UPF First Prototype

31st December 2021

Project funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	X
CO	Confidential, only for members of the consortium (including the Commission Services)	□
CL	Classified, as referred to in Commission decision 2001/844/EC	□

Deliverable number:	**D3.2**
Deliverable name:	UPF First Prototype
Work package:	WP3: Unified Planning Framework
Lead WP:	Andrea Micheli
Lead Task:	Andrea Micheli

Contents

Document Revision History

Date	Issue	Author/Editor/Contributor	Summary of main change
10/12/2021	V1	Andrea Micheli (FBK)	First Draft
23/12/2021	V2	All WP3 Partners	Feedback and edits
29/12/2021	V3	Andrea Micheli (FBK)	Final version

Abstract

Automated Planning and Scheduling is a central research area in AI that has been studied since the inception of the field and where European research has been making strong contributions over decades. Planning is a decision making technology that consists in reasoning on a predictive model of a system being controlled and deciding how and when to act in order to achieve a desired objective. It is a relevant technology for many application areas that need quick, automated and optimal decisions, like agile manufacturing, agrifood or logistics. Although there is a wealth of techniques that are mature in terms of science and systems, several obstacles hinder their adoption, thus preventing them from making the footprint on European industry that they should make. For example, it is hard for practitioners to find the right techniques for a given planning problem, there are no shared standards to use them, and there is no easy access to expertise on how to encode domain knowledge into a planner.

The AIPlan4EU project will bring AI planning as a first-class citizen in the European AI On-Demand (AI4EU) Platform by developing a uniform, user-centered framework to access the existing planning technology and by devising concrete guidelines for innovators and practitioners on how to use this technology. To do so, we will consider use-cases from diverse application areas that will drive the design and the development of the framework, and include several available planning systems as engines that can be selected to solve practical problems. We will develop a general and planner-agnostic API that will both be served by the AI4EU platform and be available as a resource to be integrated into the users' systems. The framework will be validated on use-cases both from within the consortium and recruited by means of cascade funding; moreover, standard interfaces between the framework and common industrial technologies will be developed and made available.

Executive summary

This deliverable summarizes the capabilities, characteristics and software architecture of the first iteration of the Unified Planning Framework (UPF) implementation. The development is publicly visible on the project's GitHub repository[1] and this deliverable refers to the software state in December 2021[2]. A snapshot of the code is also available as a zip file attached to this deliverable.

We will describe the abstract capabilities of the library, provide example usages and point the reader to running examples. Moreover, we will discuss the current limitations and the next steps that we will pursue in the development.

Finally, for an overview of the development process and the design choices we refer the reader to D3.1: Infrastructure Description.

[1] https://github.com/aiplan4eu/upf

[2] The repository has been tagged 'D3.2' to keep track of the exact status theis deliverable refers to.

Introduction

This report describes the Unified Planning Framework python library that the AIPlan4EU project is developing in the context of WP3: Unified Planning Framework.

The purpose of the library is to provide an abstraction layer for planning technology allowing a user to specify planning problems in a planner independent way and then use one of the available planning engines installed on the system. The library is implemented as a Python package offering high level API to specify planning problems and to invoke planning engineers. Moreover, the library offers functionalities for transforming and simplifying planning problems and to parse problems from existing formal languages.

The library is being developed publicly under a permissive open-source license (Apache 2.0) and the progress and the code can be followed at https://github.com/aiplan4eu/upf.

This report concerns the status of the library as of December 2021, after one year of project development. We will summarise the main functionalities and present the relevant code snippet showing the usage of the library, and we will point the reader to relevant material within the delivered code. Code-level documentation is being written and is automatically generated and accessible at https://upf.readthedocs.io/en/latest/ .

Functionalities and code structure

In this section, we will present the main functionalities offered by the library in its current status.

Problem specification

The main functionality offered by the library concerns the specification of a planning problem. The API provides classes and functions to populate a `Problem` object with the fluents, actions, initial states and goal specifications constituting the planning problem specification. In order to showcase the basic modelling capabilities of the library, we created an interactive Python notebook using the Google Colab service, which guides the reader in the details of the problem specification API. This documentation is publicly available[3].
The functionalities for creating model objects and to manipulate them are collected in the `upf.model` package of the library.

Example

The following example shows a simple robotic planning problem modeling a robot moving between locations while consuming battery. The example shows the basic functionalities and objects needed to declare the problem specification. A more detailed presentation of the different objects is available on the Google Colab Python notebook where we document and explain all the different classes and their semantics.

```
# Declaring types
Location = UserType('Location')
# Creating problem 'variables'
robot_at = Fluent('robot_at', BoolType(), [Location])
battery_charge = Fluent('battery_charge', RealType(0, 100))
# Creating actions
move = InstantaneousAction('move', l_from=Location, l_to=Location)
l_from = move.parameter('l_from')
l_to = move.parameter('l_to')
move.add_precondition(GE(battery_charge, 10))
move.add_precondition(Not(Equals(l_from, l_to)))
move.add_precondition(robot_at(l_from))
move.add_precondition(Not(robot_at(l_to)))
```

[3] https://colab.research.google.com/drive/1kbNu3k1SxO1CbTtqfLEUTmU1AuAyxuHG?usp=sharing

```
    move.add_effect(robot_at(l_from), False)
    move.add_effect(robot_at(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_fluent(robot_at)
    problem.add_fluent(battery_charge)
    problem.add_action(move)
    problem.add_object(l1)
    problem.add_object(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))
```

In the current version, the UPF library allows the specification of classical, numerical and temporal planning problems. In order to support the latitude expressiveness levels we have operators for arithmetic such as plus minus times and division and specific temporal operators to attach conditions and effects to specific timings within the duration of an action. the library documentation provides examples and describes the use of these functionalities.

Transformers

Another very interesting functionality offered by the UPF concerns model-to-model transformation. The library implements several simplifications and compilations that can transform one problem into an equivalent one getting rid of some of the planning constructs. For example, we offer a functionality to remove conditional effects from the planning problem specification by transforming the input problem into an equivalent one that does not make use of conditional effects. Our transforming architecture is very general and offers functionalities to transform a plan for the target problem of the compilation into a plan for the input problem. This allows the creation of pipelines of transformations that can map an input planning problem into an equivalent one supported by a target planning engine and then transform back the plan generated by the engine into a valid plan for the overall input problem.

All the available transformers are part of a class hierarchy rooted in the `Transformer` class and are contained in the `upf.transformers` package.

Example

The following example shows how to create a transformer to compile away negative conditions from a problem and to retrieve the plan for the original problem from a plan of the transformed problem. If the planner does not support negative conditions, the original problem could not be solved, while the transformer allows us to solve the problem anyway.

```
    # Creating the negative conditions remover
    neg_remover = NegativeConditionsRemover(problem)
    # Checking that the problem has negative conditions
    assert problem.kind().has_negative_conditions()
    # Asking the transformer to get the new problem
    new_problem = neg_remover.get_rewritten_problem()
    # Checking that the new problem does not have negative conditions
    assert not new_problem.kind().has_negative_conditions()
    #Solving the problem generated by the transformer
    new_plan = planner.solve(new_problem)
    #Getting the equivalent plan for the original problem
    plan = neg_remover.rewrite_back_plan(new_plan)
    #Checking that the generated plan is valid for the original problem
    assert planner.validate(problem, plan)
```

Solving Interface

The library offers primitives to invoke planning engines of different kinds on problem specifications. In particular, the library uses the concept of operation modes to account for different possible interactions that can be performed with the planning engine at hand. Such operation modes allow the standardization of APIs towards different planning engineers sharing the same interaction kind and primitives.

We currently support 3 operation modes:

- OneshotPlanning: is the classical interaction mode for the planning community, it consists in posing the planning problem entirely and then waiting for the solution. This operation mode does not support incremental reuse of information and is limited to one planning problem at a time, but is the most common operation model among the different planners available.
- PlanValidation: is an operation mode supporting the use case of checking the validity of a given plan against the problem specification. Essentially, the engine is required to analyse the given plan and report whether it is guaranteed to achieve the goal conditions or if instead it can fail due to an action not being applicable or a goal not being reached. For this operation mode, we also implemented a native engine that is part of the library itself.
- Grounding: Is an operation mode that transforms a given problem into an equivalent one that doesn't make use of action parameters or first order predicates. This is a very common operation to be done for solving a planning problem and it is needed to transform planning problems into state machines. Also in this case, the library offers a native grounder and it also integrates grounders of different engines so that more powerful grounding algorithms can be accessed in an uniform and engine-independent way.

The solving interface also features a powerful automatic filtering of planning engines. In fact, the input planning problem is automatically analysed in order to determine the features needed to tackle the problem itself. The planning engines available on the system where the library is executed are then filtered, and only the ones that are capable of tackling the problem are left for the user to select from. This mechanism simplifies the job of the user in the selection of the right planning engine to be used.

All the functionalities of the solving interface are collected under the `upf.solvers` package.

Example

The following example shows how to get a planner and solve a problem.

```
    # Getting a oneshot planner that is able to handle the given problem kind
    with OneshotPlanner(problem_kind=problem.kind()) as planner:
        # Asking the planner to solve the problem
        plan = planner.solve(problem)
        # Printing the plan
        print(plan)
```

Input/Output and Interoperability

Finally the UPF library offers primitives and functions for the interoperability with external formal languages and libraries. In particular, we offer a strong integration with the Planning Domain Definition Language (PDDL) language: we implemented a parser that can read in a problem specified in PDDL and convert it into a UPF problem data structure, and we have a comprehensive emitter that yeldsPDDL specifications from a UPF problem instance.

We also have automatic interfacing with other planning libraries. In particular, we have a conversion from a `tarski`[4] representation into a UPF problem allowing a user to import from this external data structure and simplify the interoperability between the two libraries.
The input-output classes and functions can be found in the `upf.io` package, while the interoperability with tarski (and in the future with other libraries) are in the `ups.interop` package.

Example

The following example shows how to read a PDDL problem from files and how to dump to files in PDDL format a UPF problem.

```
# Creating a PDDL reader
reader = PDDLReader()
# Parsing a PDDL problem from file
problem = reader.parse_problem('domain.pddl', 'problem.pddl')
# Creating a PDDL writer
writer = PDDLWriter()
# Writing the PDDL domain and problem in new files
writer.write_domain('new_domain.pddl')
writer.write_problem('new_problem.pddl')
```

Conclusion

In this report we provided an overview of the structure and the capabilities of the unified planning framework developed within the Work Package 3 of the AIPlan4EU project. The library is a Python package that is subdivided in modules and offers different functionalities for using planning technology at different levels of abstraction in a planner independent way. The code is being developed openly on a public repository and following a user centred approach and an agile development methodology. The body of the deliverable is constituted by the code itself, while this document provides an access point to understand the code and functionalities that have been developed. In the next months of work package 3 we will continue the development by adding new functionalities, refactoring the code and supporting the integration of both additional planning engines as well as new Technology-Specific Bridges.

[4] https://github.com/aig-upf/tarski