An Interaction Specification Language for Robot Application Development

David Porfirio, Mark Roberts, Laura M. Hiatt
Navy Center for Applied Research in Artificial Intelligence
U.S. Naval Research Laboratory
Washington, D.C., USA
{david.j.porfirio2, mark.c.roberts20, laura.m.hiatt}.civ@us.navy.mil

***Abstract*—Robot programming languages that represent tasks as graph structures are both popular and accessible among programming novices and experts. However, these languages are largely decoupled from robots' automated task planning capabilities, rendering developers unable to explicitly leverage their robot's ability to plan its own actions. We thereby created the *Interaction Specification Language* (ISL), which enables developers to import and apply elements from a robot planning domain in a graph-based programming paradigm. For developers, ISL provides flexibility in the reliance on automated planning. For researchers, the release of our open-source ISL lexer and parser is intended to promote standardization and test-driven development. We additionally provide a metric by which ISL programs can be evaluated.**

***Index Terms*—programming languages; planning; robotics**

I. Background and Summary

Robot application development (RAD) is the process of constructing robot *programs*, or static specifications of a robot's behaviors. These specifications often include control flows of the robot's actions, decisions, and goals, as well as how the robot should interpret external stimuli. A key factor that distinguishes RAD from other approaches (*e.g.,* policy learning and automated planning) is the substantial human involvement in control flow creation. RAD tools must capture human input that can assume several different forms, such as visual symbolic input [1], [2], tangible input [3], programming by demonstration [4], programming in augmented reality [5], [6], and natural language instructions [7]. Many of these tools directly represent robot control flows as, or can easily translate them into, graph-based structures (*e.g.,* state machines). These structures are both popular [8] and accessible [9].

As the autonomous capabilities of robots increase, however, modern RAD languages must keep pace with robots' ability to plan and act without needing hard-coded human input. Presently, there is a lack of standardized graph-based RAD languages that explicitly link to these planning capabilities. Without explicitly linking from a RAD language to a planning domain, the RAD language requires additional compilation or interpretation to be executed at runtime.

To tighten the connection between RAD and planning, we present Version 1.0 of the *Interaction Specification Language* (ISL). Depicted in Figure 1, ISL is intended to exist as a common backend language for RAD tools. Our core technical contribution is that ISL wraps a popular language for specifying planning problems, namely the Planning Domain Definition Language (PDDL) [10], to allow users to specify lifted, graph-based configurations of actions or goals. Rather than specifying a single goal (*i.e.,* a set of goal states) as with existing planning languages like PDDL, ISL enables users to chain both goals and actions together as paths through a finite state machine. In effect, ISL treats the connection between any two nodes as an individual planning problem going from the first node's goal conditions (or implied goal conditions if the node contains an action) to those of the second. At runtime, upon achieving the goal conditions of one node, the robot then transitions to executing the next plan heading to the next node.

ISL poses several benefits to RAD. For developers, the linkage of ISL to a planning language affords developers flexibility over the granularity of how ISL programs are specified. That is, developers are free to specify only a few high-level goals and let a task planner resolve the robot's runtime behaviors; alternatively, users can hard code incremental actions for the robot to execute in sequence. Choosing between goals and actions affords users further control over program interpretation at runtime. With actions, users enforce that the robot must "do" something; with goals, users enforce that the robot must "achieve" something irrespective of what it does. For the research community, we intend for our open-sourced ISL parsing software to facilitate test-driven development and reproducibility of RAD tools. Furthermore, we provide a metric that researchers can use to evaluate ISL programs.

Close viable alternatives to ISL exist from the task and motion planning community, such as expressing temporal goal orderings in linear temporal logic (LTL) [11]. However, our choice of ISL syntax is inspired by accessible RAD languages that more explicitly encode execution flow. Additionally, *agent planning programs* bear representational similarity to ISL [12] but have not yet resulted in a standardized RAD language.

In prior work, we proposed guidelines for ISL [13] and used an early version of ISL to represent programs created with the *Polaris* RAD tool [14]. Until now, the language itself and supporting code have not yet been released. Included in this release are an ISL lexer and parser, software that represents parsed ISL programs, and test cases that show ISL usage.

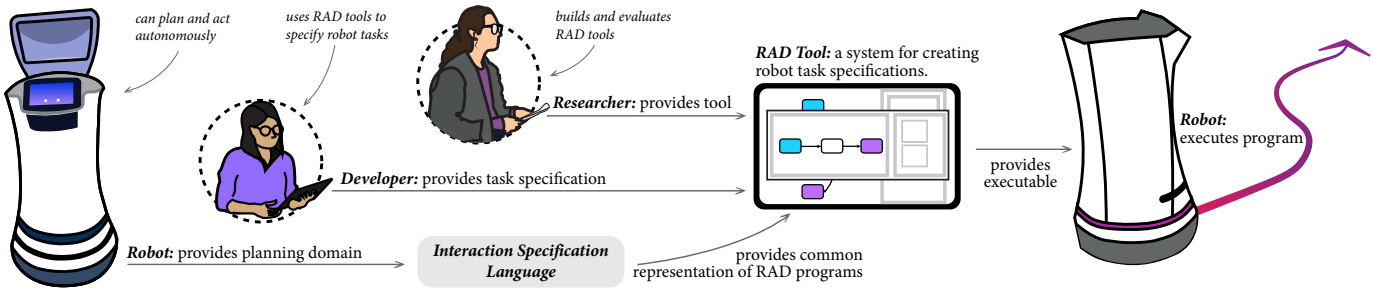

Fig. 1. Our vision of ISL's place in the robot application development pipeline. RAD researchers (top center) create tools for developers (left) to use. ISL provides a common, underlying program representation for these tools (bottom center). Programs written in ISL must accept a planning domain as input (bottom left). We envision different robot platforms providing their own planning domains. The robot can then execute ISL programs (right).

II. Purpose

ISL is intended to assist both robot application *developers* and RAD *researchers* in four different ways. First, derived from our technical contribution, ISL provides flexibility for robot application *developers*. Namely, developers are free to specify as much or as little detail as desired, depending on the needs of the context. For example, robots assisting with aircraft maintenance might require exact, hard-coded instructions due to the task's precise nature, whereas robots offering food deliveries at caregiving facilities might instead require smaller sets of higher-level goals to preserve the robot's adaptability in an inherently dynamic environment.

Second, as depicted in Figure 1, ISL is intended to promote standardization by providing a common language for RAD *researchers*. Third, we intend for our release of the test cases that guided ISL's creation to promote researcher adoption of test-driven development. Lastly, ISL is intended to assist researchers evaluate their RAD systems. We demonstrate an evaluation metric for ISL programs in §III-D.

III. Characteristics

The characteristics of ISL are twofold—(1) a *grammar* that defines the ISL language for expressing programs within a computational representation (called *goal automata*), and (2) a lexer and parser that interpret the language.

To assist in describing the grammar, consider a mobile water delivery robot. Various entities exist in the robot's environment, such as locations, containers, and people. The robot can move around, fetch and fill containers with water, and put the containers down. The robot can also sense its environment. Figure 2 depicts a planning domain snippet in PDDL-like syntax, including operators that describe what the robot can do and predicates that describe what it can sense.

The predicates, operators, and entities in a PDDL planning domain and problem description merely describe the building blocks that can be used to construct a full waterbot ISL program, depicted in Figure 3. The role of ISL is to instantiate and assemble these components into a control flow. In effect, rather than expressing just a single goal as with PDDL, developers can express an automaton of temporally linked goals and actions. Transitioning between an ISL-specified goal or action requires invoking a task planner.

```
1   domain waterbot
2   (predicates
3      (agentHas ?a - agent, ?i - item)
4      (agentNear ?a - agent, ?l - loc)
5      (isFull ?c - container)
6      ...
7   )
8   operator moveTo(?a - agent, ?l - loc) ...
9   operator grab(?a - agent, ?i - item) ...
10  operator put(?a - agent, ?i - item) ...
11  operator fill(?a - agent, ?i - item, ?s -
        source) ...
12  ...
```

Fig. 2. PDDL-like domain snippet for waterbot. Ellipses (...) indicate gaps in the code. The full domain would include operator preconditions and effects, and the full problem description would include entities and their starting states.

```
1   import waterbot
2
3   labels
4      ready: [predicate: agentHas,
5              params: [robot, cup] &
6              predicate: isFull,
7              params: [cup] &
8              predicate: agentNear,
9              params: [robot, person]
10            ],
11     delivered: [predicate: agentHas,
12                 params: [person, cup]
13                ],
14     athome: [action: moveTo,
15              params: [robot, home]
16             ]
17  endlabels
18
19  module
20     st: [0: init, 1: ready, 2: athome];
21     guard: [0: delivered]
22
23     [] 0 -> 1;
24     [] 1 & guard=0 -> 2;
25  endmodule
```

Fig. 3. ISL program for a water delivery robot.

A. Grammar

Our description of the ISL grammar follows Extended Backus–Naur Form (EBNF). Brackets (*i.e.,* "[]") represent optional notation, braces (*i.e.,* "{}") represent zero or more of the enclosed notation, and parentheses (*i.e.,* "()") group notation together. Our description omits minor details for

readability, such as semicolons, but these details are present in the delivery robot example (Figure 3). Also note that some notional details of the grammar have already been described in prior work [13], but in the course of its formalization, ISL exhibits several substantial changes from the notional version.

To begin, programs written in ISL are structured within four distinct sections—(1) *import* statements, (2) zero or more *labels*, (3) a *module*, and (4) an optional set of *options*.

$$\begin{aligned} program \;:=\; & \textbf{import } id \\ & \textbf{labels } \{label\} \textbf{ endlabels} \\ & \textbf{module } module \textbf{ endmodule} \\ & [\textbf{options } \{options\} \textbf{ endoptions}] \end{aligned}$$

The *import* statement loads PDDL domain and problem files into ISL and is depicted on Line 1 of the delivery robot code. The domain file is necessary to provide ISL with the operators that the robot can perform and predicates that describe the observable characteristics of the world. The problem file provides ISL with the initial state of the world.

Labels are defined after the import statement, shown on Lines 3 through 17. Labels are similar to the token with the same name in the PRISM model checker [15], and facilitate grouping together ground elements as sets. For example, let $a :=$ `robotIn`$(kitchen)$ and $b :=$ `isActive`$(coffee\ maker)$ be ground predicates. With labels, we can describe both predicates simultaneously, such as **label** $c := a \wedge b$. Formally, labels can be invoked via the syntax below.

$$\begin{aligned} label \;&:=\; id : [operator] \mid predicateList \\ predicateList \;&:=\; \{predicate\} \end{aligned}$$

Labels include zero to one *operators* and zero or more *predicates*. Intuitively, within a label, while zero or more predicates can be true at the same time, the robot can only be performing at most one action (*i.e.,* an instantiated operator) at any given moment. In Figure 3, the `ready` token (Line 4) depicts a label with three predicates: the robot has a cup, the cup is filled with water, and the robot is near the person. The `delivered` label (Line 11) is then comprised of a single predicate: the person has the cup. In contrast to both `ready` and `delivered`, the `athome` label (Line 14) depicts a label with one operator[1] for the robot to perform: go home. Operators and predicates in ISL are defined in EBNF, each as combinations of one symbol and zero or more parameters.

$$\begin{aligned} operator \;&:=\; \textbf{action} : id,\ \textbf{params} : \{id\} \\ predicate \;&:=\; \textbf{predicate} : id,\ \textbf{params} : \{id\} \end{aligned}$$

In Figure 3, Lines 19 through 25 depict the *module*. Modules define *goal automata*, which instantiate labels.

$$module \;:=\; states\ [guards]\ \{transition\}$$

[1]Operators use the "action" token in ISL script.

Intuitively, a goal automaton is a transition system [16], but where nodes in the transition system can be defined in terms of high-level goals or low-level actions. Transition systems, and thus goal automata, define processes or procedures as sets of nodes (called *states* in transition system vocabulary) and *transitions* between these nodes. Transitions may optionally be *guarded*; that is, a specific condition must be true in order for the transition to take place. Transition systems additionally include an initial state, a set of atomic propositions (*i.e.,* labels), and a mapping function that assigns labels to states. As more than one state can be assigned the same label, labels themselves cannot be treated as states. Thus, the first step in defining the module is to initialize the set of states and map them to labels.

$$\begin{aligned} states \;&:=\; \textbf{st} : (int : \textbf{init}, \{int : id\}) \\ guards \;&:=\; \textbf{guard} : \{int : id\} \end{aligned}$$

Transitions between states are formally defined below.

$$transition \;:=\; int\ [\textbf{guard} = int] \;\rightarrow\; int$$

For example, Lines 19 through 25 of Figure 3 assemble a small transition system that instantiates these labels. From the initial state (defined as 0 via the `init` token on Line 20), the robot will first attempt to achieve `ready` (Line 23). Crucially, a contribution of ISL is that developers need not define the sequence of steps for the robot to perform in order to achieve `ready`. Rather, the developer leverages the robot's ability to plan and act autonomously by simply defining the objective for the robot to achieve in terms of goal predicates.

After achieving `ready`, the `delivered` label is used as a guard for transitioning to state 2. Specifically, the expression on line 24 states, "from state 1, once the person has received the cup of water, the robot must travel back to home base."

Lastly, *options* encompass an optional list of parameters that affect how ISL programs are parsed or interpreted at runtime and are similar to the "contingency sheet" proposed in our prior work [13]. An example option is *conditional effects*, which allows PDDL files to be imported that use conditionals in operator effects. Figure 3 does not include any options.

B. Lexer and Parser

ISL programs *define* goal automata, which must still be compiled to create a goal automaton *instance*. In the first step of compilation, a lexer reads user-written programs and outputs tokens that can be read by a parser. The parser then assembles an abstract syntax tree from the tokens. The abstract syntax tree is then used to assemble a goal automaton. We use an off-the-shelf Python library, PLY [17], for lexing and parsing and our own custom code for goal automata assembly.

C. Integration with a Task Planner

Once parsed to goal automata, ISL programs are ready for planner integration. Prior work details how existing approaches can be used to create plans from goal automata [14]—namely, each ISL state is treated as a separate planning task. In the waterbot example portrayed in Figure 3, the plan that

results from the goal automaton expressed via ISL script could involve the following sequence of actions to achieve the `ready` label: (1) **`moveTo`**(*robot, cup*), (2) **`grab`**(*robot, cup*), (3) **`moveTo`**(*robot, sink*), (4) **`fill`**(*robot, cup, sink*), and (5) **`moveTo`**(*robot, person*). When the person has grabbed the cup, the robot will perform a final action: (6) **`moveTo`**(*robot, home*).

The algorithm for creating plans from goal automata is not a contribution of this paper, though our ISL parsing software takes steps towards facilitating this creation. In particular, the ISL parser is already tightly coupled with an off-the-shelf task planner—the *Unified Planning* library [18]. The ISL parser uses the *Unified Planning* library in order to assist representing ISL programs as goal automata. Specifically, the domain and problem files imported by ISL script are parsed by the *Unified Planning* library, and the resulting sets of operators, predicates, and entities are supplied to ISL parser. ISL predicates are then represented as *Unified Planning* fluents, operators as *Unified Planning* actions, and entities as *Unified Planning* objects.

D. Case Study on Evaluating ISL Programs

We described our vision for how ISL can facilitate the standardization of RAD tools (see Figure 1) and afford flexibility to developers. In addition, ISL has enabled us to create an evaluation metric for programs written in it based on planner reliance. By calculating the ratio of actions in a generated plan to user-specified ISL states, we can determine whether a developer relies more (*i.e.,* a high ratio of actions to ISL states) or less (*i.e.,* a low ratio of actions to ISL states) on the planner. In the example in §III-C, this ratio is 6 planner actions to the 2 ISL states defined on Line 20 of Figure 3 (not including the initial state). Thus, for every user-specified ISL state, the robot will perform an average of 3 actions.

As a case study, consider the RAD tool from prior work, *Polaris* [14], which tested a full version and an ablated version of the tool in a between-subjects user study. Both versions use the ISL as a backend program representation. In both versions, users specified goal automata to satisfy a prompt. In the full version, the average action-state ratio of user-created programs is 2.99 ($SD = 2.23$), indicating that for every state specified in ISL, the robot will perform about 3 actions. By contrast, the ablated version exhibits an average ratio of 2.43 ($SD = 2.06$). Therefore, on average, users of the full version more heavily utilized the planner. This difference is not statistically significant according to a Mann-Whitney U test.

IV. Code Structure

ISL software[2] includes three primary components—(1) the lexer and parser, (2) the goal automata representation, and (3) a test script. Calling the lexer and parser results in an ISL program being represented as a goal automaton instance. Calling the test script allows multiple programs to be parsed, which is useful for testing extensions to the language.

The software is written in the Python programming language for accessibility among researchers. For additional accessibility, we provide a suite of ISL test cases accompanied by importable PDDL domain and problem files. Test cases are bundled together within *groups*. ISL Version 1.0 is accompanied by a group of *general* test cases that check a wide variety of syntax, *waterbot* test cases that test ISL programs specified at a high, goal-oriented level, and *food-assembly* test cases that test ISL programs specified at a lower, step-by-step level.

[2]ISL software is available at https://github.com/dporfirio/ISL-Parser.

V. Usage

RAD researchers can use, modify, and extend ISL.

a) Parser Usage: ISL programs are written with the `.isl` extension. Developers or researchers who write ISL programs must be equipped with PDDL domain and problem files that define the predicates, operators, and entities available to the robot. Both files must be placed in the same directory and be named `domain.pddl` and `problem.pddl`. The *import* statement in Figure 3 thereby loads the path containing `waterbot/domain.pddl` and `waterbot/problem.pddl`. Developers must also have access to a command-line interface for parsing their programs. The command for parsing a program is shown below, where `program.isl` is the path to the ISL program.

```
python isl.py program.isl
```

ISL is equipped with a suite of test programs. The command below depicts how a batch of test cases can be run, in which `-g` is an optional argument for specifying a group of tests, and `groupname` specifies the name of the group.

```
python test.py [-g groupname]
```

The `tests/README.md` file in the ISL codebase details the procedure for writing new ISL test cases.

b) Maintaining and Extending ISL: We are releasing ISL under the MIT open-source license. Community-driven maintenance of ISL will follow a founder-led open-source governing model. In this model, community members can fork the ISL repository and make changes to how ISL programs are parsed and represented as goal automata. Community members may then submit merge requests to the public ISL repository to be reviewed by the authors. The authors will make additional changes to the public branch on a regular basis.

Ongoing work on ISL includes adding support for expressing current-state uncertainty [19], specifically to handle the case that the robot is unsure of the starting locations of known items in its environment. Other ongoing work includes the addition of *maintenance goals* to the goal automata representation to define goals that must be *maintained* throughout the remainder of a task. We also intend for ISL to support real-valued planning constraints in the future, as is often necessary for task and motion planning [20].

VI. Acknowledgements

This research was supported by the U.S. Naval Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Navy.

References

[1] S. Alexandrova, Z. Tatlock, and M. Cakmak, "Roboflow: A flow-based visual programming language for mobile manipulation tasks," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 5537–5544. [Online]. Available: http://doi.org/10.1109/ICRA.2015.7139973

[2] M. J.-Y. Chung, J. Huang, L. Takayama, T. Lau, and M. Cakmak, "Iterative design of a system for programming socially interactive service robots," in *Social Robotics*, A. Agah, J.-J. Cabibihan, A. M. Howard, M. A. Salichs, and H. He, Eds. Cham: Springer International Publishing, 2016, pp. 919–929. [Online]. Available: https://doi.org/10.1007/978-3-319-47437-3_90

[3] A. Kubota, E. I. C. Peterson, V. Rajendren, H. Kress-Gazit, and L. D. Riek, "Jessie: Synthesizing social robot behaviors for personalized neurorehabilitation and beyond," in *Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 121–130. [Online]. Available: https://doi.org/10.1145/3319502.3374836

[4] N. Patton, K. Rahmani, M. Missula, J. Biswas, and I. Dillig, "Programming-by-demonstration for long-horizon robot tasks," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632860

[5] Y. Cao, Z. Xu, F. Li, W. Zhong, K. Huo, and K. Ramani, "V.ra: An in-situ visual authoring system for robot-iot task planning with augmented reality," in *Proceedings of the 2019 on Designing Interactive Systems Conference*, ser. DIS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1059–1070. [Online]. Available: https://doi.org/10.1145/3322276.3322278

[6] E. Senft, M. Hagenow, R. Radwin, M. Zinn, M. Gleicher, and B. Mutlu, "Situated live programming for human-robot collaboration," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 613–625. [Online]. Available: https://doi.org/10.1145/3472749.3474773

[7] J. X. Liu, A. Shah, G. Konidaris, S. Tellex, and D. Paulius, "Lang2ltl-2: Grounding spatiotemporal navigation commands using large language and vision-language models," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024, pp. 2325–2332. [Online]. Available: https://doi.org/10.1109/IROS58592.2024.10802696

[8] C. Street, Y. Warsame, M. Mansouri, M. Klauck, C. Henkel, M. Lampacrescia, M. Palmas, R. Lange, E. Ghiorzi, A. Tacchella, R. Azrou, R. Lallement, M. Morelli, G. I. Chen, D. Wallis, S. Bernagozzi, S. Rosa, M. Randazzo, S. Faraci, and L. Natale, "Towards a verifiable toolchain for robotics," *Proceedings of the AAAI Symposium Series*, vol. 4, no. 1, pp. 398–403, Nov. 2024. [Online]. Available: https://doi.org/10.1609/aaaiss.v4i1.31823

[9] D. Glas, S. Satake, T. Kanda, and N. Hagita, "An interaction design framework for social robots," in *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011. [Online]. Available: https://doi.org/10.15607/RSS.2011.VII.014

[10] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003. [Online]. Available: https://doi.org/10.1613/jair.1129

[11] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[12] G. De Giacomo, A. E. Gerevini, F. Patrizi, A. Saetti, and S. Sardina, "Agent planning programs," *Artificial Intelligence*, vol. 231, pp. 64–106, 2016. [Online]. Available: https://doi.org/10.1016/j.artint.2015.10.001

[13] D. Porfirio, M. Roberts, and L. M. Hiatt, "Guidelines for a human-robot interaction specification language," in *2023 32nd IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 2023, pp. 1–8. [Online]. Available: https://doi.org/10.1109/RO-MAN57019.2023.10309563

[14] ——, "Goal-oriented end-user programming of robots," in *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 582–591. [Online]. Available: https://doi.org/10.1145/3610977.3634974

[15] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_47

[16] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT press, 2008.

[17] D. Beazley, "Ply." [Online]. Available: https://www.dabeaz.com/ply/

[18] A. Micheli, A. Bit-Monnot, G. Röger, E. Scala, A. Valentini, L. Framba, A. Rovetta, A. Trapasso, L. Bonassi, A. E. Gerevini, L. Iocchi, F. Ingrand, U. Köckemann, F. Patrizi, A. Saetti, I. Serina, and S. Stock, "Unified planning: Modeling, manipulating and solving ai planning problems in python," *SoftwareX*, vol. 29, p. 102012, 2025. [Online]. Available: https://doi.org/10.1016/j.softx.2024.102012

[19] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1194–1227, 2013. [Online]. Available: https://doi.org/10.1177/0278364913484072

[20] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, no. Volume 4, 2021, pp. 265–293, 2021. [Online]. Available: https://doi.org/10.1146/annurev-control-091420-084139