Towards Plug'n Play Task-Level Autonomy for Robotics Using POMDPs and Generative Models

Or Wertheim,
Ben-Gurion University of the Negev
orwert@post.bgu.ac.il

Dan R. Suissa
Ben-Gurion University of the Negev
danrouve@bgu.ac.il

Ronen I. Brafman
Ben-Gurion University of the Negev
brafman@cs.bgu.ac.il

To enable robots to achieve high level objectives, engineers typically write scripts that apply existing specialized *skills*, such as navigation, object detection and manipulation to achieve these goals. Writing good scripts is challenging since they must intelligently balance the inherent stochasticity of a physical robot's actions and sensors, and the limited information it has. In principle, AI planning can be used to address this challenge and generate good behavior policies automatically. But this requires passing three hurdles. First, the AI must understand each skill's impact on the world. Second, we must bridge the gap between the more abstract level at which we understand what a skill does and the low-level state variables used within its code. Third, much integration effort is required to tie together all components. We describe an approach for integrating robot *skills* into a working autonomous robot controller that schedules its *skills* to achieve a specified task and carries four key advantages. 1) Our Generative Skill Documentation Language (GSDL) makes code documentation simpler, compact, and more expressive using ideas from probabilistic programming languages. 2) An expressive *abstraction mapping* (AM) bridges the gap between low-level robot code and the abstract AI planning model. 3) Any properly documented *skill* can be used by the controller without any additional programming effort, providing a Plug'n Play experience. 4) A POMDP solver schedules skill execution while properly balancing partial observability, stochastic behavior, and noisy sensing.

1 Introduction

To build autonomous robots capable of performing interesting tasks, one must integrate multiple capabilities such as navigation, localization, different types of object manipulations, object detection, and more. Each of these areas attracts much research interest and our ability to program robots that can provide these capabilities, which we refer to as *skills*, has progressively improved. Moreover, for many skills, one can find publicly available software packages that implement them and publicly available algorithms one can implement independently. However, integrating diverse skills into a working system that can utilize them in unison to perform a given task is not easy. First, this requires designing and implementing an execution system that can initiate the execution of each skill with the suitable parameters and adequately process the output of implemented sensing skills. Second, one must provide the logic that dictates which skills to use and when. A solution must address both the software engineering challenge and the conceptual issue of generating the execution logic, i.e., the behavior policy.

The latter problem is often solved by manually writing a script. Such pre-programmed scripts (can) have the advantage of being explainable and predictable. However, writing scripts for robotic agents is hard because physical agents' actions are usually probabilistic, and robots may have a partial noisy view of the world. Moreover, a script usually addresses a specific task only. To build autonomous systems that can perform diverse tasks in diverse environments, we must constantly supply new scripts or alter existing ones. For this reason, starting with the very early days of robotics research, automated AI planning was suggested as a possible solution to the problem of generating a behavior policy [8].

There is abundant work on the use of planning algorithms in robotics but these are mostly one-of-a-kind implementations. ROSPlan [5] was one of the first systems attempting to address this issue by providing architecture and software that supports the integration of a planning engine into a ROS-based robot architecture [19]. Following ROSPlan, several other systems emerged that seek to make the integration of planners into robot software easier, such as [17, 21]. However, these systems have two main weaknesses. First, they offer limited support for robots that operate with partial observability and use noisy sensors – a basic property of many, if not most, mobile robotic systems. Second, they only partially address the integration issue discussed earlier, as they still require manual programming of standard interfaces between the skill's code and the engine. Moreover, they are often bound to specific systems, such as ROS. Finally, they rely on formal Action Description Languages (ADLs).

Indeed, most planning algorithms need, as input, an action specifications in a formal language, such as the Planning Domain Definition Language (PDDL) [9], the POMDP XML format (POMDPX) , etc. Very few programmers are familiar with these languages and it is difficult to specify stochastic effects and sensing with them except in very small models[5] [24]. Instead, [24] is able to use a relatively simple, code-based generative model to model the game of Pacman, which has 10^{56} states. Their approach for modeling this large, nontrivial domain can be divided into two. 1) Describe the planning domain via a sampling procedure, or simulator, that is able to sample the next-state, the next-observation, and the next reward given the current state and action, correctly. A model that specifies how some object is sampled, possibly dependent on some context parameters (e.g., a state and an action), is often called a generative model – it shows how the object is "generated". 2) Use code to describe this sampling procedure. Indeed, in the past decade or so, it was realized that programming languages could be adapted to serve as means of specifying complex generative models. This led to the advent of probabilistic programming languages [4, 10] that, through the use of code, can express complex generation processes and perform inference on them.

Code-based specification would not have worked with older planning algorithms that require ADL input. Yet, newer planners based on sampling procedures, such as Partially Observable Monte-Carlo Planning (POMCP) [24] and Determinized Sparse Partially Observable Tree (DESPOT) [25] directly use code-based sampling procedures. Code-based specification is typically better for such planners because it can provide more efficient samplers than ones built from declarative models[2].

Among ADLs, the Relational Dynamic Influence Diagram Language (RDDL) [23] is noteworthy for its ability to compactly specify a generative probabilistic model using a dynamic Bayesian network model [11]. Yet, it, too, does not have the expressiveness of programming languages, like advanced control structures (e.g., 'while' loops) or built-in multi-purpose functions (e.g., C++ 'cmath' library or the 'string' class that provides string manipulation functions).

A final crucial issue that requires attention is the abstraction gap. Action languages typically employ abstract concepts, such as *holding(cup)* or *at(kitchen)* to describe their model, whereas robotic code must interact with many lower-level variables.

We seek to address existing systems' limitations and provide programmers' with a plug'n play experience as follows: The robot programmers program or import skills' code of their choice. They document their code using the more abstract Generative Skill Documentation Language (GSDL) and use an expressive *Abstraction Mapping* (AM) to bridge the gap between low-level robot code and the abstract AI planning model. Next, they need only supply a goal specification for each task and the system auto-generates all needed integration code and controls the robot online. The system described here[1] is part of the *Autonomous Robot Operating System* (AOS), a general system we are developing for making

---

[1]Our system's code is available at https://github.com/orhaimwerthaim/AOS-WebAPI.

programming of autonomous software from components easy. This paper describes the decision engine of the system, which we will refer to as the AOS for brevity, despite its more limited scope.

The AOS can deal with partial observability and noisy sensing by using solution algorithms for partially observable Markov decision processes (POMDPs) and it uses ideas from probabilistic programming languages to make model specification easier and more flexible. More specifically, our system makes the following contributions. 1. It introduces the Generative Skill Documentation Language (GSDL), a new code-based action description language that supports stochastic actions and sensing and partial observability. 2. It introduces a new *Abstraction Mapping* (AM) format that addresses the model-code abstraction gap. 3. It leverages the code in the GSDL to automatically generate efficient sampling code[2] for sampling-based POMDP solvers and RL algorithms, but also supports ADL-based solvers. 4. It utilizes the knowledge in the AM to provide a plug'n play experience in which code for integrating the planner and the diverse skills is auto-generated by the system, leaving the programmers with the sole task of describing their code and the task. 5. Although currently demonstrated on ROS [19], the architecture is general and can be converted for other robot frameworks.

Our empirical evaluation, involving different systems, demonstrates these capabilities, and its modular specification makes incremental development simple.

2   Background and Related work

We review POMDPs, AI planning architectures for robotics, and robot skills' documentation languages.

2.1   Partially Observable Markov Decision Process (POMDP)

A discrete-time POMDP models the relationship between an agent and its environment. Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma, \mathcal{I} \rangle$: $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{T}$ is the state transition model, $\mathcal{R}$ is the reward model, Ω is the observation space, $\mathcal{O}$ is the observation model, $\gamma \in [0,1]$ is the discount factor, and $\mathcal{I} \in \mathcal{B}$ is the initial belief state. A belief state, which is a distribution over $\mathcal{S}$ is required since, in POMDPs the agent may not be fully aware of his current state.

Following each action $a \in \mathcal{A}$, the environment transitions from its current state $s \in \mathcal{S}$ to state *s'* with probability $\mathcal{T}(s,a,s')$. Then, the agent receives an observation $o \in \Omega$, with probability $\mathcal{O}(s',a,o)$, and a reward $r = \mathcal{R}(a,s') \in \mathbb{R}$. In the discounted case, we assume that earlier rewards are preferred and use a predefined discount factor γ to reduce the utility of later rewards. The present value of a future reward r that will be obtained at time t is hence $\gamma^t r$. Using standard probabilistic inference, the updated belief state $b' = Pr(s|a,o,b)$ can be computed from the model parameters.

A behavior *policy* for a POMDP, or simply a policy, is a mapping $\pi : \mathcal{B} \mapsto \mathcal{A}$ from belief states to actions. The goal of POMDP solvers is to find a policy π^* that maximize the expected accumulated discounted reward, i.e., $\pi^* = \max_{\pi}[\mathbb{E}_{\pi}[\sum_{t=1}^{\infty} \gamma^t r_t.]$.

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

---

[2] An experiment [28] comparing sampling rates of RDDLSim's [22] generic code vs. AOS's domain-specific auto-generated code showed significant differences in favor of the AOS (452,000 vs. 12,500 samples per second).

2.2 Planning-Based Deliberative Architectures

Our work relates to deliberative robotic architectures, which follow the sense-plan-act paradigm, specifically those designed for general purpose rather than specific application. In this respect, it includes the plan, act, observe components discussed by [13]. The influential system that motivated much of our work is ROSPlan [5]. ROSPlan is a planning and a plan execution architecture for robotics that generates plans based on a PDDL2.1 [9] (or RDDL [23]) documentation of ROS-implemented skills. It supports a rich set of planning formalisms: classical, temporal, contingent planning, and probabilistic planning with some limitations [3]. However, even its probabilistic variant maintains only a single world state that the user updates during execution, and it requires deterministic sensing. When the inner state is discovered to be incorrect, the user can invoke replanning. As such, it cannot support full-fledged POMDP planning and cannot model the effect of sensing actions on the belief state of the agent. Integration with ROSPlan requires user effort [30], although recent work [2] seeks to reduce it under certain conditions.

The CLIPS Executive (CX) [17] is a flexible robot execution and planning framework with some innovative ideas. It stores a predefined high-level plan in the form of a goal tree. CX calculates the next goal to pursue, and a PDDL solver generates a plan for this goal; based on the plan execution result, CX calculates the next goal and so on. The system preserves an extended model with the information required to activate robot skills. CX support for non-deterministic skills is limited to replanning. Nevertheless, it proved its utility in a number of robotics competitions.[3]

Unlike both systems, our system supports a full-fledged POMDP model and uses an expressive specification language.

SkiROS [21] is a platform that can auto-generate action descriptions in PDDL based on a predefined ontology and invokes a solver to schedule the different robot skills. It includes a number of innovative ideas and a variety of tools. It, too, is based on classical planning with replanning, as opposed to a POMDP model, and it requires users to work using strict patterns. Thus, code used must adapt to the architecture, whereas our system seeks to support integration of diverse code from diverse sources.

The system described in [14] and [1] proposes a formal language to specify robot skills with an expressive descriptive model used for reasoning, and an operational model. It maintains a life cycle for every robot skill and allows concurrent activation of the same skill. Code auto-generation assists users in integrating their code, yet users do need to add some code to handle changes in their skill life-cycle. This system uses a fixed policy described by an automaton or a behavior tree. Users can also use AI planning with PDDL solvers. Our system does not support concurrent skill activation, but supports the richer POMDP model and code-based generative model specification. Moreover, it requires no additional information besides the documentation.

2.3 Skill Models

Architectures that use a planner to control the execution of a set of skills require some form of skill documentation as input to the planner. This documentation describes the effect of applying this skill/action on the system's state. ADLs such as STRIPS [8], PDDL [35, 9] and RDDL [23] use formal syntax to describe the action's effect. Most relevant to us, RDDL is a language for describing dynamic Bayesian networks (DBNs) [11] that is used for specifying transition and observation functions in MDPs and POMDPs. It describes the post-action value of a state variable as a function of the pre-action variables' values. Moreover, RDDL allows the definition of intermediate effect variables for expressing more complex dependent effects. RDDL specs, as well as their classical counterparts, can be understood as

---

[3]The platform was used by the winner of RoboCup German Open 2018 and PExC 2018.

generative model specifications, as they implicitly describe how the post-action distribution is generated given pre-action values. Writing them, however, has some limitations: a) RDDL syntax is less expressive than programming languages. For example RDDL cannot describe a generative model that samples from a distribution until a condition is met since it does not support loops; b) probabilistic initial states are not supported; c) hierarchical generative processes require intermediate variables definition, which may over-complicate the model. GSDL, on the other end, has the expressive power of C++, which includes control structures and complex data structure manipulation. GSDL can easily describe real-world complex domains with probabilistic initial states, extrinsic changes, and action pre-condition. Each is in a designated area for a clear separation in the generative model. Moreover, it allows users to define hierarchical generative processes straightforwardly using code without intermediate variables. Notably, the use of code, beyond making the specification process simpler, makes the sampling process required by the solver much more efficient[2] (the generative model itself is used for sampling). This translates into faster computation or (given similar time) better decisions. The use of code (we support C++) also reduces the amount of new syntax a programmer must master to write a specification.

3   System Overview and Concept

There is a long tradition of systems and architectures for autonomous robots based on tightly coupled components, such as [15, 6, 21] that provide various reasoning and planning services and provide support for programming skills in a principled manner. Undoubtedly, such systems have shown some impressive results, yet while they offer various capabilities that can be exploited when writing new code, such code must conform to the system's requirements or methodology.

A more common approach with roots in the world of computer programming, is to try to re-use best-of-breed, (or most-accessible) components, write additional functions/skills, and put them together. In robotics, we can use, for example, various ROS libraries, recent deep-learning-based object detection or object manipulation code, together with our own code for other needed skills. Our system, the AOS, takes this latter approach.

3.1   Concept

The design process we support is the following: The user starts with a set of implemented skills, whether imported or self-programmed. Each skill is a code module that can be activated and may respond with a returned value. These skills need to be documented. Code documentation is standard practice, but we require more formal documentation, consisting of two components, as described below. The *GSDL file* describes how the execution of the code impacts the robot's and the world's state. The *Abstraction Mapping file* (AM) documents the connection between the abstract POMDP model depicted in the GSDL file and the skill code. The AM describes how to activate the code, how to map abstract parameter values to code-level parameters, and how to compute the planning model-level observation based on the robot skill execution output. This provides a clean separation between the abstract system model captured by the GSDL file and low-level aspects captured by the AM file. An additional global *Environment file* is needed to specify the state variables, initial belief state, extrinsic changes, and special states (e.g., goal states).

At this point, the user sends an HTTP request to the AOS Web API containing the path to their documented code. The AOS uses the GSDL and Environment files' code to auto-generate sampling code that samples in accordance with the model specified in the GSDL fie. The solver is then compiled and

run. Similarly, a ROS *middleware node* that communicates with the solver is auto-generated based on the AM files. The robot and the *middleware node* are initialized, and an online POMDP solver now operates the robot, attempting to optimize its behavior. We use POMCP [24], but any other online solver supporting the required API can be used. The user may query the AOS at any time for the execution status. We also support the use of an off-line solver, desirable when the model is not too large and response times must be fast. For this purpose, we use the sampling code to convert the code-based generative model into a standard POMDP model and use the SARSOP solver [27] to solve it.

The AOS auto-generates code for two purposes: 1) code required to run the POMDP solver that can sample states and observations using the GSDL files; 2) integration code, i.e., code that enables the solver to communicate with the skills, activate them, and receive 'real-world' observations using the AM files. This results in a true plug'n play experience: any executable skill on the robotic platform can be easily added to the system, provided a GSDL and an associated AM file. Once added, the planning and execution engine can activate it with no additional effort.

3.2 Skill Documentation

The idea of using a formal description of an action as an input to a control algorithm underlies the area of AI planning [8, 12], and goes back to the robot Shakey [18]. Below we explain the language we use and its semantics. We start with the latter, explaining the generative model our documentation specifies, and then, through an example, we describe the structure of our specification.

3.2.1 Semantics and Structure

Our specification describes a POMDP. Because our specification is code-based, this is not an explicit POMDP, but rather an enhanced POMDP simulator. Enhanced because it contains information about the distributions from which state, observations, and rewards are sampled, much like in probabilistic programming languages. We refer to it as a generative model because it explains how to generate the next state, observation, and reward from the current state and action.[4]

Using code, we describe how the initial state is sampled and how the world changes. Changes occur in discrete steps (i.e., at this point, we ignore the duration of an action, although it can be used within the code), and can be exogenous or action induced. An action is selected at each time step. Before it is executed, an exogenous effect may take place. Then, the action is executed leading to a new state that depends on the state following any exogenous event and the action. Depending on the resulting state and the action, an observation and a reward are received.

The model specification is divided into multiple files. A global Environment File describes the POMDP elements unrelated to any specific robot skill: state variable definitions, initial belief state, the impact and likelihood of exogenous events, and state-dependent rewards. For each skill, a separate GSDL file documents the impact of that skill: how it generates the next state, observation and reward, conditioned on the after exogenous effect state. This separation makes for a more manageable and incremental software development process, and makes it easy to export and continuously add documented skills.

Each file has sections that correspond to the different elements it describes (e.g., initial state, observation probability, etc.). These sections contain sets of assignments that use C++ code lines. In them, the modeler can refer to three copies of state variables that can be conditioned on and assigned to: 1) the

---

[4]The term *generative model* comes from the classification literature, while our models are dynamic, but it refers to models that specify the conditional probability of the observations given a class. That is, how the data collected is generated.

previous state, 2) the state *after extrinsic changes*, and 3) the *next state*. Moreover, there are variables for *met precondition*, *reward*, and *observation*.

In addition, an AM file is associated with each skill, mapping between the skill's GSDL documentation to the skill code.

3.2.2 Documentation Specification Through An Example

To illustrate actual documentation files, we describe part of the specification of a toy problem. For more complete specification of the documentation format, see [32]. In this problem, a robot with a single navigation skill must navigate as fast as possible to three known locations but we prefer that it will not visit the second location before visiting the first one. The robot's initial location is unknown: it is the first location with probability 0.5, and otherwise,most likely (80%), it starts at the third location. Moreover, there is a 5% chance that a person may occasionally move the robot, in which case it loses its orientation.

The navigation skill may fail, causing the robot to lose its orientation. Moreover, after experimenting with our navigation skill we know that: (1) Navigating the robot to its current location causes it to lose orientation. (2) It has a 10% chance of losing its orientation while navigating to a different location. (3) The skill mistakenly reports success in 20% of the cases in which the robot lost its orientation along the way. (4) When the robot loses orientation or starts navigating without knowing its location, the skill takes significantly longer to execute.

We describe abbreviated versions of the *Environment*, Navigation *GSDL*, and Navigation *AM* files for this example. In them, we distinguish between three values of each variable *x*. Its value before skill execution is denoted *state.x*. Its value after any extrinsic event is denoted *state_.x*. And its value after the skill execution is denoted *state__.x*. State variables will also be referred to as *global variables* to distinguish them from *local variables*.

Environment File   Each robot has one *Environment file* that contains four sections. 1) The list of state variables (not shown) that comprise a POMDP state $s \in \mathcal{S}$. These may be primitive (e.g., int, string, bool, or float) or compound (custom types with sub-variables that are defined in the Environment file) types. 2) A generative model of the initial belief state. Line 9 in Listing 1 describes the uncertainty regarding the robot's initial location. 3) A generative model for extrinsic changes, possibly conditioned on the previous state. For example, a certain constant probability of some malfunction when it is raining. Line 25 in Listing 1 describes the possible effect of a person moving the robot. 4) An objective function as a set of state conditions and associated rewards. We can see in lines 11-21 in Listing 1 a high reward for visiting all locations that express our goal and a smaller negative reward to express our preference of not visiting the second location before visiting the first.

GSDL files   A GSDL file is associated with a specific skill code and documents its expected behavior. It provides a quantitative description of the code's effects using concepts one would use to describe what one's code does in the world. The GSDL file describes two elements of the global generative model: (1) Calculating the *met precondition* random variable. Lines 12-19 in Listing 2 express that we don't want the robot to navigate to its current location and lose its orientation. (2) The dynamic model, i.e., how to sample the *next state*, action cost (or *reward*), and *observation* random variables. Lines 21-34 in Listing 2 describe it.

Specifically, line 23 indicates that the robot loses orientation when navigating to its location or if the navigation fails (10% chance), else it reaches its desired location. Line 27 defines the observation's generative model (called *moduleResponse*) to return a *Failed* observation 80% of the time that the robot

lost its orientation, expressing noisy sensing. Line 30 updates the *reward* model so that if the robot **starts** navigating without knowing its location, it takes more time, expressed by a negative reward of minus five; otherwise, a function on the navigation distances expresses the time it takes to navigate. Finally, line 33 states a fixed large penalty when the navigation **ends** in losing the orientation. Skills usually have parameters (e.g., destination of *move*), whose possible values are currently defined in the Environment file. For example, in Lines 8-9 in Listing 2 the navigation destination is specified. The AOS Planning Engine instantiate any parameter with any legal parameter value when activating a skill.

Abstraction Mapping File (AM)   The AM file documents the abstract mapping between the robot code and the GSDL file (POMDP model) and serves as a bridge so the AOS can smoothly control the robot and reason about its execution outcomes. Each AM file is associated with the code for one skill and has two main roles that serve to map between code-level parameters and model-level parameters.

The first role is to describe how to activate the code. Lines 18-28 in Listing 3 describe how to activate a ROS service, specifying its path, service name, and parameters. The ROS service activation requires mapping high-level POMDP action parameters into lower-level code parameters, as defined in lines 43-54 in Listing 3 and used in line 25.

The second role is to define the observation associated with the skill execution outcome. Recall that in a POMDP an observation is obtained following each action execution. The AM computes the value of this observation from lower-level code parameters. Specifically, the AM specifies the observation value, lines 6-17 in Listing Listing 3 describe the *Success* and *Failed* observations by referring to *local variables*. *Local variables* get their value in one of three ways. a) By a GSDL parameter. Lines 43-54 in Listing 3 define local variables for the 'x,' 'y,' 'z' coordinates taken from the *desired location* GSDL parameter (line 8 in Listing 2). b) As a function of public robot-framework data (e.g., ROS topics) or other local variables. Lines 36-42 in Listing 3 describe the *planSuccess* local variable, whose value is *True* if during skill execution, the */navigation/planner_output* topic published a message containing the string 'success'. c) As a function of the skill code's response. Lines 30-35 in Listing 3 describe the *skillSuccess* local variable who stores the ROS service response.

Thus, we see that the AM file can transform low-level public data into abstract observations correlated with the GSDL file and vice versa. AM files, like GSDL files, harness the expressive power of programming languages (the AM supports Python) to allow flexible integration with the AOS. Moreover, the user's sole work is to generate valid and coherent documentation, while the AOS supplies the tools to transform this documentation into a working autonomous robot. Furthermore, the AM allows more accurate reports of skill outcomes than initially coded and does so by reasoning with additional public data external to the skill code (lines 36-42 in Listing 3).

4   Experiments

We conducted several experiments [32] to validate our system in different scenarios, described below. Their goal is to test ease of use, and the impact of relying on POMDP-based planners, and their highlights can be seen in our system overview video [26]).

4.1   TurtleBot3 Gazebo simulation

Our first experiment used the TurtleBot3[20] Gazebo simulation to see how we can quickly get sophisticated behavior with little effort and existing code. The test environment included nine locations on a

```
1  {
2  "GsdlMain": {
3      ...
4      "Type": "Environment"
5  }
6  ...
7  "InitialBeliefStateAssignments": [
8      {
9          "AssignmentCode": "state.robotLocation.discrete = AOS.Bernoulli(0.5) ? 1 (AOS.Bernoulli(0.2) ? 2 : 3);"
10     }],
11 "SpecialStates": [
12     {
13         "StateConditionCode": "!state.v1.visited && state.v2.visited",
14         "Reward": -50.0,
15         "IsOneTimeReward": true
16     },
17     {
18         "StateConditionCode": "state.v1.visited && state.v2.visited && state.v3.visited",
19         "Reward": 7000.0,
20         "IsGoalState": true
21 }],
22 "ExtrinsicChangesDynamicModel":
23 [
24     {
25         "AssignmentCode": "if (AOS.Bernoulli(0.05)) state_.robotLocation.discrete = -1;"
26     }
27 ]}
```

Listing 1: *Environment* File Example.

```
1  {
2  "GsdlMain": {...
3      "Type": "GSDL"
4      ...
5  },
6  "GlobalVariableModuleParameters": [
7      {
8          "Name": "oDesiredLocation",
9          "Type": "tLocation"
10     }
11 ],
12 "Preconditions": {
13     "GlobalVariablePreconditionAssignments": [
14         {
15             "AssignmentCode": "__meetPrecondition= oDesiredLocation.discrete != state.robotLocation.discrete;"
16         }...
17     ],
18     "ViolatingPreconditionPenalty": -10
19 },
20 ...
21 "NextStateAssignments": [
22     {
23         "AssignmentCode": " state__.robotLocation.discrete = !__meetPrecondition || AOS.Bernoulli(0.1) ? -1:
                 oDesiredLocation.discrete;}"
24     },
25     ...
26     {
27         "AssignmentCode": "__moduleResponse = (state__.robotLocation.discrete == -1 && AOS.Bernoulli(0.8)) ?
                 eFailed : eSuccess;"
28     },
29     {
30         "AssignmentCode": "__reward = state_.robotLocation.discrete == -1 ? -5 : -(sqrt(pow(state.robotLocation.x-
                 oDesiredLocation.x,2.0)+pow(state.robotLocation.y-oDesiredLocation.y,2.0)))*10;"
31     },
32     {
33         "AssignmentCode": "if (state__.robotLocation.discrete == -1) __reward =  -10;"
34     }
35 ]
36 }
37 }
```

Listing 2: Navigation Skill *GSDL* File Example.

```
1   {
2   "GsdlMain": {
3       ...
4       "Type": "AM"
5   },
6   "ModuleResponse": {
7       "ResponseRules": [
8           {
9               "Response": "eSuccess",
10              "ConditionCodeWithLocalVariables": "skillSuccess and planSuccess"
11          },
12          {
13              "Response": "eFailed",
14              "ConditionCodeWithLocalVariables": "True"
15          }
16      ]
17  },
18  "ModuleActivation": {
19      "RosService": {
20          ...
21          "ServicePath": "/navigate_to_point",
22          "ServiceName": "navigate",
23          "ServiceParameters": [
24              { "ServiceFieldName": "goal",
25                "AssignServiceFieldCode": "Point(x= nav_to_x, y= nav_to_y, z= nav_to_z)"}
26          ]
27      }
28  },
29  "LocalVariablesInitialization": [
30      {
31          "LocalVariableName": "skillSuccess",
32          "FromROSServiceResponse": true,
33          "AssignmentCode": "navigateSuccess=__input.success",
34          ...
35      },
36      {
37          "LocalVariableName": "planSuccess",
38          "RosTopicPath": "/navigation/planner_output",
39          "InitialValue": "False",
40          ...
41          "AssignmentCode": "if planSuccess == True:\n\treturn True\nelse:\n\treturn __input.data.find('success') > -
                1"
42      },
43      {
44          "LocalVariableName": "nav_to_x",
45          "FromGlobalVariable": "oDesiredLocation.x"
46      },
47      {
48          "LocalVariableName": "nav_to_y",
49          "FromGlobalVariable": "oDesiredLocation.y"
50      },
51      {
52          "LocalVariableName": "nav_to_z",
53          "FromGlobalVariable": "oDesiredLocation.z"
54      }
55  ]
56  }
```

Listing 3: Navigation Skill *Abstraction Mapping* File Example.

map. The goal was to visit all locations while using a minimal length path. For navigation, we used ROS Move-Base [16], restricted to start and end positions that correspond to the nine locations. The programmer then defined a GSDL and AM files for this skill. The GSDL file uses nine boolean variables that indicate whether a position was visited, a cost function that is equal to distance travelled, and a reward for reaching all points. At this point, the AOS auto-generation code generated the needed interfaces, and when the planner was activated, the robot performed the task, traveling the minimal distance.

4.2   The Franka Emika Panda CoBot

Our second experiment involved a Panda CoBot [7] playing tic-tac-toe with a human (see video [33]). An Intel RealSense D415 camera was attached to the robot arm, and an erasable board with a tic-tac-toe grid was placed within its reach. The experiment was based on two skills: marking a circle in a specific grid cell, and detecting change in the board state and extracting the new board state. The first skill was implemented using our own PID controller based on `libfranka`, which we 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, GSDL and AM files were specified for each skill. The AOS allows the specification of an *Environment file* that describes exogenous events and are executed prior to every agent's action. This feature was used to model the human's action. We modeled the human as making random legal choices[5]. Finally, we defined the goal reward, an initial state of empty board and the starting player, in the Environment file. Again, following the automated code generation, we run the game (changing the starting player, as desired). Because the human was modeled as a random player, you can observe [34] the robot sometimes relying on a human mistake of not completing a sequence of three.

4.3   Armadillo Robot Gazebo Simulation

The prior experiments involved mostly deterministic systems with full observability and few skills, and were aimed at showing the plug'n play nature of the system. Our final experiment (see video [31]) was conducted on a Gazebo simulation of our Armadillo robot with more skills, partial observability, noisy sensing, and stochastic effects. These experiments demonstrate the advantage of using a POMDP model, and the ease of incremental development (see [29]).

The simulation environment included a room with two tables, and a corridor with a person. Each table had a can on it. One of the cans was very difficult to pick up (its true size was 10% of the size perceived by the robot). The robot was located near the table with the difficult can. The goal was to give the can to a person in the corridor. Three skills were implemented by us: *pick-can, navigate* which can navigate to a person, Table1, or Table 2, and *serve-can* which handed the can to the person. For the experiments, we used two versions of the *pick* GSDL: a "rough" model that assumes that the probability of a successful pick action is independent of the selected table, and a "finer" model in which the success probability is conditioned on the robot's position.

First, we experimented with each skill, saved statistics of their behavior, and used this information to write their GSDL files. In addition, we provided the AM files and the task specification. Again, this was sufficient to start the system going and attempt to carry out the task. However, as the plan was executed, we saw that, occasionally, the pick skill ends with the arm outstretched. Attempting to serve the person in this state causes a collision (i.e., injured the person). Moreover, *pick* returned success if

---

[5]To model the human action, we used a C++ *while* loop that repeatedly sampled a tic-tac-toe cell until an empty one was sampled. RDDL cannot compactly express behaviors of sampling until a condition is met. It would have to use an exhaustive *if-sample-else-sample* list which is only feasible for tiny distribution spaces.
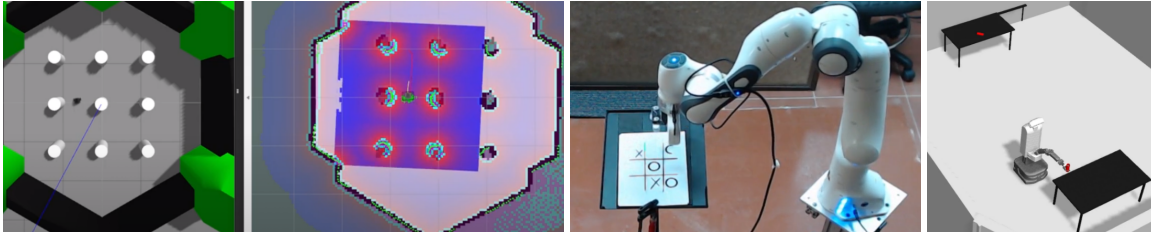
Figure 1: Experiments: (left) The TurtleBot3 Gazebo simulation and rviz. (center) The Franka Emika Panda. (right) The Armadillo Gazebo simulation.

motion planning and motion execution succeeded, but this did not imply that the can was successfully picked up. Therefore, we wrote two new skills: *detect-hold-can* and *detect-arm-stretched*. Implementing such skills that only map low-level public data (gripper pressure, arm-joint angles) to high-level insights is immediate. The user should only implement ROS services that do nothing and document them with GSDL and AM files. The AM files will describe the topics to listen to (e.g., gripper pressure, arm-joint angles) and their mappings to high-level observations. We also implemented an alternative approach where the sensing was integrated into the *pick* skill and its return value now reflected the outcome of sensing whether the can is held. This, too, is very easy to do through the output specification in the AM file. Both involve small changes 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 we saw, the robot (correctly) tries to pick the problematic can because it saves the cost of navigating to the other table. With the finer model, it first moves to the other table where pick action is more likely to succeed. Second, without the separated sensing actions, the robot serves the can, but then, because it has no feedback, goes back to the tables and tries to repeat the process. With sensing, the robot verifies success. If the result is yes, only then does it serve the can and stops. Moreover, since sensing is noisy, the robot performs multiple sense actions to achieve a belief state with less uncertainty because the results of the sensing actions are modeled as independent. However, when sensing is integrated into the pick action, it cannot do independent sensing, and repeating the pick action is not desirable.

5   Summary

We presented the decision-engine of the AOS. Given a set of implemented skills, documented using a GSDL and AM files, the initial system state, and a reward specification, the system generates software that controls the robot by activating these skills as needed, taking care of both execution logic and the software required to integrate all the components into a working system. Our empirical study demonstrated true plug'n play functionality and intelligent controller choices.

References

[1] Alexandre Albore, David Doose, Christophe Grand, Charles Lesire & Augustin Manecy (2021): *Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management*. In: *3rd IEEE/ACM International Workshop on Robotics Software Engineering, RoSE@ICSE 2021, Madrid, Spain, June 2, 2021*, IEEE, pp. 47–54, doi:10.1109/RoSE52553.2021.00015.

[2] Stefan-Octavian Bezrucav, Gerard Canal, Michael Cashmore & Burkhard Corves (2021): *An action interface manager for ROSPlan*. In: *9th ICAPS Workshop on Planning and Robotics (PlanRob)*, pp. 1751–1756, doi:10.5281/zenodo.5348002.

[3] Gerard Canal, Michael Cashmore, Senka Krivić, Guillem Alenyà, Daniele Magazzeni & Carme Torras (2019): *Probabilistic planning for robotics with ROSPlan*. In: *Annual Conference Towards Autonomous Robotic Systems*, Springer, pp. 236–250, doi:10.1007/978-3-030-23807-0_20.

[4] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li & Allen Riddell (2017): *Stan: A probabilistic programming language*. Journal of statistical software 76(1), doi:10.18637/jss.v076.i01.

[5] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcis Palomeras, Natalia Hurtos & Marc Carreras (2015): *Rosplan: Planning in the robot operating system*. In: *Twenty-Fifth International Conference on Automated Planning and Scheduling*, pp. 1751–1756, doi:10.2478/CAIT-2012-0018.

[6] Mohammed Diab, Mihai Pomarlan, Daniel Beßler, Aliakbar Akbari, Jan Rosell, John A. Bateman & Michael Beetz (2020): *SkillMaN - A skill-based robotic manipulation framework based on perception and reasoning*. Robotics Auton. Syst. 134, p. 103653, doi:10.1016/j.robot.2020.103653.

[7] Franka Emika (2021): *Franka Emika Panda cobot*. Available at `https://www.franka.de/robot-system/`.

[8] Richard E Fikes & Nils J Nilsson (1971): *STRIPS: A new approach to the application of theorem proving to problem solving*. Artificial intelligence 2(3-4), pp. 189–208, doi:10.1016/0004-3702(71)90010-5.

[9] Maria Fox & Derek Long (2003): *PDDL2.1: An extension to PDDL for expressing temporal planning domains*. Journal of artificial intelligence research 20, pp. 61–124, doi:10.48550/arXiv.1106.4561.

[10] Hong Ge, Kai Xu & Zoubin Ghahramani (2018): *Turing: a language for flexible probabilistic inference*. In: *International conference on artificial intelligence and statistics*, PMLR, pp. 1682–1690, doi:10.17863/CAM.42246.

[11] Zoubin Ghahramani (1997): *Learning dynamic Bayesian networks*. In: *International School on Neural Networks, Initiated by IIASS and EMFCSC*, Springer, pp. 168–197, doi:10.1007/BFb0053999.

[12] Malik Ghallab, Dana S. Nau & Paolo Traverso (2016): *Automated Planning and Acting*. Cambridge University Press, doi:10.1017/CBO9781139583923.

[13] Félix Ingrand & Malik Ghallab (2017): *Deliberation for autonomous robots: A survey*. Artif. Intell. 247, pp. 10–44, doi:10.1016/j.artint.2014.11.003.

[14] Charles Lesire, David Doose & Christophe Grand (2020): *Formalization of robot skills with descriptive and operational models*. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 7227–7232, doi:10.1109/IROS45743.2020.9340698.

[15] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan & François Felix Ingrand (2010): *GenoM3: Building middleware-independent robotic components*. In: *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, IEEE, pp. 4627–4632, doi:10.1109/ROBOT.2010.5509539.

[16] Eitan Marder-Eppstein (2021): *ROS Move-Base*. Available at `http://wiki.ros.org/move_base`.

[17] Tim Niemueller, Till Hofmann & Gerhard Lakemeyer (2019): *Goal reasoning in the CLIPS Executive for integrated planning and execution*. In: *Proceedings of the International Conference on Automated Planning and Scheduling*, 29, pp. 754–763.

[18] Nils J Nilsson (1984): *Shakey the robot. Institute for Software Technology*.

[19] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler & Andrew Y Ng (2009): *ROS: an open-source Robot Operating System*. In: *ICRA workshop on open source software*, 3.2, Kobe, Japan, p. 5.

[20] ROBOTIS: *TurtleBot3 e-Manual*. Available at `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`.

[21] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S Polydoros, Bjarne Großmann, Ronald Petrick & Volker Krüger (2017): *SkiROS—a skill-based robot control platform on top of ROS*. In: *Robot operating system (ROS)*, Springer, pp. 121–160, doi:10.1007/978-3-319-54927-9_4.

[22] Scott Sanner (2010): *Implements a parser, simulator, and client/server evaluation architecture for the relational dynamic influence diagram language (RDDL)*. Https://github.com/ssanner/rddlsim.

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

[24] David Silver & Joel Veness (2010): *Monte-Carlo planning in large POMDPs*. In: *Advances in neural information processing systems*, pp. 2164–2172.

[25] Adhiraj Somani, Nan Ye, David Hsu & Wee Sun Lee (2013): *DESPOT: Online POMDP planning with regularization*. Advances in neural information processing systems 26.

[26] Dan R. Suissa (2022): *A short AOS overview video*. Available at `https://www.youtube.com/watch?v=8pqZADVBLPM`.

[27] David Hsu Wee Sun Lee Hanna Kurniawati (2008): *SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces*. In: *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, pp. 5427–5433, doi:10.15607/RSS.2008.IV.009.

[28] Or Wertheim (2010): *An experiment comparing the generative model sampling rate of RDDLSim's generic code vs. the AOS's domain-specific auto-generated code*. Https://github.com/ssanner/rddlsim.

[29] Or Wertheim (2021): *Armadillo experiment, detailed description*. Available at `https://github.com/orhaimwerthaim/AOS-experiments/tree/main/armadillo_pick`.

[30] Or Wertheim (2021): *ROSPlan PDDL experiment*. Available at `https://github.com/orhaimwerthaim/AOS-OtherSystems-ROSPlanExperimentPDDL/`.

[31] Or Wertheim (2022): *AOS Armadillo robot experiment video*. Available at `https://youtu.be/1OsTQ8a_N6c`.

[32] Or Wertheim (2022): *The AOS experiments documentation files*. Available at `https://github.com/orhaimwerthaim/AOS-experiments`.

[33] Or Wertheim (2022): *The AOS Franka Emika Panda CoBot robot experiment video*. Available at `https://www.youtube.com/watch?v=-2qN4WXdvj4`.

[34] Or Wertheim (2022): *The AOS Panda CoBot experiment video: the robot sometimes loses due to an inaccurate opponent model*. Available at `https://www.youtube.com/watch?v=R4dBrP7SLe8`.

[35] Håkan LS Younes & Michael L Littman (2004): *PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects*. Techn. Rep. CMU-CS-04-162 2, p. 99.