An Action Interface Manager for ROSPlan

Stefan-Octavian Bezrucav,[1] **Gerard Canal,**[2] **Michael Cashmore,**[3] **Burkhard Corves**[1]

[1] Institute of Mechanism Theory, Machine Dynamics and Robotics, RWTH Aachen University
[2] Department of Informatics, King's College London
[3] Department of Computer and Information Sciences, University of Strathclyde

Abstract

Task planning and task execution are two high-level robot control modules that often are working with representations of the scenario at different levels of abstraction. Thus, a further mapping module is required to connect the abstract planned actions to the robot-specific algorithms that must be called in order to execute these actions.

We present a novel implementation of such a module that allows a user to define this mapping for all actions through a single configuration file. This greatly reduces the amount of effort that is required to integrate an automated planner with a robotic platform.

This module has been integrated as an Action Interface of the automated task planning framework ROSPlan, and includes a Graphical User Interface though which the configuration file can be easily generated and updated. The use of the interface is demonstrated in two scenarios: with robot actors possessing only a single action, and a more complex scenario with multiple agents and types of actions.

1 Introduction

Task planning is the process of determining the actions that must be executed, the order in which they should be dispatched, and the actors that should carry them out in order to achieve the set goals. This process is usually done on an abstract and simplified representation of the considered scenario. A complete representation of the environment, the actors, and their possible actions is not feasible for the planning process as the solvers are not capable of handling that much information. On the other hand, the execution of the planned actions, whether in a real or simulated scenario, must consider these features and interactions. Thus, a flexible, but yet robust mapping between the abstract planned actions and their execution is required.

In this paper, we present the description of such a mapping interface for actions of plans generated with automated planning approaches and written in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998). Furthermore, we describe how it has been integrated in the state-of-the-art task planning framework ROSPlan (Cashmore et al. 2015). The strength of this interface is that the mapping can be completely defined through a single configuration file, reducing the need for a hard-coded interface between PDDL action and action implementation. Moreover, this configuration can be generated through the use of a Graphical User Interface.

In the next section we provide some background on the Robot Operating System (Quigley et al. 2009) (ROS) that is necessary to understand the particulars of interfacing actions. In Section 3, we analyze how different frameworks handle this mapping process in simulated or real robotic applications working with ROS. In Sections 4 and 5 we present the implementation of our action interface. The aim of this interface is to reduce the effort required to interface PDDL actions with a ROS action implementation, while maintaining flexibility in plan execution. This benefit is demonstrated in two simulated robotic scenarios described in Section 6. The Graphical User Interface is presented in Section 7.

2 Background

The Robot Operating System (ROS) (Quigley et al. 2009) is one of the standard meta-operating systems used in the robotic community. It sustains the integration of different algorithms in so-called ROS *nodes* and provides different structures for communicating data between these nodes. The basic communication structures are the *topics*, over which messages are routed according to the publish/subscribe paradigm, and the *services* that are based on the request/reply paradigm (ROS.org 2021). In addition, *actionlibs* are a complex communication strategy based on topics, that provide long-term non-blocking request/feedback/reply structures.

The robotic software realising the action execution considered in this paper is implemented in ROS as nodes, each of which offers either a service or actionlib interface. Through these communication structures, the algorithms integrated in each ROS node can be called or triggered and responses can be received and further processed. If it is required to call or trigger multiple ROS nodes in a given sequence, this can be realized through a Moore Finite State Machine (FSM) (Moore 1956), whose states represent such calls or triggers and whose transitions are dependent on the obtained responses.

Different task planning frameworks integrated in ROS use the above presented communication structures to trigger the concrete execution of the planned actions on each involved actor. The ROSPlan framework (Cashmore et al. 2015) is composed itself of different ROS nodes. The *Plan Parsing* node is responsible for parsing the generated plan file in an internal C++ representation and for sending it to the *Dispatcher* node. The latter selects actions to be executed and sends them over a topic to *Action Interface* nodes that interpret the command and then interface with the corresponding robotic algorithms through services or actionlibs. Once the respective executions have finished, the interface passes this information to the dispatcher in form of an action feedback message. The dispatcher then decides how to continue.

Action interfaces are typically implemented per robotic algorithm with which they interface. Given the breadth of different actions offered by different robots within ROS, applying a framework for task planning such as ROSPlan might require the implementation of many such interface nodes. The contribution of this paper is a single light-weight node that is able to interface any action through the use of a configuration file.

3 Related Work

In this section we discuss how different frameworks handle the mapping of planned actions to their implementations in ROS. While there are many approaches to plan representation and execution discussed here, they must all handle the problem of interfacing with the services and actionlibs provided by ROS libraries.

In the modified ROSPlan framework developed by Sanelli et al. (2017) the generated plan is transformed into a Petri-Net Plan (PNP) and then passed to the PNP executor. The latter communicates with the software of the robot over the *PNP Service* and *PNP Action Server* modules, through services and actionlibs. Another modified version of the ROSPlan is presented in Harman et al. (2017). The authors have adapted the framework to communicate not only with the ROS nodes of the integrated robots, but also with IoT devices from the environment. They have replaced the ROSPlan *Action Interfaces* nodes with *Action Executor* nodes, but use the same communication strategies to the robots and the IoT devices, over ROS actionlibs.

In other works, as in the MaestROB framework (Munawar et al. 2018), the communication to the low-level robot controllers is done over the Intu middleware and corresponding ROS-Bridges. The communications structures are similar, as they are based on the publisher/subscriber concept. PLATINUm is another framework that uses the ROS-Bridges to send commands to the *Motion Planning* module of the serial manipulator and to receive the execution feedback from it (Cesta, Orlandini, and Umbrico 2018). ROS-TiPlEx (Viola et al. 2019), is a tool that allows the interaction between a planning and an expert in robotics for designing the planning problem and corresponding low-level control strategies for dispatching the planned actions. ROS-TiPlEx uses a timeline planning approach, while the orchestration of the low-level controls to ROS modules is based on FSMs. The tool comes with a GUI that simplifies the configuration process and improves the information sharing between the experts.

In Darvish et al. (2018), the FlexHRC architecture is presented. Similar to the above frameworks, it has a *Robot Execution Manager* that receives the high-level plan and sends individual actions to the *Controller*. The latter sends further low-level commands to the robot itself, which returns sensory data in form of a feedback, using similar communication structures to ROS. Furthermore, the *Controller* maps the received actions to one low-level command or to a sequence of low-level commands, similar to a FSM action interface. A similar approach for grouping *atomic actions* in *compound skill*, which are hierarchical and concurrent state-machines, is utilized in (Johannsmeier and Haddadin 2017). In a similar manner, in (Munawar et al. 2018) they refer to the execution triggered by only one call as a *gesture*, while the execution of a sequence of such calls integrated in a structure equivalent to an FSM is named a *skill*.

The contribution of this paper is a general solution, in ROS, for the action interface component that all of these related approaches must implement. Specifically we present a highly flexible action interface node for the ROSPlan framework. Through the new action interface the concrete implementation of planned actions can be easily and intuitively configured. The interface allows the creation and the modification of execution strategies, including complex ones such as hierarchical finite state machines, in a user-friendly manner. Only a configuration file is required, which can be created through user interface, and no code must be generated.

4 ROSPlan Action Interface

In this section, we discuss the limitations of the existing action interface used in ROSPlan. The initial implementation of the mappings between the abstract PDDL actions and the low-level concrete ROS modules is through C++ code. Users could also implement such custom interfaces between ROSPlan and lower-level executors in the language of their choice (i.e., Python) handling all the updates to the knowledge base and planning state, which are handled in ROSPlan's abstract C++ interface. For each action defined in the PDDL domain file of the planning instance, one action interface had to be defined as a ROS node. This action interface integrates the concrete implementation that interacts with the different ROS modules. For example, consider the action interface of a *move* PDDL action. This contains the generation of a ROS goal message and a call to the MOVE BASE actionlib of the actor, which achieves that goal. Further strategies, as cleaning the map in case that the first execution command to MOVE BASE has failed, must also be hard-coded into this interface.

Each planned PDDL action is a grounded action. This means that it should be executed for the specific values of its parameters. For example, the *move* action might have as parameters the actor *agent*, the starting pose *from* and the goal pose *to*. This implies that the *move* action interface must be able to interpret this and generate the correct ROS actionlib goal for all possible variations of these parameters. This challenge is also tackled in the hard-coded implementation, often with the definition and the integration of an action-specific configuration file. In the example case, the *move* action interface takes as input a *poses* configuration file, where all possible values of the *to* parameter are mapped to specific coordinates required to generate the corresponding MOVE BASE goal.

Although the actual action interfaces are apparently easy to generate, they actually have some important limitations. The most relevant one is in scalability. For each new PDDL action defined in the domain file, a new ROS node must be generated. Moreover, if the PDDL action has one or more parameters whose values are relevant for the execution, a specific configuration file must also be created.

Other issues are the reusability and readability. Each action interface may require an implementation that is unique to both the action implementation and the planning domain, as well as a configuration file that is unique to the planning instance. Thus, it is hard to define and maintain a clear structure that can be easily understood and modified by new users, or reused in case new PDDL actions are defined. As a consequence, code replication is common among interfaces for similar actions. This also complicates the code maintainability.

5 Action Interface Manager

The issues presented in the previous section motivated the remodelling of the ROSPlan action interfaces. Their new structure and the advantages that they bring with them are detailed in the following.

Implementation

This new action interface has been implemented in Python. Similar to ROSPlan's sensing interface (Canal et al. 2019), we have exploited the Python's language reflection capabilities to load at run-time all the required modules to run the low-level controllers.

As depicted in Figure 1, we developed an action interface manager (AIM) that loads the single configuration file and creates the corresponding interfaces to different low-level modules for each of the actions. This object is also in charge of receiving the action dispatch messages that indicate which (grounded) PDDL action should be executed, and starting the corresponding interface. When the action execution finishes, the interface informs the dispatcher about the outcome of the execution.

We have developed three action interface types: Actionlib, Service, and FSM. However, our architecture has been designed such that it is easy to extend to handle other types of interfaces in a similar manner, such as to IoT devices, provided users write ROS wrappers for those APIs. In addition to specifying the type of each interface, the configuration file describes how each interface should construct service requests and actionlib goals, as well as their expected feedbacks or results. In this specification, PDDL parameters and values stored in the ROS parameter server can be accessed using the syntax `($pddlparam x)` and `($rosparam x)`. This allows the same action interface configuration to scale to actions with many possible parameter variations without requiring additional lines. The interface types are described at a high-level below, followed by example configurations. The complete specification of the configuration file is presented in Appendix A.

Service and Actionblib action interfaces We provide interface implementations to the basic ROS communication

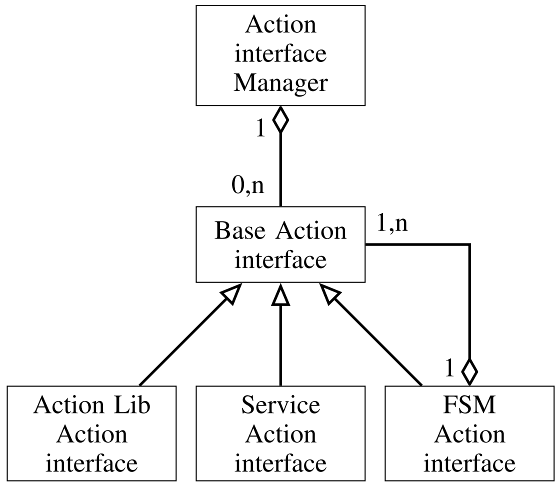


Figure 1: UML Diagram of the the Action Interface classes

protocols: actionlibs and services. They are both provide request/reply interaction, the former non-blocking and the latter blocking.

Both mechanisms require a request or goal to be achieved, and send back a response with the result once finished. The configuration file describes (a) the address of the service/actionlib, (b) how the default requests are constructed based on the PDDL parameters of the action, possibly using data from the ROS Parameter Server, (c) how these default requests may be overridden for specific combinations of PDDL parameters, for which a different request may be required, and (d) a description of the expected response to a service or actionlib call that indicates the action has been successfully executed.

When a planned action is dispatched, the AIM retrieves the configuration to run the action, and passes it to the corresponding interface. Then, the corresponding interface builds the request message, calls the underlying interface, checks its response when the action execution is completed, and returns the result to the AIM. If it has been defined, the expected response is used to determine whether the PDDL action has succeeded.

FSM action interface The third interface describes action executions as a Finite State Machine (FSM), allowing for more complex action definitions. The action interface defines a set of named states that could be executed, each linked to an action interface, including another nested FSM. Each state defines the transitions to other states in the case of successful or failed execution. Each transition defines both the next state that will be executed (by name or to the special state "goal_state" that completes the FSM), and the PDDL effects that will be applied before transitioning to the next state. The formulation of an FSM action is described fully by Bezrucav and Corves (Bezrucav and Corves 2020). The use of the composite design pattern allows the reuse of existing interfaces, while remaining powerful enough to create complex hierarchies of state machines.

Integration in ROSPlan

Figure 2 depicts the interactions with different components of the ROSPlan framework. The new action interface is an independent node subscribed to the ROSPlan Dispatcher topics to be informed when actions need to be executed, and publishes feedback with the result of the execution.

The new node is fully compatible with existing action interfaces, which can be run alongside the new action interface manager. The dispatch message of an action will be ignored by the AIM if it is not in the configuration file, and will be picked by the corresponding legacy action interface, which will start the action execution.

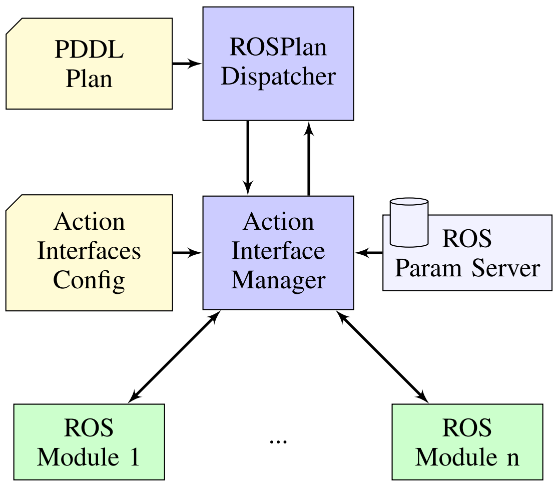


Figure 2: Interactions between the the ROSPlan framework, involved robots, ROS Parameter Server, plan file, and action interface configuration.

6 Configuration Examples

All task planning frameworks for robotics created so far contain a module through which the planned actions are mapped to their concrete execution. Because there is no standard for the development process of such a module, the implementations are diverse. This issue makes it hard to determine a set of indicators with respect to which such modules from different projects can be compared. With this motivation, in this section we demonstrate exactly what is required to map between the PDDL actions and their concrete execution with the AIM. Namely, the user needs to generate the configuration file. Two examples of the configuration files are presented:

1. The Navigation Scenario is a relatively simple scenario with only one PDDL action.
2. In the Factory Scenario, the execution of seven different complex actions are configured, for two different types of actors in a factory.

Navigation Scenario

In the navigation scenario, one or more mobile base robots are tasked with navigating around a known map. The goal

```
;; Move between any two waypoints, avoiding terrain
(:durative-action goto_waypoint
  :parameters (?v - robot ?from ?to - waypoint)
  :duration ( = ?duration (distance ?from ?to))
  :condition (and
    (at start (robot_at ?v ?from)))
  :effect (and
    (at end (visited ?to))
    (at start (not (robot_at ?v ?from)))
    (at end (robot_at ?v ?to)))
)
```

Listing 1: The *goto_waypoint* PDDL action of the Navigation Scenario.

```
1  actions:
2    - name: goto_waypoint
3      interface_type: actionlib
4      default_actionlib_topic: /($pddlparam
       ↪  v)/move_base
5      default_actionlib_msg_type:
       ↪  move_base_msgs/MoveBase
6      default_actionlib_goal:
7        target_pose.header.frame_id: "map"
8          target_pose.pose.position.x: ($rosparam
           ↪  /wp/($pddlparam to))[0]
9          target_pose.pose.position.y: ($rosparam
           ↪  /wp/($pddlparam to))[1]
10         target_pose.pose.orientation.w: 1
```

Listing 2: Complete configuration for the navigation scenario.

of the automated planning problem is that each waypoint has been visited once by any robot, and from a planning perspective is very simple. The single PDDL action is named *goto_waypoint*, and is depicted in Listing 1.

The execution of this action is carried out by the MOVE BASE module. Listing 2 shows the complete configuration file for the scenario, which connects the *goto_waypoint* action with MOVE BASE. The action configuration specifies the actionlib topic for the action, based on the *robot* PDDL parameter (line 4). The target coordinates of the MOVE BASE action are set from values stored in the ROS parameter server, using the PDDL parameter *to* as key (lines 8-9). This is everything a user needs to supply to connect ROSPlan with MOVE BASE. As an informal comparison, the preexisting C++ interface to MOVE BASE consists of 192 lines of code across two files and must be configured separately for each robot.[1] This stark contrast illustrates the utility of the new interface: it is fast and low effort (in this case 10 lines) to interface actions and assemble an automated planning system for ROS.

[1]The preexisting MOVE BASE interface in the ROSPlan repository https://github.com/KCL-Planning/rosplan_demos/tree/master/rosplan_demos_interfaces/rosplan_interface_movebase.

Factory Scenario

The AIM is demonstrated in the simulated Factory Scenario[2] developed as part of the Sharework[3] project. In this scenario there are two agents, a human and an Autonomous Guided Vehicle, which can execute seven different complex actions: *move*, *load*, *unload*, *attach tool*, *detach tool*, *manipulation action 1* and *manipulation action 2*. For the *move* action only one PDDL durative-action is defined, while for the other six actions two PDDL durative-actions are created, one for each type of actor. Furthermore, each of these actions requires a *multiple-step execution* with *failure-catching structures*. Multiple-step execution means that the execution of the planned action requires calling multiple ROS module in sequence to successfully execute. Failure-catching structures means that for the safe and robust execution of the action, failed execution of any step requires additional steps to return the system to a safe state from which further planning can occur. Given these requirements, a FSM action interface is configured for each PDDL action.

An excerpt of the configuration for the *move* PDDL action is depicted in Listing 3. As mentioned above, a finite state machine is required to describe the sequence of basic actions that need to be executed in order to reach the goal state or to prevent error situations. In Listing 3 the first two states of that FSM are presented. The execution of the first state (lines 5-17) is independent on the grounding of the PDDL *move* action. Therefore, the default values for the message type, topic, goal and result are set (lines 7 - 12). The default behaviour is to call a service named "action failure". Such services are helpful in simulations, where failures may need to be generated on purpose.

Upon success of the first state (lines 14 - 15), the execution is continued with the state *ba1*. The execution of state *ba1* (lines 18 - 31) is dependent on the grounded *agent* and *to* parameters of the *move* PDDL action (line 21). This state performs an actionlib call to MOVE BASE for a given set of parameters. For example, given specific parameters of the PDDL action (line 23) the goal description is generated with coordinates corresponding to *to* and sent on topic corresponding to the *agent* (lines 26 - 28).

This is only a snippet of the entire configuration for the *move* PDDL action that shows the high flexibility of the new action interfaces in describing complex and robust behaviours for different execution variants. In order to better comprehend this complexity, Figure 3 shows half of the FSM action interface for the *move* PDDL action. In this graph, states are represented as nodes and transitions between them are represented as edges. This picture also illustrates the failure-catching behaviour. For example, in the last displayed state *move_to_goal_reverse* the entire PDDL action is reversed to attempt to return the robot from an obstructed state to the initial position.

The implementation without the new AIM requires the creation of 13 different Action Interface ROS nodes, each with a specialized C++ class that contains the implementation of the function `concreteCallback`. In this function both the sequence of the required actions for a successful execution of the planned PDDL action, as well as all the recovery procedures must be hard coded. Furthermore, the complex actions of the Factory Scenario, (i.e., *move*, *load* or *manipulation_action_1*) interface with both standard ROS modules (i.e. MOVE BASE or MOVE IT) and also custom modules. Each such module requires an unique implementation within the action interface. Last, in order to consider the configurations for all possible grounded parameters, a configuration file must be created for each of these 13 PDDL actions. In comparison, using the AIM no code was written outside of the main configuration file.

```
1  actions:
2  - name: move
3    interface_type: fsm
4    states:
5    - name: start_state
6      interface_type: service
7      default_service_msg_type:
       ↪  action_failure_srvs/ActionFailureSimulator
8      default_service:
       ↪  /action_failure/action_failure_simulator
9      default_service_request:
10       probability: 0.0
11     default_service_response:
12       response: "success"
13     transitions:
14       succeeded:
15       - to_state: ba1
16       failed:
17       - to_state: error_state
18   - name: ba1
19     interface_type: actionlib
20     default_actionlib_msg_type:
       ↪  move_base_msgs/MoveBase
21     pddl_parameters: [agent, to]
22     parameter_values:
23     - values: [summit_xl_1, workbench11]
24       actionlib_topic: /summit_xl_1/move_base
25       actionlib_goal:
26         target_pose.header.frame_id:
           ↪  "summit_xl_1/map"
27         target_pose.pose.position.x: 0.6
28         target_pose.pose.position.y: -0.6
29     - values: [summit_xl_2, workbench12]
30       actionlib_topic: /summit_xl_2/move_base
31       actionlib_goal:
32         ...
```

Listing 3: Excerpt of the configuration file for the Factory Scenario.

7 Graphical User Interface

While the AIM offers a low-effort approach to configure new actions, and offers the ability to set up more complex FSM actions, it can still require a very long configuration file. In the configuration file for the 13 PDDL actions of the Factory Scenario, almost 130 states are defined and the description of each one can have up to 15 lines. To intuitively manage this high amount of information, a Graphical User Interface

[2]https://youtu.be/8Onh9SKF1yk

[3]https://sharework-project.eu/

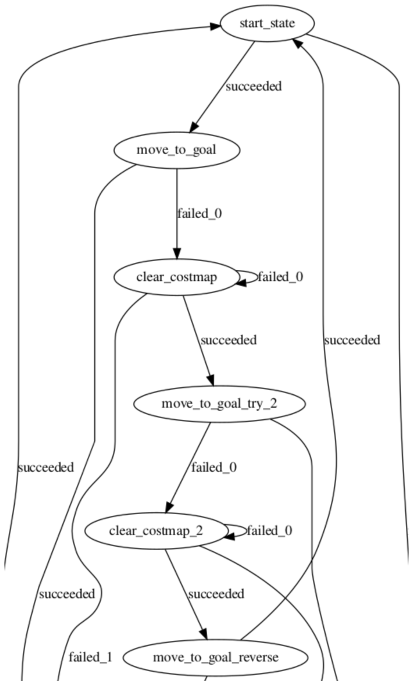


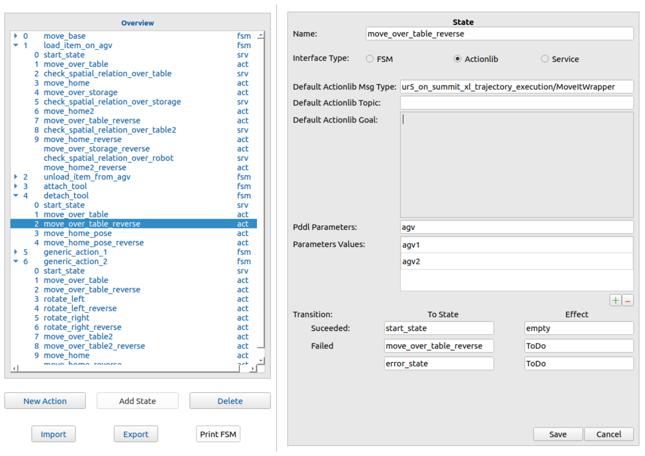
Figure 3: Part of the FSM Action Interface for the *move* action represented as a graph. After one failed attempt to move to goal, the navigation is retried. After two failed attempts the navigation map is cleared and the agent attempts to return to the initial position.

(GUI) was also developed. It sustains the generation and the maintenance of these configuration files and it is already integrated in the ROS ecosystem, as an *rqt* plug-in.

New action interfaces for the PDDL actions can be configured with the GUI. For each the user can select its type (*FSM*, *actionlib* or *service*). The *actionlib* and *service* interfaces can be directly configured by setting the *default* values and, if required, the values for the specific grounded parameters. The *FSM* interfaces require first to define its states. The states can be once again of type *FSM*, *actionlib* or *service*, can be individually configured and, for each of them, the transitions must be defined.

A screenshot of the GUI is depicted in Figure 4a. The main window of the GUI displays the overview of the action interfaces on the left. Those that are defined as *FSMs* may be expanded in a tree view. On the right side, the fields for the configurations are presented. Figure 4b presents the second window of the GUI, used to set the configuration values for the specific PDDL parameters.

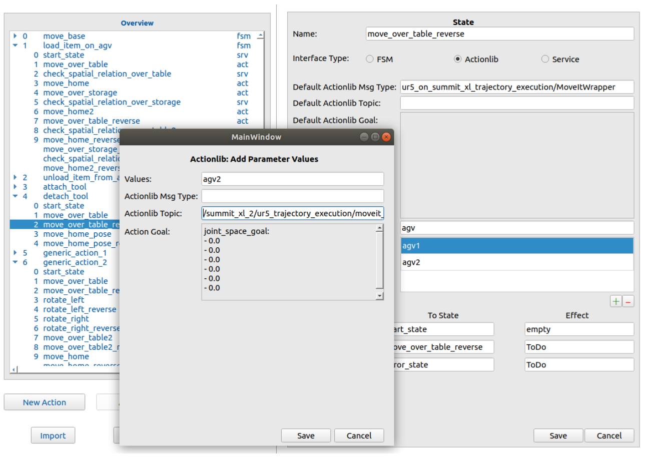
Further functionalities of the GUI are the *Import* and *Export* options, through which configuration files can be loaded into the GUI and new or modified ones can be saved on disk. In order to assist the development process of the configuration for *FSMs* action interfaces, their structures can be printed out directly from the GUI. For example, Figure 3 was generated with this functionality.[4]



```
Overview
0  move_base                              fsm
1  load_item_on_agv                       fsm
   0 start_state                          srv
   1 move_over_table                      act
   2 check_spatial_relation_over_table    srv
   3 move_home                            act
   4 move_over_storage                    act
   5 check_spatial_relation_over_storage  srv
   6 move_home2                           act
   7 move_over_table_reverse              act
   8 check_spatial_relation_over_table2   srv
   9 move_home_reverse                    act
     move_over_storage_reverse            act
     check_spatial_relation_over_robot    srv
     move_home2_reverse                   act
2  unload_item_from_agv                   fsm
3  attach_tool                            fsm
4  detach_tool                            fsm
   0 start_state                          srv
   1 move_over_table                      act
   2 move_over_table_reverse              act
   3 move_home_pose                       act
   4 move_home_pose_reverse               act
5  generic_action_1                       fsm
6  generic_action_2                       fsm
   0 start_state                          srv
   1 move_over_table                      act
   2 move_over_table_reverse              act
   3 rotate_left                          act
   4 rotate_left_reverse                  act
   5 rotate_right                         act
   6 rotate_right_reverse                 act
   7 move_over_table2                     act
   8 move_over_table2_reverse             act
   9 move_home                            act

State
Name:                        move_over_table_reverse
Interface Type:              ( ) FSM   (•) Actionlib   ( ) Service
Default Actionlib Msg Type:  ur5_on_summit_xl_trajectory_execution/MoveItWrapper
Default Actionlib Topic:
Default Actionlib Goal:
Pddl Parameters:             agv
Parameters Values:           agv1
                             agv2
Transition:          To State                   Effect
    Suceeded:        start_state                empty
    Failed           move_over_table_reverse    ToDo
                     error_state                ToDo
```

(a) Main window



```
MainWindow
Actionlib: Add Parameter Values
Values:               agv2
Actionlib Msg Type:
Actionlib Topic:      /summit_xl_2/ur5_trajectory_execution/moveit_
Action Goal:          joint_space_goal:
                      - 0.0
                      - 0.0
                      - 0.0
                      - 0.0
                      - 0.0
                      - 0.0
```

(b) Second window for the parameter-specific configuration

Figure 4: The two windows of the GUI through which the different action interfaces can be configured

Execution Monitor

The AIM provides a single node from which to monitor the progress of different action executions. A tool that monitors this execution progress was created in the ROS ecosystem as an *rqt* plug-in. It displays all the action interfaces, independent of their type, and highlights which of them are active. It also supports the asynchronous tracking of the execution of different planned PDDL actions of the same type. This feature is depicted in Figure 5, where the execution status of two *move* PDDL actions, carried out by two different actors from the Factory Scenario is shown.

8 Conclusion

In this paper we have introduced a new tool for the the mapping of planned PDDL actions to concrete execution strategies and integrated it into the ROSPlan framework. The original implementation requires the generation of many hard-coded C++ files and configuration files. The new implemen-

[4]A video demonstration of the user interface is available online https://youtu.be/7-nrpOe7hlg.

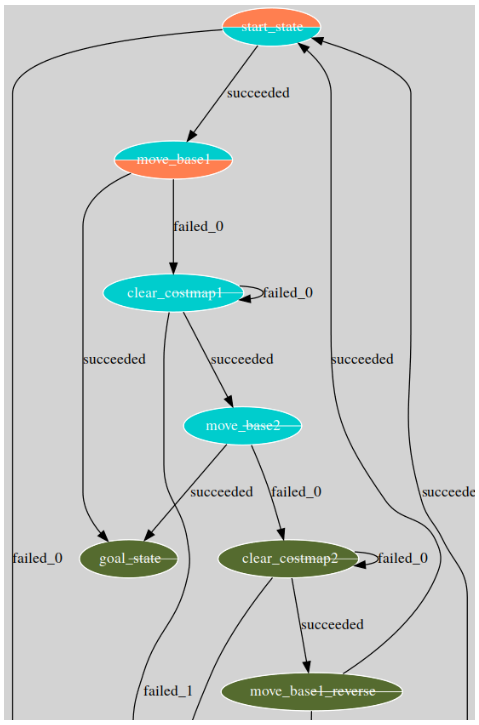


Figure 5: Monitoring of the PDDL actions execution through the configured action interfaces: the execution of one grounded PDDL action has reached the second state of the corresponding FSM action interface (orange), while the parallel execution of another PDDL action of the same type, but with different grounded parameters, has reached the fourth state of the FSM action interface (cyan).

tation is much more flexible and user-friendly, as it only requires the setup of a configuration file. In order to further improve this process, we developed different tools already integrated in the ROS ecosystem such as a GUI and Execution Monitor. The benefits the AIM have been illustrated on two scenarios of different complexity.

The aim of this work is to lower the barrier for inexperienced users of ROS and ROSPlan to build new scenarios that integrate automated planning and robotics. In future work we will continue to expand upon these tools to allow the user to define more complex interfaces, for example that account for durative constraints or non-deterministic effects.

Acknowledgements

The industrial scenario simulation used for the experimental demonstration was developed as part of the Sharework project, that is funded through the European Union's Horizon 2020 research and innovation programme under grant agreement No. 820807. This work has also been partially supported by the EPSRC grant THuMP (EP/R033722/1).

References

Bezrucav, S.-O.; and Corves, B. 2020. Improved AI Planning for Cooperating Teams of Humans and Robots. In *International Conference on Automated Planning and Scheduling workshop on Planning and Robotics (PlanROB)*.

Canal, G.; Cashmore, M.; Krivić, S.; Alenyà, G.; Magazzeni, D.; and Torras, C. 2019. Probabilistic Planning for Robotics with ROSPlan. In *Towards Autonomous Robotic Systems*, 236–250. Springer International Publishing. ISBN 978-3-030-23807-0. doi:10.1007/978-3-030-23807-0_20.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the Robot Operating System. In Brafman, R., ed., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, 333–341. Palo Alto, Calif.: AAAI Press. ISBN 9781577357315.

Cesta, A.; Orlandini, A.; and Umbrico, A. 2018. A Dynamic Task Planning System for Advanced Manufacturing Scenarios. In Finzi, A.; Karpas, E.; Nejat, G.; Orlandini, A.; and Srivastava, S., eds., *International Conference on Automated Planning and Scheduling workshop on Planning and Robotics (PlanROB)*, 65–74.

Darvish, K.; Bruno, B.; Simetti, E.; Mastrogiovanni, F.; and Casalino, G. 2018. Interleaved Online Task Planning, Simulation, Task Allocation and Motion Control for Flexible Human-Robot Cooperation. In *2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, 58–65. IEEE. ISBN 978-1-5386-7980-7. doi:10.1109/ROMAN.2018.8525644.

Harman, H.; Chintamani, K.; and Simoens, P. 2017. Architecture for incorporating Internet-of-Things sensors and actuators into robot task planning in dynamic environments. In *2017 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*, 13–18. IEEE. ISBN 978-1-5386-1342-9. doi:10.1109/IRIS.2017.8250091.

Johannsmeier, L.; and Haddadin, S. 2017. A Hierarchical Human-Robot Interaction-Planning Framework for Task Allocation in Collaborative Industrial Assembly Processes. *IEEE Robotics and Automation Letters* 2(1): 41–48. doi:10.1109/LRA.2016.2535907.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. URL https://homepages.inf.ed.ac.uk/mfourman/tools/propplan/pddl.pdf.

Moore, E. F. 1956. Gedanken-experiments on sequential machines. *Automata studies* 34: 129–153.

Munawar, A.; Magistris, G. D.; Pham, T.-H.; Kimura, D.; Tatsubori, M.; Moriyama, T.; Tachibana, R.; and Booch, G. 2018. MaestROB: A Robotics Framework for Integrated Orchestration of Low-Level Control and High-Level Reasoning. URL http://arxiv.org/pdf/1806.00802v1.

Quigley, M.; Gerkey, B.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; and Ng, A. 2009. ROS: an open-source Robot Operating System. In IEEE, ed., *International Conference on Robotics and Automation Workshop on Open Source Software*, volume 3.

ROS.org. 2021. ROS Concepts. URL http://wiki.ros.org/ROS/Concepts.

Sanelli, V.; Cashmore, M.; Magazzeni, D.; and Iocchi, L. 2017. Short-Term Human-Robot Interaction through Conditional Planning and Execution. In Barbulescu, L., ed., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling*. Palo Alto, California, USA: AAAI Press. ISBN 978-1577357896.

Viola, C. L.; Orlandini, A.; Umbrico, A.; and Cesta, A. 2019. ROS-TiPlEx: How to make experts in A.I. Planning and Robotics talk together and be happy. In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 1–6. doi:10.1109/RO-MAN46459.2019.8956417.

Appendix A. Action Interface Configuration

The BNF for the configuration file is given below in four parts: the top-level structure of the file, and then the details of the three action interface types (actionlib, service, and FSM).

Top-level Structure

```
<configuration> ::= actions:
                    <action>*
<action>        ::= - name: <var>
                      <service-action>|<actionlib-action>|<fsm-action>
<var>           ::= <string>|<rosparam>|<pddlparam>|<var>*
<rosparam>      ::= ($rosparam <var>) /* Use value from ROS param server */
<pddlparam>     ::= ($pddlparam <var>) /* Use value from PDDL parameter */
```

Actionlib Interface

The actionlib interface configuration is defined below, and an example is shown in Listing 2. Optionally, a default actionlib topic, message type, goal, and result can be set. If no default is set, then it is expected that these will be set per PDDL parameter instead. The optional parameter list *params* is a set of variables matching the parameter labels of the PDDL operator. Each parameter configuration then specifies the objects bound to those parameters, and a topic, message type, or goal that will override the default configuration.

```
<actionlib-action> ::= interface_type: actionlib
                       [default_actionlib_topic: <var>]
                       [default_actionlib_msg_type: <var>]
                       [default_actionlib_goal: <ros-msg>]
                       [default_actionlib_result: <ros-msg>]
                   ::= pddl_parameters: [<var>*] /* PDDL parameters */
                       parameter_values:
                       <al-param-config>*
<al-param-config>  ::= - values: [<var>*] /* PDDL objects */
                         [actionlib_topic: <var>]
                         [actionlib_msg_type: <var>]
                         [actionlib_goal: <ros-msg>]
                         [actionlib_result: <ros-msg>]
<ros-msg>          ::= <ros-msg-assign>|<ros-msg>*
<ros-msg-assign>   ::= <var>: <var>
```

Service Interface

The configuration for service action interfaces is described below. Similar to the actionlib interface, a default service, service type, request, and result can be set. These defaults can be overridden by specified action parameters.

```
<actionlib-action> ::= interface_type: service
                       [default_service: <var>]
                       [default_service_type: <var>]
                       [default_service_request: <ros-msg>]
                       [default_service_result: <ros-msg>]
                       pddl_parameters: [<var>*] /* PDDL parameters */
                       parameter_values:
                       <srv-param-config>*
<srv-param-config> ::= - values: [<var>*] /* PDDL objects */
                         [service: <var>]
                         [service_type: <var>]
                         [service_request: <ros-msg>]
                         [service_result: <ros-msg>]
```

FSM Interface

```
<fms-action>     ::= interface_type: fsm
                     states:
                     <fsm-state>*
<fsm-state>      ::= - <service-action>|<actionlib-action>|<fsm-action>
                       transitions:
                         succeeded:
                         - to-state: <fsm-transition>
                         failed:
                         - to-state: <fsm-transition>
<fsm-transition> ::= <var>|start_state|goal_state|error_state
```