Simplify the robot programming through an action-and-skill manipulation framework

E. Villagrossi, N. Pedrocchi
Institute of Intelligent Industrial Technologies and Systems for Advanced Manufacturing, National Research Council of Italy
via Corti 12, 20133 Milan, Italy
https://orcid.org/0000-0002-9493-4175
https://orcid.org/0000-0002-1610-001X

M. Beschi
Department of Mechanical and Industrial Engineering University of Brescia
via Branze 39, 25123 Brescia, Italy
Institute of Intelligent Industrial Technologies and Systems for Advanced Manufacturing, National Research Council of Italy
https://orcid.org/0000-0002-8845-2313

***Abstract*—The paper introduces a robotic manipulation framework suitable for the execution of manipulation tasks. Based on the ROS platform, the framework provides advanced motion planning and control functionalities for robotic systems to guarantee a high level of autonomy during the execution of an action. The integrated motion planning module can handle multiple motion planners to generate collision-free trajectories for a given planning scene that can be dynamically uploaded. In the same way, the robot controllers can be changed online on the base of the robot behavior required by the action under execution. The motion control of the robotic system is fully demanded to the manipulation framework relieving the upper control layers from the management of low-level functionalities and the task geometrical information. The framework can be used downstream to a task planner or as a standalone library to simplify the robot programming in complex manipulation tasks.**

***Keywords*—Robotic manipulation, robot programming**

I. Introduction

Providing a unique definition of *"robotic manipulation"* is not trivial. Literature presents a few definitions of manipulation that changes according to the field of the applications [1], and its use open to user interpretation. Since early research activities on grasping, manipulation was referred to the ability to produce a stable grasping of an object, sometimes with the terms manipulation and grasping used as synonyms. Over the years, robotic manipulation has moved to refer to a broader combination of motion planning and motion control of a robotic arm, equipped by a proper tool as an end-effector, to execute a predefined task where it is required to interact with objects in the surrounding environment [2]. In this regard, euRobotics association [3], through the strategic research agenda [4], defined *manipulation* as *"the function of utilising the characteristics of a grasped object to achieve a task"*. More in general robotic manipulation can also include perception to feed and monitor the goals of the manipulation task. Following this definition, it is possible to extend the manipulation over the classic pick&place, including many other actions such as screw, assembly, or any action where the robot interacts with the environment through a grasped object. With a similar approach, in [5] authors optimize the grasping contact selection, the grasping force, and the manipulator arm/hand motion planning to provide the arm/hand optimal posture able to guarantee a stable grasping contact.

Task management is, in general, demanded by a high-level *"task planner"*, that is a software tool supervises the overall process working at an abstract level, frequently supported by AI algorithms [6]. The task planner can be implemented following many different approaches, but the hierarchical approach to task planning is the most common one. In this case, the task planner acts both as a decision-maker and as an executor able to: (i) decompose a complex task into single actions, (ii) plan a feasible sequence of actions and (iii) supervise their executions. In this case, the task planner is involved in a massive number of operations that can be somehow coded and automated at a different level.

In [7], a layered approach was adopted, demanding the control and the execution of simple actions to a layer named *action planner*. Similarly, [8] introduced a framework that addresses manipulation tasks through three interconnected layers: plan and execute (high-layer), learn from the experience (medium-layer), and deal with the motion planning (low-layer).

A few examples are currently available as software applications dedicated to robotic manipulation. Primarily, they are devoted to grasping simulations such as the pioneering library *GraspIt!* [9] developed to simulate the performance of robotic hands. More in general, some well-known software packages, such as ADAMS or MATLAB, can be used to simulate grasping, even if they are not specifically designed for robotics. However, they cannot be used as an integrated framework to plan and control the motion of a robotic system (arm + grasping system) applied to a manipulation task.

Closer to the recent definition of manipulation, (i.e., integrated planning and control), the *MoveIt!* library [10], [11] provides a package able to perform a pick&place task but with limited features and a rigid definition of the manipulation pipeline. Indeed, it considers a unique grasping pose of a unique object at time, allowing the path planning for the approaching movement, the reaching of the picking or grasping pose, and the leaving movements. Recently, [12] has merged *MoveIt!* with a framework that compose tasks taking into account the planning context.

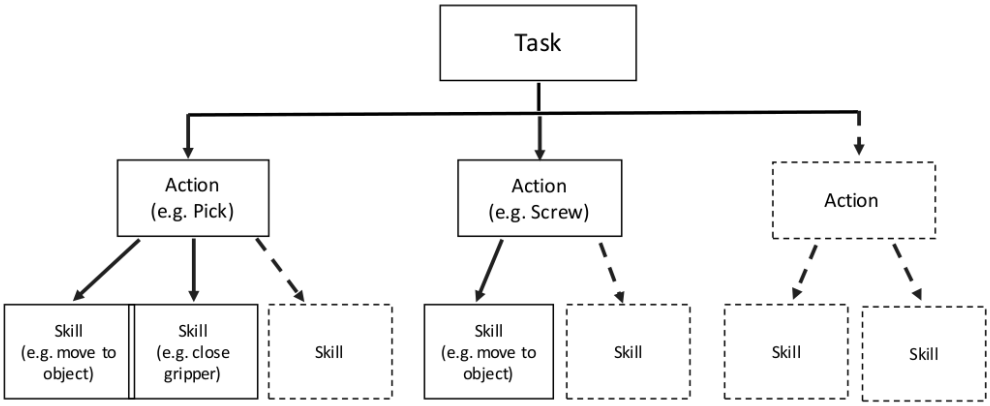

Figure 1. *Task*, *Actions* and *Skills* hierarchy.

A. Motivation and contribution

For a given task, this work aims to present a manipulation framework able to:

(i) process manipulation actions (such as pick, place, pick&place, screw, packing, etc...) with a certain level of autonomy relieving the task planner, and eventually the action planner, from the management and the execution of simple actions;
(ii) handle the kinematics model of the robotic system (robotic arm + grasping system) being able to compute forward and inverse kinematics;
(iii) embed motion planning functionalities to generate collision-free trajectories for a given planning scene and a given robotic system (robotic arm + grasping system) in a planning environment that can dynamically change;
(iv) be as modular as possible to be easily extended by the user with custom actions.

The goal was reached starting from the analysis and the formalization of a generic manipulation task. The task decomposition into simple actions (*e.g.* pick, place, screw, etc...) and consequently into skills (*e.g.* move to, close gripper, switch on screwer, etc...) allowed to define general data structures flexible enough to be reused in all the manipulation actions. In the same way, basic functionalities such as motion planning and kinematics model management were grouped into basic modules to be easily reused.

The manipulation framework embeds a motion planner to plan optimal collision-free trajectories between the robotic arm, the grasping system, and the surrounding environment. The planning scene is shared between multiple actions and it can be easily updated by information coming from a perception system. In general, the use of the manipulation framework as a standalone library, without involving task and/or action planners, can simplify the robot programming task avoiding the user to make long and tedious programming sessions to define collision-free trajectories.

The paper is organized as follows: Section II provides basic definitions, data structure introduction, and manipulation framework modules description. Section III shows how the manipulation framework was developed and Section IV provides conclusions and future works.

II. Framework Definitions and Modules

Within the scope of this work, we are going to use the following nomenclature:

***Task*:** it is the final goal to be reached by the robotic system represented by a group of *Actions*. The *Task* decomposition into single *Actions* is made by a task or an action planner that assign single *Action* to the manipulation framework.

***Action*:** it is a sequence of elementary movements made by the robotic system, examples of *Actions* are *pick, place, screw, etc....*

***Skill*:** it is the elementary movement executed by the robotic system, a group of *Skills* forms an *Action*. For example, the pick *Action* can be decomposed in the *Skills*: move to the approach position, move to the object grasping position, actuate the grasping system, move to the leave position.

Figure 1 shows *Task*, *Action* and *Skill* hierarchy.

A. Data structures

The analysis of manipulation *Actions* allows to define the following generic data structures.

1) Location: is the elementary data structure for a generic manipulation pose (position + orientation) in the free space defined as:

```
Location:
    string  name
    string  frame
    Pose    pose
    Pose    approach_pose
    Pose    leave_pose
```

where `name` and `frame` are two strings that identify the unique *Location* name and the reference frame in which the *Location* is defined. The elements `pose`, `approach_pose`

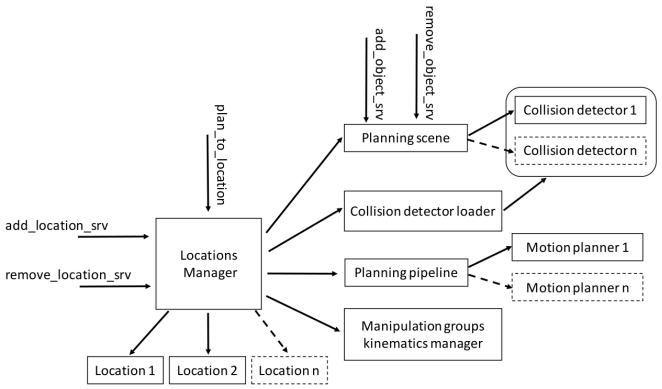

Figure 2. *Locations Manager* module description.

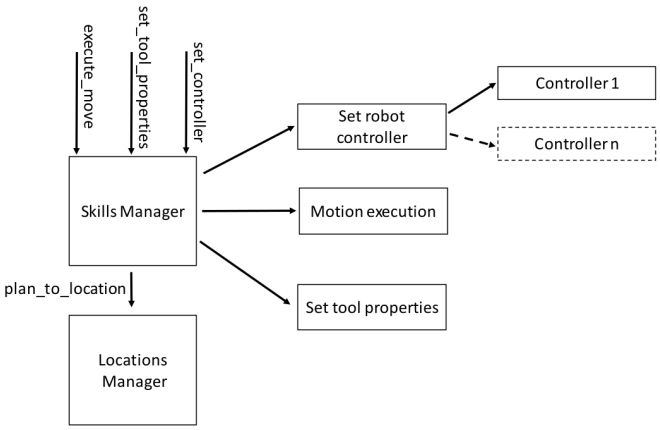

Figure 3. *Skills Manager* module description.

and `leave_pose` are respectively the exact, the approach and the leave poses for a given *Location*.

2) Grasp: is the description of a grasping *Location* defined as:

```
Grasp:
    string      tool_name
    Location    location
```

where the string `tool_name` is the grasping tool name to be used for grasping, while `location` is the grasping *Location* as defined in Section II-A1.

3) Object: is the description of an object that needs to be manipulated. An object can be grasped with different grasping poses, *e.g.* the grasping pose can have multiple tool orientation to grasp the same object in the same position. To this scope the *Object* data structure is defined as:

```
Object:
    string  name
    string  type
    Grasp[] grasping_locations
```

where `name` and `type` are two strings that represent the *Object* unique name and family type. The `grasping_location` is an array of *Grasp* type, as defined in Section II-A2, describing all the possible *Locations* that can be used to grasp the object. The grasping locations are treated with the same priority and the choice is demanded to the motion planner on the base of the minimum distance collision free trajectory.

4) Box: is the recipient where an object can be grasped, each *Box* can contain multiple *Objects*, the data structure is defined as:

```
Box:
    string name
    Object[] objects
    Location location
```

where `name` is the unique name of the *Box*, `object` is the array of the type *Objects* contained in the *Box* and `location` is its *Location*.

5) Slot: is the description of the place where an object can be released, the data structure is defined as:

```
Slot:
    string name
    int slot_size
    Location location
```

where `name` is the unique name of the *Slot*, the `slot_size` is an integer that describes the capacity of the *Slot*, the value can be: minor than 0 if the slot has infinite space (*e.g.* to simulate a conveyor track that remove immediately the object once the object is released in the *Slot*), equal to 1 for single space slot (in this case only one object at a time can be contained) and major than 1 for slot that can contain contemporary multiple objects (*e.g.* to simulate a basket where an object falls after the release). The *Slot* data structure does not include information about the dimensions of the contained objects, so the capacity need to computed and fixed by the user. Finally, `location` is the *Location* of the slot.

B. Modules

1) Locations Manager: this module handles an array of *Locations* (Figure 2). The module dynamically adds and removes *Locations*, and it embeds a planning pipeline, *i.e.*, it allows the use of different motion planners, planning scene and a kinematics modules according to the user needs. Every time a new *Location* is added the inverse kinematics for a given move group (robotic arm + grasping system) is computed and stored, when a new trajectory planning is required the planning scene is updated and used to generate the trajectory. The planning scene is evaluated only one time before planning a new trajectory, the online trajectory re-planning is not supported. The trajectory planning is made avoiding robot self collisions and collisions between the robotic system (*i.e.* arm + grasping system + manipulated object if present) and the entities in the scene. A *Locations Manager* can handle multiple move groups (*i.e.* multiple robotic arms). The planning scene is unique and shared between multiple *Locations Manager*, objects can be dynamically added, removed and uploaded depending on the

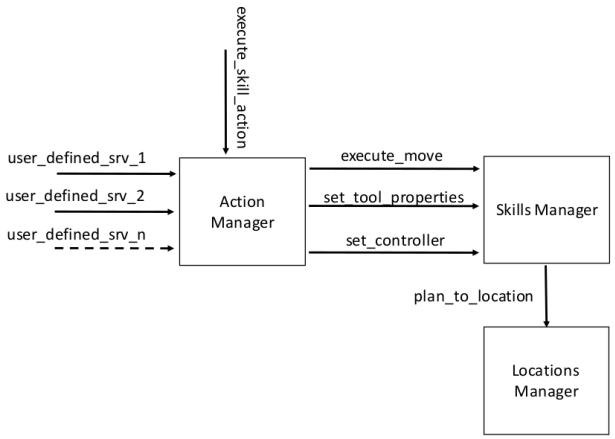

Figure 4. *Action Manager* module description.

real scene evolution, a perception system can be used to this scope. The collision detector can be also changed online.

2) Skills Manager: this module is the core element for all the *Actions*. As shown in Figure 3, the module allows loading/unloading of the robot controllers according to the running *Skill* (*e.g.* a standard trajectory tracker controller may be used for the approach movement, or a Cartesian impedance controller may be used when it is required a peg-in-hole *Skill*). The *Skills Manager* module provides the ability to start and monitor the execution of the trajectories planned by the *Locations Manager* for a specific move group. Finally, the module enables the control of the tool required by the *Action* *e.g.* for a screw action set the screwer type, set the tightening torque and the number of rounds per minute of the screwer.

3) Action Manager: this module supervises the execution of specific *Actions* defined by the user. As shown in Figure 4, the module is based on a *Skills Manager* and which in turn includes a *Locations Manager* inheriting all the functionalities previously described. The decomposition of a specific *Action* into *Skills* is defined by the user, *e.g.* the action place object can be decomposed into the following elementary *Skills*: move to the approach position, move to the release position (*i.e.* on the desired slot), release the grasped object, move to the leave position.

The manipulation framework provides some standard *Actions* such as *pick* objects or *place* objects, but the user can develop custom *Actions*. In the same way, the user has to create services to interact with the module, *e.g.* for a pick object action: add/remove objects to be picked or add/remove picking boxes. Once a new action is taken in charge, the *Action Manager* module supervises the action execution, checks if all the *Skills* are properly completed and partially manages unexpected behaviours, if severe errors occurs an error message is returned to the action planner and the action execution is stopped. This approach relieves the task and the action planners from the management of *Actions* and atomic *Skills*, as the actions geometrical information and the path planning that is completely demanded to the manipulation framework.

C. Interconnections and Architecture

A general overview for the manipulation framework is provided in Figure 5, the scheme shows the layered architecture. The three base layers are described in Sections II-B1, II-B2 and II-B3. The bottom layer (the green layer) has in charge the geometrical information management and the motion planning. The intermediate layer (the orange layer) manages the trajectories execution, the robot controllers and tools management. Finally, the top layer (the blue layer) deals with the *Actions* decomposition and the *Skills* execution.

The action planner can add and remove dynamically *Actions* and multiple *Actions* can contemporary exists. The scheduling of the *Actions* is in charge to the action planner or to a task planner, in general, the manipulation framework can be used by the user even without the presence of task/action planner on the above levels, but by simply exploiting the powerful motion planning and motion control functionalities.

Each *Action* is isolated from the others without exchanging information, *e.g.* the *Locations* referred to an *Action* are stored in its relative *Locations Manager*. The only exception is represented by the planning scene that is shared between all the *Locations Managers* and by other entities if necessary.

III. Implementation

The manipulation framework was developed as *C++* library based on ROS [13]. In Figure 6 all the external libraries used to develop the modules described in Section II are highlighted.

The motion planning features are based on the well-known *MoveIt!* [10]. *MoveIt!* was used to exploit its planning pipeline and its planning scene. Thanks to the plugins mechanism provided by ROS, the motion planners can be dynamically loaded by using the state-of-the-art motion planning algorithms provided by *MoveIt!*, such as OMPL, CHOMP, STOMP, or third party algorithms such as [14].

The planning scene enables the collision checking, where, as for the motion planners, collision detectors can be loaded as plugins. The manipulated objects can be added to the scene and all the possible collisions with the environment, during the manipulation, are evaluated by the motion planner. Every time a new trajectory planning is required, the *Locations Manager* clones the common planning scene to check collisions and generates a collision-free path plan. The evaluation of the scene is made at the beginning of the path planning, so further updates (*e.g.* movements of the operator in the scene) coming after the cloning cannot be used for an online path re-planning. Objects in the scene can be dynamically added, removed and updated (*i.e.* their position) while the integration with a perception system can provide information about the moving elements in the scene such as the presence of operators in a collaborative application.

The planning time is hardly predictable because is strongly related by several factors such as: the complexity of the planning scene, the collision detector and the motion planner.

The move groups kinematics are handled by *MoveIt!* with the exception of the inverse kinematics that is computed through the header-only library *RosDyn* [15], instead of the

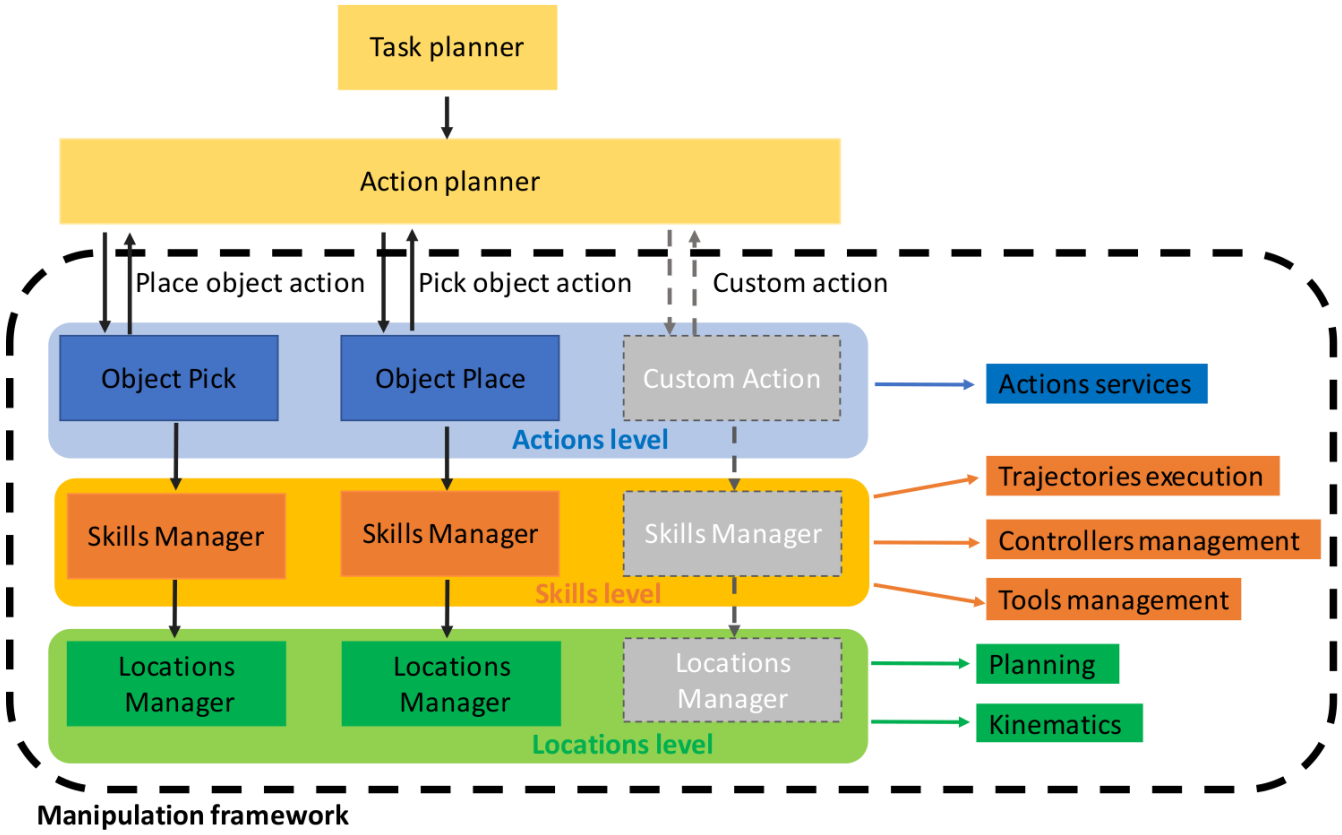

Figure 5. Manipulation framework layers description.

MoveIt! IK native function. *RosDyn* is able to get better computation time performance that in fundamental when the presence of a huge number of objects, associated to multiple grasping poses, impose the computation of thousands times the inverse kinematics.

The robot controllers can also be set online and smoothly changed during the execution of different *Skills*. The dynamic change of controllers provides a high degree of flexibility to the manipulation framework being able to adapt the robot behavior on the base of the specific *Skill* under execution, *e.g.* trajectory tracking controller when the robot needs to move accurately on the desired position or impedance control when the robot interacts with the environment. The library [16], an extension of standard ROS control architecture [17], allows to seamlessly start/stop ROS controllers, with the main difference that complex control architecture, with multiple nested controller, can be defined and loaded. The library [16] enables controllers communication by shared memory, instead of the standard publisher/subscriber ROS mechanism, providing high communication performance and enabling the concatenation of a high number of controllers. The library [16] provides standard ROS controllers interfaces supporting also the use of state-of-the-art ROS controllers.

The manipulation framework is an open source library available in the public repository: https://github.com/JRL-CARI-CNR-UNIBS/manipulation. Currently there are three predefined *Actions* available in the manipulation library: *pick* objects, *place* objects and *go to* location.

Examples about the use of the manipulation frameworks are available in the public repository: https://github.com/JRL-CARI-CNR-UNIBS/manipulation_examples.

IV. Conclusion and Future Works

The paper introduced a manipulation framework designed to address manipulation tasks. The framework is designed to have a certain level of autonomy in processing *Actions* and *Skills* relieving task and action planners from the management of low level functionalities as the robotic system (arm + grasping system) motion plan and control and objects geometry information.

The framework manages the robotic system kinematics and embeds motion planners able to generates collision-free trajectories. The collision check is made by querying a planning scene that can be dynamically updated and connected to a perception system. The robot controllers can be started/stopped on the base of the required *Skills*.

The framework is provided with three basic *Actions*, *pick*, *place* and *go to* location, but it is designed to be easily extended by the user with custom actions.

The framework is developed also with the intention to simplify the robot programming for the users that want to

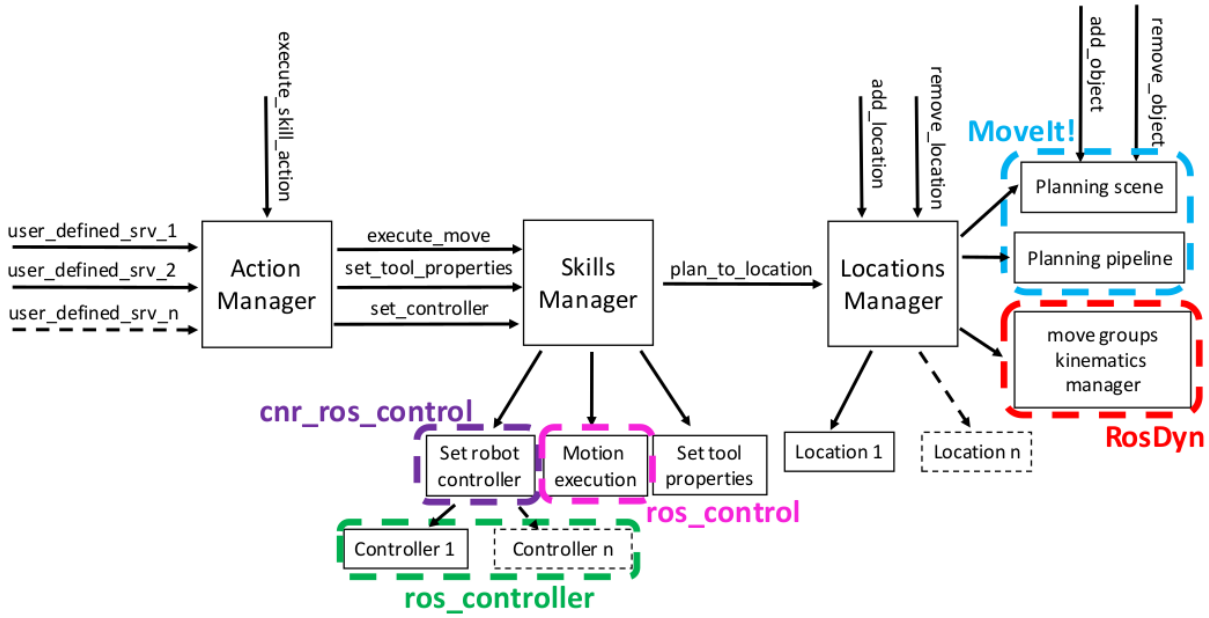

Figure 6. Manipulation framework external dependencies.

use it as a standalone library avoiding tedious robot program development to generate collision-free trajectories.

The package is an open source project that is continuously evolved and improved, future works will include the development of new *Actions* to cover a wide range of real robotic applications. Moreover, a Human Machine Interface (HMI) will be developed to further simplify the usage of the package also for the users that are not familiar with ROS and more in general with robotic programming. The combination of the framework and the HMI has the final goal to reduce the programming time w.r.t. the standard programming interfaces (*i.e.* robot tech pendant and offline programming softwares).

ACKNOWLEDGEMENT

This work is partially supported by ShareWork project (H2020, European Commission – G.A. 820807).

REFERENCES

[1] M. T. Mason, "Toward robotic manipulation," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, no. 1, pp. 1–28, 2018. [Online]. Available: https://doi.org/10.1146/annurev-control-060117-104848

[2] R. Terake, "Robot manipulation: Perception, planning, and control," Downloaded on March 2021 from http://manipulation.csail.mit.edu/ 2020.

[3] euRobotics aisbl, "eurobotics aisbl (association internationale sans but lucratif)." [Online]. Available: https://www.eu-robotics.net/

[4] euRobotics, "Strategic research agenda for robotics in europe 2014-2020," Downloaded on March 2021 from https://www.eu-robotics.net/cms/upload/topic_groups/SRA2020_SPARC.pdf 2014.

[5] M. B. Horowitz and J. W. Burdick, "Combined grasp and manipulation planning as a trajectory optimization problem," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 584–591.

[6] G. Evangelou, N. Dimitropoulos, G. Michalos, and S. Makris, "An approach for task and action planning in human–robot collaborative cells using ai," *Procedia CIRP*, vol. 97, pp. 476–481, 2021, 8th CIRP Conference of Assembly Technology and Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827120314931

[7] M. Faroni, M. Beschi, S. Ghidini, N. Pedrocchi, A. Umbrico, A. Orlandini, and A. Cesta, "A layered control approach to human-aware task and motion planning for human-robot collaboration," in *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 2020, pp. 1204–1210.

[8] M. Diab, M. Pomarlan, D. Beßler, A. Akbari, J. Rosell, J. Bateman, and M. Beetz, "Skillman — a skill-based robotic manipulation framework based on perception and reasoning," *Robotics and Autonomous Systems*, vol. 134, p. 103653, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889020304930

[9] A. T. Miller and P. K. Allen, "Graspit! a versatile simulator for robotic grasping," *IEEE Robotics Automation Magazine*, vol. 11, no. 4, pp. 110–122, 2004.

[10] O. source community, "Moveit! moving robots into the future." [Online]. Available: https://moveit.ros.org/

[11] D. T. Coleman, I. A. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 3–16, 2014.

[12] M. Görner, R. Haschke, H. Ritter, and J. Zhang, "Moveit! task constructor for task-level motion planning," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 190–196.

[13] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org

[14] M. Beschi and M. Faroni, "Human aware motion planning." [Online]. Available: https://bitbucket.org/iras-ind/human_aware_motion_planners/src/master/

[15] M. Beschi and N. Pedrocchi, "Rosdyn library." [Online]. Available: https://github.com/CNR-STIIMA-IRAS/rosdyn

[16] ——, "Cnr ros controller." [Online]. Available: https://github.com/CNR-STIIMA-IRAS/cnr_ros_control

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