Tutorial Setup

Optional steps to follow along

ROS

http://wiki.ros.org/kinetic/Installation/Ubuntu (ros-kinetic-desktop-full)

sudo apt install ros-kinetic-turtlebot-stage

Set up Workspace

mkdir -p catkin_ws/src; cd catkin_ws/src

Catkin_init_workspace

ROSPlan

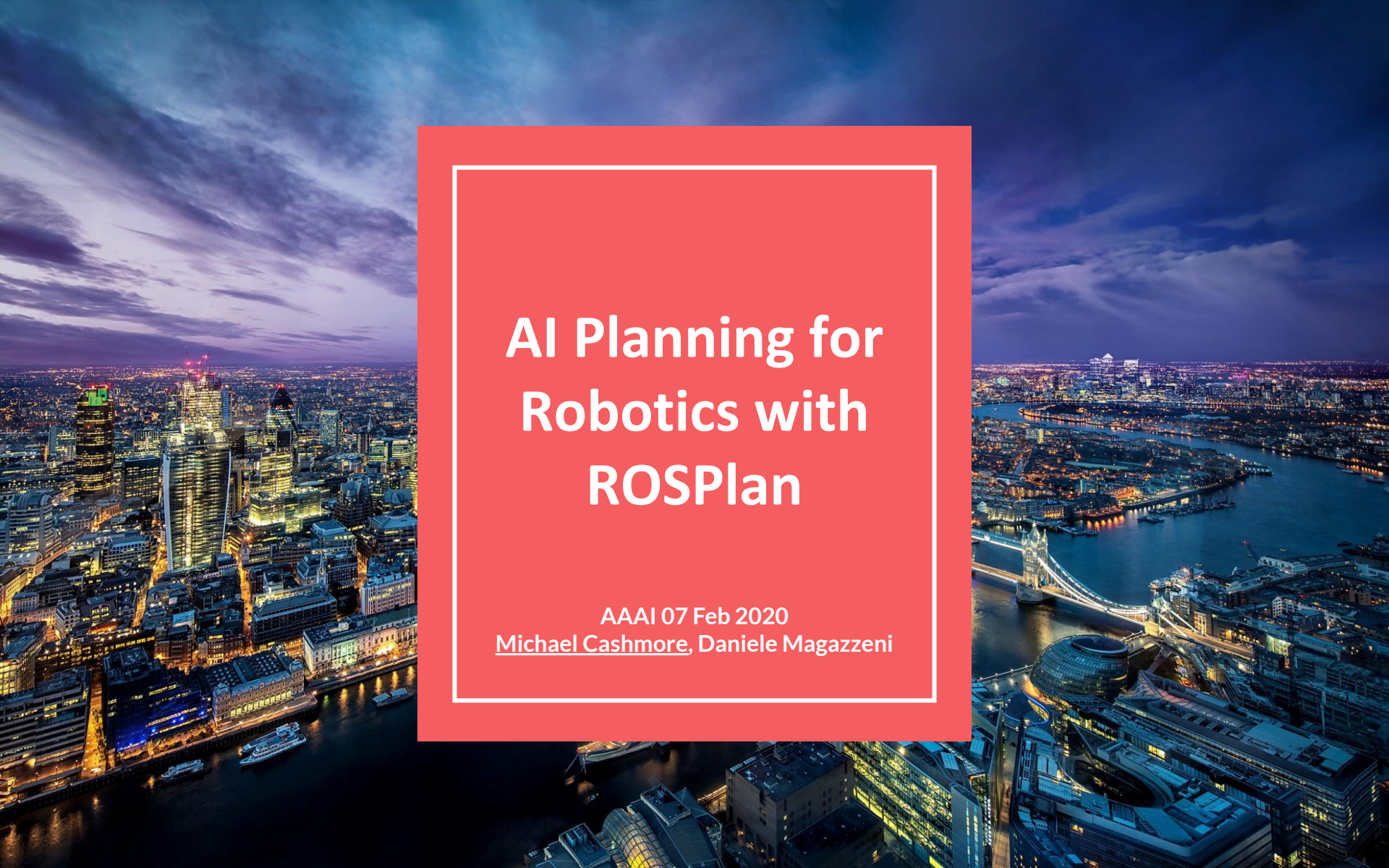
https://github.com/kcl-planning/rosplan (follow instructions to install)

https://github.com/kcl-planning/rosplan_demos

git clone https://github.com/clearpathrobotics/occupancy_grid_utils

catkin build #or (cd ..; catkin_make)

https://github.com/KCL-Planning/ROSPlan/blob/gh-pages/_demos/conference_pages/aaai2020rosplan.zip



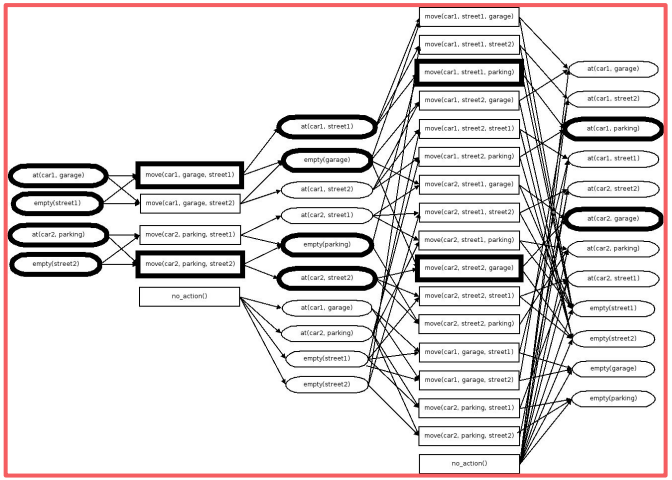
AI Planning for Robotics with ROSPlan

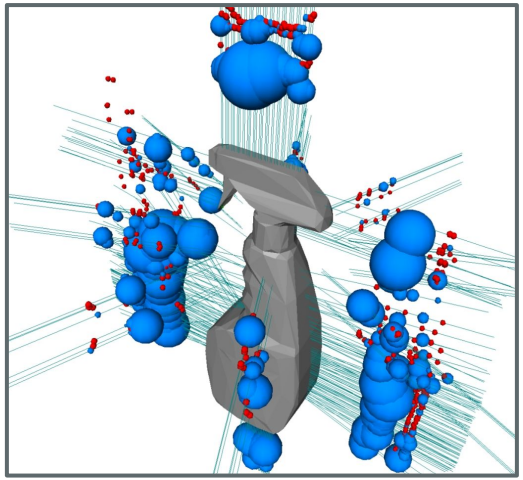
AAAI 07 Feb 2020

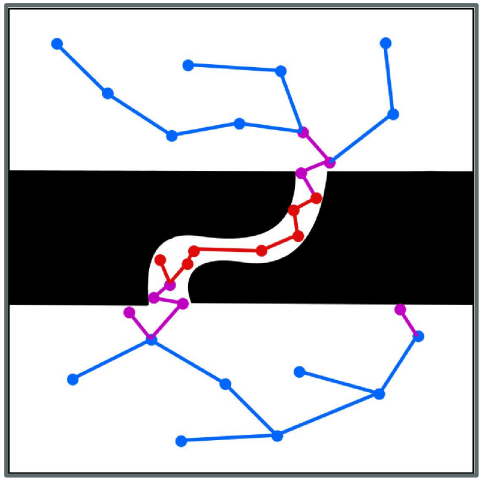
Michael Cashmore, Daniele Magazzeni

Disclaimer

“Planning” is many things.







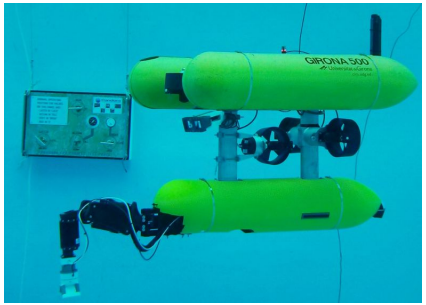
In this tutorial planning is “Task Planning”

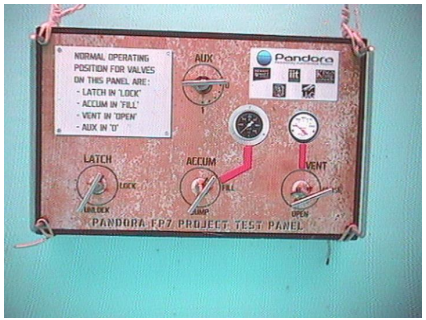
We focus on planning for real applications:

- In robotics
  - Autonomous underwater vehicles.
  - Probabilistic planning with service robots.
  - Strategic-Tactical planning for long-term missions.
- With humans
  - XAIP
  - Human-robot teaming.
- In domains with complex dynamics
  - Energy technology.
  - Smart buildings.
  - Urban traffic control.

Our research in robotics:

PANDORA





Planning long-term inspection and maintenance missions for Autonomous Underwater Vehicles.

Toward Persistent Autonomous Intervention in a Subsea Panel. Autonomous Robots. (2016)

Opportunistic Planning in Autonomous Underwater Missions. IEEE Transactions on Automation Science and Engineering. (2017)

AUV mission control via temporal planning. IEEE international conference on robotics and automation (ICRA 2014)

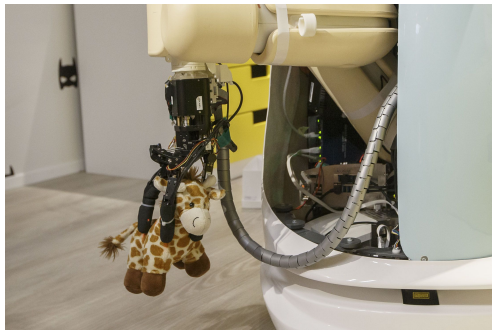
Planning Inspection Tasks for AUVs. Proceedings of the MTS/IEEE Oceans 2013 Conference, San Diego (OCEANS 2013)

KING'S College LONDON

Human-AI Teaming Lab

Our research in robotics:

SQUIRREL



"Clearing clutter bit-by-bit"

Planning to explore environments, tidy areas, and interact with children in a clutter home or kindergarten environment.

On-the-fly detection of novel objects in indoor environments
IEEE International Conference on Robotics and Biomimetics (ROBIO 2017)

Short-Term Human Robot Interaction through Conditional Planning and Execution (ICAPS 2017)

Strategic Planning for Autonomous Systems over Long Horizons. Proceedings of the 4th ICAPS Workshop on Planning and Robotics (PlanRob 2016)

ROSPlan: Planning in the Robot Operating System. Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)

Task Planning and Robotics

From (Ingrand & Ghallab, 2017)

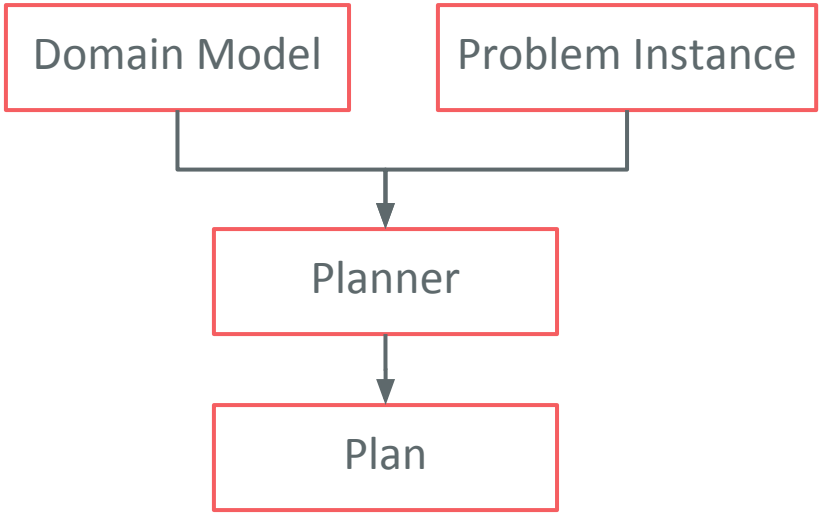
- Planning
- Acting
- Observing
- Monitoring
- Goal reasoning
- Learning

Task Planning and Robotics

From (Ingrand & Ghallab, 2017)

- **Planning**
- Acting
- Observing
- Monitoring
- Goal reasoning
- Learning

Organising actions to achieve goals, before starting to execute them.



Task Planning and Robotics

Planning

- uses a model of the world in order to predict and anticipate its behaviour in order to choose actions that will lead to desirable states.
- assumes the world can be modelled as a finite collection of state variables and that actions cause changes in the values of those variables.

Given such a model including *actions*, an *initial state*, and *goal*. Planning will find actions (a *plan*) that when applied from the initial state, is predicted to achieve the goals.

Robots can follow a plan.

Task Planning and Robotics

Planning will find actions (a *plan*) that when applied from the initial state, is predicted to achieve the goals.

A robot can follow that plan

ACTION 1

ACTION 2

ACTION 3

Task Planning and Robotics

Planning will find actions (a *plan*) that when applied from the initial state, is predicted to achieve the goals.

A robot can follow that plan until...

- It can’t because the model was wrong.
- It can’t because the model was not accurate enough.
- The state is too mismatched with reality. It just doesn’t want to any more.

ACTION 1

ACTION 2

ACTION 3

Task Planning and Robotics

- Planning online: deciding when to plan and how long to plan.
- Detecting plan failure during execution.
- Handling risk.
- Planning in response to partial information.
- Planning with nondeterminism.
- Reaction to unexpected events.
- Recovering the state.
- Integration with control (e.g. task and motion planning).
- Replanning (online).
- Planning and execution with imperfect communication.
- Humans.

Task Planning and Robotics

From (Ingrand & Ghallab, 2017)

- **Planning**
- **Acting**
- **Observing**
- **Monitoring**
- Goal reasoning
- Learning

Acting: Executing the chosen actions, often through closed-loop reactive functions.

Observing: Collecting data from sensors, and using data collected from sensors to update the current state description.

Monitoring: Comparing the current state description and system state with what was predicted by the model.

Task Planning and Robotics

Systems	Associated components	Delibe-ration Functions	Knowledge Representation	Robotic Domains	Architecture	Main Additional Features
DS1/RAX [157]	PS [110] EXEC Livingstone [227]	PAMGO	State Variables time-points Qualitative model Transition automata with probabilities and costs	Autonomous Spacecraft	Layered (RA)	Time management Resource management Diagnostic and recovery
IxTeT/PRS [108]	IxTeT [90] Exec [137] Reco [65] PRS [106] GenoM [76]	PAMO	State Variables, time-points Chronicles Procedures	Exploration & rescue robots	Layered (LAAS)	Time management Resource management Verification and validation (functional layer)
IDEA [155]	IDEA EUROPA [79]	PA	State Variables, intervals	Exploration robots	Agent-based (IDEA)	Time management Integrated knowledge representation Refinement with lookahead
T-REX	T-REX [147] EUROPA [79] APSI[81]	PA	State Variables, intervals	AUV Service and domestic robots	Teleo reactive	Time management Integrated knowledge representation Refinement with lookahead
RMPL[225]	Titan [226] Burton [228] Kirk [224] Drake [52] Pike [140]	PAM	Probabilistic Timed hier-archical automata Temporal Plan Networks Probabilistic TPNs, TPNs with Uncertainty	Exploration & rescue robots Autonomous Spacecraft	Specific	Time management Diagnostic and recovery Integrated knowledge representation Refinement with lookahead Handling of uncertainty
WITAS [63]	TALPlanner [133] DyKnow [101]	PAMO	Temporal Action Logic Stream, policy Chronicles	Autonomous aerial vehicles	Layered [62]	Time management Resource management Integrated knowledge representation Explicit handling of uncertainty Verification and validation
ASPEN/ CASPER	ASPEN [178] CASPER [43] TCA [200] TDL [201] Plexil [216]	PA	State Variables Intervals Programs	Exploration & rescue robots	Layered	Time management Resources management
XFRM	XFRM [16] RPL [15] SRC [14] SRP [17] CRAM [18]	PAL	RPL, CRAM Plan Language Prolog	Service and domestic robots	Open world (CRAM)	Integrated knowledge representation Refinement with lookahead Open models
Cypress [223]	Sipe [222] PRS [158] CPEF [159]	PAG	ACT Formalism	Military		Time management Resource management Integrated knowledge representation Refinement with lookahead

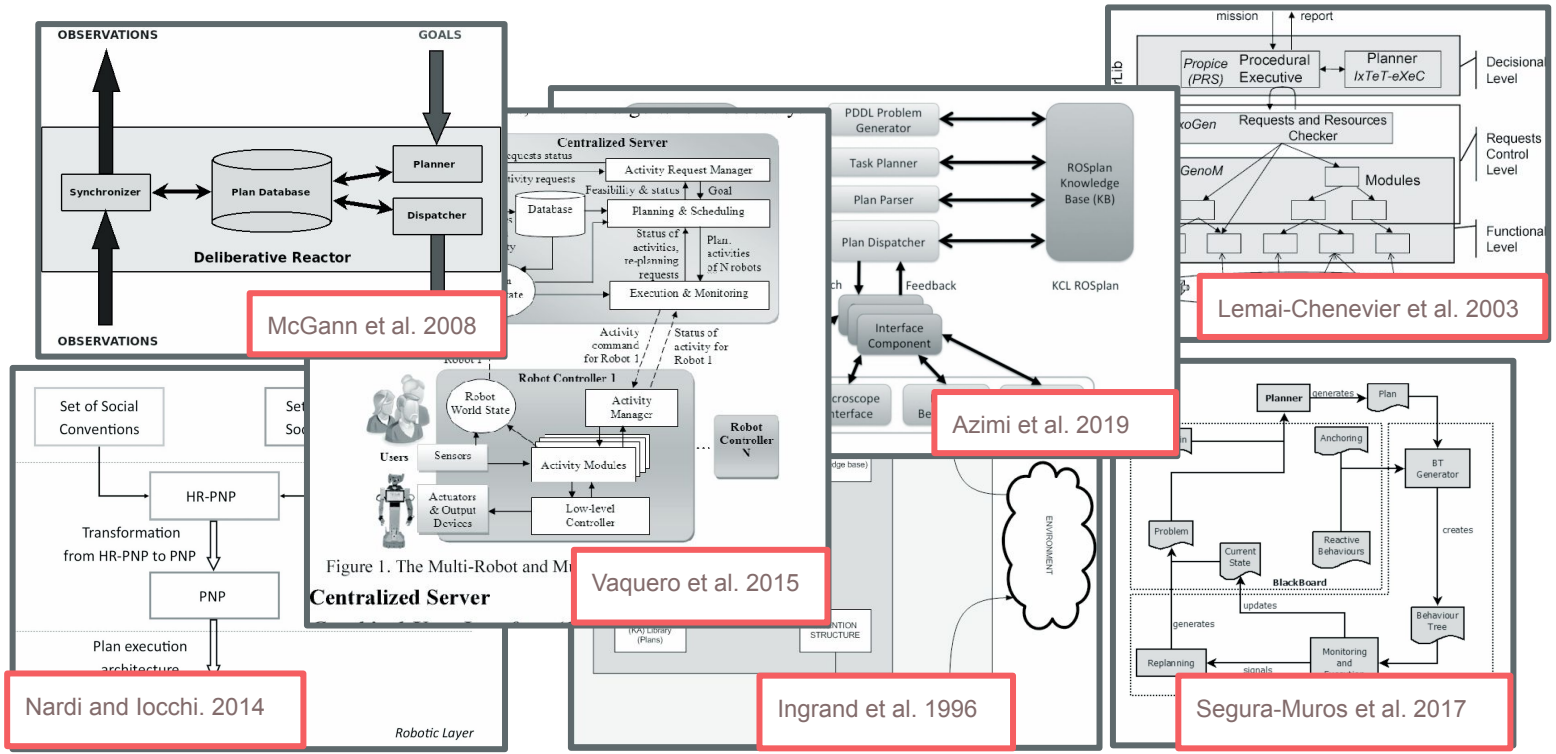
Table 2: Analyzed systems, clustered into approaches and associated components, with their main deliberation functions (P: planning, A: acting, M: monitoring, O: observing, G: goal reasoning, L: learning) and other features.

Survey of systems for planning and robotics (part 1 of 2)

[Ingrand & Ghallab, 2017]

Task Planning and Robotics

A wealth of system architectures.



AI Planning for Robotics with ROSPlan

How to connect planning and robotics?

AI Planning for Robotics with ROSPlan

How to connect planning and robotics?

- Reusing existing software.

  Many existing systems and algorithms to be leveraged.

AI Planning for Robotics with ROSPlan

How to connect planning and robotics?

- Reusing existing software.
  Many existing systems and algorithms to be leveraged.

- Without restricting the system.
  Completely new architectures can be built, incorporating existing components.

AI Planning for Robotics with ROSPlan

How to connect planning and robotics?

- Reusing existing software.
  Many existing systems and algorithms to be leveraged.

- Without restricting the system.
  Completely new architectures can be built, incorporating existing components.

- In a straightforward (easy) way.
  Time should be spent on the part of the system that is interesting.

(Disclaimer 2: This is a tutorial and aims to be accessible.)

AI Planning for Robotics with ROSPlan

ROSPlan: Connecting standards: PDDL* and ROS

- **ROS**
- **ROSPlan**
- **Using ROSPlan**
  - Planning
  - Acting and Monitoring
  - Observing and more Monitoring
- **Simulation Examples**

* PDDL, PDDL2.1, PDDL+, PPDDL, and RDDL

Part 1:
ROS

What is ROS?

A set of software libraries and tools that help you build robot applications.

- A ROS system is distributed into ***nodes***.
- A system is configured by ***parameters***, and then all the nodes are ***launched***.
- Nodes talk by publishing ***messages*** and calling ***services***.

What is ROS?

A set of software libraries and tools that help you build robot applications.

- A ROS system is distributed into ***nodes***.
- A system is configured by ***parameters***, and then all the nodes are ***launched***.
- Nodes talk by publishing ***messages*** and calling ***services***.

ROS tutorials online:

http://wiki.ros.org/ROS/Tutorials

Nodes

A node is its own separate program.

It is usually one process, but it can be multiple processes.

Nodes have:

- A node type.
- A unique name.
- Launch parameters.

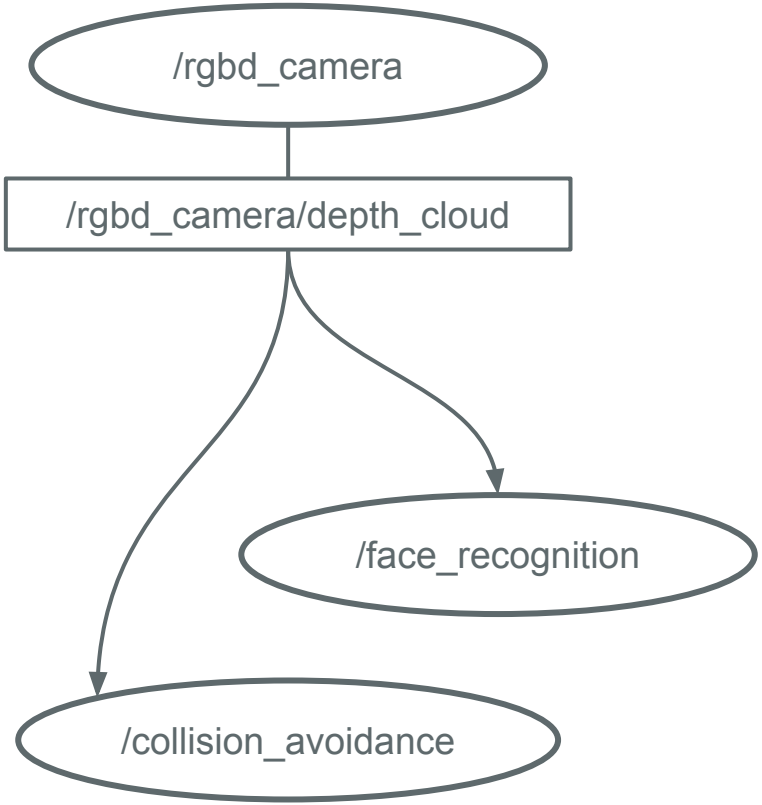
Parameters

A parameter server keeps track of parameters, which can be accessed by name.

Parameters are intended to be fairly static values that do not change.

They are well suited for initial configuration.

Messages



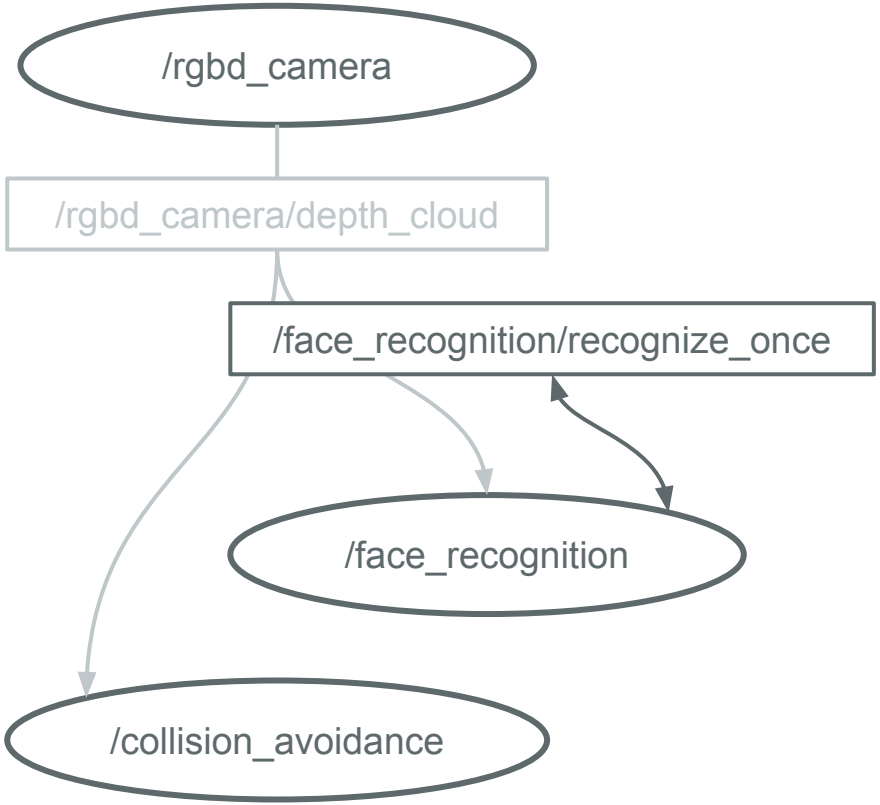
Nodes talk to one another through ROS messages.

A ROS message is published on a topic. Every node subscribed to that topic will recieve the message.

Each Topic has:

- A list of publishers and subscribers.
- A message type.
- A globally unique name.

Services



Some nodes provide a remote procedure call, called a service.

Services have:

- a globally unique topic name.
- A single service provider.
- A request and response type.

They are different from messages:

- Calling a service is blocking.
- Services have a return value.

Nodes (and launch files)

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

> roslaunch roslaunch rospy_tutorials talker_listener.launch

> rosnode list

> rosnode info /talker

Parameters

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

> rosparam list

> rosparam get /run_id

Messages

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

> rostopic list

> rostopic info /chatter

> rostopic echo /chatter -n 5

Services

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

> rosservice list

> rosservice info /talker/get_loggers

> rosservice call /talker/get_loggers "{}"

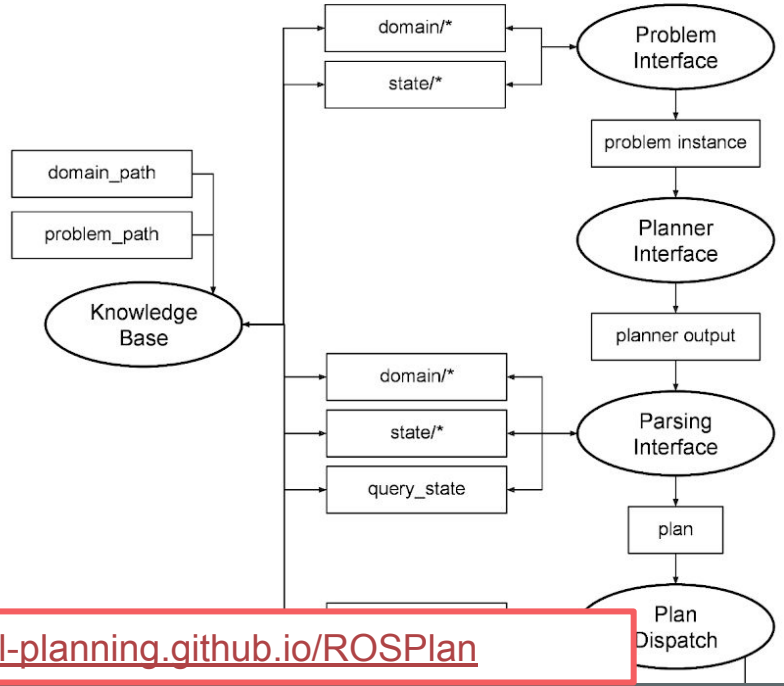
Part 2: ROSPlan
Nodes, Parameters, and Launch

ROSPlan

Home | Documentation & Tutorials | Virtual Machine | Demos and Conferences | Publications | View on GitHub | Contact

ROSPlan Overview

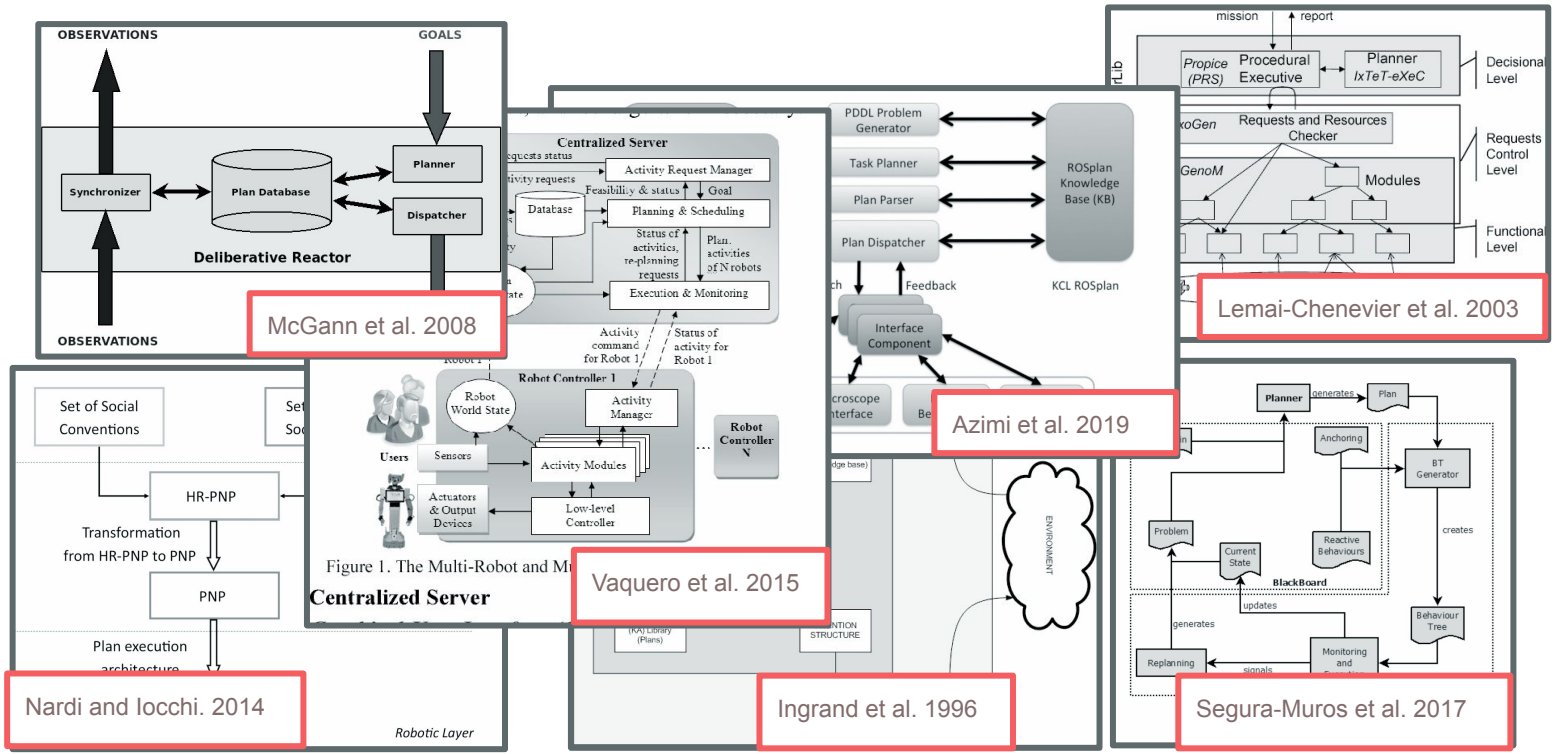
The ROSPlan framework provides a collection of tools for AI Planning in a ROS system. ROSPlan has a variety of nodes which encapsulate planning, problem generation, and plan execution.



https://kcl-planning.github.io/ROSPlan

Task Planning and Robotics

A wealth of system architectures.



Task Planning and Robotics

A wealth of system architectures.

What components (or interfaces) are:

- Common to all system architectures?
- Atomic (perform a single function that cannot be broken down)?

Task Planning and Robotics

A wealth of system architectures.

What components (or interfaces) are:

- Common to all system architectures?
- Atomic (perform a single function that cannot be broken down)?

The following nodes can be combined in different ways to form a variety of system architectures.

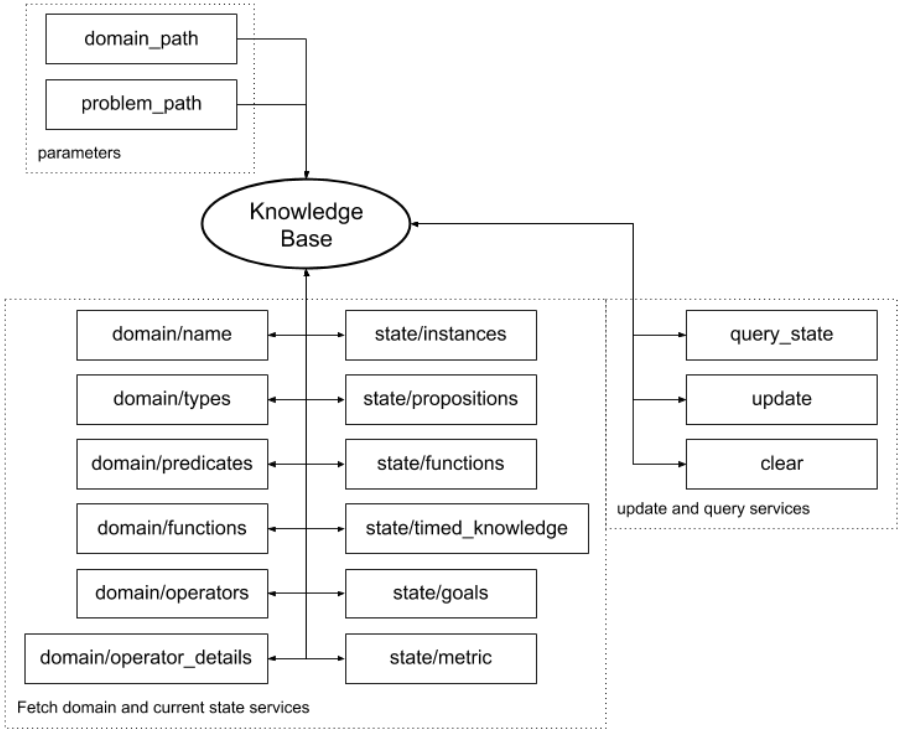
Each performs a single function and can be replaced by more sophisticated methods, or used alone within a custom framework.

Knowledge Management System

The KMS stores the domain model and the current state.

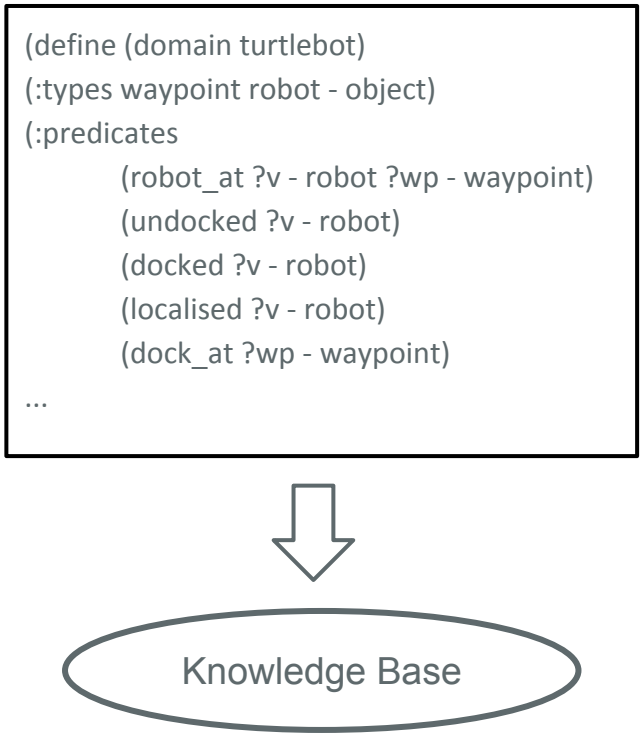
The node is launched with a domain file parameter, and an optional problem file parameter, which specifies the initial state.

There are many services for fetching domain details, the current state, and performing queries.



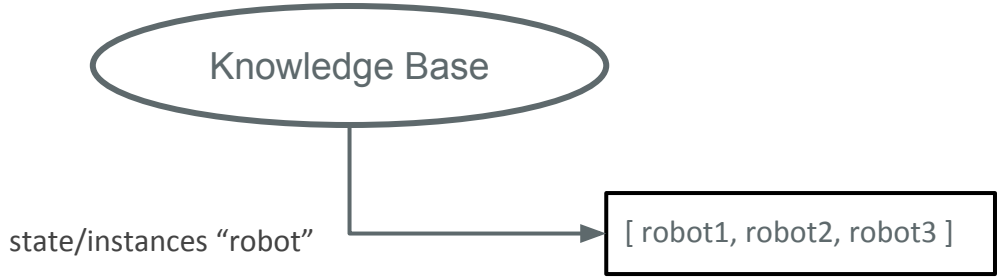
Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.



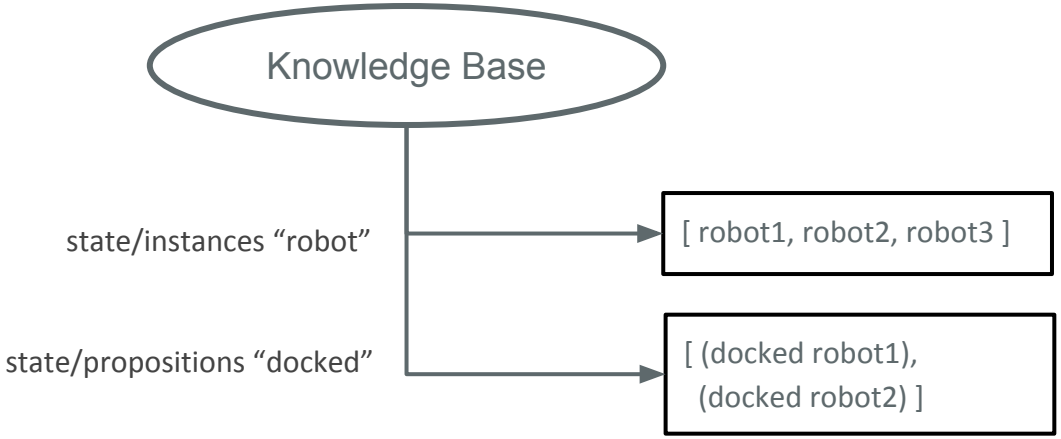
Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.



Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.

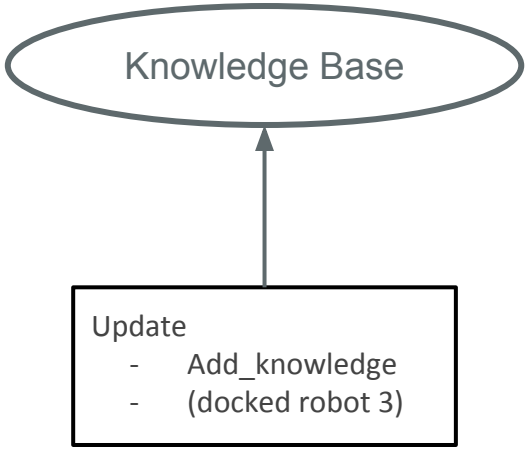


Knowledge Management System

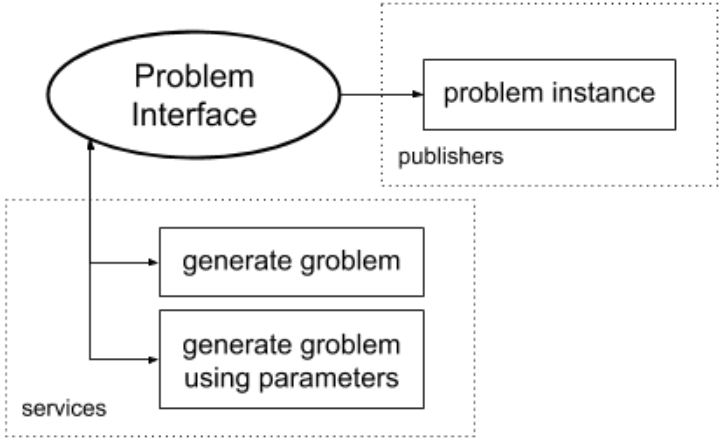
The Knowledge Base node is used to access a domain and state from any node.

The Knowledge Base can be updated based on sensed data to represent the current state.

A persistent state is built from **observations** over time and is essential for online planning.



Problem Interface



This node is used to generate a problem instance.

It fetches the domain details and current state through service calls to the KMS.

The problem instances can be written to a file and/or published as a ROS message.

Problem Interface


```
(define (problem task)
(:domain turtlebot)
(:objects
   wp0 wp1 wp2 wp3 - waypoint
   kenny - robot
)
(:init
   (robot_at kenny dock-station)
   (docked kenny)
…
(:goal (and
   (docked kenny)
…
```

This node is used to generate a problem instance.

It fetches the domain details and current state through service calls to the KMS.

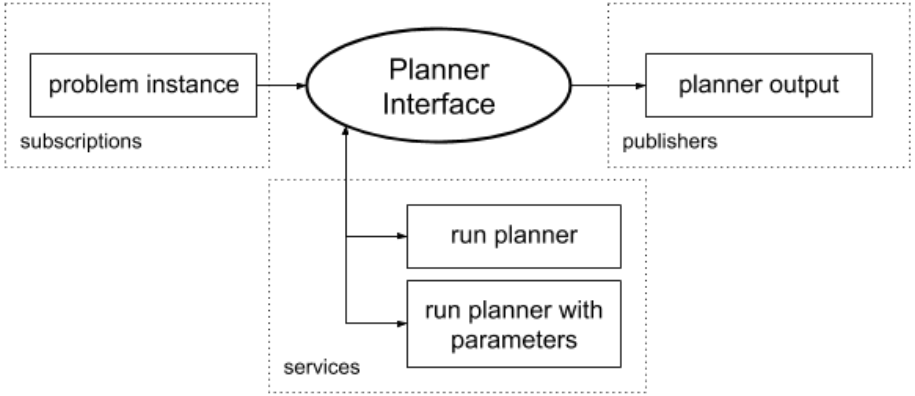
The problem instances can be written to a file and/or published as a ROS message.

Planner Interface

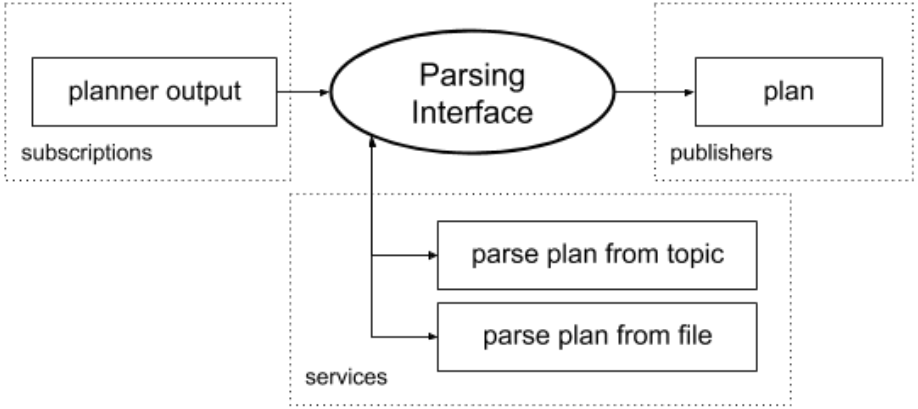
This node is a wrapper for the AI Planner. The *run planner* service returns true if a solution is found by the planner.

The command line used to run the planner is specified by a launch parameter.

The resulting solution, if one was found, can be written to a file and/or published as a ROS message.



Parsing Interface



This node is used to convert planner output into a representation for execution.

A ROS message is created for each individual action.

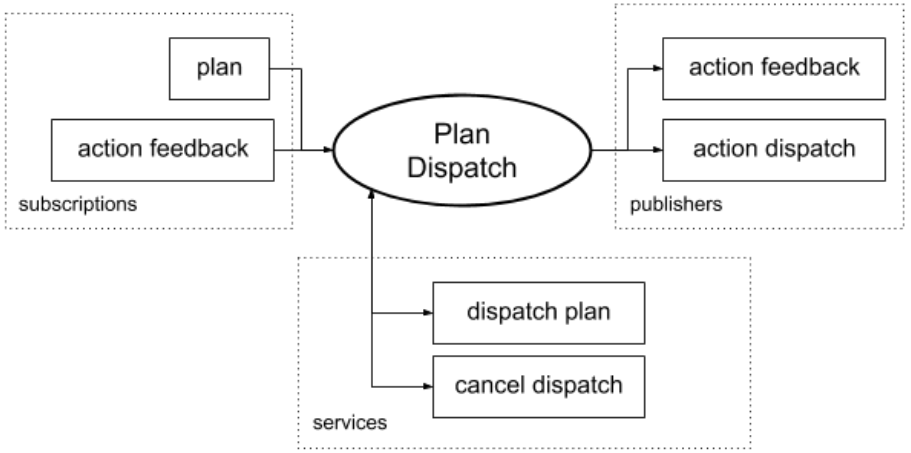
The ROS messages are stored together in one plan message, which can also describe how the plan should be executed.

Plan Dispatch

This node is responsible for executing a plan message by publishing the action messages at the right times.

The Plan Dispatch node is closely tied to the Parsing Interface. They must use the same plan representation.

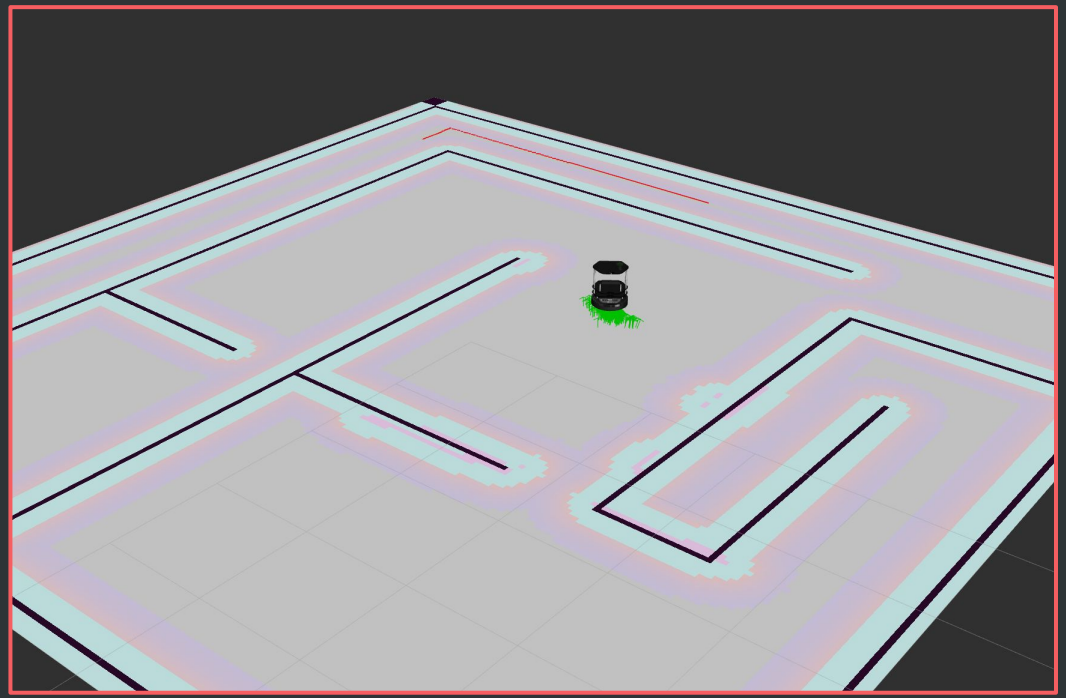
The plan is executed as a service, whose response includes whether the plan was executed without errors, and achieved the goal.



```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"    value="rosplan_knowledge_base" />
        <arg name="domain_path"       value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"      value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"     value="problem_instance" />
    </include>

</launch>
```

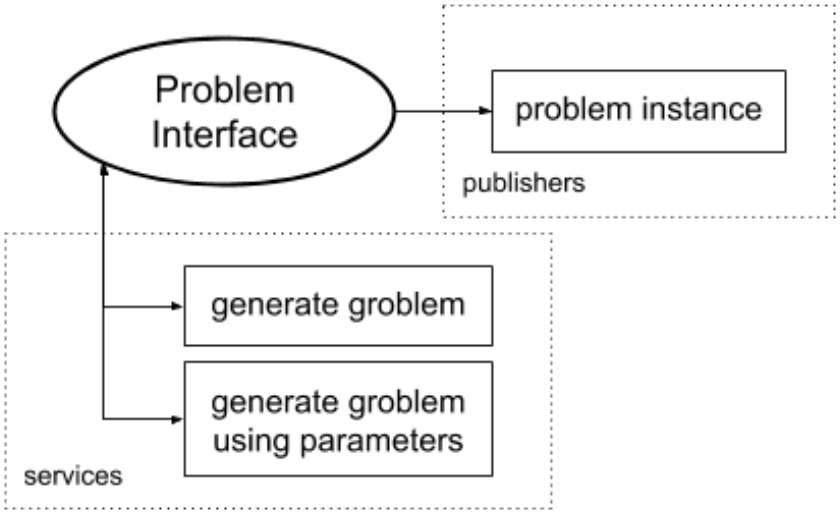
```
(define (domain turtlebot)

(:requirements :strips :typing :fluents :disjunctive-preconditions :durative-actions)

(:types
    waypoint
    robot
)

(:predicates
    (robot_at ?v - robot ?wp - waypoint)
    (visited ?wp - waypoint)
    (undocked ?v - robot)
    (docked ?v - robot)
    (localised ?v - robot)
    (dock_at ?wp - waypoint)
)

;; Move to any waypoint, avoiding terrain
(:durative-action goto_waypoint
    :parameters (?v - robot ?from ?to - waypoint)
    :duration ( = ?duration 60)
    :condition (and
        (at start (robot_at ?v ?from))
        (at start (localised ?v))
        (over all (undocked ?v)))
    :effect (and
        (at end (visited ?to))
        (at end (robot_at ?v ?to))
        (at start (not (robot_at ?v ?from))))
)

;; Localise
(:durative-action localise
    :parameters (?v - robot)
    :duration ( = ?duration 60)
    :condition (over all (undocked ?v))
    :effect (at end (localised ?v))
)

;; Dock to charge
(:durative-action dock
    :parameters (?v - robot ?wp - waypoint)
    :duration ( = ?duration 30)
    :condition (and
        (over all (dock_at ?wp))
        (at start (robot_at ?v ?wp))
        (at start (undocked ?v)))
    :effect (and
        (at end (docked ?v))
        (at start (not (undocked ?v))))
)

(:durative-action undock
    :parameters (?v - robot ?wp - waypoint)
    :duration ( = ?duration 10)
    :condition (and
        (over all (dock_at ?wp))
        (at start (docked ?v)))
    :effect (and
        (at start (not (docked ?v)))
        (at end (undocked ?v)))
)
```

```
(define (problem task)
(:domain turtlebot)
(:objects
    wp0 wp1 wp2 wp3 wp4 - waypoint
    kenny - robot
)
(:init
    (robot_at kenny wp0)
    (docked kenny)
    (dock_at wp0)
)
(:goal (and
    (visited wp0)
    (visited wp1)
    (visited wp2)
    (visited wp3)
    (visited wp4)
    (docked kenny)
)))
```



Problem Generator

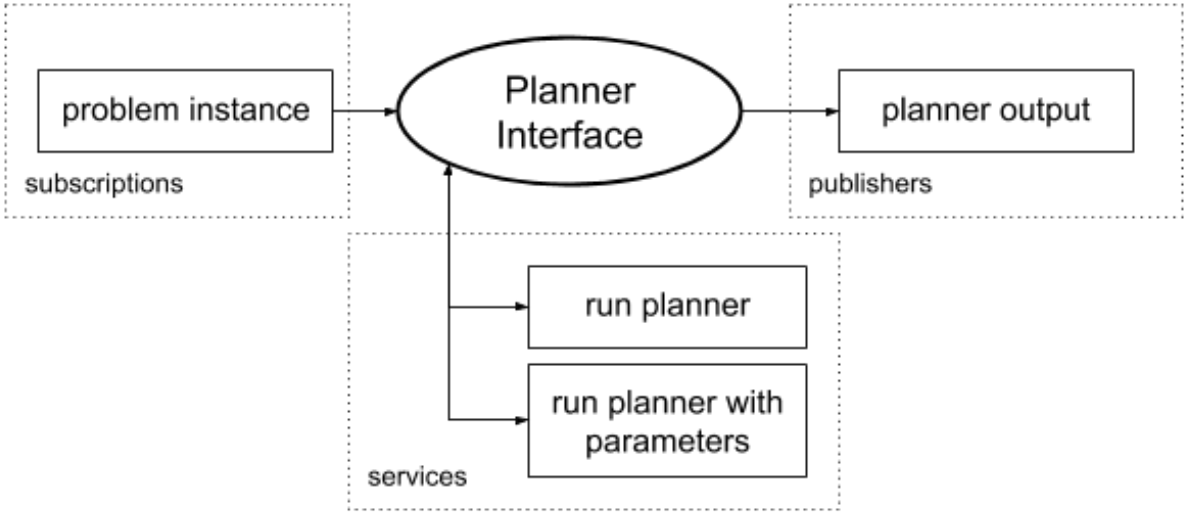
Problem Generation



```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"    value="rosplan_knowledge_base" />
        <arg name="domain_path"       value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"      value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"     value="problem_instance" />
    </include>

</launch>
```

Planner Interface

Planner Interface



```
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"   value="rosplan_knowledge_base" />
        <arg name="domain_path"      value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"     value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

</launch>
```

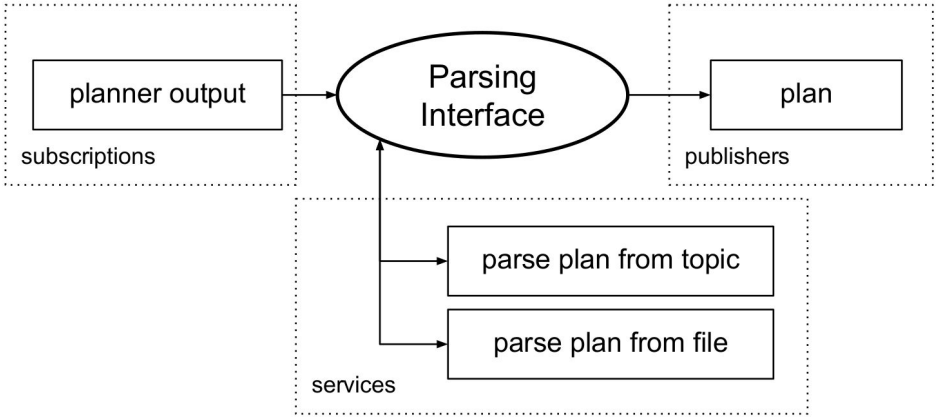
```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"   value="rosplan_knowledge_base" />
        <arg name="domain_path"      value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"     value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

</launch>
```

```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"   value="rosplan_knowledge_base" />
        <arg name="domain_path"      value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"     value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

</launch>
```

```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />
    <arg name="autom_gen_problem_path" default="$(find rosplan_demos)/common/problem.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="knowledge_base"   value="rosplan_knowledge_base" />
        <arg name="domain_path"      value="$(arg domain_path)" />
        <!-- problem_path: pddl problem will be automatically generated and placed in this location -->
        <arg name="problem_path"     value="$(arg autom_gen_problem_path)" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

</launch>
```

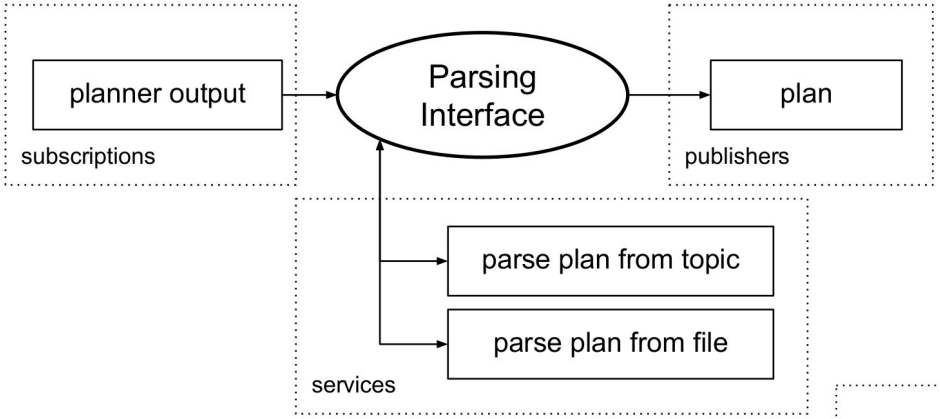
Plan Execution 01

Plan Execution

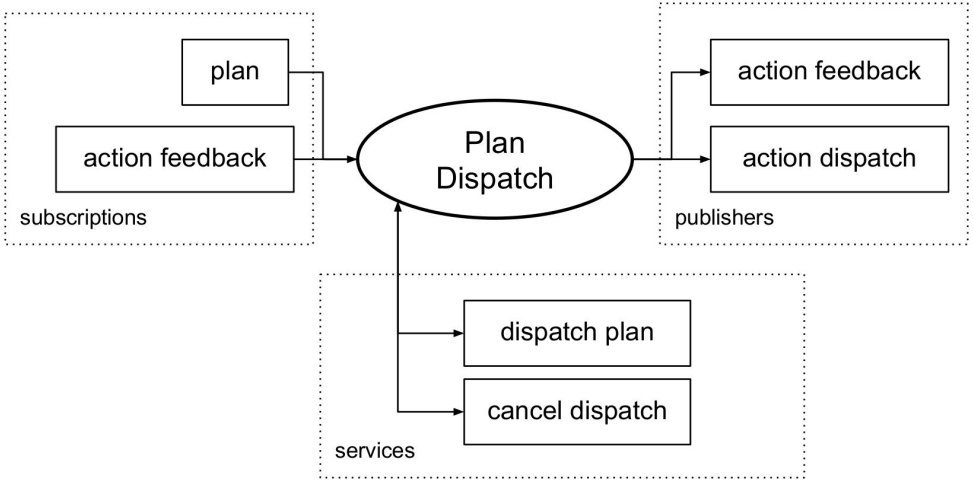


- Converts the PDDL plan into ROS messages for dispatch.

Plan Execution



- Converts the PDDL plan into ROS messages for dispatch.



- Publishes PDDL actions as ROS messages.
- Implements a plan execution algorithm.

```
<!-- planner interface -->
<include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
    <arg name="use_problem_topic"    value="true" />
    <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
    <arg name="planner_topic"        value="planner_output" />
    <arg name="domain_path"          value="$(arg domain_path)" />
    <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
    <arg name="data_path"            value="$(find rosplan_demos)/common/" />
    <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
</include>

<!-- plan parsing -->
<node name="rosplan_parsing_interface" pkg="rosplan_planning_system" type="pddl_simple_plan_parser" respawn="false" output="screen">
    <param name="knowledge_base" value="rosplan_knowledge_base" />
    <param name="planner_topic"  value="/rosplan_planner_interface/planner_output" />
    <param name="plan_topic"     value="complete_plan" />
</node>

<!-- plan dispatching -->
<node name="rosplan_plan_dispatcher" pkg="rosplan_planning_system" type="pddl_simple_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="rosplan_knowledge_base" />
    <param name="plan_topic"            value="/rosplan_parsing_interface/complete_plan" />
    <param name="action_dispatch_topic" value="action_dispatch" />
    <param name="action_feedback_topic" value="action_feedback" />
</node>
```

Plan Execution 02
Simulated Actions

Action Interfaces

A ROS message representing a PDDL action must be connected to the implementation of that action.

- To an existing library through ROS service(s).
- To an existing library through ROS actionlib message topics.
- To a custom implementation.
- Or to a combination of the above.

Action Interfaces

(goto_waypoint robot_10 wp3 wp9)

Action Interfaces

(goto_waypoint robot_10 wp3 wp9)

- Is the arm already stowed for movement?
- What are the coordinates of the waypoint?
- What is the orientation of the waypoint - is it correct for the next action?
- Does the action have its own timeout?
- Can the action be preempted?

Action Interfaces

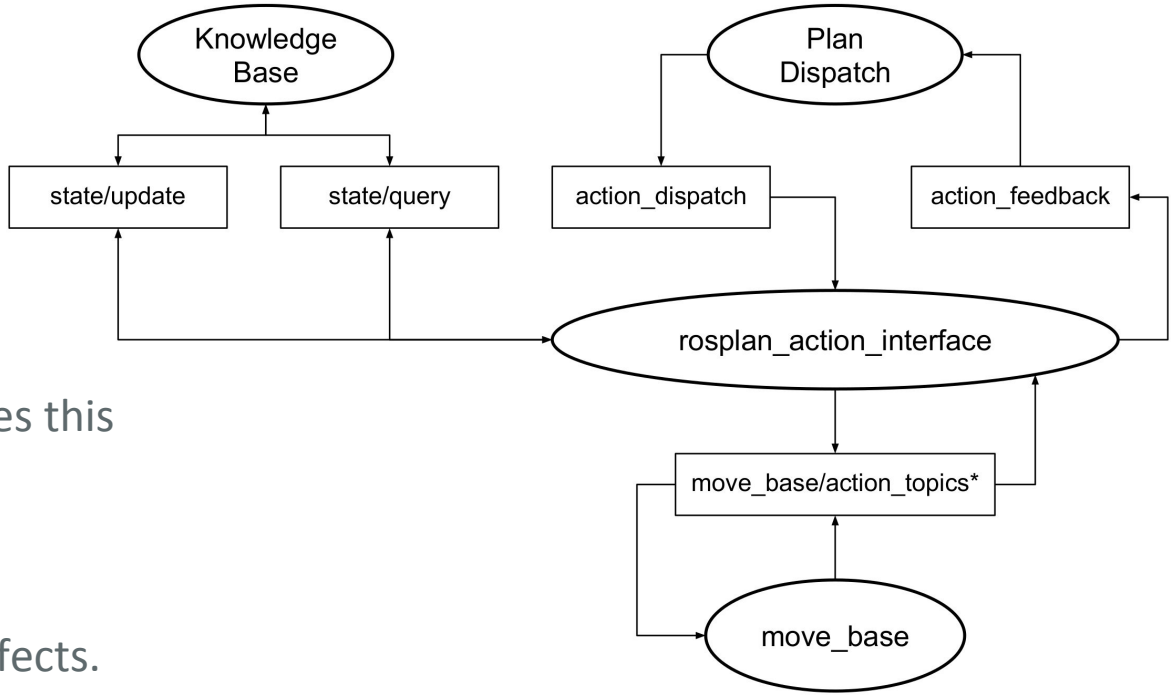
(goto_waypoint robot_10 wp3 wp9)

- Are all the effects of the action possible to sense?

Action Interfaces

- Subscribes to PDDL actions.
- Calls execution.
- (Optionally) updates the Knowledge Base with action effects.

Action Interfaces



The action interface node makes this simpler to implement.

- Checks PDDL parameters.
- Updates KB with action effects.
- Subscription to monitor action cancellation.
- **Calls abstract execution method.**
- Returns action feedback.

```
<!-- planner interface -->
<include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
    <arg name="use_problem_topic"    value="true" />
    <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
    <arg name="planner_topic"        value="planner_output" />
    <arg name="domain_path"          value="$(arg domain_path)" />
    <arg name="problem_path"         value="$(arg autom_gen_problem_path)" />
    <arg name="data_path"            value="$(find rosplan_demos)/common/" />
    <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
</include>

<!-- plan parsing -->
<node name="rosplan_parsing_interface" pkg="rosplan_planning_system" type="pddl_simple_plan_parser" respawn="false" output="screen">
    <param name="knowledge_base" value="rosplan_knowledge_base" />
    <param name="planner_topic"  value="/rosplan_planner_interface/planner_output" />
    <param name="plan_topic"     value="complete_plan" />
</node>

<!-- plan dispatching -->
<node name="rosplan_plan_dispatcher" pkg="rosplan_planning_system" type="pddl_simple_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="rosplan_knowledge_base" />
    <param name="plan_topic"            value="/rosplan_parsing_interface/complete_plan" />
    <param name="action_dispatch_topic" value="action_dispatch" />
    <param name="action_feedback_topic" value="action_feedback" />
</node>
```

```
<!-- sim actions -->
<include file="$(find rosplan_planning_system)/launch/includes/simulated_action.launch" >
    <arg name="pddl_action_name" value="undock" />
</include>
<include file="$(find rosplan_planning_system)/launch/includes/simulated_action.launch" >
    <arg name="pddl_action_name" value="dock" />
</include>
<include file="$(find rosplan_planning_system)/launch/includes/simulated_action.launch" >
    <arg name="pddl_action_name" value="localise" />
</include>
<include file="$(find rosplan_planning_system)/launch/includes/simulated_action.launch" >
    <arg name="pddl_action_name" value="goto_waypoint" />
</include>
```

```
<!-- simulated action -->
<node name="rosplan_interface_$(arg pddl_action_name)" pkg="rosplan_planning_system" type="simulatedAction" respawn="false" output="screen">
    <param name="knowledge_base"          value="$(arg knowledge_base)" />
    <param name="pddl_action_name"        value="$(arg pddl_action_name)" />
    <param name="action_duration"         value="$(arg action_duration)" />
    <param name="action_duration_stddev" value="$(arg action_duration_stddev)" />
    <param name="action_probability"      value="$(arg action_probability)" />
    <param name="action_dispatch_topic"   value="$(arg action_dispatch_topic)" />
    <param name="action_feedback_topic"   value="$(arg action_feedback_topic)" />
</node>
```

The simulated action node

Extends the action interface, implementing the action execution by sleeping. Useful for simulation and testing a system!

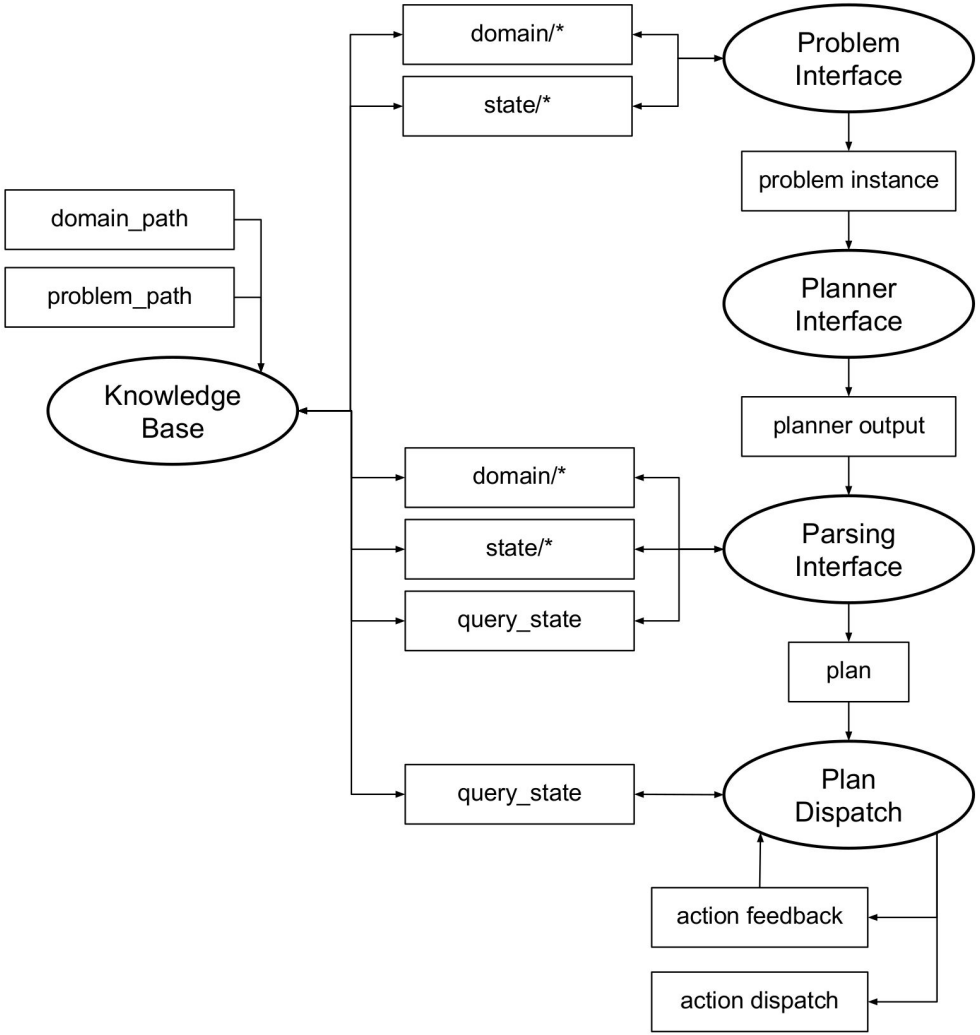
Parameters:

- Action success probability
- Fixed action duration.
- Action duration normal distribution.

Plan Execution 03

Example System

All of the nodes required to perform a simple replanning system.



```
<!-- plan parsing -->
<node name="rosplan_parsing_interface" pkg="rosplan_planning_system" type="pddl_simple_plan_parser" respawn="false" output="screen">
    <param name="knowledge_base" value="rosplan_knowledge_base" />
    <param name="planner_topic"  value="/rosplan_planner_interface/planner_output" />
    <param name="plan_topic"     value="complete_plan" />
</node>

<!-- plan dispatching -->
<node name="rosplan_plan_dispatcher" pkg="rosplan_planning_system" type="pddl_simple_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="rosplan_knowledge_base" />
    <param name="plan_topic"            value="/rosplan_parsing_interface/complete_plan" />
    <param name="action_dispatch_topic" value="action_dispatch" />
    <param name="action_feedback_topic" value="action_feedback" />
</node>
```

```
<!-- plan parsing -->
<node name="rosplan_parsing_interface" pkg="rosplan_planning_system" type="pddl_simple_plan_parser" respawn="false" output="screen">
    <param name="knowledge_base" value="rosplan_knowledge_base" />
    <param name="planner_topic"  value="/rosplan_planner_interface/planner_output" />
    <param name="plan_topic"     value="complete_plan" />
</node>

<!-- plan dispatching -->
<node name="rosplan_plan_dispatcher" pkg="rosplan_planning_system" type="pddl_simple_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="rosplan_knowledge_base" />
    <param name="plan_topic"            value="/rosplan_parsing_interface/complete_plan" />
    <param name="action_dispatch_topic" value="action_dispatch" />
    <param name="action_feedback_topic" value="action_feedback" />
</node>
```

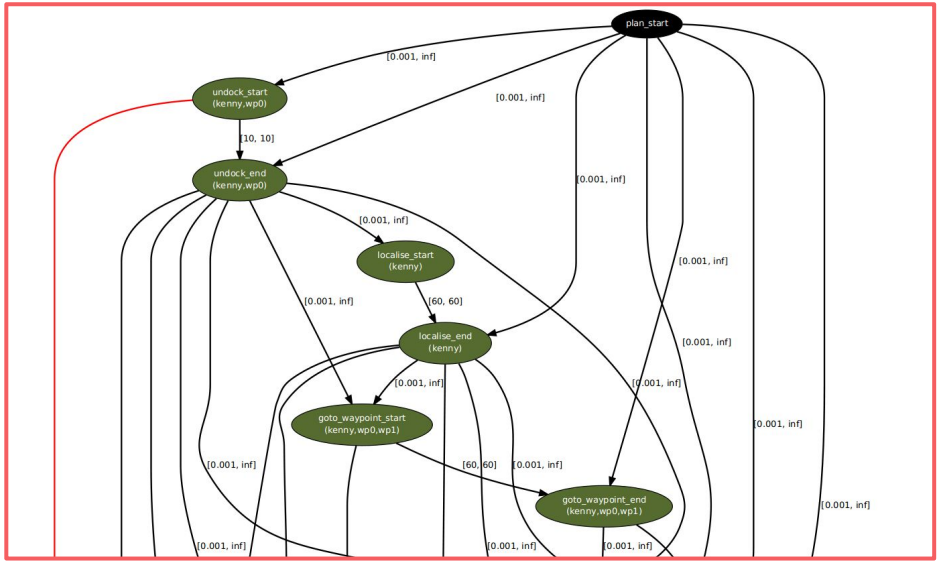
```
<!-- plan parsing -->
<include file="$(find rosplan_planning_system)/launch/includes/parsing_interface.launch">
    <arg name="knowledge_base"   value="rosplan_knowledge_base" />
    <arg name="planner_topic"    value="/rosplan_planner_interface/planner_output" />
    <arg name="plan_topic"       value="complete_plan" />
</include>

<!-- plan dispatch -->
<include file="$(find rosplan_planning_system)/launch/includes/dispatch_interface.launch">
    <arg name="knowledge_base"           value="rosplan_knowledge_base" />
    <arg name="plan_topic"               value="/rosplan_parsing_interface/complete_plan" />
    <arg name="action_dispatch_topic"    value="action_dispatch" />
    <arg name="action_feedback_topic"    value="action_feedback" />
    <arg name="display_edge_type"        value="true" />
</include>
```

Plan Execution Algorithms

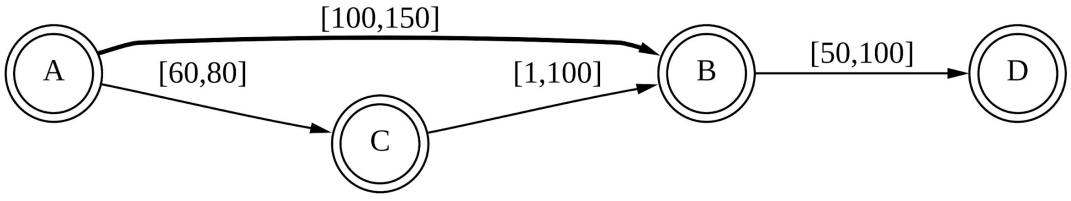
Simple Plan Dispatcher

0.000: (undock kenny wp0) [10.000]
10.001: (localise kenny) [60.000]
70.002: (goto_waypoint kenny wp0 wp0) [60.000]
130.003: (goto_waypoint kenny wp0 wp1) [60.000]
190.004: (goto_waypoint kenny wp1 wp0) [60.000]
250.005: (goto_waypoint kenny wp0 wp2) [60.000]
310.006: (goto_waypoint kenny wp2 wp0) [60.000]
370.007: (goto_waypoint kenny wp0 wp3) [60.000]
430.008: (goto_waypoint kenny wp3 wp0) [60.000]
490.009: (goto_waypoint kenny wp0 wp4) [60.000]
550.010: (goto_waypoint kenny wp4 wp0) [60.000]
610.011: (dock kenny wp0) [30.000]

Esterel Plan Dispatcher



STNU Dispatch



1. A = Set of all control points; current_time = 0
2. Arbitrarily pick a control point TP in A that is live and enabled in all boundary projections.
3. Set TP execution time to current_time, and remove TP from A (propagate constraints). Halt if A is now empty.
4. Advance current time, propagating all updates until some point in A is live and enabled.
5. Go to 2.

[Morris and Muscettola 1999, 2000]

```
<!-- plan dispatching -->
<node name="$(arg node_name)" pkg="rosplan_planning_system" type="pddl_esterel_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="$(arg knowledge_base)" />
    <param name="plan_topic"            value="$(arg plan_topic)" />
    <param name="action_dispatch_topic" value="$(arg action_dispatch_topic)" />
    <param name="action_feedback_topic" value="$(arg action_feedback_topic)" />
    <param name="display_edge_type"     value="$(arg display_edge_type)" />
    <param name="timeout_actions"         value="$(arg timeout_actions)" />
    <param name="action_timeout_fraction" value="$(arg action_timeout_fraction)" />
</node>
```

```
<!-- plan dispatching -->
<node name="$(arg node_name)" pkg="rosplan_planning_system" type="pddl_esterel_plan_dispatcher" respawn="false" output="screen">
    <param name="knowledge_base"        value="$(arg knowledge_base)" />
    <param name="plan_topic"            value="$(arg plan_topic)" />
    <param name="action_dispatch_topic" value="$(arg action_dispatch_topic)" />
    <param name="action_feedback_topic" value="$(arg action_feedback_topic)" />
    <param name="timeout_actions"         value="$(arg timeout_actions)" />
    <param name="action_timeout_fraction" value="$(arg action_timeout_fraction)" />
</node>
```

Not all problems have deadlines:

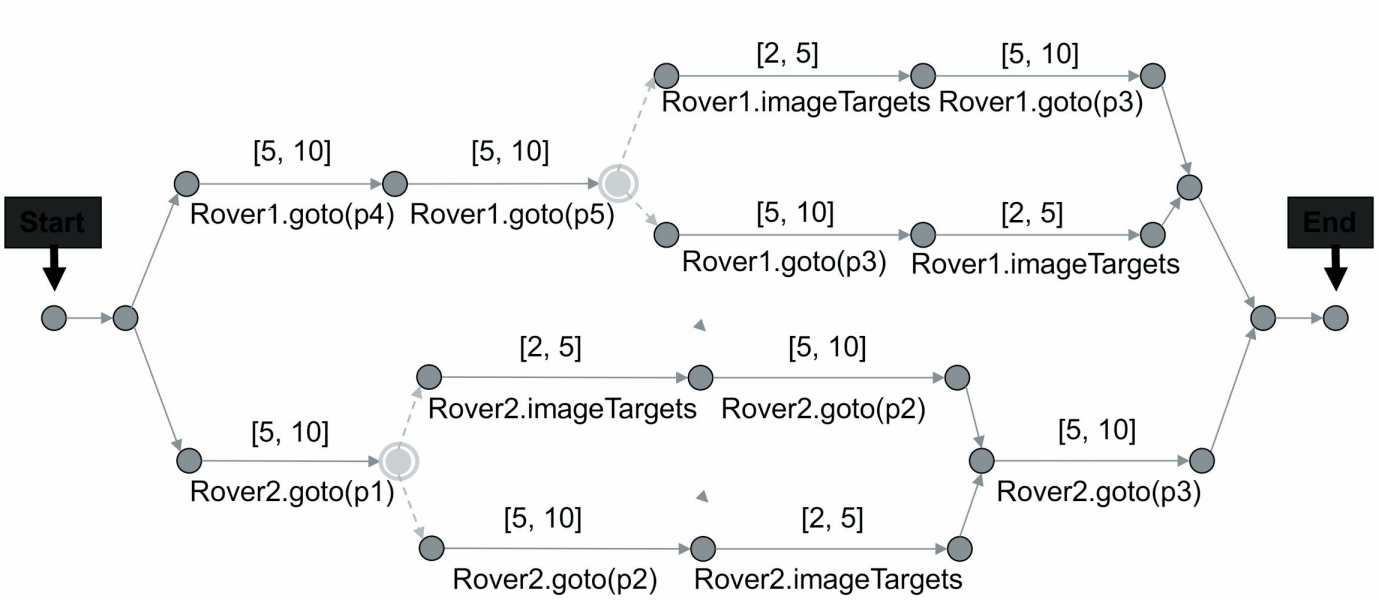
Timeout actions: when action durations are elapsed, fail the plan.

Action timeout fraction *x*: time out an action after

*action_timeout_duration*duration*

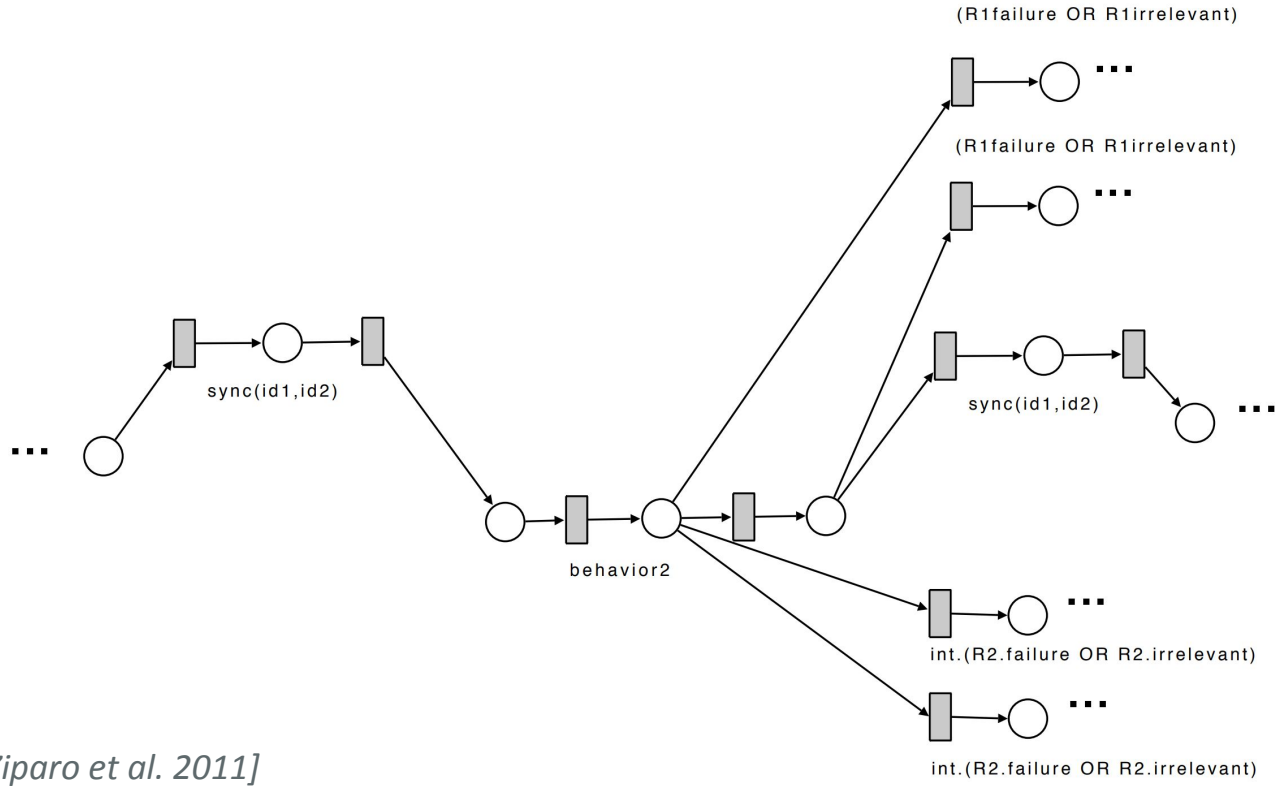
seconds.

Reactive Model-based Programming Language



[Ingham et al.2001]

Petri-Net-Plans

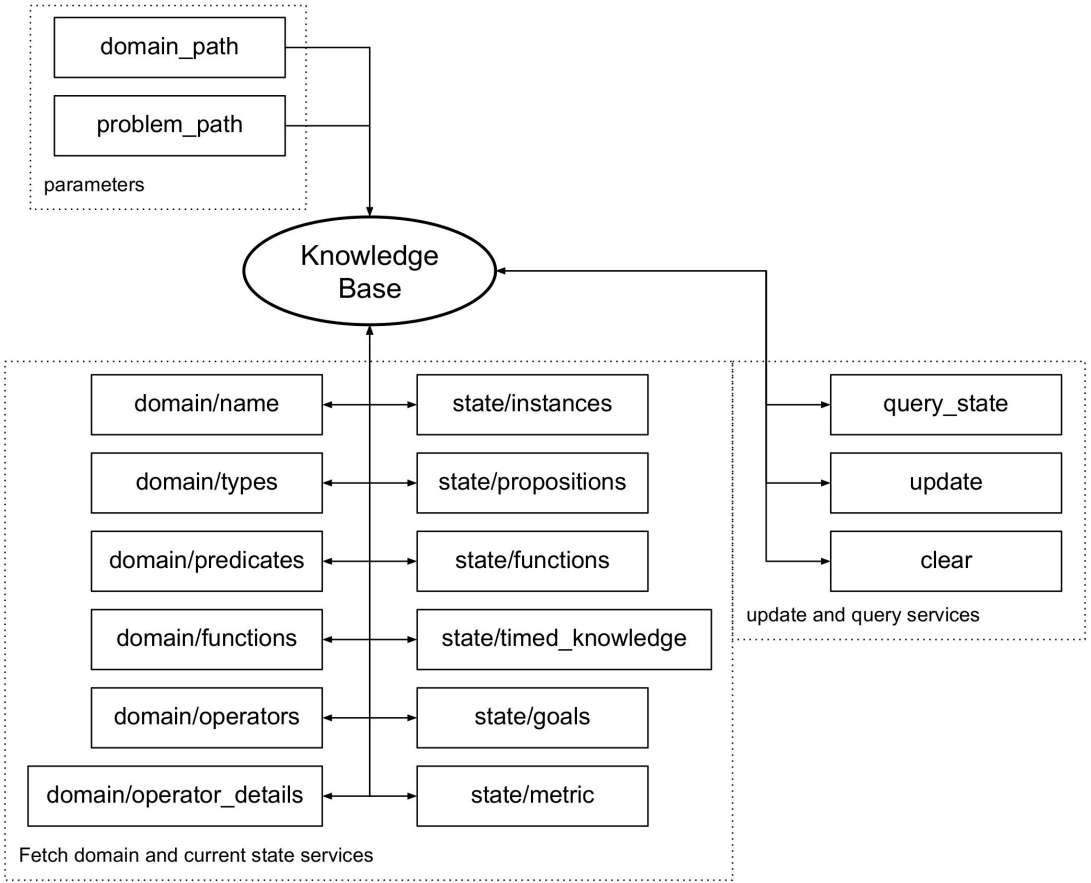


[Ziparo et al. 2011]

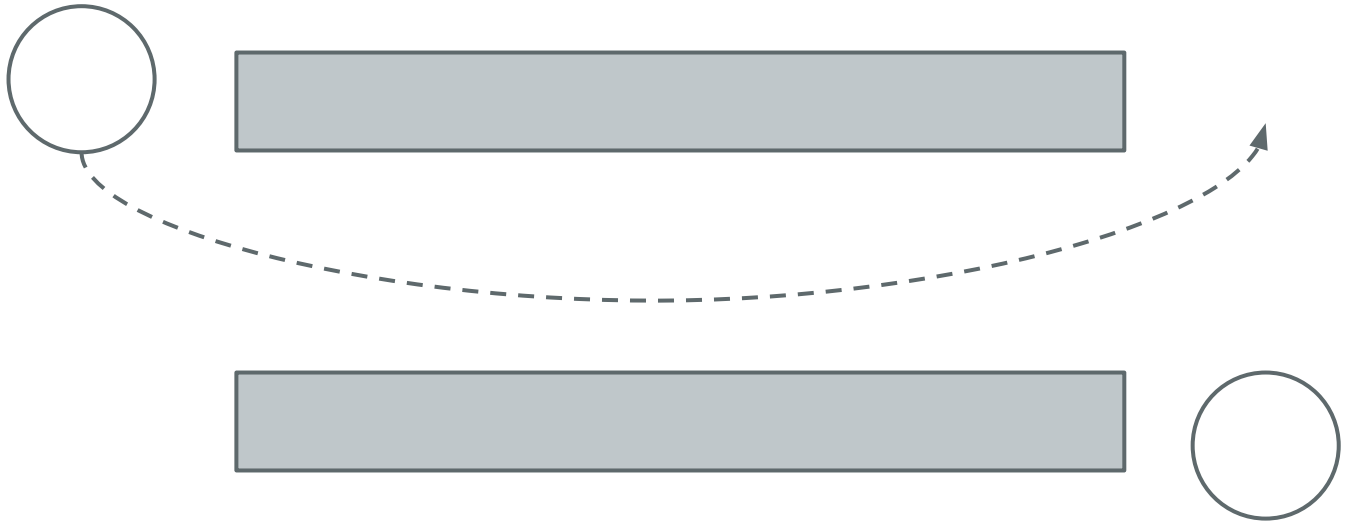
Knowledge Base

Knowledge Base

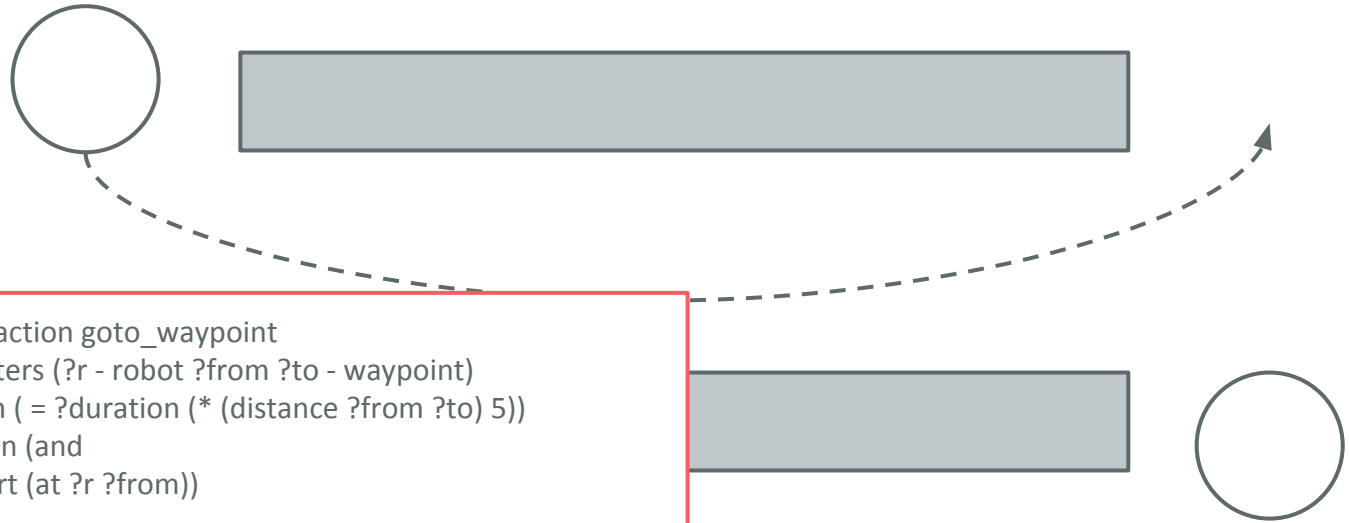
- Services for fetching **domain** and **state**.
- Services for updating current state.
- Service for query.



A Robot Plan

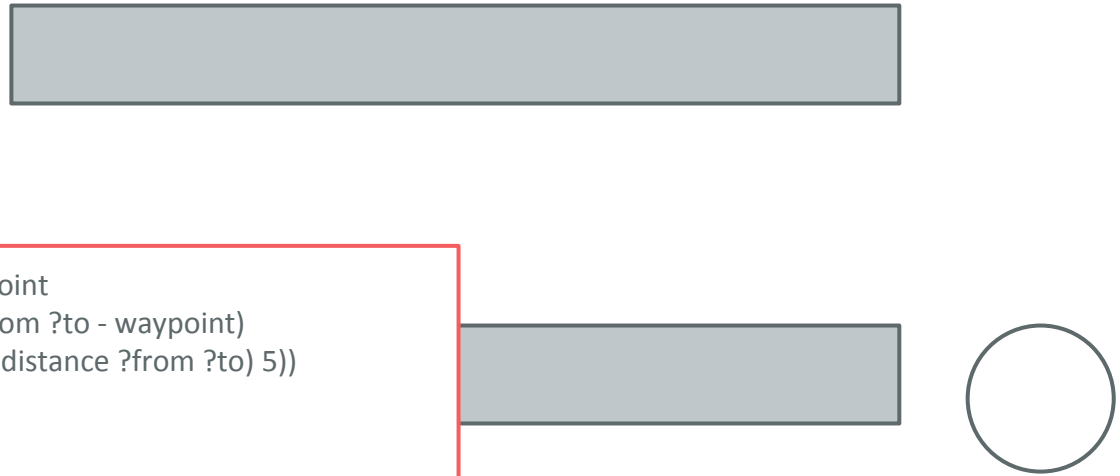


A Robot Plan



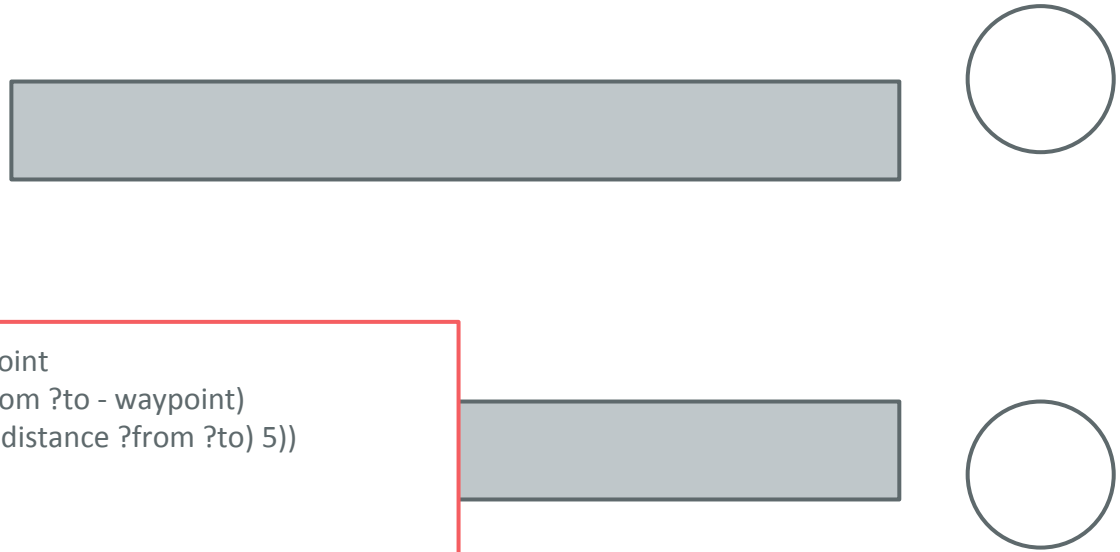
```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan



```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan



```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan

- Where was the robot during the action?
- When can the other robot move between the shelves?
- What is the state if the plan fails (for replanning)?

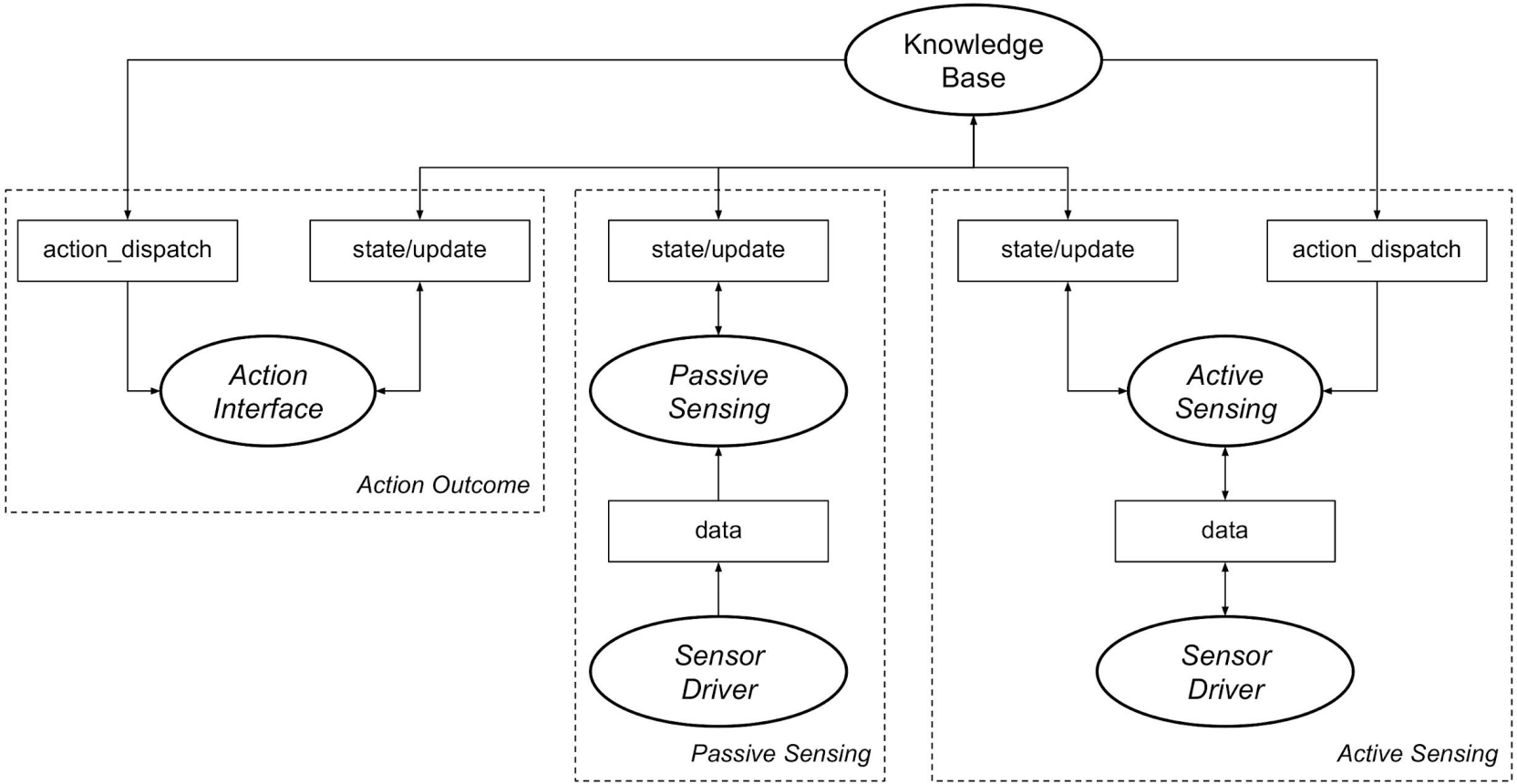
```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan

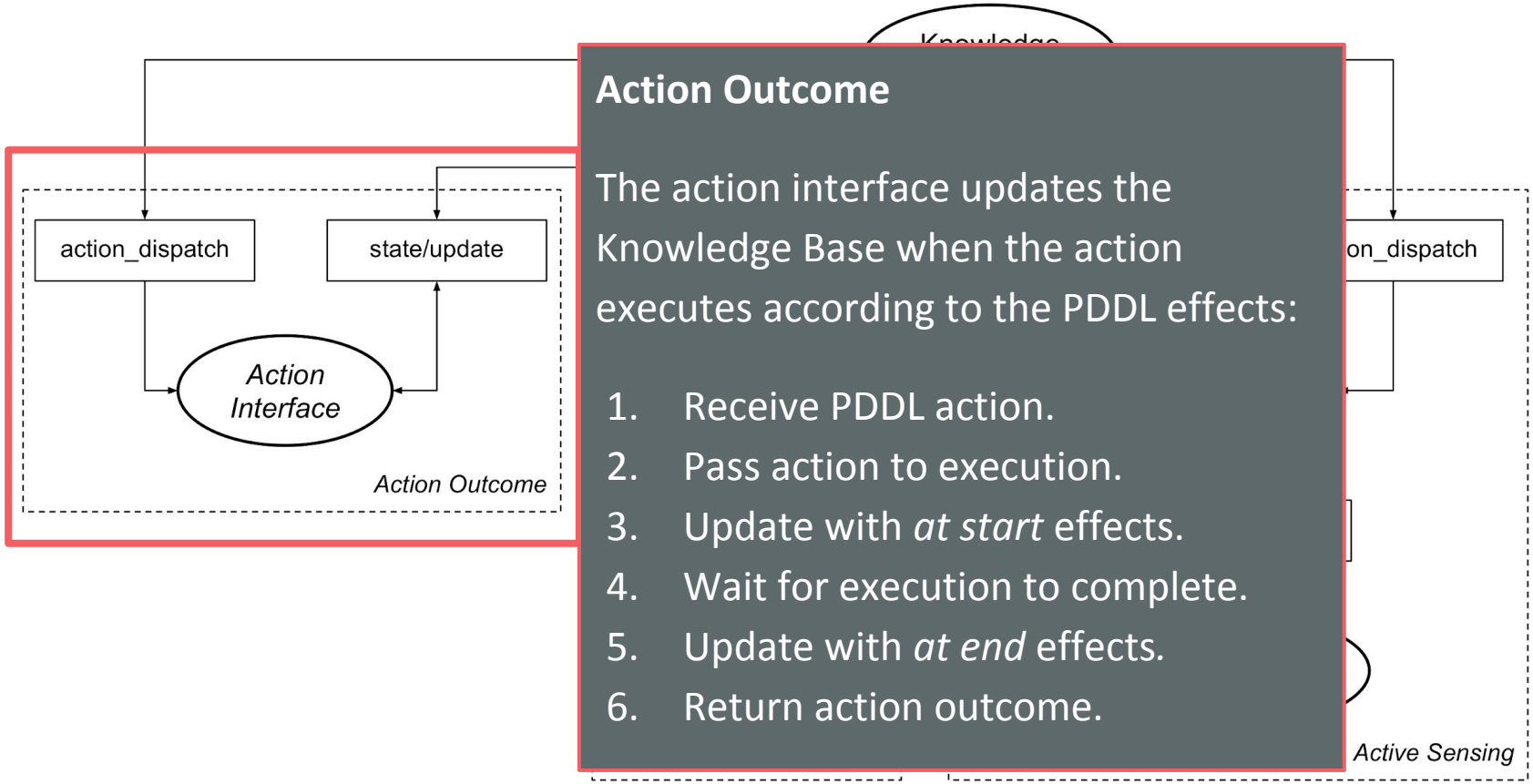
- **Where was the robot during the action?**
- **When can the other robot move between the shelves?**
- What is the state if the plan fails (for replanning)?

```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

Updating the Knowledge Base



Updating the Knowledge Base

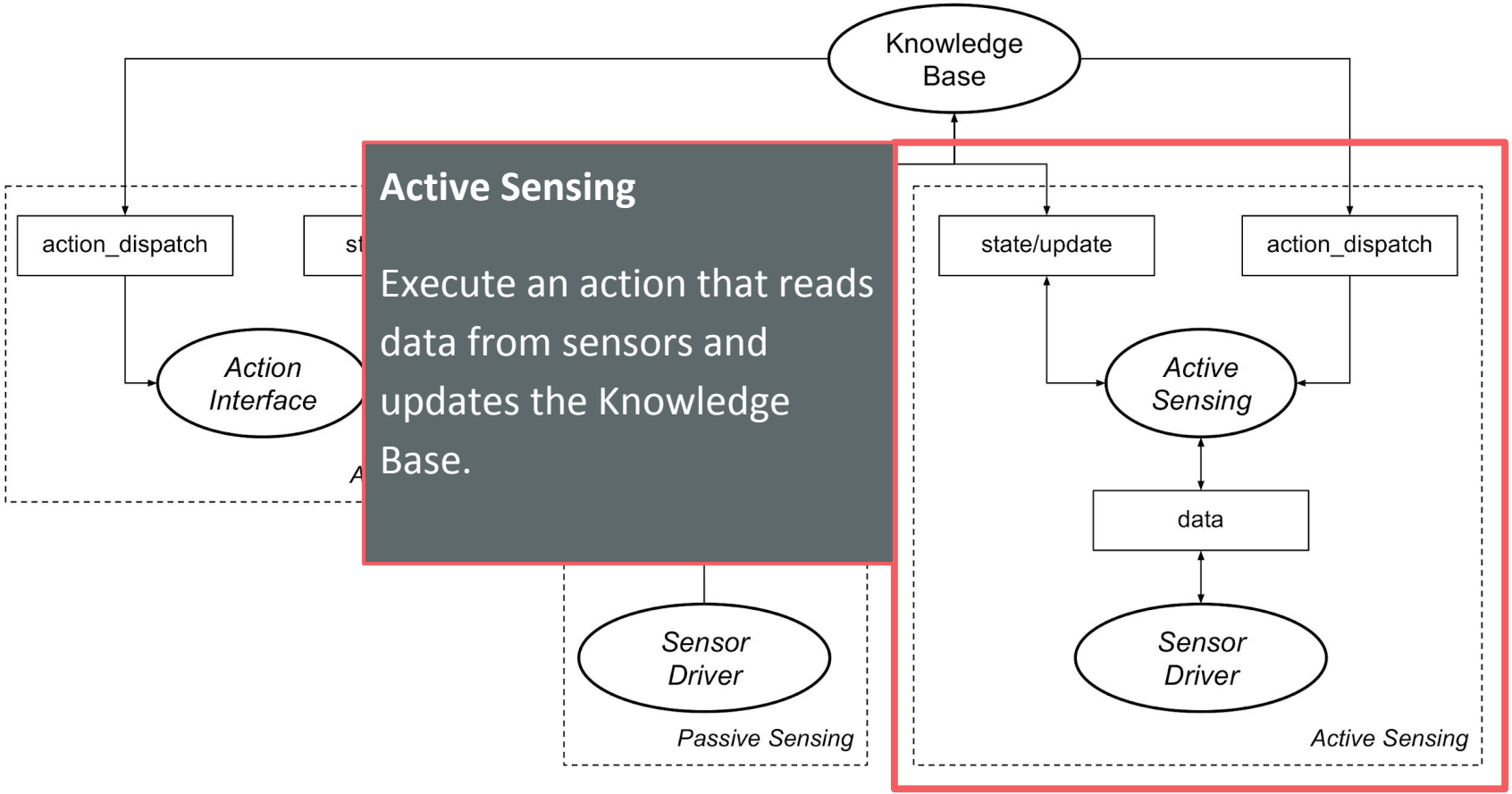


Action Outcome

The action interface updates the Knowledge Base when the action executes according to the PDDL effects:

1. Receive PDDL action.
2. Pass action to execution.
3. Update with *at start* effects.
4. Wait for execution to complete.
5. Update with *at end* effects.
6. Return action outcome.

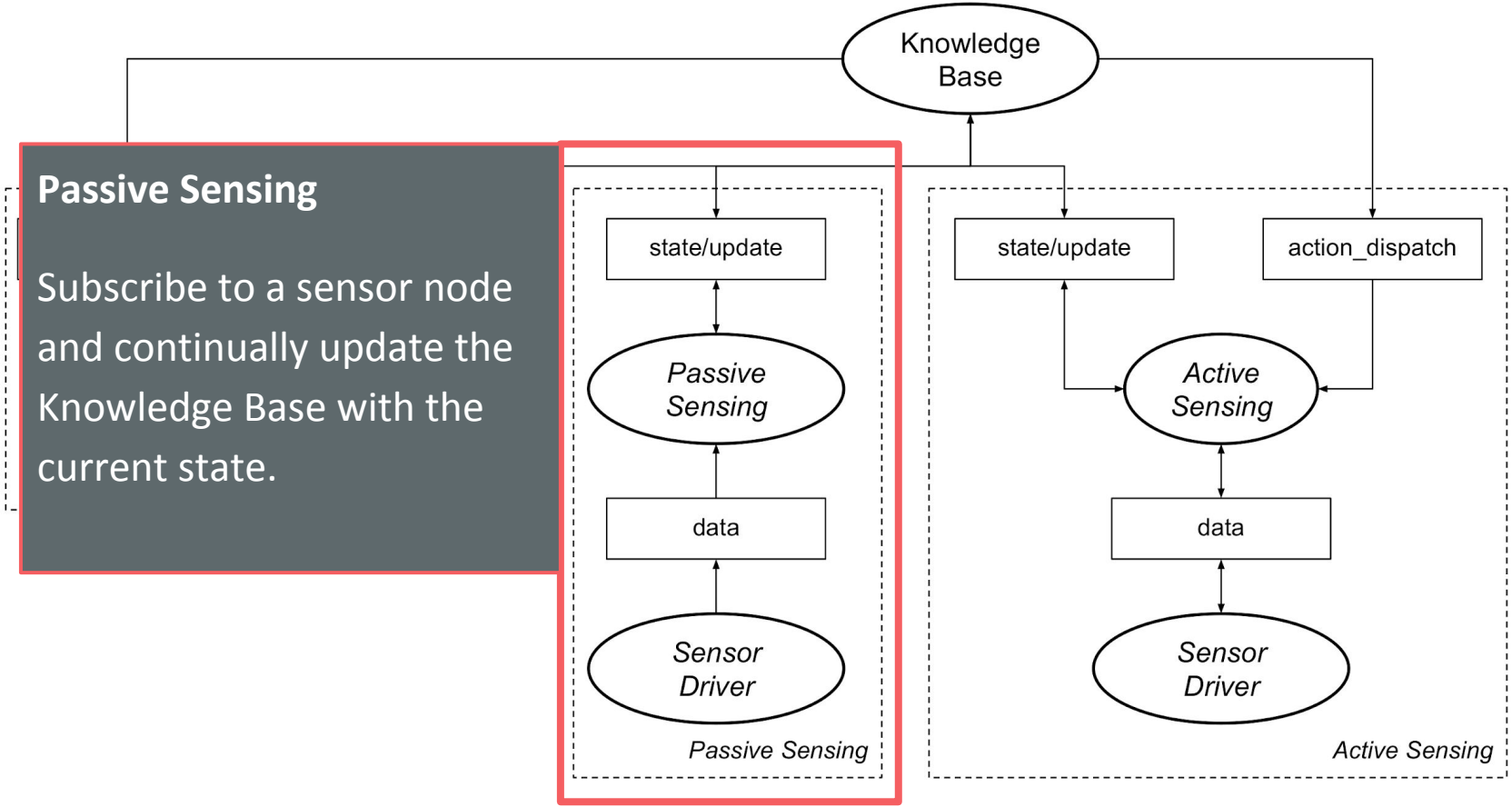
Updating the Knowledge Base



Updating the Knowledge Base

Passive Sensing

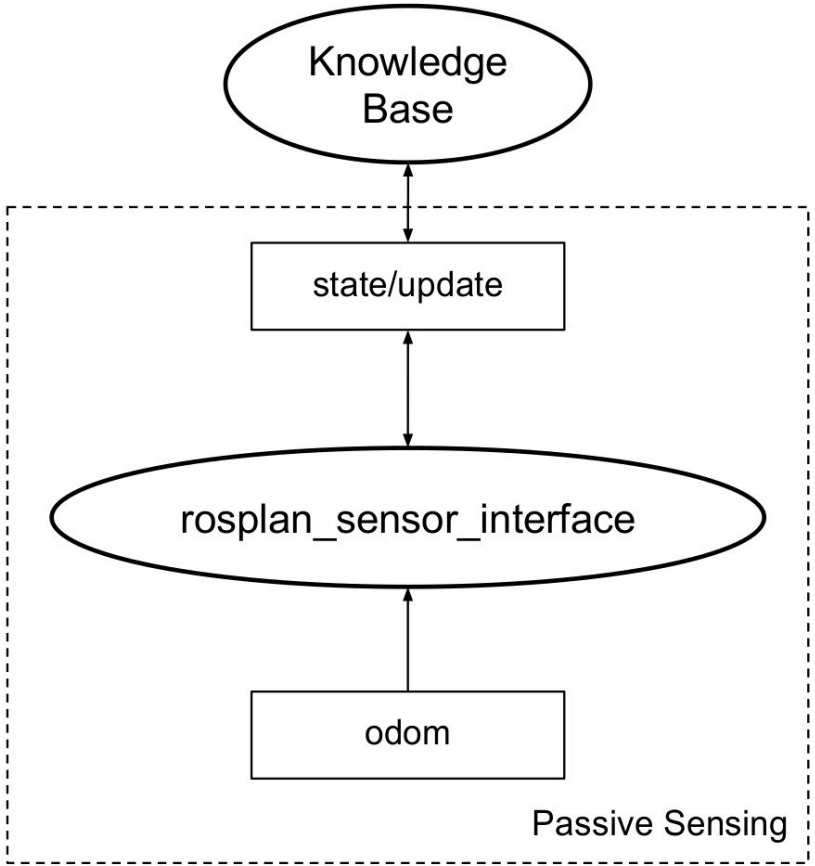
Subscribe to a sensor node and continually update the Knowledge Base with the current state.



Updating the Knowledge Base

The sensor interface simplifies *passive sensing*.

- A single node.
- Connects topics and services to PDDL predicates.



Warning: Task Planning and Robotics

An example planning model and system:

- Planning Model
  - Implicitly Models distance constraints in the **preconditions of activities**.
  - Models positions as waypoints.
- Action Interface
  - Post goal to path planner with coordinates and orientation matching the waypoint.
- Action Implementation
  - E.g. report success when “distance < 30cm”
- Sensing interface
  - E.g. set “at_waypoint” to true when “distance < 5cm”

Without care, the meaning of “at_waypoint” might vary.

```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path"     default="$(find rosplan_demos)/common/domain_turtlebot.pddl" />
    <arg name="problem_path"    default="$(find rosplan_demos)/common/problem_turtlebot.pddl" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path"  value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>

    <arg name="main_rate"   default="10"/>

    <node name="rosplan_sensing_interface" pkg="rosplan_sensing_interface" type="sensing_interface.py" respawn="false" output="screen">
        <rosparam command="load" file="$(find rosplan_sensing_interface)/icaps2019_tutorial.yaml" />
        <param name="main_rate"  value="10" />
    </node>

</launch>
```

```
functions:
    - $(find rosplan_sensing_interface)/icaps2019_tutorial.py
topics:

    localised:
        params:
            - kenny
        topic: /localised_mock
        msg_type: std_msgs/Bool
        operation: msg.data

    docked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger != msg.DISCHARGING

    undocked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger == msg.DISCHARGING

    robot_at:
        params:
            - kenny
            - '*'
        topic: /amcl_pose
        msg_type: geometry_msgs/PoseWithCovarianceStamped
```

```
functions:
    - $(find rosplan_sensing_interface)/icaps2019_tutorial.py
topics:
    localised:
        params:
            - kenny
        topic: /localised_mock
        msg_type: std_msgs/Bool
        operation: msg.data

    docked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger != msg.DISCHARGING

    undocked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger == msg.DISCHARGING

    robot_at:
        params:
            - kenny
            - '*'
        topic: /amcl_pose
        msg_type: geometry_msgs/PoseWithCovarianceStamped
```

```
functions:
    - $(find rosplan_sensing_interface)/icaps2019_tutorial.py
topics:

    localised:
        params:
            - kenny
        topic: /localised_mock
        msg_type: std_msgs/Bool
        operation: msg.data

    docked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger != msg.DISCHARGING

    undocked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger == msg.DISCHARGING

    robot_at:
        params:
            - kenny
            - '*'
        topic: /amcl_pose
        msg_type: geometry_msgs/PoseWithCovarianceStamped
```

```
functions:
    - $(find rosplan_sensing_interface)/icaps2019_tutorial.py
topics:

    localised:
        params:
            - kenny
        topic: /localised_mock
        msg_type: std_msgs/Bool
        operation: msg.data

    docked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger != msg.DISCHARGING

    undocked:
        params:
            - kenny
        topic: /mobile_base/sensors/core
        msg_type: kobuki_msgs/SensorState
        operation: msg.charger == msg.DISCHARGING

    robot_at:
        params:
            - kenny
            - '*'
        topic: /amcl_pose
        msg_type: geometry_msgs/PoseWithCovarianceStamped
```

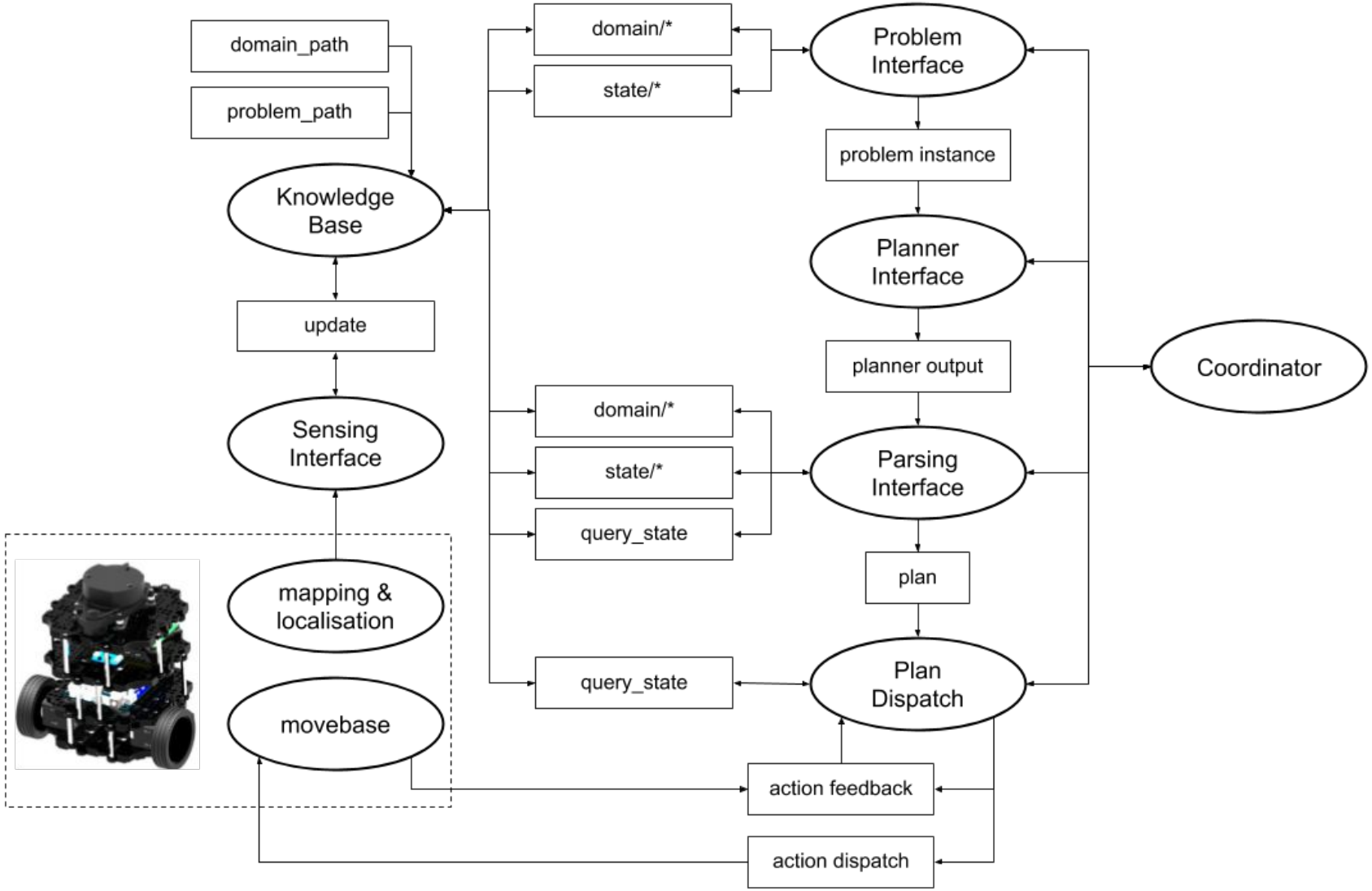
```python
#!/usr/bin/env python
from geometry_msgs.msg import PoseStamped
from math import sqrt

def robot_at(msg, params):

    assert(msg.header.frame_id == "map")
    assert(len(params) == 2)
    ret_value = []
    attributes = get_kb_attribute("robot_at")
    curr_wp = ''

    # Find current robot_location in knowledge base
    for a in attributes:
        if not a.is_negative:
            curr_wp = a.values[1].value
            break

    # for each robot parameter
    for robot in params[0]:

        # find closest waypoint
        distance = float('inf')
        closest_wp = ''
        for wp in params[1]:
            pose = rospy.get_param("/rosplan_demo_waypoints/"+wp)
            assert(len(pose) > 0)
            x = pose[0] - msg.pose.pose.position.x
            y = pose[1] - msg.pose.pose.position.y
            d = sqrt(x**2 + y**2)
            if d < distance:
                closest_wp = wp
                distance = d

        # set state in knowledge base
        if curr_wp != closest_wp:
            ret_value.append((robot + ':' + curr_wp, False)) # Set current waypoint to false
            ret_value.append((robot + ':' + closest_wp, True))  # Set new wp to true

    return ret_value
```

```python
#!/usr/bin/env python
from geometry_msgs.msg import PoseStamped
from math import sqrt

def robot_at(msg, params):

    assert(msg.header.frame_id == "map")
    assert(len(params) == 2)
    ret_value = []
    attributes = get_kb_attribute("robot_at")
    curr_wp = ''

    # Find current robot_location in knowledge base
    for a in attributes:
        if not a.is_negative:
            curr_wp = a.values[1].value
            break

    # for each robot parameter
    for robot in params[0]:

        # find closest waypoint
        distance = float('inf')
        closest_wp = ''
        for wp in params[1]:
            pose = rospy.get_param("/rosplan_demo_waypoints/"+wp)
            assert(len(pose) > 0)
            x = pose[0] - msg.pose.pose.position.x
            y = pose[1] - msg.pose.pose.position.y
            d = sqrt(x**2 + y**2)
            if d < distance:
                closest_wp = wp
                distance = d

        # set state in knowledge base
        if curr_wp != closest_wp:
            ret_value.append((robot + ':' + curr_wp, False)) # Set current waypoint to false
            ret_value.append((robot + ':' + closest_wp, True))  # Set new wp to true

    return ret_value
```

Example in Simulation

```
<!-- main coordinator -->
<node pkg="rosplan_stage_exploration_demo" type="main_executor.py" name="coordinator" respawn="false" required="true" output="screen">
    <param name="max_prm_size"    value="$(arg max_prm_size)" />
    <param name="planner_command" value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    <param name="domain_path"     value="$(arg domain_path)" />
    <param name="problem_path"    value="$(arg autom_gen_problem_path)" />
    <param name="data_path"       value="$(arg data_path)" />
</node>
```

```
<!-- main coordinator -->
<node pkg="rosplan_stage_exploration_demo" type="main_executor.py" name="coordinator" respawn="false" required="true" output="screen">
    <param name="max_prm_size"    value="$(arg max_prm_size)" />
    <param name="planner_command" value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    <param name="domain_path"     value="$(arg domain_path)" />
    <param name="problem_path"    value="$(arg autom_gen_problem_path)" />
    <param name="data_path"       value="$(arg data_path)" />
</node>
```


domain_path
problem_path
Knowledge Base
update
Sensing Interface
mapping & localisation
movebase
domain/*
state/*
domain/*
state/*
query_state
query_state
Problem Interface
problem instance
Planner Interface
planner output
Parsing Interface
plan
Plan Dispatch
Coordinator
action feedback
action dispatch

Part 3:
Strategic-Tactical
(The Problem)

Strategic-Tactical Decomposition

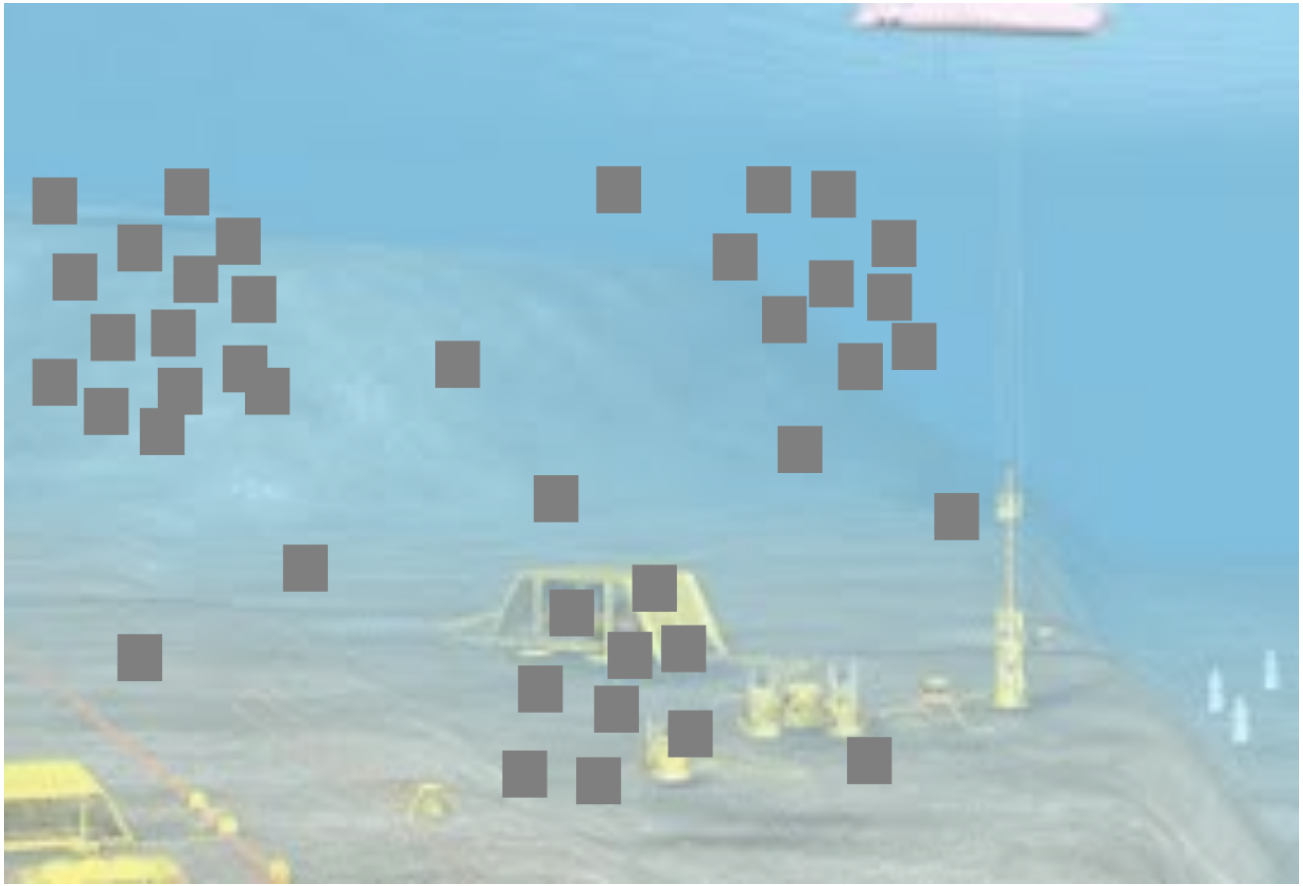
Our approach is based on hierarchical abstraction.

A plan is generated using the following steps:

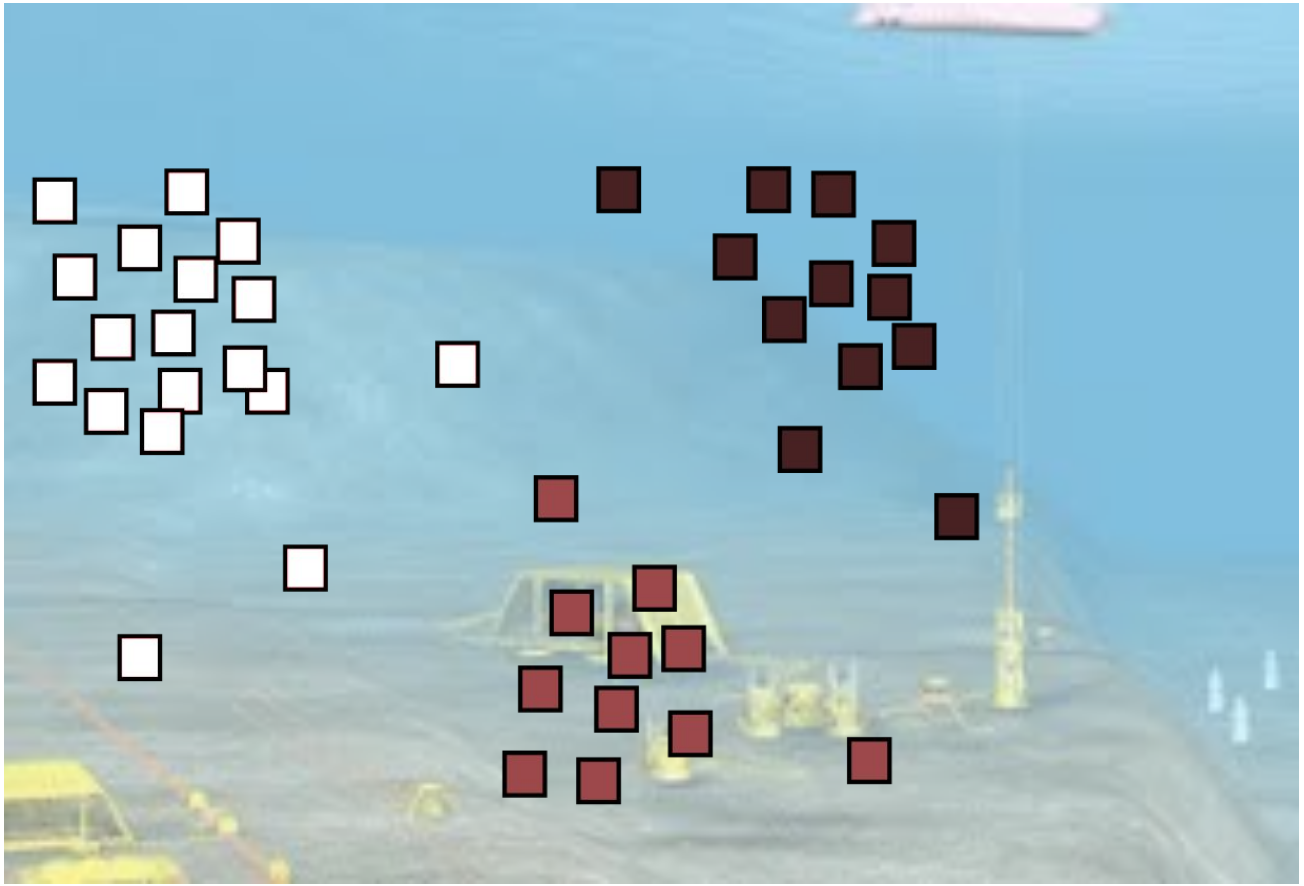
1. **Automatically decompose a large problem into a set of smaller problems (subgoals).**
2. Generate a tactical plan for each subgoal.
3. Construct a strategic problem, containing strategic actions.
4. Generate a strategic plan for the whole mission.

Buksz et al. Strategic-Tactical Planning for Autonomous Underwater Vehicles over Long Horizons. IROS 2018

Strategic-Tactical Decomposition



Strategic-Tactical Decomposition



Strategic-Tactical Decomposition

Our approach is based on hierarchical abstraction.

A plan is generated using the following steps:

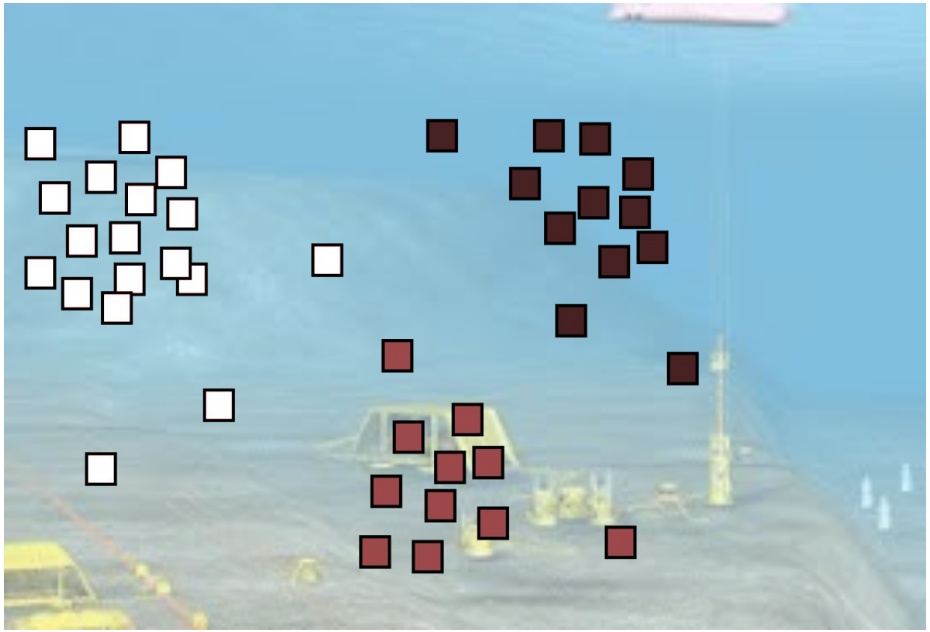
1. Automatically decompose a large problem into a set of smaller problems (subgoals).
2. **Generate a tactical plan for each subgoal.**
3. Construct a strategic problem, containing strategic actions.
4. Generate a strategic plan for the whole mission.

Tactical Planning

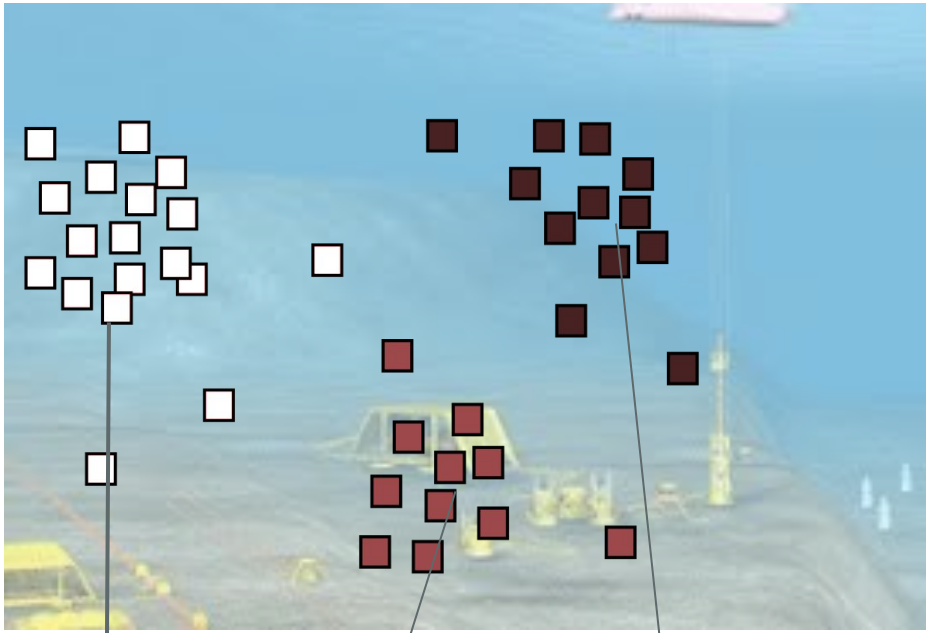
A tactical plan is generated for each subgoal.

- The temporal constraints are ignored.
- The planner is given a maximum of 10 seconds to plan.
- The plans are not executed.
- The duration and energy cost of the tactical plan is saved.

Tactical Planning



Tactical Planning

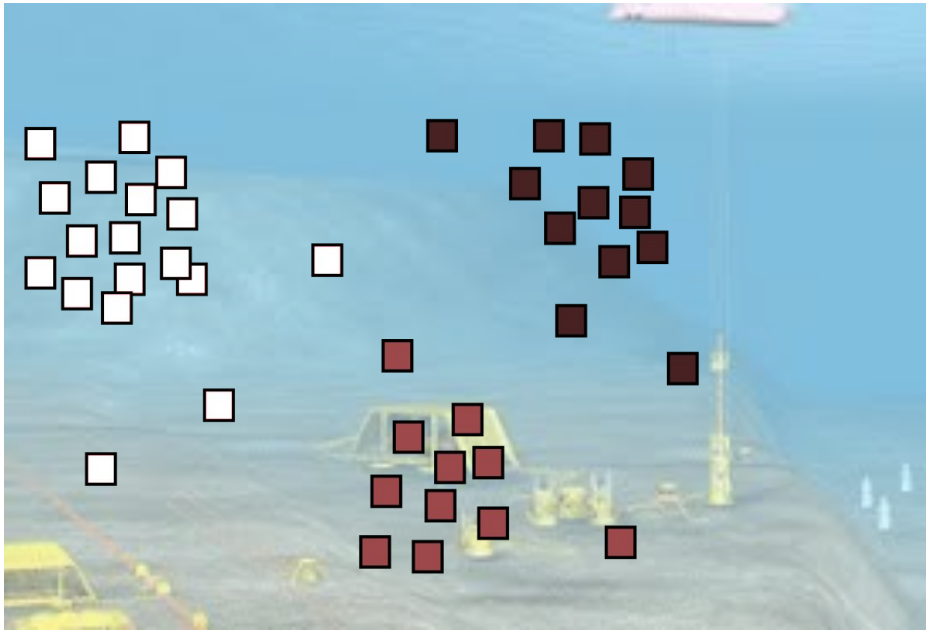


Tactical Plan

Tactical Plan

Tactical Plan

Tactical Planning



Tactical Plan

0 949.001

Tactical Plan Tactical Plan

```
33.371: (do_hover_controlled auv wp1 wp2) [14.340] ; (41)
47.712: (do_hover_controlled auv wp2 wp23) [3.342] ; (58)
51.055: (observe_inspection_point auv wp23 ip1) [10.000] ; (296)
61.056: (correct_position auv wp23) [3.000] ; (271)
64.057: (observe_inspection_point auv wp23 ip4) [10.000] ; (315)
74.058: (correct_position auv wp23) [3.000] ; (271)
77.059: (observe_inspection_point auv wp23 ip5) [10.000] ; (322)
87.060: (correct_position auv wp23) [3.000] ; (271)
90.061: (do_hover_controlled auv wp23 wp17) [4.282] ; (33)
94.344: (observe_inspection_point auv wp17 ip3) [10.000] ; (308)
104.345: (correct_position auv wp17) [3.000] ; (264)
107.346: (do_hover_controlled auv wp17 wp27) [4.720] ; (67)
112.067: (observe_inspection_point auv wp27 ip2) [10.000] ; (305)
122.068: (correct_position auv wp27) [3.000] ; (275)
125.069: (observe_inspection_point auv wp27 ip6) [10.000] ; (329)
135.070: (correct_position auv wp27) [3.000] ; (275)
138.071: (observe_inspection_point auv wp27 ip8) [10.000] ; (341)
148.072: (correct_position auv wp27) [3.000] ; (275)
151.073: (do_hover_controlled auv wp27 wp17) [4.720] ; (34)
155.794: (do_hover_controlled auv wp17 wp6) [1.276] ; (115)
157.071: (observe_inspection_point auv wp6 ip7) [10.000] ; (336)
167.072: (correct_position auv wp6) [3.000] ; (291)
170.073: (do_hover_controlled auv wp6 wp2) [6.040] ; (44)
176.114: (do_hover_controlled auv wp2 wp36) [4.290] ; (95)
180.405: (observe_inspection_point auv wp36 ip9) [10.000] ; (348)
190.406: (correct_position auv wp36) [3.000] ; (285)
193.407: (do_hover_controlled auv wp36 wp2) [4.290] ; (43)
197.698: (do_hover_controlled auv wp2 wp1) [14.340] ; (2)
212.039: (observe_inspection_point auv wp1 ip3) [10.000] ; (307)
222.040: (correct_position auv wp1) [3.000] ; (256)
225.041: (do_hover_controlled auv wp1 wp16) [3.854] ; (27)
228.896: (observe_inspection_point auv wp16 ip2) [10.000] ; (302)
238.897: (correct_position auv wp16) [3.000] ; (263)
241.898: (do_hover_controlled auv wp16 wp21) [1.540] ; (47)
243.439: (observe_inspection_point auv wp21 ip2) [10.000] ; (304)
253.440: (correct_position auv wp21) [3.000] ; (269)
256.441: (observe_inspection_point auv wp21 ip3) [10.000] ; (310)
266.442: (correct_position auv wp21) [3.000] ; (269)
269.443: (do_hover_controlled auv wp21 wp27) [4.252] ; (68)
273.696: (do_hover_controlled auv wp27 wp12) [4.506] ; (14)
278.203: (observe_inspection_point auv wp12 ip2) [10.000] ; (301)
288.204: (correct_position auv wp12) [3.000] ; (259)
291.205: (do_hover_controlled auv wp12 wp27) [4.506] ; (66)
599.001: (turn_valve auv wp27 p1 v1) [120.000] ; (350)
```

Strategic-Tactical Decomposition

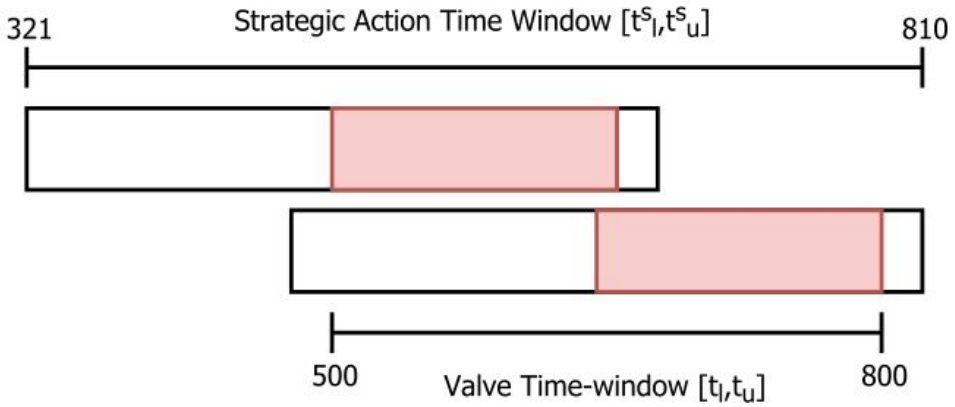
Our approach is based on hierarchical abstraction.

A plan is generated using the following steps:

1. Automatically decompose a large problem into a set of smaller problems (subgoals).
2. Generate a tactical plan for each subgoal.
3. **Construct a strategic problem, containing strategic actions.**
4. Generate a strategic plan for the whole mission.

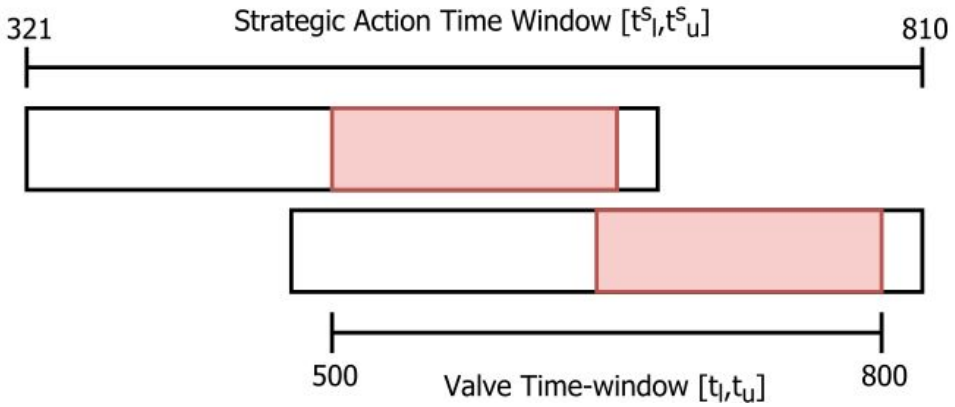
Strategic Problem Compilation

Upper and lower bounds are extracted from tactical plans to impose temporal constraints in the strategic problem.

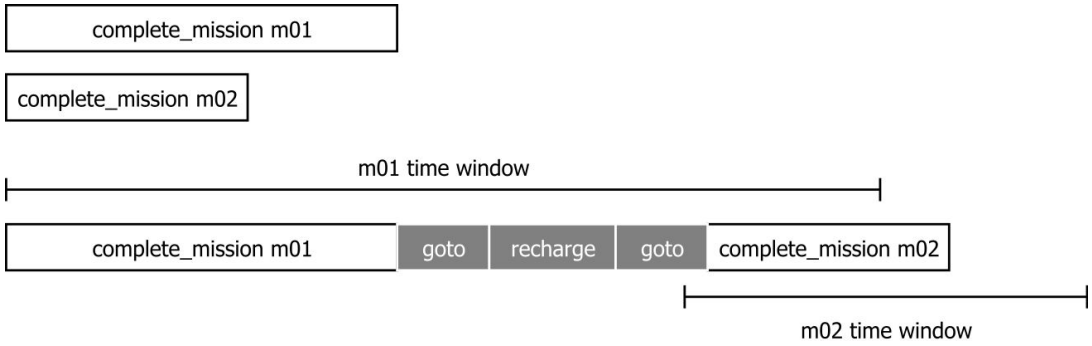


Strategic Problem Compilation

Upper and lower bounds are extracted from tactical plans to impose temporal constraints in the strategic problem.



The strategic plan schedules the subgoals within new constraints.



Strategic Problem Compilation

(active mission0)
(active mission1)
(active mission2)
(active mission3)
(active mission4)
(active mission5)
(active mission6)
(active mission7)
(active mission8)
(active mission9)

(in mission0 mission_site_start_point_1)
(in mission1 mission_site_start_point_1)
(in mission2 mission_site_start_point_1)
(in mission3 mission_site_start_point_1)
(in mission4 mission_site_start_point_1)
(in mission5 mission_site_start_point_1)
(in mission6 mission_site_start_point_1)
(in mission7 mission_site_start_point_1)
(in mission8 mission_site_start_point_1)
(in mission9 mission_site_start_point_1)

(= (mission_duration mission0) 261.867)
(= (mission_duration mission1) 242.066)
(= (mission_duration mission2) 336.17)
(= (mission_duration mission3) 365.49)
(= (mission_duration mission4) 340.969)
(= (mission_duration mission5) 337.334)
(= (mission_duration mission6) 608.425)
(= (mission_duration mission7) 800.794)
(= (mission_duration mission8) 692.803)
(= (mission_duration mission9) 629.094)

(at 4100 (not (active mission0)))
(at 7100 (not (active mission1)))
(at 99999 (not (active mission10)))

Strategic-Tactical Decomposition

Our approach is based on hierarchical abstraction.

A plan is generated using the following steps:

1. Automatically decompose a large problem into a set of smaller problems (subgoals).
2. Generate a tactical plan for each subgoal.
3. Construct a strategic problem, containing strategic actions.
4. **Generate a strategic plan for the whole mission.**

Strategic Plan Execution

0.000: (do_hover auv wp_auv0 mission_site_start_point_1) [112.805]
112.806: (complete_mission auv mission2 mission_site_start_point_1) [336.170]
448.977: (complete_mission auv mission1 mission_site_start_point_1) [242.065]
691.043: (do_hover auv mission_site_start_point_1 mission_site_start_point_0) [269.444]
960.488: (dock_auv auv mission_site_start_point_0) [20.000]
980.488: (recharge auv mission_site_start_point_0) [1800.000]
2780.489: (undock_auv auv mission_site_start_point_0) [10.000]
2790.489: (do_hover auv mission_site_start_point_0 mission_site_start_point_1) [269.444]
3059.934: (complete_mission auv mission5 mission_site_start_point_1) [337.334]
3397.269: (complete_mission auv mission0 mission_site_start_point_1) [261.867]
3659.137: (complete_mission auv mission7 mission_site_start_point_1) [337.334]

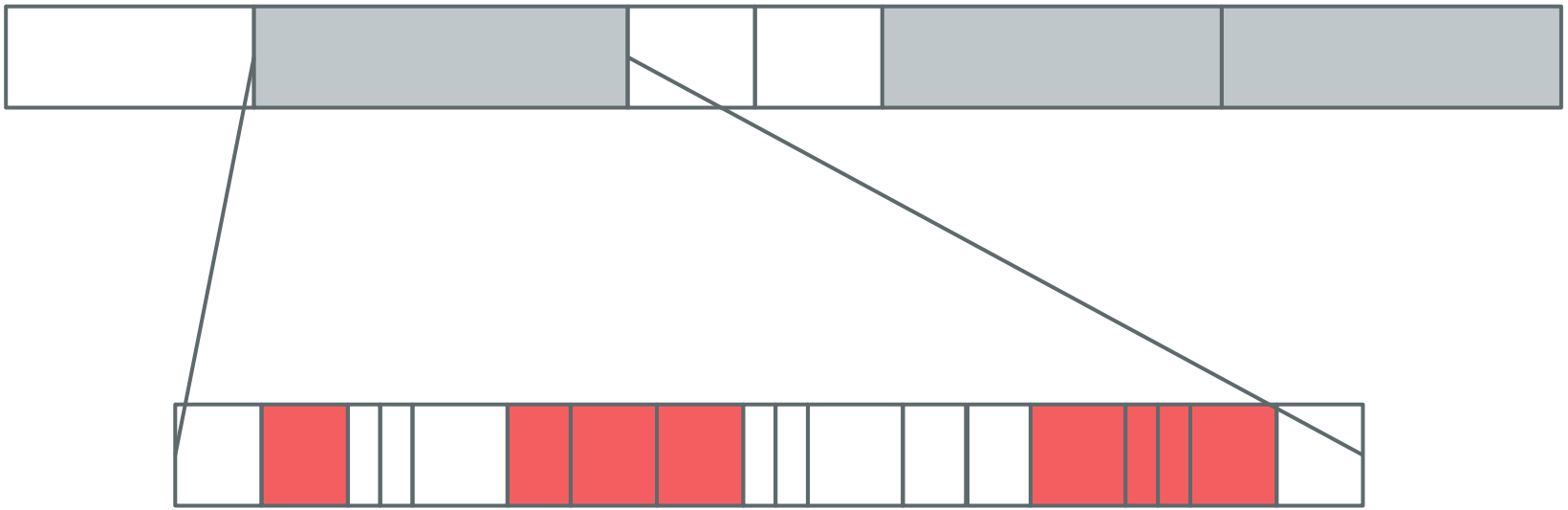
Strategic Plan Execution

0.000: (do_hover auv wp_auv0 mission_site_start_point_1) [112.805]
112.806: (complete_mission auv mission2 mission_site_start_point_1) [336.170]
448.977: (complete_mission auv mission1 mission_site_start_point_1) [242.065]
691.043: (do_hover auv mission_site_start_point_1 mission_site_start_point_0) [269.444]
960.488: (dock_auv auv mission_site_start_point_0) [20.000]
980.488: (recharge auv mission_site_start_point_0) [1800.000]
2780.489: (undock_auv auv mission_site_start_point_0) [10.000]
2790.489: (do_hover auv mission_site_start_point_0 mission_site_start_point_1) [269.444]
3059.934: (complete_mission auv mission5 mission_site_start_point_1) [337.334]
3397.269: (complete_mission auv mission0 mission_site_start_point_1) [261.867]
3659.137: (complete_mission auv mission7 mission_site_start_point_1) [337.334]

When the *complete_mission* action is dispatched, the tactical plan is executed.

Strategic Plan Execution

Strategic Plan Execution



Strategic Plan Execution

When the *complete_mission* action is dispatched, the tactical plan is executed.

The plans are executed:

1. The tactical problem is regenerated, including the temporal constraints.

Strategic Plan Execution

When the *complete_mission* action is dispatched, the tactical plan is executed.

The plans are executed:

1. The tactical problem is regenerated, including the temporal constraints.
2. A new tactical plan is generated.

Strategic Plan Execution

When the *complete_mission* action is dispatched, the tactical plan is executed.

The plans are executed:

1. The tactical problem is regenerated, including the temporal constraints.
2. A new tactical plan is generated.
3. If the tactical plan fails, then it is replanned.

Strategic Plan Execution

When the *complete_mission* action is dispatched, the tactical plan is executed.

The plans are executed:

1. The tactical problem is regenerated, including the temporal constraints.
2. A new tactical plan is generated.
3. If the tactical plan fails, then it is replanned.
4. The strategic plan execution monitors the execution of a tactical plan as any other action. The tactical plan can fail (action failure) or be cancelled (timeout).

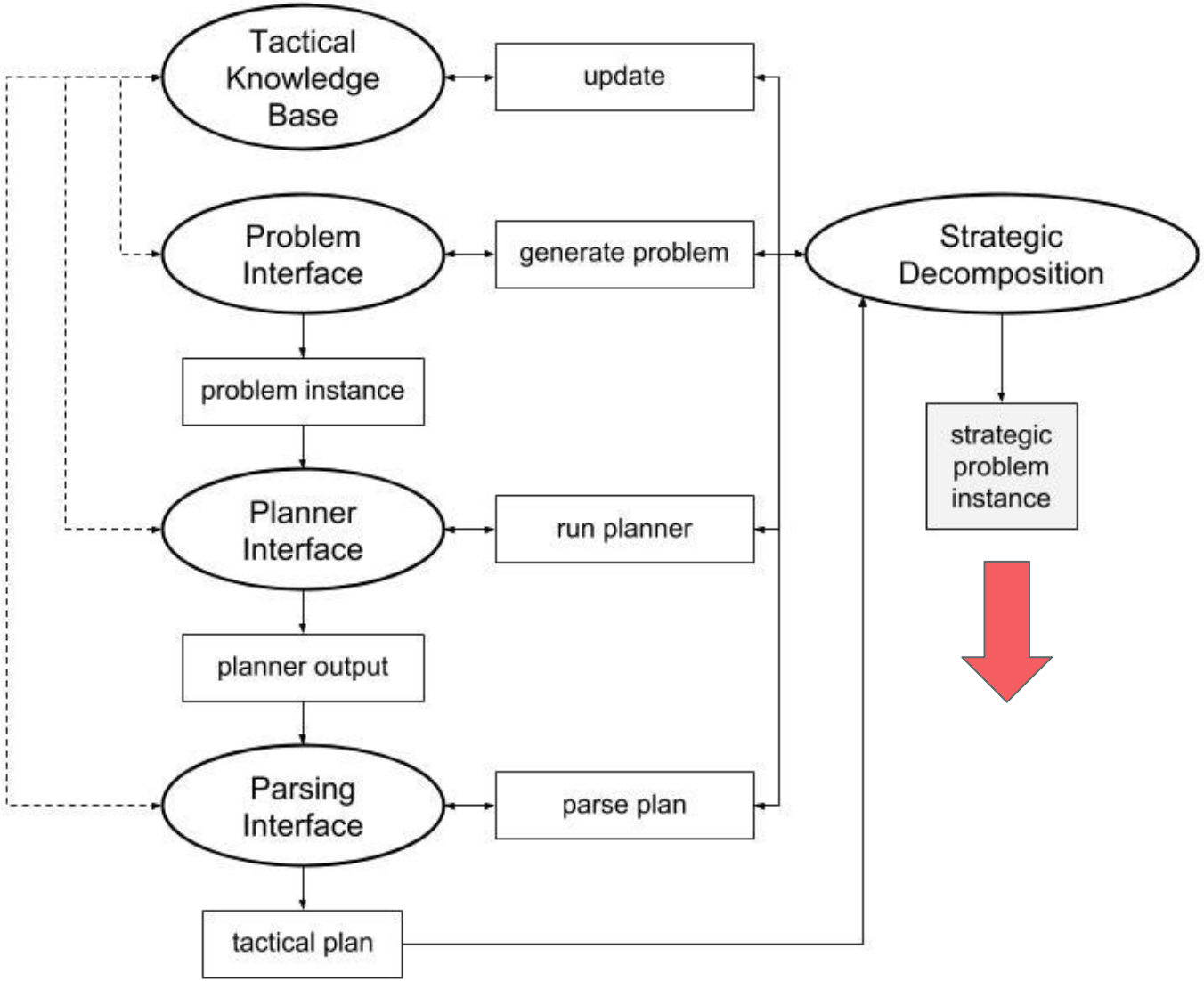
Strategic Plan Execution

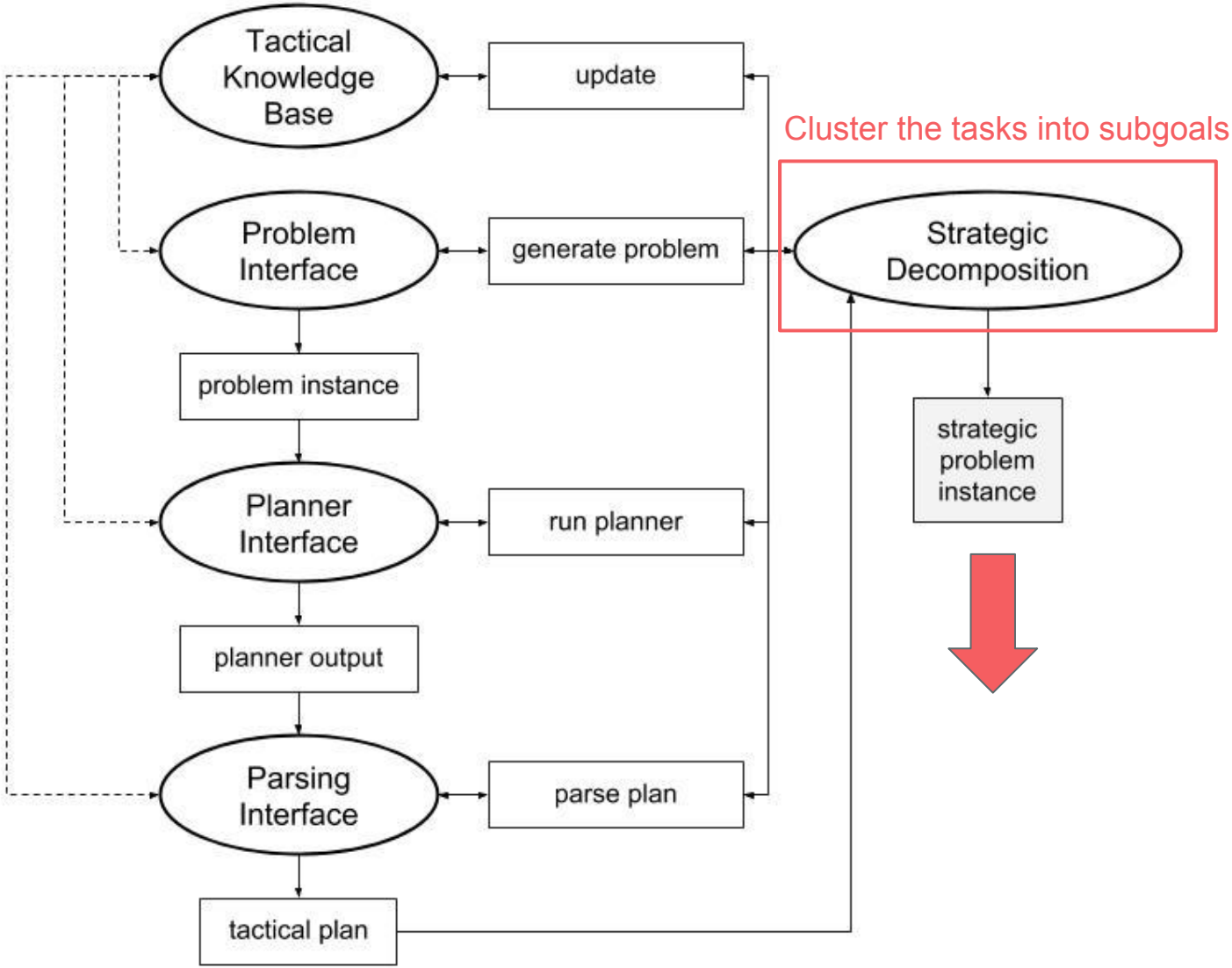
When the *complete_mission* action is dispatched, the tactical plan is executed.

The plans are executed:

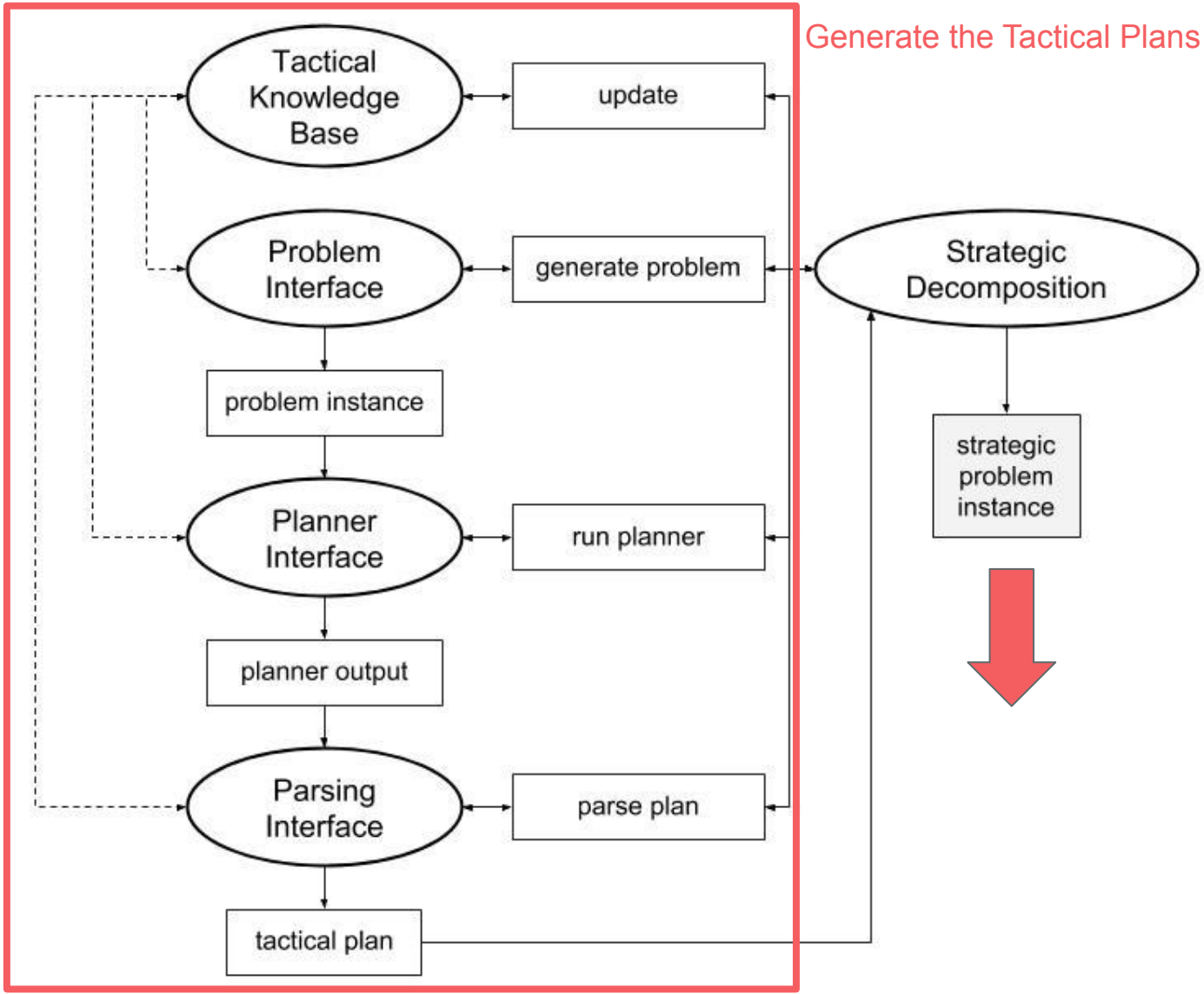
1. The tactical problem is regenerated, including the temporal constraints.
2. A new tactical plan is generated.
3. If the tactical plan fails, then it is replanned.
4. The strategic plan execution monitors the execution of a tactical plan as any other action. The tactical plan can fail (action failure) or be cancelled (timeout).
5. If the strategic plan fails, then replanning takes place at the strategic layer (without replanning the tactical problems).

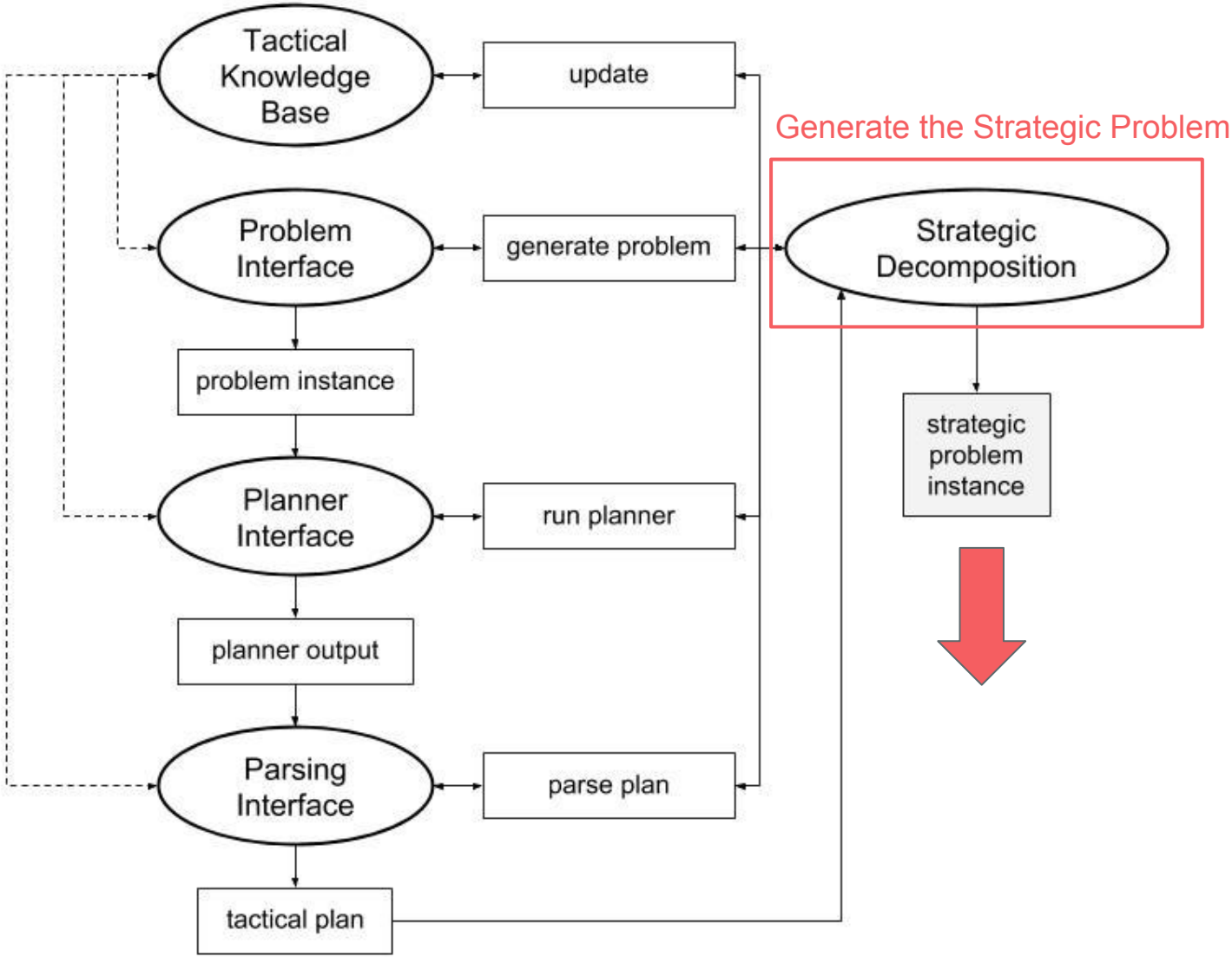
Part 4:
Strategic-Tactical
(ROSPlan setup)


Tactical Knowledge Base
update
Problem Interface
generate problem
Strategic Decomposition
problem instance
strategic problem instance
Planner Interface
run planner
planner output
Parsing Interface
parse plan
tactical plan

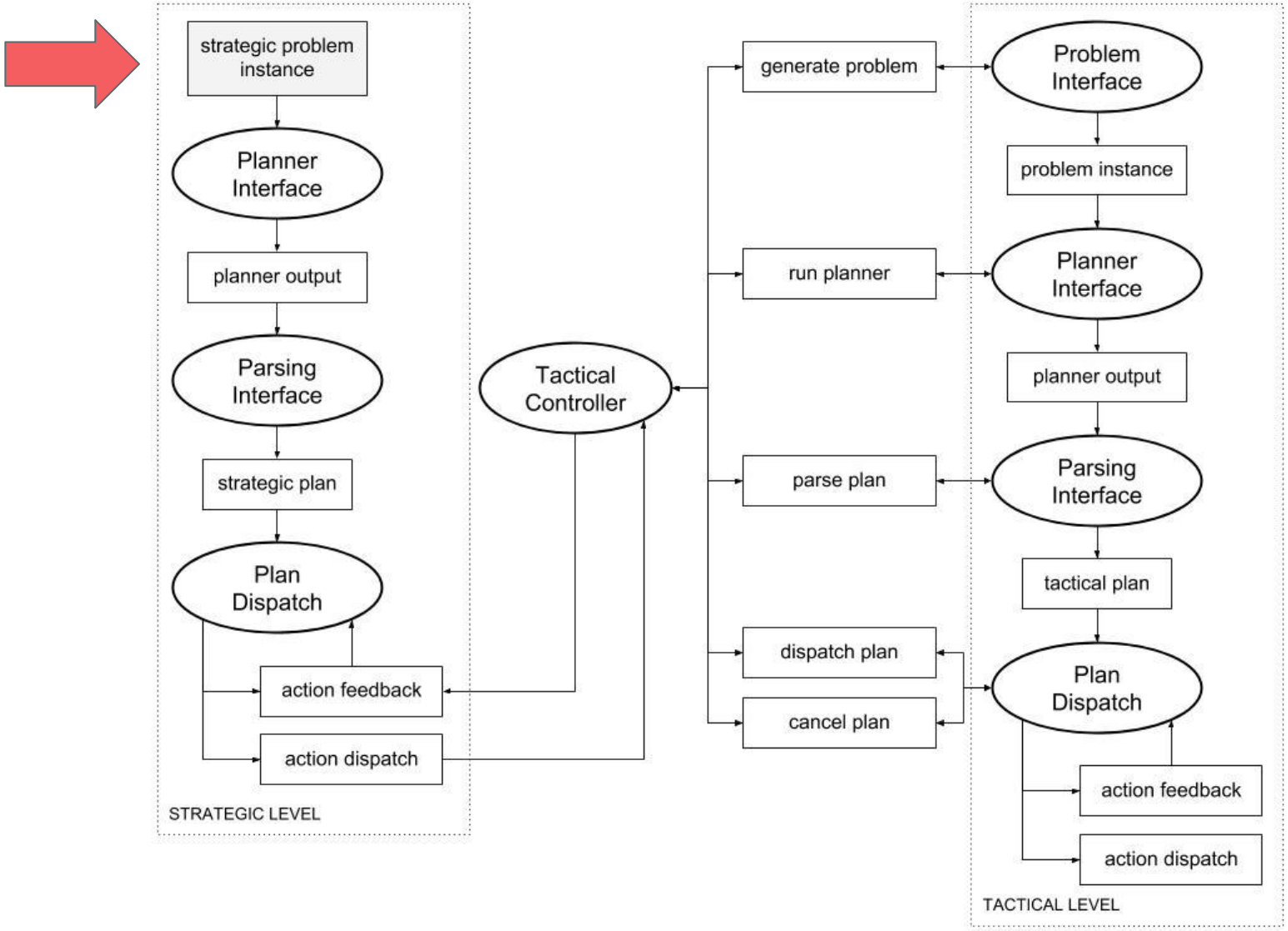

Tactical Knowledge Base
update
Problem Interface
generate problem
Strategic Decomposition
problem instance
strategic problem instance
Planner Interface
run planner
planner output
Parsing Interface
parse plan
tactical plan
Cluster the tasks into subgoals

Generate the Tactical Plans

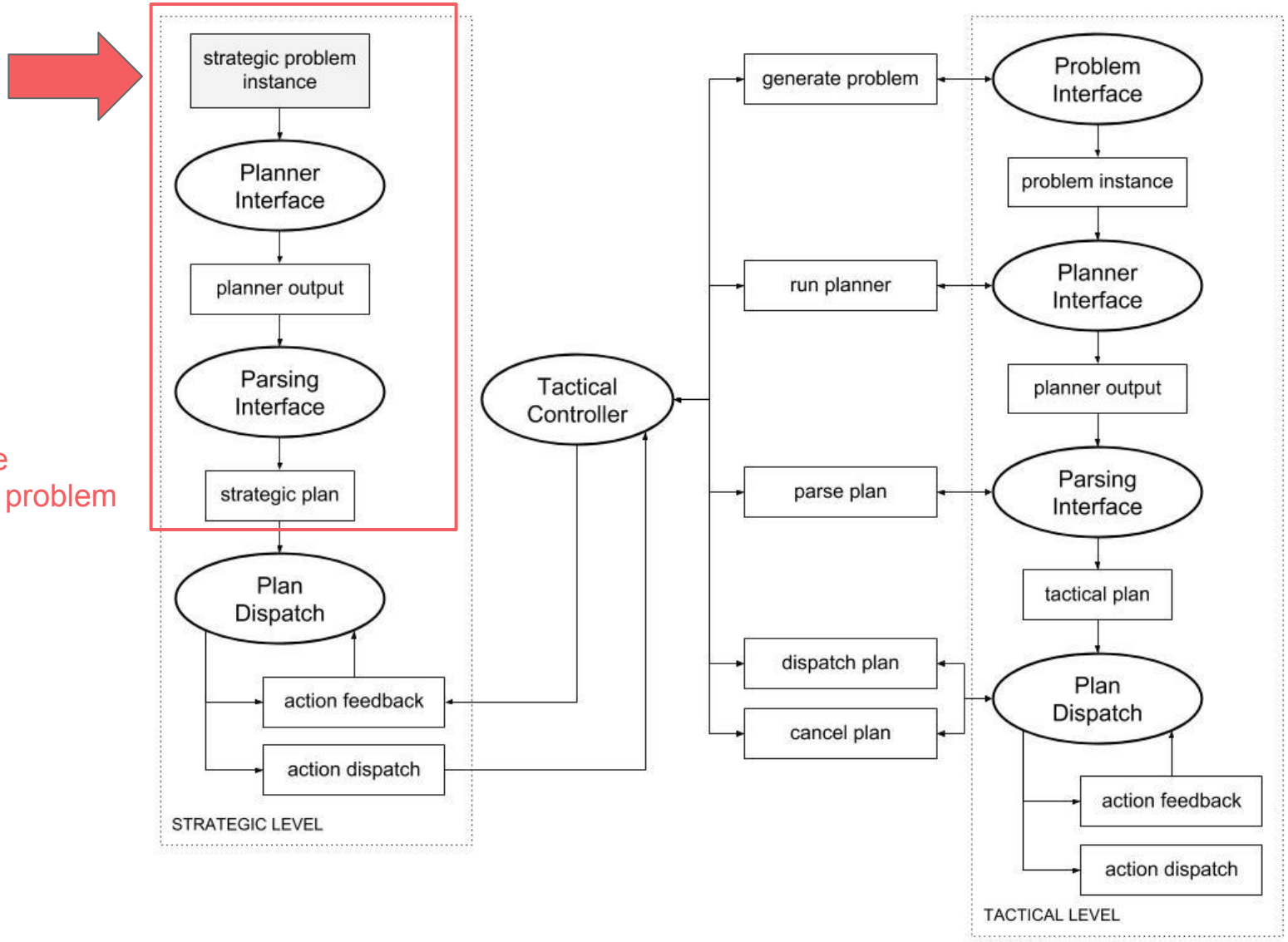




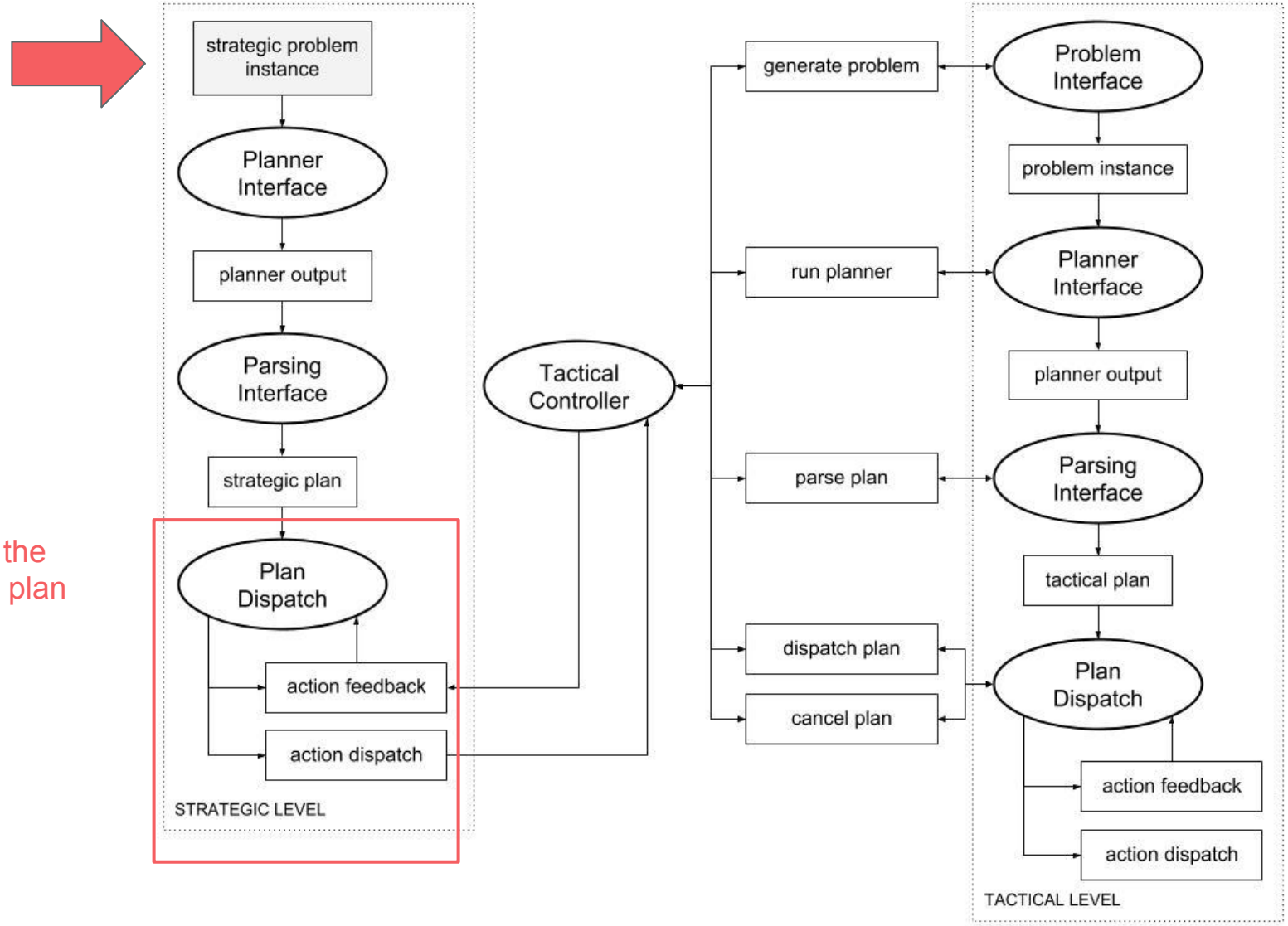
Generate the Strategic Problem

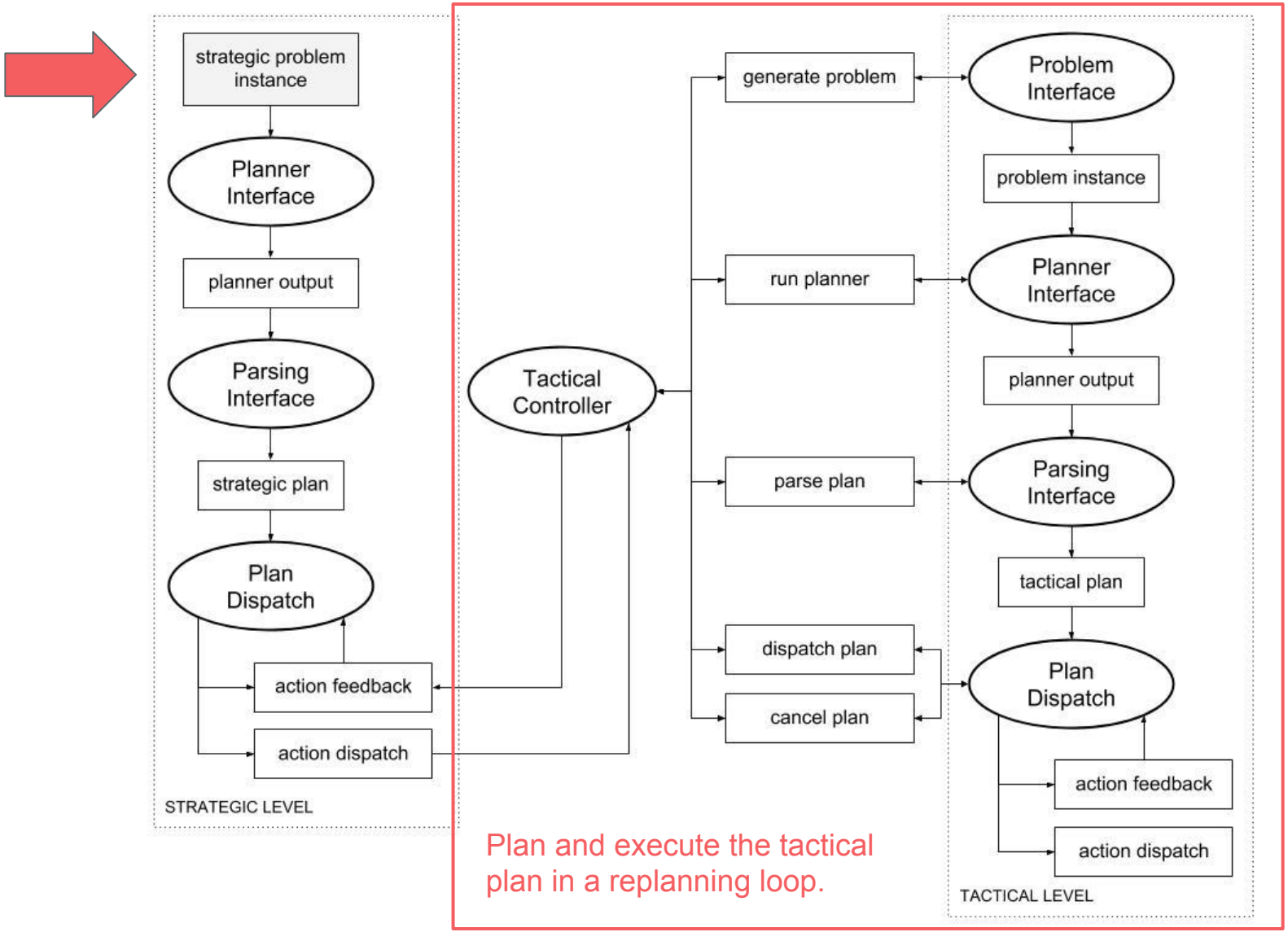

strategic problem instance
Planner Interface
planner output
Parsing Interface
strategic plan
Plan Dispatch
action feedback
action dispatch
STRATEGIC LEVEL
Tactical Controller
generate problem
run planner
parse plan
dispatch plan
cancel plan
Problem Interface
problem instance
Planner Interface
planner output
Parsing Interface
tactical plan
Plan Dispatch
action feedback
action dispatch
TACTICAL LEVEL

Solve the strategic problem



Execute the strategic plan



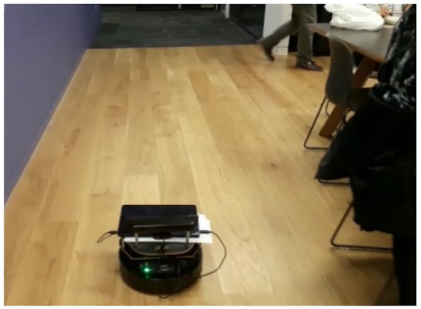

strategic problem instance
Planner Interface
planner output
Parsing Interface
strategic plan
Plan Dispatch
action feedback
action dispatch
STRATEGIC LEVEL
Tactical Controller
generate problem
run planner
parse plan
dispatch plan
cancel plan
Problem Interface
problem instance
Planner Interface
planner output
Parsing Interface
tactical plan
Plan Dispatch
action feedback
action dispatch
TACTICAL LEVEL
Plan and execute the tactical plan in a replanning loop.

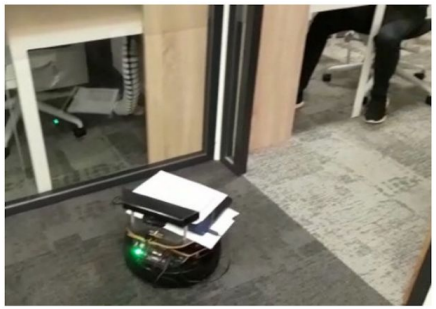
Part 5:
Probabilistic Planning

Probabilistic Planning

The robot is given the task to navigate through a busy office to deliver papers from the printer.

- The printer is often busy.
- The robot relies on nearby humans to load the printed papers.



What is RDDL?

Relational Dynamic Influence Diagram Language

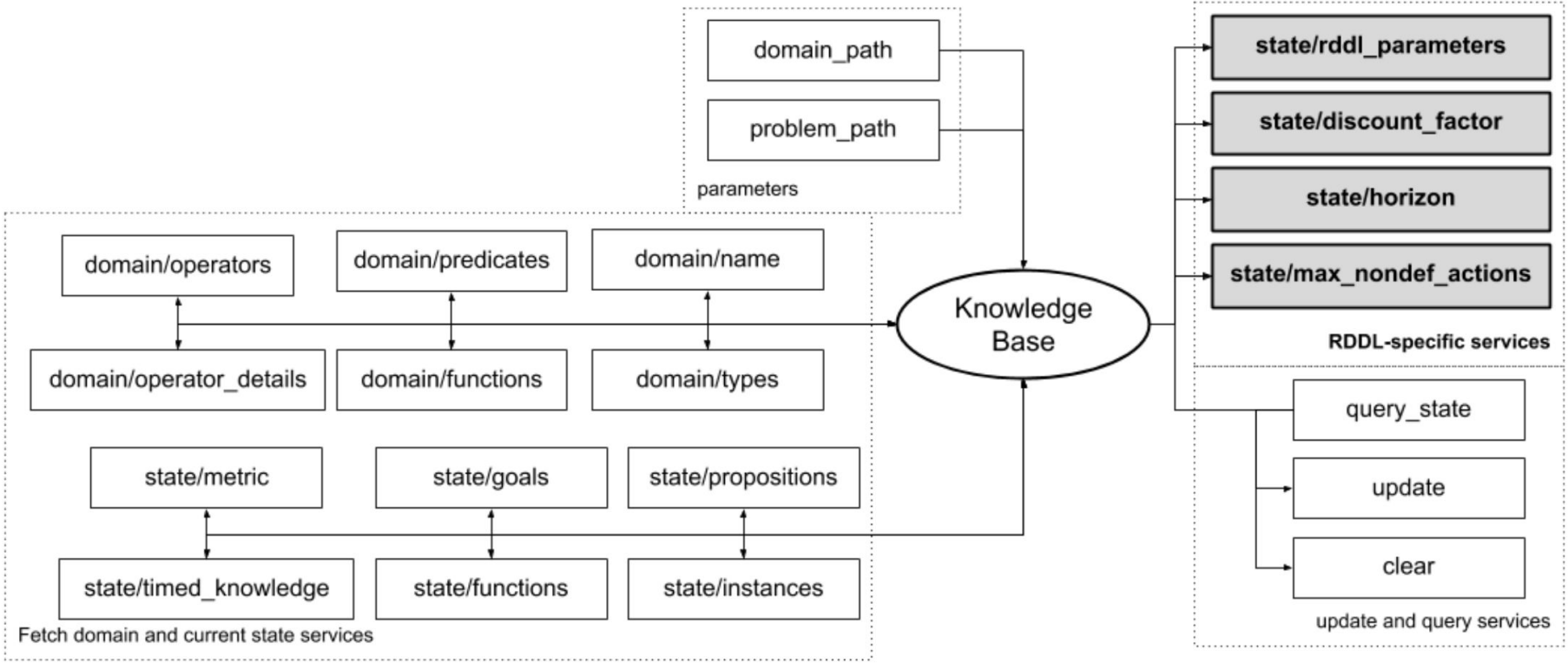
A language for probabilistic planning based on a Dynamic Bayes Net.

Allows domains with:

- Nondeterministic action outcome.
- Domains with partial observation.
- Unrestricted concurrency.

[Sanner 2010]

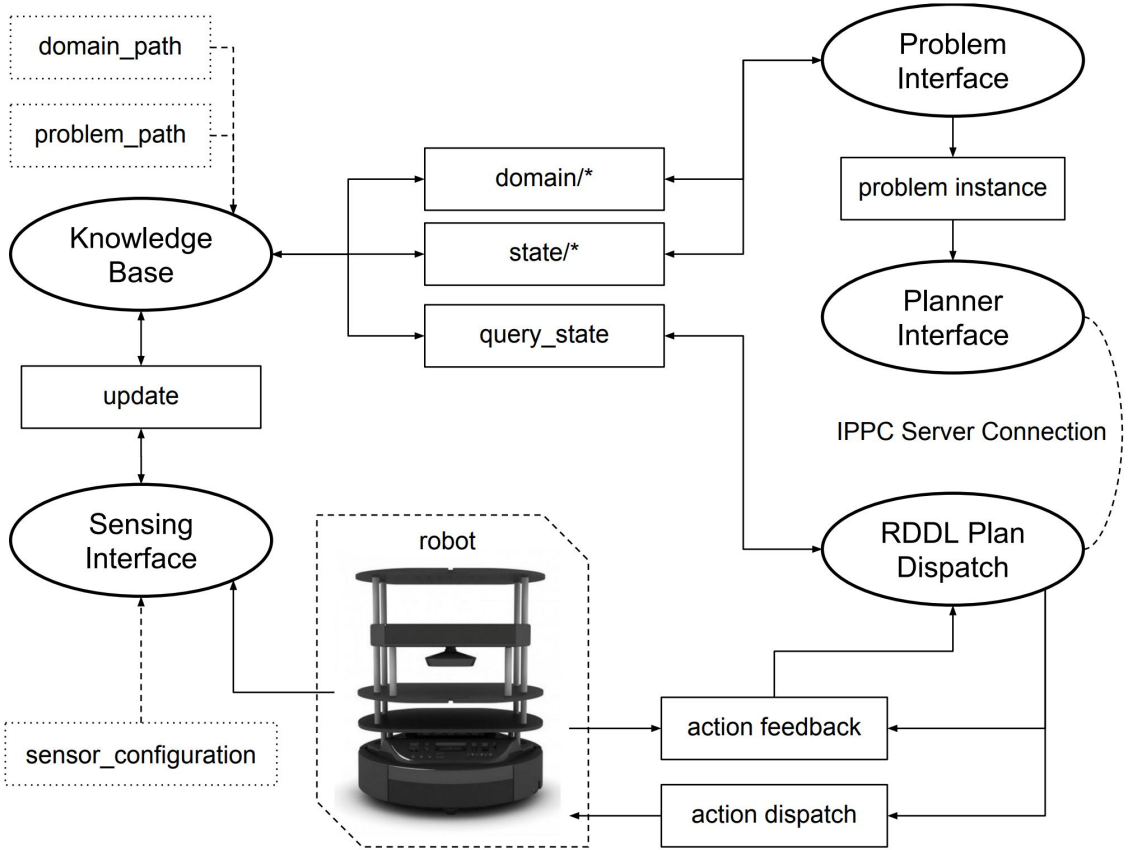
RDDL


domain_path
problem_path
parameters
domain/operators
domain/predicates
domain/name
domain/operator_details
domain/functions
domain/types
state/metric
state/goals
state/propositions
state/timed_knowledge
state/functions
state/instances
Fetch domain and current state services
Knowledge Base
state/rddl_parameters
state/discount_factor
state/horizon
state/max_nondef_actions
RDDL-specific services
query_state
update
clear
update and query services

System Architecture

The default *planner interface*, *plan parser*, and *plan dispatcher* are replaced by a single custom component.

The new node performs *online planning* using the IPPC server connection.



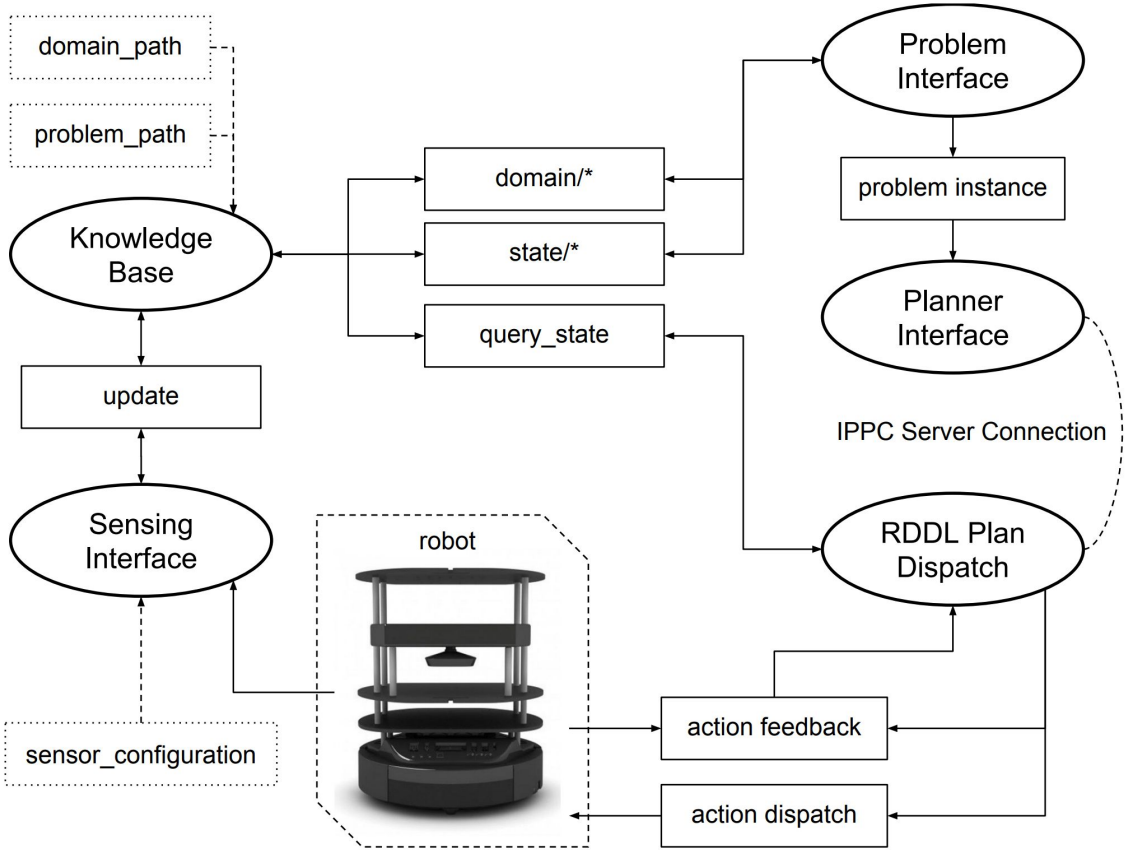
System Architecture

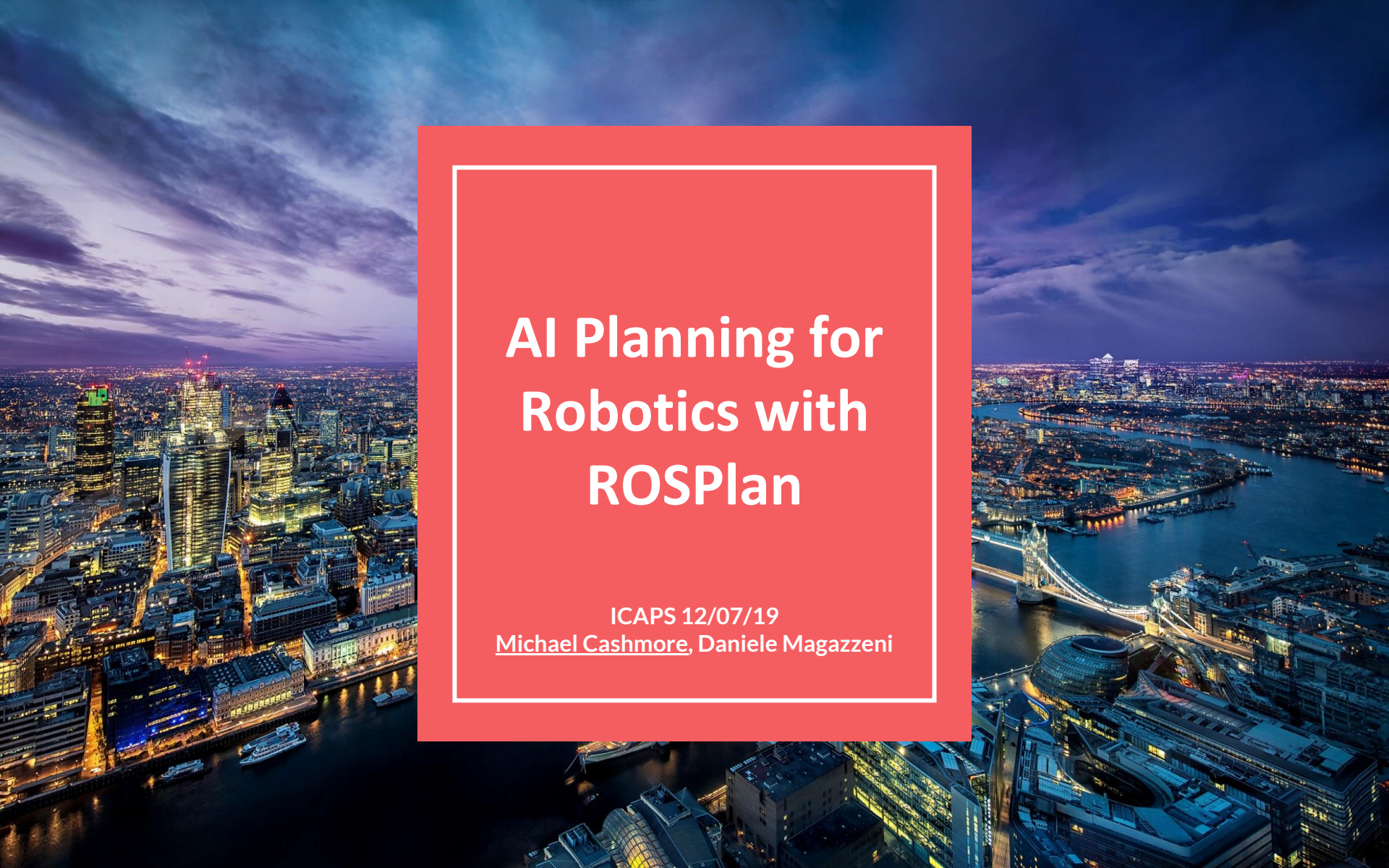
The default *planner interface*, *plan parser*, and *plan dispatcher* are replaced by a single custom component.

The new node performs *online planning* using the IPPC server connection.

The planners PROST and GLUTTON perform planning interleaved with execution.

[Keller and Eyerich 2012]
[Kolobov et al. 2012]





AI Planning for Robotics with ROSPlan

ICAPS 12/07/19

Michael Cashmore, Daniele Magazzeni

References

- Félix Ingrand, Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, Elsevier, 2017, 247 pp.10-44.
- Vaquero, Tiago Stegun et al. The Implementation of a Planning and Scheduling Architecture for Multiple Robots Assisting Multiple Users in a Retirement Home Setting. AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments (2015).
- Ingrand, F., Chatilla, R., Alami, R., & Robert, F. (1996). PRS: a high level supervision and control language for autonomous mobile robots. In IEEE International Conference on Robotics and Automation.
- McGann, C., Py, F., Rajan, K., Thomas, H., Henthorn, R., & McEwen, R. S. (2008). A deliberative architecture for AUV control. In ICRA, pp. 1049–1054.
- Lemai-Chenevier, Solange and François Félix Ingrand. “Interleaving Temporal Planning and Execution: IXTET-EXEC.” (2003).
- Segura-Muros, José Ángel, and Juan Fernández-Olivares. "Integration of an Automated Hierarchical Task Planner in ROS Using Behaviour Trees." 2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT). IEEE, 2017.
- Nardi L., Iocchi L. (2014) Representation and Execution of Social Plans through Human-Robot Collaboration. In: Beetz M., Johnston B., Williams MA. (eds) Social Robotics. ICSR 2014. Lecture Notes in Computer Science, vol 8755. Springer, Cham
- Shaun Azimi, Emma Zemler and Robert Morris. Autonomous Robotics Manipulation for In-space Intra-Vehicle Activity. PlanROB 2019
- Buksz et al. Strategic-Tactical Planning for Autonomous Underwater Vehicles over Long Horizons. IROS 2018
- Paul H. Morris, Nicola Muscettola, Execution of Temporal Plans with Uncertainty. AAAI/IAAI 2000: 491-496
- Paul H. Morris, Nicola Muscettola, Managing Temporal Uncertainty Through Waypoint Controllability. IJCAI 1999: 1253-1258
- Ingham, Michel, Robert Ragno, and Brian C. Williams. "A reactive model-based programming language for robotic space explorers." ISAIRAS-01 (2001).
- V.A. Ziparo, L. Iocchi, Pedro Lima, D. Nardi, P. Palamara. Petri Net Plans - A framework for collaboration and coordination in multi-robot systems. Autonomous Agents and Multi-Agent Systems, vol. 23, no. 3, pp. 344-383, 2011.
- Cashmore, M. Fox, M. Long, D. Larkworthy, T. and Magazzeni D. AUV Mission Control Via Temporal Planning 2014 IEEE International Conference on Robotics and Automation (ICRA 2014)
- Cashmore, M. Fox, M. Long, D. and Larkworthy, T. Planning Inspection Tasks for AUVs Proceedings of the MTS/IEEE Oceans 2013 Conference, San Diego (OCEANS 2013)
- Cashmore, M. Fox, M. Long, D. Magazzeni, D. Carrera, A. Palomeras, N. Hurtos, N. and Carreras, M. ROSPlan: Planning in the Robot Operating System Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)
- N. Palomeras, A. Carrera, N. Hurts, G. C. Karras, C. P. Bechlioulis, M. Cashmore, D. Magazzeni, D. Long, M. Fox, K. J. Kyriakopoulos, P. Kormushev, J. Salvi and M. Carreras Toward persistent autonomous intervention in a subsea panel Autonomous Robots, 2016

References

- Cashmore, M. Fox, M. Long, D. Magazzeni, D and Ridder, B. Strategic Planning for Autonomous Systems over Long Horizons Proceedings of the 4th ICAPS Workshop on Planning and Robotics (PlanRob 2016)
- Cashmore, M. Fox, M. Long, D. Magazzeni, D and Fox, M. Long, D. and Magazzeni, D. Opportunistic Planning for Increased Plan Utility Proceedings of the 4th ICAPS Workshop on Planning and Robotics (PlanRob 2016)
- Langer, E. Ridder, B. Cashmore, M. Magazzeni, D. Zillich, M. Vincze, M. On-the-fly detection of novel objects in indoor environments IEEE International Conference on Robotics and Biomimetics (ROBIO), 2017
- Cashmore, M. Fox, M. Long, D. Magazzeni, D and Ridder, B. Opportunistic Planning in Autonomous Underwater Missions IEEE Transactions on Automation Science and Engineering, 2018
- Cashmore, M. Fox, M. Long, D. Magazzeni, D and Ridder, B. Short-Term Human-Robot Interaction through Conditional Planning and Execution Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017)
- Keller, T., Eyerich, P.: PROST: Probabilistic planning based on UCT. In: ICAPS (2012)
- Kolobov, A., Dai, P., Mausam, M., Weld, D.S.: Reverse iterative deepening for finite-horizon MDPs with large branching factors. In: ICAPS (2012)
- Sanner, S.: Relational dynamic influence diagram language (RDDL): Language description. Unpublished Manuscript (2010)