ROSPlan: Planning in the Robot Operating System

Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, and Bram Ridder
King's College London
London WC2R 2LS
firstname.lastname@kcl.ac.uk

Arnau Carrera[a], Narcís Palomeras[b], Natàlia Hurtós[b], and Marc Carreras[a]
University of Girona
17071 Girona, Spain
[a]*firstname.lastname*@udg.edu
[b]n*lastname*@eia.udg.edu

Abstract

The Robot Operating System (ROS) is a set of software libraries and tools used to build robotic systems. ROS is known for a distributed and modular design. Given a model of the environment, task planning is concerned with the assembly of actions into a structure that is predicted to achieve goals. This can be done in a way that minimises costs, such as time or energy. Task planning is vital in directing the actions of a robotic agent in domains where a causal chain could lock the agent into a dead-end state. Moreover, planning can be used in less constrained domains to provide more intelligent behaviour. This paper describes the ROSPLAN framework, an architecture for embedding task planning into ROS systems. We provide a description of the architecture and a case study in autonomous robotics. Our case study involves autonomous underwater vehicles in scenarios that demonstrate the flexibility and robustness of our approach.

1 Introduction

Planning is concerned with organising instances of actions in order to achieve certain goals (Ghallab, Nau, and Traverso 2004). It begins with a domain model describing the actions available to the planner and a description of the current state. The actions are then assembled into a structure that is causally valid, with an attempt to optimise some cost function. In order to do this, planning must forecast interactions with future constraints, avoid moving the executor into dead-end situations, and still achieve the goals.

The Robot Operating System (ROS) (Quigley et al. 2009) is a set of software libraries and tools used in building robotic systems. ROS has become a popular platform for robotics research and has also proved a flexible foundation on which to build robotic control via task planning (Bernardini, Fox, and Long 2014; Cashmore et al. 2014; Dornhege, Hertle, and Nebel 2013).

Combining task planning and robotics presents several challenges, in particular:

- Given a domain model that matches the capabilities of the robot, an initial state must be generated that matches the current environment.
- Actions planned by the task planner, on an abstracted model of the world, must be made concrete and dispatched to lower level controllers.
- Plans must be executed according to some strategy. This must account for action failure, plan failure due to ignorance or change in a dynamic environment, and changing mission requirements.

We introduce ROSPLAN[1]. ROSPLAN is a framework for embedding a generic task planner in a ROS system. The architecture is highly modular and deals with the stated challenges by providing tools to: automatically generate the initial state from the knowledge parsed from sensor data and stored in a knowledge base; automate calls to the planner, then post-process and validate the plan; handle the main dispatch loop, taking into account changing environment and action failure; and match planned actions to ROS action messages for lower level controllers.

Integrating planning with robotics through a plan execution architecture has been done successfully by others, for example by McGann et al. (2008) in the T-REX system. T-REX is a timeline based plan execution architecture supporting distributed deliberation amongst a collection of *reactors*. Each physical component of the robotic system comprises one or more inter-dependent reactors which are responsible for evolving state variables on the different timelines. Each reactor has a look-ahead, determining how far ahead it can plan, and a latency, bounding the planning time available to it. The system is synchronised using a clock, and concurrent activity is achieved by enforcing interdependencies between the timelines of the relevant state variables. Generative planning is done using the Europa system (Frank and Jonsson 2003) with a timeline-based modelling language. T-REX has been used successfully to plan and execute underwater AUV missions at the Monterey Bay Aquarium Research Institute (MBARI) (Graham et al. 2012; Py, Rajan, and McGann 2010; Magazzeni et al. 2014). Task planning has also been embedded into robotic systems in a number of other ways. In particular, Ponzoni et al. (2014) describe a configurable anytime meta-planner that drives a (PO)MDP planner, anticipating the probabilistic evolution of the system; Srivastava et al. (2014) provide

[1]The source code and documentation for ROSPlan can be obtained from *https://github.com/KCL-Planning/ROSPlan*

an interface between task and motion planning, reasoning about geometric constraints and communicating those to a task planner; and Gaschler et al. (2013) use a *knowledge-of-volumes* approach, which treats volumes as an intermediary representation between continuous-valued robot motions and discrete symbolic actions. Tennorth et al have explored the connection between planning, execution and knowledge management in a significant body of work, including (Tenorth, Bartels, and Beetz 2014; Tenorth and Beetz 2009). There is also considerable work exploring the planning-execution connection, including RAP (Firby 1987), work of Beetz and McDemott (1994), PRS (Ingrand et al. 1996), Simmon's Task Description Language (1992) and subsequent work (Kortenkamp and Simmons 2008), the IxTeT-Exec system (Lemai-Chenevier and Ingrand 2004) and IDEA (Muscettola et al. 2002). Although these systems all confront similar problems in mediating between sensor-actuator level behaviour and the symbolic representations and causal reasoning used in planning, they present different approaches to managing the levels of abstraction, the delegation of executive control from planner to lower levels, the handling of uncertainty and the precise mix of planned and reactive behaviour.

This paper describes a framework for linking generic task-planning with an execution interface provided by ROS. While T-REX, and other plan execution frameworks developed to date, are powerful and effective, they exploit individual and specific methods and languages that are not widely adopted standards. By contrast, our approach links two standards together: PDDL2.1, the temporal and numeric standard planning domain description language, and the Robot Operating System (ROS). Our objective is to provide a modular architecture into which different temporal planners can easily be plugged: for example, POPF (Coles et al. 2010) (used in the case study described in this paper) can be replaced by Temporal Fast Downward (Eyerich, Mattmüller, and Röger 2012), LPG (Gerevini and Serina 2002), UPMurphi (Della Penna, Magazzeni, and Mercorio 2012), or any other planner capable of reasoning with PDDL2.1. With an appropriately implemented plan dispatcher, even non-temporal planners and planning models can be exploited. A new robotics application requiring planning can then be achieved simply by providing the relevant ROS action messages for the controllers of the robotic system. We see our main contribution to be the provision of an open standard and implementation of an integrated task planning and execution framework that brings together all of these standardised components.

We demonstrate our approach with a case study planning inspection and valve-turning missions for autonomous underwater vehicles (AUVs). In these missions an AUV equipped with a manipulator is placed in an underwater structure, with the task to inspect certain areas and to ensure that valves are turned to correct angles. The AUV has no initial knowledge of the structure, the location of the valve panel, or the angles of the valves. We run the mission in both simulation and live trials with an instantiation of ROSPLAN. To illustrate its generality, we point out that the same architecture has been instantiated to control two different physical AUVs and one simulated AUV, each performing a different variety of missions with different PDDL domain descriptions. ROSPlan is also used with the Festo Robotino platform in the EU FP7 Squirrel project[2].

2 Architecture

The ROSPLAN framework is intended to run with any PDDL 2.1 (Fox and Long 2003) domain and planner, and to automate the planning process in ROS, coordinating the activities of lower level controllers. An overview of the ROSPLAN framework is shown in figure 1.

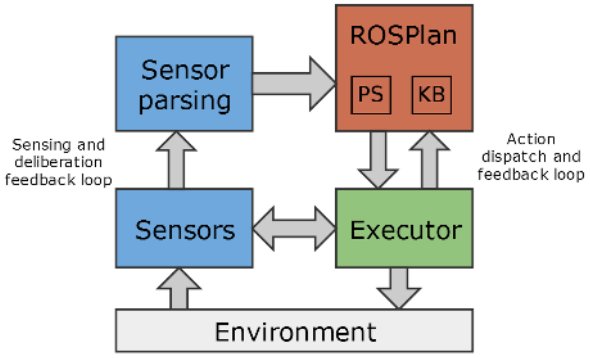

Figure 1: General overview of the ROSPLAN framework (red box), consisting of the Knowledge Base and Planning System ROS nodes. Sensor data is passed continuously to ROSPlan, used to construct planning problem instances and inform the dispatch of the plan (blue boxes). Actions are dispatched as ROS actions and executed by lower-level controllers (green box) which respond reactively to immediate events and provide feedback.

ROSPLAN includes two ROS nodes, the Knowledge Base and the Planning System. The Knowledge Base is simply a collection of interfaces, and is intended to collate the up-to-date model of the environment. The Planning System acts as a wrapper for the planner, and also dispatches the plan. The Planning System:

- builds the initial state automatically – as a PDDL problem instance – from the knowledge stored in the Knowledge Base;
- passes the PDDL problem instance to the planner and post-processes, then validates the plan; and
- dispatches each action, deciding when to reformulate and re-plan.

The architecture can be described in three parts: *knowledge gathering*, *planning*, and *dispatch*.

Knowledge gathering refers to the process of populating the Knowledge Base, generally from sensor data, parsed to correspond to the domain (such as waypoints generated from a geometric map) and as real information used to direct the low-level planners (such as the real coordinates of

[2]The authors acknowledge support for this research from EU FP7 projects PANDORA (288273) and Squirrel (610532) and from the UK Engineering and Physical Sciences Research Council.

those waypoints). The Knowledge Base is then used in three ways by the Planning System:

- to generate the PDDL problem file;
- when translating PDDL actions to ROS action messages, to populate the messages with real data; and
- to notify the planner if there is a change in the environment that may invalidate the plan.

The first is described as a part of planning, the rest as part of dispatch. The architectural overview is shown in figure 2, using an example instantiation of the Knowledge Base described further in Section 3.

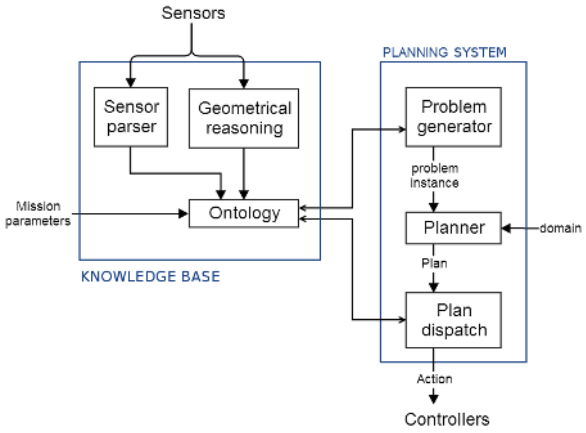

Figure 2: Architectural overview of the ROSPLAN framework. Blue boxes are ROS nodes. An example instantiation of the Knowledge Base is illustrated. Sensor data is interpreted for addition to the ontology, and actions are dispatched as ROS actions from the Planning System. The Planning System (PS) and Knowledge Base (KB) communicate during construction of the intial state and plan dispatch. During dispatch the PS will update the KB with the planning filter, and the KB may notify the PS of changes to the environment.

Knowledge Gathering

To use the ROSPLAN framework a user must first instantiate the Knowledge Base. In figure 2 this is shown as an ontology. The ontology is then used to implement the ROS interface used by the Planning System. This ontology implementation of the Knowledge Base is described in Section 3. The Knowledge Base is modular, to exploit the structural similarities between many robotics planning domains. For example, in our case study we represent the areas the AUV can move between as waypoints. For this reason the Knowledge Base and domain contain waypoints, and this translation is handled by a generic component.

The Knowledge Base is updated as soon as new information becomes available. Each change to the Knowledge Base is checked against a planning filter supplied by the Planning System. If the filter contains the object, or object type, that has been added or removed, a notification message is sent to the Planning System. This alerts the Planning System that something has changed in the environment that could potentially invalidate the plan. The construction of the filter is handled by the Planning System, and described below.

The Knowledge Base interface is used by the Planning System to generate the PDDL problem instance by supplying an initial state and goals. This is done through an interface comprised of ROS services.

```
1:  procedure PLAN DISPATCH(Domain D, Mission M)
2:      while M contains goals do
3:          I := generateProblem(D, M);
4:          P := plan(D, I);
5:          F := constructFilter(D, I, P);
6:          while execute do
7:              a := pop(P);
8:              dispatch(a);
9:              while actionExecuting do
10:                 if filterViolated then
11:                     execute := ⊥;
12:                     cancel(a);
13:                 end if
14:             end while
15:             execute := execute ∧ actionSuccess(a);
16:         end while
17:     end while
18: end procedure
```

Figure 3: Detail of the plan, re-plan and dispatch loop used in ROSPLAN. $generateProblem(D, M)$ automatically generates a PDDL problem instance from the model. $constructFilter(D, I, P)$ extracts the filter, used by the Knowledge Base. These procedures are explained fully in the text.

Planning and Dispatch

The Planning System constructs a PDDL problem instance, sends this to an external PDDL planner, extracts a filter from the plan, dispatches the actions, and handles re-planning. This is illustrated in figure 3.

To generate the PDDL problem (line 3 of figure 3), the Planning System parses the supplied domain file, then queries the Knowledge Base for the object instances, facts, and fluents that comprise the initial state, and also the current goals. This problem file is then handed to a PDDL planner (line 4), which produces a plan. ROSPLAN can be used with any planner, so long as it can handle the syntactic requirements of the domain. In our case study we use an anytime version of POPF, a temporal and numeric planner. The amount of time allowed for planning is provided as a parameter to the node. Once the plan has been found it can be validated using the Plan Validation Tool (VAL) (Howey, Long, and Fox 2004), which is included in ROSPLAN.

The filter is constructed (line 5) by taking the intersection of static facts in the problem instance with the union of all preconditions of actions in the plan. In addition, each object instance involved in these facts is added to the filter. For example, suppose we have a PDDL domain with the object type `waypoint`, the static fact `(connected ?from`

?to - waypoint) and the *do_hover_fast* action shown in figure 4. For each *do_hover_fast* action scheduled in the plan, the waypoint instances bound to from and to and the ground fact (connected ?from ?to) are included in the filter. If these objects are removed, or altered in the Knowledge Base, a notification will be sent to the Planning System, as described above.

```
(:durative-action do_hover_fast
  :parameters (?v - vehicle
      ?from ?to - waypoint)
  :duration ( = ?duration
      (\ (distance ?from ?to) (vel ?v)))
  :condition (and
    (at start (at ?v ?from))
    (at start (connected ?from ?to)))
  :effect (and
    (at start (not (at ?v ?from)))
    (at end (near ?v ?to)))
)
```

Figure 4: The *do_hover_fast* action for the inspection and valve turning task.

The action is dispatched by linking the PDDL action to a ROS action message (lines $7-8$). The relationship between the high-level PDDL actions and low-level control has been investigated in many ways (Srivastava et al. 2014; Gaschler et al. 2013; Geib et al. 2006; McGann et al. 2008; Kortenkamp and Simmons 2008). To take advantage of pre-existing work, the translation layer can be instantiated as a separate node. The Planning System will dispatch PDDL actions – using a message type defined by ROSPLAN – to the intermediate layer, where the translation can be carried out. As well as this, ROSPLAN includes several modules, similar to those provided by the Knowledge Base, that correspond to common actions found in robotics domains. For example, the mapping from movement actions (analogous to the waypoint example in the Knowledge Base) to the *move_base* library in ROS.

In ROSPLAN, re-planning is based on reformulation of the problem. There are three reasons that the system may re-plan:

1. the current action returns failure;
2. the Knowledge Base informs the planner of a change that invalidates the plan, or of new information important to the mission;
3. the current action, or plan, has gone significantly over or under budget (such as time or energy).

In the case of (1), the dispatch loop is broken (line 15), the problem instance is rebuilt from the current model of the environment, and a new plan is generated. In the case of (2) or (3), the current action may still be executing. In these cases the original plan is re-validated and then, if found to be invalid, the current action is cancelled and the dispatch loop broken as before (lines $10-13$).

3 Planning with ROSPLAN

In this section we discuss implementing a ROS system with ROSPLAN. In particular we consider the scenario of planning inspection and valve-turning missions for autonomous underwater vehicles (AUVs). This scenario is fully described in section 4. To build a plan-based control architecture with ROSPLAN the user must supply:

1. ROS action messages,
2. components to interpret the sensor data,
3. a PDDL domain file, and
4. an instantiation of the Knowledge Base.

As mentioned in sections 1 and 2, ROS libraries already exist for (1), (2) and (4), and they need only to be linked to ROSPlan. The contribution of ROSPLAN is an open, standard framework to link these existing components together, and to automate the planning and dispatch process.

With respect to our case study, we describe our instantiation of the Knowledge Base; then the construction of the PDDL problem instance and integration with the planner; finally, we discuss plan dispatch.

Instantiating the Knowledge Base

The Knowledge Base was implemented in our case study to include an ontology, using the *Web Ontology Language* (OWL) (Hitzler et al. 2009). An *ontology* is a way to represent knowledge that can be expressed by description logics. Tenorth et al. (2010) combines semantic knowledge with spatial data to form a *semantic map* of the environment, attaching semantic information to a spatial map using an ontology in a ROS framework for indoor household tasks. We used a similar approach in instantiating the Knowledge Base for our case study.

The interface to the ontology was adpated to the Knowledge Base interface and is used by ROSPLAN to construct the initial state of the problem. Then, the ontology interface was augmented to check information against the planning filter when the information is added or removed.

The onotolgy is used to store symbolic representation of the environment. This includes: locations that the AUV can visit, inspection points the AUV is intended to observe, the valve panels, and the valves. The objects, and the relationships between them are derived from sensor data throughout the execution of a plan and collated in the ontology.

The possible trajectories the AUV can traverse are determined by a set of *waypoints*. As mentioned in section 2, we use a generic component to handle the description and construction of waypoints. These waypoints are created using a probabilistic road map (PRM) (Kavraki, P. Svestka, and Overmars 1996), and stored in the ontology. We use an octomap to check if a waypoint collides with an obstacle in the world. Similarly we use the octomap to determine whether the AUV can traverse between two waypoints. The octomap is built from sensor data and continuously updated. If an edge or waypoint is discovered to be colliding with the environment, then it is removed from the ontology.

We use another component to describe valve panels and valves. This component simply listens to the feature matching process described in section 4 and adds discovered objects to the ontology.

Finally, inspection points are added to the ontology. These are areas that must be observed by the AUV. The PRM is augmented with additional waypoints – called *strategic waypoints* – these waypoints are stored in the ontology to provide a denser collection of waypoints around points of interest (in our case the possible locations of valve panels and unexplored space).

PDDL domain and problem instance

The domain for the valve turning mission is partially shown in figure 5. The *do_hover_controlled* and *do_hover_fast* actions are used to move an AUV between connected waypoints. After using *do_hover_fast* the position of the AUV must be corrected. There are two modes of movement used by the vehicle: one for the *correct_position* action and controlled movement, another for fast movement.

The inspection points can be observed from possibly many waypoints, with varying amounts of visibility. Observing an inspection point from a waypoint increases the `(observed ?ip)` function by the visibility amount, `(obs ?ip ?wp)`. An inpsection point `ip` is fully observed when `(observed ip)` $>= 1$.

The valves must be corrected before some deadline, specified by a timed-initial-literal (Hoffmann and Edelkamp 2005). At the deadline the valve becomes blocked and cannot be turned. Before the deadline, the *turn_valve* action can be used to change `(valve_state ?v)` to equal `(valve_goal ?v)`. After turning the valve, the AUV's position must be corrected, and the panel re-examined.

When reformulating the problem for (re-)planning the Planning System queries the Knowledge Base for instances of all object types and their attributes, the problem instance is automatically generated and passed to the planner. An example problem instance from our case study is shown in figure 6. In this problem there are five mission goals on a single valve. The time windows are known a priori and stored in the ontology as with the information derived from sensors.

Plan Dispatch

Plans are produced by POPF, a temporal metric planner widely used in the AI Planning community, which can cope with a wide variety of common features (Piacentini et al. 2013; Fox, Long, and Magazzeni 2012).

As described in Section 2, each PDDL action is linked to a ROS action message and dispatched. This translation is performed by separate components, corresponding to those used in the Knowledge Base. For example, figure 7 shows the translation of a PDDL action (*do_hover_controlled*) to a *GotoWithYawRequest* ROS message from the COLA2 control architecture for AUVs (Palomeras et al. 2012). This translation is performed by a component paired with the PRM generation. This component performs the translation of each *do_hover_fast* action by first fetching the real coordinates from the Knowledge Base.

The plans are dispatched by sending the ROS actions to the lower-level controllers. The dispatcher is agnostic to the choice of lower level controller, allowing us to run ROSPLAN in both simulation and physical trials with no changes.

```
(define (domain valve_turning)
...
(:types
  waypoint inspectionpoint
  vehicle panel valve)
(:predicates
  (at ?v - vehicle ?wp - waypoint)
  (near ?v - vehicle ?wp - waypoint)
  (connected ?wp1 ?wp2 - waypoint)
  (cansee ?v-vehicle
      ?ip-inspectionpoint ?wp-waypoint)
  (canexamine ?v - vehicle
      ?p - panel ?wp - waypoint)
  (canreach ?v-vehicle
      ?p-panel ?wp-waypoint)
  (examined ?p - panel)
  (on ?a - valve ?p - panel)
  (valve_blocked ?a - valve)
  (valve_free ?a - valve)
)
(:functions
  (distance ?from ?to - waypoint)
  (observed ?ip - inspectionpoint)
  (obs ?ip - inspectionpoint
      ?wp - waypoint)
  (valve_goal ?va - valve)
  (valve_state ?va - valve)
  (valve_goal_completed ?va - valve)
)
(:action do_hover_controlled ...)
(:action do_hover_fast ...)
(:action correct_position ...)
(:action observe_inspection_point ...)
(:action examine_panel ...)
(:action turn_valve ...))
```

Figure 5: Fragment of the domain for the inspection and valve turning task.

4 Case study

We demonstrate our approach with a case study planning inspection and valve-turning missions for autonomous underwater vehicles (AUVs). The scenario was run in a simulation environment (UWSim) and in physical trials. The scenario involved a single AUV performing a valve turning mission.

The mission seeks to achieve persistent autonomy, with the key goal to show significantly reduced frequency of assistance requests during subsea inspection and intervention. This requires a strategic capability on the part of the vehicles, which can be achieved by planning over long-horizon activities, selecting sequences of actions that will achieve long-term objectives. Ensuring that the planning model remains robust requires careful matching of the model to the real world, including dynamically updating the model from continuous sensing actions.

The AUV was told to correct the positions of four valves, without prior knowledge of the size of the tank, existing obstacles, or location of the valve panel. The valve panel was placed nearby one of eight locations in an otherwise empty tank (as shown in Figure 8).

```
(define (problem valve_task)
(:domain valve_turning)
(:objects
        auv - vehicle
        wp1 ... wp20 - waypoint
        p1 - panel
        v1 - valve)
(:init
        (at auv wp1)
        (canreach auv wp8 p1)
        (on v1 p1)  (valve_blocked v1)
        (= (valve_goal v1) 0)
        (= (valve_state v1) 0)
        (= (valve_goal_completed v1) 0)

        ;; VALVE TIME WINDOW
        (at 1 (not (valve_blocked v1)))
        (at 1 (valve_free v1))
        (at 1 (= (valve_goal v1) 144))
        (at 251 (valve_blocked v1))
        (at 251 (not (valve_free v1)))
        ...
        (connected wp1 wp2)
        (= (distance wp1 wp2) 5.072)
        (connected wp2 wp1)
        (= (distance wp2 wp1) 5.072)
        ...)
(:goal (and
        (>= (valve_goal_completed v1) 5))))
```

Figure 6: Fragment of the problem instance for the inspection and valve turning tasks.

PDDL action
423.199: (do_hover_controlled auv wp2 wp36) [10.726]

GotoWithYawRequest
north_lat:=0.012, east_lon:=2.146, z:=1.383, altitude_mode:=False, yaw := 1.541, tolerance := 0.3

Figure 7: An example conversion from the PDDL action *do_hover_controlled* to a *GotoWithYawRequest* action. The real values are extracted from the Knowledge Base by the action dispatcher before the action is sent to the movement controller.

The expected mission outcome is for the AUV to first explore the space and observe the possible valve panel locations (inspection points) in an efficient way. Once a valve panel has been located, the remaining inspection actions will be discarded and the AUV will move to the panel and closely inspect the valves. Finally, having detected the current angles of the valves, the AUV will use its gripper to correct the positions of any misaligned valves.

We aim to show from this study that the ROSPLAN framework:

1. through task planning queries, generates a reasonable plan to achieve the mission goals;
2. is able to adapt to unexpected discoveries or changes in the environment, in a way that is more intelligent than simple reactive behaviour; and
3. is robust to inaccuracy of the sensors and low-level controllers and other uncertaintry in the real world.

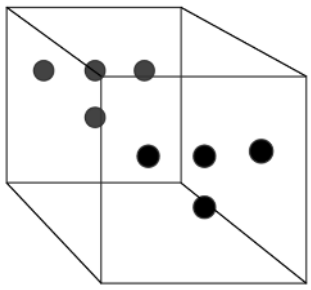

Figure 8: Scenario for the valve turning task. The possible locations of the valve panel are shown by the dark circles on opposite sides of the tank. The tank is roughly 5m wide, with 8m between opposing panel locations.

The problem itself is simple in task planning terms. However, as the ROSPLAN framework is general with respect to choice of domain and planner, here we focus on the challenges encountered at the meta-level; in dynamically constructing the problem and dispatching the plan. The valve turning mission involves:

1. a series of planned and uninterrupted dispatched actions (searching for the valve panel),
2. the discovery of new information, interrupting the execution of a plan (the discovery of a valve panel),
3. action failure (the temperamental turning of valves underwater) and sensor inaccuracies (in the detected position of the panel, and angles of the valves).

Thus, the valve turning mission is a suitable example for our three aims.

Girona 500 I-AUV

The Girona 500 I-AUV (Ribas et al. 2012) is a compact and lightweight AUV with hovering capabilities, shown in figure 9. The vehicle can be actuated in *surge*, *sway*, *heave* and yaw (ψ), while it is stable in $roll$ (Φ) and *pitch* (θ). The payload area is equipped with an ECA CSIP manipulator which is an under-actuated manipulator with 4 degrees of freedom (DoF). This manipulator can control the Cartesian position (x,y,z) and the $roll$ (Φ) of the end-effector. However, because it is under-actuated, $pitch$ and yaw are defined by the Cartesian position.

The manipulator has a custom end-effector that includes a compliant *passive gripper* to absorb small impacts, designed in a V-Shape to easily drive the handle of a T-bar valve towards its center. It also contains a force/torque sensor to detect the contact force, keep grasping, and control the torque needed to turn the valves.

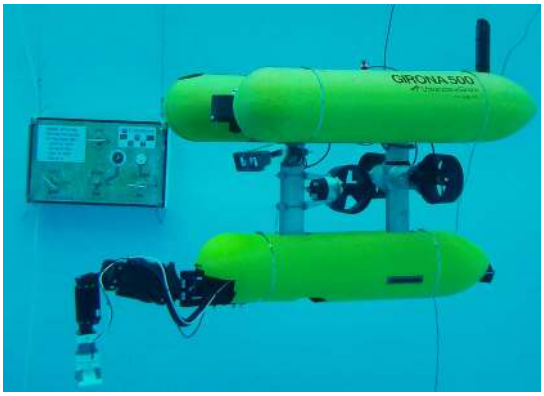

Figure 9: Girona 500 AUV in the water tank, equipped with the manipulator and a customised end-effector. In the background there is a mock-up of a valve panel.

Action Implementation

The Girona 500 AUV uses a simultaneous localisation and mapping algorithm (SLAM), based on an EKF filter, to acheive a robust vehicle localisation. The EKF-SLAM combines the information of different navigation sensors (i.e. an AHRS, a DLV and a depth sensor) with a motion model to obtain an estimation of the vehicle position. It also uses detected landmarks in the environment to improve the AUV position, and maintain a map of their location. In our implementation, the valve panel (see figure 10) is a landmark. To identify the panel, a vision algorithm analyses the images gathered by the AUV's main camera and compares them with a template image of the panel. Then, when a sufficient number of features are matched across both images, the presence of the panel is identified, and its position/orientation accurately estimated. This process is embodied in the action *observe_inspection_point*.

Once the position of the panel is known, it is possible to estimate the valves' orientations, which corresponds to the *examine_panel* action. The action extracts a region of interest around the expected valve position and applies a Hough transform to estimate the main line orientation.

The Girona 500 can be controlled by means of body force and velocity requests, and waypoints requests. A cascade control scheme is used to link these controllers: the pose controller generates the desired velocity as output, which is the input for the velocity controller. This in turn generates the force and torque that are fed to the force controller to finally obtain the required setpoints for each thruster.

Two motion modes have been implemented to guide the vehicle. The first (*do_hover_controlled* and *correct_position*) follows the cascade scheme moving the AUV holonomically. However, if the waypoint is far from the vehicle's current position this motion mode becomes too slow. Then, a variation of the line-of-sight pure pursuit algorithm described by Fossen (Fossen 2011) is used (*do_hover_fast*).

To perform the valve turning action (*turn_valve*) we have implemented a learning by demonstration algorithm (LbD). LbD is a machine learning technique designed to transfer the knowledge from an expert to a machine through an abstraction process that generalises a task from a set of operator demonstrations. Here we address a particular case study, targeting valve turning. However, the implementation through a LbD algorithm can be easily adapted to other manipulation actions, and is ideal for facing new tasks.

The LbD algorithm used in this action is an extension of dynamic movement primitives (DMP). DMP is a technique where the learned skill is encapsulated in a superposition of basis motion fields. Unlike other methods, DMP dynamically generates the required commands to perform the reproduction of the task trajectory. This makes the approach more robust to external perturbations.

In our particular implementation, the extended version of the DMP (Carrera et al. 2014) learns a trajectory of 8 DoF to represent the position (x, y, z) and orientation (*yaw*) of the AUV and the position (x, y, z) and alignment (*roll*) of the end-effector. Because this trajectory is recorded with respect to a valve center, the model learned for changing the reference valve is good enough for grasping any other valve. The roll motion performed to turn the valve is not included in the learning algorithm. Instead, a simple controller executes the turn for any rotation angle once the valve is grasped.

Figure 10: Camera image of the valve panel, taken from the AUV's end-effector.

Outcome

The physical trials were successful, with the AUV correcting all the valves. Replanning was performed for two reasons: intially discovering the panel, and when a *turn_valve* action failed. The *turn_valve* action did not have a perfect success rate, as sometimes the valve would be missed by the gripper, or knocked when the AUV retreated from the panel.

In the first case, the panel was discovered by the AUV's main camera as described above. The position of the valve panel was entered into the Knowledge Base by the visual detection component, as described in section 3. This change of information violated the mission filter, prompting a replan. A notification was sent to the dispatcher and the current action was canceled (in this case a *do_hover_controlled*). The Planning System queried the Knowledge Base to construct a

PDDL action	ROS action message
0.000: (observe_inspection_point auv wp1 ip3) [10.000]	-
10.001: (correct_position auv wp1) [10.000]	GotoWithYawRequest
20.002: (do_hover_fast auv wp1 wp2) [35.848]	GotoWithYawRequest
55.851: (correct_position auv wp2) [10.000]	GotoWithYawRequest
65.852: (observe_inspection_point auv wp2 ip4) [10.000]	-
75.853: (correct_position auv wp2) [10.000]	GotoWithYawRequest
85.854: (do_hover_controlled auv wp2 wp23) [16.710]	GotoWithYawRequest
...	...
423.199: (do_hover_fast auv wp2 wp36) [10.726]	GotoWithYawRequest
433.926: (correct_position auv wp36) [10.000]	GotoWithYawRequest
443.927: (observe_inspection_point auv wp36 ip9) [10.000]	-
0.000: (turn_valve auv wp1 p0 v1) [100.000]	valve_turning_action
100.001: (correct_position auv wp1) [10.000]	GotoWithYawRequest
110.002: (turn_valve auv wp1 p0 v3) [100.000]	valve_turning_action
210.003: (correct_position auv wp1) [10.000]	GotoWithYawRequest
220.004: (turn_valve auv wp1 p1 v2) [100.000]	valve_turning_action
...	...
1310.002: (correct_position auv wp1) [10.000]	GotoWithYawRequest
1311.003: (turn_valve auv wp1 02 v4) [100.000]	valve_turning_action

Figure 11: Fragments of two PDDL plans produced during the valve turning mission. The first plan fragment shows the beginning and end of an inspection mission, searching for the valve panel. The second fragment shows the plan to correct the valves, once the panel has been found.

new problem instance, which included the newly discovered panel, and then passed this to the planner.

In the second case – when a *turn_valve* action failed – the action itself did not return failure, as the learned action does not self-validate. Instead, the action incorporated an examination of the panel following the attempted turn. This examination updated the angle of the valve in the Knowledge Base. If the *turn_valve* action failed, then the planning filter would be violated and a replan would take place. The new plan would repeat the attempt to correct the valve.

Figure 11 shows fragments of the plans generated during the valve turning mission. The table shows the PDDL actions and their corresponding ROS action messages. The *observe* PDDL action does not have a corresponding ROS action, as the visual sensing is continuous and passive. The *observe* action merely causes the AUV to wait for several seconds after orienting itself towards the inspection point – this aids the visual detection. The two movement types, fast (*do_hover_fast*) and controlled (*do_hover_controlled* and *correct_position*), have the same action type: *GotoWithYawRequest*. This ROS action message is shown in more detail in figure 7. These messages are sent to different action servers depending upon the movement type, as described above. Finally, the *valve_turning_action* message is sent to the LbD controller described above.

In the physical trials the inspection points did not line up with the borders of the tank, nor with the actual location of the valve panel. This was due to safety concerns; it is better to keep the AUV further from the walls, as any drift in the accuracy of its current estimated position could cause it to collide. This proved to highlight a strength of the approach. As the PDDL problem and its underlying PRM are generated dynamically from the sensor data, once the valve panel was discovered the waypoints of the new PRM would be repositioned to allow the AUV access to the valve panel – closer than the safety constraints would have allowed. Simultaneously, the real position of the panel would be known, and used as a landmark to help keep the AUV's estimated position accurate, removing the need for the safety net.

5 Conclusion

In this paper we have introduced the ROSPLAN planning framework for ROS. We have proposed this architecture as a standard solution to embedding a task planner in the PDDL family into a robotic system. While others have proposed similar architectures, we have focussed on providing a standardised solution for the integration, through ROS, of modern heuristic search planners with robotic systems. We have described the architecture in detail, and shown its success in application. Our case study, which considers the persistently autonomous behaviour of underwater vehicles, demonstrates the flexibility and robustness of reasoning dynamically over the environment combined with reformulation and replanning. The planner remains in constant communication with the knowledge base throughout plan execution, and each replanning cycle relies on knowledge discovery and the autonomous construction of a revised initial and goal state description. ROS provides the framework for this communication, so that the burden on the application designer is focused on defining the PDDL domain description. Then, existing action messages and other ROS components specific to the robotic systems being deployed can be simply linked together. We use ROSPLAN as an open, standard framework to link existing ROS components to planning tools. We have implemented several original components, used in our case study, that can be re-used in other robotics applications. We propose partial PDDL domain descriptions that match these components, such as a standardised waypoint description.

References

Beetz, M., and McDermott, D. 1994. Improving Robot Plans During Their Execution. In *Proc. International Conference on AI Planning Systems (AIPS)*.

Bernardini, S.; Fox, M.; and Long, D. 2014. Planning the Behaviour of Low-Cost Quadcopters for Surveillance Missions. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Carrera, A.; Palomeras, N.; Ribas, D.; Kormushev, P.; and Carreras, M. 2014. An intervention-auv learns how to perform an underwater valve turning. In *OCEANS - Taipei, 2013 MTS/IEEE*.

Cashmore, M.; Fox, M.; Larkworthy, T.; Long, D.; and Magazzeni, D. 2014. AUV Mission Control Via Temporal Planning. In *Proc. Int. Conf. on Robots and Automation (ICRA)*.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 42–49.

Della Penna, G.; Magazzeni, D.; and Mercorio, F. 2012. A universal planning system for hybrid domains. *Applic. Intell.* 36(4):932–959.

Dornhege, C.; Hertle, A.; and Nebel, B. 2013. Lazy Evaluation and Subsumption Caching for Search-Based Integrated Task and Motion Planning. In *Proc. IROS workshop on AI-based robotics*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2012. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments*. 49–64.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Proc. National conference on Artificial intelligence (AAAI)*, 202–206.

Fossen, T. I. 2011. *Handbook of marine craft hydrodynamics and motion control*. John Wiley & Sons, Ltd.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of AI Research (JAIR)* 20:61–124.

Fox, M.; Long, D.; and Magazzeni, D. 2012. Plan-based policies for efficient multiple battery load management. *J. Artif. Intell. Res. (JAIR)* 44:335–382.

Frank, J., and Jonsson, A. 2003. Constraint-based attribute and interval planning. *Journal of Constraints* 8(4).

Gaschler, A.; Petrick, R. P. A.; Giuliani, M.; Rickert, M.; and Knoll, A. 2013. KVP: A knowledge of volumes approach to robot task planning. In *Proc. Int. Conf. on Intelligent Robots and Systems (IROS)*, 202–208.

Geib, C.; Mourão, K.; Petrick, R.; Pugeault, N.; Steedman, M.; Krueger, N.; and Wörgötter, F. 2006. Object Action Complexes as an Interface for Planning and Robot Control. In *Proc. Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*.

Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs. In *Proc. Int. Conf. on Automated Planning and Scheduling (AIPS)*. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Graham, R.; Py, F.; Das, J.; Lucas, D.; Maughan, T.; and Rajan, K. 2012. Exploring Space-Time Tradeoffs in Autonomous Sampling for Marine Robotics. In *Proc. Int. Symp. Experimental Robotics*.

Hitzler, P.; Krötzsch, M.; Parsia, B.; Patel-Schneider, P. F.; and Rudolph, S., eds. 2009. *OWL 2 Web Ontology Language: Primer*. W3C Recommendation. Available at *http://www.w3.org/TR/owl2-primer/* (Nov 2014).

Hoffmann, J., and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *Journal of AI Research (JAIR)* 24:519–579.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *Proc. Int. Conf. on Tools with AI (ICTAI)*, 294–301.

Ingrand, F.; Chatilla, R.; Alami, R.; and Robert, F. 1996. PRS: a high level supervision and control language for autonomous mobile robots. In *IEEE Int.l Conf. on Robotics and Automation*.

Kavraki, L. E.; P. Svestka, J.-C. L.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Trans. on Robotics and Automation*.

Kortenkamp, D., and Simmons, R. G. 2008. Robotic Systems Architectures and Programming. In *Springer Handbook of Robotics*. Springer. 187–206.

Lemai-Chenevier, S., and Ingrand, F. 2004. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Magazzeni, D.; Py, F.; Fox, M.; Long, D.; and Rajan, K. 2014. Policy learning for autonomous feature tracking. *Autonomous Robots* 37(1):47–69.

McGann, C.; Py, F.; Rajan, K.; Thomas, H.; Henthorn, R.; and McEwen, R. S. 2008. A deliberative architecture for AUV control. In *Proc. Int. Conf. on Robotics and Automation (ICRA)*.

Muscettola, N.; Dorais, G.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. IDEA: Planning at the Core of Autonomous Reactive Agents. In *Proc. of AIPS Workshop on On-line Planning and Scheduling*.

Palomeras, N.; El-Fakdi, A.; Carreras, M.; and Ridao, P. 2012. COLA2: A Control Architecture for AUVs. *IEEE Journal of Oceanic Engineering* 37(4):695–716.

Piacentini, C.; Alimisis, V.; Fox, M.; and Long, D. 2013. Combining a Temporal Planner with an External Solver for the Power Balancing Problem in an Electricity Network. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Ponzoni, C.; Chanel, C.; Lesire, C.; and Teichteil-Knigsbuch, F. 2014. A Robotic Execution Framework for Online Probabilistic (Re)Planning. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Py, F.; Rajan, K.; and McGann, C. 2010. A systematic agent framework for situated autonomous systems. In *Proc. of Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)*.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3.

Ribas, D.; Palomeras, N.; Ridao, P.; Carreras, M.; and Mallios, A. 2012. Girona 500 AUV: From Survey to Intervention. *Mechatronics, IEEE/ASME Transactions on* 17(1):46–53.

Simmons, R. 1992. Concurrent planning and execution for autonomous robots. *IEEE Control Systems* 12(1):46–50.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel., P. 2014. Combined Task and Motion Planning through an Extensible Planner-Independent Interface Layer. In *Proc. Int. Conf. on Robotics and Automation (ICRA)*.

Tenorth, M., and Beetz, M. 2009. KnowRob – Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4261–4266.

Tenorth, M.; Bartels, G.; and Beetz, M. 2014. Knowledge-based Specification of Robot Motions. In *Proc. European Conf. on AI (ECAI)*.

Tenorth, M.; Kunze, L.; Jain, D.; and Beetz, M. 2010. KNOWROB-MAP - knowledge-linked semantic object maps. In *Humanoids*, 430–435.