D4.2: Repository of the Test Cases

December 30th 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:	D4.2
Deliverable name:	Repository of the Test Cases
Work package:	WP4: Planning Engines
Lead WP:	UNIBS
Lead Task:	UNIBS

Contents

Document Revision History

Date	Issue	Author/Editor/Contributor	Summary of main change
13/12/2021	V1	UNIBS	Initial Draft
20/12/2021	V2	Andrea Micheli (FBK)	Revised intro and general context
28/12/2021	V3	UNIBS	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

In the AIPlan4EU project we are developing an abstraction layer, called Unified Planning Framework (UPF), which abstracts away the specificities of planning engines that are being integrated in the context of Work Package 4. Nonetheless, in order to ease the integration of planning engines and to guarantee a good quality of the connection code being developed, we need a flexible and comprehensive test suite that stresses several correctness aspects of the integrations.

We created a repository with the associated infrastructure that will contain several tests for evaluating the correctness of all planning approaches studied in Work Package 4. The body of deliverable 4.2 is the repository itself, which is available online publicly and under a permissive open-source license (a zip file snapshot is attached to this document). In this report, we describe the infrastructure we set up and the structure of the repository that will be populated with interesting semantic cases and examples throughout the development of WP4.

Introduction

The main purpose of the repository of planning test cases is to collect a comprehensive set of planning problems and specifications that can be easily tested on each engine being integrated with the Unified Planning Framework. The repository is structured as a collection of Python scripts that can be invoked automatically and can be used on different planning engines.
Within the scope of the work package 4, we created the structure of the repository reflecting the different tasks that constitute the work package, and we separated the problem specification from the testing code that is used to assess the correctness of the test.
We adopted a standard testing suite framework as basis for our repository, and we use the functionalities offered by the UPF as commodities to make it easy to contribute new tests and to run existing tests on newly developed planning engines. The use cases that we plan to add into this repository are focused on stimulating specific characteristics of different planning problems: for example, it will be possible to test the capability of planners to support a certain feature such as conditional effects, or to test specific configurations (e.g., stressing the capability of a planner to work in presence of big number of actions or to remove unnecessary variables from the problems).
The repository is concretely available on GitHub,[1] and it features and automatic continuous integration and continuous deployment allowing to assess the correctness of tests. Finally, we explicitly support versioning of the different tests in order to allow the evolution of tests and their life cycle ensuring a general consistency throughout the repository.
All the code problems and test cases within this repository is released under the Apache 2.0 license, which is a permissive open source license allowing the reuse of the material on the repository freely for partners of the consortium as well as third parties.

Structure and use of the repository

The repository structured in a way that makes it easy to add new tests, and to find existing tests for a certain class of planning problems. In particular, we followed the structure of the tasks of work package 4 creating one folder for each kind of planning problem considered within the project. Specifically, we have:

- `classical_planning`, which contains problems and testing code concerning planning over finite state representations without continuous time.
- `temporal_planning`, which contains problems and testing code for planning with continuous time and durative actions.
- `numeric_planning`, which contains problems and testing code for planning with infinite state representation involving numbers and arithmetic operations.
- `multiagent_planning`, which contains problems and testing code for planning with more than one agent, possibly with privacy constraints and imperfect information.
- `refinement_planning`, which contains problems and testing code for planning with hierarchies.
- `combined_task_motion_planning`, which contains problems and testing code for planning problems involving a combination between high-level decisions concerning what to do and how to realize the motions in continuous or discrete spaces.

For technical reasons, the repository includes root folder called `planning_tests` that serves as containing package for the entire repository. In fact, the whole repository is a Python package that can be imported from outside of the

[1] https://github.com/aiplan4eu/Planning-Test-Cases

repository, making it possible to reuse the tests from client applications or derived code. All the code supporting the functionalities of the repository is contained in a folder named `commons`.

In order to separate the problem definition from the code responsible for checking that a certain functionality is correctly implemented by a planning engine, each planning-kind folder (e.g., `classical_planning`) has two subfolders, called `problems` and `operation_modes`. The former contains the problem formulations, each one stressing one aspect of the planning approach. The problem can be formulated either by an ad-hoc language for the description of planning problems, like PDDL, or by directly using the Unified Planning Framework API developed in WP3.

The following python code gives an example of a test for classical planning.[2] In the initial state of the problem, proposition `y` is true while proposition `x` is false. The goal of the problem requires that `x` is true. The problem features only one action called `a`. The precondition of `a` is proposition `y`, while the positive and negative effects of a are respectively `x` and `y`. Therefore, the execution of the action makes `x` true and `y` false. This minimal example shows how to create a problem using the UPF API; nonetheless, it is possible to create a problem using any of the facilities offered by the UPF: for example, we can invoke a parser and create a problem from a formal language such as PDDL, or use the interfacing towards external libraries such as `tarski`[3].

```
import upf
from upf.shortcuts import *

from planning_tests.commons.problem import TestCaseProblem

class UPFBasic(TestCaseProblem):

    def __init__(self, expected_version):
        TestCaseProblem.__init__(self, expected_version)

    def get_problem(self):
        x = Fluent('x')
        y = Fluent('y')
        a = InstantaneousAction('a')
        a.add_precondition(y)
        a.add_effect(x, True)
        a.add_effect(y, False)
        problem = Problem('basic')
        problem.add_fluent(x)
        problem.add_action(a)
        problem.set_initial_value(x, False)
        problem.set_initial_value(y, True)
        problem.add_goal(x)
        return problem

    def get_description(self):
        return 'Just a basic test'

          def version(self):
        return 1
```

The folder `operation_modes` contains different folders, one for each operation mode (e.g. `oneshot`, see deliverable D3.2 for more details) for which a test has been developed. All the tests in these folders use the problem

[2] The code is available in `planning_tests/classical_planning/problems/problem_basic.py`
[3] https://github.com/aig-upf/tarski

formulations contained in problems to verify the correct behavior of the integrated AI planners when they are launched to find a solution of the given problem without considering its quality (it can be any, optimal or suboptimal, solution). An example of python code launching planner `pyperplan` for the planning problem defined above is the following.[4]

```
import upf
from upf.shortcuts import *

from unittest import TestCase, main
from planning_tests.classical_planning.problems.problem_basic import UPFBasic

class Check1(TestCase):
    def setUp(self):
        TestCase.setUp(self)
        self.problem = UPFBasic(expected_version=1)

    def test_basic(self):
        upf_problem = self.problem.get_problem()

        planner_names = [n for n, s in get_env().factory.solvers.items() if s.is_oneshot_planner()]

        for p in planner_names:
            with OneshotPlanner(name=p) as planner:
                if planner.supports(upf_problem.kind()):
                    plan = planner.solve(upf_problem)
                    with PlanValidator(problem_kind=upf_problem.kind()) as validator:
                        check = validator.validate(upf_problem, plan)
                        self.assertTrue(check)


if __name__ == "__main__":
    main()
```

Specifically, the code runs all the oneshot planners available (e.g. `pyperplan`), and a validator in sequence. The goal of the validator is to check if the solution produced by each planner is valid.

The expected_version variable of the test problem ("`UPFBasic(expected_version=1)`") is compared with the output of the version function of the corresponding problem definition, and if the values do not match, an exception is raised. This check is needed in order to be sure that revisions in the problem definition correspond to revisions in the corresponding test file.

Finally, a general launching script is provided in order to execute all the tests present in the repository with all the planners available on the system as discovered by the UPF. The script called `runner.py` is available in the top-level folder of the repository, and it is intended as a commodity script that a user can launch. Alternatively, all the tests are in fact valid `pytest`[5] classes, and hence they can be manually executed with the functionality offered by the `pytest` framework.

[4] The code of the example is available in `planning_tests/classical_planning/operation_modes/oneshot/test_basic.py`

[5] https://pytest.org

Conclusion

This document provides an overview of the code being delivered as D4.2. The repository constituting the deliverable itself is publicly available and inspectable at the following address: https://github.com/aiplan4eu/Planning-Test-Cases. In the first two months of WP4, leading to this deliverable, we worked for the creation of this repository by designing its structure, and creating a viable proof of concept that will support the future development of the work package.

In the next months, while the planning engines will be integrated in the context of WP4, we will populate the repository and we will extend the testing functionality as needed, in order to support the integration efforts and to ensure the quality of the code being developed.