Planning with Semantic Attachments: An Object-Oriented View

Andreas Hertle
Christian Dornhege
Thomas Keller
Bernhard Nebel

SAPIENZA
UNIVERSITÀ DI ROMA

Elective in Artificial Intelligence

Reasoning Agents

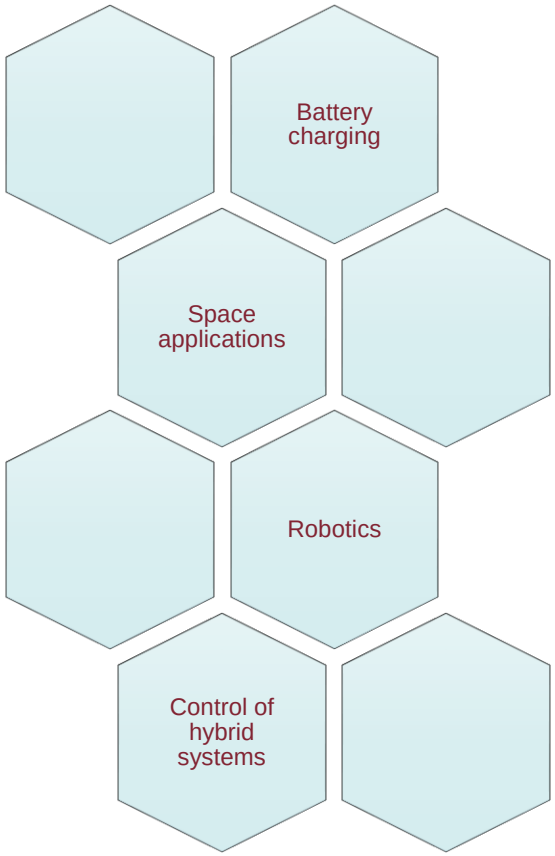
Prof. Fabio Patrizi

Olga Sorokoletova - 1937430

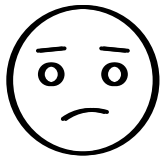
Introduction

Object-oriented Planning Language

Introduction: problem



- **Domain-independen**t planning has been applied to a rising number of real-world applications
- The most common description language for planning tasks is the Planning Domain Definition Language (**PDDL**)
  - suited to describe planning problems on an **abstract symbolic level**
- **Not sufficient** to model subproblems beyond the scope of symbolic planners
  - geometric computations
    - object manipulation
    - navigation
    - …
  - …



problem

Introduction: how to solve

Decompose!

Hierarchical composition, top-down: specialized planners refine the high-level symbolic plan

- Successful execution is not guaranteed

Hierarchical composition, bottom-up: information for the symbolic planner is precomputed by the lower level reasoners

- Costly

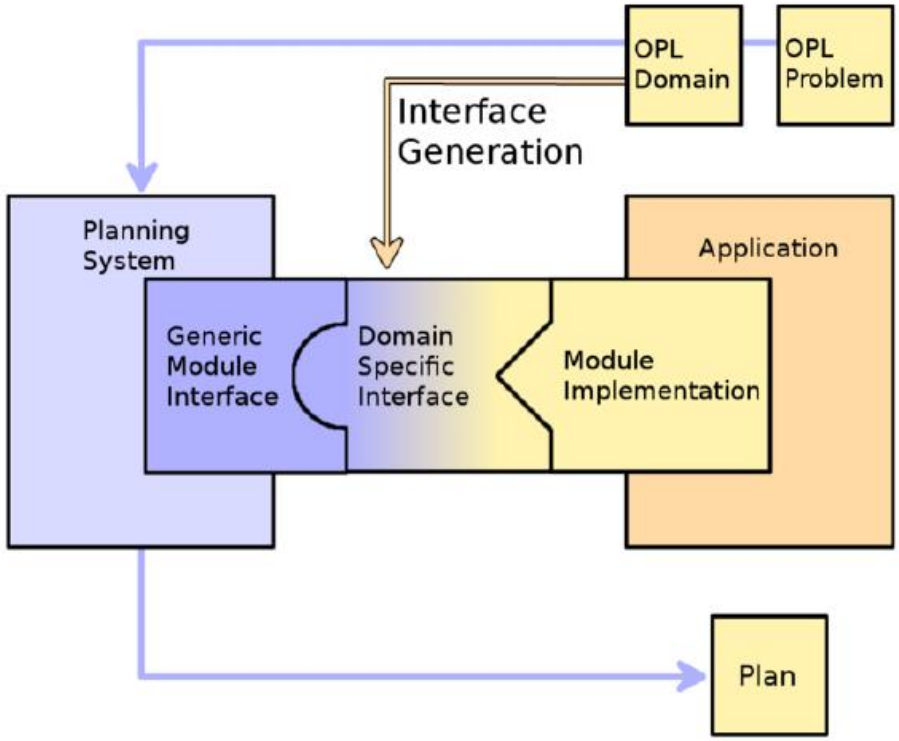
Semantic Attachments

- ?

- Semantic attachments compute the semantics of Boolean fluents by an **external process** during planning
- They are realized as **modules**
- **PDDL/M** is the slightly modified version of PDDL that allows to attach such a module to a Boolean fluent
- **TFD/M** is a version of the Temporal Fast Downward (TFD) planning system which implements an interface that calls a module
- In the implementation of a module it is usually **necessary to access the current planning state**
- To acquire state information from the planning system a module developer must perform manual **requests through a callback** interface
  - **inefficient!**
  - **error-prone!**

Introduction: how to improve

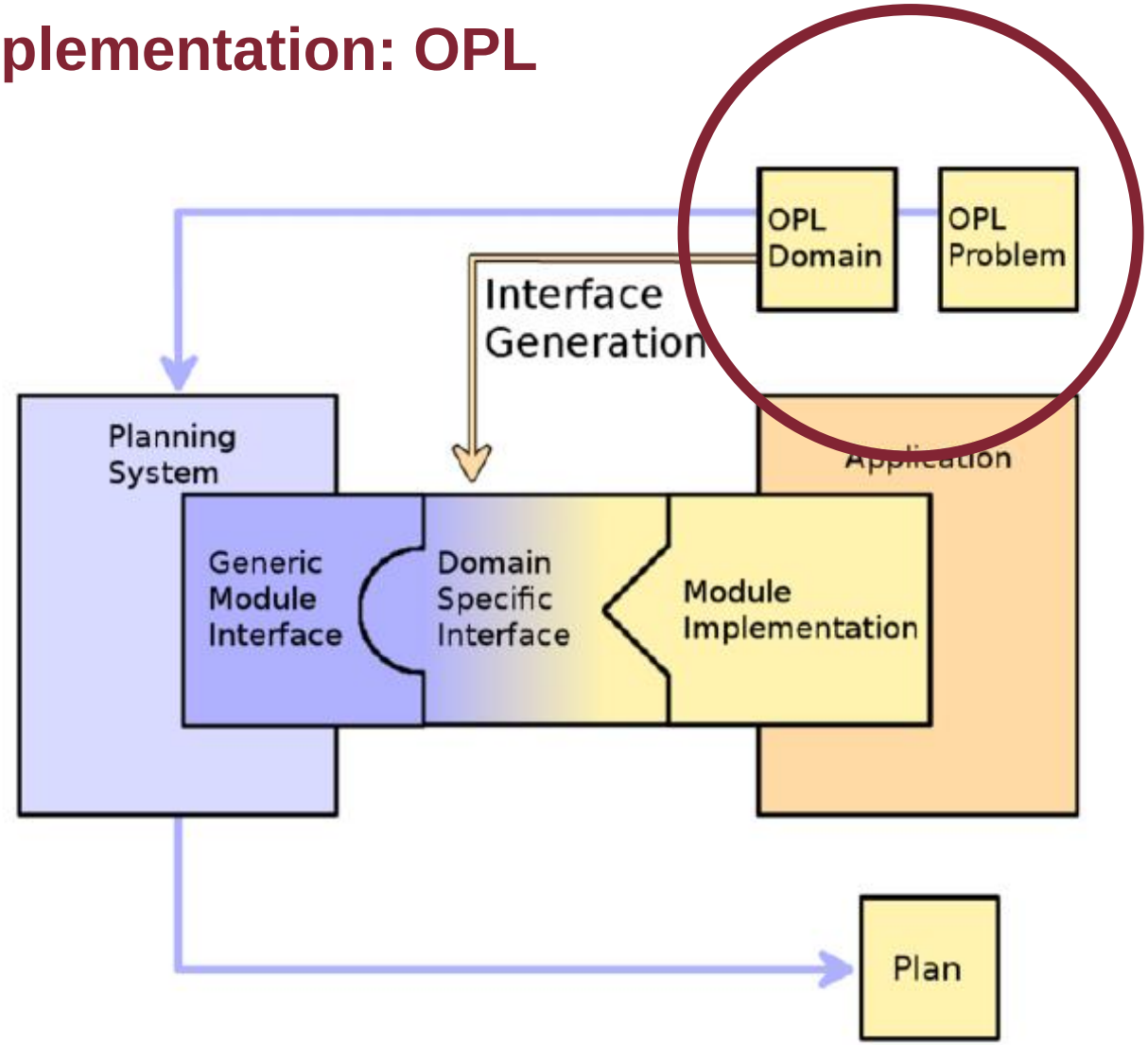
Object-oriented Planning Language (OPL)!
Incorporates structure and advantages of modern object-oriented programming languages (Java, C++): class definitions for module developers, and **objects** as instances of these class definitions reflect the current internal planning state - type-safe and **efficient access to the internal state of the planning system**.



Implementation

Integration of the Components

Implementation: OPL



Implementation: OPL

OPL task definitions are separated into a **problem** and **a domain** as in PDDL

```
Problem Scenario1(RoomScanning) {
  Pose p1;
  Pose p2 { x = 5; y = 1; th = 0.5; }
  Target t1 { x = 0; y = -13; th = 0.5; }
  Robot r1 { currentPose = p1; }
  Goal { and(equals(r1.currentPose, p2),
      t1.explored); } }
```

```
Domain RoomScanning {
  Type Pose {number x; number y; number th;}
  Type ScanTarget : Pose {boolean scanned;}
  Type Door {Pose approachPose;
             boolean open;}

  ConditionModule pathExists(
      Pose from, Pose to);

  Type Robot {
   Pose currentPose;
   CostModule driveCost(Pose dest);
   Action drive(Pose dest) {
    Cost {driveCost(dest);}
    Condition {
     and(not(equals(this.currentPose, dest)),
         pathExists(this.currentPose, dest));}
    Effect {assign(this.currentPose, dest);}}
```

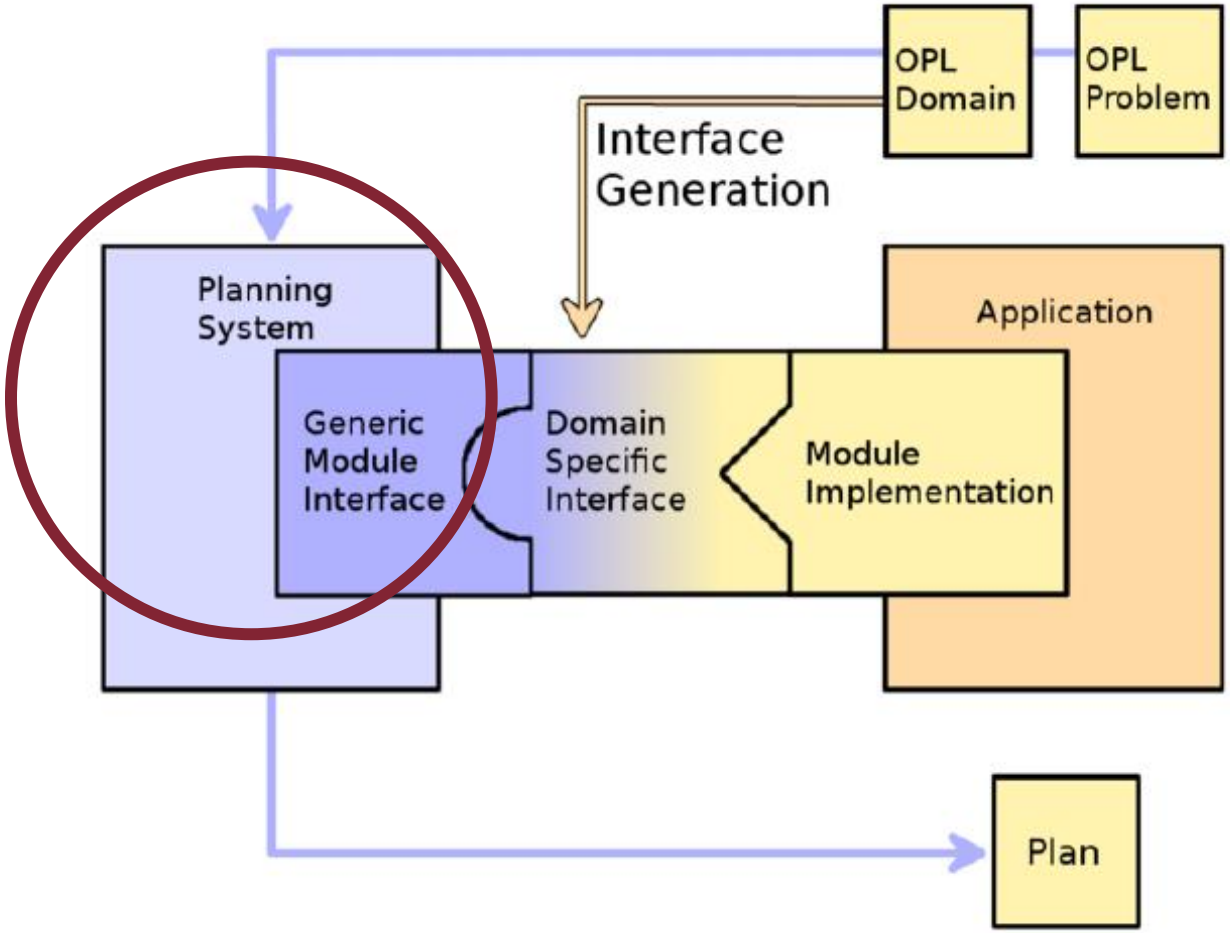
Like PDDL, types can inherit from other types, but will also inherit the **super types'** member fluents and actions. If no base type is specified, it defaults to `Object`, a built-in type with no members.

Greater importance of **types**: class declarations

Like PDDL actions, OPL **actions** have a name and parameters and define a (pre-)condition and an effect

The dot `.` acts as the structure access operator
The keyword `this` is used to address the current object in member actions

Implementation: Translation to PDDL



Implementation: OPL to PDDL

```
Domain RoomScanning {
  Type Pose {number x; number y; number th;}
  Type ScanTarget : Pose {boolean scanned;}
  Type Door {Pose approachPose;
             boolean open;}
```

Member fluents and actions are specific to an object instance. This is resolved by adding an additional parameter named `?this` as the first parameter.

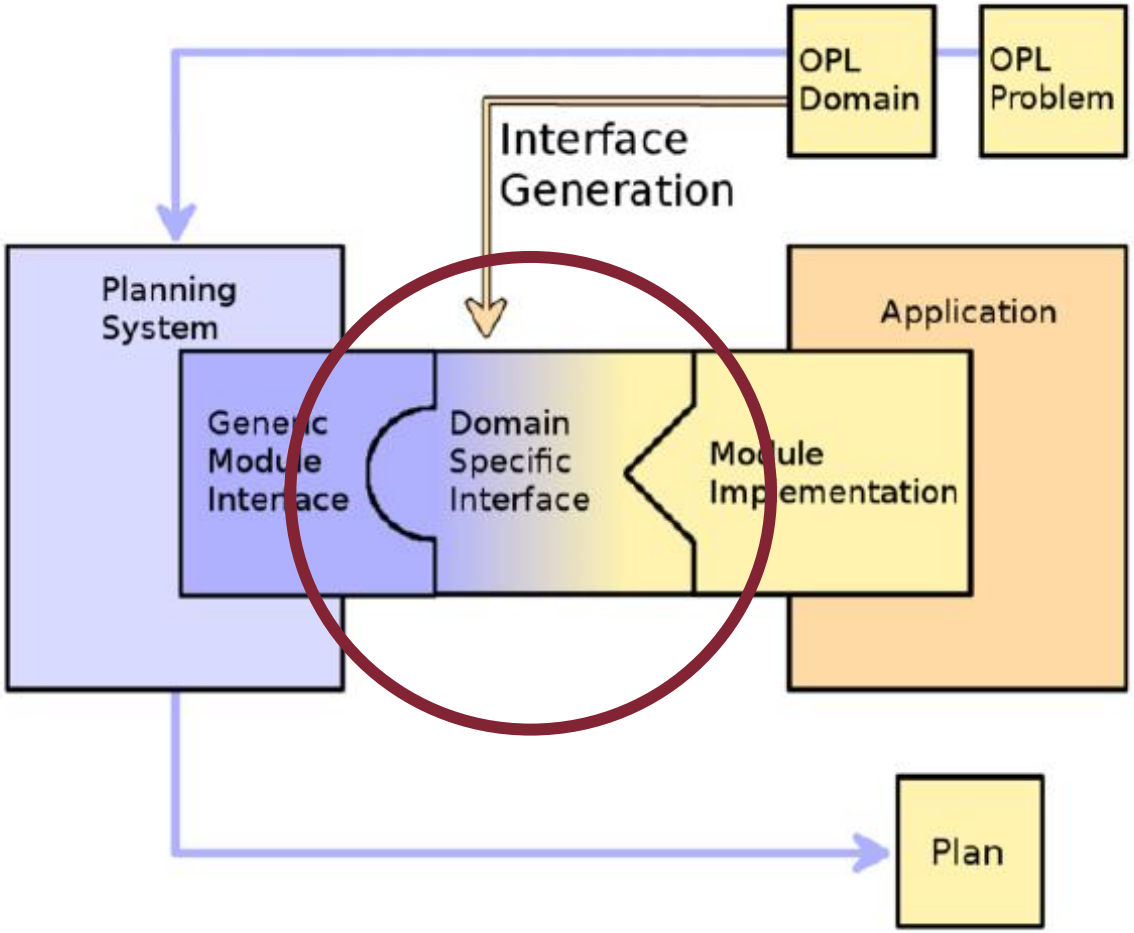
```
(:types
  Door Pose - object
  ScanTarget - Pose)

(:predicates
  (ScanTarget_scanned ?this - ScanTarget)
  (Door_open ?this - Door) ...)
(:functions
  (Pose_x ?this - Pose) - number
  (Pose_y ?this - Pose) - number
  (Pose_th ?this - Pose) - number
  (Door_approachPose ?this - Door) - Pose
  ...)
```

```
boolean inRange(Pose p1, Pose p2);
Type Robot {
 Action openDoor(Door door) {
 Condition {and(
  inRange(this.currentPose,door.approachPose),
  not(door.open));}
 Effect {door.open; } } }
```

As we allow to chain expressions over member fluents using the `.`-operator to access structure elements, we need to translate `o.m` to `m ?o <parameters>.`

```
(:predicates
  (inRange  ?p1 - Pose  ?p2 - Pose))
(:action Robot_openDoor
  :parameters
    (?this - Robot ?door - Door)
 :condition (and
   (inRange (Robot_currentPose ?this)
            (Door_approachPose ?door))
   (not (Door_open ?door)))
 :effect (Door_open ?door)
```

Implementation: Interface Generation

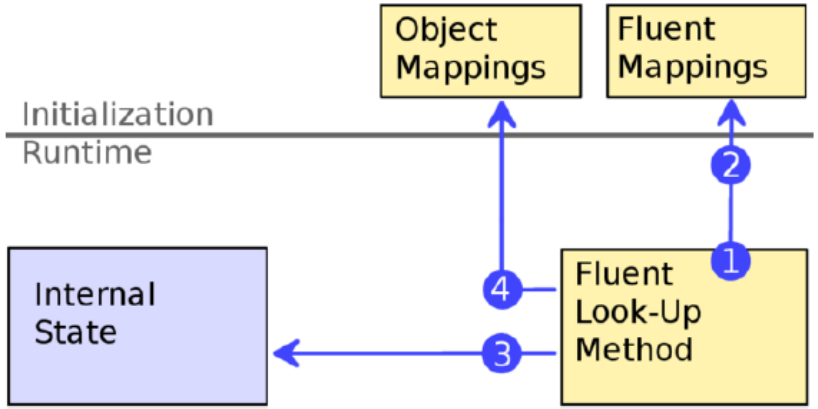


Implementation: interface generation

OPL objects are represented as classes. For TFD/M, this is realized in C++. Module calls receive object instances as parameters.

```
ConditionModule pathExists(
    Pose from, Pose to);

Type Robot {
 Pose currentPose;
 CostModule driveCost(Pose dest);
 Action drive(Pose dest) {
  Cost {driveCost(dest);}
  Condition {
   and(not(equals(this.currentPose, dest)),
       pathExists(this.currentPose, dest));}
  Effect {assign(this.currentPose, dest);}}
```

```
double Robot_driveCost(
  const State* currentState,
  const Robot* thisRobot,
  const Pose* dest);
```



The generated classes provide member functions for each member fluent to access the planner state**. Boolean** fluents lead to bool return values and **numerical** fluents return float values**. Object** fluents return a pointer to the class

Then to acquire a fluent's value from the planner's internal state **mapping tables** are created. For Boolean and numerical fluents they are used to acquire values directly, for object fluents - by the pointer.

Experiments

TFD/M (with vs without OPL)

Experiments: International Planning Competition (IPC) 2008

CREW-PLANNING

#	Base [s]	TFD/M [s]	[%]	OPL [s]	[%]
11	0.22	0.23	7.09	0.22	2.20
12	0.24	0.26	7.19	0.25	2.51
13	0.02	0.02	10.06	0.02	3.25
14	0.03	0.04	10.13	0.03	1.79
15	0.02	0.02	10.78	0.02	2.85
16	0.20	0.22	5.56	0.21	1.47
17	0.24	0.26	5.76	0.25	2.60
18	0.34	0.35	3.19	0.34	1.11
19	0.68	0.71	3.89	0.69	1.61
20	0.86	0.89	3.78	0.87	1.54
21	1.00	1.04	4.64	1.02	2.37
22	0.05	0.06	10.79	0.06	3.28
23	0.04	0.05	14.40	0.04	4.09
24	0.06	0.07	10.61	0.06	3.47
25	0.56	0.59	5.09	0.57	2.08
26	0.56	0.58	4.69	0.57	1.75
27	0.61	0.63	4.09	0.62	2.08
28	1.83	1.87	2.03	1.84	0.66
29	2.04	2.11	3.05	2.08	1.71
30	2.11	2.19	3.94	2.14	1.66

The experiment is intended **to show the computational overhead** of TFD/M and OPL compared to a base line planner not using semantic attachments at all.

The module simply acquires one (most common) predicate from the planner state either with the TFD/M generic module interface or via OPL's domain-specific interface and returns its truth value.

Runtimes in seconds are averaged over 150 trials. The computational overhead in percent is given in comparison to the base line without modules.

Table shows that the runtime scales well for both approaches, and is lower for OPL in all problems instances.

Experiments: International Planning Competition (IPC) 2008

TRANSPORT-NUMERIC

The is designed **to show the impact of acquiring fluent values from the planner state**.

The domain models a logistics task where packages are transported by trucks. The simple volume based condition check to see if another package can be loaded is replaced by a semantic attachment that checks if the package fits geometrically into the truck given the current cargo.

The module requests the sizes of all packages to be stored in the vehicle and the vehicle's capacity from the internal planner state.

L, V, P list the number of locations, vehicles and packages for each problem. The last column gives the relative runtime of OPL compared to TFD/M.

#	L	V	P	TFD/M [s]	OPL [s]	Relative [%]
1	5	2	2	0.01	0.00	-61.6
2	10	2	4	0.12	0.04	-67.7
3	15	3	6	0.52	1.02	94.0
4	20	3	8	1.31	2.10	60.1
5	25	3	10	3.36	2.89	-13.9
6	30	4	12	87.53	17.92	-79.5
7	35	4	14	140.87	68.34	-51.5
8	45	4	18	54.42	29.17	-46.4

The results in the table show that the runtime using the OPL module interface is significantly lower in most cases than the original TFD/M interface's. This is especially visible in the more complex tasks (6 – 8) as in these cases the time for initialization is negligible. As problems (3 – 4) show, the initialization time might dominate the positive runtime effects for simple tasks.

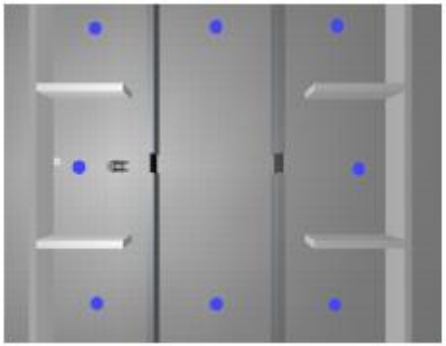
Experiments: real-world environment

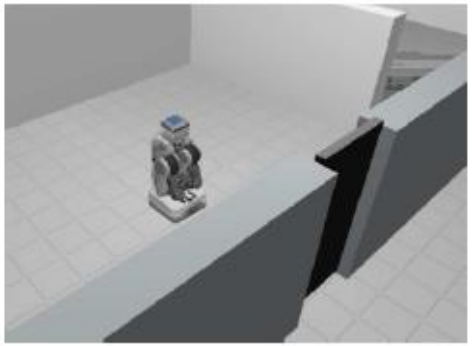
ROOM-SCANNING

The main objective of the experiment **is to show how a complex system is built easily by integrating** an existing path planner into TFD/M using the OPL planner interface.

The domain models an autonomous robot searching for items in various rooms.

The module implementation calculates the real path cost for the drive action by calling the external path planner used by the navigation component of the ROS.





#	Targets	Doors	Search time [s]	Total time [s]
1	2	0	0.09	7.69
2	3	1	0.13	11.92
3	4	1	1.05	17.23
4	5	1	1.76	23.54
5	5	2	2.37	24.44
6	6	2	4.64	33.18
7	7	2	6.17	41.90
8	8	2	11.79	55.24

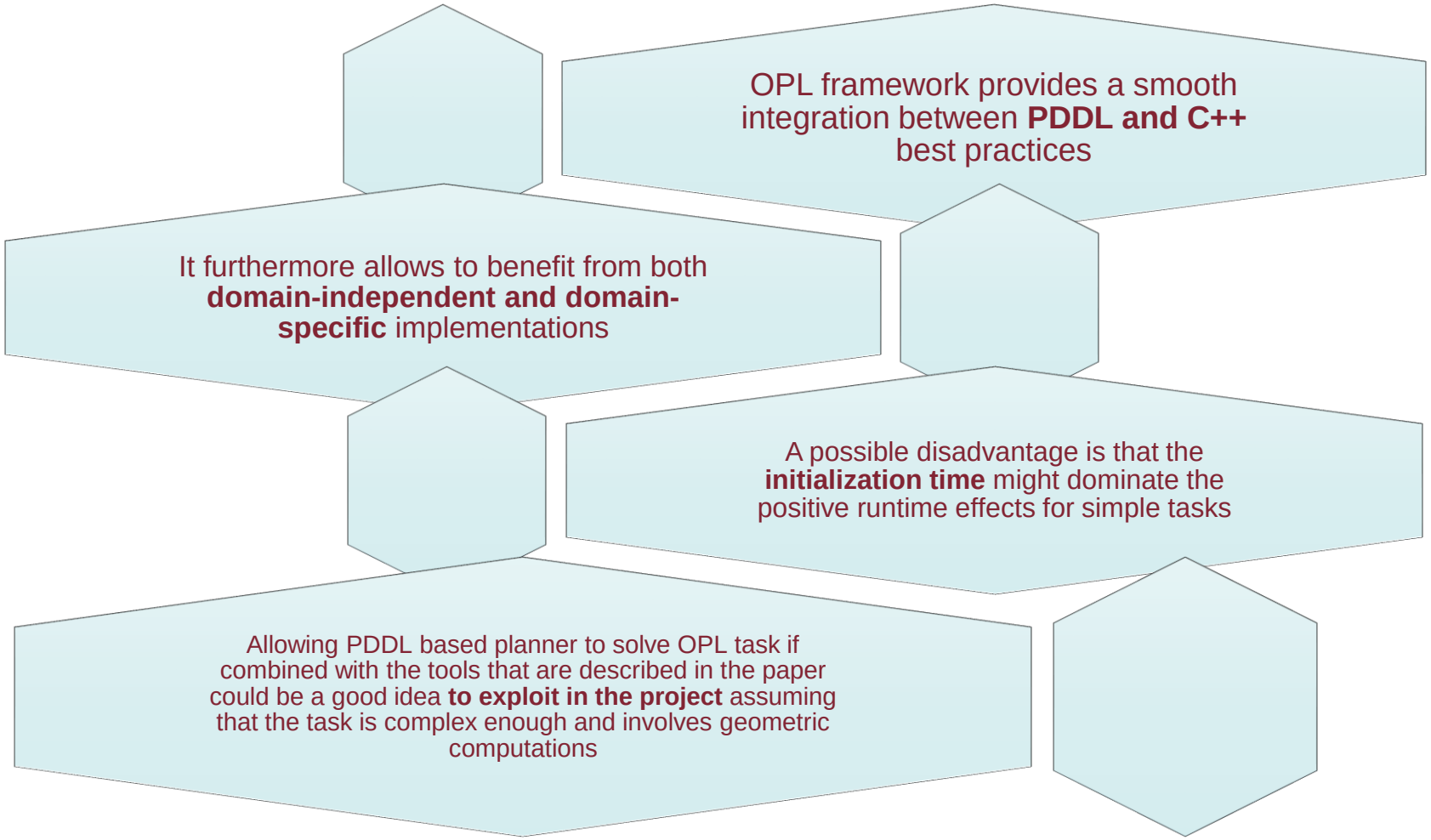
The total time is the time the planner needed to come up with a plan, and the search time is the total time without the runtime of the path planner, separately.

As can be seen in table, the time required to find the first valid plan increases with problem complexity. The observed runtimes are acceptable for use in a real world robotics system and could be improved by using a more efficient path planner.

Conclusion

Personal Comments

Conclusion: personal opinion



Thank you for the listening!