task domain that defines the available actions and their preconditions and effects. This field evolved largely from pioneering work on the Stanford Research Institute Planning System [5]. The leading approaches for efficient task planning are heuristic search [6] and constraint satisfaction [7].

Off-the-shelf task planners typically focus on efficiently finding a single plan. In contrast, TMP often requires searching through multiple alternative task plans, as previously discussed. This raises an inherent challenge: motion planners that are used to compute paths are, at best, probabilistically complete for high-dimensional systems. Consequently, we cannot generally prove the nonexistence of corresponding motion plans. To address this challenge, our system does not use an off-the-shelf task planner but rather employs a newly introduced task planner capable of efficiently generating alternative plans.

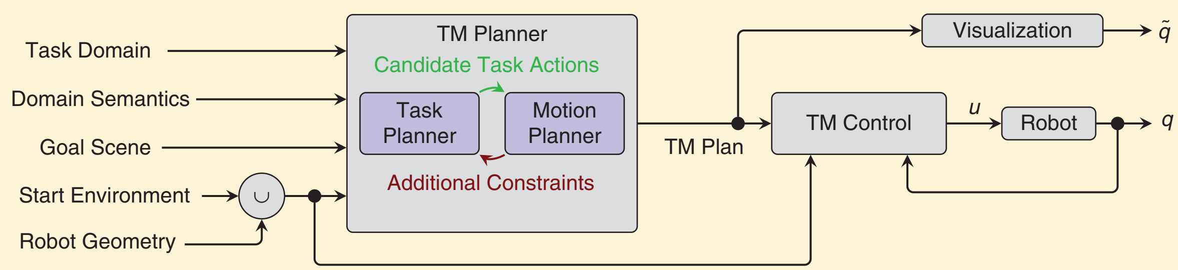


Figure 2. A high-level planning and execution block diagram. The inputs are the task domain definition; the environment and robot geometries, combined to produce the scene graph; and the domain semantics that relate the task and motion layers. The TM planner generates a plan based on these inputs. The TM control layer executes the plan, sharing a geometric representation—the scene graph—with the planning layer. The control output u drives the robot, resulting in configuration q. In a parallel layer, we visualize the system at simulated configuration $\tilde{q}$.

Motion Planning

Motion planning identifies a continuous path of valid configurations, i.e., joint positions, from an initial state to a desired goal state. Sampling-based motion planners are widely used for high-dimensional systems [8]. Such sampling-based planners offer probabilistic completeness, guaranteeing that the planner will eventually find a solution if one exists. However, if a solution does not exist, a sampling-based planner cannot prove this negative; in such a case, the planner would not terminate or would run until reaching a timeout. Motion planners based on gradient descent or optimization are also common and highly efficient, but they do not offer the same probabilistic completeness guarantees as the sampling-based motion planners. Consequently, this work uses such sampling-based planners because probabilistic completeness of the overall framework is a desired property. Conveniently, high-quality, open-source implementations of such planners are available [9]. Future integration of alternative motion planning approaches is possible, with their accompanying set of tradeoffs, but the integration of motion planners in TMP needs special attention to address the coupling of task planning and motion planning.

Off-the-shelf motion planning frameworks often abstract the details of robot kinematics or assume that the kinematic equations are fixed or change infrequently, with only configurations changing during planning [9]. In contrast, TMP requires rapid updates to kinematic equations. As the robot grasps and transfers objects, these objects' poses change between fixed values and functions of robot configuration. Moreover, these changes may involve more than just the individual grasped object, such as in the case of moving a tray or pushing a cart containing other objects. Consequently, kinematic representations capable of efficient updates are required for TMP.

Combining Task and Motion Planning

TMP takes an initial state to a desired goal state through the concurrent or interleaved production of high-level, discrete action sequences via task planning and continuous, collision-free paths via motion planning. Most prior work on TMP has focused on computational performance rather than completeness or generality, which are emphasized in this article. Lagriffoul and Andres [10] applied geometric constraints to limit the motion planning space or prove motion to be infeasible in special cases. Hierarchical Planning in the Now [11] interleaved planning and execution, reducing search depth but requiring reversible actions, e.g., rearranging objects but not pouring a cup down a drain, when backtracking. The work in [12] extends a hierarchical task planner with geometric primitives, using shared literals that relate task-level symbols with motion-level geometric entities. Gharbi et al. [13] interfaced an off-the-shelf task planner and motion planning using a heuristic method to remove objects that would potentially block the robot's path. The researchers in [14] formulated the motion side of TMP as a constraint satisfaction problem over a discretized, preprocessed subset of the configuration space. The Robosynth framework [15] uses a satisfiability modulo theories (SMTs) solver to generate task and motion plans from a static road map, employing plan outlines to guide the planning process. FFRob [16] developed a task-layer heuristic similar to the Fast-Forward planner [6] by using a lazily-expanded road map. Overall, these methods set aside the broad challenge of ensuring probabilistic completeness that arises from interactions between the task and motion layers. In contrast, the framework we implement focuses on probabilistically complete TMP.

A smaller number of task and motion planners do achieve probabilistic completeness. The aSyMov planner [17] combines a heuristic-search task planner with lazily expanded road maps. Our implementation of [1] and [2] in TMKit operates differently at the task, motion, and interface levels, yielding different performance characteristics than aSyMov. For example, aSyMov's composed road maps could be amortized over multiple runs, but composing road maps for object interactions may be expensive. In contrast, [1] and [2] find a new motion plan each run but efficiently update scene data structures to handle object interaction. Furthermore, TMKit is extensible to both forward-search [6] and constraint-based [7] task planners.

While source code is available for some specific methods, such as that in [13], we believe TMKit is the first publicly available framework that is extensible to multiple methods and domains. A key to this extensibility is our abstraction of the interaction between task and motion layers via the domain semantics, which enables the introduction of new actions and domains without any necessary changes to the framework itself.

Plan Execution

Motion planners make certain assumptions to achieve sufficient performance, and the execution step must correct those assumptions in real time. Specifically, motion planners typically assume 1) a given model for the kinematics and geometry of the robot and environment and 2) that motion between nearby joint configurations is possible. In reality, geometric models contain numerous errors due to imprecise lengths, encoder calibration error, flexing of assumedly rigid bodies, inaccurate object detection, inaccurate camera calibration, and so forth. Thus, despite the precision or repeatability of many robots, accurate motion to correct poses still presents challenges. In addition, robot motion is subject to dynamic constraints on such variables as velocity, force, and current. The execution step must track the planned path in a way that is physically feasible, and it must correct for the inevitable and sundry errors.

Input to TMP

The input to TMP includes the discrete task domain, the continuous motion domain, and the coupling of these two sides.

Task domain: The task domain defines the discrete actions the robot can take, including their preconditions and effects. For example, the `pick-up` action may have a precondition that the object is on the table and the effect that the object is in the robot's hand. The "Task Domain" section describes our implementation of task domains, and Figure 3 provides a complete example of the pick-up action with preconditions and effects.

Motion domain: The motion domain defines the three-dimensional (3-D) positions of objects in the environment, the kinematic structure of robot joints, and the geometry, i.e., meshes, of objects and robot links. Collectively, we call the robot and environment the *scene* and the tree or graph of the local coordinate frames of environmental objects and robot links the *scene graph*, which defines the configuration space of a robot. For a given configuration, computing the forward kinematics of each frame in the scene graph provides the mesh positions; then, specialized collision checkers [18] determine whether those positions are in collision.

We specify both an initial scene, consisting of the robot and the environment, and a goal scene for the planner. Then we map from these scenes to task states using the domain semantics.

Domain semantics: The domain semantics define the coupling between the discrete task domain and the continuous motion domain. Two types of functions are necessary. First, we need a function to map from a scene graph to a discrete state for the task planner. For example, if the scene graph defines some object `a`'s position relative to the robot's hand, the domain semantics would set to "true" a discrete variable `holding-a`, indicating that the robot is holding object `a`. Second, we need a function to map from a discrete task action to a motion planning problem (start and goal states) for the motion planner. For example, the `pick-up` action would start at the robot's current configuration and move to a goal that is a grasping configuration for the object to be picked up. The "TMP" section discusses our implementation of the domain semantics.

TM Planner

The TM planner finds the sequence of discrete task actions from the task domain and their corresponding motion plans—based on the domain semantics—that will take the

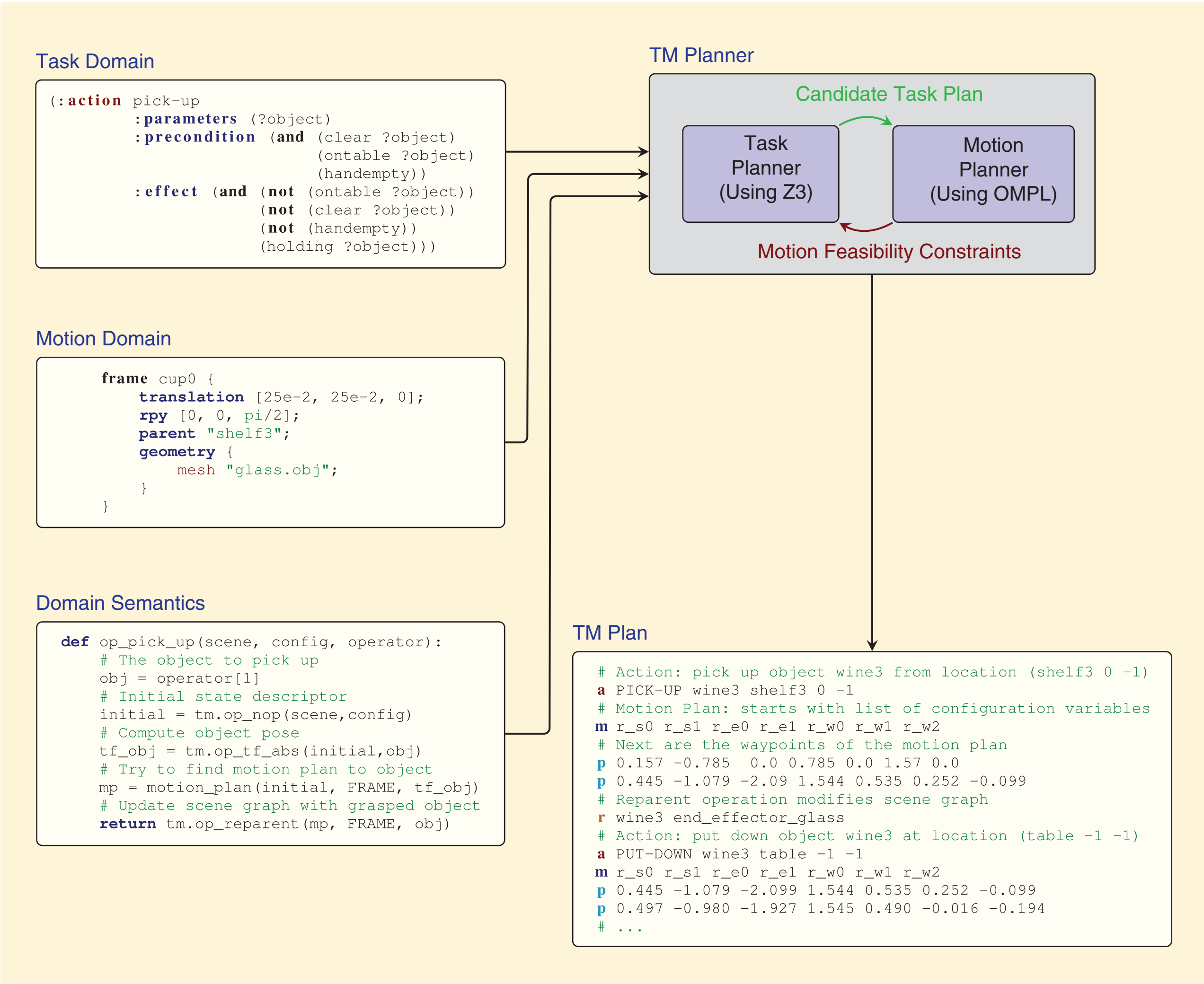


Figure 3. A TM planner implementation diagram, showing fragments of the planner's input (i.e., the task domain, domain semantics, and motion domain) and output (i.e., the TM plan). OMPL: Open Motion Planning Library.

system from some initial state or scene to a desired state or scene. This planning process is structured as an alternation between task planning to identify the discrete actions and motion planning to identify the paths for each action. Some task plans may include infeasible actions, e.g., picking up an object that is blocked by something in front of it. In this case, motion planning would fail, i.e., exceed a timeout, and we would go back to the task planner to find a different task plan, e.g., first moving the blocking object out of the way. The "TMP" section discusses our implementation of a TM planner.

TM Control

The TM control phase executes the plan in real time. Each path produced by the motion planner is a sequence of waypoints the robot must move through. To execute this motion plan, we compute a reference position, velocity, and so forth for the robot at each time step by interpolating between the waypoints. In addition, we must correct the positioning error in following the motion plan through feedback control. Finally, we operate the gripper to grasp and release objects as specified by the actions in the plan. The "Output and Execution" section discusses our control and execution implementation.

TMKit Implementation

Our TM system, TMKit, may interest researchers looking to use TMKit for the algorithm of [1] and [2] or for implementing new TMP approaches. Figure 4 outlines the major software components in our system implementation. TMP involves many different software modules, and our design choices were also influenced by the need to support real-time execution. The key to integrating these components in our system was identifying the appropriate abstractions for the task and motion domains and relating these abstractions through the domain semantics. Using these suitable abstractions not only eases development but also increases flexibility by providing a uniform interface to domain information such as task state or scene geometry.

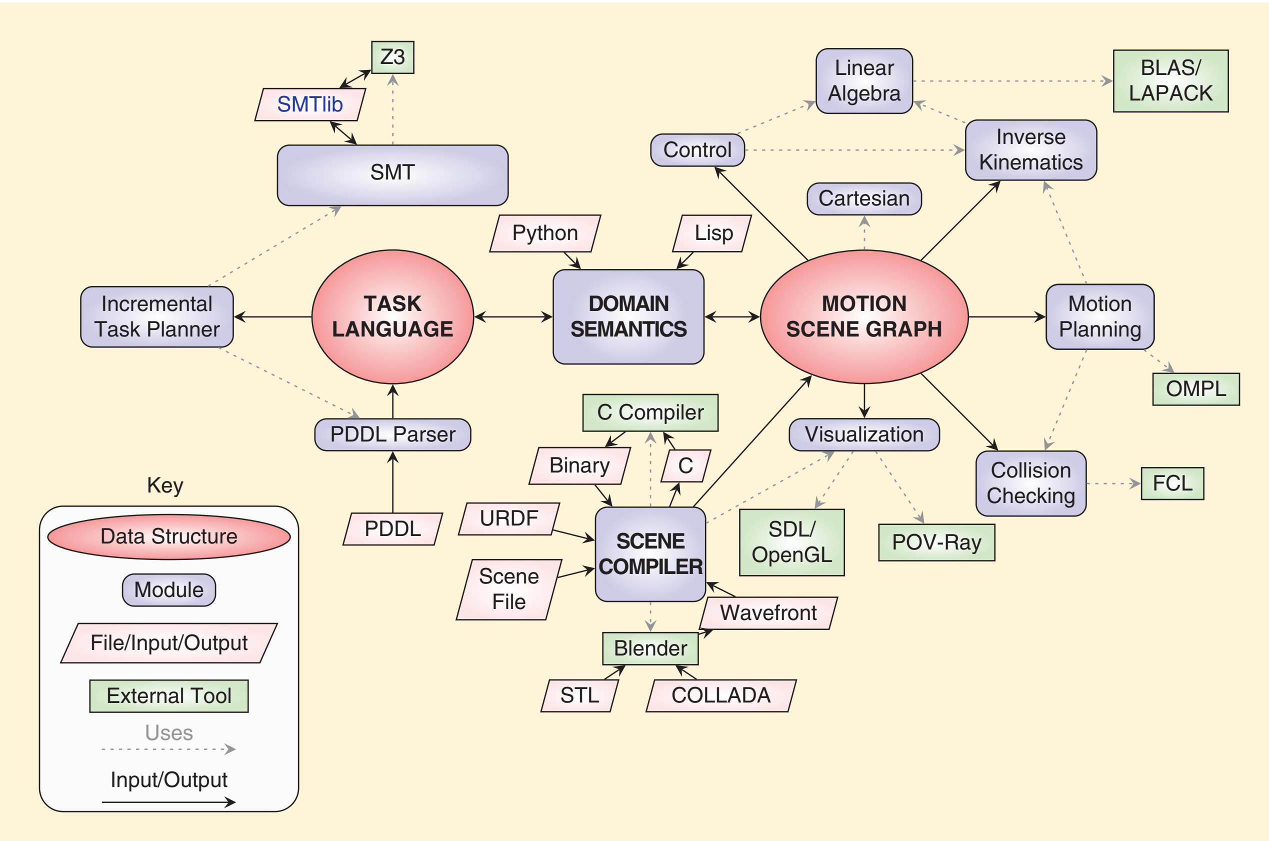


Figure 4. A map of software components. The key data structures are the task language and the motion scene graph. These data structures are connected by the domain semantics definitions. The scene compiler is also an important component. This system integrates the following external tools and formats: a basic linear algebra subprograms/linear algebra package (BLAS/LAPACK), high-performance linear algebra routines with many vendor-optimized implementations; collaborative design activity (COLLADA), an interchange file format for 3-D applications; a flexible collision library (FCL), a popular software library for collision checking; a GNU compiler collection (GCC), a compiler suite from the GNU project; the OMPL, a popular software library for sampling-based motion planning; Persistence of Vision Raytracer (POV-Ray), an open-source ray-tracing program; a planning domain definition language (PDDL), a cross-platform library to access graphics, audio, mouse, keyboard, and so forth; the Simple DirectMedia Layer (SDL), a cross-platform library to access graphics, audio, mouse, keyboard, and such; SMT, a decision-problem-combining logic and additional theories, e.g., integer constraints, lists, and arrays; stereolithography (STL), a file format for computer-assisted design software; universal robot definition format (URDF), an XML file type for robot kinematics; XML, a tree-structured, general-purpose file format; and Z3, a high-performance theorem prover/SMT solver.

Task Domain

We represent the task domain by the task language in Figure 4. Generally, task domains are specified using a variety of notations and logics, but, at a fundamental level, all these representations define some type of transition system, automaton, or formal language. The de facto standard syntax for task planning is the planning domain definition language (PDDL) [19], which our framework also takes as input. The PDDL (see Figure 3) defines parameterized actions with preconditions and effects based on first-order logic. Our task planning algorithm [1], [2], however, is not specific to PDDL and assumes only that the state space is finite and compactly represented with a set of variables. Thus, new task domains can be created in PDDL, and the underlying algorithm is adaptable to other notions.

Motion Domain

The motion domain is represented by the motion scene graph in Figure 4. Motion planning algorithms are typically defined in terms of abstract configuration spaces [9], while robot manipulators are modeled as kinematic trees or scene graphs of joints and links in packages such as OpenRAVE [22], Orocos KDL [23], and MoveIt! [24]. Existing implementations, however, focus on only a subset of the TMP pipeline shown in Figure 2. Consequently, TMKit uses a new, streamlined scene graph representation that enables direct TM translation, efficient updates, and real-time kinematics.

The scene graph is a tree representing relative Cartesian poses, with data attached at each node for geometry (e.g., meshes), inertial parameters, joint limits, and such. Figure 5 shows how the scene graph edges correspond to symbolic multiplication or chaining of transformations in the Cartesian space. Starting from the global root zero (see Figure 5) and multiplying the relative pose of each local coordinate frame along a chain yields the global pose of the frame at the end of the chain. Picking and placing objects, common operations in TMP, are represented by reparenting a frame in the scene graph, i.e., changing the object's parent between the hand and a support surface, such as a table.

Our scene graph implementation offers a unique set of features that make it suitable for both TMP and real-time execution. We use two variations on the structure: a mutable version and a persistent version, i.e., a purely functional one. Both variations can be efficiently updated at runtime, e.g., when the robot picks up a tray of objects, and they share underlying data for geometric objects via reference counting so that data for large meshes are not copied. The mutable version is based on indexed arrays and avoids heap allocations, which may impose unacceptable pauses, after construction, making it suitable for real-time operation. The persistent version is based on weight-balanced binary trees that efficiently create partial copies on updates, useful during planning when we backtrack to a previous point in the search and a previous version of the scene graph. To enable efficient multithreaded access, e.g., when performing inverse kinematics, motion planning, and visualization in separate threads, we separate the scene graph object from the data for states and configurations.

We also provide a compiler (see Figure 6) enabling scene graphs to be specified in domain-specific languages. Our

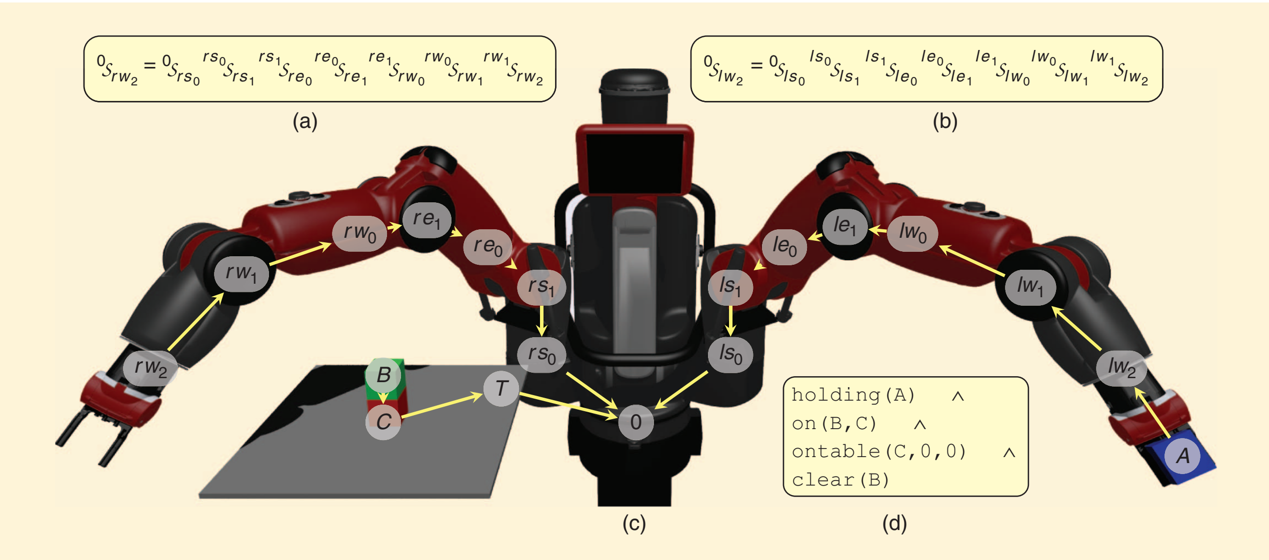


Figure 5. The scene graph abstraction for a simplified version of the Baxter robot and the corresponding task state: (a) a kinematic equation to compute the right wrist pose; (b) a kinematic equation to compute the left wrist pose; (c) the local coordinate frames of the corresponding scene graph overlaid on the Baxter; and (d) the task state corresponding to the scene graph for a pick-and-place task domain, partially shown in Figure 3. This state abstracts the object relationships in the scene graph. Note that the task state is computed automatically via the domain semantics and that additional or different task predicates may be used by modifying the domain semantics. The transform $^{a}S_{b}$ is the relative Cartesian pose between parent a and child b. The coordinate frame labels for Baxter consist of the left (l) or right (r) arm; the shoulder (s), elbow (e), or wrist (w); and the zeroth (0), first (1), and second (2) joint. The other frame labels represent the global root (zero), table (T), or blocks (A, B, *and* C).

compiler supports the widely used Robot Operating System (ROS) Universal Robot Definition Format (URDF). Additionally, because of the difficulty of writing URDF by hand, we also introduce a new, compact scene file syntax (see Figure 3). Compiling scenes offers several performance and administrative advantages:

- Compiled scenes are fast to load, because the OS directly maps into memory the included mesh data (via `mmap`), eliminating runtime parsing and processing.
- Compiled scenes reduce memory use compared to runtime parsing when multiple OS processes operate on the same scene, because the memory-mapped scene graphs in different processes share physical memory.
- Compiled scenes are easy to distribute to other machines, e.g., a cluster, which may lack scene sources, utilities, or support libraries. Only the executable or shared library is required to load the compiled scene, reducing potential runtime dependencies.
- Compiled scenes avoid the need to include large parsing libraries, e.g., an XML parser, in real-time processes and reduce the dynamic allocations necessary to load scenes.
- Multiple compiled scenes can be flexibly composed both statically ahead of time and dynamically at runtime, improving overall scene construction efficiency when some portions of the scene are fixed and others changing. For example, we can precompile the scene for the robot, which remains constant based on the robot's mechanical design. However, the locations of objects on a table may change frequently, so we can compose the precompiled robot scene with the separate or dynamically generated scene for these objects.

The scene graph data structure and scene compiler provide the necessary geometric support for TMP and plan execution.

TMP

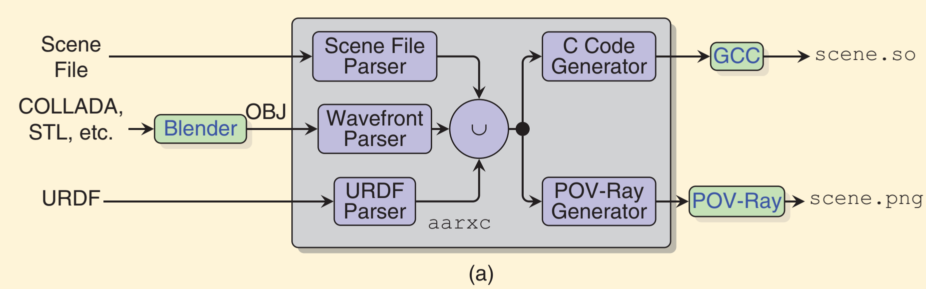
Our TMP implementation follows the overall structure of Figure 2, based on the algorithm of [1] and [2]. In the task layer, we use an incremental, constraint-based task planner. In the motion layer, we include a variety of sampling-based motion planners through the Open Motion Planning Library [9]. The key to achieving generality in our planner is the selection of abstractions. Our task languages can model arbitrary, finite-state task domains, and our scene graphs can model arbitrary, rigid-body robots and environments.

We relate the task and motion domains by defining a domain semantics (see Figure 4). The domain semantics define the conversion of the scene graph to a task state and define functions to refine high-level task actions by computing the corresponding motion plans. Concretely, the domain semantics in TMKit are functions written in Python or Common Lisp. Figure 5 includes an example of a task state computed from a scene graph, and Figure 3 contains an example of a refinement function in the domain semantics for pick-and-place manipulation. The same semantics definition may be used across multiple scenes or problem instances. Changes to the task domain, e.g., new actions, do require updating the domain semantics but require no changes to the planner itself. By abstracting TM interaction to these separate domain semantics definitions, our planning system generalizes across domains.

Output and Execution

The immediate output of our system is a TM plan describing the sequence of task actions and corresponding motion plans. We benchmark the performance and scalability of the overall approach in [1] and [2]. Figure 3 shows a fragment of such a plan for the table-setting example, represented using a plain-text, line-based file format that is human-readable and can be efficiently parsed. Each line indicates either a task action, the joints moved during a motion plan, a waypoint in a motion plan, or a reparenting operation, e.g., picking or placing an object by changing an edge in the scene graph from the table to the hand or vice versa. The resulting file defines the interleaving of task actions and motion plans.

Then, we execute the TM plan by interpolating the given motion plans and performing the indicated reparentings to grasp and release objects. There are numerous methods to



Robot	Compile	Load
Baxter	14.0 s	0.33 ms
UR10	11.7 s	0.17 ms
Jaco	15.6 s	0.20 ms

(b)

Figure 6. The scene graph compiler `aarxc`. (a) A compiler block diagram. The compiler includes parsers for scene files, Wavefront Object (OBJ) meshes, and ROS URDF files. It uses the Blender 3-D modeling program to convert a variety of meshes to the conventional Wavefront OBJ format. The compiler translates the loaded scene graph to optimized C code for later fast loading and real-time execution. It can also translate scene graphs to input for the POV-Ray raytracer for high-quality visualization. (b) Compile times—including mesh processing, code generation, and C compilation—and load times for common robots, using Blender 2.77 and GCC 4.9.2 on an Intel Core i7-4790. Example plans and planning times are presented in Figure 3 and Figure 8 [1], [2].

interpolate the waypoint sequence of a motion plan so as to satisfy the physical limits of a robot, e.g., maximum acceleration and velocity. Given any such interpolation, we use a feedback control law to compute the command for a robot:

$$\dot{q}_u(t) = \dot{q}_r(t) - k(q_a(t) - q_r(t)), \tag{1}$$

where $\dot{q}_u$ is the velocity command, $\dot{q}_r$ is the reference velocity from the interpolated waypoints, k is a feedback gain, q_a is the actual position, and q_r is the interpolated reference position.

Finally, we must communicate with the robot hardware at each time step to retrieve the actual state q_a and $\dot{q}_a$ and to send the velocity command $\dot{q}_u$. For example, we would use Controller Area Network bus message for the Schunk LWA4, transmission control protocol for the Universal Robot, or ROS communication for the Rethink Baxter.

Use Case Example

Figure 3 illustrates a use case example of TMKit for a domain such as the table setting in Figure 1, including the planner's specific input and output.

Task domain: The task domain block shows the `pick-up` action, with its preconditions and effects. This action picks up an object from a table, so the precondition requires the object to be on the table and uncovered. The effect is that the robot holds the object and the object is not on the table. The full task domain includes similar definitions for other actions, e.g., to put down objects and unstack objects.

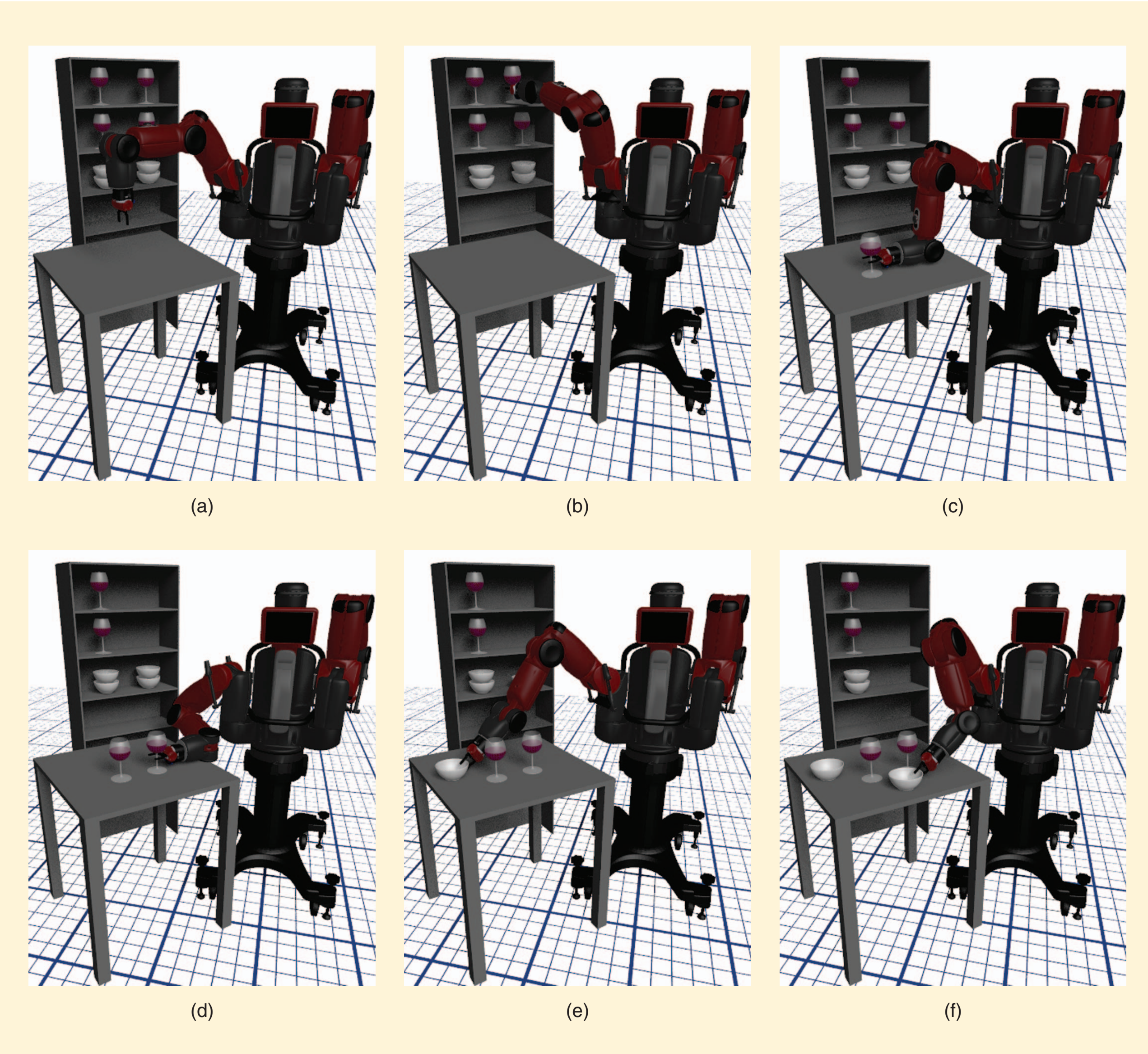


Figure 7. A TM plan to set a table using the Rethink Robotics Baxter. The average planning time for ten trials was 64.8 s on an Intel Core i7-4790. (a) The initial state, (b) picking the first glass, (c) placing the first glass, (d) placing the second glass, (e) placing the first bowl, and (f) placing the second bowl.

Motion domain: The motion domain block shows the definition for a single object (a glass). The definition includes the object's relative position to its parent (the shelf) and the object's geometric mesh. The full task domain includes similar definitions for the other glasses and bowls as well as the links and joints of the robot.

Domain semantics: The domain semantics block shows the function to find a motion plan for the `pick-up` action. This function computes the current position of the object, then attempts to find a motion plan to bring the robot's hand to a grasping pose for that object. If motion planning fails (exceeds a timeout), the motion planning function generates an exception that the TM planner will catch and handle by finding a different task plan based on the feedback from the motion planner.

TM planner: The TM planner block illustrates the alternation and feedback between task planning and motion planning. The task planner identifies a high-level plan. The motion planner attempts to find corresponding paths. Failing to do so, the motion planner provides additional constraints to the task planner, which then finds a different task plan. This process iterates until it finds a task plan where all actions have corresponding motion plans.

TM plan: The TM plan block shows the first two actions of the plan: picking and placing an object. The first action (`pick-up`) includes the joint waypoints to move the robot's hand to the grasping position for an object, then changes the object's parent in the scene graph to the robot's hand. The second action (`put-down`) includes the waypoints to move the robot's hand and the grasped object to the desired location, and then (unshown) the object's parent will change in the scene graph to the table, placing the object. The full TM plan contains the rest of the actions necessary to achieve the desired goal.

Plan execution: Figures 7 and 8 show two TM plans and planning times, one for the Rethink Robotics Baxter and one for the Universal Robots UR5. The same overall framework produces the plan for each system. We apply the framework in each case by using the URDF model of the robot for the specific system.

These examples demonstrate the modularity and extensibility of TMKit. TMKit works on multiple robots, supports multiple types of actions (e.g., picking, placing, stacking, and pushing), and handles coupling between objects [e.g., moving cans into a bin (Figure 8)]. Additional benchmark results are presented in [1] and [2].

Conclusions

We have presented TMKit, a new software framework for TMP and execution that is available under an open-source, permissive license [21]. We believe TMKit is the first open-source TMP framework that is extensible to multiple domains and different planning methods and that supports end-to-end planning and execution. Its modular design enables TMKit to generalize across hardware platforms, task domains, and TMP algorithms.

We produced a general-purpose, easy-to-use, and extensible framework for TMP. There are numerous avenues to improve and build upon this framework. Going beyond our implemented method of [1] and [2], we will extend the feedback between the task and motion layers, improve plan reuse, and incorporate additional rich constraint capabilities. We are already adapting and integrating the additional TMP methods of [3] and [4] with TMKit. We hope the community will find this end-to-end system both easy to use and a helpful platform to demonstrate other methods for TMP.

Currently, TMKit focuses on the geometric case of motion planning, which is often sufficient for manipulation. Planning with dynamics, e.g., considering torques in the planning layer, may be necessary for other cases, such as bipedal walking. However, including dynamics during motion planning may impact completeness [20], so careful analysis is necessary.

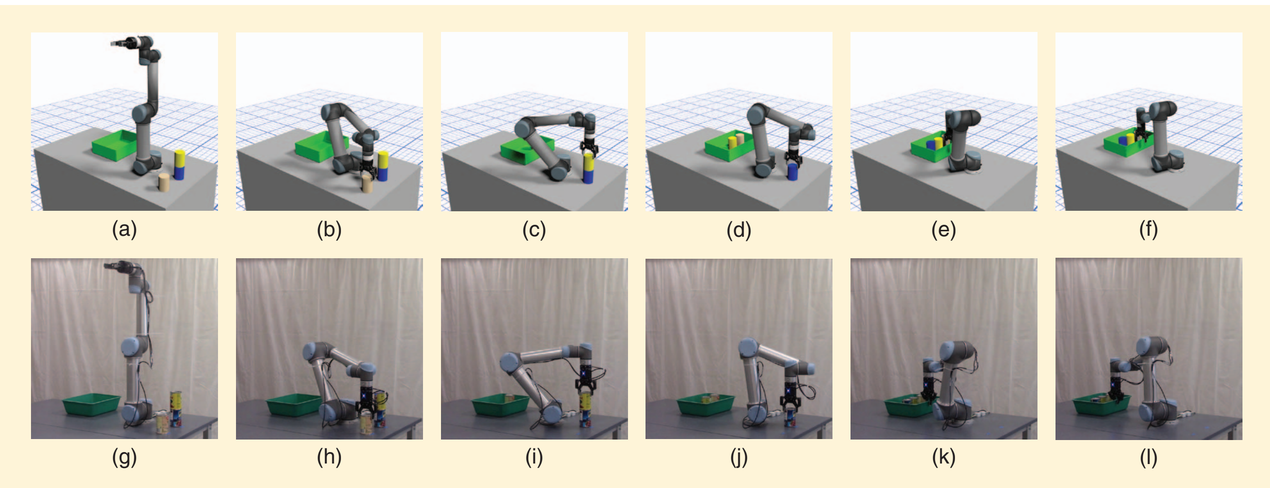


Figure 8. A TM plan to load and move a bin using the Universal Robots UR5. The same overall framework with a different URDF for the robot produces this plan for a different system. The average planning time for ten trials was 8.78 s on an Intel Core i7-4790. (a)–(f) The simulated execution and (g)–(l) the physical execution.

Improved considerations for planning with dynamics remains an area of future work for TMKit.

An ongoing need in TMP is support to compare and benchmark different TMP algorithms and implementations. We believe that TMKit, as an extensible framework supporting common formats such as PDDL and URDF, can help meet that need. Furthermore, modular components, such as the scene graph compiler (see Figure 6), could aid the development of alternative TMP methods and implementations. We hope that TMKit will be a useful tool for other researchers to evaluate existing algorithms and extend to new approaches.

Acknowledgments

We thank Andrew Wells, Cannon Lewis, and Logan Farrell for their help with the demonstration of TMKit on the UR5. This work was supported in part by grants NSF IIS-1317849 and NSF CCF-1514372, by the ERIT program of the K2I Institute at Rice University, and by the Rice University Funds.

References

[1] N. T. Dantam, Z. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach," *Robotics: Science and Systems Conf.*, 2016. doi: 10.15607/RSS.2016.XII.002.

[2] N. T. Dantam, Z. Kingston, S. Chaudhuri, and L. E. Kavraki. (2018). An incremental constraint-based framework for task and motion planning. *Int. J. Robotics Res.* [Online]. Available: https://doi.org/10.1177/0278364918761570

[3] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Towards manipulation planning with temporal logic specifications," in *Proc. IEEE Int. Conf. Robotics and Automation*, 2015, pp. 346–352.

[4] Y. Wang, N. T. Dantam, S. Chaudhuri, and L. E. Kavraki, "Task and motion policy synthesis as liveness games," in *AAAI Int. Conf. Automated Planning and Scheduling (ICAPS)*, 2016, pp. 536–540.

[5] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3, pp. 189–208, 1972.

[6] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *J. Artif. Intell. Res.*, vol. 14, no. 1, pp. 253–302, Jan. 2001.

[7] H. Kautz and B. Selman, "Unifying SAT-based and graph-based planning," in *Proc. Int. Joint Conf. Artificial Intelligence*, 1999, vol. 99, pp. 318–325.

[8] L. E. Kavraki and S. M. LaValle, "Motion planning," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Cham, Switzerland: Springer-Verlag, 2016, ch. 5, pp. 109–128.

[9] I. Şucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *Robot. Autom. Mag.*, vol. 19, no. 4, pp. 72–82, 2012.

[10] F. Lagriffoul and B. Andres, "Combining task and motion planning: A culprit detection problem," *Intl. J. Robot. Res.*, vol. 35, pp. 890–927, July 2016.

[11] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *Int. J. Robot. Res.*, vol. 32, no. 9-10, pp. 1194–1227, Aug. /Sept. 2013.

[12] M. Gharbi, R. Lallement, and R. Alami, "Combining symbolic and geometric planning to synthesize human-aware plans: Toward more efficient combined search," in *Proc. IEEE Int. Conf. Intelligent Robots and Systems*, 2015, pp. 6360–6365.

[13] 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 *Proc. IEEE Int. Conf. Robotics and Automation*, 2014, pp. 639–646.

[14] T. Lozano-Pérez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *Proc. IEEE Int. Conf. Intelligent Robots and Systems*, 2014, pp. 3684–3691.

[15] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "SMT-based synthesis of integrated task and motion plans from plan outlines," in *Proc. IEEE Int. Conf. Robotics and Automation*, 2014, pp. 655–662.

[16] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: An efficient heuristic for task and motion planning," in *Algorithmic Foundations of Robotics XI*, H. L. Akin, N. M. Amato, V. Isler, and A. F. Stappen, Eds. Cham, Switzerland: Springer-Verlag, 2015, pp. 179–195.

[17] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *Int. J. Robot. Res.*, vol. 28, no. 1, pp. 104–126, 2009.

[18] J. Pan, S. Chitta, and D. Manocha, "FCL: A general purpose library for collision and proximity queries," in *Proc. IEEE Int. Conf. Robotics and Automation*, 2012, pp. 3859–3866.

[19] M. Ghallab, D. McDermott, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL: The planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, Oct. 1998.

[20] T. Kunz and M. Stilman, "Kinodynamic RRTs with fixed time step and best-input extension are not probabilistically complete," in *Algorithmic Foundations of Robotics XI*, H. L. Akin, N. M. Amato, V. Isler, and A. F. Stappen, Eds. Cham, Switzerland: Springer-Verlag, 2015, pp. 233–244.

[21] TMKit: A task-motion planning framework. [Online]. Available: http://tmkit.kavrakilab.org

[22] OpenRAVE. (2018). [Online]. Available: http://openrave.org/

[23] Orocos Kinematics and Dynamics Library (KDL). [Online]. Available: http://www.orocos.org/kdl

[24] MoveIt! (2017). [Online]. Available: http://moveit.ros.org

Neil T. Dantam, Department of Computer Science, Rice University, Houston, United States, and Department of Computer Science, Colorado School of Mines, Golden, United States. E-mail: ndantam@mines.edu.

Swarat Chaudhuri, Department of Computer Science, Rice University, Houston, United States. E-mail: swarat@rice.edu.

Lydia E. Kavraki, Department of Computer Science, Rice University, Houston, United States. E-mail: kavraki@rice.edu.