IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

# Plug'n Play Task-Level Autonomy for Robotics Using POMDPs and Probabilistic Programs

Or Wertheim[1], Dan R. Suissa[1], and Ronen I. Brafman[1]

*Abstract*—**We describe AOS, the first general-purpose system for model-based control of autonomous robots using AI planning that fully supports partial observability and noisy sensing. The AOS provides a code-based language for specifying a generative model of the system, making model specification easier and model sampling efficient. It provides a language for specifying the relation between the model and the code, using which it auto-generates all required integration code. This allows Plug'n Play behavior, which facilitates incremental and modular system design. Extensive experiments on real and simulated robotic platforms demonstrate these advantages.**

*Index Terms*—**AI-Enabled Robotics, Autonomous Agents, Integrated Planning and Control, Planning under Uncertainty, Software Architecture for Robotic and Automation**

## I. Introduction

INTEGRATING different *skills,* such as navigation, localization, object detection, manipulation, etc., into a working autonomous robot capable of performing diverse tasks remains a major challenge for robot programmers. One must provide a *behavior policy* that dictates which skill to use and when, and one must *integrate* these diverse software packages, properly passing data between them. Any solution must address both issues.

In most current systems, the behavior policy is specified manually using a script or a more structured format such as a state-machine (SM). But scripts and SMs are usually task and environment specific. So autonomous systems that must perform diverse tasks in diverse environments will constantly need new or modified scripts. Automated AI Planning [1] offers an alternative: use an *action description language* (ADL) to describe and document each skill's impact on the world, the current environment, and the goal; and let a planning algorithm automatically generate the behavior policy. This model-based approach has important benefits: Planners can optimize decisions, and their models are transparent and understandable. Integration remains a major software engineering task under both alternatives.

Until recently, most robotic systems that used planning algorithms were based on one-of-a-kind implementations in which off-the-shelf or dedicated planners controlled a robot. ROSPlan [2] was the first generic platform for using a classical planner to control ROS-based robots [3]. It was followed by additional systems seeking to make this task easier, most of which still require some integration effort. But, their fundamental weakness is their reliance on classical planning models that offer limited or no support for skills with stochastic effects, partial observability and noisy sensors – properties common to most mobile robots. Typically, they address uncertainty using replanning, a sub-optimal approach that reacts to, rather than anticipates unexpected events. Some model partial observability to some extent; none models noisy sensing.

POMDPs offer a principled, expressive approach to dealing with these issues. They model probabilistic uncertainty, partial observability, noisy sensing, and complex reward functions. Our main *engineering contribution* is the Autonomous Robot Operating System (AOS), an experimentally validated, general-purpose system for using POMDPs to control ROS-based robots. The first of its kind.

The main impediment to using a POMDP-based architecture to program autonomous robots is the effort of specifying the rich POMDP model, and the effort of integrating all components into a working system. The AOS addresses these challenges by making two *technical and conceptual contributions*: introducing the *abstraction mappings (AM)* formalism and using code-based generative models.

AMs provide a principled, structured format for specifying the relation between skill models and skill code. Using them, the AOS can auto-generate *all* needed integration code. This capability makes the AOS a *Plug'n Play* system – a term that refers to the ability to add components to a system, documented skills in our case, without additional user programming and integration effort. The Plug'n Play feature simplifies autonomous robot design by making *incremental and modular* development easy: one can focus on adding or modifying new skills. If the skills are properly documented, the system will take care of the rest.

Code-based generative POMDP models make model specification easier by letting programmers use code to specify model elements, treating them as components of a simulator. Code offers powerful constructs programmers know well, and the AOS exploits it to sample more efficiently from the model to obtain better decisions faster.

This paper describes the AOS's key concepts and illustrates its potential power via experiments and case studies. Our GitHub repository [4] offers additional technical information, system code, examples, and experiment videos.

## II. Background and Related work

The AOS relates to probabilistic planning in partially observable domains, as well as general-purpose autonomous

Manuscript received: July, 23, 2023; Revised September, 13, 2023; Accepted November, 9, 2023.

This paper was recommended for publication by Editor Tetsuya Ogata upon evaluation of the Associate Editor and Reviewers' comments. This work was supported by ISF Grant 1651/19, the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Initiative, by the Marcus Endowment Fund, and by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University of the Negev.

[1]The Authors are with the Department of Computer Science, Ben-Gurion University of the Negev, Israel.

Digital Object Identifier (DOI): see top of this page.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

robotic system architectures in which planning is a key component for control and, indirectly, for skill integration. We use *skill* and *action* interchangeably throughout.

**Classical Planning Models.** Classical models assume perfect information, deterministic skill outcomes, and reachability goals only. They cannot reason about contingencies, trade-offs between probability and reward, or the need to gather information. Even if replanning is used following each observation, decisions ignore future uncertainty and will not actively seek information. Contingent extensions exist, but they are non-probabilistic, goal-oriented and typically assume deterministic actions and sensing.

**POMDPs.** A discrete-time POMDP models the relationship between an agent and its environment as a tuple $\langle \mathbf{S}, \mathbf{A}, \mathbf{T}, \mathbf{R}, \Omega, \mathbf{O}, \gamma, \mathbf{I} \rangle$: $\mathbf{S}$ is the state space, $\mathbf{A}$ the action space, $\mathbf{T}$ the state transition model, $\mathbf{R}$ the reward model, $\Omega$ the observation space, $\mathbf{O}$ the observation model, $\gamma \in (0,1]$ is the discount factor, and $\mathbf{I} \in \mathbf{B}$ is the initial belief state. A *belief state* is a distribution over $\mathbf{S}$ that models the likelihood of each concrete world state based on available information.

Following an action $a \in \mathbf{A}$, the environment transitions from its current state $s \in \mathbf{S}$ to state $s' \in \mathbf{S}$, with probability $\mathbf{T}(s,a,s')$. Then, the agent receives an observation $o \in \Omega$, with probability $\mathbf{O}(s',a,o)$, and a reward $r = \mathbf{R}(s,a) \in \mathbb{R}$. Now, one can update its belief state $b$ to $b' = Pr(s|a,o,b)$ using the model parameters.

POMDPs are a natural model for robots acting in the world because they capture the stochastic nature of robots' actions, their noisy and partial sensing, and allow for diverse task specifications using the reward function.

A *policy* for a POMDP is a mapping $\pi : \mathbf{B} \mapsto \mathbf{A}$ from belief states to actions. The goal of POMDP solvers is to find a policy $\pi^*$ that maximize the expected accumulated discounted reward, i.e., $\pi^* = \underset{\pi}{max}[\mathbb{E}_\pi[\sum_{t=1}^{\infty} \gamma^t r_t.]$. $r_t$ is the reward at step $t$, discounted by $\gamma^t$, so that when $\gamma < 1$, receiving a reward earlier is preferred.

Offline POMDP solution algorithms such as SARSOP [5], generate a complete policy offline and typically expect explicit matrices specifying $\mathbf{T}, \mathbf{O}$. Online algorithms such as POMCP [6] compute the next action to execute, given the current state. They typically require a *generative* model only, i.e., a model that for every state and action can sample the next state, observation and reward correctly. Efficient sampling is important for their performance.

**Skill Models.** Planning-based control architectures view each robot's skill as an action template and use an ADL such as PDDL [7] to describes the skill's effect on the system's state. RDDL [8] is the best-known ADL for stochastic models. It uses dynamic Bayesian networks (DBNs) [9] to specify the transition and observation functions of a POMDP by describing the probability of state-variable values and observations after executing an action, *conditional* on their values before. Writing RDDL specs requires mastering their syntax, and complex distributions may be difficult to specify. Intermediate computations are accomplished by adding intermediate DBN layers, which may be tedious since a variable can only be assigned once. But RDDL also offers a broad set of mathematical and logical expressions for specifying the conditional probability of variable values.

Like more recent POMDP specification languages, such as POMDP.py [15] and POMDPs.jl [16], the AOS goes beyond RDDL to support the full expressive power of a probabilistic programming language (PPL) by using C++ code augmented with standard distributions to specify models. Among similar systems for robotics, only the AOS provides this expressive power. This enables simpler, more compact and readable specifications, as argued in [10]. Moreover, the AOS exploits this code to get more efficient sampling, while RDDL specifications must be parsed and sampled by a generic tool. Furthermore, code descriptions are debuggable whereas RDDL is more of a black box and difficult to use in highly complex domains. Finally, unlike RDDL, we can use code to specify open-world domains.

**Deliberative Autonomous Robotic Platforms.** ROSPlan [2] is an influential system that motivated much of our work. It offers a planning and execution architecture for robotics that generates plans based on a PDDL2.1 (or a subset of RDDL) documentation of ROS-implemented skills. It supports classical, temporal, and contingent planners (with PDDL) and probabilistic planning [11] (with RDDL [8]). It does not support partial observability and only maintains a single possible state of the world, updated during execution by sensing information. User code should invoke replanning if the current state contradicts the expected state. It supports active sensing actions invoked by the planner and passive sensing [11] that runs in the background, not invoked by the planner, and automatically updates state variables. Users must manually code mappings of planned actions to robot code activations. An action interface [12] was added to reduce this mapping effort in some cases. ROSPlan is tightly bound to ROS and supports ROS1.

Motivated by ROSPlan, ProbPlan [13] supports full RDDL, including concurrent actions, but only for MDPs (i.e., precise sensing of the entire state), and was only tested in simulations. Plansys2 [14] supports ROS2, focusing on PDDL domains. It auto-generates a behavior tree based on the solver-generated plan, which supports concurrent and durative actions. For each skill, the user must implement nodes that: a) update the state with start and end effects, b) check concurrent conditions and end effects, and c) activate the skill code. Neither system fully addresses the abstraction gap, partial information and noisy sensing. SkiROS [15] addresses the abstraction gap by auto-generating PDDL action descriptions from skill code. But requires a predefined ontology and skill code that follows strict patterns. Classical planning with replanning is used to schedule skills.

CLIPS Executive (CX) [16] is a robot execution and planning framework based on CLIPS, a rule-based engine for expert systems. It stores a predefined high-level plan as a goal tree. The engine uses the execution outcome to update the goal tree and select a new goal. Selected goals are achieved using a classical planner or a predefined sequence of actions. Each skill can insert facts into CLIPS, which the engine uses to monitor and validate plan execution asynchronously. CX support for non-deterministic skills is

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

| | AOS | ROSPlan | CLIPS Ex. | DSL | SkiROS | Plansys2 |
|---|---|---|---|---|---|---|
| **ADL** | Structured PPL | PDDL/RDDL | PDDL | PDDL | PDDL | PDDL |
| **Model Generation** | X | X | X | From DSL | Requires Ontology | X |
| **Conc. & Durative Actions** | X | ✓ | ✓ | ✓ | only durative | ✓ |
| **Probabilistic Effects** | ✓ | ✓(with RDDL) | X | X | X | X |
| **Partial Observability** | ✓ | X | X | X | X | X |
| **Noisy Sensing** | ✓ | X | X | X | X | X |
| **Plug'n Play** | ✓ | X | X | X | X | X |
| **Supported Platforms** | ROS (extendable) | ROS | ROS2 & Fawkes | ROS & ROS2 | ROS | ROS2 |
| **Automated Verification** | X | X | X | ✓ | X | X |
| **Sensing** | Active | Active+Passive | Active | Active | Active | Active |
| **Auto Mapping** | ✓ | Passive sensing only | X | X | X | X |
| **Execution Monitoring** | ✓ | X | ✓ | ✓ | X | Active |

TABLE I. System Comparison. **Model Generation**: Planning model generated automatically from code. **Plug'n Play:** System can use a new skill given its documented code, only. **Sensing:** *Active* = sensing based on skill return values. *Passive* = sensing processes run in the background, updating the state without explicit activation. **Auto Mapping:** Structured translation of action parameters into code parameters and low-level sensor to model-level data. **Execution Monitoring:** Model-level data triggers automated plan update.

limited to replanning. Moreover, it is tightly coupled to ROS2 and Fawkes frameworks.

DSL [17], [18] is a formal language and system for specifying robot skills within ROS1 and ROS2. DSL contains declarative aspects, automatically translated to PDDL and used for planning, and operative aspects for monitoring. It allows concurrent skill activation and auto-generates code to assist skill integration. Automatic execution of actions is not supported. DSL can output state machines that describe the domain and the generated controller. A model-checker can then be used to verify their properties.

Table I summarizes the key features of sufficiently mature existing systems. It highlights the specific advantages of the AOS: (1) Full support for AI planning with partial and noisy sensing and stochastic effects. 2) Explicit documentation of the mapping between more abstract planning models and skill code. 3) Plug-n-Play behavior. 4) Skill modeling using probabilistic program code.

As far as actual usage of planning-based platforms, CLIPS, which combines planning, goal reasoning and pre-scripted plans outperformed winners of the Robocup Logistics League [19]. ROSPlan was used in diverse research projects [20]. Yet, planning-based platforms are not widely adapted, perhaps because current operational autonomous robots focus on repetitive delivery tasks. But even in Robocup@home, with its diverse tasks, competitors do not use planners to the best of our knowledge. Practitioners are either not familiar with these systems, find them too complex to use, or not mature enough.

Recently, LLMs were touted as good decision makers (e.g., [21]), removing the need for formal models and planning algorithms. [22] convincingly argue that this is not the case even in simple classical models. However, as demonstrated by [23], their use in model construction is very promising and can greatly reduce modeling effort.

## III. System Overview and Concept

We describe the process of autonomous robot design using the AOS from the robot programmer's perspective. (The system perspective is described in Section IV.) Then, we describe its key step: documenting skill code, illustrating it using an example. Finally, we explain the semantics of this documentation, i.e., the POMDP model it specifies.

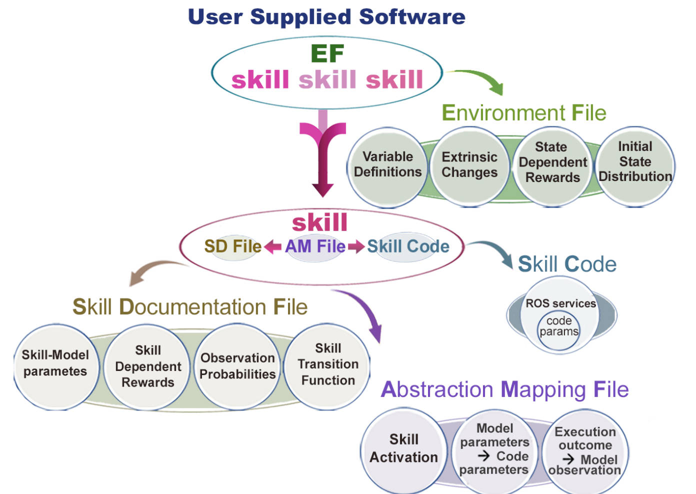


Fig. 1: Structure of User Supplied Content

### A. Using the AOS to Develop Autonomous Robots

The process starts by implementing or obtaining relevant skills. The AOS requires these skills to be documented in the *Skill Documentation Language* (*SDL*), an ADL that utilizes PPL code. SDL has two components: The *Skill Description (SD) file* describes how the code's execution impacts the robot's and world's state using statements in a PPL, specifically, C++ augmented with standard distributions. The *Abstraction Mapping (AM)* file connects the model-level description in the *SDs* with the skill's code. The content the programmer needs to provide the AOS is summarized in Figure 1 and described in detail below.

*SD* files specify generative transition, observation, and reward functions for a particular skill. They reside at a higher abstraction level than the actual code, more semantically meaningful to users. For example, the skill of picking up an object is usually described using propositions such as *holding-object* or *broken-object*. The actual code, however, manipulates variables denoting object locations and poses, gripper status, arm joints, and the relevant 3D free-space.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

The AM file connects the two. It describes how to activate the code, how to map skill-model parameters to skill-code parameters, and how to transform code parameters and return values to observations for the POMDP model.

To perform a specific task, the user must describe the current state of the system (or request to use the state of the last execution) and provide a reward function encoding the goal, using an *Environment File (EF)*. Then, one sends an HTTP request to the AOS Web API with the path to the documentation files. The AOS uses the documentation to auto-generate and compile a generative POMDP specification. The AOS then runs a POMDP solver that computes near-optimal behavior and uses it to control the robot – either POMCP [6], an online solver, or SARSOP [5], an offline solver. Since a good rollout policy significantly improves POMCP's generated policy, the user may specify one in the *SD* files, or let the AOS RL server auto-generate one by generating and solving a simplified model: the MDP obtained from the POMDP by assuming full observability.

During the development process, the user can add new skills or modify existing ones. The user sends a new HTTP request, and as long as the skills are properly documented, the AOS will regenerate the model and integration code. The user can also make modifications online during robot operation if the new documentation does not change the state variables. It must then stop for 30 seconds for the AOS to process the changes. Both with online and offline updates, there is no formal issue of consistency – a well-formed POMDP is generated. Semantic consistency between skills, original or added, such as usage of different variable names in different skill files, is the user's responsibility, although we hope that future integration of an ontology would help address this issue. Finally, the AOS also provides a GUI for inspecting belief states, debugging code, and more. See [4].

## *B. AOS Documentation*

We describe the SDL language using the following illustrative example: A robot with a single *navigate* skill must navigate as fast as possible to three known locations, preferably visiting location $v1$ before location $v2$. The robot's initial location is unknown: With probability 0.5, it is $v1$. Otherwise, with probability 0.8, it is $v3$. As it operates, there is a 5% chance that an occasional person will move the robot, in which case it loses its orientation.

The *navigate* skill may fail, causing the robot to lose orientation. After some experiments, we conclude that a) Navigating to the robot's current location causes orientation loss. b) There is a 10% chance of orientation loss even when navigating to a different location. c) The skill mistakenly reports success in 20% of the cases in which the robot lost orientation. d) When the robot loses orientation or starts navigating without knowing its location, the skill takes significantly longer to execute.

We describe the EF and the *navigate* SD and AM files for this example. While intentionally simple, it demonstrates all POMDP aspects, and the use of a code-based generative model of transition dynamics, observations, rewards, and initial belief state. For variable $x$, $state.x$ denotes its value before skill execution; $state_.x$ its value after any extrinsic event; and $state__.x$ its value after skill execution. State-variable values persist from one step to the other, except for those that are explicitly reassigned. Blue words mark start of a section. Sections may appear in any order, except project name, which appears first. Brown words specify section properties. Teal words further elaborate them. Reserved variable names are in red.

**Environment File (EF):** We first define the *EF*. A single EF must be specified for each task, and it contains the following parts: a) General POMDP properties: the discount factor and the planning horizon, i.e., how many steps ahead will the AOS search for the next action. Lines 2,3 define them to be 0.97 and 10. b) A list of state variables defining the system state. Variable types can be any C++ primitive, primitive vector, or compound type. Compound variable types or enums can be defined with C++ primitive types as the building blocks. In our example, we define one type for robot location and one type for visited location. Then, Lines 12-21 declare variables of these types.

```
1  project: example
2  horizon: 10
3  discount: 0.97
4  define_type: tLocation
5  variable: float x 0.0
6  variable: float y 0.0
7  variable: float z 0.0
8  variable: int discrete −1.
9  define_type:
       tVisitedLocation
10 variable: int discrete 1
11 variable: bool visited false
12 state_variable: tLocation robotLocation
13 state_variable: tVisitedLocation v1
14 code:
15 state.v1.discrete = 1
16 state_variable: tVisitedLocation v2
17 code:
18 state.v2.discrete = 2
19 state_variable: tVisitedLocation v3
20 code:
21 state.v3.discrete = 3
```

c) Constant definitions, e.g., locations $l_1, l_2, l_3$ in Lines 22-30.

```
22 const: tLocation l1
23 code:
24 state.l1.x = −1.01606154442; state.l1.y = 0.660750925541; state.l1.z
       =−0.00454711914062; state.l1.discrete = 1
25 const: tLocation l2
26 code:
27 state.l2.x = 0.00500533776358; state.l2.y = 0.640727937222; state.l2.z
       =−0.0014343261; state.l2.discrete = 2
28 const: tLocation l3
29 code:
30 state.l3.x = 0.986030161381; state.l3.y = 0.610693752766; state.l3.z
       =−0.00143432617188; state.l3.discrete = 3
```

d) A generative model of the initial belief state. That is, C++ code that can randomly sample an initial state. Line 31-32 describe initial uncertainty about the robot's initial position: [1:0.5, 2:0.1, 3:0.4]. In continuous use, the system can start a new task from its last belief state.

```
31 initial_belief:
32 state.robotLocation.discrete = AOS.Bernoulli(0.5) ? 1 : (AOS.Bernoulli
       (0.2) ? 2 : 3);
```

e) A generative model of extrinsic changes not invoked by a specific skill, conditioned on the previous state. Lines 33-34 describe a 5% chance of a person moving the robot.

```
33 extrinsic_code:
34 if (AOS.Bernoulli(0.05)) state_.robotLocation.discrete = −1;
```

f) Code for computing state-dependent rewards. Line 35-39 express our preference to visit point one before two, and Lines 40-44 express our goal of visiting all locations.

```
35 reward_code:
36 if (!state.v1.visited && state.
       v2.visited)
37 {
38   __reward=−50;
39   __oneTimeRewardGiven =true;}
40 reward_code:
41 if (state.v1.visited && state.v2.
       visited && state.v3.visited)
42 {
43   __reward =7000;
44   __isGoalState =true;}
```

***SD* File for each skill:** An *SD* file specifies a generative

model of the next state, skill-dependent reward, and observation, as a function of the current state, when this skill code is executed. The specification relates the pre and post execution values of state variables described in the *EF* and the possible observations. This model describes the code's behavior based on concepts meaningful to code users rather than code programmers. It contains three components: a) List of (model-level) parameters the skill receives. Line 2 states that a location parameter of type *tLocation* is the parameter of *navigate*. The skill can be called with any constant of this type declared in the EF (i.e., $l_1, l_2, l_3$ in our example). Alternatively, using code, users can define the possible parameter values in the SD file (not shown). b) A soft precondition, such as not calling *navigate* with its current position as its target (Lines 3-5). This is specified separately from the reward model (Line 23) since it is used to strengthen POMCP's rollout policy by excluding actions that violate these preconditions.

```
1 project: example
2 parameter: tLocation oDesiredLocation
3 precondition:
4 __meetPrecondition = oDesiredLocation.discrete != state.robotLocation.
      discrete;
5 violate_penalty: -10
```

c) The skill model describes a generative model of the post-action state variables' value, observation and reward, given the variables' previous values and the model parameters. The *navigate* skill's model (Lines 6-23) starts by describing the 10% probability of not reaching the desired destination (Line 7) followed by code that updates the robot's location after the action (Lines 8-13) and the list of visited locations (Lines 14-20). Line 21 describes the probability that the module will report failure when it actually failed (0.8). Otherwise, it reports success. Lines 22-23 describe a reward function. Failure cost is −10; navigating when disoriented cost is −5. Otherwise, *navigate*'s cost is 100 times the Euclidian distance. It is possible to access an array grouping all the state variables of a particular type using the *<type>Objects* variables as in the code below (e.g., line 16).

```
6 dynamic_model:
7 state__.robotLocation.discrete = ! __meetPrecondition || AOS.Bernoulli
      (0.1) ? -1: oDesiredLocation.discrete;
8 if(state__.robotLocation.discrete == oDesiredLocation.discrete)
9 {
10   state__.robotLocation.x = oDesiredLocation.x;
11   state__.robotLocation.y = oDesiredLocation.y;
12   state__.robotLocation.z = oDesiredLocation.z;
13 }
14 for(int i=0; i < state__.tVisitedLocationObjects.size();i++)
15 {
16   if(state__.tVisitedLocationObjects[i]->discrete == state__.
        robotLocation.discrete)
17   {
18     state__.tVisitedLocationObjects[i]->visited = true;
19     break;
20 } }
21 __moduleResponse = (state__.robotLocation.discrete == -1 && AOS.
      Bernoulli(0.8)) ? eFailed : eSuccess;
22 __reward = state_.robotLocation.discrete == -1 ? -5 : -(sqrt(pow(state.
      robotLocation.x-oDesiredLocation.x,2.0)+pow(state.robotLocation.y
      -oDesiredLocation.y,2.0)))*100;
23 if (state__.robotLocation.discrete == -1) __reward =  -10;
```

***AM* File for each skill:** The *SD* file specifies an abstraction that corresponds to our concept of what the skill does. Reality, however, is expressed in code. The *AM* file specifies the relation between these two abstraction levels. It contains:
a) A definition of local variables and how to initialize their value. These variables can be used in other assignment statements in the *AM* file. Local variables can be initialized using skill-model parameters, ROS topics, and code return values, or values returned by Python code that manipulates any of these elements. In Lines 13-18, local variables are initialized based on the model parameter's *x,y*, and *z* values. In Lines 2-12, the *goal_reached* local variable is initially set to *False* and then changes to *True* using code that checks whether the */rosout* topic published a message containing the "Goal reached" text. In Lines 19-24, we define the *skillSuccess* variable using *navigate* skill code's return value. The reserved word '__input' refers to an object AOS autogenerates that stores a topic message's recent value (Line 12) or the skill code's returned value (Line 24).

```
1 project: example
2 local_variable: goal_reached
3 topic: /rosout
4 message_type: Log
5 imports: from: rosgraph_msgs.msg
      import: Log
6 type: bool
7 initial_value: False
8 code:
9 if goal_reached == True:
10     return True
11 else:
12     return __input.msg.find('Goal
           reached') > -1
13 local_variable: nav_to_x
14 sd_parameter: oDesiredLocation.x
15 local_variable: nav_to_y
16 sd_parameter:
        oDesiredLocation.y
17 local_variable: nav_to_z
18 sd_parameter:
        oDesiredLocation.z
19 local_variable: skillSuccess
20 imports: from: std_msgs.msg
        import: Bool
21 type: bool
22 from_ros_reservice_response:
         true
23 code:
24 skillSuccess=__input.success
```

b) How to compute the observation returned by the skill model following skill execution using the value of local variables. Users can specify a sequence of rules (expressed in Python) with associated values. The return value is that associated with the first rule evaluated to *True*. In Lines 25-28, we return *eSuccess* if skillSuccess and goal_reached are true and otherwise, *eFailed*. Another option, not shown, yet essential for large observation spaces, is to return a specified local variable value as the POMDP's observation.

```
25 response: eSuccess
26 response_rule: skillSuccess and goal_reached
27 response: eFailed
28 response_rule: True
```

c) Instructions on how to activate the code and how to generate code-parameters. Code-parameters may be unrelated to the model, e.g., a camera's sampling frequency, or they can be derived from the model parameters. The *AM* file defines how they are computed. For example, a navigation skill may specify motions between discrete locations: kitchen, office, lab. Its code requires actual coordinates. The *AM* file will provide a mapping between them. Lines 29-36 specify how to activate the */navigate* ROS service, provide its path and name, and specify its code parameters' value using local variables.

```
29 module_activation: ros_service
30 imports: from: geometry_msgs.msg import: Point
31 imports: from: simple_navigation_goals.srv import: navigateResponse,
      navigate
32 path: /navigate_to_point
33 srv: navigate
34 parameter: goal
35 code:
36 Point(x= nav_to_x, y= nav_to_y, z= nav_to_z)
```

In summary, *AM* files map model-level skill descriptions to parameterized skill-code calls, and translate low-level information about skill execution to model-level observations.

### C. The POMDP Semantics of SDL

The semantics of an SDL specification is a POMDP defined as follows: a) Each state $s \in \mathbf{S}$ is an assignment

to the state variables in the EF. b) Every skill model corresponds to an action template, i.e., an action with free parameters. The possible values of these parameters are constants defined in the EF. Each skill with an instantiation of these parameters is an action in **A** that corresponds to applying this skill with these parameter values. Thus, the specification of **A** is distributed across multiple SD files, allowing for incremental addition of new skills. If an added skill refers to new variables, these must be added to the EF. c) Ω is the set of all possible observations defined in all *AM* files. Observations defined for each skill may differ, making incremental updates easier. The probability of an observation not present in a skill's AM file is 0. d) The transition function **T**, observation function **O** and reward function **R** are *implicitly* defined by the generative model each skill's SD file describes, similarly to PPLs. As noted earlier, variable values persist unless explicitly changed. **R** is the sum of action-specific rewards, described in the SD file, and state-only dependent reward specified in the EF. e) The initial belief state, **I**, is implicitly defined in the EF via a generative model or can be specified to be the final belief state of the previous task.

The distributed specification of different skills does not cause a formal consistency issue because value persistence implies that **T** is well-defined for all variables, including those not listed in the skill's SD file, while **O** is well specified because an observation's probability is 0, unless otherwise specified. Semantic consistency is left to the user, who should use variable names consistently. Maintaining such semantic consistency is helped by requiring that all variables be declared in the EF.

## IV. System Architecture and Operation

We describe the system architecture, shown in Fig. 2, and its operation. The *AOS Web API* exposes a RESTful API. Users can request to integrate their documented code and, in each step, query skill selection and belief state. Once an integration request is received, the AOS generates two components: (1) A POMDP model usable by the solver, and (2) the *Middleware Layer* that enables the solver to activate the skills and receive observations.

The POMDP model is a code-based generative model that can sample the next state, observation and reward following action execution. The AOS generates the model's code by parsing the SD files, and reusing the actual code inside them. This greatly contributes to its sampling efficiency. (See Sec. V-B.) The model's code is compiled as part of the *Planning Engine*, which uses it at run-time to process observations and select the next action.

The *Middleware Layer* is generated using the AM files. It is a ROS node that connects the AOS to the robot framework – the *Functional Layer* in Fig. 2. This node can activate robot skills following the solver's request, and can process data at the *Functional Layer* to generate observations for the solver. At run-time, it waits for new action requests by the solver and activates the corresponding skill. When skill code terminates, it computes and reports back the execution outcome (= observation). The AOS generates this node by

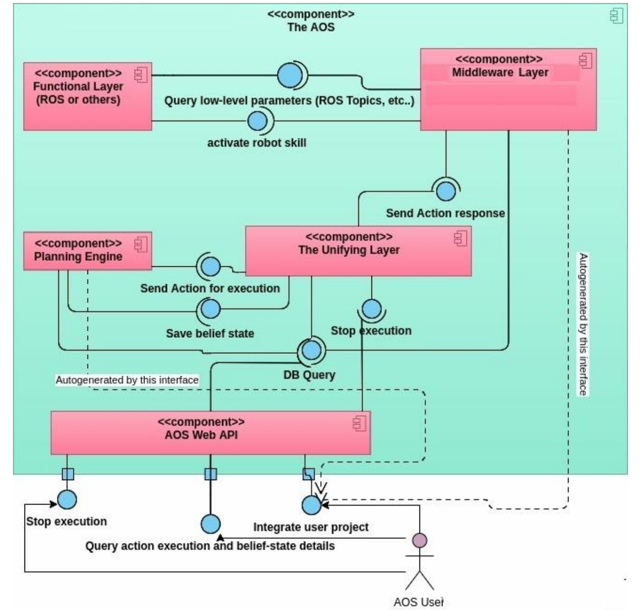


Fig. 2: Dashed lines connect components (in pink) to APIs (in blue) that autogenerates parts of their code.

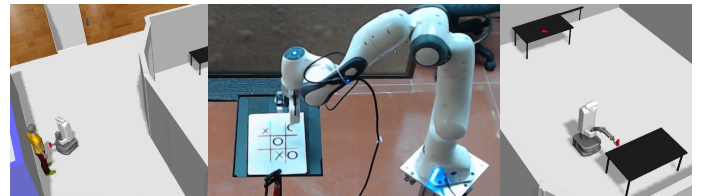

Fig. 3: Experiments: Franka Emika Panda (center); Simulated Armadillo (sides).

translating each section of the AM file into appropriate code within this node. For example, if the *module_activation* section specifies a ROS service, the AOS generates code that calls this ROS service.

Finally, the *Unifying Layer* serves as a communication bus and stores all statistics throughout system operations. This modular design facilitates the replacement of the *Planning Engine* or the *Middleware Layer* to support other decision-making mechanisms or robotic frameworks.

## V. Experiments and Use Cases

We describe two AOS case-studies implemented on different platforms and tested in multiple scenarios, and two experiments that assess its sampling efficiency and robustness under lengthy operation. Our goal was to test the AOS's usability in diverse contexts, its Plug'n Play behavior, its robustness, and the quality of its decisions. In these experiments, the AOS was deployed on the robot or simulation machine, but it can be deployed on any device within the same network when using ROS. We used an Ubuntu 20.04 machine with 4-core Intel i5 processor and 7.6 GB of RAM. Code auto-generation is instant. Building and running the generated code took 40 seconds. When using the previous skill documentation, it takes only a few seconds because code rebuild it not needed.

### A. Case Studies

**The Franka Emika Panda CoBot.** In this experiment, we programmed a Panda CoBot to play tic-tac-toe with a human (Fig. 3). An Intel RealSense D435 camera was attached to the robot arm, and an erasable board with a tic-tac-toe grid was placed within its reach. The goal of this experiment was (1) To validate the AOS's ability to function out-of-the-box given code and documentation files on a physical robot. (2) To test and demonstrate the ease of adapting to task and environment changes. (3) To test and demonstrate its ability to optimize behavior given stochastic skills and sensing.

The robot had two skills: marking a circle in a specific grid cell, detecting changes in the board state and extracting the new board state. The first skill was implemented using our own PID controller based on libfranka, wrapped as a ROS service. The second skill was adapted from code found on the web. After experimenting with the code to see its properties, SD and AM files were specified for each skill. The AOS allows specifying exogenous events in the EF. They are applied prior to the agent's action. This feature was used to model the human as making random legal choices. Finally, in the EF we defined the goal reward, the initial state of an empty board and the starting player. From this point, it was Plug'n Play, requiring no additional effort. The AOS auto-generated the code, and we ran the game, changing starting player, as desired. (Because the human was modeled as a random player, you can observe in the videos [4] that the robot sometimes "counts" on a human mistake of not completing a sequence of three.)

Next, we tested the AOS's ability to adapt to changes in the robot, environment, and task. First, we changed the circle drawing skill to emulate an arm that succeeds in drawing in the center square with probability 0.5. The turn changes following the robot's attempt, regardless of the outcome. Only three lines in the *draw circle* skill's SD file had to be changed to reflect this modified behavior. The POMDP-solver optimizes given this updated SD file. As evident in the videos, the robot now prefers drawing a circle in other positions, selecting the center square only when crucial. Imagine the effort of changing a script to adapt to this capability change. Notice that a classical solver cannot model the new stochastic skill.

Finally, we considered a task change: players score when marking positions adjacent to corner squares they marked before. This new game can be played with the same skills. All we need is a simple change of the reward specification in the EF! The resulting behavior is quite different, as can be seen in the videos [4]. For this, a completely new script or state-machine would have to be written after spending time figuring out good strategies for this game.

**The Armadillo Robot.** Our next experiment used a simulated Armadillo Robot (Fig. 3) to test and demonstrate: (1) POMDP algorithms' ability to optimize behavior given partial observability, noisy sensing, and stochastic effects. (2) The ease of incremental development by gradually adding skills. The simulation environment included a room with two tables, each with a can on it, and a corridor with a person. One can was very difficult to pick (its true size was 10% of the size perceived by the robot). The robot starts near the table with the difficult can. Its goal was to give the can to a person. We implemented three skills: *pick-can, navigate* to a person or a table, and *serve-can* to a person. We used two versions of *pick*'s SD file: the "rough" model assumes that the probability of a successful pick is independent of the selected table; in the "finer" model the success probability is conditioned on the robot's position.

First, we experimented with each skill and used the observed statistics to write their SD files. Then, we specified the AM files and the task. This information suffices for the AOS to control the robot throughout the task. During plan execution, we observed that, occasionally, *pick* ended with the arm outstretched. Serving a person in this state causes a collision. Moreover, *pick* returned success if motion planning and motion execution succeeded, which did not imply the can was successfully picked. Therefore, we wrote two new skills: *detect-hold-can* and *detect-arm-stretched*. They were easy to write because they simply map low-level data published by ROS (gripper pressure, arm-joint angles) into variables used by the SDs. These mappings were specified in the AM files, and their SD files were trivial. We also implemented an alternative *pick* with integrated success sensing. Its return value was the outcome of sensing whether the can is held. We simply changed the SD's output specification and added the above mapping to the AM. Both changes involved adding two lines to the respective file. *Detect-hold-can* is noisy and was modeled as such. *Detect-arm-stretched* is not noisy.

First, with the rough model, the robot (correctly) tries to pick the problematic can, saving the cost of navigating to the other table. With the finer mode, it first moves to the other table where pick is more likely to succeed. Second, without sensing actions, the robot serves the can, but because it has no feedback, goes back to the tables and tries to repeat the process. With sensing, the robot verifies success and only if the result is *yes* does it serve the can and stop. Moreover, since sensing is noisy, yet *sense* instances are modeled as independent, the robot performs multiple sense actions to reduce uncertainty. However, when sensing is integrated into the *pick* action, it does not perform repeated sensing through *pick* because it is costly and may result in the can falling.

### B. Quantitative Experiments

**Efficiency and expressiveness: RDDLSim vs. the AOS.** We use PPL code to specify generative POMDP models because it makes complex models easier to specify, and we can exploit this code directly to get efficient sampling, which improves online solver's efficiency. To test the latter property, we compare in Fig. 4 the AOS sampling rates with those of RDDLSim, an RDDL-based simulator used in the international probabilistic planning competition. The AOS SD files were *automatically* generated from the original RDDL specification. We see a 2x to 25x improvement.

| Domain | RDDLSim | AOS |
|---|---|---|
| Wild Fire [24] | 26,589±1,758 | 253,550±3,463 |
| Academic Advising [24] | 88,890±16,065 | 172,524±3,697 |
| Tamarisk [24] | 5,172±295 | 127,701±5,245 |
| Swarm of Locusts(2) | 28,703±471 | 354,334±6,818 |

Fig. 4: Samples per second: RDDLSim vs. AOS.

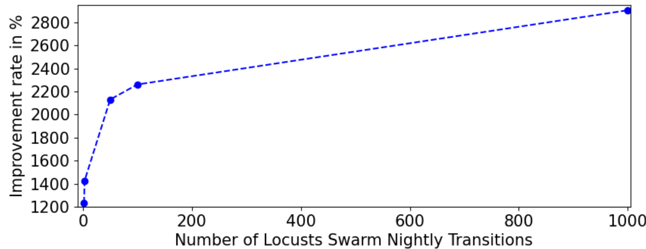


Fig. 5: AOS/RDDLSim sampling rate ratio as fun. of moves.

To demonstrate ease of expressiveness, we introduced the *Locusts Swarm* domain [10]: A nature-reserve team must predict nightly moves of a locust swarm in a 9x9 grid. The swarm moves a few times at night, eating half the plants in each cell it visits. The swarm moves probabilistically, conditioned on the amount of food in adjacent cells. The team can intervene by spraying a cell. To be effective, it must predict the swarm's final resting cell. This domain is easily captured with short, fixed-sized code. Its RDDL spec [4] is more complex and opaque, growing linearly with the number of nightly moves. RDDLSim's sampling rate, much slower to begin with, becomes linearly worse with this parameter, as shown in Fig. 5.

**Robustness to Lengthy Operation.** During the above experiments, described earlier, we did not encounter system errors. To more explicitly test system robustness during prolonged use, we used a set-up motivated by [15] in a simulated pickup-and-delivery task with stochastic actions, partial observability, and noisy sensing: Four toys with different values are located in four different locations. The robot can *navigate* to locations, *pickup* an object, and *give* it to the child. *Navigate* fails with probability 0.2-0.05, depending on the target position. *Pickup* and *give* are deterministic. Each action returns a *success* or *fail* observation. *Navigate*'s observations are noisy, while *pickup* and *give*'s are accurate. A different *pickup* action is required for each toy. Only the correct one succeeds. Initially, the robot does not know the toys' locations, cannot observe them, and can only learn about them from the result of *pickup* actions. It can use up to 16 actions.

We ran the experiment for 24 hours, during which the AOS performed 9505 actions with no errors! To give a sense of the noise inherent in this domain: 13% of the *navigate* attempts and 61% of *pickup* attempts failed. The solver, POMCP, was given 2 seconds to make each decision. It used an auto-generated rollout policy, as described earlier, and delivered 2.97 toys, on average. This problem is too large for exact solvers, so its optimal value is unknown. As a reference, giving 30 seconds per step yielded only a 1% improvement, indicating that the AOS made near-optimal choices. System overhead for each action was 0.77 seconds – 0.4 for action dispatch and 0.37 for observation processing.

## VI. Summary

We described the AOS and some of the empirical work used to validate its capabilities. The AOS makes formal documentation easier by relying on PPLs and explicitly addressing the model-code gap, reducing the (one-time) documentation effort (shareable by the community) required. Its rich POMDP semantics, zero integration effort, plug'n play nature, configurability, ability to support incremental development, and efficiency make it an attractive approach for autonomous robot design.

## References

[1] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3/4, pp. 189–208, 1971.
[2] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, "Rosplan: Planning in the robot operating system," in *ICAPS*, 2015.
[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, 2009.
[4] O. Wertheim, *AOS Repository*, https://github.com/aosbgu
[5] H. Kurniawati, D. Hsu, and W. S. Lee, "SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces," in *RSS'08*, 2008.
[6] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," *NIPS*, pp. 2164–2172, 2010.
[7] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *JAIR*, vol. 20, pp. 61–124.
[8] S. Sanner, "Relational dynamic influence diagram language (rddl): Language description," *Unpublished ms. ANU*, 2010.
[9] T. Dean and K. Kanazawa, "A model for reasoning about persistence and causation," *Comp. Intel.*, vol. 5, no. 3, pp. 142–150, 1993.
[10] R. I. Brafman, D. Tolpin, and O. Wertheim, "Probabilistic programs as an action description language," in *AAAI'23*, 2023.
[11] G. Canal, M. Cashmore, S. Krivić, G. Alenyà, D. Magazzeni, and C. Torras, "Probabilistic planning for robotics with rosplan," in *Annual Conference Towards Autonomous Robotic Systems*, 2019.
[12] S.-O. Bezrucav, G. Canal, M. Cashmore, and B. Corves, "An action interface manager for rosplan," in *ICAPS PlanRob*, 2021.
[13] D. Rao, G. Hu, and Z. Jiang, "Probplan: A framework of integrating probabilistic planning into ROS," *IEEE Access*, 2020.
[14] F. Martín, J. G. Clavero, V. Matellán, and F. J. Rodríguez, "Plansys2: A planning system framework for ros2," in *IROS*, 2021.
[15] F. Rovida, M. Crosby, D. Holz, A. S. Polydoros, B. Großmann, R. Petrick, and V. Krüger, "Skiros—a skill-based robot control platform on top of ros," in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, 2017, pp. 121–160.
[16] T. Niemueller, T. Hofmann, and G. Lakemeyer, "Goal reasoning in the clips executive for integrated planning and execution," in *ICAPS*, 2019.
[17] C. Lesire, D. Doose, and C. Grand, "Formalization of robot skills with descriptive and operational models," in *IROS*. IEEE, 2020.
[18] A. Albore, D. Doose, C. Grand, J. Guiochet, C. Lesire, and A. Manecy, "Skill-based design of dependable robotic architectures," *Robotics and Autonomous Systems*, vol. 160, 2023.
[19] T. Hofmann, T. Viehmann, M. Gomaa, D. Habering, T. Niemueller, G. Lakemeyer, and C. Team, "Multi-agent goal reasoning with the clips executive in the robocup logistics league." in *ICAART (1)*, 2021, pp. 80–91.
[20] [Online]. Available: https://kcl-planning.github.io/ROSPlan
[21] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *ICML'22*, 2022, pp. 9118–9147.
[22] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the planning abilities of large language models – a critical investigation." in *NeurIPS'23*, 2023.
[23] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, "LLM+P: empowering large language models with optimal planning proficiency," *CoRR*, vol. abs/2304.11477, 2023.
[24] S. Sanner, *RDDL Domains*, https://github.com/ssanner/rddlsim/tree/master/files/final_comp_2014/rddl.