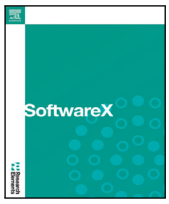
SoftwareX 29 (2025) 102012

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

# Unified Planning: Modeling, manipulating and solving AI planning problems in Python

Andrea Micheli [a,*], Arthur Bit-Monnot [b], Gabriele Röger [c], Enrico Scala [d], Alessandro Valentini [a], Luca Framba [a], Alberto Rovetta [d], Alessandro Trapasso [g], Luigi Bonassi [d], Alfonso Emilio Gerevini [d], Luca Iocchi [g], Felix Ingrand [b], Uwe Köckemann [e], Fabio Patrizi [g], Alessandro Saetti [d], Ivan Serina [d], Sebastian Stock [f]

[a] Fondazione Bruno Kessler, Trento, Italy
[b] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
[c] University of Basel, Switzerland
[d] University of Brescia, Italy
[e] Örebro University, Sweden
[f] DFKI, Osnabrück, Germany
[g] Sapienza University of Rome, Italy

ARTICLE INFO

*Keywords:*
Automated planning and scheduling
Python library
Interoperability

ABSTRACT

Automated planning is a branch of artificial intelligence aiming at finding a course of action that achieves specified goals, given a description of the initial state of a system and a model of possible actions. There are plenty of planning approaches working under different assumptions and with different features (e.g. classical, temporal, and numeric planning). When automated planning is used in practice, however, the set of required features is often initially unclear. The Unified Planning (UP) library addresses this issue by providing a feature-rich Python API for modeling automated planning problems, which are solved seamlessly by planning engines that specify the set of features they support. Once a problem is modeled, UP can automatically find engines that can solve it, based on the features used in the model. This greatly reduces the commitment to specific planning approaches and bridges the gap between planning technology and its users.

Code metadata

| | |
|---|---|
| Current code version | v1.1 |
| Permanent link to code/repository used for this code version | https://github.com/ElsevierSoftwareX/SOFTX-D-24-00504 |
| Permanent link to Reproducible Capsule | https://doi.org/10.5281/zenodo.11127268 |
| Legal Code License | Apache 2.0 license |
| Code versioning system used | git |
| Software code languages, tools, and services used | Python |
| Compilation requirements, operating environments & dependencies | Linux, MacOS, Microsoft Windows |
| If available Link to developer documentation/manual | https://unified-planning.readthedocs.io/ |
| Support email for questions | unified-planning@googlegroups.com |

## 1. Motivation and significance

Automated planning is the area of artificial intelligence concerned with identifying a suitable course of action to achieve a goal based on a predictive model of the environment. Due to its abstract nature, planning technology is relevant to a wide range of application areas, such as agile manufacturing [1], space operations [2,3], robotics [4] or logistics [5].

* Corresponding author.
*E-mail address:* amicheli@fbk.eu (Andrea Micheli).

https://doi.org/10.1016/j.softx.2024.102012
Received 20 September 2024; Received in revised form 26 November 2024; Accepted 9 December 2024
Available online 19 December 2024

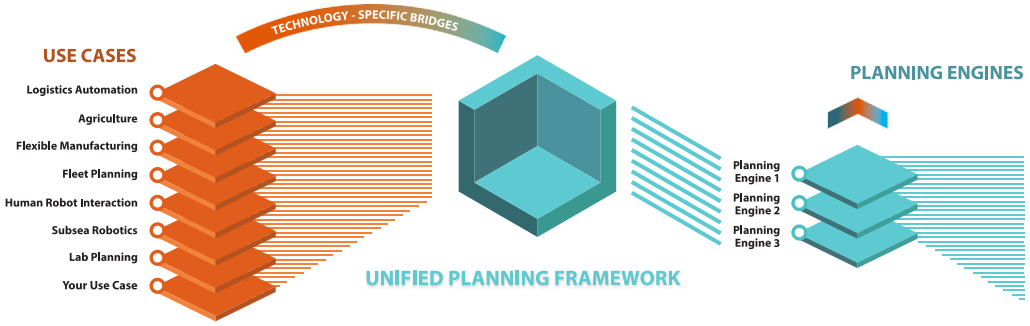



**Fig. 1.** The unified planning framework in context.

To illustrate the general concept, consider a smart warehouse environment with a number of storage locations and one packing location. Robots can carry boxes from the storage to the packing location and back, and the task is to get a set of products to the packing location. The planning task can then be described by the current state of the world (the *initial state*, e.g. the locations of all products and robots), the objective (a *goal* condition, e.g. having the required products at the packing location, regardless of the locations of the robots) and the possible *actions*. In such a scenario, we would expect `move` actions to model the robots movements between locations or `load` actions to model the act of loading a box containing a required product onto a movable robot.

World states of planning tasks are typically described by means of so-called *fluents*. For example, there could be a boolean fluent `carrying(robot,box)` that is true whenever a robot is carrying a specific box, and a symbolic fluent `location(robot)` whose value is the current position of the robot. Actions typically specify the states in which they are applicable through *conditions* on fluents, while their *effects* capture how they would alter the current state when applied. In our example, an action for *loading* a box on a robot would then be applicable whenever the robot is empty and at the same location as the box; after its application the box would be carried by the robot, which in turn would no longer be empty.

The task for the planning system is to identify a *plan*, i.e. a suitable course of action that leads from the initial state to the goal. The exact form of the plan depends on the details of the planning task. For example, in classical planning, actions are instantaneous and the plan is simply a *sequence* of actions; in temporal planning instead, actions have an associated duration and their execution can overlap, so a plan would be an exact schedule that specifies a start time for every action instance in the plan.

Even though there are many planning techniques and tools available, it is still hard to apply them in practice. The *Unified Planning* (UP) Python library, developed as part of the AIPlan4EU project funded by the European Union, helps in overcoming some of the major challenges.

The first challenge is the **modeling of a real-world problem as a planning task**, defining suitable fluents and actions. Planning systems typically expect the input task as a text file in some description language (such as the Planning Domain Definition Language (PDDL) [6,7] or the Action Notation Modeling Language (ANML) [8]), which is usually created manually or by some dedicated piece of code for the specific application. The UP library enables the user to model the planning task programmatically with Python; it retains the formality of the planning model, but replaces a concrete formal description language with an API in a commonly used programming language. This makes it easier to incorporate data from different sources of information, and allows the modeling of the task without committing to a specific description language (Python knowledge is required, but this skill is arguably more available than niche modeling languages). The UP library also implements a number of task transformations that allow reformulating a task to make it suitable for a wider range of planning systems, e.g. by compiling away features that are convenient for modeling but not widely supported by the engines.

The second challenge for solving the task is to **select and run a suitable planning engine**. Planning systems typically only support a certain kind of planning tasks: in fact, different fragments of planning have different computational complexity (e.g. deciding plan existence for STRIPS classical planning is PSPACE-complete [9], whereas for numeric planning it is only decidable for finite domains [10,11]), so limiting a planner to a certain fragment allows using more specific and efficient approaches. For the user, however, it can be very hard to understand what systems can be used for their problem at all. Moreover, even as suitable planners are selected, each engine comes with its own installation and usage instructions, so there is a substantial setup effort before actually solving the task with the engine.

With the UP library, all of this requires just a few lines of code. Its plugin system allows developers to easily make their planning engine available, and several planning systems have already been integrated by the AIPlan4EU partners and through the AIPlan4EU open call program. A user can now easily install such planning engines with the Python package installer `pip`. The UP analyzes which engines are suitable for the modeled task, and the user can call them directly from the UP through a common interface, simplifying the experimentation with different engines or running several of them in parallel. Moreover, the resulting plan is a structured Python object, facilitating further processing in the desired context.

Fig. 1 shows the UP framework in the overall context: it allows users to leverage planning technology without committing to any specific planning engines, which are however available via UP's API.

## 2. Related work

Several tools for modeling planning problems exist, in particular for creating models in PDDL [6,7], an input language supported by most planners as it is used in International Planning Competitions.

The model acquisition tool *itSIMPLE* [12] allows modeling tasks in the diagram-based UML language, and provides analyses based on Petri nets.

*Tarski* [13] is a Python library for modeling, manipulating and analyzing planning tasks that can also parse languages other than PDDL, such as functional STRIPS [14] or RDDL [15]. The tool is intended for planning researchers and can be used for reachability analysis or problem reformulations like the showcased compilation from deterministic conformant planning (with uncertainty about the initial state) to classical planning.

PDDL4J [16] is a Java PDDL parser delivering an internal representation of the parsed task that can be used via an API. For example, the PlanX Toolbox [17] uses PDDL4J to build a composition of planning-based services over a Docker infrastructure.

An extension for PDDL exists for the Visual Studio Code editor [18]. Besides standard IDE support, it also provides plan visualization, support for regression tests of the model, and the possibility to use templating in PDDL problem files.

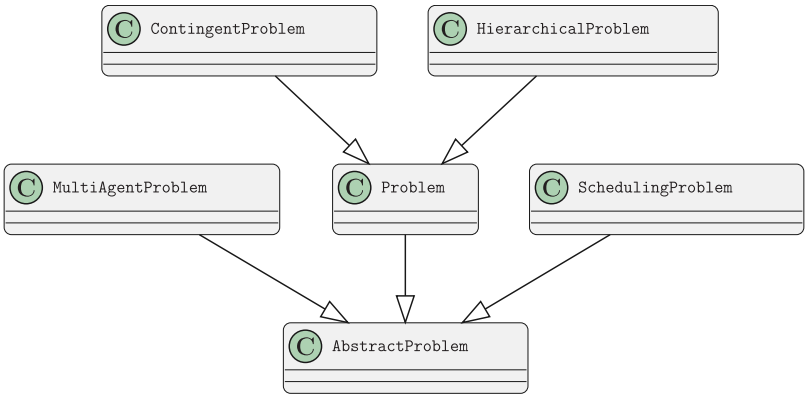



**Fig. 2.** Class diagram of the supported problems classes.

The online suite *planning.domains*[1] [19] includes a PDDL editor, a comprehensive database of benchmark tasks, and a planner-in-the-cloud service (with limited time and memory resources) accessible through a RESTful API. Web Planner [20] is a similar tool that provides several visualizations of the search space of a problem in addition to basic PDDL editing and remote solving. These tools are indeed useful for first tests on small instances, but do not scale enough to process larger instances or production environments.

Similarly to UP, the *planutils* [21] project aims at facilitating running different planning engines. It provides several environments for uniformly interacting with different planning systems, overcoming each one's peculiarities in terms of build instructions, dependencies, or calling interface. Individual planning systems are integrated as packages (typically Singularity/Apptainer images).

Furthermore, there are several frameworks for the creation of planning algorithms in specific contexts. In particular, LAPKT [22] is a framework to build classical planners, EUROPA [23] provides the infrastructure for timeline-based temporal planners, F4Plan [24] allows the integration in the EnTiMid home-automation software system, and PyHOP [25] is a Python implementation of a hierarchical planner where a problem can be modeled directly in code.

## 3. Software description

### 3.1. Software architecture

In this section, we focus on the core concepts required to leverage the capabilities of UP.

The core modeling feature provided by UP is the representation of planning problems. We tackle a variety of types of planning problems with different expressiveness. In particular, the UP library currently supports the following classes of planning problems: Classical, Numeric, Temporal, Scheduling, Multi-Agent, Hierarchical, Task and Motion Planning (TAMP) and Contingent. In order to properly represent this variety, we created the class structure depicted in Fig. 2, where each class represents a kind of planning problems (`Problem` can represent multiple kinds: classical, numeric, temporal and TAMP).

All the classes of planning problems inherit from `AbstractProblem`, which is the class used for the generic interfaces in our plugin system. This class hierarchy is meant to facilitate adding new types of problems and to make the library as generic as possible. Moreover, we used the "mixin" design pattern to avoid code duplication between sibling classes: this is an implementation detail, but greatly simplifies the maintainability of the code. One of the key elements of the problem specifications is the `ProblemKind` class (automatically computed by all the planning problems classes via the kind property), implemented as a collection of flags[2] that lists the modeling features used in any problem specification. Each integrated engine must specify which of those features it supports, so that the library can determine its applicability with respect to a given problem.

### 3.2. Software functionalities

Operation Modes (OMs) represent and standardize the possible interactions with a planning engine. Each OM defines an API that an engine willing to support the OM shall implement: in this way, engines declaring to support the same OM can be used interchangeably. In addition, the interface of each OM includes a set of methods designed to be mutually non-interfering and documented with their respective assumptions and guarantees, allowing an engine to support multiple OMs. Moreover, each engine will inherit from the `Engine` abstract class, which provides the basic machinery for the plug-in mechanism and for declaring the supported kinds of problem. The currently available OM are:

- **OneshotPlanner**: single call to a planning engine that, given a problem, returns a plan or a failure response;
- **PlanValidator**: given a planning problem and a plan, indicates whether the plan is valid;
- **SequentialSimulator**: given a problem, provides functionalities to explore the reachable states;
- **Compiler**: transforms a given problem into an equivalent one, performing some kind of rewriting;
- **AnytimePlanner**: iteratively generates solutions to a planning problem (e.g. incrementally better plans);
- **Replanner**: generates plans when a base problem is changed, possibly re-using previous computation;
- **PlanRepairer**: given a planning problem and a possibly invalid plan, returns a valid plan;
- **PortfolioSelector**: given a planning problem, selects the best engines to solve the problem.

All the OMs can be invoked using the `Factory` class, which implements the engine selection mechanism based on our plug-in system. Each environment contains a private `Factory`, allowing different subsets of engines and selection priorities to coexist in the same process: in addition, each OM is also exposed as a top-level function for the global environment in the `unified_planning.shortcuts` module. When calling the OM constructor (either from the factory or from the shortcuts), it is always possible to specify the name of the engine to be used (i.e. to force the selection of a specific engine, if available)

[1] https://planning.domains

[2] See https://unified-planning.readthedocs.io/en/latest/problem_representation.html#problem-kinds for the full details.

```python
import unified_planning as up
from unified_planning.shortcuts import *

# Declaring types
Location = UserType("Location")

# Creating problem variables
robot_at = Fluent("robot_at", BoolType(), location=Location)
battery_charge = Fluent("battery_charge", RealType(0, 100))

# Creating actions
move = InstantaneousAction("move", l_from=Location, l_to=Location)
move.add_precondition(GE(battery_charge, 10))
move.add_precondition(robot_at(move.l_from))
move.add_precondition(Not(robot_at(move.l_to)))
move.add_effect(robot_at(move.l_from), False)
move.add_effect(robot_at(move.l_to), True)
move.add_effect(battery_charge, Minus(battery_charge, 10))

# Declaring objects
l1 = Object("l1", Location)
l2 = Object("l2", Location)

# Populating the problem with initial state and goals
problem = Problem("robot")
problem.add_fluents([robot_at, battery_charge])
problem.add_action(move)
problem.add_objects([l1, l2])
problem.set_initial_value(robot_at(l1), True)
problem.set_initial_value(robot_at(l2), False)
problem.set_initial_value(battery_charge, 100)
problem.add_goal(robot_at(l2))

# Solving via OneshotPlanner Operation Mode
with OneshotPlanner(problem_kind=problem.kind) as planner:
    result = planner.solve(problem)
    if result.status in up.engines.results.POSITIVE_OUTCOMES:
        print(f"{planner.name} found this plan: {result.plan}")
    else:
        print("No plan found.")
```

**Fig. 3.** Simple example of a planning problem modeled using the UP Python API and solved by one of the engines via the "OneshotPlanner" OM.

or to let UP select an engine automatically, by specifying only the `ProblemKind` (which can be retrieved automatically from a problem `p` using the `p.kind` property).

A Custom Resolution Strategy (CRS) is a procedural specification (i.e. a piece of Python code) that can be used to guide an engine or to specify some action behaviors. The UP library currently offers two types of CRS: *Simulated Effects* and *Custom Heuristics*. The former allows for the specifications of changes in the values of the fluents for which only a Python function is provided. This is useful when complex dynamics cannot be faithfully modeled and can only be evaluated (e.g. an effect can be modeled as a neural network). With the latter, the user can guide the search in the underlying planning engine by providing a Python function estimating the distance from a given state to the goal.

In addition to the planning API, the library also offers a Protobuf[3] interface for inter-process integration.

## 4. Illustrative examples

Fig. 3 shows a small example of the API offered by the UP. More specifically, the displayed Python code models a robot moving on a graph as a simple numeric planning problem. The code constructs a model of a system and then solves a planning problem on this model. The problem state is described using a predicate `robot_at`, indicating the current location of the robot, and a numeric fluent `battery_charge`, modeling the remaining battery in percentage. In this simple problem, we have a single action `move` that moves the robot between two locations, `l1` and `l2`. The problem is then fed into a planner, which is selected automatically by the library. While this is a very minimal example constructed for the sake of brevity, many more interactive demonstrations are available at https://unified-planning.readthedocs.io/en/latest/examples.html.

## 5. Impact

The Unified Planning (UP) project provides several benefits to the research community and facilitates the efficient and effective use of automated planning technologies in various contexts. Here follow three notable contexts where the UP has already yielded some impact or can be utilized.

**Knowledge compilation techniques**: UP offers a programmatic interface for specifying planning problems of different types and paradigms. This flexibility enables researchers to easily transform a problem expressed in one formulation into another, facilitating the application of techniques that may not be available for the original formulation. For example, it is known that planning problems with conditional effects can be transformed into equivalent problems without such effects [26, 27]: this allows the use of planning tools that do not provide native support for conditional effects. Other compilations are also implemented; for example, UP has been used to transform problems with numeric temporal constraints expressed in PDDL3 into problems without such constraints [28].

**Integration of planning into complex systems**: UP serves as an interface for integrating planning technologies into broader applications, including robotics. By using UP, developers can easily incorporate

[3] https://protobuf.dev

planning technologies into robotic platforms, reducing integration complexity and allowing the platform to adopt a wider range of planning engines, enhancing flexibility and adaptability. For example, UP is used to interface an underwater robotic platform for surveillance with planning and replanning facilities [29], allowing automatic adaptation and long-term autonomy. Similarly, UP is employed as a middle-ware to bridge the gap between automated planning and embedded systems [30]. Finally, the Protobuf interface is used for the UP integration in the AIBuilder[4] component of the European AI-On-Demand Platform,[5] which allows the creation of AI pipelines involving planning components.

**Homogeneous approach to using planning engines**: UP provides users with a consistent approach to utilizing planning engines that may be written in different programming languages. This streamlines experimental analysis, particularly in scenarios like planning competitions (e.g., [31]): researchers can seamlessly switch between different planning engines without needing to adapt to different programming languages or interfaces. This capability contributes to speeding up experimental analyses and fostering collaboration.

UP is an open-source project available on GitHub (https://github.com/aiplan4eu/unified-planning): its popularity in the research community and its overall adoption across research groups have steadily increased over time. At the time of writing, the project counts 194 stars and 42 forks.

## 6. Conclusion

This paper has presented the Unified Planning (UP) library that provides programmatic access to planning technologies through Python. UP offers a comprehensive Python API for modeling and manipulating different kinds of planning problems. In addition, the library also provides a plugin system for easily integrating planning engines over standardized APIs: at the time of writing, more than 40 planning engines of different kind have been integrated.

## CRediT authorship contribution statement

**Andrea Micheli:** Writing – review & editing, Writing – original draft, Software, Project administration, Methodology, Funding acquisition, Conceptualization. **Arthur Bit-Monnot:** Writing – review & editing, Software, Methodology, Conceptualization. **Gabriele Röger:** Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization. **Enrico Scala:** Writing – review & editing, Writing – original draft, Methodology, Conceptualization. **Alessandro Valentini:** Software, Project administration, Methodology, Conceptualization. **Luca Framba:** Software. **Alberto Rovetta:** Software, Conceptualization. **Alessandro Trapasso:** Software, Methodology, Conceptualization. **Luigi Bonassi:** Software, Conceptualization. **Alfonso Emilio Gerevini:** Methodology, Conceptualization. **Luca Iocchi:** Methodology, Conceptualization. **Felix Ingrand:** Writing – review & editing, Methodology, Conceptualization. **Uwe Köckemann:** Writing – review & editing, Software, Methodology, Conceptualization. **Fabio Patrizi:** Methodology, Conceptualization. **Alessandro Saetti:** Writing – review & editing, Methodology, Conceptualization. **Ivan Serina:** Writing – review & editing, Methodology, Conceptualization. **Sebastian Stock:** Writing – review & editing, Methodology, Conceptualization.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Andrea Micheli reports financial support was provided by European Commission. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[4] https://aiexp.ai4europe.eu

[5] https://aiod.eu

## Acknowledgments

We are grateful for the AIPlan4EU project support, which was funded by the European Union's Horizon 2020 research and innovation programme under GA n. 101016442. Andrea Micheli is also supported by the STEP-RL project funded by the European Research Council under GA n. 101115870. We also thankful to Daniele Calisi, Guglielmo Gemignani, Malte Helmert, Joachim Hertzberg, Oscar Lima, Federico Pecora, Alessandro Saffiotti, Selvakumar Hastham Sathiya Satchi Sadanandam, Alexander Sung, Florent Teichtel-Koenigsbuch, Elisa Tosello, and Paolo Traverso for fruitful discussions that helped steering the development of the UP library during the AIPlan4EU project.

## References

[1] Ruml W, Do MB, Fromherz MP. On-line planning and scheduling for high-speed manufacturing. In: International conference on automated planning and scheduling. 2005, p. 30–9.

[2] Vaquero T, Chien SA, Agrawal J, Chi W, Huntsberger TL. Temporal brittleness analysis of task networks for planetary rovers. In: International conference on automated planning and scheduling. 2019, p. 564–72, URL https://ojs.aaai.org/index.php/ICAPS/article/view/3553.

[3] Pralet C, Doose D, Anxionnat J, Pouly J. Building resource-dependent conditional plans for an earth monitoring satellite. In: International conference on automated planning and scheduling. 2022, p. 490–8, URL https://ojs.aaai.org/index.php/ICAPS/article/view/19835.

[4] Ingrand F, Ghallab M. Deliberation for autonomous robots: A survey. Artificial Intelligence 2017;247:10–44.

[5] García J, Flórez JE, de Reyna ÁTA, Borrajo D, López CL, Olaya AG, et al. Combining linear programming and automated planning to solve intermodal transportation problems. European J Oper Res 2013;227(1):216–26. http://dx.doi.org/10.1016/j.ejor.2012.12.018.

[6] Fox M, Long D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. J Artificial Intelligence Res 2003;20:61–124.

[7] Gerevini AE, Long D. Plan constraints and preferences in PDDL3. Tech. Rep. R. T. 2005-08-47, University of Brescia, Department of Electronics for Automation; 2005.

[8] Smith DE, Frank J, Cushing W. The ANML language. In: The ICAPS-08 workshop on knowledge engineering for planning and scheduling. 2008.

[9] Bylander T. The computational complexity of propositional STRIPS planning. Artificial Intelligence 1994;69(1–2):165–204.

[10] Helmert M. Decidability and undecidability results for planning with numerical state variables. In: International conference on artificial intelligence planning and scheduling. 2002, p. 303–12.

[11] Gigante N, Scala E. On the compilability of bounded numeric planning. In: International joint conference on artificial intelligence. 2023, p. 5341–9.

[12] Vaquero TS, Silva JR, Ferreira M, Tonidandel F, Beck JC. From requirements and analysis to PDDL in itSIMPLE$_{3.0}$. In: Proceedings of the third international competition on knowledge engineering for planning and scheduling. 2009, p. 54–61.

[13] Francés G, Ramirez M. Collaborators, tarski: An AI planning modeling framework. 2018, https://github.com/aig-upf/tarski.

[14] Geffner H. Functional Strips: A more flexible language for planning and problem solving. In: Minker J, editor. Logic-based artificial intelligence. Kluwer international series in engineering and computer science, vol. 597, Kluwer; 2000, p. 187–209, [Ch. 9].

[15] Sanner S. Relational dynamic influence diagram language (RDDL): Language description. 2010.

[16] Pellier D, Fiorino H. Pddl4j: a planning domain description library for java. J Exp Theor Artif Intell 2018;30(1):143–76.

[17] Georgievski I. Planx: A toolbox for building and integrating ai planning systems. In: 2023 IEEE international conference on service-oriented system engineering. IEEE; 2023, p. 130–4.

[18] Dolejsi J, Long D, Fox M, Besançon G. PDDL authoring and validation environment for building end-to-en planning solutions. In: ICAPS system demonstrations and exhibits. 2018.

[19] Muise C. Planning.Domains. In: ICAPS system demonstrations and exhibits. 2016.

[20] Magnaguagno MC, Fraga Pereira R, Móre MD, Meneguzzi FR. Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In: 2017 workshop on user interfaces and scheduling and planning. USA; 2017.

[21] Muise C, Pommerening F, Seipp J, Katz M. Planutils: Bringing planning to the masses. In: ICAPS system demonstrations and exhibits. 2022.

[22] Miquel Ramirez CM. Nir Lipovetzky, Lapkt: Lightweight automated planning toolkit. 2024, https://github.com/LAPKT-dev.

[23] Barreiro J, Boyce M, Do M, Frank J, Iatauro M, Kichkaylo T, et al. Europa: A platform for ai planning, scheduling, constraint programming, and optimization. In: 4th international competition on knowledge engineering for planning and scheduling. 2012, p. 6–7.

[24] André F, Daubert E, Nain G, Morin B, Barais O. F4plan: An approach to build efficient adaptation plans. In: International conference on mobile and ubiquitous systems: computing, networking, and services. Springer; 2010, p. 386–92.

[25] McGreggor D, Nau D. Pyhop: a simple hierarchical task network (htn) planner written in python. 2024, https://github.com/oubiwann/pyhop.

[26] Gerevini AE, Percassi F, Scala E. An effective polynomial technique for compiling conditional effects away. In: Association for the advancement of artificial intelligence conference. 2024, p. 20104–12.

[27] Nebel B. On the compilability and expressive power of propositional planning formalisms. J Artificial Intelligence Res 2000;12:271–315.

[28] Bonassi L, Gerevini AE, Scala E. Dealing with numeric and metric time constraints in PDDL3 via compilation to numeric planning. In: Association for the advancement of artificial intelligence conference. 2024, p. 20036–43.

[29] Tosello E, Bonel P, Buranello A, Carraro M, Cimatti A, Granelli L, et al. Opportunistic (re)planning for long-term deep-ocean inspection: An autonomous underwater architecture. IEEE Robot Autom Mag 2024;31(1):72–83.

[30] Hastham Sathiya Satchi Sadanandam S, Stock S, Sung A, Ingrand F, Lima O, Vinci M, et al. A closed-loop framework-independent bridge from AIPlan4EU's unified planning platform to embedded systems. In: ICAPS 2023 workshop on planning and robotics. 2023.

[31] Taitler A, Alford R, Espasa J, Behnke G, Fišer D, Gimelfarb M, et al. The 2023 international planning competition. AI Mag 2023;45(2):280–96. http://dx.doi.org/10.1002/aaai.12169.