The Task Motion Kit

Neil T. Dantam*  Swarat Chaudhuri*  Lydia E. Kavraki*

I. INTRODUCTION

Expanding the capabilities of robots to achieve complex objectives in new environments requires novel reasoning systems. Everyday tasks in the physical world, such as the table setting in Fig. 1, couple discrete decisions about objects and actions with geometric decisions about collision free motion. Robotics has traditionally treated these issues—task planning and motion planning—in isolation, thus missing their potential interactions. Instead, the joint approach of Task–Motion Planning (TMP) addresses this inherent coupling. Moreover, reasoning in concert about overall objectives and concrete motions enables the high-level specification of behavior, mitigating typically intensive system integration efforts required in robotics. We address the need for underlying models and principles in integrated robot manipulation with a new planning and execution framework that is adaptable to new robots, domains, and algorithms.

Developing a TMP framework presents challenges both in algorithmic design and software engineering. The need to consider interaction between the task and motion layers presents additional requirements not faced by stand-alone task planners and motion planners. A fundamental challenge is that we cannot prove motion plan nonexistence in the general case. Due to this inability to generally determine the motion-level infeasibility of high-level task actions, we may need to explore—and reconsider—alternate task plans of high-level actions to achieve the goal. In contrast, typical task planners generate only a single plan. In addition, as the robot grasps and rearranges objects, we must represent the changing kinematics and configuration space in which the robot can move. Typical motion planners, however, assume a fixed configuration space. Thus, existing tools for isolated task planning and motion planning do not provide the necessary capabilities to handle these interactions. In addition, applying TMP to new objects, actions and domains requires decoupling the high-level, syntactic actions or operators of a domain from the geometric semantics of underlying motion. We adapt and extend the abstractions and methods for task planning and motion planning, coupled with a recently developed TMP algorithm [1], to produce a new, integrated TMP framework.

We present the Task–Motion Kit (TMKit), an end-to-end system for probabilistically complete task–motion planning (TMP) and real-time execution. Our general task–motion framework supports multiple methods for task planning, motion planning, and task-and-motion integration. Fundamental to the design of TMKit are the shared abstractions and data structures that enable coupling of task planning, motion planning, and real-time estimation and control. TMKit is modular, extensible, and we are adapting it to additional methods for TMP [2], [3]. 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 geometric and logical specifications, demonstrating key abstractions that enable integrated robot action. To our knowledge, this is the first publicly available, general purpose TMP framework.

II. BACKGROUND

A. Task Planning

Task planning identifies a discrete sequence of actions from a given initial state to a desired goal. This field evolved largely...
from pioneering work on the Stanford Research Institute Planning System (STRIPS) [4]. The leading approaches for efficient task planning are heuristic search [5] and constraint satisfaction [6].

Off-the-shelf task planners typically focus on efficiently finding a single plan. In contrast, TMP often requires search through multiple, alternate task plans because we cannot generally prove the non-existence of corresponding motion plans. Consequently, our system does not use an off-the-shelf task planner but rather employs a newly-introduced task planner capable of efficiently generating alternate plans.

B. Motion Planning

Motion planning identifies a continuous path of valid configurations from an initial state to a desired goal condition, e.g., an end-effector position. Efficient motion planners for high-dimensional systems are typically sampling-based, and open source implementations are available [7].

Off-the-shelf motion planning frameworks often abstract the details of robot kinematics or assume the kinematics are fixed or changing slowly [7]. In contrast, TMP requires rapid updates to kinematics—and possible backtracking—to compute motion plans for the robot to grasp and transfer objects. Consequently, our system uses a streamlined kinematic representation capable of both efficient modification and backtracking within the planner and of real-time computation during execution (see subsection IV-B).

C. Task–Motion Planning

We briefly summarize recent work on TMP; a more thorough review than possible in this short paper is presented in [1].

Most prior work on TMP focuses on performance rather than completeness or generality. [8] interleaves task and motion planning at the level of individual task actions, calling the motion planner directly from the task planner for feasibility checks using semantic attachments. [9] produces a knowledge base for household robots in the logic programming paradigm. [10] applies geometric constraints to limit the motion planning space or prove motion infeasibility in special cases. Hierarchical Planning in the Now (HPN) [11] interleaves planning and execution, reducing search depth but requiring reversible actions when backtracking. [12] extends a hierarchical task planner with geometric primitives, using shared literals to control backtracking between the task and motion layers. [13] interfaces an off-the-shelf task planner and motion planning using a heuristic to remove potentially-interfering objects. [14] formulates 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 (SMT) solver to generate task and motion plans from a static roadmap, employing plan outlines to guide the planning process. FFRob [16] develops an FF-like [5] task-layer heuristic based on a lazily-expanded roadmap. 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 present 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 roadmaps. Our framework operates differently at the task level, motion level, and interface level, yielding different performance characteristics than aSyMov. For example, aSyMov’s composed roadmaps could be amortized over multiple runs but composing roadmaps for object interactions may be expensive. In contrast, we find a new motion plan each run, but efficiently update scene data structures to handle object interaction. Furthermore, our framework is extensible to both forward-search and constraint-based task planners.

Our system implements and extends the approach described in [1] which couples a new, incremental, constraint-based task planner with sampling-based motion planners. A related approach using breadth-first, forward search is the Synergistic Framework [18] and related methods [19], [2]. The system we now present generalizes the problem specification with definitions for domain semantics and incorporates support for real-time execution.

III. Requirements

Integrated Task–Motion Systems raise additional challenges compared to isolated task planning, motion planning, and plan execution. We consider the overall system structure shown in Fig. 2. The input to this system consists of a task domain describing the high-level transitions and objective, the robot and environment geometries which are combined to define the scene graph, and the domain semantics which relate the task state and scene geometry. The task–motion planner produces a plan, and the task–motion control layer executes the plan, parallel with a visualization or simulation layer.

A. Planning Requirements

TMP combines the discrete decisions of task planning with the continuous decisions of motion planning. When treated separately, task planners and motion planners employ distinct algorithms. Efficient task planners commonly use heuristic search or constraint satisfaction while high-dimensional motion planners commonly use sampling-based methods. Due to the fundamental differences between task planning and motion planning algorithms, typical methods for TMP perform separate task planning and motion planning phases, and iteratively combine the results. As shown in Fig. 2, the task planner produces candidate task plans which the motion planner attempts to refine by searching for motion plans corresponding to each task operator.

The iterative nature of TMP presents additional requirements compared to separate task planning and motion planning. Typical task planners focus on generating single-shot plans for a domain. In contrast, for TMP, we must generate alternate task plans, incrementally incorporating additional information from the motion planner. Typical motion planners either employ tree-construction for single-shot plans or roadmap construction for multiple plans for the same geometry.
Fig. 2: 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 (see subsection IV-B), and the domain semantics that relate the task and motion layers. The Task–Motion Planner generates a plan based on these inputs. The Task–Motion 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, with simulated configuration $\bar{q}$.

Fig. 3: Map of software components. The key data structures are (a) the task language and (b) the scene graph. These data structures are connected by (c) the domain semantics definitions. (d) the scene compiler is also an important component whose operation is outlined in Fig. 5.
and kinematics. In contrast, for TMP, we must generate multiple motion plans for changing geometry and kinematics as the robot grasps, transfers, and releases objects. TMP, in addition, must establish the correspondence between task information—operators and state—and motion information—kinematic structure, start, and goal conditions. Finally, we face one additional requirement to ensure some level of completeness. Sampling-based planners can offer at most probabilistic completeness. Consequently, failure to find a motion plan is not a proof that a plan does not exist. Thus, to ensure probabilistically-complete TMP, we cannot definitively rule out failed motion planning problems but must instead retry them later.

In summary, the planning requirements are:

- Replan at task level with additional constraints
- Replan at motion level over updated scenes
- Translate between task-level and motion-level information
- Retry unrefined task operators to ensure probabilistic completeness

B. Execution Requirements

Motion planners make certain assumptions to achieve reasonable performance, and it is left to the execution layer to, in real-time, correct those assumptions. Specifically, geometric motion planners assume (1) a given model for scene kinematics and geometry 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, etc., while robot motion is subject to dynamic constraints on velocity, force, current, etc. The execution layer must track the planned path in a way that is physically feasible, and it must correct for the inevitable and sundry errors.

In summary, the execution requirements are:

- Operate in real-time
- Track planned trajectories
- Correct for errors in modeling, perception, and execution

IV. IMPLEMENTATION

We now discuss our task-motion system TMKit based on the requirements presented in Sec. III. Fig. 3 outlines the major software components in our system implementation. TMP involves many different software modules, and our design choices were also influenced by the requirements for real-time execution. The key to integrate these components in our system was identifying the appropriate abstractions for 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.

A. Task Domain

We represent the task domain by the Task Language (a) in Fig. 3. 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) [20], which our framework also takes as input. PDDL (see Fig. 7a) defines parameterized actions with preconditions and effects based on first-order logic. Our task planning algorithm [1], however, is not specific to PDDL and assumes only that the state space is finite and represented as the product of variables. Thus, new task domains can be created in PDDL, and the underlying algorithm is adaptable to other notions as well.

B. Motion Domain

The motion domain is represented by the Motion Scene Graph (a) in Fig. 3. Motion planning algorithms are typically defined in terms of abstract configuration spaces [7], while robot manipulators are modeled as kinematic trees or scene graphs of joints and links in packages such as OpenRave2, Orocos KDL3, and MoveIt!4. We streamline typical scene graph representations to enable direct task-motion translation, efficient updates, and real-time kinematics.

The scene graph is a tree representing relative SE(3) poses, with attached data at each node for geometry (e.g., meshes), inertial parameters, joint limits, etc. Fig. 4 shows how the scene graph edges correspond to symbolic multiplication in SE(3). Starting from the global root node and multiplying each frame along a chain yields the global pose of the frame at the end of the chain.

Our system uses two variations on this structure: a mutable version suitable for real-time operation and a persistent, i.e., purely functional, version suitable for backtracking during planning. Both variations share underlying data for geometric objects via reference counting so that data for large meshes is not copied.

Our scene graph implementation provides a unique set of features that make it ideally suited to task-motion planning and execution:

1) Our scene graphs can be modified at runtime, such as when the robot picks up a tray of objects. Moreover, in the persistent version, such modifications efficiently create a new partial copy that shares structure with the original; both new and original scene graphs remain valid, simplifying the implementation of backtracking during planning.

2) We separate the scene graph object from the representation for states and configurations, allowing multi-threaded concurrent access without locks, e.g., when performing inverse kinematics, motion planning, and visualization all in separate threads.

3) Our scene graphs are suitable for real-time use because they perform no heap allocations—which may impose unacceptable latency—after construction.

---

2http://openrave.org/
3http://www.orocos.org/kdl
4http://moveit.ros.org
Fig. 4: The scene graph abstraction for a simplified version of the bi-manual Baxter robot: (a) symbolic multiplication to compute the right wrist pose. (b) symbolic multiplication to compute the left wrist pose. (c) the frames of the corresponding scene graph overlaid on the Baxter. The transform $^aS_b$ is the relative $SE(3)$ pose between parent $a$ and child $b$. The 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 ($0$), table ($T$), or blocks ($A$, $B$, $C$).

4) Our scene graphs can be bound in high-level languages such as Python and Lisp and can be extended with new features while preserving binary and source compatibility because we expose a C interface for the scene graphs.

We also provide a compiler enabling scene graphs to be specified in domain specific languages: the widely-used ROS Universal Robot Definition Format and our own simplified based file format, which is both 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 task state.

C. Task–Motion Planning

Our TMP implementation follows the overall structure of Fig. 2, based on the algorithm of [1]. 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 (OMPL) [7].

The key to achieving generality in our planner is careful selection of abstractions. Our task languages can model arbitrary finite state task domains, and our motion scene graphs can model arbitrary rigid body robots and environments.

We relate the task and motion domains by defining a Domain Semantics. The domain semantics defines the conversion of the scene graph to a task state and defines functions to refine task operators to motion plans. Concretely, the domain semantics in TMKit are functions written in Python or Common Lisp. Fig. 7b contains an example of a refinement function that maps the abstracted domain semantics for pick-and-place manipulation to a concrete motion plan. The same semantics definition may be used across multiple tasks.

D. Output

The immediate output of our system is a Task–Motion plan describing the sequence of task actions and corresponding motion plans. Fig. 8 shows a fragment of such a plan for the table-setting example, represented using a plain-text, line-based file format, which is both 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.
Fig. 5: The scene graph compiler aarxc. (a) Compiler block diagram. The compiler includes parsers for scene files, Wavefront OBJ meshes, and ROS URDF files. It uses the Blender 3D 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.

```
1 def x 1;
2 def y 2;
3 def s .1;
4
5 frame block {
6   translation [x, y, 0];
7   rpy [0, 0, pi/2];
8   geometry {
9     dimension [s, s, s];
10    shape box;
11 }
12 }
```

(a)

```
1 <?xml version="1.0" ?>
2 <robot name="example">
3   <link name="base">
4     <link name="block_link">
5       <visual>
6         <origin rpy="0 0 0" xyz="0 0 0"/>
7         <geometry>
8           <box size="0.1 0.1 0.1"/>
9       </geometry>
10     </visual>
11     <collision>
12       <origin rpy="0 0 0" xyz="0 0 0"/>
13       <geometry>
14         <box size="0.1 0.1 0.1"/>
15       </geometry>
16     </collision>
17   </link>
18 </link>
19 <joint name="block_joint" type="fixed">
20   <origin rpy="0 0 0.5708" xyz="1 0 0"/>
21   <parent link="base"/>
22   <child link="block_link"/>
23 </joint>
24 </robot>
```

(b)

Fig. 6: Comparison of our scene file syntax and ROS URDF. (a) our scene files use a conventional, curly-brace syntax and include the ability to define constants and perform arithmetic. (b) ROS URDF is an XML format and is often used with the ROS XACRO XML preprocessor.

```
!(:action pick-up
  :parameters (?object)
  :precondition (and (clear ?object) (ontable ?object) (handempty))
  :effect (and (not (ontable ?object)) (not (clear ?object)) (not (handempty)) (holding ?object)))
```

(a)

```
def op_pick_up(scene, config, operator):
    # The object to pick up
    obj = operator[1]
    # Initial state descriptor
    initial = tm.op_nop(scene, config)
    # Try to find motion plan to object location
    mp = motion_plan(initial, FRAME, tm.op_tf_abs(initial, obj))
    # Update scene graph with grasped object
    return tm.op_reparent(mp, FRAME, obj)
```

(b)

Fig. 7: An example operator: PICK-UP(?object). (a) The operator’s definition in the Planning Domain Definition Language (PDDL). (b) The operator’s semantics. This Python function attempts to compute a motion plan to the object’s current location. If successful, it modifies the scene graph to indicate a grasped object.

Finally, we execute the task–motion plan by interpolating the given motion plans and performing the indicated reparentings to grasp and release objects. Fig. 9 shows the simulated execution sequence for the table-setting example.

V. CONCLUSION

We have presented a new software framework for Task–Motion Planning (TMP), the Task–Motion Kit (TMKit) system. TMKit is available under an open source, permissive
TMKit will be a useful tool for other researchers working on alternate TMP methods and implementations. We hope that common formats such as PDDL and URDF, can help meet that We believe that TMKit, as an extensible framework supporting benchmark different TMP algorithms and implementations. Our focus in this work was to produce a generally-applicable, easily-usable, and extensible framework for TMP. There are numerous avenues to improve and build upon this Our 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 Fig. 5) could aid in the development of alternate TMP methods and implementations. We hope that TMKit will be a useful tool for other researchers working on TMP.

**REFERENCES**


Fig. 9: Example of a task–motion plan to set a table. (a) the initial state. (b) picking the first glass. (c) placing the first glass. (d) placing the second glass. (e) placing the first bowl. (f) placing the second bowl.