Making Unified Planning Dynamic by Incorporating World State Updates via Feedback Loops

Björn Döschl
University of the Bundeswehr Munich
bjoern.doeschl@unibw.de

Jane Jean Kiam
University of the Bundeswehr Munich
jane.kiam@unibw.de

Abstract

In real-world applications, most robot platforms operate in highly dynamic environments, where purely static planning approaches are insufficient. We present an architectural and functional extension to the UP framework providing a PDDL-based solution to reactive replanning. Our method constructs UP problem instances directly from updated world states, by integrating the latest world state as mission context into the planning process, thereby enabling a traditionally static planner to react during mission execution. An execution monitor checks the progress, provides feedback, and can initiate a replanning process if necessary, ensuring that mission objectives are met. We demonstrate our method by integrating it within the AUSPEX framework and connecting it to the REAP simulation environment. In a simulated UAV-based search and rescue mission, this setup effectively handles dynamic goal completion in two given scenarios by replanning.

Introduction

The demand for increasing the fleet size of unmanned autonomous vehicles in real-world missions is beneficial for higher mission efficiency but can result in substantial operational resources. For example, in state-of-the-art Search and Rescue (SAR) operations, or disaster response in general, multiple human operators are required per Unmanned Aerial Vehicle (UAV) (Bundesamt für Bevölkerungsschutz und Katastrophenhilfe 2024). Therefore, automated planning and acting is the key to facilitating fleet expansion without increasing human resource demands or cognitive workload on the human operators.

Yet fleet scale brings a second challenge: collecting and managing the vast amount of data generated in the system. A knowledge base must manage not only the real-time states of every cooperating asset, including UAVs, helicopters, ground teams, and K9 units, but also integrate and merge GIS layers, mission resources, and accumulated prior experience. All of this data must be taken into account in an automated planning process.

Existing automated planning systems typically rely on rigid, expert-designed, string-based problem definitions and manual command-line inputs. Their deployment for dynamic planning and plan execution is inconvenient, as the software workflow prevents a seamless integration with real-time information retrieval and updating, thereby reducing planning responsiveness in dynamic environments. The Unified Planning (UP) (Micheli et al. 2025) library significantly improved automated planning by unifying multiple planners via a structured Python API, simplifying problem formulation by providing a programmatic interface for Planning Domain Definition Language (PDDL) problem definitions, and wrapping the previous string-based approach in a less complex and more flexible planning interface.

However, dynamic missions continuously generate mission-relevant information during operation, which is essential not only for the monitoring of plan execution but also for defining up-to-date problem instances, should replanning take place. The UP library lacks a native mechanism to address dynamic missions. In this paper, we propose an architectural and functional extension to incorporate real-time world state updates, coupled with capabilities for real-time retrieval of execution feedback. The functional extension includes

- automated translation of real-time world states into UP planning problems, by leveraging the UP Python API for on-the-fly (re)planning;
- automated integration of execution feedback for monitoring and (re)planning.

Furthermore, we propose a software architecture that is suitable for coordinating multiple UAVs from a ground control station and demonstrate how the extended functions are used for planning and plan execution monitoring within a SAR operation.

We start by outlining the relevant related works, followed by presenting our planning framework in a high-level overview and then in detail while demonstrating the effectiveness in a small SAR UAV use case within the AUSPEX (Döschl, Sommer, and Kiam 2025) framework.

Related Works

Automated PDDL Generation

Conventional ways of modelling planning problems in PDDL have been an expert-driven manual process, i.e., hand-crafted planning problem models require detailed understanding of the problem and its logical correlations, as well as an aptitude with the PDDL-syntax. Introducing code-based PDDL generation was a first step to overcome the limitation of manual modelling problem domains.

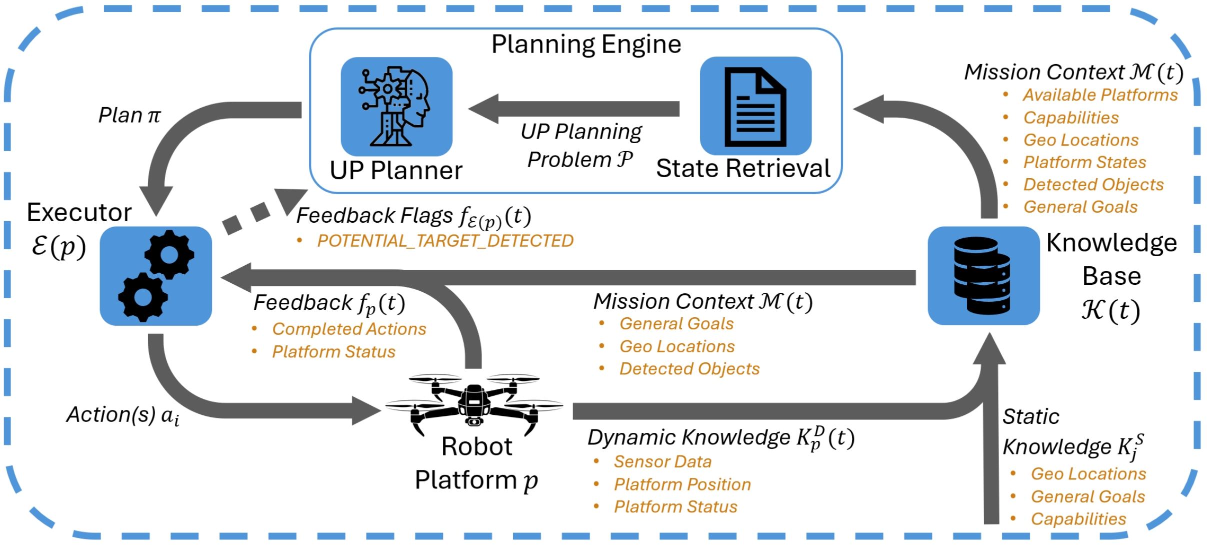

Figure 1: Software Architecture Diagram of the Dynamic UP Extension, illustrating the iterative process of knowledge solicitation, planning, plan execution and monitoring, and knowledge updates. Black italic labels denote abstract variables, while orange italic labels show concrete example values.

Several tools now enable PDDL generation through programmatic interfaces. Tarski (Francés and Ramirez 2018) and PY2PDDL (bin Karim 2020) both provide a Python-based library for defining planning problems declaratively, enabling the developer to automatically generate and export corresponding PDDL representations. Similarly, UP (Micheli et al. 2025) bypasses manual representation in PDDL by providing a programmatic interface in Python, thereby reducing the syntactic overhead of problem modelling. In UP, the generated PDDL files are handled internally, i.e., only created temporarily for planning using planners such as Fast Downward (Helmert 2006), Tamer (Valentini, Micheli, and Cimatti 2020), and automatically deleted afterwards.

Alternative Approaches towards PDDL-based Dynamic Planning

Open-world dynamic planning poses unique challenges to the problem instance acquisition due to underspecified, continuous, or partially observable state spaces. To overcome these difficulties, one approach is to enable agents to act upon hypothetical objects during execution (Jiang et al. 2019). Such an approach assumes a rather structured and limited hypothesis space, posing a challenge in more dynamic environments. To address these adaptability issues, Stern et al. (2022) allow agents to detect changes in the environment and update their internal models accordingly. The Continual Open-World Planning (COWP) combines symbolic planning with Large Language Models (LLMs) to model common sense knowledge and to handle unstructured environments (Ding et al. 2024), but inherits the need for validating an LLM's output (Kambhampati et al. 2024).

Extending the UP Framework for Dynamic Planning and Plan Execution

In this work, we introduce an architectural and functional extension to the UP framework for on-the-fly problem instance modelling in dynamic robot [1]. Rather than relying on a one-shot configuration, we treat planning as a dynamic closed loop that includes knowledge solicitation, planning, plan execution and monitoring, and knowledge updates. A detailed illustration is shown in Figure 1.

A robot $p \in P$, with P being the set of available robot platforms, is usually equipped with an array of sensors (e.g., Global Positioning System (GPS), Inertial Measurement Units (IMUs), and a camera) to capture and stream dynamic knowledge $K_p^D(t)$ about its environment and internal state at time step t (see Figure 1, right). Each knowledge snapshot $K_p^D(t)$ consists of a set of key-value pairs. A knowledge base $\mathcal{K}(t)$ *listens* to the incoming data streams and stores each snapshot $K_p^D(t)$, while keeping only the most up-to-date data (Sun, Zhang, and Chen 2019; Bernardo, Sousa, and Gonçalves 2025). To formalise this update step, we define the update policy as follows:

$$\hat{K}_p^D(t) = \left(K_p^D(t-1) \setminus K_p^D(t)\right) \cup K_p^D(t) \qquad (1)$$

In other words, we first remove from the previous snapshot every key-value pair that also occurs in the new knowledge snapshot and merge the result with the new snapshot. Together with the static knowledge K_j^S from static knowledge source $j \in J$, inserted into the knowledge base at startup, we define:

$$\mathcal{K}(t) = \{\hat{K}_p^D(t) \mid p \in P\} \cup \{K_j^S \mid j \in J\} \qquad (2)$$

[1]Our system can be found on GitHub at AUSPEX-PLAN as part of our AUSPEX framework.

as the set of knowledge at time step t, also called knowledge base or world state. Therefore, by requesting a set of mission-relevant knowledge snapshots, stored in the knowledge base at time step t, a mission context $\mathcal{M}(t)$, with $\mathcal{M}(t) \subseteq \mathcal{K}(t)$, is obtained (see Figure 1). In Algorithm 1 this process is illustrated.

Algorithm 1 GetRelevantMissionContext (t)

```
Given: Planning Domain $d$
Require: Knowledge base $\mathcal{K}(t)$
1: $\mathcal{M}(t) \leftarrow \varnothing$
2: for all $k \in \mathcal{K}(t)$ do
3:     for all key $\in k$ do
4:         if key $\in d$.types then
5:             $\mathcal{M}(t) \leftarrow \mathcal{M}(t) \cup k$
6:         end if
7:     end for
8: end for
9: return $\mathcal{M}(t)$
```

To construct the relevant mission context, the function `GetRelevantMissionContext` (t), iterates over each knowledge snapshot in $\mathcal{K}(t)$, selects those relevant to the given UP domain d, and inserts them into the current snapshot $\mathcal{M}(t)$. In practice, the mission context is encoded as a structured collection of JSON-formatted data, serving as input for downstream modules, i.e., planning and execution modules.

During planning, $\mathcal{M}(t)$ is used for the corresponding generation of UP-processable input in the `State Retrieval` module. For each plan request, this module calls the `GetRelevantMissionContext(t)` function to obtain the current mission context $\mathcal{M}(t)$ as JSON-formatted data. This data is translated into a UP planning problem object $\mathcal{P}$ by parsing each element in $\mathcal{M}(t)$ into an UP Python object and adding it to the problem. UP provides Python classes for PDDL definitions, including types, fluents, predicates, actions, and goals. Missing content of the problem instance, such as initial values, are set to the default value false if not given otherwise in $\mathcal{M}(t)$. The planning domain, consisting of fluents and several platform-specific actions, is mostly pre-defined. The resulting UP planning problem $\mathcal{P}$ is the input to our `UP Planner` and, based on this, computes a plan trajectory π (see Figure 1, Planning Engine).

An `Executor` $\mathcal{E}(p)$ associated with a `Robot Platform` p sequentially dispatches the actions a_i of plan π, where $\pi = \{a_1, \dots, a_i, \dots, a_N\}$ with $N = |\pi|$ and continuously monitors their execution (Figure 1, left). For effective monitoring, the executor extracts goals from the knowledge base $\mathcal{K}(t)$ belonging to the current plan π. By using the updated mission context $\mathcal{M}(t)$ and the platform execution feedback $f_p(t)$, the executor is able to check for discrepancies between the expected and actual action outcomes. If substantial deviations are detected, the executor $\mathcal{E}(p)$ can send feedback $f_{\mathcal{E}(p)}(t)$ in the form of flags back to the `Planning Engine` to, among other things, initiate a replanning process. These flags are set either due to execution failures, goal completions, or goal confirmation. If a flag is set, the executor cancels the current plan and requests a new plan.

When a *replan* flag is received, the `State Retrieval` module fetches the latest relevant data from the world state as updated mission context $\mathcal{M}(t)$, now containing any new environmental data, and forwards it to the planner, who computes a new refactored plan π.

Experiments and Preliminary Results

To validate our architectural and functional extension of UP, we integrated it into AUSPEX (Döschl, Sommer, and Kiam 2025), a modular, open-source, decision-making framework for UAVs using R0S 2 as the middleware. AUSPEX can be connected to real and simulated UAVs. For the simulation environment, the REAP (Kraus, Mair, and Kiam 2023) framework was deployed. REAP is configured in Unreal Engine 5.2 from Epic Games, with the AirSim (Shah et al. 2017) fork Colosseum from CodexLabsLLC for the UAV simulation. With Unreal Engine and its plug-ins, it is possible to simulate GPS-based UAV flights in real-world environments. For our system validation we deploy a simulated UAV, called "uav1". We define the mission goal as "`Search the fields and find a person with a red jacket`", which is encoded as PDDL goals in AUSPEX:

$$((\texttt{search}(\texttt{openarea1}) \land \texttt{search}(\texttt{openarea2})) \lor \texttt{found}(\texttt{person})) \land \texttt{landed}(\texttt{uav1}) \tag{3}$$

In the REAP simulation environment, we included the Cesium Plugin for Google Photorealistic Tileset to have an accurate real-world visual representation for the simulated UAV to fly in. The UAV's home position is set to an asphalt field. A person is placed as an Unreal Engine asset in one of two close meadows, and the application area is partitioned into symbolic areas loaded into the knowledge base $\mathcal{K}(t)$ as static knowledge K_1^S.

Scenario 1

Upon receiving a planning request, the `State Retrieval` module queries the knowledge base $\mathcal{K}(t)$ for relevant data, receiving $\mathcal{M}(t)$, containing available location objects `woods1`, `woods2`, `woods2`, `openarea1`, `openarea2`, and `waters1`, which symbolically partition the application area. $\mathcal{M}(t)$ also incorporates a list of available UAVs and possible detected objects. A detailed list is given in Figure 1 in the orange examples for the mission context. In a second step, the elements in $\mathcal{M}(t)$ are parsed into UP objects. For example, the `State Retrieval` converts the available geographic locations to PDDL objects:

$$\texttt{Object}(\texttt{areas}[\texttt{'name'}], \texttt{Location}), \tag{4}$$

and adds them as type `Location` to the current planning problem. Similarly, detected objects and available platforms are also added as objects to the problem. When generating the UP planning problem $\mathcal{P}$, fluents such as `found(person)` are initialised. Since no object of class `person` has been detected yet, `found(person)` is set to false. The problem domain entails a predefined set of actions, consisting of `takeoff(platform)`, `land(platform)`, `fly(platform, Location)`, `search(platform, Location)`, and `confirm(platform, object)`. Together with the predefined goals, given in Equation 3, $\mathcal{P}$ is forwarded to the `UP Planner`, and a plan π is requested, invoking the OneShotPlanner instance. Internally, UP utilises the Fast Downward planning engine (Helmert 2006) to compute a sequence of compound actions. The resulting plan is shown in detail below:

```
1. takeoff(uav1)
2. fly(uav1, openarea1)
3. search(uav1, openarea1)
4. fly(uav1, openarea2)
5. search(uav1, openarea2)
6. fly(uav1, home)
7. land(uav1)
```

A dedicated low-level path planner decomposes the compound `search` actions into a sequence of atomic `fly(platform, Location)` commands. The resulting trajectory is displayed in Figure 2.

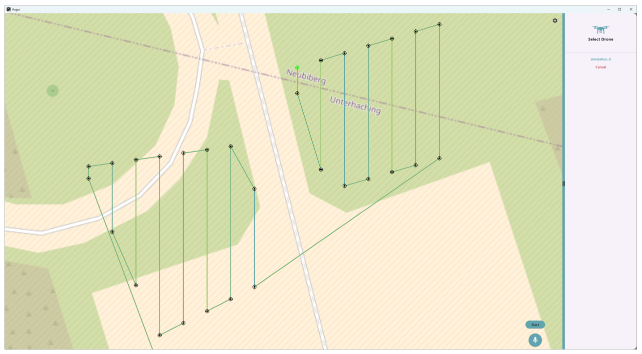

Figure 2: The planned trajectory of the UAV with two given search areas.

Figure 2 visualises the initial mission plan. Once the user accepts this plan, it is forwarded to the executor $\mathcal{E}(p)$, which sequentially dispatches single actions a_i to the associated UAV for execution.

The executor requests the relevant mission context from $\mathcal{M}(t)$ extracted from the knowledge base $\mathcal{K}(t)$ to obtain monitoring information, displayed in orange in Figure 1. Together with the execution information $f_p(t)$ from the UAV p, the executor is able to compute a nominal/actual value comparison. With the platform states, the executor can assess the coverage of the UAVs's trajectory and can compare it to the given search area. If at least 80 % coverage is achieved, the goal is marked as completed. In other cases, upon detecting an object of class person with a confidence score of at least 60 %, using YOLOv8 (Varghese and M. 2024) on the camera stream of the UAV, the executor sets a flag `POTENTIAL_TARGET_DETECTED`, cancels the current execution, and notifies the planner about this flag.

The `State Retrieval` module queries the latest world state for the mission context $\mathcal{M}(t)$, which includes the knowledge about the detected person, converts this context into an updated UP planning problem $\mathcal{P}$, and sends it to the `UP Planner`. If the confidence score of the detected person is below 80 %, the planner will compute a `confirm(uav1,person)` action. If the score is already above 80 %, the fluent `found(person)` will be set to true, and the respective goal will be set as completed. Consequently, the planner focuses solely on completing the remaining goal, resulting in a new plan π:

```
1. fly(uav1, home)
2. land(uav1)
```

Since the only remaining goal is `landed(uav1)`, UP computes a corresponding plan trajectory. After the UAV has successfully landed, this final goal is marked as completed. Consequently, all mission goals are completed, indicating a successful execution of the mission. Due to the logical "or" concatenation of the `search` and `found` goals, the mission is considered successful if either a person is found or all specified areas have been searched and the UAV has landed.

Scenario 2

To demonstrate responsiveness to exogenous, unplanned events, we simulate a second scenario. In typical SAR missions, manned helicopters share the airspace with UAV teams. In practice, to prevent collisions between the aircraft, a helicopter's operation area must be set as a no-fly zone for the UAVs. To continue with our example, when instead of a person detected, a helicopter pilot announces that it will search `openarea0`, this region is immediately flagged as a no-fly zone for the UAV, triggering replanning. This no-fly zone is inserted into the knowledge base $\mathcal{K}(t)$ and will be included in the next mission context snapshot $\mathcal{M}(t)$. During replanning the `State Retrieval` module treats the new no-fly zone as a hard constraint and ignores `openarea0` when creating the UP objects. The updated planning problem $\mathcal{P}$ is fed back to the `UP Planner`, which returns an updated plan π:

```
1. fly(uav1, openarea2)
2. search(uav1, openarea2)
3. fly(uav1, home)
4. land(uav1)
```

This new plan skips taking off (if the UAV is already airborne) and searching the location `openarea1`. Since the plan is only updated, the executor will consider previous search progress.

Conclusion & Future Work

In this work, we presented an functional and architectural extension to the Unified Planning framework for dynamic missions with on-the-fly problem instance modelling based on updating world states, integrated within AUSPEX. By extracting planning problems directly from the current world state as mission context, we demonstrated how planning problem instances can be instantiated and updated during execution. Our method allows for reaction to execution feedback, enabling mission replanning when new observations emerge, thereby overcoming the static nature of the traditional UP while leveraging the benefits of automated planning.

As future work, we aim to extend this system by incorporating a knowledge fusion, rather than just keeping the newest knowledge snapshots. Additionally, we plan to scale the planning problem and engine to support multi-UAVs and introduce a broader set of actions to enable planning in more complex mission scenarios.

Acknowledgments

This work is funded by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR) (Project MENTHON) and by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr which we gratefully acknowledge. dtec.bw is funded by the European Union – NextGenerationEU. The modelling of the validation scenario benefited from valuable input provided by domain experts of the Penzberg Mountain Rescuers (Bergwacht Penzberg).

References

[Bernardo, Sousa, and Gonçalves 2025] Bernardo, R.; Sousa, J. M.; and Gonçalves, P. J. 2025. Ontological Framework for High-Level Task Replanning for Autonomous Robotic Systems. *Robotics and Autonomous Systems* 184:104861.

[bin Karim 2020] bin Karim, R. 2020. PY2PDDL. `https://github.com/remykarem/py2pddl`.

[Bundesamt für Bevölkerungsschutz und Katastrophenhilfe 2024] Bundesamt für Bevölkerungsschutz und Katastrophenhilfe. 2024. *EGRED 2: Empfehlungen für Gemeinsame Regelungen zum Einsatz von Drohnen im Bevölkerungsschutz*.

[Ding et al. 2024] Ding, Y.; Zhang, X.; Amiri, S.; Cao, N.; Yang, H.; Esselink, C.; and Zhang, S. 2024. Robot Task Planning and Situation Handling in Open Worlds.

[Döschl, Sommer, and Kiam 2025] Döschl, B.; Sommer, K.; and Kiam, J. J. 2025. AUSPEX: An Integrated Open-Source Decision-Making Framework for UAVs in Rescue Missions.

[Francés and Ramirez 2018] Francés, G., and Ramirez, M. 2018. Tarski: An AI Planning Modeling Framework. `https://github.com/aig-upf/tarski`.

[Helmert 2006] Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

[Jiang et al. 2019] Jiang, Y.; Walker, N.; Hart, J.; and Stone, P. 2019. Open-World Reasoning for Service Robots. *Proceedings of the International Conference on Automated Planning and Scheduling* 29(1):725–733.

[Kambhampati et al. 2024] Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and Murthy, A. B. 2024. LLMs Can't Plan, but Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*.

[Kraus, Mair, and Kiam 2023] Kraus, O.; Mair, L.; and Kiam, J. 2023. REAP: A Flexible Real-World Simulation Framework for AI-Planning of UAVs. In *ICAPS 2023 System Demonstrations*.

[Micheli et al. 2025] Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, Manipulating and Solving AI Planning Problems in Python. *SoftwareX* 29:102012.

[Shah et al. 2017] Shah, S.; Dey, D.; Lovett, C.; and Kapoor, A. 2017. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Field and Service Robotics*.

[Stern, Piotrowski, and Klenk 2022] Stern, R.; Piotrowski, W.; and Klenk, M. 2022. Model-Based Adaptation to Novelty for Open-World AI. In *Proceedings of the Workshop on Planning and Robotics (PRL) at ICAPS*.

[Sun, Zhang, and Chen 2019] Sun, X.; Zhang, Y.; and Chen, J. 2019. RTPO: A Domain Knowledge Base for Robot Task Planning. *Electronics* 8(10).

[Valentini, Micheli, and Cimatti 2020] Valentini, A.; Micheli, A.; and Cimatti, A. 2020. Temporal Planning with Intermediate Conditions and Effects. *Proceedings of the AAAI Conference on Artificial Intelligence* 34(06):9975–9982.

[Varghese and M. 2024] Varghese, R., and M., S. 2024. YOLOv8: A Novel Object Detection Algorithm with Enhanced Performance and Robustness. In *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, 1–6.