ROS-POMDP – A Platform for Robotics Planning using PLPs and RDDL in ROS

Or Wertheim and Ronen I. Brafman and Shashank Shekhar and Tal Feiner and Igal Pinsky
Ben-Gurion University of the Negev, Beer Sheva, Israel
{orwert,brafman,shekhar,talfe,pinsky}@post.bgu.ac.il

Abstract

ROS-based software provides many basic skills for robotics, including navigation, arm-movement, mapping, object recognition, and more. Yet, there is little support for task-level autonomy, with the ROSPlan platform being the major exception. ROSPlan supports planning and plan-execution within ROS. Originally, for deterministic, fully observable models, but more recently, for contingent planning, and limited types of probabilistic planning. More specifically, most contingent planners assume deterministic actions and sensing, and ROSPlan's support for partial observability is limited in scope. ROS-POMDP attempts to fill this gap. It builds on the more realistic POMDP model, with stochastic actions and sensing, and seeks to make it very easy for roboticists to replace hand-written scripts/controller with principled POMDP-based controllers. This paper describes ROS-POMDP and our initial experiments with its use.

INTRODUCTION

Most autonomous robotic systems are built by utilizing software components, locally written, or imported, each of which handles a particular capability. More sophisticated behavior is then obtained by combining these behaviors in various ways using an adequate controller. The problem with such pre-written controllers is their brittleness. In some sense, their designer is expected to foresee all possible scenarios and tasks the robot will face. Instead, the planning-based approach uses an online planner to select which component is to be activated and when. Online planners are able to react to new events, and they can often generate near-optimal behavior that a human designer would find difficult to synthesize on her own.

The ROSPlan platform (Cashmore et al. 2015) was a major development in this area. It provided tools and a methodology for supporting the use of planners in ROS-based systems (Quigley et al. 2009). The ROSPlan platform supports both planning and plan-execution of ROS-based software, and provides support for a rich set of planning formalism: classical, temporal, contingent, and most recently, stochastic planning. The latter two models are especially important because realistic robotics application involve uncertainty and partial observability – key elements in this model. But the contingent planning model has many weaknesses. First, its notion of a *goal* is too weak to specify the diverse objectives we often have, and their relative tradeoffs. Second, and related, contingent planning models uncertainty using non-determinism. Non-determinism is fine when one seeks to find a solution that is guaranteed to achieve a certain goal. But, if the goal is not always reachable, or can be costly to reach, one would like to make tradeoffs between likelihood of success of different plans and their cost. Moreover, given limited computational resources, one would prefer to invest effort in more likely scenarios. This is not possible within the contingent model.

Instead, most work in robotics has focused on the POMDP (partially observable Markov decision process) model (e.g., (Hanna Kurniawati 2008; Seiler, Kurniawati, and Singh 2015; Chen et al. 2016). POMDPS capture partial observability, stochastic actions, stochastic sensing, and complex goals expressed using a reward function.

POMDP algorithms support this full range of behavior, and recent sampling-based solution methods for POMDPS have been able to scale up to real-time behavior in very complex domains (Silver and Veness 2010). Finally, statistical estimation techniques and reinforcement learning algorithms can help us improve the model with experience.

Indeed, recently, the ROSPlan platform has been extended to allow, standardized integration of RDDL and ROSPlan, enabling the straightforward application of probabilistic planners in robotic domains using ROS. In addition, there is a ROSPlan interface with any RDDL planner that can be used with the RDDLSim server used in the IPPC (Canal et al. 2019). However, this extension allows actions with non-deterministic effects only if this uncertainty is on *sensed* state variables. These sensed variables are state variables for which there is an action that can deterministically sense their value. If we combine the supported probabilistic action with their sensing actions, the combined action is non-deterministic, but its outcome is fully-observable. Hence, this model is a weak version of POMDP. Indeed, in ROSPlan's Online Planning and Execution with RDDL Planners, the architecture is a Client/Server architecture similar to that of the International Probabilistic Planning Competition (IPPC). ROSPlan's knowledge base holds the server state, and even though planning considers probabilistic effects and rewards, the state is always observable to ROSPlan. In this paper we describe the ROS-POMDP framework, which supports the full POMDP model.

Planners, however, are not a silver bullet. They are as good as their model is, and in robotics we have to model complex temporal components that perform navigation, manipulation, object detection etc. Their rich behavior is not always easy to model using standard planning languages. For this reason, we have been developing the language of *Performance Level Profiles* (PLPs) (Brafman, Bar-Sinai, and Ashkenazi 2016), a language for specifying the expected behavior of functional components. PLPs describe a number of key aspects of the performance of functional modules. They combine ideas from planning language (PDDL 2.1 (Fox and Long 2003), probabilistic PDDL (Younes and Littman 2004), RDDL (Sanner 2010)), achievement and maintenance goals (Ingrand et al. 1996; Kaminka et al. 2007), and new notions such as progress measures and a *repeat* construct aimed at making explicit the frequency by which input parameters are read and output parameters are published. Unlike action languages that limit their expressiveness to meet the requirements imposed by state-of-the-art planning technology, PLPs seek to provide expressiveness that can be used for other tasks. Thanks to their structured, machine readable syntax, PLPs can be manipulated automatically for the purpose of online monitoring (Brafman, Bar-Sinai, and Ashkenazi 2016), validation, and planning (Ashkenazi, Bar-Sinai, and Brafman 2016a).

In this paper we describe the ROS-POMDP platform. ROS-POMDP provides support for POMDP-based planning, and in particular, Monte-Carlo-based planning in POMDPs, where the basic "actions" correspond to calls to functional code documented using PLPs or RDDL. Given a PLP describing each component, ROS-POMDP automatically generates an RDDL description of the component as well as a java-based simulator of the component. The RDDL code can be used as input to diverse, offline and online, POMDP solvers, while the JAVA simulator provides the service of sampling a random effect of each action. Using this component, ROS-POMDP runs the POMCP algorithm (Silver and Veness 2010) to solve the planning problem online. Each time it selects the next action, it executes the corresponding code, and continues the process. The reward function for the solver is build based on the users specified goal, but it can be combined with diverse other objectives (e.g., safety, speed) easily.

In the rest of this paper we provide some background on PLPs, POMDPs and MCTS-based methods for their solution, we describe the architecture of the system, and we describe an initial empirical evaluation conducted in simulation on a ROS-based mobile robot with an arm. We are currently working on testing the system on the real-world Armadillo robot used in this simulation.

Background

We briefly describe PLPs (Brafman, Bar-Sinai, and Ashkenazi 2016), POMDPs (Kaelbling, Littman, and Cassandra 1998), and Monte-Carlo-based planning for POMDPs, as used in the POMCP planner (Silver and Veness 2010).

RDDL

The Relational Dynamic Influence Diagram Language (RDDL) (Sanner 2010) is a language for describing the evolution of structured fully or partially observed (stochastic) processes. In this language, it is assumed that states, actions, and observations (whether discrete or continuous) are parameterized variables and the system's evolution is specified via (stochastic) functions over next state variables conditioned on current state and action variables. Parameterized variables are simply templates for ground variables that can be obtained when given a particular problem instance defining possible domain objects. Because actions are described by means of variables, the language supports concurrent execution of actions, too.

Semantically, RDDL is simply a dynamic Bayes net (DBN) (Dean and Kanazawa 1989) (with potentially many intermediate layers) extended with a simple influence diagram (ID) (Howard and Matheson 2005) utility node representing immediate reward. An objective function specifies how these immediate rewards should be optimized over time for optimal control. For a ground instance, RDDL is just a factored MDP (or POMDP, if partially observed).

POMDP

A partially observable Markov decision process (POMDP) is a model for decision making under uncertainty and partial observability.

Formally, it is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, O, \gamma, \mathcal{I} >$

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the actions set.
- $\mathcal{T}$ is the state transition function, $\mathcal{T}(s, a, s')$ is the probability to reach $s' \in \mathcal{S}$ from $s \in \mathcal{S}$ using $a \in \mathcal{A}$.
- $\mathcal{R}$ is the reward function, $\mathcal{R}(a, s)$ is the reward for applying $a \in \mathcal{A}$ from state $s \in \mathcal{S}$.
- Ω is the state observation space.
- $O(s, a, o)$ is the probability to see observation $o \in O$ after applying action $a \in \mathcal{A}$ and reaching state $s \in \mathcal{S}$.
- $b_0 \in \Pi[\mathcal{S}]$ is a distribution over $\mathcal{S}$, that represents the initial state.

Modeling robot planning problems using POMDPs is natural since a robot actions' outcomes are usually not deterministic, the state of the world is usually not fully observable to the robot and robots' sensors are typically noisy.

Because the agent's observations provide partial information, it typically does not know the state it is in. Instead, it maintains a belief state $b \in \Pi[\mathcal{S}]$, i.e., a distribution over the state space, starting from an initial belief b_0. Using Bayes' rule, the agent can compute a new belief state b' given its previous belief state b, the action excuted, a, and the observation received, o.

This exact computation, when the state space is large, is infeasible. In fact, just representing one's belief state explicitly, i.e., maintaining an explicit distribution over a large state space, in infeasible. Instead, one usually uses an approximate representation of b using a *particle filter*. A particle filter $\hat{b}_k$ consists of some k particles, or states $s_1, \ldots, s_k$. $\hat{b}_k$ represents the following distribution over states: $Pr(s) = \frac{1}{k}\Sigma_{i=1}^{k}\delta_{s_i,s}$ where $\delta_{i,s}$ is the Kronecker delta function. The initial set of particles is sampled from the initial belief state, b_0, and the update function uses the action and the observations to sample new particles/states given the old particles. This approximation method approaches the true belief state with sufficient particles, $lim_{k\to\infty}\hat{b}_k(s) = b(s)$

POMCP

POMCP (Silver and Veness 2010) uses Monte-Carlo Tree Search (Tesauro and Galperin 1997) to solve large POMDPs online. The algorithm combines Monte-Carlo update of the agent's belief state with Monte-Carlo tree search from the current belief state. POMCP has two important properties that enable it to plan effectively in significantly larger POMDPs than has previously been possible. First, Monte-Carlo sampling is used to break the curse of dimensionality both during belief state updates and during planning. Second, only a black box simulator of the POMDP is required, rather than explicit model. Many realistic domains have very large state space, making an explicit POMDP model difficult to formulate and manipulate, whereas a simulator is much easier to write.

In Monte-Carlo planning, the agent uses a simulator G as a generative model of the POMDP. The simulator provides a sample of a successor state, observation and reward, given a state and action: $(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t)$. It can also be reset to a

start state s. The simulator is used to generate sequences of states, observations and rewards. These simulations are used to update the value function, without ever looking inside the black box describing the model's dynamics. In addition, Monte-Carlo methods have a sample complexity that is determined only by the underlying difficulty of the POMDP, rather than the size of the state space or observation space (Silver and Veness 2010)

PLPs

The primary objective of a PLP is to clarify the role and expected/normal behavior of a module. There are four PLP types, corresponding to four module types. *Achieve* modules attempt to achieve a new state of the world or generate a new object. For example, changing the orientation of the robot to some goal orientation. *Maintain* modules attempt to maintain some property. For example, maintaining some orientation; or, ensuring that the robot remains within some confined area. *Observe* modules attempt to recognize some property of the current state of the world. For example, the robot's location, or whether there is a cup on the table. Finally, *Detect* modules monitor the state of the world until some condition holds.

PLPs XML documents. Each document must conform to an XML Schema Definition (XSD) that defines the syntax of PLPs, with one XSD for every PLP type. The schema can be found in https://github.com/PLPbgu/PLP-repo together with an example of a PLP of each type. Below we provide an informal description of the information contained in the respective XML/XSD documents.

PLPs have two abstract components. The second component specifies the code's expected behavior – its "guarantees": what success means, possible failure modes and their probabilities, a distribution over running times, progress rates, and various statistical invariants. The first component provides the conditions under which the "guarantees" are valid: properties of the world before and during execution and constraints on available resources. These properties are necessarily observable by the robot. For example, a sensor may guarantee normal operation under some temperature range, independent of whether the robot has a thermostat.

The formal definition of PLPs rests on the specification of properties of states of the world. These are defined by specifying properties of various state variables. In addition, each module may need access to certain resources. These resources could be energy or memory, some actuator, or some region of space. These must be specified, much like state variables, and coherent and consistent use of these names is required. In fact, resources can be viewed as a special class of state variables, whose state indicates the status of the resource (e.g., available, > 100 gallons, etc.). However, because they carry special significance to programmers and operators, we distinguish them from other variables.

Common Elements All modules specify the following elements: *Parameters* (values supplied to the module as input or provided by the module as its output), *local variables and their ranges*, and the following set of conditions specifying the contexts in which the PLP is valid: required resources, optional bounds on the maximal rate of change for resources, concurrency conditions that must hold at execution time, invariants, other code modules that must or must-not be executed concurrently, and the frequency by which each parameter must be read or written (optional).

Each module has an intended effect, or role. However, it may also have side-effects that are a result of executing this module, but are not a measure of its success or failure. Resource consumption is a primary example. In addition, modules that perform continuous work to achieve or maintain their goals may specify a minimal rate of change per time unit. For example, the rate of change of a position while navigating. Making these expectations explicit makes it easier to recognize problematic behaviour while the module executes.

PLP Types *Achieve* modules attempt to reach a state of the world in which some desirable property holds. For example, fuel tank is full, robot is standing, plane has landed, etc. *Achieve* also covers cases where the goal is to generate some virtual object, such as a map or a path. Beyond the common elements, their PLP contain an *the achievement goal*, *failure modes*, *probabilities* associated with success and each failure mode, and the *running-time distribution* given success and given failure.

Maintain modules attempt to maintain the value of a variable or the truth value of a boolean condition, e.g., maintain speed or maintain perimeter clean. The condition need not be true initially, and

so the module may need to initially attain the condition. It may also become false during execution and regained, as in the case of a cleaning robot. This is reminiscent of a closed-loop controller that always attempts to decrease some distance to the desired goal condition. PLP of *maintain* modules contains: the *condition* to be maintained, whether it is *initially true*, *termination conditions*, one for successful termination (optional) and one for failure, failure modes, the *probability* of successful termination and different failure modes, and the *runtime distribution* given success and failure.

Observe modules attempt to identify the value of some variable(s) or a Boolean state condition, e.g., distance to wall or whether an object is held. *Observe* PLPs contain additional fields for the *observation goal*, the *probability* of failure to observe, the *probability* the observation is correct or some form of error specification, such as confidence interval and confidence level, and the *running-time* distribution given success and given failure.

Detect modules attempt to identify some condition that is either not true now, or that is not immediately observable. For example, detect intruder or detect temperature change. Their PLPs contain additional fields for the condition being detected, and the *probability* the condition will be detected given that it holds (*true* positive) and given that it does not hold (*false* positive).

PLP Glue files

Glue files connect PLP file meta-data concepts to code module operation components. They may contain mappings between PLP parameters to their location in ROS (e.g., the topic), or required imports, in order to work with the needed messages and classes (Brafman, Bar-Sinai, and Ashkenazi 2016).

The ROS-POMDP Platform

ROS-POMDP is part of our plug-n-play vision for operating autonomous robots. The robot user can write new software, or import an available package, and as long that these packages come with machine-readable documentation in the form of a PLP and RDDL, ROS-POMDP will be able to plan and execute behaviors that utilize this software in order to autonomously operate the robot and reach the user's goals.

To support true plug-n-play, documentation of different packages must use compatible terminology and consistent modeling choices, and stronger code-generation support is still required. We started addressing the first issue by providing PLPs specification tools that maintain a library of variable names. However, stronger consistency is needed and calls for more powerful tools and the creation of appropriate ontologies. In earlier work in the context of classical planing using ROSPlan, we provided tools for autogeneration of code (Ashkenazi, Bar-Sinai, and Brafman 2016b), and we intend to extend them to support our current system.

To provide the automated planning functionality, ROS-POMDP takes the RDDL or PLP files and uses them to create a POMDP model of the problem. More precisely, it uses these files to create a black box, generative model of the POMDP. It then solves the POMDP using a POMCP based online *solver* that continuously outputs the best next action to execute, dispatches it for execution by the robot, receives an observation returned from that action, updates its state, and repeats. ROS-POMDP translates actions received from the *solver* to an activation call to the corresponding code module. The code module may return a response, that is translated to an observation delivered back to the *solver* updating its current belief state. This execution loop continues until *goal reached* observation is received and the process ends. ROS-POMDP currently supports online planning using Monte-Carlo sampling. Domains with parameterized actions, parameterized boolean state variables, and discrete observations.

We now describe in more details the components of the system and its operation:

Solver Our current *solver* module supports a version of the POMCP algorithm, but support for other planners should be easy to add.

Simulator The *simulator* module contains both the black box generative model of the POMDP, and an initial state generator that can sample initial states from the POMDP initial state distribution. It is needed for the creation of the particle filter that approximates the initial belief state. The POMDP may be modeled using either PLPs or RDDL. When PLPs are used, we automatically generate the Java *simulator* module using our *PLP con-*

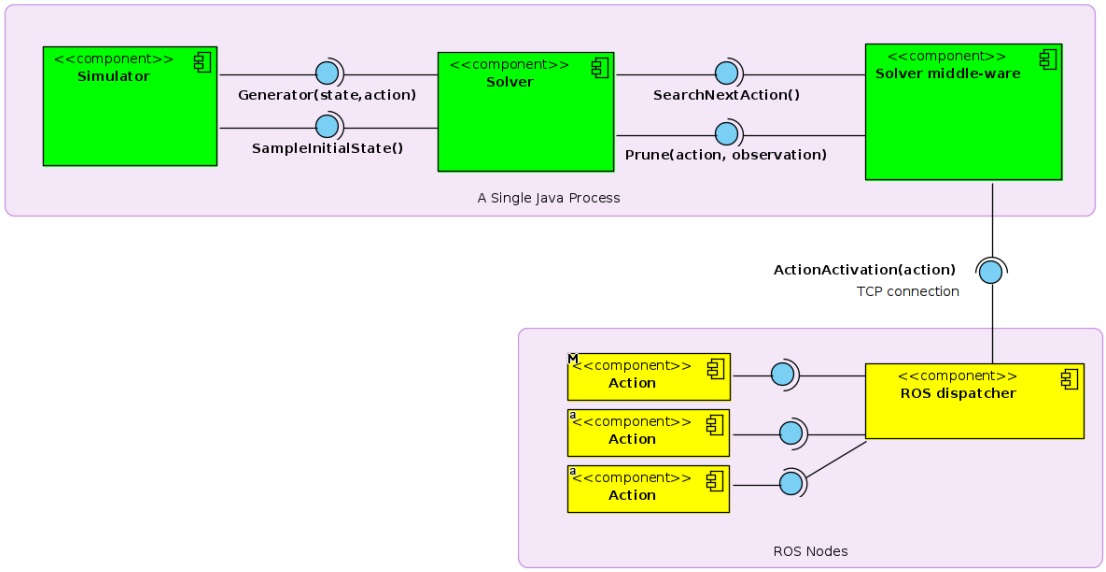

Figure 1: ROS-POMDP Components and Interfaces

verter module. This module can also generate an RDDL description from the PLP. When POMDP is modeled using RDDL, the *simulator* module is based on a version of Scott Sanner's Java server implementation for the International Probabilistic Planning Competitions (IPPCs). When comparing the efficiency of these *simulator* modules on an identical POMDP, the RDDL *simulator* black box generated 957 samples per second, while the PLP *simulator* generated 82,844 samples per second. In addition, we allow the user to define the initial state using a PLP *Environment file*, because, unlike RDDL, it allows probabilistic initial states.

Action Dispatching For communication between the *solver* and ROS, we use the *solver middleware* and *ROS dispatcher* modules. Messages are transferred using TCP. *ROS dispatcher* translates actions received to activation call to the corresponding code module.

Observation Dispatching ROS services responses are converted by the *ROS dispatcher* to valid observations, sent to the *solver middle-ware*, and from it to the *solver*. When using RDDL The *ROS dispatcher* will convert each ROS service response to a symbolic observation, known to the *solver*. The number of observation types that can be defined is unlimited (e.g., "success", "failure: lack of fuel", "partial success", etc..). When PLPs are used, the *ROS dispatcher* will convert the ROS services response to symbolic observation, known to the *solver*. But at present, only pre-defined fixed types of observations are supported for PLPs ('success' or 'failure' for *Achieve* and 'failure', 'success, observed false' or 'success, observed true' for *Observe*).

In both PLPS and RDDL, an additional special observation of 'invalid action' is sent when a service detects that the pre-conditions of a requested action are not met.

Generating the Model When working with RDDL, ROS-POMDP receives as input the RDDL domain and instance files. The domain file must contain an action-fluent for each ROS action. If a ROS service returns an observation, an observe-fluent should be defined, and some naming conventions must be followed to allow for the translation.

When using PLPs, each PLP file defines a single POMDP action, while environment objects and state variables are defined in the newly introduced PLP XML *Environment file*. In the future, we intend to automate the generation of some of these

elements.

PLPs support domains with parameterized actions and parameterized state variable (predicates) as in RDDL. Each parameter has a type, and only objects from that type can be used to ground it. The *Environment file* has an *objects declaration* section that allows us to define typed objects so they can be used as parameters for actions and/or state variables.

PLPs also support preconditions, which are not typically supported by POMDP solvers. Indeed, it may be the case that some action a is optimal for most states in the agent's current belief state, but there is some probability that a's preconditions are not satisfied. Ruling a out completely in such a belief state is too restrictive. Instead, the model ROS-POMDP constructs associates a fixed penalty (negative reward) with the application of an action in a state that does not satisfy its preconditions. This allows the planner to trade off the benefits of applying the action on legal states with the disadvantage of applying it in an illegal state.

ROS-POMDP also support intermediate variables, in the spirit of RDDL. Intermediate variables were added because PLPs, like RDDL, capture information that cannot be described by a two-layer dynamic bayesian network, such as correlated post-action values. Other aspects, such as side effects, cost, success probability, etc., are taken from the respective fields in the PLP. Finally, both RDDL and PLPs support an initial state and goal specification formats.

Examples for modeling using RDDL, PLP and RDDL with a PLP *Environment file* can be found on (tttMaor Ashkenzi, Bar-Sinai, and Brafman) https://github.com/bguplp/ROS-POMDP-Examples. Naturally, the more accurate the PLP model of action effects and observations, the better will a robot using ROS-POMDP be able to achieve its goals.

Empirical Validation

Currently, due to the challenges of COVID-19, our work is focused on using ROS-POMDP to solve simulation models of a ROS operated, autonomous service robot in the Gazebo 3D simulation platform. The experiments test ROS-POMDP's ability to effectively find appropriate actions online, receive observations, perform appropriate belief update, and achieve its goal.

The set up the experiment, having implemented or imported various low-level actions, we first had to describe the action's modules by PLP, writing the *Environment file* description, and adding code to the *ROS dispatcher* module that translates a planner' invoke-action message to an appropriate module activation, and translating back the module' response to an observation the planner can read. This is the effort needed from a programmer using ROS-POMDP, and we hope to reduce the latter elements farther in the future using automated code-generation.

Figure 2: ARMadillo service robot

ARMadillo

The robot used is an ARMadillo service robot (see Figure 2) (of which we have two in our lab). Its dimensions are H 110-150 x W 50 x L 50 cm, its weight is 80 kg, and has a torso joint range of 400 mm Vertical prismatic. It is equipped with a Head HD camera mounted on Pan-Tilt system, skid drive steering with a max speed of 1 m/s, designed to operate on flat surfaces, a laser scanner range 2000°/4 m or 30 m, a GPS, a front sonar, a microphone and speaker, Intel i7 CPU, 16 GB of RAM, 156 GB of SSD storage, 10.1 inc multi-touch touch screen, and runs Ubuntu 16.04 OS with ROS Kinect. The ARMadillo has an arm with a max payload of 2.3 kg, a wrist RGB-D camera, an arm joints feedback of position / velocity and effort, and a two fingers arm gripper with position and torque control.

Experiment Layout

We created two duplicate Gazebo worlds, simulating our office floor. We located the ARMadillo near the elevator. Tables were positioned at the corridor, auditorium, and outside lab 211. In 'world 1' we

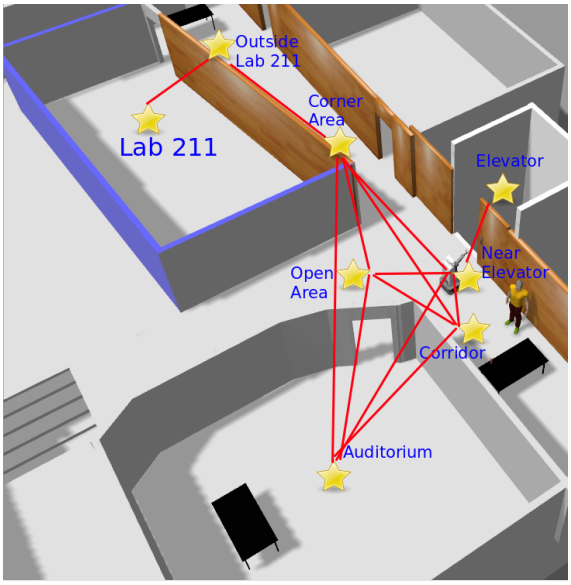

Figure 3: Experiment Layout, stars are used to mark the discrete locations, navigation is only possible between locations that are *connected* with a red line

placed a can on the corridor table, while in 'world 2' the can was placed on the table outside lab 211 (See Figure 3). Our goal is to have the can at the auditorium table, and the robot in its original location.

As in POMDPs, ROS-POMDP only knows its original belief state (0.6 chance of 'world 1' and 0.4 chance of world '2'), and receives the observations resulted by its actions. Any other information is not observable.

Actions

We implemented the following actions for AR-Madillo, often using well-known ROS packages. *Sense* observes if the can is located at the robot's location. *Pick* can be applied when the robot has an adjacent pickable object. *Place* places an object on an adjacent table. *Push button* is used to open the elevator door (applicable from within or outside of the elevator). Finally the *Navigation* action is used to navigate between connected discrete locations (as marked in Figure 3). Estimated actions success probability is 0.93, except for *Sense* that succeeds 99% of the time. When *Sense* is executed successfully, its observation is accurate with probability 0.8 - that it, it is a noisy sensor. Each action returns a boolean value indicating whether it succeeded or not. If the *Sense* action succeeds, its observation tells us if the can was observed or not. *Pick* returns "invalid" if the robot was not near the can when it was applied.

Modeling

We used PLPs with an *Environment file* to model the POMDP actions and domain. We believe PLPs are more appropriate as a documentation method for code. Second, because the PLP-based simulator is much faster than the RDDL one.[1]

An *Environment file* was created, describing the planning domain (e.g., state variables, typed objects, etc.) and specific problem details (e.g horizon, discount, initial belief state, goal, etc.) For each action module, a PLP model was created, describing its parameters, preconditions, effects and observations.

The experiment layout map can be seen in Figure 3.

Reward Model

POMDP solution attempt to maximize expected reward. The reward model we used is as follows: Each action was assigned with a cost (negative reward). The costs were, *Sense* -120, *Push button* -160, *Navigate* -120, *Pick* -100, *Place* -100. Reaching the goal results with a reward of 1400, and executing an invalid action – i.e., one whose preconditions are not satisfied in the current true state – causes a negative reward of -3400.

Results

We used the *PLP Converter* to generate a *simulator* module and we used a particle filter with 80 particles. For each decision, ROS-POMDP was given 10 seconds to plan, which is negligible in respect with the execution time of each action.

We briefly describe some of the plans generated, so one can see how the use of POMDPS allows us to address the issue of partial observability with noisy sensing. Videos of the experiments can be found on https://youtu.be/2DjMdOZUruk (world 1) and https://youtu.be/w3F5vhDF6oI (world 2).

[1]Of course, this is not an inherent property of the representation, simply a byproduct of the more efficient simulator generator we wrote.

The planning phase is identical to that of POMCP (Silver and Veness 2010), which our code implements, and we expect ROS-POMDP to behave similarly.

World 1 Experiment:

1. Navigate to corridor was successful.
2. A sense action was executed 3 times. The first sense action yielded a noisy observation indicating that the can is not there. The other two observations were accurate. We can see here that the planner takes into account the accuracy of the sensor, and for this reason, does not rely on a single observation.
3. The can was picked successfully.
4. Navigation to the auditorium was successful.
5. The can was placed on the table, successfully.
6. Navigation to elevator area succeeded, and the goal was reached.

World 2 experiment:

1. Navigate to corridor was successful.
2. A sense action was executed 3 times. All observations were correct – indicating that the can is not there.

From this point on, all actions were performed successfully: Navigate to corner area, navigate to outside lab 211, pick can, navigate to corner area, navigate to auditorium, place can, navigate to elevator area (goal).

Summary and Future Work

In this work we introduced ROS-POMDP, our new planning and execution framework for ROS. The platform's main innovation is its support for full-fledged POMDPs and for their documentation using PLPs. The platform was validated in a simulation of an autonomous service robot task with some partial observability. The experiment showed that ROS-POMDP configuration can handle multiple real world non-observable scenarios, operating the robot until reaching its goal. Robustness was demonstrated in the planner's response to action failure, demonstrating the power of probabilistic planning. We believe that the strengths of the POMDP model will be essential for good real-world performance: sensing in the real world is noisier than in simulation, and failures in the real-world are more common, and the POMDP model is able to make more informed choices than models that are currently supported by other systems.

In future work we intend to enhance the ROS-POMDP platform to support model learning, using reinforcement learning techniques, and provide additional automated code-generation to farther reduce the need for integration code by users of the system.

Acknowledgements We thank the reviewers for their useful comments. This work was supported by ISF Grants 1651/19, by the Israel Ministry of Science and Technology Grant 54178, and by the Lynn and William Frankel Center for Computer Science, and by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Center of Ben-Gurion University of the Negev.

References

[Ashkenazi, Bar-Sinai, and Brafman 2016a] Ashkenazi, M.; Bar-Sinai, M.; and Brafman, R. I. 2016a. Planning and monitoring with performance level profiles. In *ICAPS'16 Workshop on Planning and Robotics (PlanRob)*.

[Ashkenazi, Bar-Sinai, and Brafman 2016b] Ashkenazi, M.; Bar-Sinai, M.; and Brafman, R. 2016b. Planning and monitoring with performance level profiles. In *ICAPS'16 Workshop on Planning and Robotics (PlanRob)*.

[Brafman, Bar-Sinai, and Ashkenazi 2016] Brafman, R. I.; Bar-Sinai, M.; and Ashkenazi, M. 2016. Performance level profiles: A formal language for describing the expected performance of functional modules. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1751–1756. IEEE.

[Canal et al. 2019] Canal, G.; Cashmore, M.; Krivić, S.; Alenyà, G.; Magazzeni, D.; and Torras, C. 2019. Probabilistic planning for robotics with rosplan. In *Annual Conference Towards Autonomous Robotic Systems*, 236–250. Springer.

[Cashmore et al. 2015] 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 *Twenty-Fifth International Conference on Automated Planning and Scheduling*.

[Chen et al. 2016] Chen, M.; Frazzoli, E.; Hsu, D.; and Lee, W. S. 2016. Pomdp-lite for robust robot planning under uncertainty. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, 5427–5433.

[Dean and Kanazawa 1989] Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computational intelligence* 5(2):142–150.

[Fox and Long 2003] Fox, M., and Long, D. 2003. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research* 20:61–124.

[Hanna Kurniawati 2008] Hanna Kurniawati, David Hsu, W. S. L. 2008. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Proceedings of Robotics: Science and Systems IV*.

[Howard and Matheson 2005] Howard, R. A., and Matheson, J. E. 2005. Influence diagrams. *Decision Analysis* 2(3):127–143.

[Ingrand et al. 1996] Ingrand, F.; Catilla, R.; Alami, R.; and Robert, F. 1996. A high level supervision and control language for autonomous mobile robots. In 43-49., ed., *IEEE ICRA*.

[Kaelbling, Littman, and Cassandra 1998] Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101:99–134.

[Kaminka et al. 2007] Kaminka, G. A.; Yakir, A.; Erusalimchik, D.; and Cohen-Nov, N. 2007. Towards collaborative task and team maintenance. In *Autonomous Agents and Multi-Agent Systems*.

[Quigley et al. 2009] 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, 5. Kobe, Japan.

[Sanner 2010] Sanner, S. 2010. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University* 32:27.

[Seiler, Kurniawati, and Singh 2015] Seiler, K. M.; Kurniawati, H.; and Singh, S. P. N. 2015. An online and approximate solver for pomdps with continuous action space. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, 2290–2297.

[Silver and Veness 2010] Silver, D., and Veness, J. 2010. Monte-carlo planning in large pomdps. 2164–2172.

[Tesauro and Galperin 1997] Tesauro, G., and Galperin, G. R. 1997. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems*, 1068–1074.

[tttMaor Ashkenzi, Bar-Sinai, and Brafman] tttMaor Ashkenzi; Bar-Sinai, M.; and Brafman, R. I. *PLP-Repository*.

[Younes and Littman 2004] Younes, H. L., and Littman, M. L. 2004. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162* 2:99.