# A Closed-Loop Framework-Independent Bridge from AIPlan4EU's Unified Planning Platform to Embedded Systems

**S. Hastham Sathiya Satchi Sadanandam*[1], S. Stock*[2], A. Sung*[2], F. Ingrand[1], O. Lima[2], M. Vinci[2], J. Hertzberg[2,3]**

[1]LAAS-CNRS, University of Toulouse, France.
[2]German Research Center for Artificial Intelligence DFKI.
[3]University of Osnabrück, Germany.
selvakumar.h-s@laas.fr, sebastian.stock@dfki.de, alexander.sung@dfki.de,
felix@laas.fr, oscar.lima@dfki.de, marc.vinci@dfki.de, joachim.hertzberg@dfki.de

**Abstract**

The main goal of the AIPlan4EU project is to give easy and unified access to a large and diversified number of planning technologies and approaches. Yet, to make beneficial use of the produced plans, the project also develops and deploys Technology Specific Bridges (TSB) to connect and close the loop with real-world applications. In this paper we present one of these bridges which specifically targets robotic applications and more generally embedded systems. In these systems, planning and plan execution lead to commands on effectors in the real world, while sensors, intrinsic and extrinsic, report on the state of the world, which is then fed back to monitor the plan execution, with replanning or plan repair when needed. To this effect, we have developed a Python library that simplifies the creation of planning domains for existing applications and allows to execute and monitor resulting plans. Despite being developed for robotic applications, the library is not dependent on a specific robot framework, nor any middleware, and can also be used for non-robotic domains. We demonstrate this TSB on two robotic experiments: a ROS (Quigley et al. 2009) based service robot and a GenoM (Dal Zilio et al. 2023) based multi drones application.

## 1 Introduction

Planning technologies can be useful for applications in many different areas, and a broad range of mature domain-independent planners and formalisms are now available. However, it often takes a big effort to use those planning systems in actual applications. The AIPlan4EU[1] project aims to increase the use of planning technologies in various real-world applications ranging from agriculture and manufacturing to robotics by facilitating access to a broad range of planning technologies. As one step in this direction, AIPlan4EU is developing the Unified Planning[2] library (abbreviated as UP). UP is a Python3 library that gives access to multiple planning approaches and engines. Not only can planning domains and problems be created using existing languages such as PDDL (Ghallab et al. 1998), ANML (Smith, Frank, and Cushing 2008) or HDDL (Höller et al. 2020), but one can also write domains and problems directly in Python and have access to planners as well as to additional functionalities like grounding or converting plans.

Despite UP's expressive planner interface, the problem of connecting it or building a bridge to the actual applications can still be complex. In some applications it can be straightforward: the planning domain and problem are passed to the planner which solves it and produces a plan or a schedule to be executed, e.g., by a human. But in other domains, the bridge may contain glue code that is specific to a particular application. This includes for instance collecting and transforming the relevant application data for generating the planning problem by connecting to an external knowledge base or computing the state of individual fluents based on non-symbolic data. For example, for a mobile robot we might need to discretise the robot's arm pose into a symbolic pose. Furthermore, to use the resulting plans in the applications, the plan's action instances need to be mapped back to their executable counterparts on the application side. Another important aspect to address is the plan's execution itself. Actions need to be dispatched at the appropriate time, their execution monitored, and appropriate measures taken for dealing with errors or failures during execution. To facilitate these more complex connections, the AIPlan4EU project develops and proposes several Technology Specific Bridges (TSB) between application domains and Unified Planning.

In this paper we present one such TSB, named Embedded Systems Bridge (ESB), a Python3 library, aimed at robotic applications. The choice of Python as the programming language is mostly motivated by the fact that the UP is written in Python (even if most planning engines are not) and we do not envision the need for some CPU intensive computation. It is available on GitHub[3] as open-source under the Apache-2.0 license. It contains generic functionalities for connecting applications to the UP library and for executing the plan actions on physical systems.

## 2 Related Work

One of the first AI planning systems, STRIPS (Fikes and Nilsson 1972), when running on the Shakey robot, was de-

*These authors contributed equally.
[1]https://www.aiplan4eu-project.eu/
[2]https://github.com/aiplan4eu/unified-planning
[3]https://github.com/aiplan4eu/embedded-systems-bridge

ployed along a plan execution component: Planex (Fikes 1971). Thus, Shakey was able to navigate in various rooms, push boxes and take pictures. Yet, plans produced by STRIPS were executed without any action refinement, as Planex directly mapped plan steps into robot commands. For robotic systems, and more generally for embedded systems, planning only makes sense when jointly deployed with a plan execution component.

Yet, plan execution covers many problems that need to be addressed: action refinement, action monitoring, temporal monitoring, temporal dispatching, non-nominal execution and failure handling, command and sensing interfaces, and more. In this rich field, we focus on systems and approaches that have made a point in bridging generic planning systems to robotic systems.

ROS[4] is a de facto standard in the robotic community, and as such is an ideal framework to bridge planning systems to robotic systems. The most recent version, ROS 2, has some design advantages over ROS 1, yet it will take years to transition the large volume of legacy ROS 1 packages to ROS 2. Nevertheless, two ROS-based generic bridges must be presented:

**The ROSPlan framework.** ROSPlan (Cashmore et al. 2015) offers a bridge between various task planning approaches and robotics via ROS 1. While its main support is for temporal PDDL 2.1 with the popf planner (Coles et al. 2010), it also offers experimental support for probabilistic planning (Canal et al. 2019) and time-constrained hierarchical task networks with resources via the CHIMP planner (Stock et al. 2015). ROSPlan splits up the different elements of planning and plan execution into multiple components in the form of ROS nodes, e.g., for domain and problem representation, plan generation, post-processing, execution. The communication between those components is handled via ROS's topic publish/subscribe mechanism (string message in this case).

At the core of ROSPlan's execution component is the Esterel plan dispatch algorithm, which relaxes a temporal PDDL 2.1 planner output to re-assemble it into an execution graph, whose edges represent ordering constraints of three types: start-end of an action, interference edges and causal links. Once the graph is formed, the execution is similar to a Petri net in the sense that only after all edges are activated, the node is activated, which means that the action is ready to be executed. This allows a flexible execution under time constraints, where we don't care that any hard deadlines imposed by the planner are exactly met, but rather dispatch the actions as soon as all appropriate conditions are met.

One of the caveats of ROSPlan is that it is tightly coupled to ROS 1, which is scheduled to lose support in 2025 in favor of the new ROS 2 release.

**PlanSys2: ROS 2 Planning System** Inspired by ROSPlan, PlanSys2 (Martín et al. 2021) is a ROS 2-compatible planning and execution system optimized for Behavior Tree (BT) execution of parallel actions. Currently, temporal PDDL planners popf (Coles et al. 2010) and Fast Downward (Helmert 2006) are supported.

At the core of PlanSys2, an algorithm is used to convert the output of a temporal PDDL planner into a behavior tree (Martín Rico et al. 2021) from an execution graph previously created using causal links.

For each action, the generated BT steps are wait for action, wait at start request, apply at start effect, check overall preconditions, execute action, check at end preconditions, and apply at end effects.

PlanSys2 comes with developer tools such as a terminal that allows users to set instances, facts, goals and get domain, problem, and plans. ROSPlan does a similar job in this regard with its GUI interface, allowing an easier way to set instances, facts, and goals.

Compared to ROSPlan, PlanSys2 is a priori better suited for multi-robot systems, as ROS 2 runs without a roscore, which is a potential single point of failure on distributed ROS 1 experiments. Note that PlanSys2 is one of the TSB ports funded within the AIPlan4EU open calls. As such it will be among the bridges available on the platform.

Beyond (and before) ROS-based systems, other architectures and approaches have been proposed to bridge planning and execution in robotics. They often propose a software organization along a few layers (Bonasso et al. 1997) with different temporal requirements and abstraction levels. We often find a so-called functional layer containing the low-level sensors–effectors–processing modules, as well as a decisional layer containing some of the deliberation functions (e.g., planning, execution, monitoring, etc). Some rely on specific tools to implement the various components along the different layers. For example, the LAAS architecture (Ingrand et al. 2007) proposes GenoM to implement all the modules of its functional level, OpenPRS, and IxTeT for the deliberative level. (Doherty, Kvarnström, and Heintz 2009) proposes a temporal planner (TAL Planner) and a stream-based architecture (DyKnow) for perception, anchoring, and plan recognition. Similarly, CLARATy (Nesnas et al. 2003) provides C++ classes for the basic functional components and TDL or ASPEN/Casper for planning and acting. PLEXIL, a language for the execution of plans, illustrates a representation where the user specifies nodes as computational abstractions (Verma et al. 2005). A node can monitor events, execute commands, or assign values to variables; it may refer hierarchically to a list of lower-level nodes. PLEXIL has been developed for space applications and used with several planners such as CASPER. RMPL (for Reactive Model-based Programming Language) (Ingham, Ragno, and Williams 2001) proposes a common representation for planning, acting and monitoring. It combines a system model with a control model. The former uses hierarchical constraint-based automata to specify nominal as well as failure state transitions, together with their constraints. The latter uses reactive programming constructs (including primitives to address constraint-based monitoring, e.g., as in Esterel (Coste-Maniere, Espiau, and Rutten 1992)). Moreover, RMPL programs are transformed into Temporal Plan Networks (TPN) (Kim, Williams, and Abramson 2001), an ex-

[4]https://www.ros.org/

tension of STN (Simple Temporal Network) with symbolic constraints and decision nodes. The result of each RMPL program is a partial temporal plan which is analyzed by removing flaws and transformed for execution taking into account online temporal flexibility. Within the RoboCup Logistics League, some teams deployed OpenPRS as an acting language but also the CLIPS Executive (CX) (Hofmann et al. 2021) based on the CLISP rule-based production system. CX is an integrated goal reasoning system that provides an explicit goal representation, implements a goal lifecycle, and structures goals in trees.

Teleo-reactive architectures (Finzi, Ingrand, and Muscettola 2004) are more recent. They propose an integrated planning–acting paradigm, which is implemented at different levels from deliberative down to pure reactive, using different planning–acting horizons and time quantum. Each planner–actor is responsible for ensuring the consistency of a constraint network on temporal and state variables. Each shares a subset of these variables with other planners–actors to provide communication between them. Interestingly, this approach was developed and proposed by Willow Garage (McGann et al. 2008) as part of the original ROS environment to plan and execute missions for PR2 robots. Yet, the performance of these systems is often a problem when it comes to planner–actors that need to solve large CSPs, in a time quantum below one second.

Our proposed approach, which is still work in progress, has the following specificities:

- It is strongly linked to the UP and as such, it will try to support:
  - the various plan formats produced by UP,
  - the various repair/replan mechanisms proposed by UP.
- It is middleware agnostic and will try to support ROS and non-ROS robotic frameworks.

Some of these features are already partially supported in the current implementation, which will evolve as the project moves on.

## 3 ESB: A bridge between embedded systems and UP

The standard approach to apply task planning is to describe the planning problem in a planning domain language, then run a solver to retrieve a plan. In the domain description language, representations of objects, states, and goals are declared. The ESB aims at facilitating this declaration step by providing an interface to which existing application domain declarations in Python can be provided. By using the referenced objects which are provided in the calls to the bridge's interface, it creates most of the problem description in the UP domain automatically.

Figure 1 presents the ESB architecture that combines the UP and the application. The architecture consists of three components: application definition, Unified Planning and the bridge itself. The application definition must be a Python interface wrapped around the application to be used with the

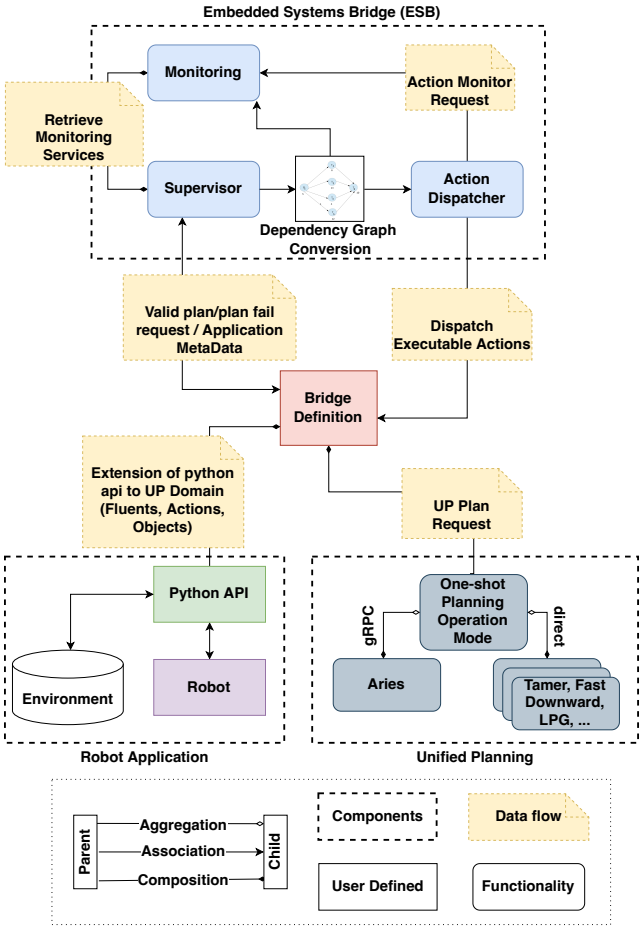


Figure 1: Architecture of the Embedded Systems Bridge.

bridge. Unified Planning (UP) is a planning framework developed in Python that is compiled of various planning technologies. The user can choose the desired planning engine for their application or let UP handle on its own based on the provided problem definition. ESB attempts to extend the UP to the application domain by connecting the gap with orchestration. At the core of the bridge are the dependency graphs which are used for the orchestration process. The use of dependency graphs in task execution promotes better handling of task dependencies between each action, better error handling and recovery, and better monitoring. The bridge automatically maps executable functions from the application definition to the action instances returned by the planner. This makes executing a resulting plan straightforward and will be demonstrated in the following section about plan execution and monitoring in more detail.

The ESB interface supports many application domain formats, as long as they use Python with type annotations. A short example shall demonstrate its ease of use. See Listing 5 for the full example code. (The line numbers refer to the ones in that listing.)

With the given class declarations of `Pose` and `Robot`, the user can create UP type representations for them using the

instruction:

```
33 bridge.create_types([Robot, Pose])
```

After that, whenever these classes are referred to at the bridge interface, it will automatically use the corresponding UP representations when needed.

With the classes being declared, object instances are declared similarly by, e.g.,

```
34 up_robot = bridge.create_object("robot",
       robot)
```

Since `robot` is an instance of `Robot`, the bridge knows that its UP representation `up_robot` must be an instance of the UP type corresponding to `Robot`.

Now one can pass a function representing a fluent or an action to the ESB using

```
36 bridge.create_fluent_from_function(
       robot_at)
```

or

```
37 bridge.create_action_from_function(Robot
       .move)
```

respectively. The latter instruction automatically creates a UP representation of a `move` action with its parameters, which will be automatically included in subsequent UP problem descriptions unless specified otherwise. The former instruction provides a so-called fluent function, which calculates the current value of the fluent for concrete parameters. The bridge calls this fluent function whenever it is required to initialize the fluent values of the planning problem. The user typically will not need to call this fluent function herself anymore but instead execute

```
44 bridge.set_initial_values(problem)
```

to initialize all fluents of a `problem` using their fluent functions. Once the goals are defined for the problem, one can request a valid plan from UP either using a specific planning engine or through the problem kind. In the code example, UP decides the planning engine based on the type of problem provided, which is a functionality provided by UP itself.

```
46 plan = bridge.solve(problem)
```

The UP library can generate sequential, partial-order and time-triggered plans. Sequential plans consist of a list of action instances that need to be executed one after another. An action instance contains the action definition itself as well as a list of actual parameters. A partial-order plan is represented by UP as a directed acyclic graph in the form of an adjacency list. Time-triggered plans are represented by UP as lists of tuples where each action instance also has an assigned start time and an optional duration. An example of a time-triggered plan is presented in Listing 4 in Section 5. UP also has a representation of plans in the form of a Simple Temporal Network (STN) (Dechter, Meiri, and Pearl 1991). STNs are currently not supported by the ESB, but we aim to do so in the future. In section 5, we will show more detailed uses of the bridge in our applications.

In section 5, we present exchangeable application blocks for two robotic systems using a mobile robot (Fig. 4) and a drone (Fig. 6). The Unified Planning block could be considered the core component of planning which provides different planning engines. Finally, the ESB component contains the bridge definition, plan execution and monitoring.

## 4 Plan Execution and Monitoring

Plan execution is a critical part of many applications, and it includes multiple aspects and issues that need to be taken into account. While the requirements and results of plan generation are relatively well defined, the overall execution of the plan is more dependent on the application domain and its constraints. Therefore, the ESB aims to make plan execution flexible and adaptable to the needs of the specific application. Many planning approaches, especially classical planning, make simplifying assumptions that may not hold true in real application environments, leading to discrepancies between plan and reality that we have to deal with at execution time. Therefore, in addition to dispatching the plan's actions, plan execution also needs to provide means for monitoring the execution and for resolving deviations from the plan or action failures. The ESB aims at providing functionalities for all of those three aspects.

UP can provide plans in different formats. It has representations for sequential plans (sequence of actions), time-triggered plans (a list of time-stamped durative/instantaneous actions), partial-order plans in the form of Directed Acyclic Graphs (DAG), hierarchical plans that additionally provide the used decompositions, plans in form of a Simple Temporal Network (STN) as well as contingent plans. The generated plan format depends on the problem description and the internally used planner. UP can also transform sequential plans into partial-order plans by considering the causal links which enable parallel execution.

The way in which actions are dispatched in order depends on the kind of plan that is to be executed. Currently, the ESB can execute sequential, partial-order and time-triggered plans. Hierarchical plans can be dispatched by using its internal flat plan representation, but the hierarchy is not considered, yet. We aim to add support for the execution of STNs in the future.

For dispatching UP plans, the ESB converts the plans into dependency graphs. The semantics of dependency graphs are directly related to different plan types produced by UP. The execution of dependency graphs with sequential plan type only needs to iterate through a plan and dispatch the next action once the previous action is finished; partial-order plans can allow the parallel execution of multiple actions and time-triggered plans will check for actions being performed within the duration specified in the plan in a flexible way.

The contents of the graph nodes are based on the type of action event from the UP plan. The common event types that are currently supported in the ESB are Instantaneous Action Event and Durative Action Event. A graph node holds its information based on its type with an Instantaneous Action

holding the node name, node ID, executable action representation and action parameters, while a Durative Action additionally contains the start time and duration of the action.

Once the plan is acquired from UP, we translate the mapping between the UP representation of fluents or actions and parameters and its equivalent functions into such a dependency graph by

```
graph = bridge.get_executable_graph(
    plan)
```

In our example, the action `move` with the two parameters `from_pose` and `to_pose` are mapped to a graph node with concrete poses in the application.

The created dependency graph is given to the plan dispatcher which executes the actions and monitors the states. As the graph represents the causal links between actions with directed edges the dispatcher only executes an action if all predecessor actions are finished. The execution can be started by using the provided `PlanDispatcher` by

```
dispatcher = PlanDispatcher()
dispatcher.execute_plan(graph)
```

This steps through the dependency graph and dispatches an action once it's predecessor actions are successfully completed. For dispatching it uses a callback that can either be provided to the dispatcher, or a default dispatching function is used that calls the respective execution function that corresponds to the action.

The described way of dispatching the actions automatically in a while-loop inside the dispatcher reflects the current implementation of the bridge. But we also consider to change this in the future, such that alternatively, the user has to actively query the dispatcher for new actions that can be dispatched. This would give the user more control of the execution process which could have advantages for some application domains.

## Dealing with execution failures

In many real-world applications, unexpected events and execution failures may occur, which makes it important to monitor the plan. If an execution error occurs, the system should be able to detect this and take appropriate measures that allow continuing to pursue its goal. Execution failures can be dealt with in different ways, depending on the type of failure and the type of action. One category of failure results from a failure in the execution of the action itself. A robot might fail to grasp an object, for example, because the motion planner does not find a valid trajectory for the arm that lets it grasp the object. In this case, one could try to execute the action again. For some actions, it could even help to try them multiple times. For other execution failures, retrying might not be suitable, for example, if the action is consuming resources or has to adhere to some time constraints. Furthermore, it can be useful to consider the cause of the failure if available.

Another option for dealing with failures is to replan from the current situation or to try to repair the plan. Replanning has the advantage that it only requires updating the problem based on the new state and is possible with all planners. Plan-repair, on the other hand, also provides the planner with additional information about the executed actions and the initial plan. This feature is not provided by all planners, but it could result in a lower run-time that is needed to repair the plan. Furthermore, it is often more likely that the repaired plan is similar to the original one than if it was created from scratch. In the current implementation, the dispatcher deals with failed actions by replanning one time based on the updated state. If this does not lead to a plan or the same action fails again, it aborts the execution. We aim to include plan-repair that is partly available in UP in future work.

A second type of execution failure manifests itself in unfulfilled preconditions of the action to be executed. These failures often result from unexpected changes in the environment, e.g., as a result of the actions of other actors, leading to unfulfilled preconditions of an action. For example, for the `grasp`-action the object might not be at the position where it was expected. In this case, there is no point in carrying out the action. Such a kind of failure could be dealt with by replanning or plan-repair as well. The ESB currently provides means for checking preconditions and effects of actions. This is done currently in the previously mentioned dispatching function that is used by default in the dispatcher. We aim to extend this in the future to also monitor the full plan, e.g., requirements and time constraints of later actions.

The appropriate way to deal with such execution failures heavily depends on the use-case and application. Thus, the bridge has been developed to make this setting flexible and configurable. Currently, the bridge provides an interface to handle execution failures due to failed preconditions which can trigger replanning. For redefining the problem based on the new state of the system or any unachieved goals by the agent, the concept of rules-based problem redefinition is introduced. The rules primarily depend on condition-checking functions which in our case can be the functions created in the application as a fluent definition or any external function which can read the state of the system. These condition-checking functions are used for evaluating the current state thereby allowing the planning problem to be updated in accordance with user-defined constraints.

Listing 1: Replan rule for locating the object if the `grasp`-action fails

```
def replan_rule(pb):
    # pb: UP Problem
    # is_obj_located: UP Fluent
    # checked_location: UP Object
    # search_obj: UP Action
    # check_obj_location: condition-
        checking function
    if not check_obj_location:
        pb.initial_states -= [
            is_obj_located(1)] # 1: True
        pb.objects += [checked_location]
        pb.goals += [is_obj_located(1)]
        pb.actions += [search_obj]
    return pb
```

For example, for the `grasp`-action mentioned previously, if the desired object is not at the position, a simple rule can be defined to modify the planning problem to search for

the object in the possible locations and remove the already searched locations. A pseudo-code of the mentioned rule can be represented as in listing 1. The rules can also be defined for unachieved goals allowing the system to retry until the desired goal is achieved. Thus, the problem can be redefined to achieve the goals when certain expected actions fail in the system by applying the specific rule(s) for the action.

## 5 Applications

Our library is independent of specific robotic frameworks and could be used for non-robotic applications as well. Nevertheless, it was designed with robotics as the main use-case. Although the library has no direct dependencies on robotics frameworks, it can be easily integrated into robotic applications. We demonstrate this for two different kinds of robots and frameworks to emphasize that the ESB is usable for a broad range of applications, as long as they shall make use of task planning and can access the ESB's Python interfaces.

### Mobile Robot using ROS

Figure 2: The mobile robot example scenario.

The first experiment involves a mobile robot called Mobipick, which is shown in Figure 2. It consists of a MiR base, a UR5 arm and a Robotiq gripper attached to the arm. Mobipick runs ROS as its middleware and its capabilities and actions are implemented in ROS or use the ROS ecosystem. The mobile robot can sense, navigate, and manipulate its environment, to pick up and bring objects to specified locations. It can navigate the mapped world autonomously avoiding dynamic obstacles using the laser scanners of the mobile base. Additionally, it is able to classify objects and estimate their 3D poses using a camera attached to the robot's end effector, to then pick them up and bring and place them at a given location. The objects that the robot can identify and grasp are inspired by a workshop environment, they include multimeters, screwdrivers, relays, power drills and boxes, as presented in Figure 3. Boxes can be used to transport multiple objects at once by inserting them, with the exception of the too heavy power drill, which is too heavy.

Figure 3: The objects the robot can detect and manipulate, left to right: multimeter, screwdriver, relay, power drill, box.

Figure 2 shows the testing environment that we use. It consists of three tables on which objects of the previously mentioned types are positioned. For test and experiment purposes, we also have a Gazebo simulation with the same environment layout. While it will be difficult for the reader to reproduce our real setup, we do provide this simulation environment along with the entire software stack based on our real robot as an open source repository called *mobipick_labs* [5] as a testbed for AI planning and acting algorithms.

We show our approach in two demonstration scenarios in the given environment layout. The robot has to achieve two kinds of goals in different runs. The goal of the first example involves collecting a power drill and handing it over to a human worker. The goal of the second example is to collect specified objects, insert them into a box and transport the box onto a specified target table. The objects are scattered on the tables around the robot, and initially the robot does not know the locations of the objects it needs to collect. To accomplish the goal, the robot searches the tables for the desired objects, interrupts the search if it finds one and replans to pick it up and to bring it to the target table.

The Mobpick-specific part of our architecture is shown in Figure 4. The upper part of the architecture is similar to the general architecture in Figure 1. The bridge connects the ROS- or robot-specific actions and representations with the planners and representations provided through the Unified Planning library.

The robot's actions are implemented in ROS. Besides the well-known actions for robot base movement (`move_base`) and robot arm movement (`moveit`), we implemented sensing actions like perceiving the objects on the table next to the robot, and action sequences for, e.g., picking up an object. The search action is also realized as a composite action and requests a sub-plan for moving to and looking at individual tables not recently seen. All these higher-level actions can be interfaced through Python scripts and are bundled in the Robot API. More details about Mobpick's action implementations and the Robot API are presented in (Lima et al. 2023). The Python interface to the robot's capabilities was already available for easily accessing the robot when we started to connect the Mobipick to Unified Planning via the ESB to formulate and solve the planning problem. We use the aforementioned function `create_action_from_function` from the ESB to create UP representations from those robot capabilities provided in the Robot API and enrich those UP action representations afterwards with preconditions and effects.

[5] https://github.com/DFKI-NI/mobipick_labs

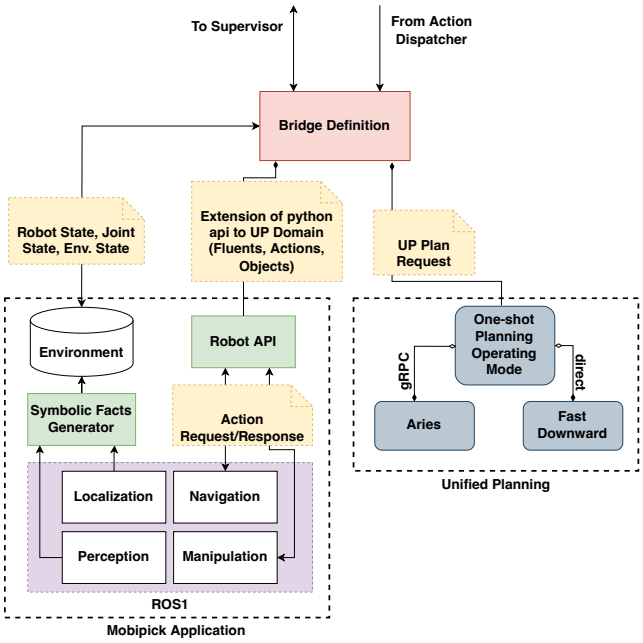


Figure 4: Architecture of the Mobipick application.

For describing the fluents of the planning problem, we implemented simple functions that return what the robot remembers from its environment. When the robot places the box onto the target table, it will remember this until a contradicting observation is made. If, for example, a person would move the box away to another table, Mobipick could detect it there later (objects are unique in this scenario) or observe the target table without the box, thus updating the box's location to unknown. New information that the robot perceives via object detection or comes from its localization module are transformed into symbolic facts through a symbolic fact generation module. For example, object detection can perceive the type of object and its pose if the object is in the camera's view. Together with prior knowledge about the static placement and size of tables the symbolic fact generator creates the information of the object being `On` a certain table. We employ the ESB to use those functions for representing the robot's and the environment's state in the planning problem as fluents and objects.

We solve the problem via UP using its `OneshotPlanner` interface and providing the additional requirement to find an optimal plan in terms of number of actions. For our current configuration UP uses the planner Fast Downward (Helmert 2006) for our problems. An example plan for the goal of handing-over a powerdrill to a worker is shown in Listing 2. Here, Mobipick first moves its arm into a pose that is suitable for driving without the risk of collision with obstacles. The arm configuration is discretized by the symbolic fact generator into a set of symbolic poses. If that pose does not match any of those poses, the value `unknown` is used to represent the arm's position. Mobipick then drives to a pose near the table and picks up the powerdrill. Afterwards, it again moves its arm to a safe pose and drives to the worker where it hands over the object before finally driving to a goal home pose.

Listing 2: Plan for getting and handing-over a powerdrill.

```
move_arm(mobipick, unknown, home)
move_base(mobipick, base_table_3_pose,
    base_table_2_pose)
pick_power_drill(mobipick)
move_arm(mobipick, unknown, transport)
move_base_with_item(mobipick,
    power_drill, base_table_2_pose,
    base_handover_pose)
hand_over(mobipick)
move_arm(mobipick, handover, home)
move_base(mobipick, base_handover_pose,
    base_home_pose)
```

Listing 3: Initial plan for placing a multimeter into a box and transporting the box onto `table2`. The locations of the box and multimeter are unkown at the beginning.

```
search_tool(mobipick, multimeter)
pick_item(mobipick, tool_search_pose,
    tool_search_location, multimeter)
search_box(mobipick)
store_item(mobipick, box_search_pose,
    box_search_location, multimeter)
pick_item(mobipick, box_search_pose,
    box_search_location, box)
move_base_with_item(mobipick, box,
    box_search_pose, base_table_2_pose)
place_item(mobipick, base_table_2_pose,
    table_2, box)
```

On the real robot the `handover` action reacts to a slight pressure on the gripper given by the user. When testing this scenario in Gazebo, the successful execution of the handover is done via a ROS service.

After receiving the resulting sequential plan from UP we execute it by dispatching the actions one after the other via the ESB. In the current state we monitor the plan by checking the actions preconditions directly before dispatching them. At the moment this is done by the `SequentialPlanMonitor` in the ESB which internally uses a plan simulator provided by UP. However, we aim to restructure this approach for monitoring in the future.

In the second scenario, the robot must collect specified objects, insert them into a box and transport the box onto a specified target table. Listing 3 shows an example of an initial plan for this goal. Note that the positions of the objects are initially unknown to the robot. Therefore, the plan contains placeholder actions, e.g., `search_tool`, and symbols, e.g., `tool_search_location` to search for those items first. Those search actions are handled differently in the execution loop in the sense that they are not directly executed on the robot. Instead, they create a sub-planning problem that is again solved with UP and executed in its own loop before returning to the initial plan. Those sub-plans consist of driv-

ing to the tables one after another and looking at them. If the object is found during that subplan, the internal execution loop stops and returns to the original plan. Furthermore, if an action fails, we abort the plan and replan once. We implemented sensing and replanning this way to explicitly allow user interference that causes planned actions to fail. Proper error handling and adaptation to the new situation are expected from the robot in such cases. However, as mentioned previously, we aim to generalize such means for dealing with errors inside the ESB in future work.

## Drones using GenoM

The second experiment involves one or more drones, each with a camera, deployed to survey an area. First, the UP produces a plan for one drone to quickly survey the area for objects of interest (the onboard camera is used to locate blobs of a specified color in our example) and makes a list of the approximate locations. Then the UP extends the plan to visit each location and identify each object (reading and decoding an ARUCO tag). According to the identification, various actions can be taken to properly handle the objects. While each drone performs its part of the plan, unexpected events such as critically low battery levels, or communication problems may lead to plan failure (requiring to abort the current mission) and plan repair (amending the plan to consider new goals such as charging or changing the battery). The produced plans by the UP (the ARIES planning engine) are hierarchical, for example, the **Survey** action is broken down into several `move` actions, while in parallel, the camera looks for the objects of interest and localize them (using the drone position/attitude and the camera parameters). The plan includes parallel action execution and temporal information (start time and action duration).

The drones we fly in this experiment are developed at LAAS. Figure 5 shows the software components running onboard each of them. All these components are specified and deployed using GenoM and can be synthesized for the ROS-Com middleware or the PocoLibs middleware (we favor the latter for its shared memory communication model and its overall memory footprint). The reader can check (Dal Zilio et al. 2023) for a more detailed description of the GenoM framework; the drone software components and how they work together. Note that the exact same GenoM components run on the real drones (indoor using motion capture for localisation, or outdoor with GPS RTK) but also in a Gazebo simulation. GenoM offers a TCL (tcl-genomix) or a Python (py-genomix) interface to supervise the GenoM components of the experiment. As a result, the ESB can act on and perceive the environment via this asynchronous interface. E.g., the **goto(x,y,z)** service of the MANEUVER component flies the drone to the specified location, and reading the state port of the POM component gives the position of the drone.

In simulation, the experiment uses Gazebo. Upon starting the experiment, we spawn a given number of colored plates with ARUCO markers (Figure 5) in random locations. The drones' goal is to find all the plates in the environment and inspect each of them. The functional components of GenoM provide the basic actions through py-genomix. For example, the `survey` action is a combination of multiple GenoM asyn-

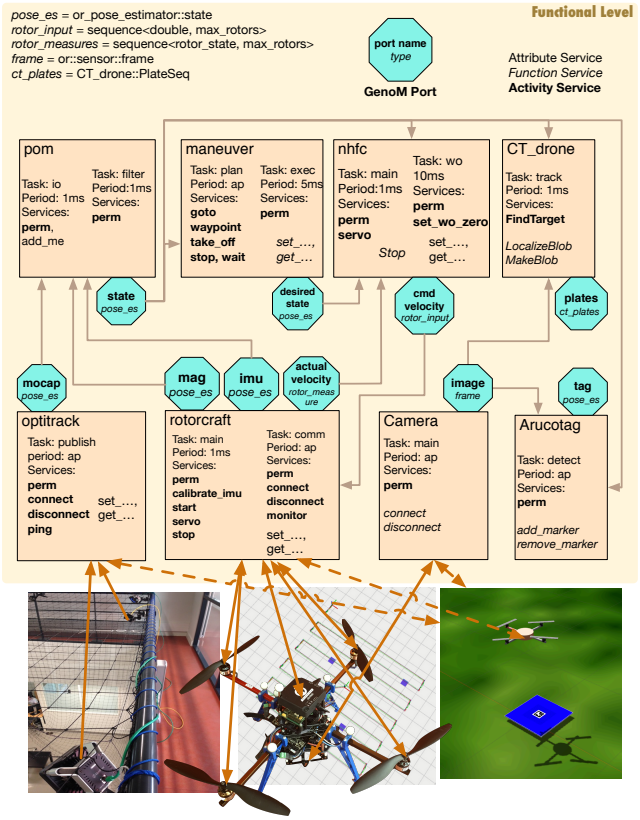


Figure 5: Software architecture of a drone.

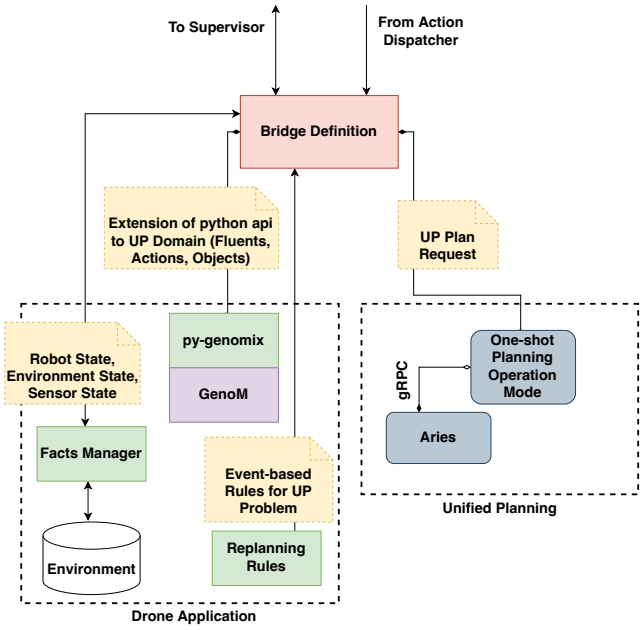


Figure 6: Architecture of the drone application.

Listing 4: Time-triggered plan example for the drones.

```
# r*, area, station*, charging_dock*, l*
    - objects
# survey, send_info,
   acquire_plates_order, inspect_plates
   - actions
# Action instance is of format 'start,
   action,duration'
0.0,survey(r1, area, station1),100.0
100.1,send_info(r2),1.0
101.1,acquire_plates_order,1.0
102.1,move(r2, station2, l2),20.0
102.1,move(r1, station1, l4),20.0
122.2,inspect_plate(r1, l4),2.0
122.2,inspect_plate(r2, l2),2.0
124.3,move(r2, l2, l3),20.0
124.4,move(r1, l4, charging_dock1),20.0
144.4,inspect_plate(r2, l3),2.0
146.5,move(r2, l3, l1),20.0
166.6,inspect_plate(r2, l1),2.0
168.7,move(r2, l1, charging_dock2),20.0
```

chronous services like **waypoint** of the MANEUVER component, and **FindTarget** of the CT_DRONE component allowing the drone to survey the area and look for colored plates. Likewise, the `move` action is translated in **goto(x,y,z)** from MANEUVER and the `inspect_plate` action also uses **goto(x,y,z)** and **detect** of the ARUCOTAG component. Similarly, the GenoM functional components access the robot and environment states which are automatically updating a database. Thus, to retrieve planning fluents, some Python functions query information from this database to evaluate the current state of the experiment and set the fluents value. The architecture of the drone experiment is presented in Figure 6 (with respect to the generic architecture Figure 1).

Listing 4 presents a plan for the case when the planning system already knows the expected number of plates to be found within a certain area for inspection. In this case, the planning problem is set up to let the drones find the plates location, i.e. to survey the environment by looking for them. The corresponding dependency graph for this plan is presented Figure 7. In another case, only the number of plates to be found is available but the region of interest for survey is not completely known. Here the drones plan starts with a default region to survey and until the proper number of colored plates is found the surveyed region is enlarged. This means the plan may need to be redefined for one drone to keep surveying new areas to find the missing plates, while the another drone inspect the already found plates.

To allow different cases to be handled for the experiment, we introduce some rules for redefining the problem. For the above-mentioned cases, we can introduce three rules. If the drone is able to gather information on all the plates in the first attempt, a simple rule can be applied to remove the achieved goals like `is_surveyed` and `gathered_all_plates_info` and add goals to inspect all plates like `is_plate_inspected(plate)`. Suppose the drone cannot find the information for all plates, in addition to

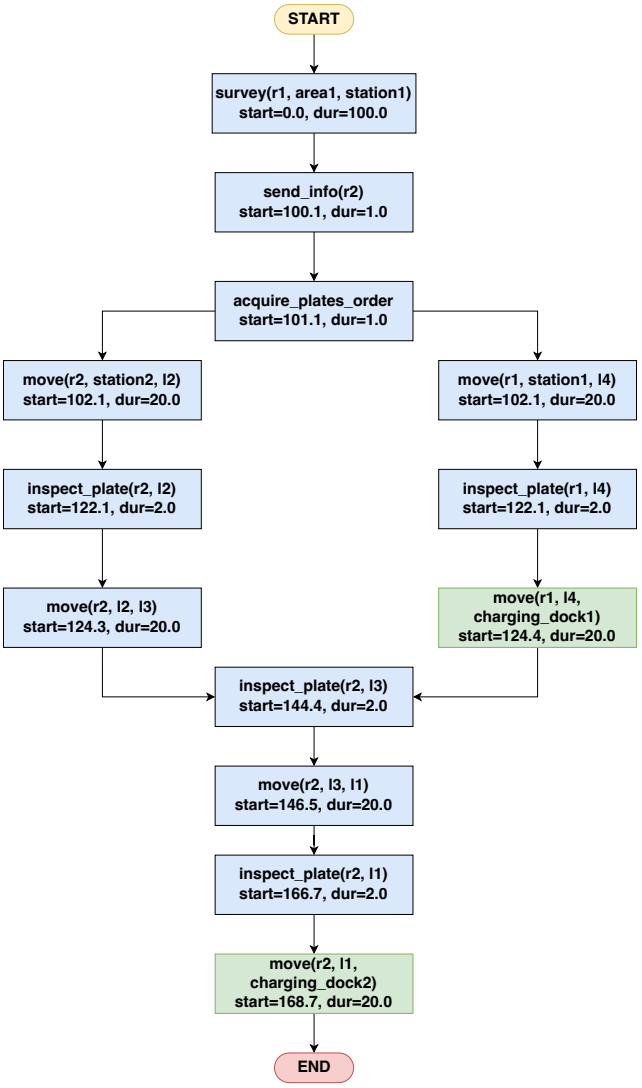


Figure 7: Dependency graph representation of the temporal plan from Listing 4.

the previously applied rule to the detected plates, the survey area can be extended by updating parameters for the `area` if the region of interest is known and large. If the region of interest is unknown, the altitude parameter set for the `survey` action can be altered to approximate the coloured plates and further refine the `inspect_plate(robot, location)` action to survey the sub-regions before inspecting the plate closer. This way, the drones can attempt to find the plates until the battery runs out. These rules allow replanning to be performed when it is not able to succeed the experiment in the first run.

The entire project including Gazebo, UP, used UP Engine (ARIES), ESB and GenoM components is available in an open source repository[6] and can be installed and tested in simulation (preferably on a computer with Ubuntu 20.04.).

[6]https://github.com/franklinselva/genom3-experiment

## 6 Conclusion and Outlook

The AIPlan4EU Unified Planning library, as part of the AI On Demand platform[7], is a suitable candidate to become a one-stop solution where students, academics, industry people can learn and experiment with various planning techniques and approaches. Yet, providing Technology Specific Bridges (TSB) to connect this platform to real world applications is paramount and required to ensure its success. In this paper we propose a TSB to embed planning within robotics systems: ESB. ESB remains middleware agnostic and tries to provide execution for several types of plans (sequential, action graph, STN, etc.), with parallelism and asynchronously. We propose a unified action representation which can be both used as part of the planning domain description, but also link plan actions to robot commands on effectors; while sensors percepts are mapped in planning fluents. Although the development of the UP and the ESB is still ongoing, we can demonstrate it in two robotic experiments using various planning engines and different robotic middleware. In fact, the bridge we propose is not necessarily limited to robotic applications and can be deployed within other embedded systems. Currently, the ESB provides a very simple API to perform plan repair and replan when needed. We intend to further develop these capabilities as they become available with more engines in the UP. There are many other functionalities pertaining to action execution which are not yet available in the current version of the bridge, still, we believe that the modular development should ease their integration in future versions. Such functionalities could include, but are not limited to, support for temporal condition evaluations, actions refinement, different strategies for replanning (such as the replanning rules mentioned above), local failures handling, skills model, etc.

## 7 Acknowledgments

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101016442 (AIPlan4EU) and by the Artificial and Natural Intelligence Toulouse Institute - Institut 3iA (ANITI) under grant agreement No ANR-19-P3IA-0004. The DFKI Niedersachsen (DFKI NI) is sponsored by the Ministry of Science and Culture of Lower Saxony and the VolkswagenStiftung.

## A Code Examples

Listing 5: Small example of using the bridge interfaces.

```python
#!/usr/bin/env python3
from up_esb.bridge import Bridge
from up_esb.plexmo import PlanDispatcher


# Application domain definitions
class Pose:
    def __init__(self, position: list):
        self.pose = position

    def __repr__(self):
        return f"Pose({self.pose})"

class Robot:
    def __init__(self, pose: Pose) ->
        None:
        self.pose = pose

    def move(self, from_pose: Pose,
        to_pose: Pose) -> bool:
        """Move from from_pose to
            to_pose."""
        if self.pose != from_pose:
            return False
        self.pose = to_pose
        return True

def robot_at(pose: Pose) -> bool:
    return robot.pose == pose

pose1, pose2 = Pose(position=[1, 0, 0]),
     Pose(position=[2, 0, 0])
robot = Robot(pose1)

# Pass application representations to
    bridge to get UP representations.
bridge = Bridge()
bridge.create_types([Robot, Pose])
up_robot = bridge.create_object("robot",
     robot)
up_pose1, up_pose2 = bridge.
    create_objects({"pose1": pose1, "
    pose2": pose2})
up_robot_at = bridge.
    create_fluent_from_function(robot_at)
up_move, (robot_param, from_pose,
    to_pose) = bridge.
    create_action_from_function(Robot.
    move)
up_move.add_precondition(up_robot_at(
    from_pose))
up_move.add_effect(up_robot_at(from_pose
    ), False)
up_move.add_effect(up_robot_at(to_pose),
     True)

# Define and solve planning problem.
problem = bridge.define_problem()
bridge.set_initial_values(problem)
problem.add_goal(up_robot_at(up_pose2))
plan = bridge.solve(problem)

# Initial Pose
print(robot.pose)

# Execute actions.
graph = bridge.get_executable_graph(plan
    )
dispatcher = PlanDispatcher()
dispatcher.execute_plan(graph)

# Final Pose
print(robot.pose)
```

[7] https://www.ai4europe.eu/

## References


Bonasso, R. P.; Firby, R. J.; Gat, E.; Kortenkamp, D.; Miller, D. P.; and Slack, M. 1997. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2/3): 237–256.

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: 20th Annual Conference, TAROS 2019, London, UK, July 3–5, 2019, Proceedings, Part I 20*, 236–250. Springer.

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 *Proceedings of the international conference on automated planning and scheduling*, volume 25, 333–341.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 42–49.

Coste-Maniere, E.; Espiau, B.; and Rutten, E. 1992. A task-level robot programming language and its reactive execution. In *IEEE International Conference on Robotics and Automation*.

Dal Zilio, S.; Hladik, P.-E.; Ingrand, F.; and Mallet, A. 2023. A formal toolchain for offline and run-time verification of robotic systems. *Robotics and Autonomous Systems*, 159: 104301.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial intelligence*, 49(1-3): 61–95.

Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3): 332–377.

Fikes, R. E. 1971. Monitored Execution of Robot Plans Produced by STRIPS. In *IFIP Congress*. Ljubljana, Yugoslavia.

Fikes, R. E.; and Nilsson, N. 1972. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4): 189–208.

Finzi, A.; Ingrand, F.; and Muscettola, N. 2004. Model-based Executive Control through Reactive Planning for Autonomous Rovers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Ghallab, M.; Knoblock, C.; Wilkins, D.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D.; Sun, Y.; and Weld, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, AIPS.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hofmann, T.; Viehmann, T.; Gomaa, M.; Habering, D.; Niemueller, T.; and Lakemeyer, G. 2021. Multi-Agent Goal Reasoning with the CLIPS Executive in the RoboCup Logistics League. In *ICAART*, 80–91.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891.

Ingham, M. D.; Ragno, R. J.; and Williams, B. C. 2001. A Reactive Model-based Programming Language for Robotic Space Explorers. In *International Symposium on Artificial Intelligence, Robotics and Automation for Space*.

Ingrand, F.; Lacroix, S.; Lemai-Chenevier, S.; and Py, F. 2007. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7): 559–580.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *International Joint Conference on Artificial Intelligence*.

Lima, O.; Günther, M.; Sung, A.; Stock, S.; Vinci, M.; Smith, A.; Krause, J. C.; and Hertzberg, J. 2023. A Physics-Based Simulated Robotics Testbed for Planning and Acting Research. In *ICAPS Workshop on Planning and Robotics (PlanRob 2023)*.

Martín, F.; Clavero, J. G.; Matellán, V.; and Rodríguez, F. J. 2021. Plansys2: A planning system framework for ros2. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9742–9749. IEEE.

Martín Rico, F.; Morelli, M.; Espinoza, H.; Rodríguez-Lera, F. J.; and Matellán Olivera, V. 2021. Optimized execution of PDDL plans using behavior trees. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, 1596–1598.

McGann, C.; Py, F.; Rajan, K.; Ryan, J.; and Henthorn, R. 2008. Adaptive Control for Autonomous Underwater Vehicles. In *AAAI Conference*, 6.

Nesnas, I. A.; Wright, A.; Bajracharya, M.; Simmons, R. G.; and Estlin, T. A. 2003. CLARAty and Challenges of Developing Interoperable Robotic Software. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

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 *ICRA workshop on Open Source Software*. Kobe, Japan.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 6459–6464. IEEE.

Verma, V.; Estlin, T. A.; Jónsson, A. K.; Pasareanu, C.; Simmons, R. G.; and Tso, K. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *International Workshop on Planning and Scheduling for Space*. Citeseer.