Manipulation Task Planning with Constrained Kinematic Controller

Marcos S. Pereira ∗ Bruno V. Adorno **

∗ Programa de Pós-Graduação em Engenharia Elétrica, - Universidade Federal de Minas Gerais, MG, (e-mail: marcos.si.pereira@gmail.com)
** Departamento de Engenharia Elétrica, Universidade Federal de Minas Gerais, MG, (e-mail: adorno@ufmg.br)

Abstract: This paper addresses the integration of task planning and motion control in robotic manipulation, where automatically generated feasible manipulation sequences are executed by a controller that explicitly accounts for the task geometric constraints. To cope with the high dimensionality of the manipulation problem and the complexity of specifying the tasks, we use a multi-layered framework for task and motion planning adapted from the literature. The adapted framework consists of a high-level planner, which generates task plans for linear temporal logic specifications, and a low-level motion controller, based on constrained optimization, that allows to define regions of interest instead of exact locations while being reactive to changes in the workspace. Thus, there is no low-level motion planning time added to the total planning time. Moreover, since there is no replanning phase due to motion planner failures, the robot actions are generated only once for each task because the search for a plan occurs on a static graph. We evaluated this approach with two pick-and-place tasks with similar complexity to the original framework and showed that the number of plan nodes generated is smaller than the one in the original framework, which implies less total planning time.

Keywords: Linear temporal logic; task planning; constrained control.

1. INTRODUCTION

As robots have become physically capable of executing highly complex manipulation tasks, there is a need to define plans in a longer time horizon and with a considerable number of manipulable objects in cluttered environments. Those plans determine the task execution order and how to execute them. Furthermore, there is also a need to increase robot autonomy and enable the robot to automatically generate those plans, which motivates the integration of task and motion planning (ITMP). In this sense, this work focuses on solving the ITMP problem for manipulation tasks in a computational efficient manner by combining state of the art techniques including task planning and motion controllers instead of motion planning.

Motion planning and task planning are studied in robotics from different point of views. Task planners generate plans by processing a high number of action states, that is, states that describe what the robot must do regardless of where objects are located in the environment (Kaelbling and Lozano-Perez, 2011). On the other hand, a motion planner treats the geometry of the problem, that is, it must know the location of each object to plan how to manipu-
late them (Kaebbling and Lozano-Perez, 2011). Neither of the approaches alone is enough to do manipulation task planning. Thus, there is a need to do ITMP.

The goal of ITMP is to use a task planner to define a high-level manipulation task sequence, that is used to generate a trajectory in the task space for a low-level motion planner that plans the trajectories in the configuration space. This low-level trajectory is then executed by a motion controller. This way, continuous motion planning is combined with discrete task reasoning (He et al., 2015). Discrete high-level task plans are appropriately described by using an abstract discrete model of the robot and formal specifications and, then, task planning can be done through model-checking techniques (Bhatia et al., 2011).

Task planning appears in many guises and usually are dealt with by using Artificial intelligence (AI). Fikes and Nilsson (1971) propose a problem solver called STRIPS (Stanford Research Institute Problem Solver) that searches for a model in a space of world models to reach a desired goal. However, STRIPS does not regard temporal specifications, only uses linear sequences of operators, and requires lower layers to translate its very high-level plans to robot actions (Fikes and Nilsson, 1971). In this sense, Lana et al. (2015) propose the Manipulation Task Model (MTM) that is dedicated to the planning of robotic manipulation tasks in a bottom-up manner.

The MTM describes actions and their sequence to take an object from an initial state to a desired state. The task primitive actions are defined as elements of dual quaternion algebra while the sequence is described by a small subset of process algebra. One of the advantages of the MTM is that it considers the task’s physical parameters and, thus, facilitates the use of motion controllers to execute the task. However, MTM is specific to manipulation tasks, which is a drawback when describing tasks such as navigation. Furthermore, the MTM also does not allow temporal specifications.

An interesting alternative for STRIPS and the MTM is linear temporal logic (LTL), which allows to specify Boolean and temporal constraints. Also, LTL has correctness and completeness guarantees (Baier and Katoen, 2008, ch. 5), but the number of states of the specified task has combinatorial growth, which is also known as the state-explosion problem (Wongpiromsarn et al., 2010). Therefore, frameworks based on LTL usually build a discrete abstraction of the robotic system (Kress-Gazit et al., 2007; He et al., 2015; Kloetzer and Belta, 2008; Bhatia et al., 2011), but such construction is usually non-trivial.

LTL has been used mainly in mobile robotics (Kress-Gazit et al., 2007; Kloetzer and Belta, 2008; Bhatia et al., 2011). Nonetheless, the abstraction techniques for mobile robotics depend on the discrete representation of the state-space, which make them intractable to manipulation planning due to the large dimensionality of the manipulation problem. To solve this problem, He et al. (2015) propose a framework that allows the planning of high-level manipulation tasks specified in LTL and also handles the ITMP problem.

To solve the ITMP problem, Kambhampati et al. (1991) investigated the use of a hybrid planning model that contributes with expressiveness and reasoning power to traditional hierarchical planners through the use of a set of specialists, which adapt the plan to satisfy unexpected additional constraints. Hence, the planner and the specialists must know the constraints imposed by their decisions to avoid inconsistent tasks. In order to add more flexibility to ITMP, some works started using multi-layered frameworks that use a task planner to generate a motion sequence for the motion planner, which checks the viability of executing the task (Erdem et al., 2011; Kaebbling and Lozano-Perez, 2011; Srivastava et al., 2014; Lozano-Perez and Kaebbling, 2014; He et al., 2015).

In addition to using LTL, He et al. (2015) also propose a multi-layered framework to solve the ITMP problem, which uses a coordinating layer between high- and low-level to decide about the difficulty of executing the task. However, it also uses a motion planner, which requires motion planning time, in addition to the high-level planning time. Therefore, we adapt the framework of He et al. to solve the ITMP challenge by using a new approach to replace the motion planning layer.

In order to replace the motion planning layer, our work proposes the use of a constrained motion controller (Marinho et al., 2019), which allows the definition of regions of interest instead of discrete poses for the end-effector and the system to be reactive. The idea is to specify geometrical constraints, based on those regions of interest, to accomplish a desired task generated by the high-level task-planner without resorting to low-level motion planning. Constrained motion controllers depend on the specification of a set of constraints that must be respected by the system. They can be described by minimization problems (Laumond et al., 2015) that make use of extra available degrees of freedom of the robot to achieve the task while respecting the constraints. This enables the generation of collision-free motions by using mathematical programming. Nonetheless, in the general case there are no analytical solutions and numeric solvers must be used (Goncalves et al., 2016; Escande et al., 2014).

The main contribution of this work is to adapt the framework of He et al. (2015), which allows the specification of manipulation tasks in LTL, to use a constrained motion controller (Marinho et al., 2019), which allows imposing joint limits and constraints to prevent collisions with the workspace. This strategy greatly reduces the computational burden of the overall method, with the additional advantages that there is no need of motion planning and the system is reactive. Moreover, the merging of both frameworks combines the expressivity and temporal reasoning of LTL at the task level with the closed-loop stability guarantees of low level constrained motion controllers.

2. PRELIMINARIES

2.1 Linear Temporal Logic

In LTL, a proposition is a statement that can be true or false, but not both, and atomic propositions are the ones that do not depend on the truth or falsity of any other proposition. Let $A = \{a_0, a_1, ..., a_n\}$ be a set of atomic propositions. LTL semantics is defined over infinite traces, that is, words over the alphabet $2^A$, where $2^A$ is the
power set of \( A \). Given letters \( A_i \in 2^A \), with \( i = 0, 1, 2, \ldots \), words are infinite sequences such as \( \sigma = A_0 A_1 A_2 \cdots \). For example, consider \( A = \{q_0, a_1, a_2\} \), hence, a possible word would be \( \sigma = A_0 A_1 A_0 A_1 A_2 A_1 A_0 \).\footnote{In LTL literature, “eventually” is commonly represented as \( \mathcal{F} \). Because in our works \( \mathcal{F} \) denotes coordinate systems, we use \( \mathcal{E} \) for “eventually”.} Given order.

The formulas for \( a \) that, eventually \( k \) an unknown time \( \varphi \) trarily more complex. For instance, simple formulas for the sake of clarity, they can be arbi-
containing the operators in (1) with an arbitrary number \( a \) for the sake of example, that \( a \) will eventually be true. More specifically, the operation \( \mathcal{E}(a_0 \land \mathcal{X}a_1) \) means that eventually \( a_0 \) will be true and, afterwards, \( a_1 \) will eventually be true.

Specifically, the operation \( \mathcal{E}(a_0 \land \mathcal{X}a_1) \) implies that \( a_0 \) and \( \mathcal{X}a_1 \) will surely be true in an unknown time in the future. Let us assume, for the sake of example, that \( a_0 \) becomes true in time \( j \). The operation \( \mathcal{E}a_1 \) implies that \( a_1 \) will surely be true in an unknown time \( k \) in the future and \( \mathcal{X}a_1 \) implies that \( k \geq j + 1 \). Therefore, eventually \( a_0 \) will be true, and, after that, eventually \( a_1 \) will be true. The formula \( \varphi_2 \) could be used, for instance, to specify a pick-and-place task that requires two objects to be eventually manipulated in a given order.

In our current work, manipulation tasks must be achieved over a finite time horizon. Hence, we use only co-safe LTL formulas, which are the ones that can be interpreted by considering finite words (Kupferman and Y. Vardi, 2001). Syntactically, co-safe LTL formulas contain only the temporal operators \( \mathcal{X}, \mathcal{E}, \mathcal{U} \), and the negation operator is only allowed over atomic propositions, but not over temporal formulas.

See the works of Kupferman and Y. Vardi (2001); Baier and Katoen (2008) for more formal definitions of the syntax and semantics of LTL. Examples of LTL specifications for manipulation tasks are given in Section 3.2.

2.2 Constrained Kinematic Controller

The constrained controller (Marinho et al., 2019) is based on an optimization problem that minimizes the joint velocities, \( \dot{\mathbf{q}} \in \mathbb{R}^n \), in the \( \ell_2 \)-norm sense while respecting hard constraints, such as obstacles in the workspace, joints limits, etc. The controller is derived from the desired closed-loop task-error dynamics. The differential kinematics, given by \( \ddot{\mathbf{x}} = \mathbf{J}_d \mathbf{q} \), provides the relation between the joint velocity vector and the task-space velocity vector in which \( \ddot{\mathbf{x}} = \ddot{\mathbf{q}}(t) \in \mathbb{R}^n \) is the robot configuration vector, \( \mathbf{x} \equiv \mathbf{x}(\mathbf{q}) \in \mathbb{R}^m \) is the task vector, and \( \mathbf{J} \equiv \mathbf{J}(\mathbf{q}) \in \mathbb{R}^{m \times n} \) is the task Jacobian. Given a desired task vector \( \mathbf{x}_d \in \mathbb{R}^n \), we define the task error \( \ddot{\mathbf{x}} \equiv \mathbf{x} - \mathbf{x}_d \). Considering \( \dot{\mathbf{x}}_d = \mathbf{0} \) for all \( t \), the error dynamics is given by \( \ddot{\mathbf{x}} \) to zero with an exponential convergence rate, the closed-loop dynamics is given by \( \ddot{\mathbf{x}} + \eta \ddot{\mathbf{x}} = \mathbf{0} \) where \( \eta \in (0, \infty) \) is the gain. Thus, the desired closed-loop dynamics is given by \( \mathbf{J}_d \mathbf{q} + \eta \ddot{\mathbf{x}} = \mathbf{0} \) and the control input \( \mathbf{u} \) that minimizes the joint velocities \( \ddot{\mathbf{q}} \in \mathbb{R}^n \) is obtained as

\[
\mathbf{u} = \arg\min_{\mathbf{q}} ||\mathbf{J}_d \mathbf{q} + \eta \ddot{\mathbf{x}}||^2 + \lambda ||\dot{\mathbf{q}}||^2_2 \\
\text{subject to } \mathbf{W} \ddot{\mathbf{q}} \leq \mathbf{w}
\]

where \( \lambda \in [0, \infty) \) is a damping factor and \( \mathbf{W} \in \mathbb{R}^{l \times n} \) and \( \mathbf{w} \in \mathbb{R}^l \) are used to impose linear constraints in the control inputs. In addition to describing poses, the task vector \( \mathbf{x}_d \) can also be used to describe geometric primitives (e.g., points, lines, planes) in the workspace. In this sense, the end effector will converge to the specified desired task vector.

Constrained controllers allow the definition of regions of interest, eliminating the need of low-level motion planners, thus saving computational time and resulting in a reactive system. In order to prevent collisions with the workspace, we use the Vector Field Inequalities (VFI) framework (Marinho et al., 2019), which requires distance functions between two collidable entities and the corresponding Jacobian matrices, as better described in Section 3.3.

3. METHODOLOGY

The framework for integration of task planning and motion control that we adapt from the literature is composed of a high-level planning framework and a low-level constrained motion controller. We use the high-level planner proposed by He et al. (2015) and the constrained motion controller proposed by Marinho et al. (2019). Figure 1 shows both steps.
Consider a robot in the workspace that can pick-and-place objects between locations. Given a finite set of objects \( O \), a finite set of locations \( \mathcal{L} \), and a finite set of actions \( \mathcal{M} \), find a sequence of actions \( a_0, a_1, \ldots, a_m \in \mathcal{M} \) that manipulate the objects \( o_0, o_1, \ldots, o_n \in O \) between locations \( \text{loc}_0, \text{loc}_1, \ldots, \text{loc}_k \in \mathcal{L} \) to satisfy a linear temporal logic (LTL) specification \( \varphi \). The sequence of actions is obtained by the planning framework that will be presented in Section 3.2 and the actions are executed by the constrained motion controller presented in Section 2.2 with the constraints that will be presented in Section 3.3.

### 3.2 Planning Framework

The planning framework of He et al. (2015) can be divided into the four steps depicted in Figure 1, which are described next.

**Linear Temporal Logic Task Specification** In the first step, a manipulation task \( \varphi \) is specified using co-safe LTL and it depends only on the objects and the locations. For instance, in a pick-and-place task we can specify where each object must be at the end without mentioning anything about the robot. Therefore, the propositions of the LTL formulas are defined as \((o_i, l_i)\), which means that "object \( o_i \) is in location with the label \( l_i \)" (He et al., 2015).

A given scene with a finite set of objects \( O \), a finite set of labels \( \Gamma \), and a finite set of locations \( \mathcal{L} \), with a function \( \mathcal{L} \) that maps locations to labels, the atomic propositions of this scene are elements of \( O \times \Gamma \). For instance, first, suppose a robot has to place an object \( o_1 \) at the location with label \( l_1 \) at \( \Gamma \). The specification for this task can be given by \( \varphi_1 = \mathcal{E}(o_1, l_1) \), which means that "eventually object 1 is in location 1". For the sake of clarity, let us define \( p_1 = (o_1, l_1) \in O \times \Gamma \). We specify a second task "Keep object 1 at location 1 until object 2 and object 3 are sequentially placed at their locations", which is defined as \( \varphi_2 = p_1 \cup (p_2 \land X p_3) \). More specifically, \( p_1 \) will be true until \( p_2 \land X p_3 \) becomes true. If \( p_2 \) becomes true in time \( k \), then \( X p_3 \) will be true in time \( k + 1 \). Therefore \( p_1 \) will be true, at least, until time \( k \). If we had multiple objects to be sorted, we would define \( E p_1 \land E p_2 \land E p_3 \land \cdots \land E p_n \) that reads "eventually place object 1 in location 1 and so on".

**Manipulation Abstraction** The next step is to define an abstraction graph \( \mathcal{R} = (V, E, L) \) that captures all the ways the robot can manipulate the objects, in which the set of nodes \( V \) consists of the Cartesian product between the set \( \mathcal{M} \) of motion primitives, the set \( \mathcal{L} \) of locations, the set \( O \) of objects, and a set \( \mathcal{O}_L \) of objects locations. The set \( E \) contains the edges of \( \mathcal{R} \) and \( L \) is a labeling function that is defined later in this section when we introduce the product graph. The set of motion primitives contains the robot actions needed for pick-and-place tasks and is defined as \( \mathcal{M} \triangleq \{ \text{GRASP}, \text{PLACE}, \text{HOLD}, \text{MOVE} \} \). The motion graph \( \mathcal{M} \triangleq (\mathcal{M}, E, \mathcal{L}) \), with \( E, L \subseteq \{(e_i, e_j) \in \mathcal{M} \times \mathcal{M} \} \), which is shown in Figure 2a, defines the allowed sequence of motions.

The location set \( L \triangleq \{ \text{loc}_1, \ldots, \text{loc}_k, \text{loc}_{\text{inter}} \} \) represents the locations of interest in the robot workspace where objects can be placed. Also, when an object is in the gripper, it is considered to be in the intermediate location \( \text{loc}_{\text{inter}} \). Furthermore, the location graph \( \mathcal{L} = (\mathcal{L}, E, \varphi) \) with \( E, \varphi \subseteq \{ (e_i, e_j) \in \mathcal{L} \times \mathcal{L} \} \), which is shown in Figure 2b, represents how objects can be transferred between locations.

The construction rules of the set of edges \( E \subseteq \{(e_i, e_j) \in V \times V \} \) that connect the nodes \( V \) of \( \mathcal{R} \) can be found in the work of He et al. (2015). Generally speaking, at least each edge \( (v, v') \in E \) in the abstraction must satisfy that \((\alpha, \alpha') \in E, \varphi \). For instance, assume two nodes \( v = (\text{HOLD}, \text{loc}, o, \alpha_2) \) and \( v' = (\text{PLACE}, \text{loc}', \alpha_2) \). There will be an edge transition between them if \( \text{loc} = \text{loc}' \neq \text{loc}_{\text{inter}} \), meaning that the robot will be holding the object at one of the \( k \) locations in \( v \) and placing it at that location in \( v' \). On the other hand, if the action was \( \text{MOVE} \) in \( v' \), there would be no edge between \( v \) and \( v' \) because there is no edge connecting \( \text{HOLD} \) to \( \text{MOVE} \) in \( \mathcal{M} \).

Invalid nodes may be generated by the Cartesian product (He et al., 2015). For instance, it is not possible that object 1 be at the gripper while at location 1. Therefore, in our work, the set of reachable nodes is incrementally constructed by starting from an initial node \( v_0 \in V \) and following the construction rules of \( E \) that only point to valid nodes.

**Deterministic Finite Automaton** After constructing the abstraction graph \( \mathcal{R} \), the formula \( \varphi \) is converted into a deterministic finite automaton (DFA) \( \mathcal{A}_\varphi \) (Duret-Lutz and Poirtreau, 2004) that specifies all the ways the robot can execute the task while fulfilling the formula \( \varphi \) (He et al., 2015).

The DFA is defined as \( \mathcal{A}_\varphi = (Z, z_0, \Sigma, \delta, F) \), where \( Z \) is the finite set of states, \( z_0 \) is the initial state, \( \Sigma \) is the...
set of events that causes transitions in the automaton,\( \delta : Z \times \Sigma \rightarrow Z \) is the transition function, and \( F \) is the set of final states (also known as accepting or marked states). The set of events \( \Sigma \) is also the alphabet of the LTL formula, that is, \( \Sigma = 2^Z \). The automaton begins in the initial state \( z_0 \) and when an event represented by the letters from \( \Sigma \) occurs, there is a transition to the next state following the transitions in \( \delta \). This happens until a state in \( F \) is reached. The transitions between each state are represented by letters and the path that leads to a state in \( F \) represents the sequence of truth assignments of the propositions that satisfy the specification (He et al., 2015). In other words, a sequence of letters represents a word accepted by the automaton.

For example, consider the automaton shown in Figure 3.\(^2\) with \( \mathcal{A} = \{p_0, p_1, p_2\} \), where \( p_i = (o, l_i) \) and \( i \in \{0, 1, 2\} \). Therefore, \( \Sigma = \{ \emptyset, \{p_0\}, \{p_1\}, \{p_2\}, \{p_0, p_1\}, \{p_1, p_2\}, \{p_0, p_2\}, \{p_0, p_1, p_2\} \} \). Given \( A_i = \{p_i\} \), an example of accepted word is \( \sigma = A_0 A_1 A_2 \). The letter \( A_0 = \{p_0\} \) means that object \( o \) will be placed at location with label \( l_0 \), which causes the transition from state 3 to state 2 because the letter \( \{p_0\} \) is equivalent to the logical condition \( p_0 \land \neg p_1 \land \neg p_2 \). Similarly, letters \( A_1 \) and \( A_2 \) cause the transitions to state 1 and 0, respectively. Again, the letter \( \{p_1\} \) is equivalent to the logical condition \( \neg p_0 \land \neg p_1 \land \neg p_2 \) and \( \{p_2\} \) is equivalent to \( \neg p_0 \land \neg p_1 \land \neg p_2 \). Therefore, the object \( o \) will be in locations with labels \( l_0, l_1, \) and \( l_2 \) in this sequence, but not simultaneously.

Nonetheless, there may be words or paths that do not respect the physical world. For instance, one transition may require that an object be at multiple locations at the same time, that is, \((o, l_j) \land (o, l_k)\). This is exemplified in Figure 3 by the one-letter word \( \sigma = \{p_0, p_1, p_2\} \), which causes the transition from state 3 to the final state 0. This transition means that the object will be simultaneously in locations with labels \( l_0, l_1, \) and \( l_2 \).

**Product Graph** Last, the nodes in the set \( V \) of the abstraction graph \( \mathcal{R} \) are combined with the states in the set \( Z \) of the automaton \( \mathcal{A}_\varphi \) into a product graph \( \mathcal{P} \equiv (V_\mathcal{R}, E_\mathcal{P}) \), where \( V_\mathcal{R} = V \times Z \) and \( E_\mathcal{P} \subseteq \{(e_i, e_j) \in V_\mathcal{R} \times V_\mathcal{R} \} \). This combination represents how the robot can move the objects to achieve the specified task (He et al., 2015). There is an edge \((p, p') \in E_\mathcal{P}\) from \( p = (v, z) \) to \( p' = (v', z') \), if \((v, z) \) and \((v', z') \) are combined with the states in \( Z \) in this sequence, but not simultaneously.

\(^2\) Generated at https://spot.lrde.epita.fr/app/ and then adapted.

![Figure 2. Abstraction graphs.](image1)

![Figure 3. Automaton \( A_\varphi \) obtained from LTL specification \( \varphi = E p_0 \land E p_1 \land E p_2 \). State 3 is the initial state and 0 is the final state. In this diagram the negation operator is given by \( \neg \) and the Boolean operator “and” is given by \( \land \). Recall that \( p_i = (o, l_i) \) means that object \( o \) is in location with label \( l_i \). States marked red, green, blue, and magenta represents the object in locations with labels \( l_0, l_1, \) and \( l_2 \) and \( l_{\text{inter}} \), respectively. The cyan arrows represents an accepted path for an accepted word \( \sigma = A_0 A_1 A_2 \). States marked with multiple colors mean that the object can be at one of the locations indicated. For instance, in state 1 the object may be in location with label \( l_0 \) or \( l_1 \) depending on the path taken.

**edge** \((v, v') \in E\) and \( \delta(z, \Pi(L(v'))) = z' \), with \( L(v) \) being the labeling function

\[ L(v) \equiv \{(p, \varPsi(v, p)) \in \mathcal{A} \times \{T, F\} : \varPsi(v, p) = T\}, \]

where

\[ \varPsi(v, p) = \begin{cases} T, & \text{if } \mathcal{L} \circ \mathcal{S}(o_C, o_p) = l_p, \\ F, & \text{otherwise}. \end{cases} \]

The function \( \varPsi(v, p) \) determines if the object label \( l_p \) in \( p = (o_p, l_p) \) matches the label of the corresponding object in the tuple \( o_C \) of \( v = (o, \text{loc}, o_C) \). Furthermore, \( \mathcal{L} : \mathcal{L} \rightarrow \Gamma \) provides the corresponding label of a given location and \( \mathcal{S} : O^{|\mathcal{L}|} \times O \) returns the object location in \( o_C \) corresponding to \( o_C \). Lastly, \( \Pi(L) \equiv \{p : (p, T) \in E\} \) generates the label that causes the transition in the automaton.

For example, consider \( O = \{o_1, o_2\} \), \( \Gamma = \{l_1, l_2\} \), \( \mathcal{L} = \{\text{loc}_1, \text{loc}_2, \text{loc}_3\} \), \( \mathcal{A} = \{p_1, p_2, p_3\} \) with \( p_1 = (o_1, l_1) \), \( p_2 = (o_2, l_1) \), and \( p_3 = (o_2, l_2) \); also, \( \mathcal{L}(\text{loc}_1) = l_1 \), \( \mathcal{L}(\text{loc}_2) = l_1 \), and \( \mathcal{L}(\text{loc}_3) = l_2 \); lastly \( v = (\text{MOVE}, \text{loc}_{\text{inter}}, o, \varphi_C) \), with \( \varphi_C = (\text{loc}_1, \text{loc}_2) \). In this case,

\[ L(v) = \{(p_1, T), (p_2, T), (p_3, F)\} \]

because the label of \( p_1 = (o_1, l_1) \) and the location label of \( o_1 \) (i.e., \( \mathcal{L}(\text{loc}_1) \) ) are \( l_1 \); hence, \( (p_1, T) \). Similarly, the label of \( p_2 = (o_2, l_1) \) and the location label of \( o_2 \) (i.e., \( \mathcal{L}(\text{loc}_2) \) ) are \( l_1 \); therefore, \( (p_2, T) \). The label of \( p_3 = (o_2, l_2) \) is different from the location label of \( o_2 \); thus, \( (p_3, F) \). Furthermore, the corresponding letter is \( \Pi(L_{A_\varphi}(v)) = \{p_1, p_2\} \).
As a result from the building process of $\mathcal{P}$, a path from the start node $p_0 = (v_0, z_0)$ to an accepting node $p_f = (v_f, z_f)$—which consists of a node in which $z_f$ is an accepting state—on $\mathcal{P}$ induces a path on $\mathcal{R}$ and a run on $A_z$. Furthermore, the path on $\mathcal{R}$ considers that each object will be at exactly one location, at a given instant, and that objects can be moved only by the manipulator (He et al., 2015). Finally, each node $(v, z) \in \mathcal{P}$ contains the information $v$, which indicates what action the robot must do, where the end-effector should be located, the object that should be on the gripper, and where the objects should be placed in the world, and the state $z$ that indicates the current state of the automaton. Moreover, if an accepting state is reached, then the task plan executes the task satisfying the specification. Therefore, a path is searched from the start node to the accepting node. For instance, assume that $(v, v') \in E$ with $v = \{\text{HOLD, loc}_1, o_i, \sigma_\mathcal{L}\}$ and $v' = \{\text{HOLD, loc}_\text{inter}, o_i, \sigma'_\mathcal{L}\}$. This means that object $i$ is on the gripper, which will move from loc$_1$ to loc$_\text{inter}$ and $\sigma_\mathcal{L} = (\ldots, \text{loc}_\text{inter}, \ldots)$ and $\sigma'_\mathcal{L} = (\ldots, \text{loc}_\text{inter}, \ldots)$, since the object begins in the gripper and stays in the gripper.

Since there may be more than one solution in $\mathcal{P}$, the Dijkstra’s algorithm is used to search for the shortest accepting path on the product graph $\mathcal{P}$, that is, the shortest path from an initial node to an accepting one. In our work, differently from the work of He et al. (2015), the high-level plan is executed by the constrained motion controller (Marinho et al., 2019) defined by (2). More specifically, the controller is used to execute the actions of type HOLD and MOVE. The actions GRASP and PLACE are executed by closing and opening the end-effector gripper, respectively.

### 3.3 Constrained workspace manipulation

We use three planes in the environment to prevent end-effector collisions with two walls and the table in the workspace and four lateral planes to constrain the end-effector inside a region of interest—the relaxed task region—shown in Figure 4. Similarly, to the work of Quiroz-Omana and Adorno (2019), the seven constraints can be shown as

$$
-J_{\text{semi},j} \dot{q} \leq \eta d_{p,\text{object},j},
$$

$$
-J_{p,n_\mathcal{L}} \dot{q} \leq \eta d_{p,R},
$$

where $i \in \{1, 2, \ldots, 7\}$ and $d_{p,\text{object},j} = d_{p,n_\mathcal{L}} - d_{p,\text{safe},i}$, with $d_{p,\text{safe},i}$ being the safe distance to each plane and $d_{p,R}$ and $J_{p,n_\mathcal{L}}$ are the point-static-plane distance and its Jacobian, respectively (Marinho et al., 2019). The error vector in the control law (2) is given by $\tilde{\mathbf{e}} = d_{\text{eff},n_\mathcal{L}}$, where $d_{\text{eff},n_\mathcal{L}}$ is the (signed) distance from the end-effector to the horizontal plane passing through the regions of interest shown in Figure 5. In addition to the plane constraints, we use joint velocities constraints to prevent saturation of actuators (Quiroz-Omaia and Adorno, 2018). Moreover, to prevent end-effector collisions with non-manipulated objects while a given object is being manipulated, we add semi-infinite cylindrical constraints to each non-manipulated object (see Figure 4). Therefore, each object is constrained by a cylinder cut by a plane. Given $k$ objects, to each one of them is associated a cylindrical and a plane constraint, which yields the following inequalities:

$$
-J_{\text{semi},j} \dot{q} \leq \eta d_{p,\text{object},j},
$$

$$
-J_{p,n_\mathcal{L}} \dot{q} \leq \eta d_{p,R},
$$

where $j \in \{1, \ldots, k\}$, and $d_{p,\text{object},j} = d_{p,l_j} - R_j$, with $R_j$ being the radius of the cylinder around the $j$-th object; $d_{p,l_j}$ and $J_{\text{semi},j}$ are in $\mathbb{R}^{1 \times n}$, with $n$ being the number of robot joints, are the point-static-line distance and its Jacobian, respectively (Marinho et al., 2019), and

$$
J_{\text{semi},j} = \begin{cases} J_{p,l_j} & \text{if } d_{p,n_\mathcal{L}} < 0, \\ 0^{1 \times n} & \text{otherwise.} \end{cases}
$$

### 4. SIMULATION & DISCUSSION

The implementation of the planning framework proposed by He et al. (2015) was done in C++ with the Boost Graph Library and the automata utilities from the Open Motion Planning Library (OMPL). The LTL task is processed using Spot (Duret-Lutz and Poitrenaud, 2004) and we performed simulations on CoppeliaSim using ROS. Furthermore, we used the DQ Robotics library (Adorno and Marinho, 2020) for robot modeling and control and to define the geometrical constraints, and constrained convex optimization was implemented using IBM ILOG CPLEX Optimization Studio.

To test the planning framework, we created a simulation scene on CoppeliaSim where a Kinova JACO robot must execute assistive tasks for a seated person, with limited or no lower limbs mobility, who cannot reach farther objects in the scene. The robot is placed on a table and the

---

3 https://www.boost.org/doc/libs/1_71_0/libs/graph/doc/index.html
4 http://ompl.kavrakilab.org/
5 http://www.coppeliarobots.com/
6 https://www.ros.org/
To express task 1 in co-safe LTL, we define $p_{m,h} = (o_{\text{meat}}, l_{\text{heat}})$, $p_{s,p} = (o_{\text{salad}}, l_{\text{pers}})$ and $p_{m,p} = (o_{\text{meat}}, l_{\text{pers}})$. Hence,

$$\varphi_1 = \mathcal{E}(p_{m,h} \land p_{s,p} \land \mathcal{X} \mathcal{E}(p_{m,p})).$$

(7)

The automaton generated from $\varphi_1$ has three states and the planner explores 16975 nodes in the product graph. The total planning time average is 1.65 seconds. Since task 1 is a sequential task, the planner generates a task plan that follows the specified order of manipulation. The formula for this task does not specify what should be done with the book in front of the person. Hence, the planner generates a task plan that heats the meat, then serves the salad and then serves the meat. However, after the salad is served, both locations in front of the person are occupied by the book and the salad. Thus, the planner decides to remove the salad instead of the book. Should both the meat and the salad be served together, it would be necessary to explicitly specify that.

Task 2: “First, move book to the book region. Next, serve the salad and heat the meat in any order. Afterward, serve the meat while the salad is being eaten. Keep the book on the book position during the whole task execution.”

For task 2, we define $p_{o,b} = (o_{\text{book}}, l_{\text{book}})$, $p_{s,p} = (o_{\text{salad}}, l_{\text{pers}})$, $p_{m,h} = (o_{\text{meat}}, l_{\text{heat}})$, and $p_{m,p} = (o_{\text{meat}}, l_{\text{pers}})$. As a result, $\varphi_2$ is given by

$$\varphi_2 = \mathcal{E}(p_{o,b} \land \neg p_{s,p} \land \neg p_{m,h} \land \mathcal{X} \mathcal{E}(p_{o,b} \land \mathcal{E}(p_{s,p}) \land \mathcal{E}(p_{m,h}))$$

$$\mathcal{X} \mathcal{E}(p_{o,b} \land p_{s,p} \land \mathcal{X} \mathcal{E}(p_{s,p} \land p_{m,p})))$$

(8)

After moving the book to the book region, task 2 gives freedom to the robot to decide if the salad will be served first or, instead, if the meat should be heated first. Next, it will serve the meat while the salad is being eaten. In comparison to task 1, task 2 specifies that the salad and the meat must be placed in front of the person. Otherwise, the robot would be free to remove the salad and only then serve the meat. The automaton generated from $\varphi_2$ has eight states and the planner explores 25098 nodes in the product graph. The total planning time average is 2.41 seconds.

Table 1 shows the planning data for the two tasks, which have the same number of objects (four) and locations (eight). The number of states and edges in the DFA indicates the complexity of the task. The planning time $T_{\text{planning}}$ with the associated standard deviation (s.d.) corresponds to an average of 50 runs. As expected, an increase in the task complexity increases the planning time because more nodes are explored in the product graph and then Dijkstra runs on a larger graph. In addition, Table 1 also shows the results for two tasks with similar complexities in the original framework of He et al. (2015).

5. CONCLUSION

This work presented an improvement of the manipulation framework proposed by He et al. (2015), which uses sampling-based motion planners as the low-level planner, to use instead a constrained motion controller that allows the definition of regions of interest. Our contribution is that the low-level layer from the original framework,
Table 1. Planning data for $\varphi_1$ and $\varphi_2$.

| Task | $|A_\varphi|$ | $|E_\varphi|$ | $|V_\varphi|$ | $T_{\text{planning}}$(s) |
|------|--------------|--------------|--------------|----------------|
| $\varphi_1$ | 3 | 5 | 16975 | 1.65418 (0.152028) |
| $\varphi_2$ | 8 | 20 | 25908 | 2.40551 (0.203752) |

| Original framework | $|A_\varphi|$ | $|E_\varphi|$ | $|V_\varphi|$ | $T_{\text{planning}}$(s) |
|-------------------|--------------|--------------|--------------|----------------|
| $\varphi_3$ | 2 | 3 | 44100 | 2.76 |
| $\varphi_4$ | 8 | 27 | 75511 | 4.48 |

$|A_\varphi|$ and $|E_\varphi|$ are the number of states and edges in the DFA, respectively. $|V_\varphi|$ is the total number of nodes in the product graph, $T_{\text{planning}}$ is the total high-level planning time accounting for the generation of graphs and the Dijkstra search for a path in the product graph $\mathcal{P}$.

Acknowledgements

This work was supported by the Brazilian funding agencies CAPES and CNPq.

References


