To appear in: Proceedings of AIAA SciTech Forum, January 2023.

A General Framework for Remote Command and Control of Robots in Space

Mark Moll*
PickNik Robotics, 4730 Walnut St, Suite 106, Boulder, Colorado 80301

Future space missions are expected to be increasingly uncrewed and robots are expected to perform station keeping duties. The state of the art in robotics makes full autonomous operation at rigorous space safety standards extremely challenging. At the same time, the communication delay between robots in space and operators on earth makes traditional teleoperation painfully slow. We have developed a hardware-agnostic framework for remotely operating robots that bridges the gap between full autonomy and teleoperation. The framework provides parametrized, reusable building blocks (called *behaviors*) for performing robot operations such as opening doors, pressing buttons, and picking objects. When using these behaviors, an operator provides a few inputs and approves an automatically computed plan before it is executed. These behaviors can be combined into much more complex objectives that can encode fallback behaviors if a particular behavior fails or conditional behaviors that depend on user or sensor inputs.

I. Introduction

In future space missions robots are expected to take on caretaker duties and perform payload operations autonomously. We have developed a software framework that fills the gap in existing robotics software by providing an easy-to-use interface for trained operators who may not have a background in robotics or software engineering. By offering various levels of autonomy, the operator remains firmly in control, but is not required to provide low-level inputs for (sub)tasks that are routine or low risk. By providing control modes with increased autonomy it also becomes practical to operate robots over long communication delays, which is critical for future missions. Over time, through our software, we expect the degree of autonomy leveraged in practice to increase.

The framework presented here is targeting manipulation tasks over long time horizons. Outside of carefully controlled factory settings, it is currently difficult to have robotic manipulators perform such tasks with minimal supervision. The complexity lies in part in finding feasible motions for robots with many degrees of freedom, subject to complex motion constraints (including intermittent contact). In addition, human oversight is often still required for reliable perception, e.g., to help correctly identify possible targets and obstacles.

Our work is inspired in part by TaskForce, the supervisory control framework for humanoid robots developed by NASA JSC [1] (and its precursor Robot Task Commander [2]). Similar to TaskForce, we rely on so-called *affordance templates* [3–5] to capture the motion constraints for manipulating classes of objects. As an example, affordance templates can be used to concisely capture how to grasp a door handle and move it in such a way that a door is rotating around its hinges. There are several differences between TaskForce and our work. First, there is more of an emphasis on integration with a motion planning pipeline and a more formal approach to modeling high-level behaviors. Second, we enforce a more clear separation between end user and developer: an end user cannot execute arbitrary code, but can combine behaviors in arbitrary ways. Our framework supports third-party development of additional behaviors that can be easily added to the user interface (UI) via plugins.

The use cases for our software framework are across all areas of space robotics: Intravehicular Activities (IVA), Extravehicular Activities (EVA), In-orbit Servicing, Assembly, and Manufacturing (ISAM), and manipulation tasks in future lunar and planetary surface missions. For IVA, sample tasks that are enabled by our technology include cargo transfer, routine maintenance tasks, and tending to science experiments. These are tasks with a well-defined logical structure, but that cannot be solved by simply hard-coding some trajectories. This combination of clear high-level structure and on-the-fly motion planning to avoid collisions makes it a good fit for our technology. For EVA, a robot could prepare repairs for an astronaut to minimize human exposure to radiation or perform inspections/repairs semi-autonomously via remote supervision. For ISAM, we are working on visual servoing capabilities and capture/manipulation of moving targets. This is useful in the context of servicing satellites, for instance. Finally, for lunar and planetary rovers we

*Director of Research

envision the same framework being used to coordinate navigation and manipulation tasks (e.g., to navigate to a rock of interest and collect some samples).

The rest of the paper is organized as follows. Section II describes the overall design objectives for our system. In section III we cover the control and planning capabilities. With planning, we can differentiate between robot motion planning, basic task planning, and high-level behavior. Section IV describes the end-user interface that enables remote operation of space robots. Section V contains concluding remarks as well as a summary of our ongoing and future work to extend the capabilities of our software framework.

II. System Design Objectives

The main objectives in designing our software framework for supervised autonomy are (1) to accelerate the development of novel robotic systems and (2) make operating such systems more accessible to non-robotics experts. There are thus two clear target audiences we have in mind who would benefit from this technology. The first audience consists of robotics and software engineers who would benefit from hardened software components that provide reusable, parameterized building blocks that can be used to build new robotic solutions in a hardware-agnostic way. For this audience, the goal is to shorten development time, reduce the need for robotics experts, and support building and validating advanced robotics applications. The second audience consists of operators that need a user-friendly mechanism to task robots remotely at various levels of autonomy. Targeted use cases range from full autonomy over long time horizons to more manual human-in-the-loop control while providing strong safety guarantees in all cases. The same reusable building blocks that are accessible to developers are also available through a UI for end users and can be combined by a user to script new robot behaviors on the fly.

Our software framework is hardware agnostic, but is currently primarily aimed at (mobile) manipulators. We have the same software running on five types of manipulators with different numbers of degrees of freedom and sensors: Universal Robots UR5, Kinova Gen3, Kinova Gen3 Lite, HDT Adroit, and Sarcos Sapien. Typically, each manipulator is equipped with a Robotiq 2-finger gripper and an Intel RealSense camera at the wrist. An additional Intel RealSense camera is mounted overhead to provide better situational awareness.

Our work builds on top of an open source core centered on MoveIt [6], a motion planning and manipulation software package that has been used on over 150 robots. MoveIt is itself part of the robot software ecosystem formed by the Robot Operating System 2 (ROS 2) [7]. This facilitates the integration of our software with a broad range of hardware, including other sensors and manipulators. Several currently available space-hardened manipulators currently already use MoveIt and could be integrated with our software. ROS 2 is designed to be used in production environments. For space applications, a hardened, flight-ready version of ROS 2 called Space ROS is actively developed by Open Robotics. MoveIt already runs in the development version of Space ROS. Our long-term plan is to support both a ROS 2-based version of our software and a flight-certified Space ROS-based version.

III. Planning and Control Software

Below, we will give an overview of the planning and control infrastructure, in a bottom-up fashion, starting with low-level control, moving on to robot motion planning, task planning, and concluding with high-level behavior specification. The architecture is designed to be extensible, allowing users to define and use their own controllers, motion planning algorithms, and task planning algorithms via a plugin-based system.

A. Control

The control interface between hardware and our software is based on ROS 2 control [8]. This abstracts away many hardware differences between different manipulators and makes it possible for higher-level software layers (such as MoveIt) to be hardware agnostic. The same software is currently being used on five different hardware configurations mentioned in section II.

Controllers can be made task-dependent, with parameters that can be overridden on the fly as needed. For example, we have recently added an admittance controller where the admittance parameters can easily be tuned separately for each axis of translation and rotation in a given coordinate frame. This is useful to control compliance and applied force/torque in contact-rich tasks like opening doors/drawers and pushing buttons.

We have also created low-level control interfaces for controlling robots through different physics simulators (like Open Robotics' Gazebo simulator and NVIDIA's Isaac Sim™). This is useful for two different reasons. First, it enables robot software developers to prototype new behaviors in simulation until the code can safely be deployed on hardware.

It is key that the interface abstracts away between simulation and hardware, so that any differences between simulation and hardware experiments can be attributed mostly to modeling of hardware and physical realism of the simulator (rather than differences in high-level code). The second use of simulation is to enable verification and validation of robot behavior in many hypothetical scenarios that might be difficult to replicate on hardware.

While the focus in our software is on autonomy, we also support remote control of the end effector. For many manipulators teleoperation can be a frustrating experience, because it is often unclear to the operator whether the manipulator is close to joint limits or singularities. Furthermore, once the robot is "stuck" at joint limits or in a singularity, it is often unclear how to get unstuck and move towards the desired pose. To mitigate this problem, we have added tolerances to user-specified control inputs and use an inverse kinematics solver that can find joint commands (positions, velocities, accelerations) in real-time that keep the manipulator as far away from the limits and singularities as the tolerances allow. In our testing, this has made a big difference in usability for remote operators. However, using motion planning to move between pre-defined or user-defined waypoints is, in our experience, a more efficient way to perform large maneuvers that avoids singularities, joint limits and collisions automatically.

B. Motion planning

The underlying motion planning capabilities are provided by MoveIt [6]. MoveIt has a modular planning architecture and several different classes of motion planning algorithms are available for use. These algorithms make different trade-offs in terms of planning speed, completeness, and optimality. They also differ in the way that they can accommodate task-specific constraints or preferences. There is not one class of algorithms that is strictly better across all use cases, which is why MoveIt allows users to select, tune, and combine different algorithms based on the task.

The default planning backend uses the Open Motion Planning Library (OMPL), a library of many sampling-based algorithms [9]. These algorithms sample random configurations in configuration space and connect them to other nearby configurations, typically through simple interpolated motions [10]. This results in a graph that concisely represents which parts of the configuration space are reachable. These algorithms are typically probabilistically complete (i.e., the probability of *not* finding a solution goes to 0 with each iteration, often at an exponential rate). Under certain conditions, they can also provide asymptotic (near-)optimality guarantees [11, 12]. Recent work has shown how these algorithms can be used in combination of arbitrary sets of hard constraints [13]. The constraints implicitly define a (often lower-dimensional) subspace of valid configurations, and with—the appropriate abstractions—sampling-based planners can operate on such implicit spaces in the same way as in "normal" configuration spaces. In practice this means that we can add, e.g., one or more position and orientation constraints to any part of the robot, and have any sampling-based planner produce continuous paths that respect these constraints. On top of this, it is also possible to independently choose a cost function such that optimizing planners produce paths that both respect the hard constraints and optimize the cost with respect to the specified cost function.

Other possible planning backends are available as well such as CHOMP [14] and STOMP [15], two trajectory optimization algorithms, and the Pilz planner, which computes traditional industrial motion plans using inverse kinematics-based methods.

C. Integrated task and motion planning

Integrated task and motion planning is concerned with combining planning over discrete state changes in objects with continuous motion planning [16]. In principle one could take a purely hierarchical approach by computing a high-level discrete plan and heuristically translate each discrete step into a continuous motion plan. This can lead to unexpected failures if the heuristic choices lead to infeasible motion planning problems (e.g., an inverse kinematics solution for placing an object on a table is not reachable). A tighter integration between task and motion planning can avoid such failures and is an active area of research [16].

While integrated task and motion planning can be done over long time horizons based on basic problem domain definitions and geometric constraints, computation times can in such cases be prohibitively expensive for interactive use by an operator. However, even the most basic manipulation tasks are not simply one call to a motion planning algorithm. Take, for example, the task of picking up an object. First, an end effector grasp pose needs to be selected. Given a grasp pose, one or more inverse kinematics (IK) solutions for the joint angles can be computed. For a given IK solution, it is customary to compute a Cartesian approach motion from some fixed distance to the object to be grasped. This is one (constrained) motion planning problem. Separately, we can plan an (unconstrained) motion from the robot's current configuration to the start of the approach motion. There are several dependencies between these stages. Arbitrary choices in some stages can make it impossible to find solutions in other stages. To get around this problem, we use a

task and motion planning framework called MoveIt Task Constructor [17], which provides a mechanism to create a mechanism to sketch out the task structure and constraints. It will then compute an end-to-end feasible solution path. By providing a sketch of the task structure, the typical combinatorial explosion associated with general task planning is mitigated and complete task and motion plans can often be computed in a matter of seconds.

D. High-level behavior

The definition of new behaviors within our framework is kept as general as possible. Motion planning, trajectory execution, collecting and processing sensor data, and prompting an operator for input are all modeled as individual behaviors. To help organize behaviors into more complex objectives, we rely on Behavior Trees [18], a concise, human-interpretable representation of a robot policy. Behavior trees originated in the game industry, but have also found significant traction in the robotics community. Behavior trees are modular and composable: basic behaviors can be combined into larger trees with well-defined semantics and such trees can be reused as subtrees within even larger trees. The particular implementation of Behavior Trees that we use, BehaviorTree.cpp, allows for parameters to be passed between behaviors in the tree, enabling relevant context to be passed on as well as allowing users to set values for parameters in parametric behaviors. Behavior trees are not limited to simple linear sequences of actions, but can also model conditional behavior dependent on operator or sensor input and use fallback actions (in case execution of an action fails).

While complex motions can be planned using MoveIt Task (MTC), this is not a requirement. For example, we have also defined a visual servoing behavior, which requires a tight loop between perception and control which is terminated by some event (e.g., based on image features, but possibly also based on force/torque sensor data). This type of servoing is outside the scope of MTC, which assumes that the only changes in the environment are caused by robot actions.

To compose behavior trees (i.e., objectives) there are several building blocks that can be divided into four different categories:

Primitive behaviors This includes planning various types of trajectories, executing those trajectories, prompting the user for plan approval, updating the model of the environment by incorporating a depth image from an RGBD camera, visual servoing, and so on. Different types of paths can be planned: free space paths to named waypoints, Cartesian paths, and constrained manipulation paths specified using affordance templates. The affordance templates encode paths as a sequence of so-called screw motions [19], which are characterized by a screw axis and pitch as well as force/torque parameters that characterize the physical interaction and the object associated with the affordance template [4]. For example, opening a hinged door with a door handle can be described by two screw motions: one for turning the door handle a given amount and another for moving the door along an arc.

Control flow blocks This category includes: a block to run a sequence of behaviors, a block to run several behaviors in parallel, if-then-else blocks, looping blocks, and so on.

Decorators Decorators are convenience functions that wrap around a behavior. They can be used to add a timeout or delay to a behavior or invert the success/failure exit status of a behavior, for example.

Named behavior trees Predefined behavior trees can be used as subtrees to compose larger, more complex behaviors. Including a named behavior tree is the equivalent of a function call in traditional programming.

These building blocks can be combined programmatically or through the operator interface (see next section) to form new objectives. In essence, the operator interface provides a visual programming language. The goal is to avoid most of the complexity of full-fledged programming languages (which would create a barrier to entry), but still provide enough expressiveness to capture common robot behavior patterns in an easy-to-use way. *Parallel programming*, for example, is often seen as an advanced programming skill, but is exposed as a basic primitive in MoveIt Studio. In many robotics applications, parallel programming is often needed to interleave planning and execution. One would like to be able to plan several pick and place operations, while executing the first pick. Similarly, when a mobile manipulator is navigating to a particular pose, it could compute manipulation plans. Similarly, some *reactive programming* patterns are also supported. For instance, a target can be tracked through visual servoing until it disappears from the field of view or an arm can push a button until a force threshold is reached.

Let us consider now a few examples of how behavior trees can be used to model some robot behaviors. First, consider the behavior tree for obtaining a 3D snapshot from a RGBD camera. This is a very basic objective that consists of a sequence of two primitive behaviors (Fig. 1a): obtain a depth image and update the model of the world (i.e., the planning scene). Note that each behavior can have input and output parameters. This is useful to pass context information from one behavior to another. Some of the input parameters can also be set explicitly by the operator. For instance, we can specify which camera to use.

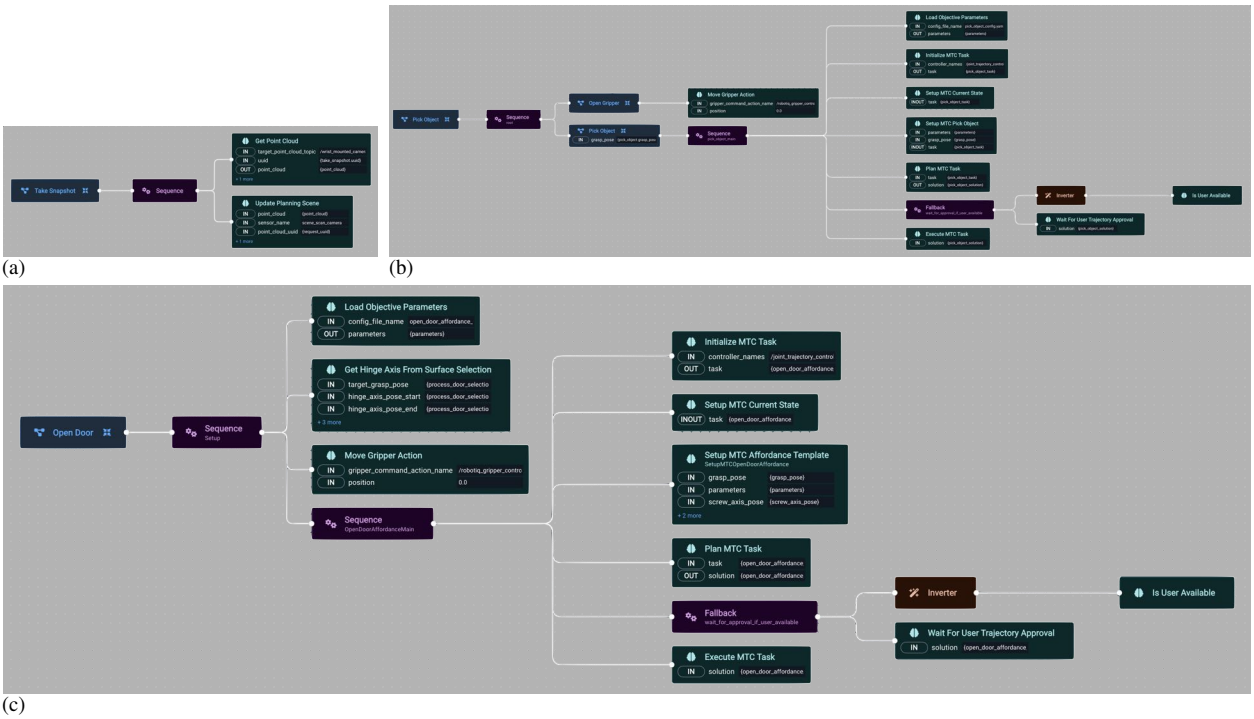

Fig. 1 Example behavior trees for some of the built-in objectives: (a) take a 3D snapshot of the environment, (b) pick an object, (c) open a door.

Next, consider the behavior tree for the "pick object" objective (Fig. 1b). When using the web interface, an operator can click in a camera view to pick a desired object. The 2D image coordinates are transformed to a 3D location and are associated with a segmented-out 3D shape. Programmatically, the same operation can also be performed. In the current implementation, the object selection is an implicit part of the objective and does not show up as a separate behavior. To pick the object, we first ensure the gripper is open. Next, we define, plan, and execute the grasp motion. MTC is used to create a pick object task. This includes several steps: a free-space motion, a Cartesian approach, grasp planning, etc. This task is parametric and does not consist of hard-coded motions. To instantiate this task, we need to prepend the current robot state to the task ("Setup MTC Current State"). With this information, the task is fully defined and a complete trajectory can be planned. If a user is available, a preview of the computed plan is shown to the user, who can either approve the plan or cancel. If approved, the plan is executed (and shown through the user interface, if used).

Finally, consider the behavior tree for opening a door (Fig. 1c). This objective is similar in structure to the "pick object" objective. The user inputs in this case consists of three points in a camera image: the first one marks the location of the rotational axis of a door handle, while the last two mark a line segment corresponding to the door's hinge side. The door's surface is automatically extracted from the depth image data. This, combined with the user inputs, is sufficient to compute the door handle's rotational axis in 3D, orthogonal to the surface, and the hinge location in 3D, embedded in the door plane.

IV. Operator Interface

The user interface (UI) for our software is web-based and requires no software other than a web browser to be installed on the operator's computer. The UI is organized around different modes of interaction with the environment and a remote robot. See Fig. 2 for an overview. The tabs at the top of the window are organized (left to right) from more autonomous modes of operation to more manual modes of operation. The first mode is called "Objective Builder" (Fig. 2a). In this mode one can browse and inspect a list of predefined objectives. For the selected objective the corresponding behavior tree is visualized. This behavior tree can be edited through the UI. New objectives can be defined from scratch or by copying and modifying existing ones. The next mode of operation is to have the robot perform specific objectives (Fig. 2b). A number of "favorite" objectives are listed at the top. By selecting "Show all

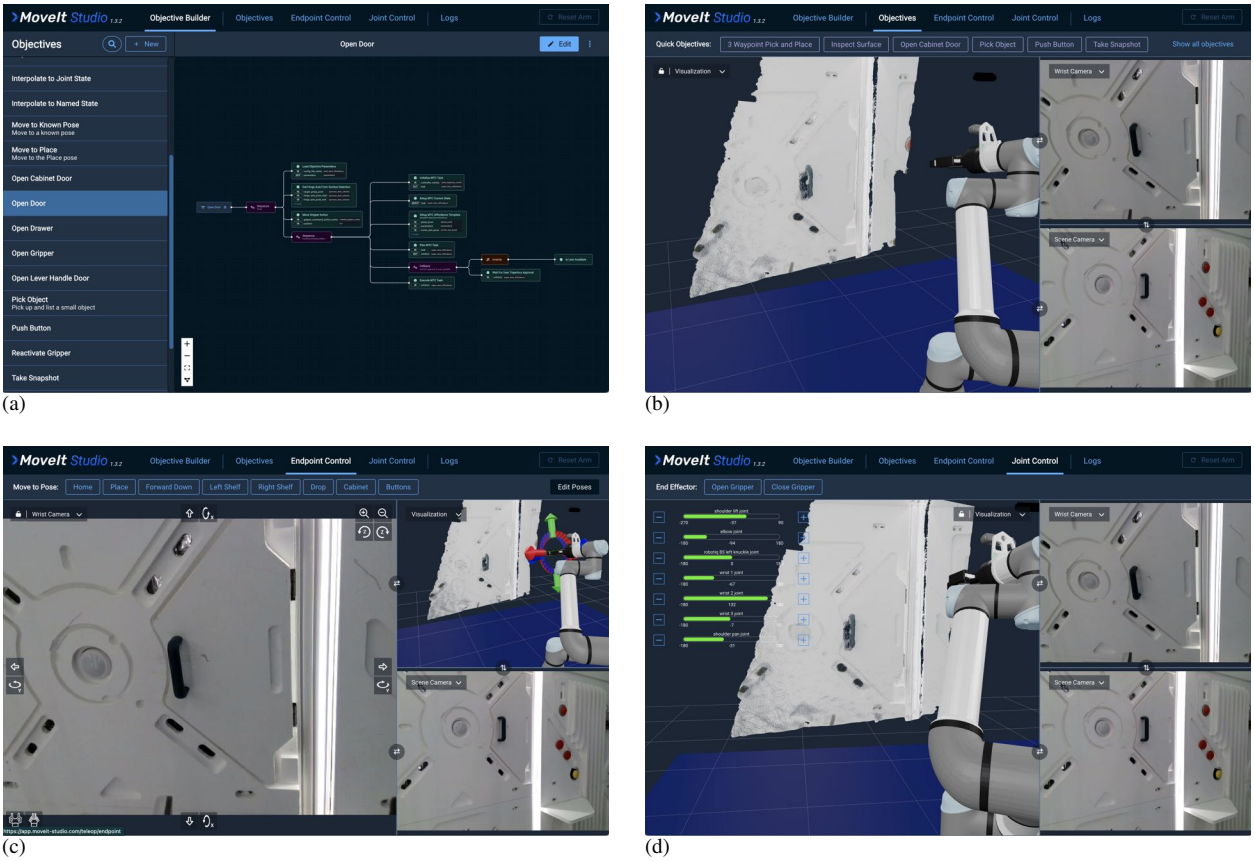

Fig. 2 User interface overview. (a) The objective builder enables users to edit built-in objectives, modeled as behavior trees, or construct entirely new ones using a library of behaviors and other building blocks. (b) Objectives capture routine operations a robotic manipulator can perform. Different views provide the operator with situational awareness. (c) Via endpoint control the view of the camera at the end of the manipulator can be adjusted via predefined poses (top row of buttons) or via arrow buttons in the main camera view pane. (d) As a fallback mechanism there is also the option to control joints directly.

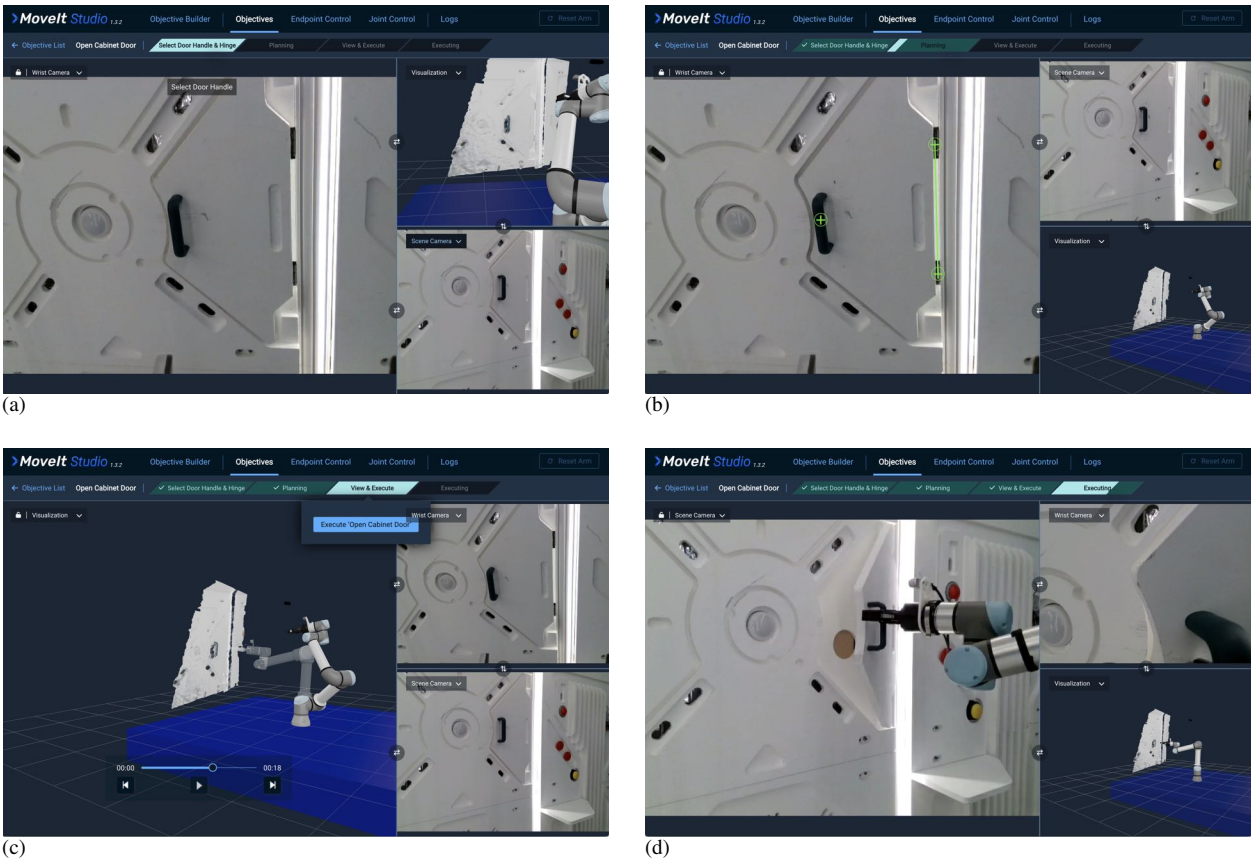

Fig. 3 Door opening objective. (a) The operator selects the inputs for the objective: a door handle location and hinge axis. (b) After the handle and hinge is selected, a plan is computed. c A solution path is shown to the operator for approval. (d) The path is executed.

objectives," the user can select any of the other defined objectives (and mark others as favorites, if desired). In this mode and the following ones, three different views are shown. By default, it shows two different camera views (from the end effector and overhead cameras) and a view of the model of the world that the robot uses for planning. The views can be changed. For example, it is possible to change one of the views to a behavior tree view. When executing an objective, the currently active behavior is highlighted, providing insight into the symbolic state of the overall system. To explore an environment and get better situational awareness, an operator can navigate a camera mounted on the wrist of the arm to different predefined poses (Fig. 2c). Additional poses can be defined by the operator for later reuse. When clicking on a button for a predefined pose, the system automatically computes a collision-free path from the current arm configuration to the desired pose. The operator can also make view adjustments via arrow buttons in the main camera view pane or open/close a robot gripper. In rare cases (e.g., to force a robot out of a singularity or joint limit), an operator may want manual joint control (Fig. 2d). When the system exhibits unexpected behavior, a developer may also find it useful to inspect some of the low-level logging information without having to log into a remote system. The UI provides access to this in the "Logs" tab (not shown). In different panes, a developer can select different types of logging information, related to different robot subsystems.

Figure 3 shows the steps of executing a specific objective, door opening, in more detail. The operator is prompted to click in the 2D camera image window to select a door handle location and hinge axis. This information is internally transformed to 3D poses of the handle and hinges using a depth image. An affordance template describes the motion constraints for opening a door. Appropriate motions for, first, grasping the door handle and, second, opening the door are previewed for approval by the operator. Once the operator approves the plan, it is executed on the robot and visualized in the UI.

V. Discussion

We have presented a software framework that allows operators to remotely control robot manipulators at various levels of autonomy. Through pre-defined primitive behaviors, more complex objectives can be defined. As the state of the art in robotics advances, our framework will seamlessly support more autonomy over longer and longer time horizons.

There are several capabilities that we plan to incorporate into our software framework. First, we will add support for mobility by adding support for Nav2 [20], the ROS 2 framework for navigation. Nav2 is already leveraging the same behavior tree implementation as used by MoveIt Studio, which facilitates this integration. With this capability, one can script, e.g., collecting samples with a planetary rover (which involves both navigation and manipulation). Second, we plan to add support for multi-armed robots, where behaviors can be executed by a given arm or as a coordinated motion by several arms. We plan to further generalize this idea by allowing users to plan for arbitrary groups of joints. This is useful for planning and execution of motions for multiple arms, an arm and torso, or arm and mobile base. Third, we are currently actively working on extending the perception capabilities to reduce the dependency on a human operator. Specifically, we are leveraging machine learning techniques to perform semantic and instance segmentation of depth images. This will make it possible to replace, e.g., manual selection of door handles in the current door opening objective with automatic door handle/hinge prediction. Perception capabilities will also be critical in inspection tasks and anomaly detection tasks. Finally, we plan to develop an operator- and developer-friendly interface for our low-level admittance controller, such that it is each to specify compliance/stiffness for each translational and rotational axis in a given coordinate frame of reference. This coordinate frame need not be fixed in the world, but can also be a moving frame. This facilitates manipulation of moving targets (e.g., door handles and valves), where the dynamical constraints are most conveniently expressed in object frame rather than world frame.

The need for the capabilities above is driven in part by client projects where MoveIt Studio is integrated in experimental space robot platforms. In the near future we plan to collaborate with manufacturers of space-rated manipulators and other space industry stakeholders to advance the technology readiness level of complete hardware and software solutions for increasing autonomy in future space missions. As the library of behaviors and list of support hardware continue to increase, we expect that it will become even easier to develop solutions for future supervised autonomy needs.

Acknowledgments

This work was supported in part by NASA contracts 80NSSC20C0609 and 80NSSC21C0539, SpaceWERX Orbital Prime contract FA8750-22-C-0093, and an Advanced Industries Early-Stage Capital and Retention Grant from the state of Colorado.

References

[1] Farrell, L. C., Strawser, P., Hambuchen, K., Baker, W., and Badger, J., "Supervisory Control of a Humanoid Robot in Microgravity for Manipulation Tasks," *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2017, pp. 3797–3802.

[2] Hart, S., Dinh, P., Yamokoski, J. D., Wightman, B., and Radford, N., "Robot Task Commander: A framework and IDE for robot application development," *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2014, pp. 1547–1554. https://doi.org/10.1109/IROS.2014.6942761.

[3] Hart, S., Dinh, P., and Hambuchen, K., "The Affordance Template ROS package for robot task programming," *IEEE Intl. Conf. on Robotics and Automation*, 2015, pp. 6227–6234. https://doi.org/10.1109/ICRA.2015.7140073.

[4] Pettinger, A., Alambeigi, F., and Pryor, M., "A Versatile Affordance Modeling Framework Using Screw Primitives to Increase Autonomy During Manipulation Contact Tasks," *IEEE Robotics and Automation Letters*, 2022, pp. 1–8. https://doi.org/10.1109/LRA.2022.3181732.

[5] Hart, S., Quispe, A. H., Lanighan, M. W., and Gee, S., "Generalized Affordance Templates for Mobile Manipulation," *IEEE Intl. Conf. on Robotics and Automation*, 2022, pp. 6240–6246.

[6] Coleman, D., Şucan, I. A., Chitta, S., and Correll, N., "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study," *J Software Engineering for Robotics*, Vol. 5, No. 1, 2014, pp. 3–16.

[7] Macenski, S., Foote, T., Gerkey, B., Lancette, C., and Woodall, W., "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics*, Vol. 7, No. 66, 2022, p. eabm6074. https://doi.org/10.1126/scirobotics.abm6074.

[8] Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Rodríguez Tsouroukdissian, A., Bohren, J., Coleman, D., Magyar, B., Raiola, G., Lüdtke, M., and Fernández Perdomo, E., "ros_control: A generic and simple control framework for ROS," *The Journal of Open Source Software*, 2017. https://doi.org/10.21105/joss.00456, URL http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf.

[9] Şucan, I. A., Moll, M., and Kavraki, L. E., "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, Vol. 19, No. 4, 2012, pp. 72–82. https://doi.org/10.1109/MRA.2012.2205651, http://ompl.kavrakilab.org.

[10] Elbanhawi, M., and Simic, M., "Sampling-Based Robot Motion Planning: A Review," *IEEE Access*, Vol. 2, 2014, pp. 56–77. https://doi.org/10.1109/ACCESS.2014.2302442.

[11] Karaman, S., and Frazzoli, E., "Sampling-based algorithms for optimal motion planning," *Intl. J. of Robotics Research*, Vol. 30, No. 7, 2011, pp. 846–894. https://doi.org/10.1177/0278364911406761.

[12] Dobson, A., and Bekris, K. E., "Sparse Roadmap Spanners for Asymptotically Near-Optimal Motion Planning," *International Journal of Robotics Research*, Vol. 33, No. 1, 2014, pp. 18–47. https://doi.org/10.1177/0278364913498292.

[13] Kingston, Z., Moll, M., and Kavraki, L. E., "Exploring Implicit Spaces for Constrained Sampling-Based Planning," *Intl. J. of Robotics Research*, Vol. 38, No. 10–11, 2019, pp. 1151–1178. https://doi.org/10.1177/0278364919868530.

[14] Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A., and Srinivasa, S. S., "CHOMP: Covariant Hamiltonian optimization for motion planning," *The International Journal of Robotics Research*, Vol. 32, No. 9-10, 2013, pp. 1164–1193. https://doi.org/10.1177/0278364913488805.

[15] Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S., "STOMP: Stochastic trajectory optimization for motion planning," *IEEE Intl. Conf. on Robotics and Automation*, 2011, pp. 4569–4574. https://doi.org/10.1109/ICRA.2011.5980280.

[16] Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T., "Integrated task and motion planning," *Annual review of control, robotics, and autonomous systems*, Vol. 4, 2021, pp. 265–293.

[17] Görner, M., Haschke, R., Ritter, H., and Zhang, J., "MoveIt! Task Constructor for Task-Level Motion Planning," *IEEE Intl. Conf. on Robotics and Automation*, 2019, pp. 190–196. https://doi.org/10.1109/ICRA.2019.8793898.

[18] Colledanchise, M., and Ögren, P., *Behavior trees in robotics and AI: An introduction*, CRC Press, 2018.

[19] Lynch, K. M., and Park, F. C., *Modern Robotics: Mechanics, Planning, and Control*, Cambridge University Press, 2017.

[20] Macenski, S., Martin, F., White, R., and Ginés Clavero, J., "The Marathon 2: A Navigation System," *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2020, pp. 2718–2725. https://doi.org/10.1109/IROS45743.2020.9341207.