ROS Basics

ROS offers a message passing interface that provides inter-process communication.

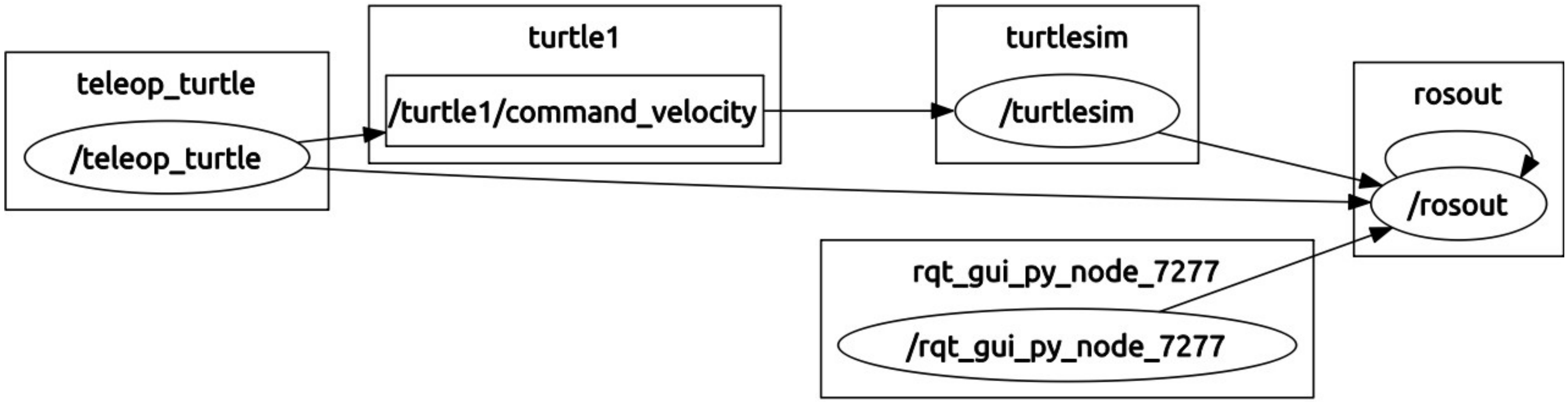
A ROS system is composed of nodes, which pass messages, usually in two forms:

1. ROS messages are published on topics and are many-to-many.
2. ROS services are used for synchronous request/response.

ROS Basics

ROS offers a message passing interface that provides inter-process communication.

A ROS system is composed of nodes, which pass messages, usually in two forms:
1. ROS messages are published on topics and are many-to-many.
2. ROS services are used for synchronous request/response.



ROS Basics

ROS offers a message passing interface that provides inter-process communication.

A ROS system is composed of nodes, which pass messages, usually in two forms:
1. ROS messages are published on topics and are many-to-many.
2. ROS services are used for synchronous request/response.

```
<launch>
  <include file="$(find turtlebot_navigation)/launch/includes/velocity_smoother.launch.xml"/>
  <include file="$(find turtlebot_navigation)/launch/includes/safety_controller.launch.xml"/>

  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="odom" />
  <arg name="laser_topic" default="scan" />


  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <rosparam file="$(find turtlebot_navigation)/param/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find turtlebot_navigation)/param/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find turtlebot_navigation)/param/local_costmap_params.yaml" command="load" />
    <remap from="cmd_vel" to="navigation_velocity_smoother/raw_cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>
  </node>
</launch>
```

ROS Basics

ROS offers a message passing interface that provides inter-process communication.

A ROS system is composed of nodes, which pass messages, usually in two forms:
1. ROS messages are published on topics and are many-to-many.
2. ROS services are used for synchronous request/response.

The actionlib package standardizes the interface for pre-emptable tasks.
For example:
- navigation,
- performing a laser scan
- detecting the handle of a door...

Aside from numerous tools, Actionlib provides standard messages for sending task:
- goals
- feedback
- result

ROS Basics

Aside from numerous tools, Actionlib provides standard messages for sending task:
- goals
- feedback
- result

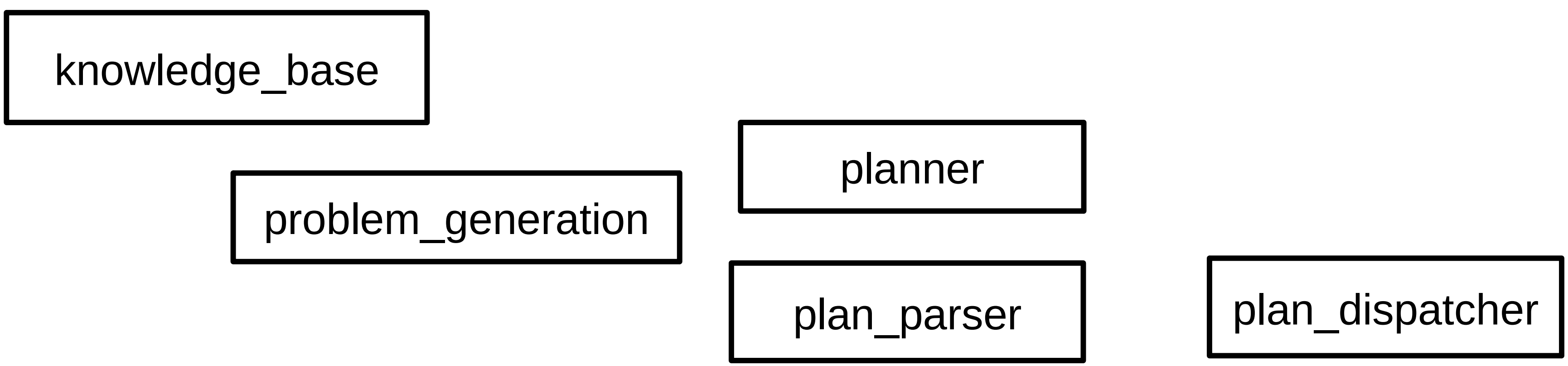
move_base/MoveBaseGoal

```
geometry_msgs/PoseStamped target_pose
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
```

ROSPlan Basics

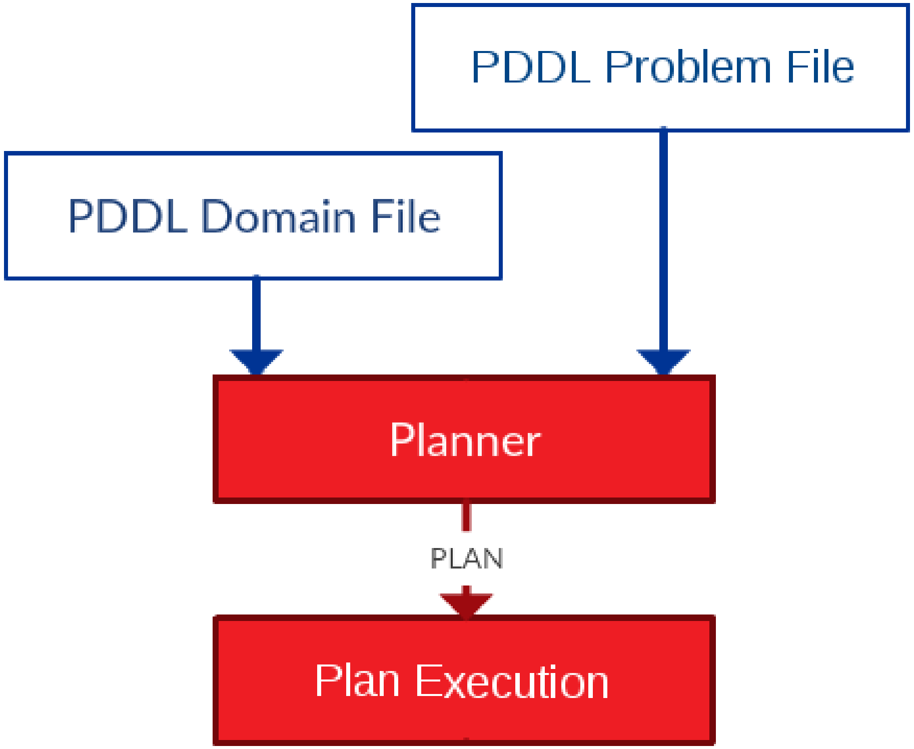
The ROSPlan package provides a standard interface for PDDL planners in ROS.

The purpose of the ROSPlan package is to integrate planners within a ROS system without having to write an architecture from scratch.



Plan Execution 1: Very simple Dispatch

The most basic structure.
- The plan is generated.
- The plan is executed.

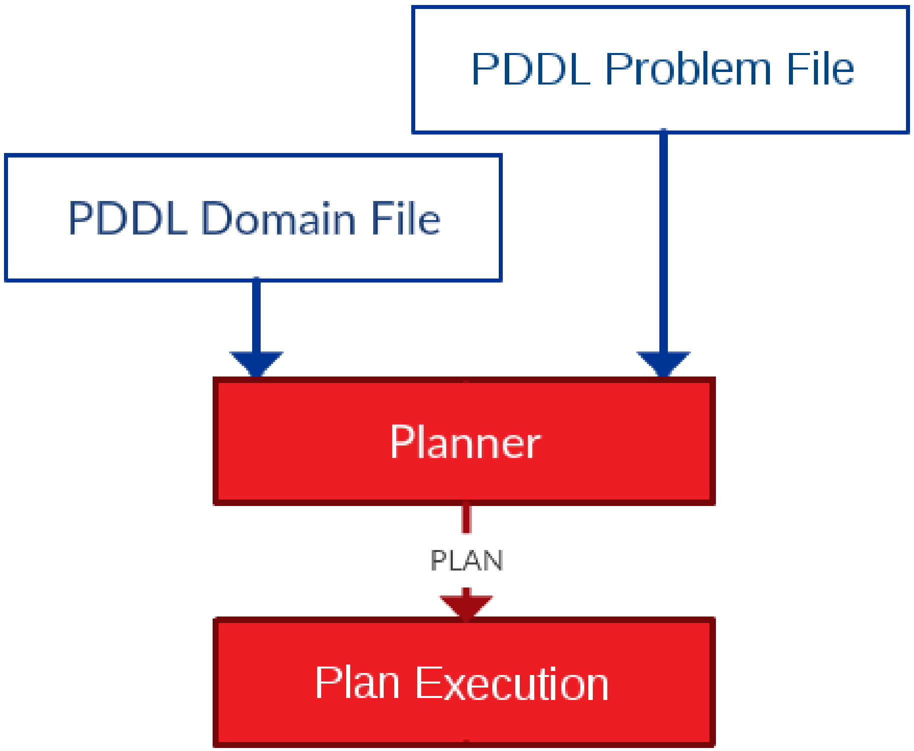


Plan Execution 1: Very simple Dispatch

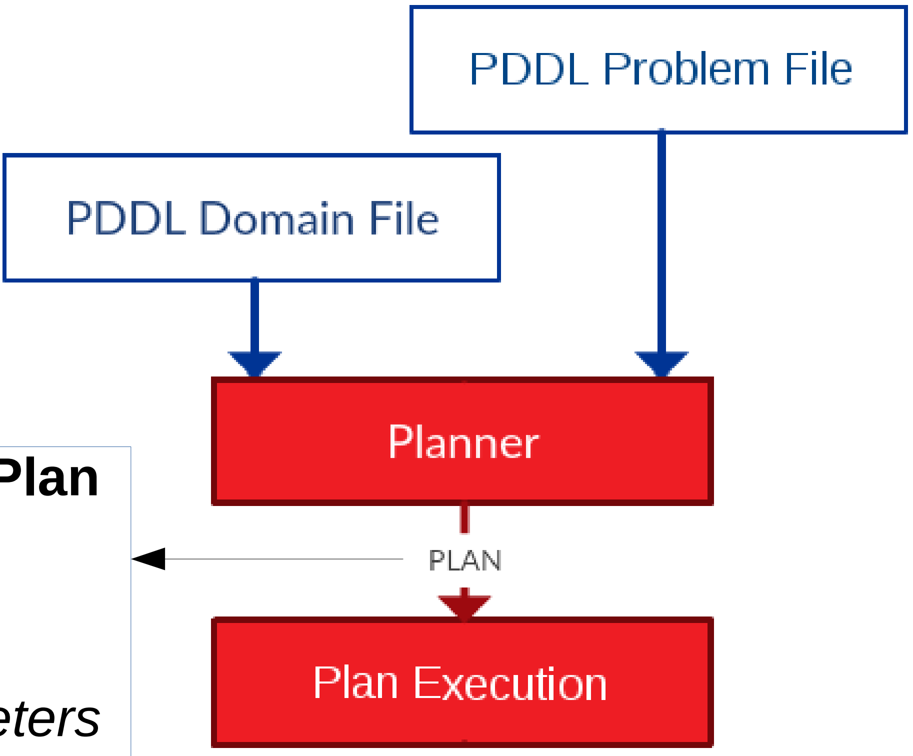
The most basic structure.
- The plan is generated.
- The plan is executed.

The red boxes are included in ROSPlan. They correspond to ROS nodes.

The domain and problem file can be supplied
- in launch parameters
- as ROS service parameters



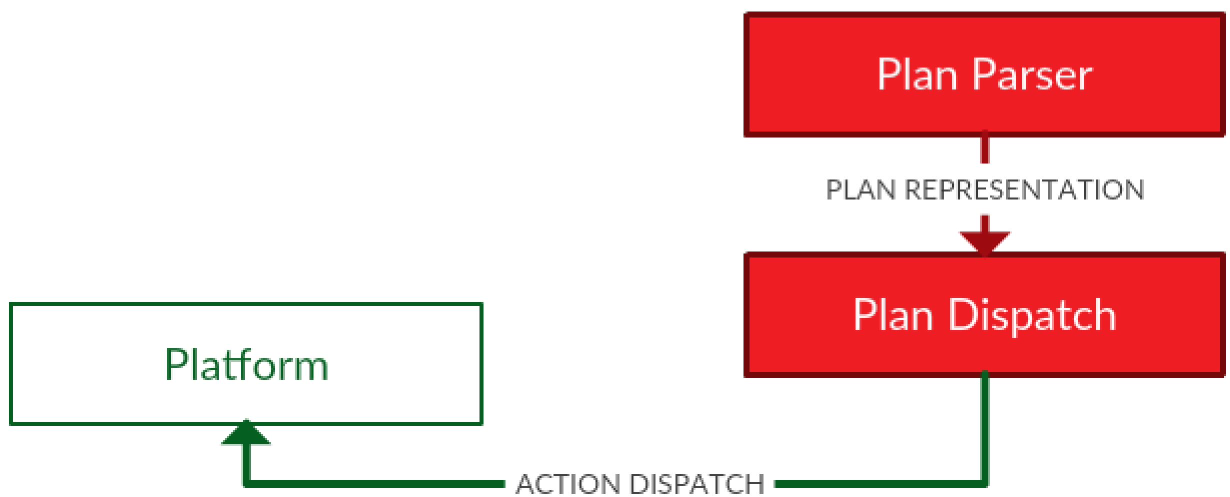
Plan Execution 1: Very simple Dispatch



rosplan_dispatch_msgs/CompletePlan
ActionDispatch[] plan
  int32 action_id
  string name
  diagnostic_msgs/KeyValue[] parameters
    string key
    string value
  float32 duration
  float32 dispatch_time

Dispatch Loop without feedback

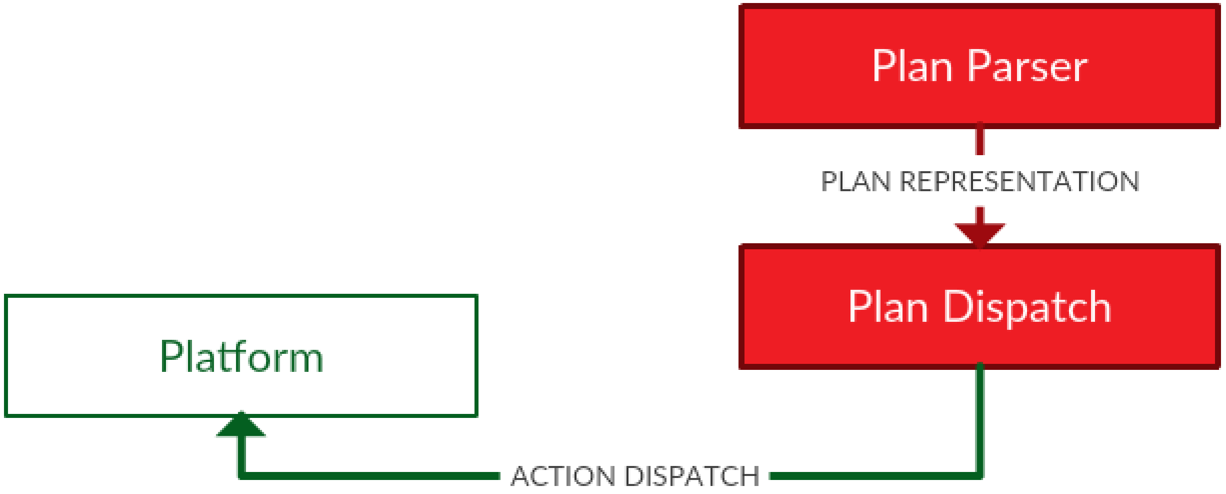
How does the “Plan Execution” ROS node work? There are multiple variants:
- simple sequential execution
- timed execution
- Petri-Net plans
- Conditional Contingent Temporal Constraint Network.
- etc.



Dispatch Loop without feedback

How does the “Plan Execution” ROS node work? There are multiple variants:
- simple sequential execution

1. Take the next action from the plan.
2. Send the action to control.
3. Wait for the action to complete.
4. GOTO 1.

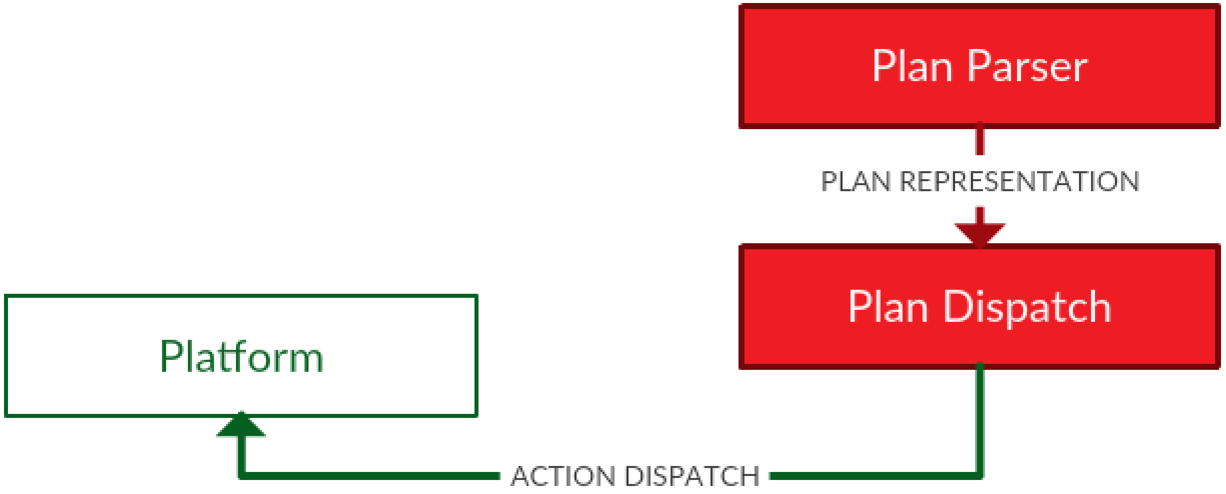


Dispatch Loop without feedback

How does the “Plan Execution” ROS node work? There are multiple variants:
- simple sequential execution

1. Take the next action from the plan.
2. Send the action to control.
3. Wait for the action to complete.
4. GOTO 1.

An action in the plan is stored as a ROS message *ActionDispatch,* which corresponds to a PDDL action.

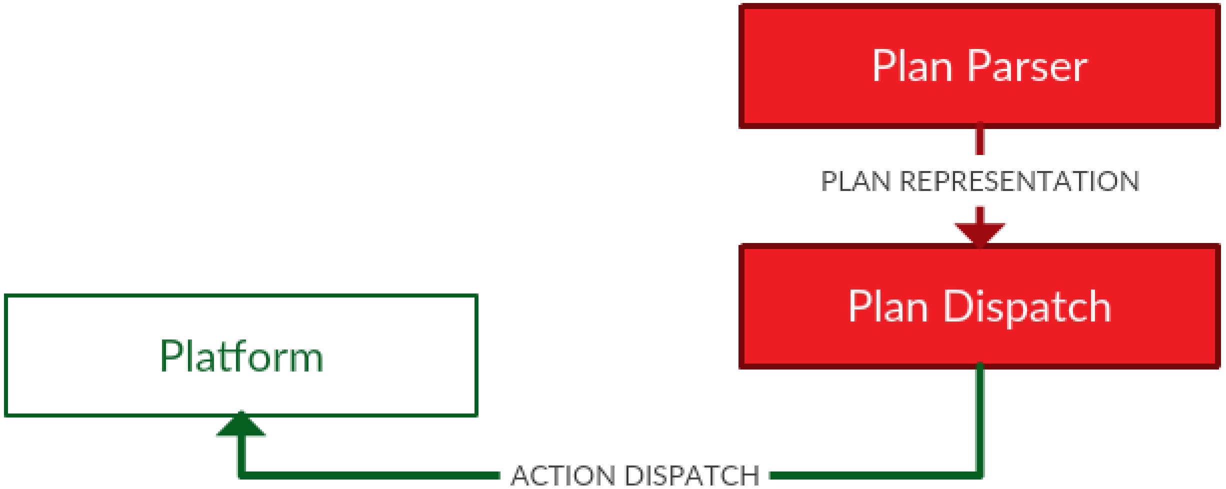


Dispatch Loop without feedback

How does the “Plan Execution” ROS node work? There are multiple variants:
- simple sequential execution

1. Take the next action from the plan.
2. Send the action to control.
3. Wait for the action to complete.
4. GOTO 1.

The *ActionDispatch* message is received by a listening interface node, and becomes a goal for control.



Dispatch Loop without feedback

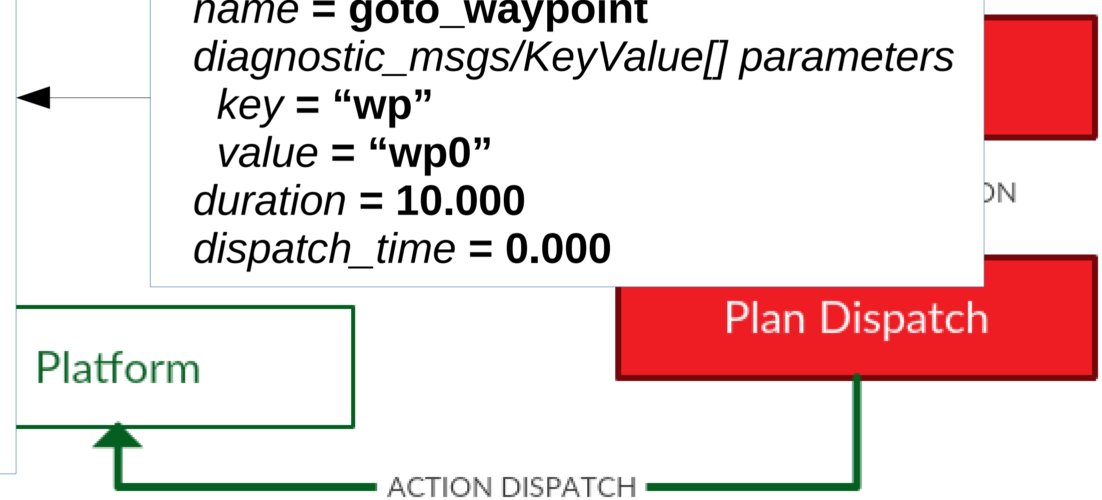
How does the “Plan Execution” ROS node work? There are multiple variants:
- simple sequential execution

1. Take the next action from the plan.
2. Send the action to control.
3. Wait for the action to complete.
4. GOTO 1.

0.000:	*(goto_waypoint wp0)*	*[10.000]*
10.01:	*(observe ip3)*	*[5.000]*
15.02:	*(grasp_object box4)*	*[60.000]*

ActionDispatch
action_id **= 0**
name **= goto_waypoint**
diagnostic_msgs/KeyValue[] parameters
key **= “wp”**
value **= “wp0”**
duration **= 10.000**
dispatch_time **= 0.000**

move_base/MoveBaseGoal
geometry_msgs/PoseStamped target_pose
std_msgs/Header header
...
geometry_msgs/Pose pose
geometry_msgs/Point position
float64 x
float64 y
float64 z
geometry_msgs/Quaternion orientation
...

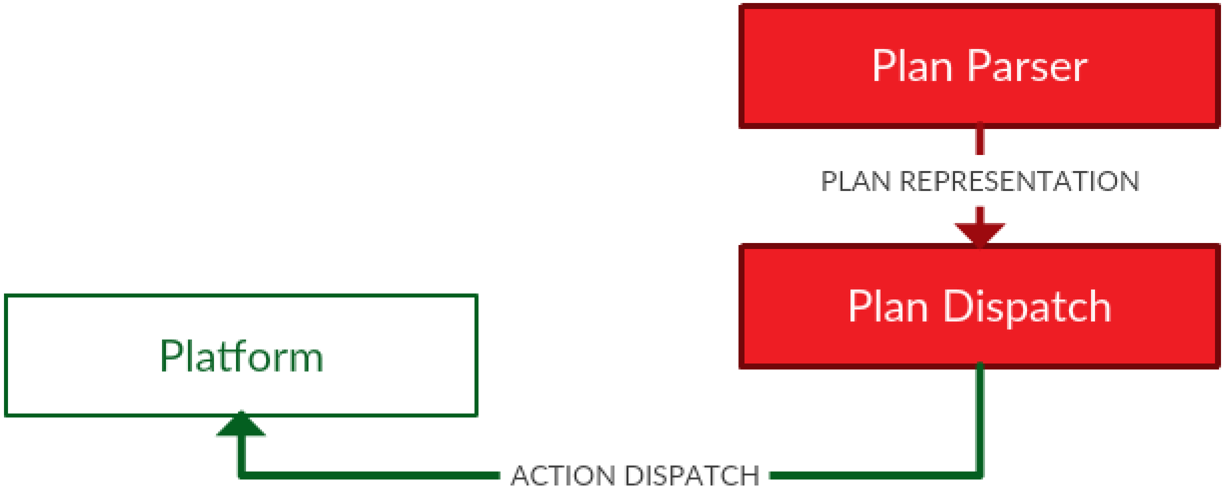


Dispatch Loop without feedback

How does the "Plan Execution" ROS node work? There are multiple variants:
- simple sequential execution

1. Take the next action from the plan.
2. Send the action to control.
3. Wait for the action to complete.
4. GOTO 1.

Feedback is returned to the simple dispatcher (action success or failure) through a ROS message *ActionFeedback.*

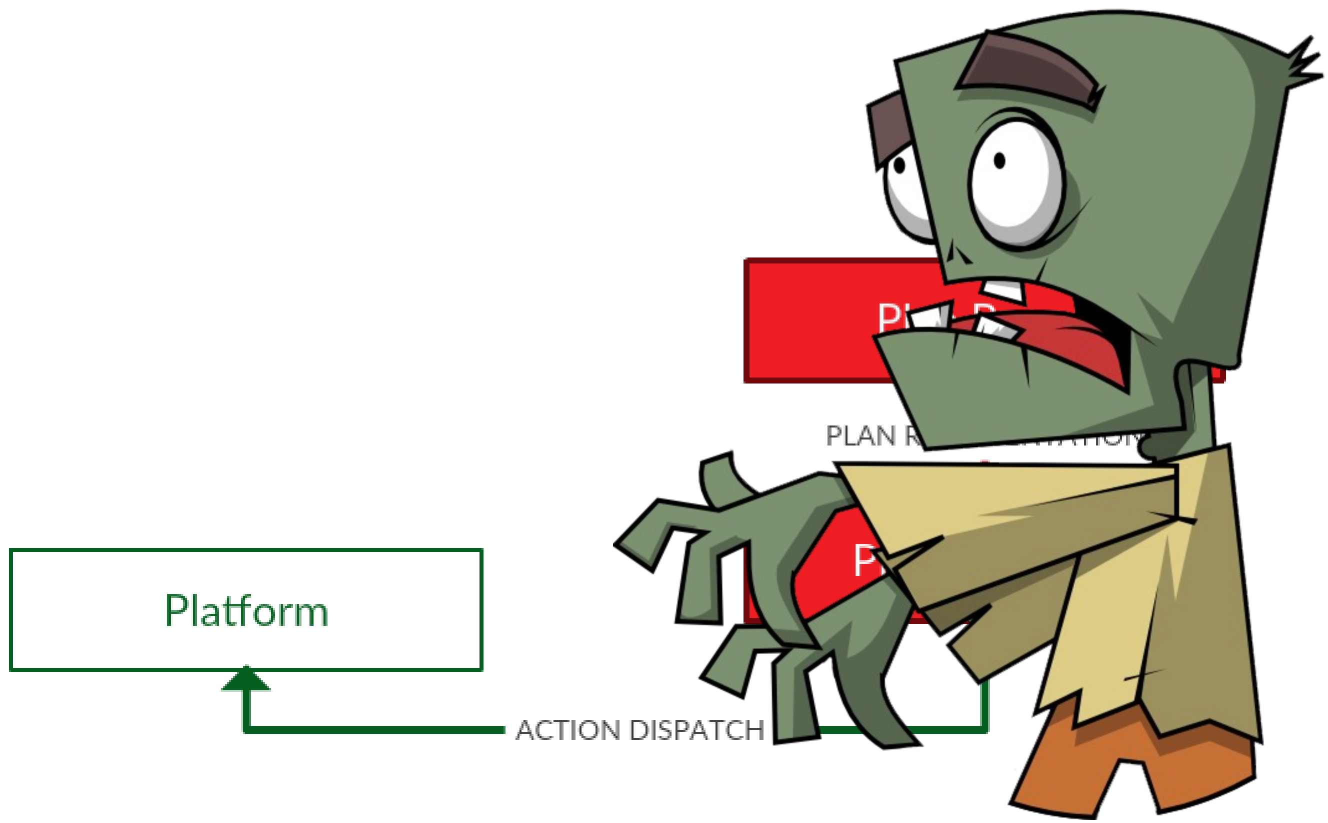


Plan Execution Failure

This form of simple dispatch has some problems. The robot often exhibits zombie-like behaviour in one of two ways:

1. An action fails, and the recovery is handled by control.

2. The plan fails, but the robot doesn't notice.



Bad behaviour 1: Action Failure

An action might never terminate. For example:
- a navigation action that cannot find a path to its goal.
- a grasp action that allows retries.

At some point the robot must give up.

Bad behaviour 1: Action Failure

An action might never terminate. For example:
- a navigation action that cannot find a path to its goal.
- a grasp action that allows retries.

At some point the robot must give up.

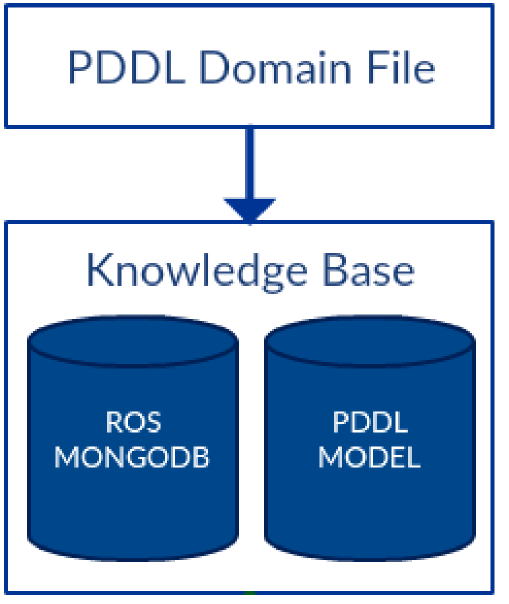
If we desire persistent autonomy, then the robot must be able to plan again, from the new current state, without human intervention.

The problem file must be regenerated.

PDDL Model

To generate the problem file automatically, the agent must store a model of the world.

In ROSPlan, a PDDL model is stored in a ROS node called the Knowledge Base.

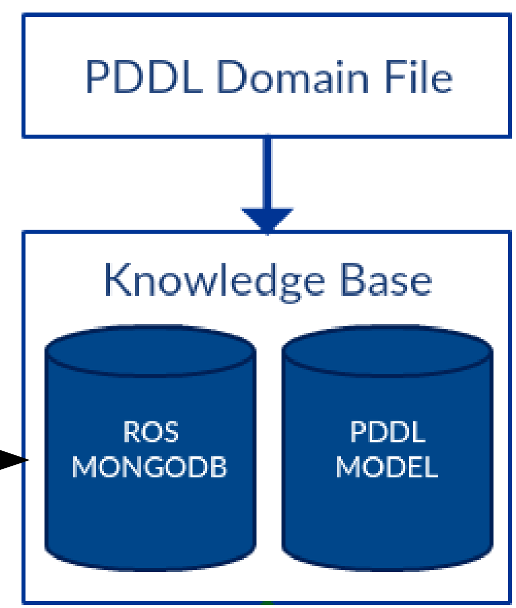


PDDL Model

To generate the problem file automatically, the agent must store a model of the world.

In ROSPlan, a PDDL model is stored in a ROS node called the Knowledge Base.

rosplan_knowledge_msgs/KnowledgeItem
uint8 INSTANCE=0
uint8 FACT=1
uint8 FUNCTION=2
uint8 knowledge_type
string instance_type
string instance_name
string attribute_name
diagnostic_msgs/KeyValue[] values
  string key
  string value
float64 function_value
bool is_negative



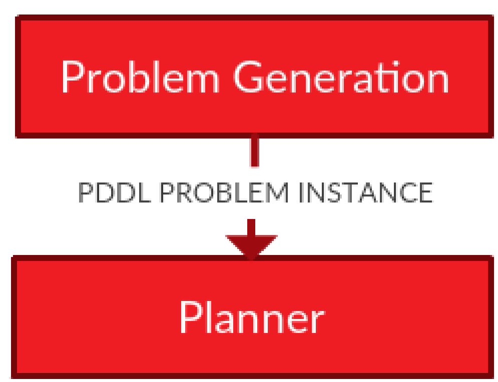
PDDL Model

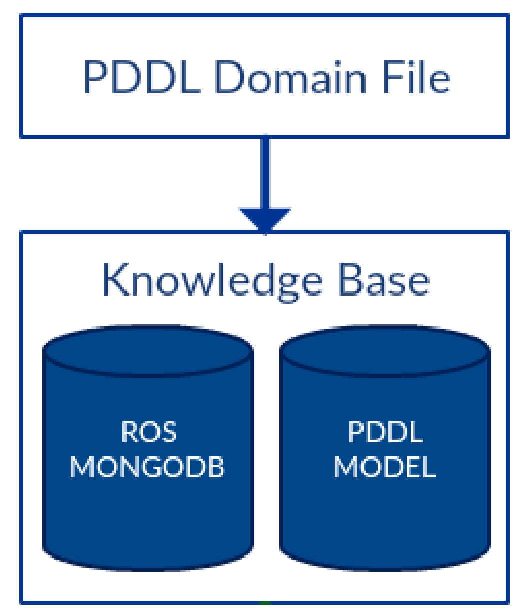
To generate the problem file automatically, the agent must store a model of the world.

In ROSPlan, a PDDL model is stored in a ROS node called the Knowledge Base.

From this the initial state of a new planning problem can be created.

ROSPlan contains a node which will generate a problem file for the ROSPlan planning node.

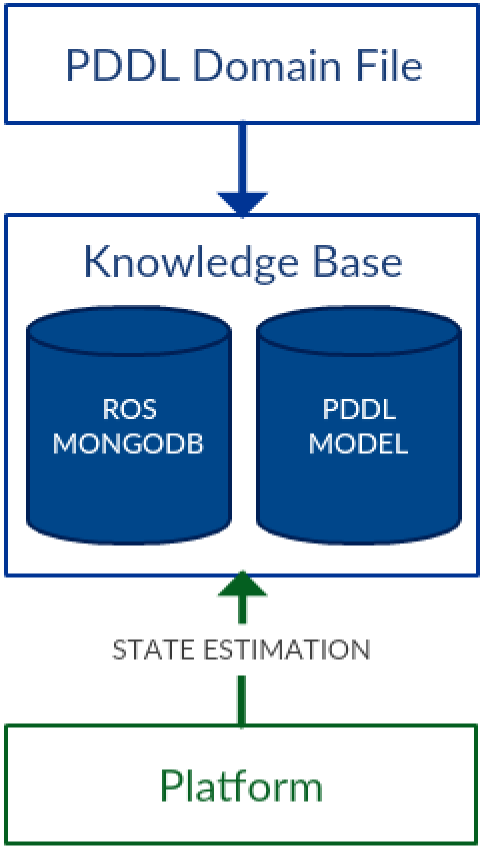




PDDL Model

The model must be continuously updated from sensor data.

For example a new ROS node:
1. subscribes to odometry data.
2. compares odometry to waypoints from the PDDL model.
3. adjusts the predicate (robot_at ?r ?wp) in the Knowledge Base.



PDDL Model

The model must be continuously updated from sensor data.

For example a new ROS node:
1. subscribes to odometry data.
2. compares odometry to waypoints from the PDDL model.
3. adjusts the predicate (robot_at ?r ?wp) in the Knowledge Base.

PDDL Domain File

nav_msgs/Odometry
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
    geometry_msgs/Vector3 angular
  float64[36] covariance

rosplan_knowledge_msgs/KnowledgeItem
uint8 INSTANCE=0
uint8 FACT=1
uint8 FUNCTION=2
uint8 knowledge_type
string instance_type
string instance_name
string attribute_name
diagnostic_msgs/KeyValue[] values
  string key
  string value
float64 function_value
bool is_negative

Bad Behaviour 2: Plan Failure

What happens when the actions succeed, but the plan fails?

This can't always be detected by lower level control.

Bad Behaviour 2: Plan Failure

What happens when the actions succeed, but the plan fails?

This can't always be detected by lower level control.

PLAN COMPLETE

Bad Behaviour 2: Plan Failure

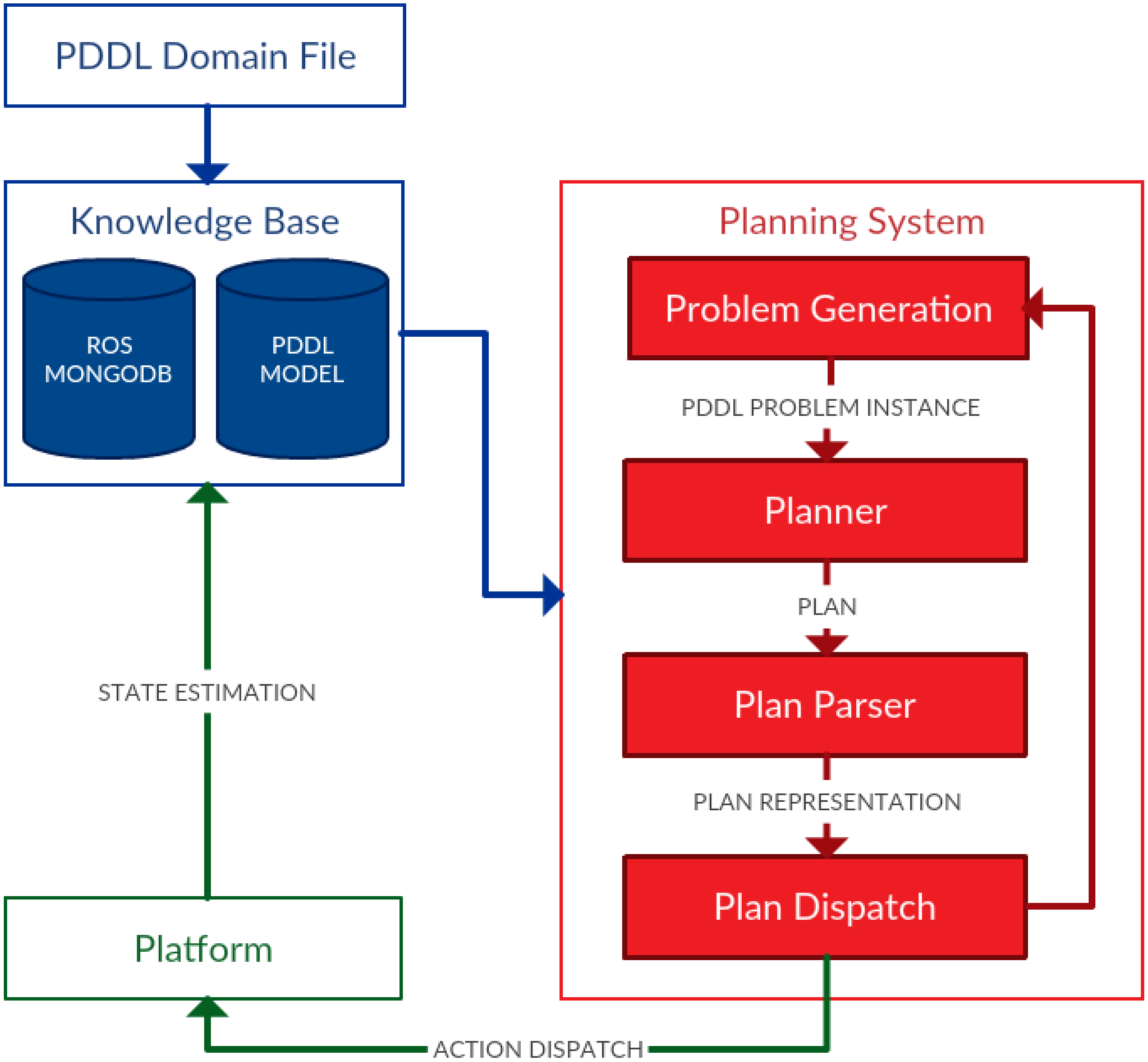
There should be diagnosis at the level of the plan.

If the plan will fail in the future, the robot should not continue to execute the plan for a long time without purpose.

Bad Behaviour 2: Plan Failure

There should be diagnosis at the level of the plan.

If the plan will fail in the future, the robot should not continue to execute the plan for a long time without purpose.



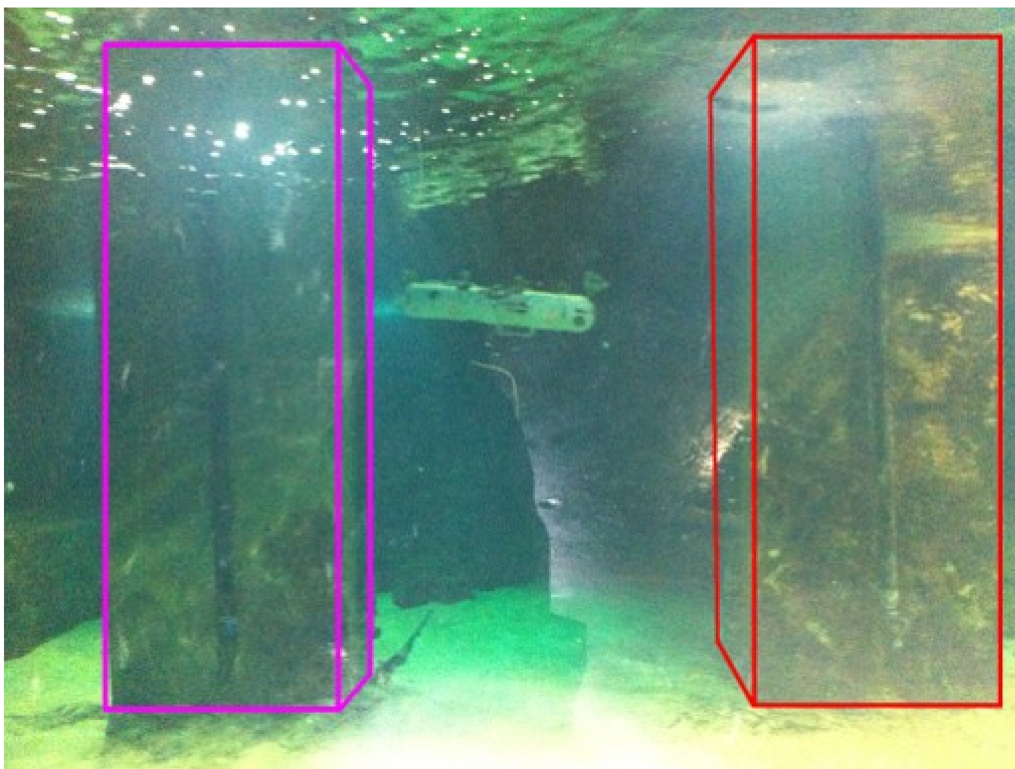
Bad Behaviour 2: Plan Failure

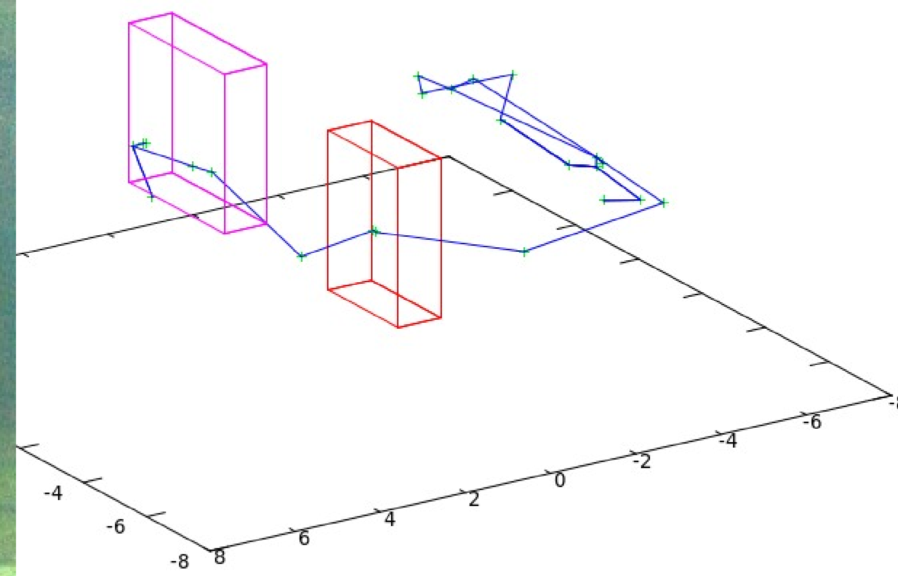
The AUV plans for inspection missions, recording images of pipes and welds.

It navigates through a probabilistic roadmap. The environment is uncertain, and the roadmap might not be correct.

Bad Behaviour 2: Plan Failure

The plan is continuously validated against the model.



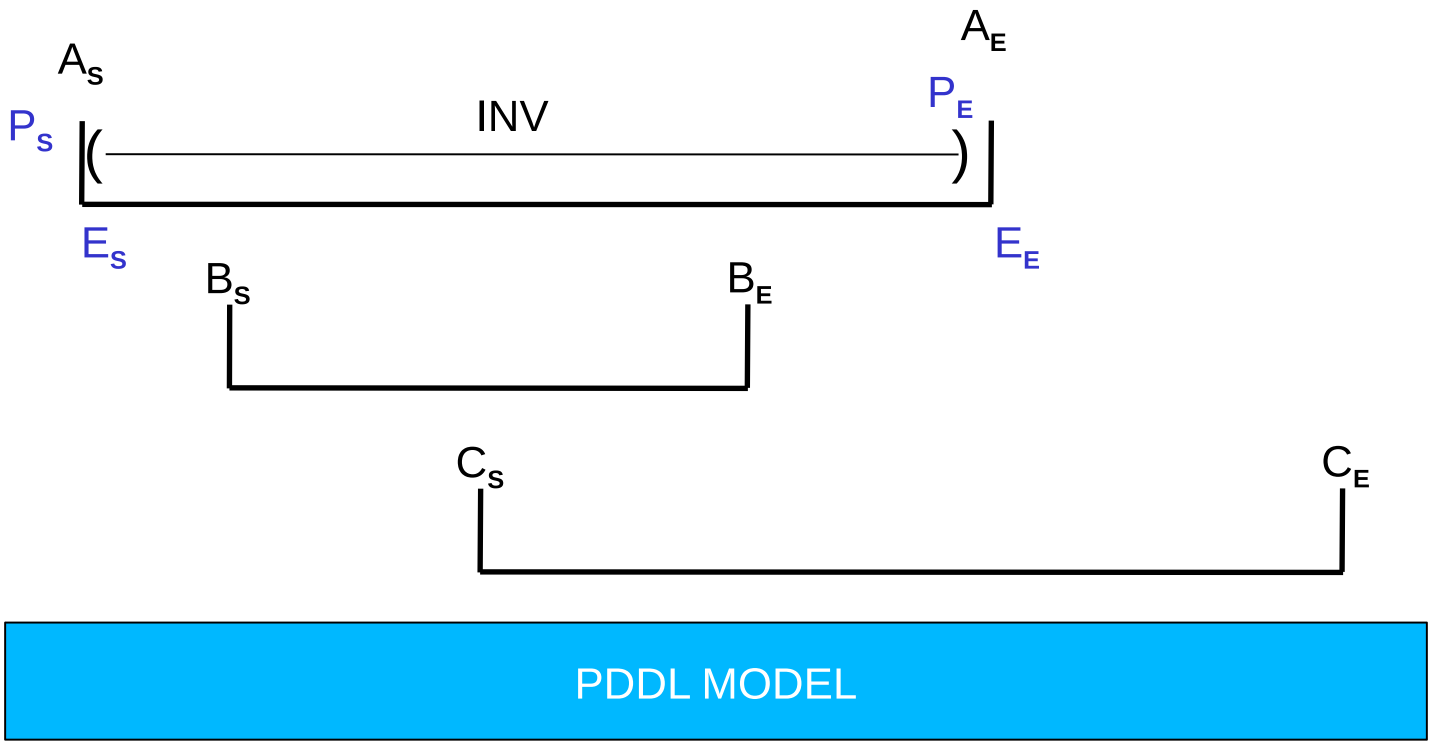


The planned inspection path is shown on the right. The AUV will move around to the other side of the pillars before inspecting the pipes on their facing sides.

After spotting an obstruction between the pillars, the AUV should re-plan early.

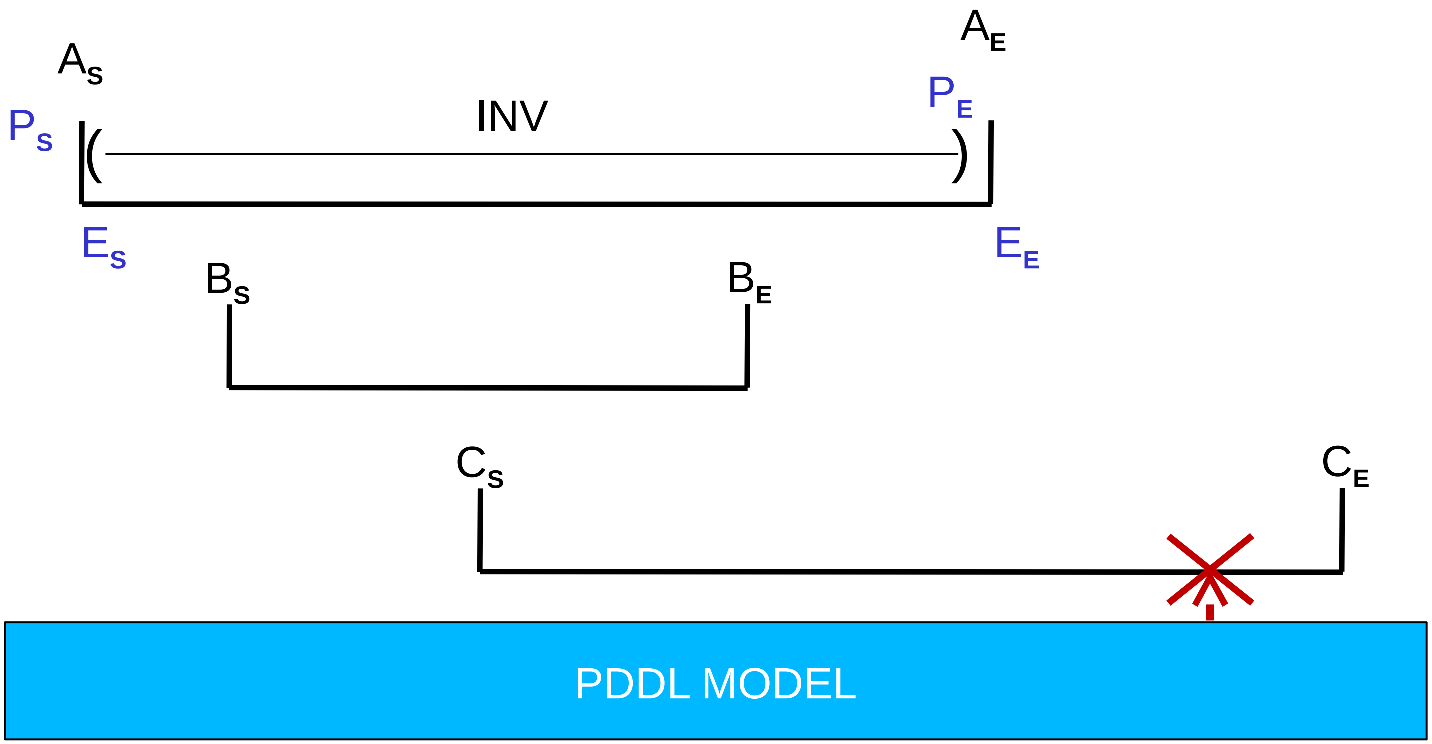
Bad Behaviour 2: Plan Failure

The plan is continuously validated against the model.



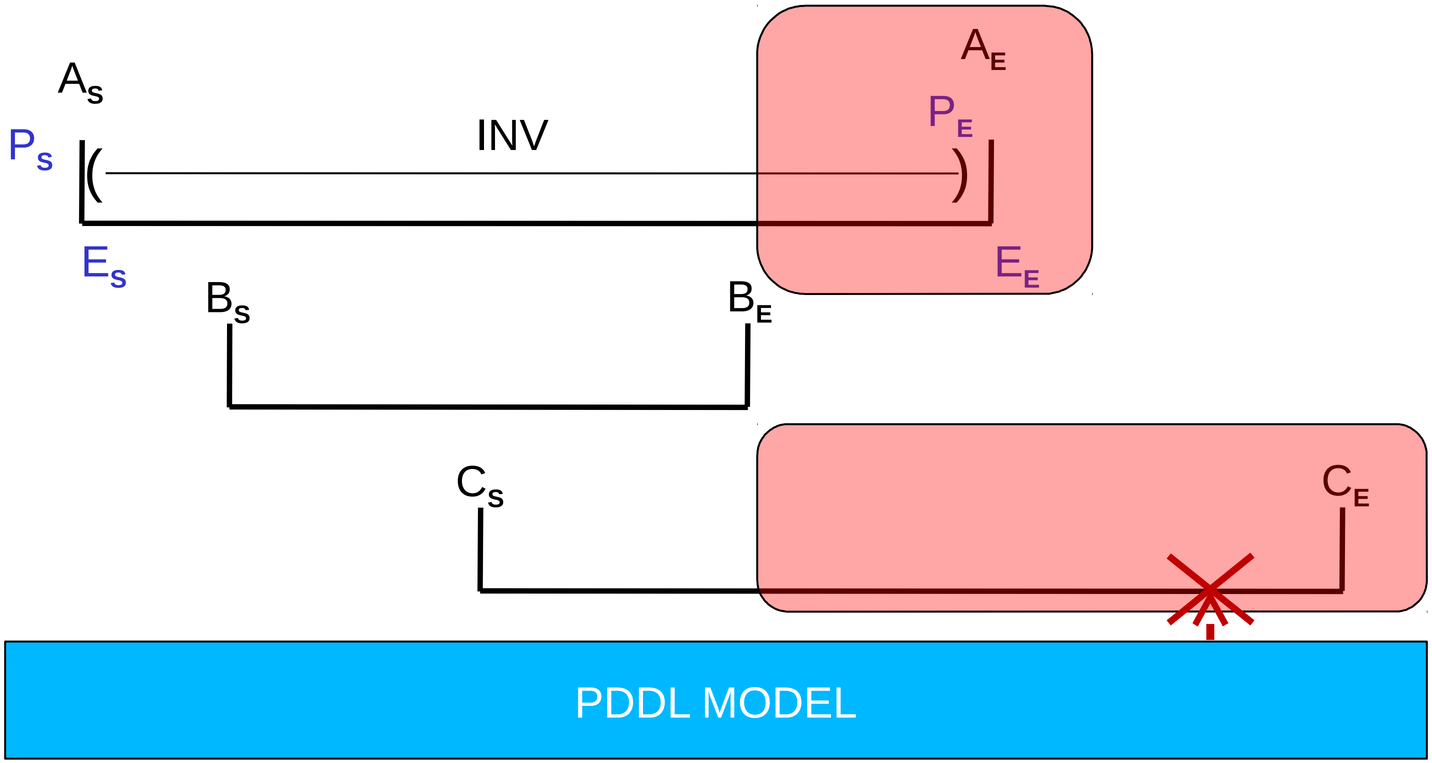
Bad Behaviour 2: Plan Failure

The plan is continuously validated against the model.



Bad Behaviour 2: Plan Failure

The plan is continuously validated against the model.

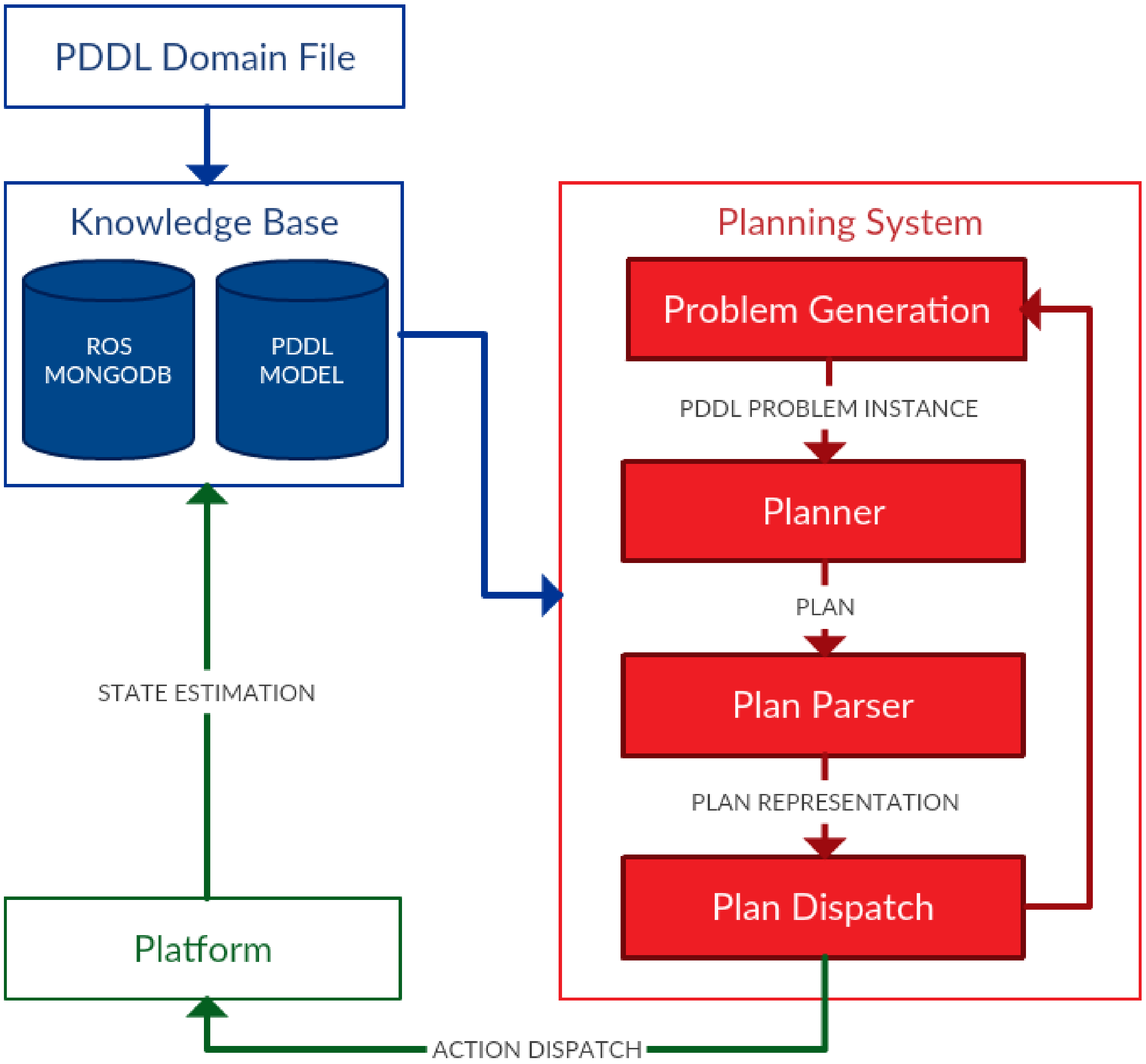


Bad Behaviour 2: Plan Failure

ROSPlan validates using VAL. [Fox et al. 2005]

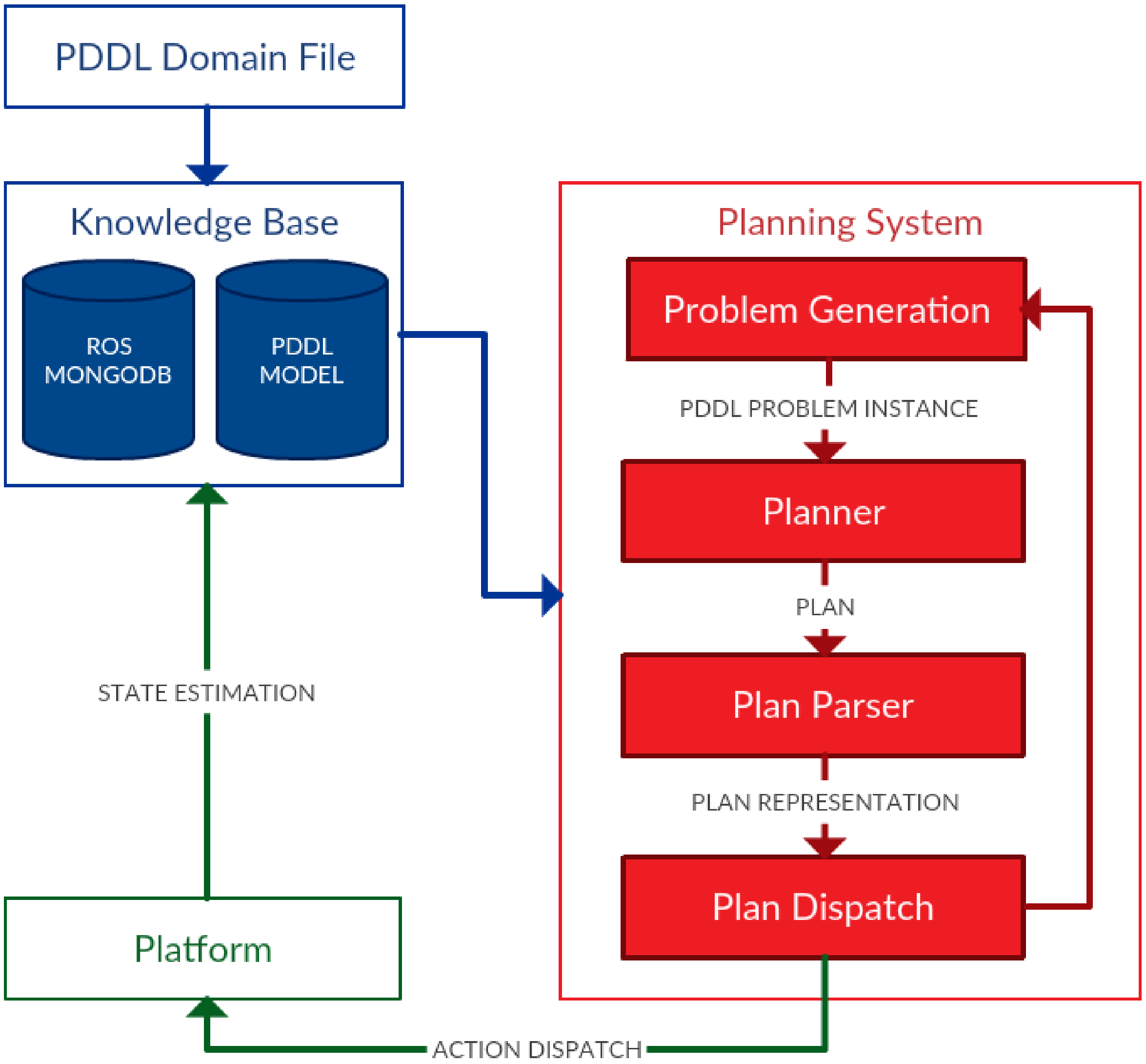
ROSPlan: Default Configuration

Now the system is more complex:
- PDDL model is continuously updated from sensor data.
- problem file is automatically generated.



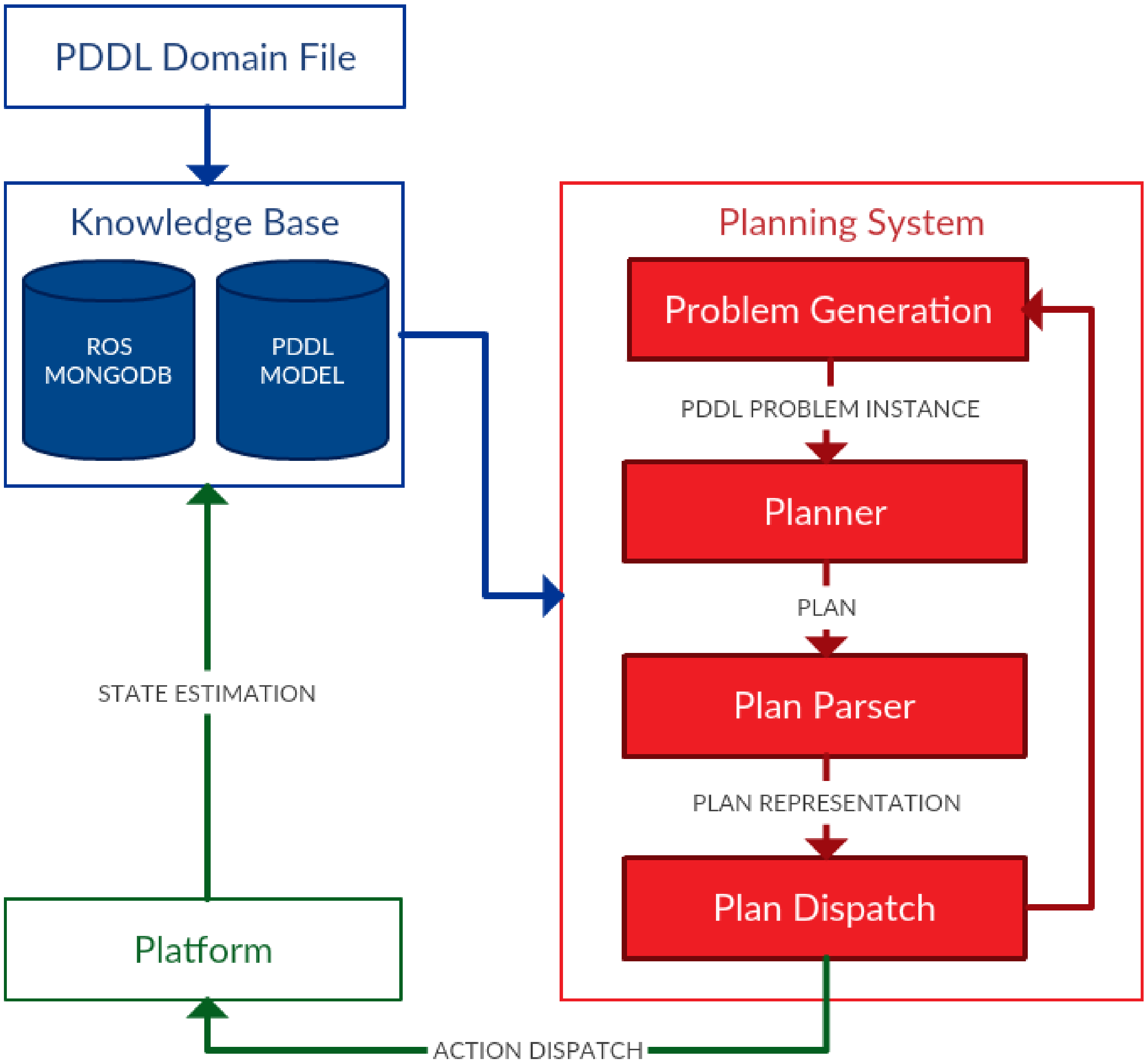
ROSPlan: Default Configuration

Now the system is more complex:
- PDDL model is continuously updated from sensor data.
- problem file is automatically generated.
- the planner generates a plan.
- the plan is dispatched action-by-action.



ROSPlan: Default Configuration

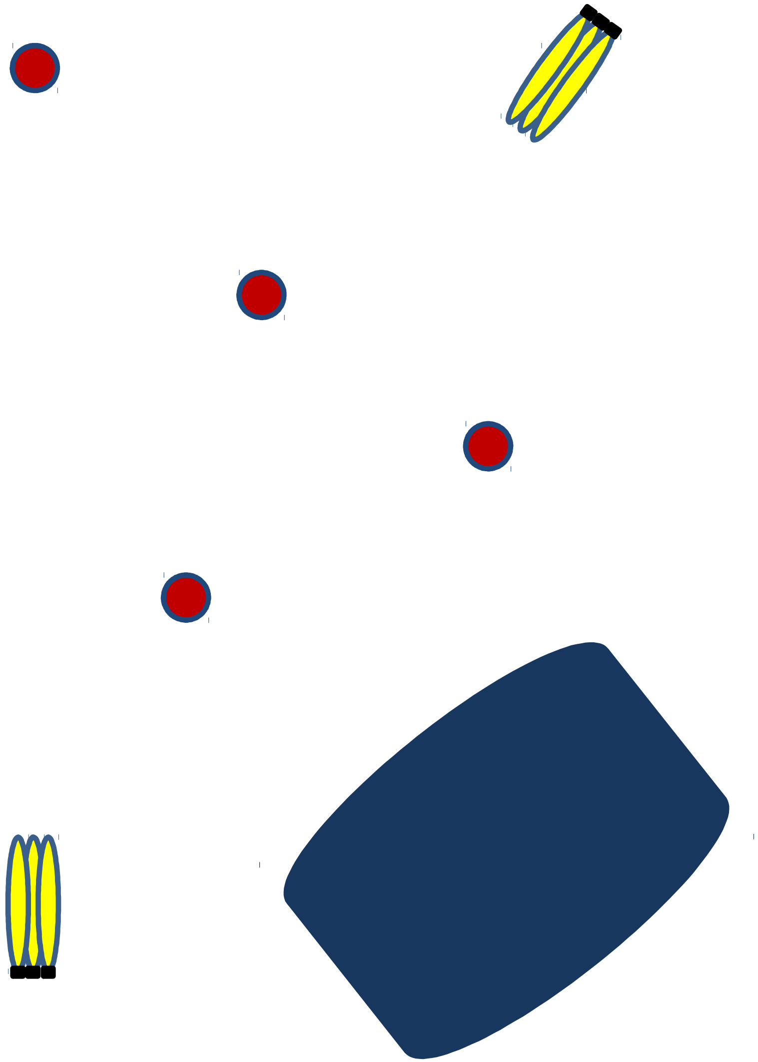
Now the system is more complex:
- PDDL model is continuously updated from sensor data.
- problem file is automatically generated.
- the planner generates a plan.
- the plan is dispatched action-by-action.
- feedback on action success and failure.
- the plan is validated against the current model.



Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

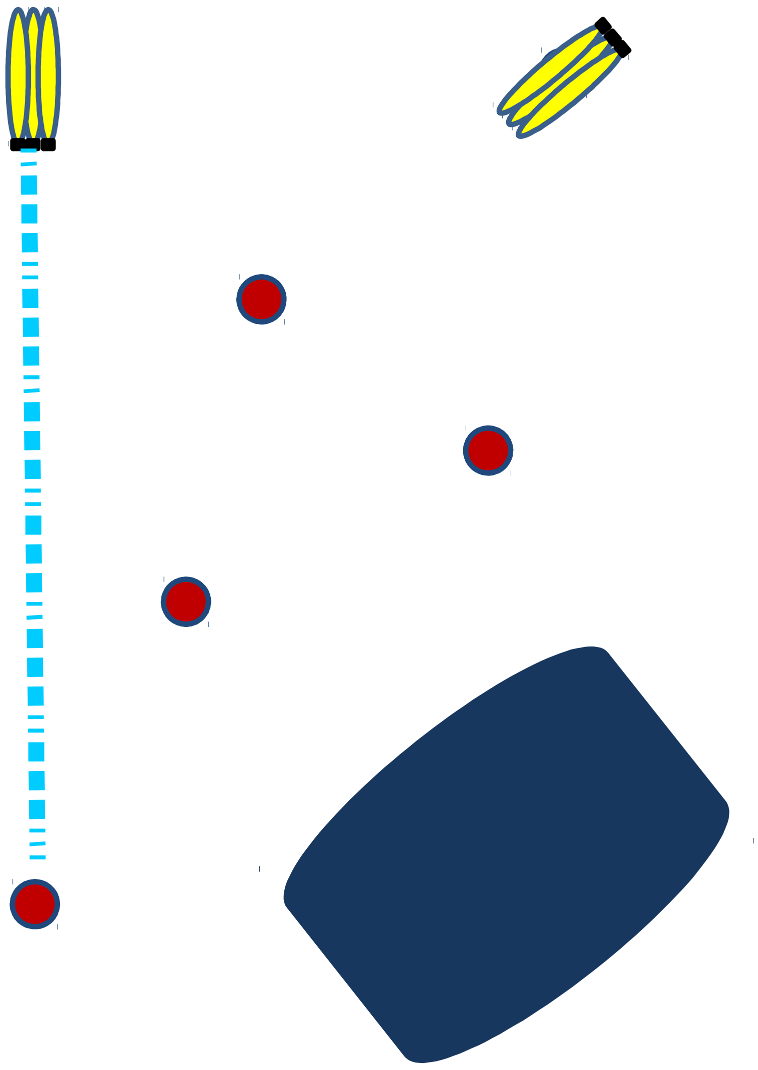
What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?



Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

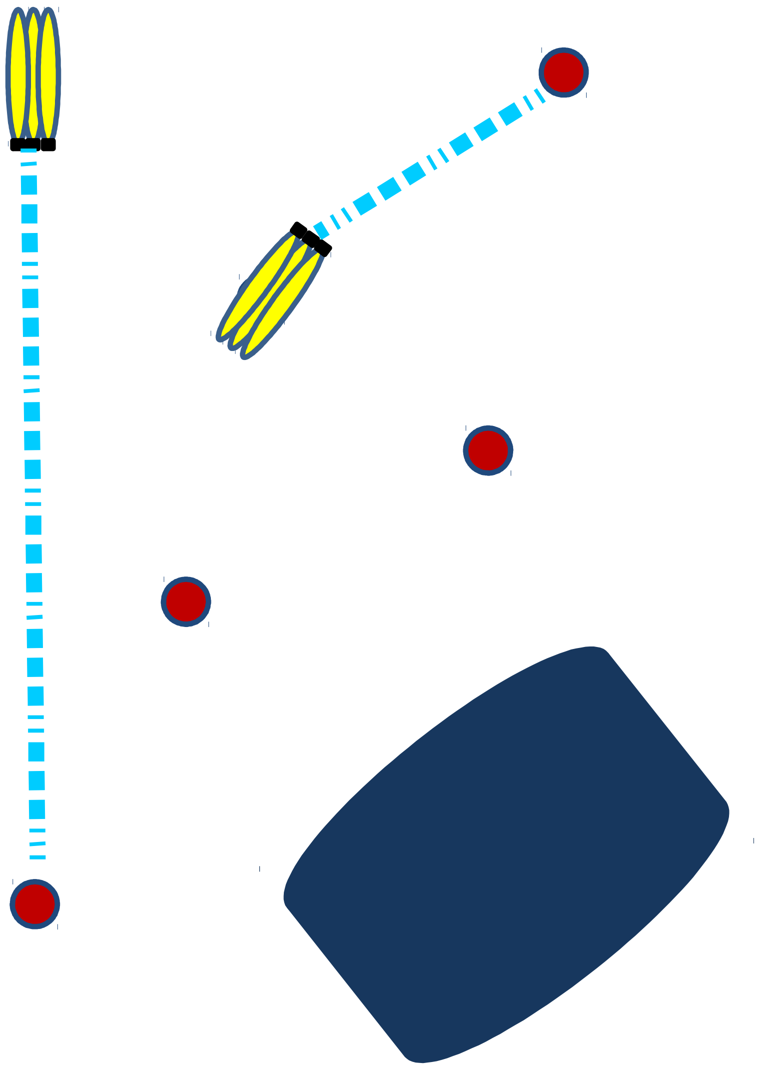
What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?



Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

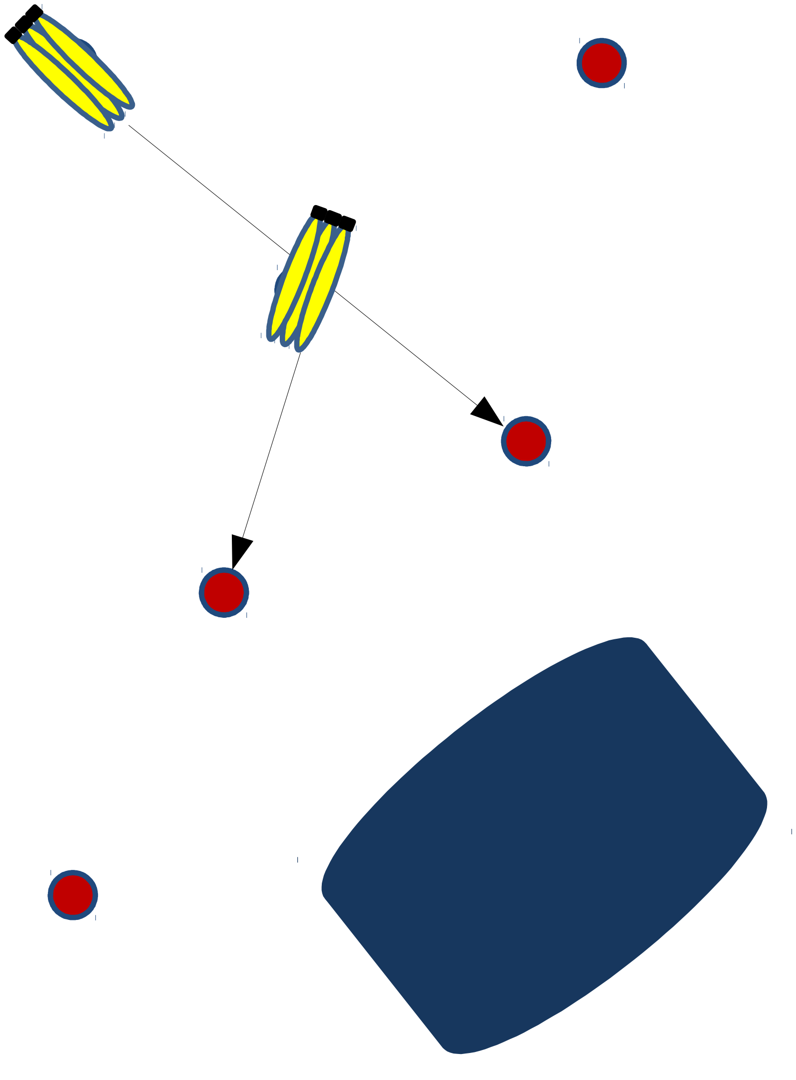
What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?



Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?

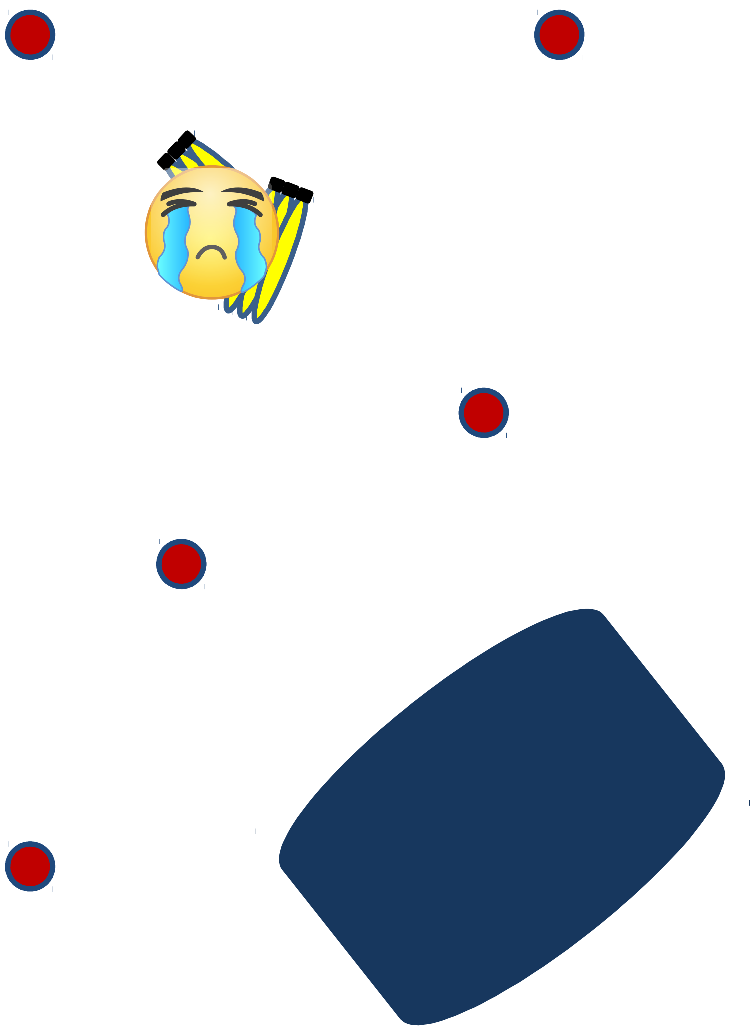


Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?

The plan is not only less efficient, but it may become incorrect and unsafe!



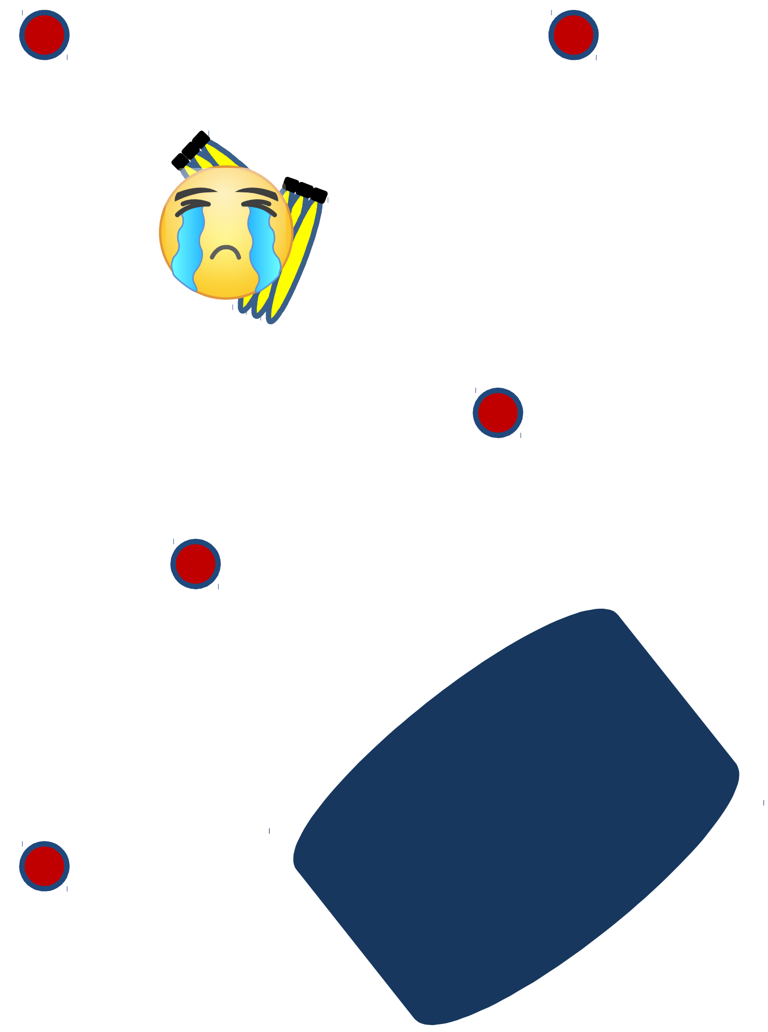
Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

What happens when we add temporal constraints, and try to dispatch the plan as a sequence of actions?

The plan is not only less efficient, but it may become incorrect and unsafe!

The plan execution loop could instead dispatch actions at their *estimated* timestamps.



Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

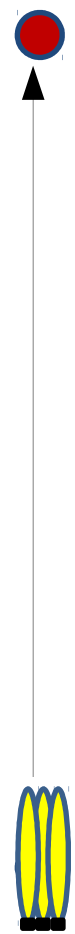
However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: clean_chain wp2) [60.0]

Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.



0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: clean_chain wp2) [60.0]

Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

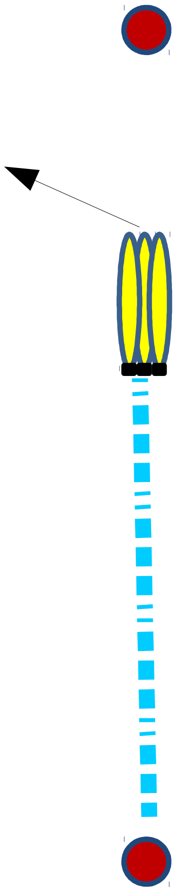
However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

0.000: (goto_waypoint wp1) [10.0]	
10.01: (goto_waypoint wp2) [14.3]	
24.32: clean_chain wp2) [60.0]	

Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.



0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: clean_chain wp2) [60.0]

Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

The plan execution loop could dispatch actions, while respecting the causal ordering between actions.

0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: clean_chain wp2) [60.0]

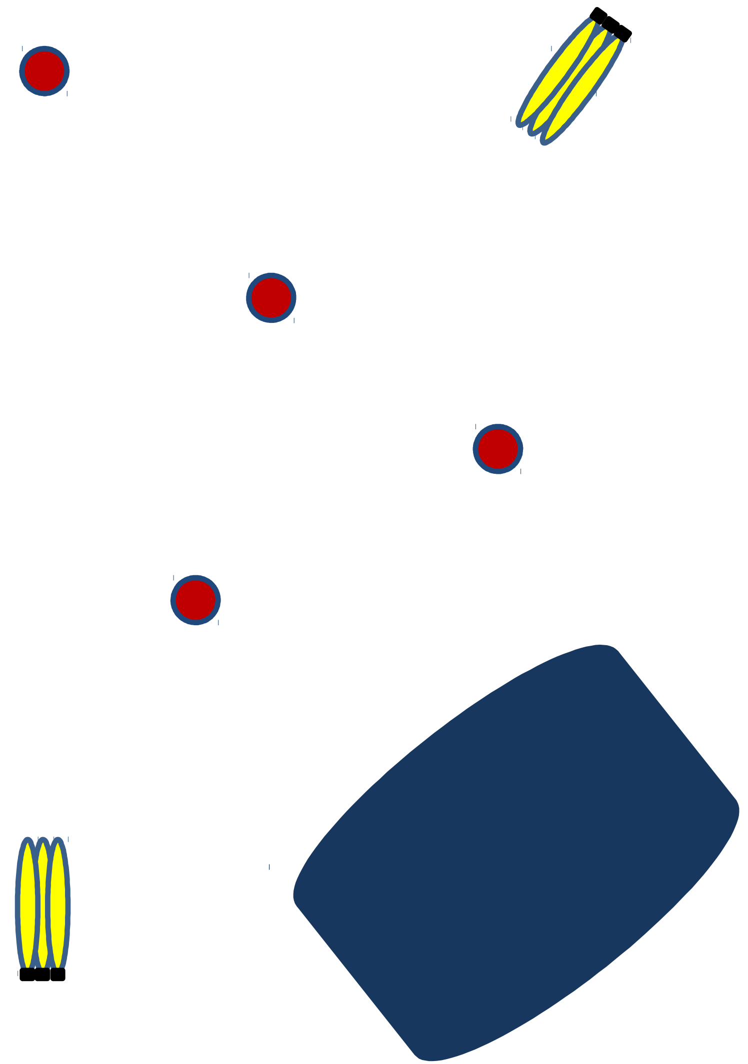
Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

The plan execution loop could dispatch actions, while respecting the causal ordering between actions.

However, some plans require *temporal coordination* between actions, and the controllable durations might be very far apart.



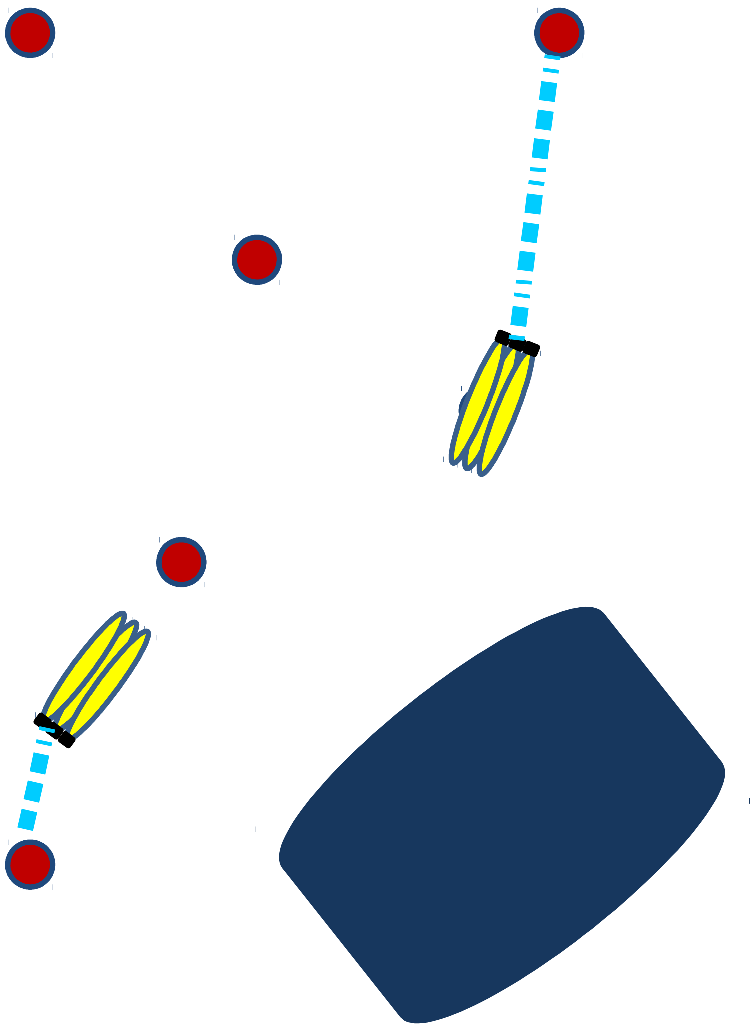
Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

The plan execution loop could dispatch actions, while respecting the causal ordering between actions.

However, some plans require *temporal coordination* between actions, and the controllable durations might be very far apart.



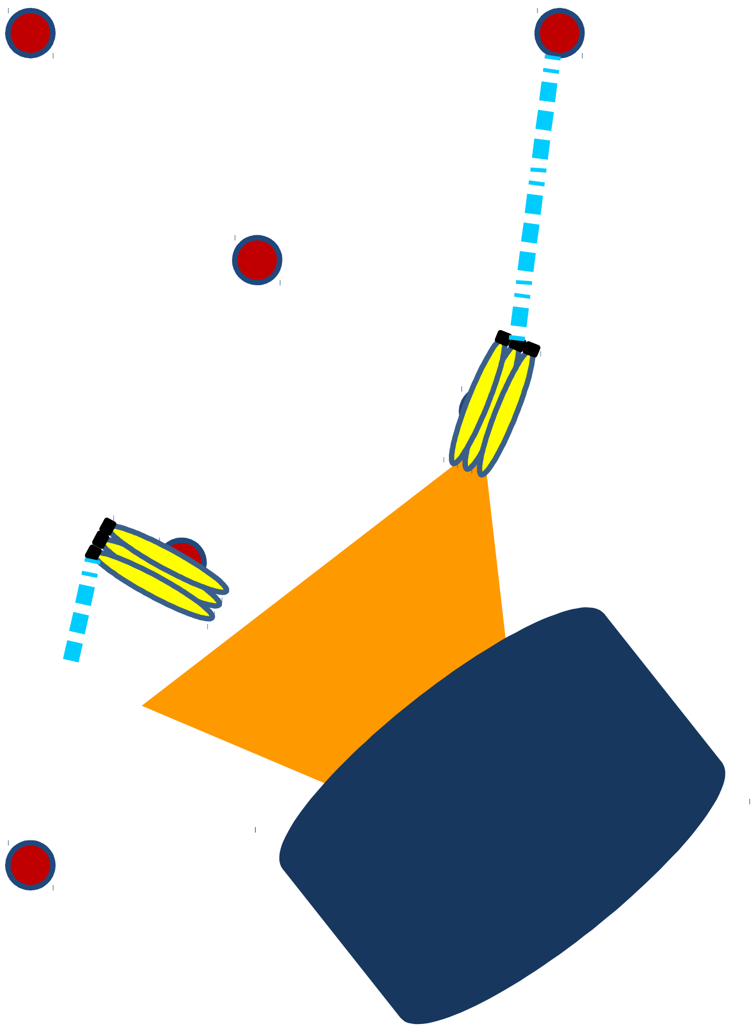
Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

The plan execution loop could dispatch actions, while respecting the causal ordering between actions.

However, some plans require *temporal coordination* between actions, and the controllable durations might be very far apart.



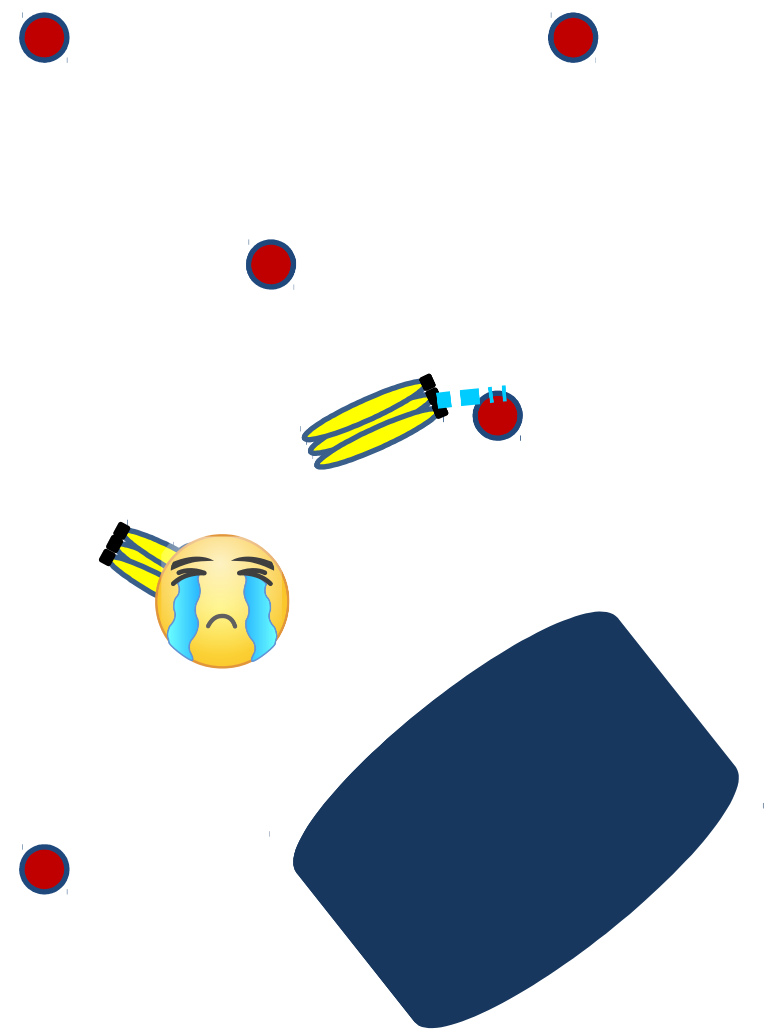
Temporal Constraints

The plan execution loop could instead dispatch actions at their *estimated* timestamps.

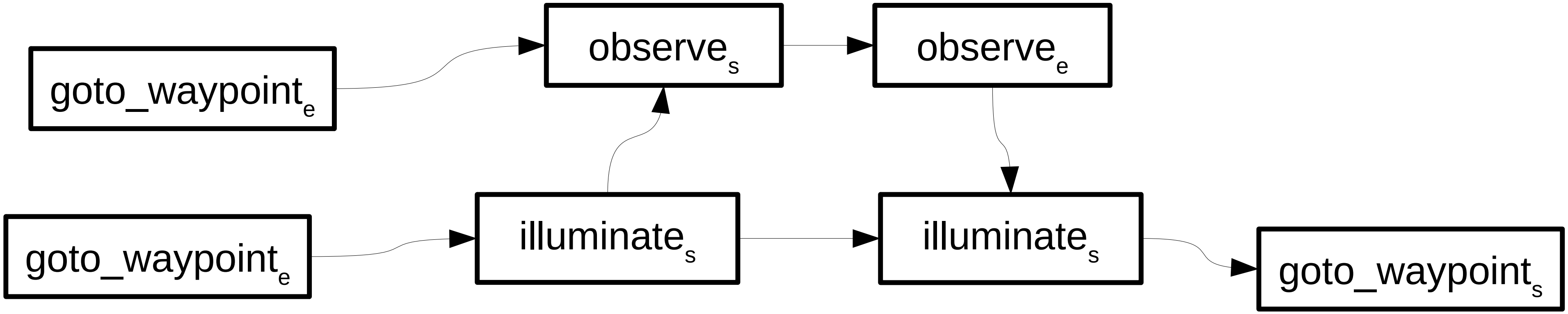
However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.

The plan execution loop could dispatch actions, while respecting the causal ordering between actions.

However, some plans require *temporal coordination* between actions, and the controllable durations might be very far apart.



Temporal Constraints

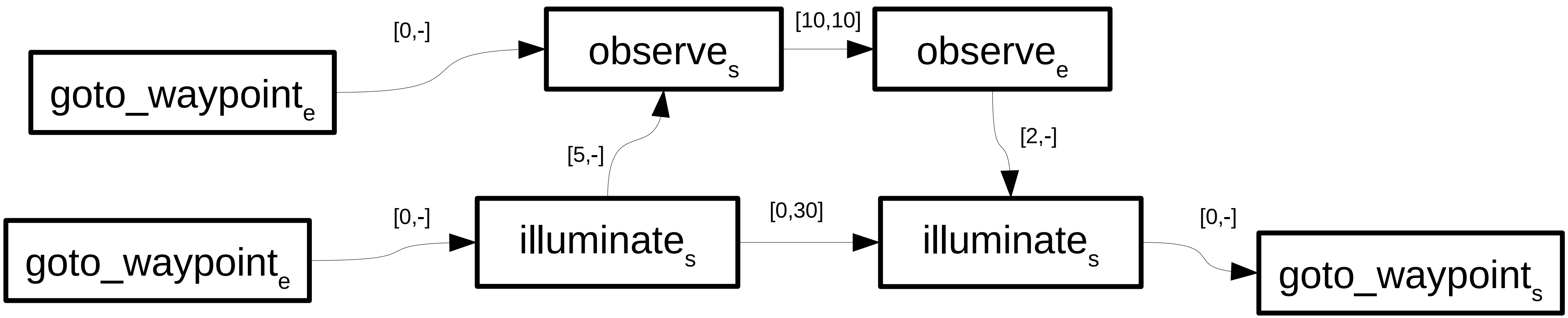


The temporal plan in which every action and duration can be controlled could be represented as a Simple Temporal Network.

The real upper and lower bounds, and ordering constraints on actions can be represented explicitly.

With this representation, the system is able to dispatch actions at times to maintain the consistency of the STN.

Temporal Constraints

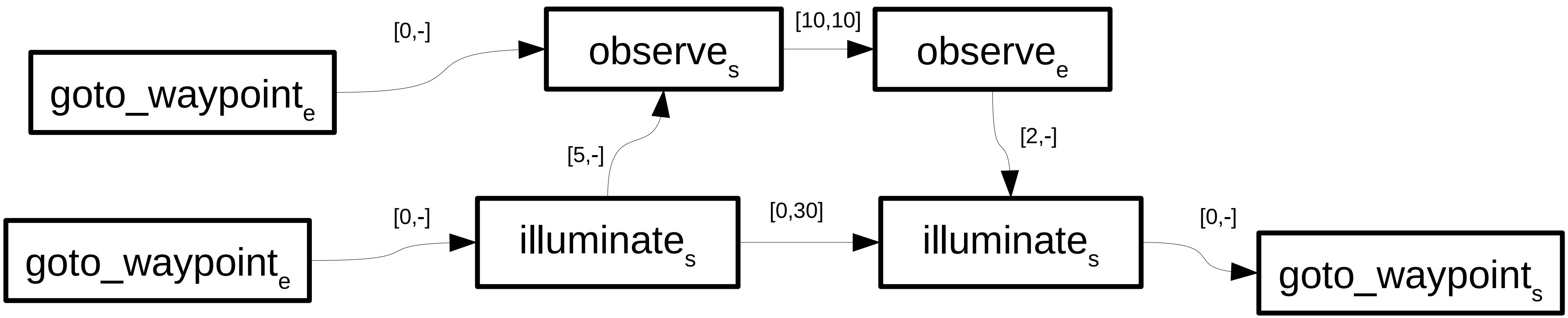


The temporal plan in which every action and duration can be controlled could be represented as a Simple Temporal Network.

The real upper and lower bounds, and ordering constraints on actions can be represented explicitly.

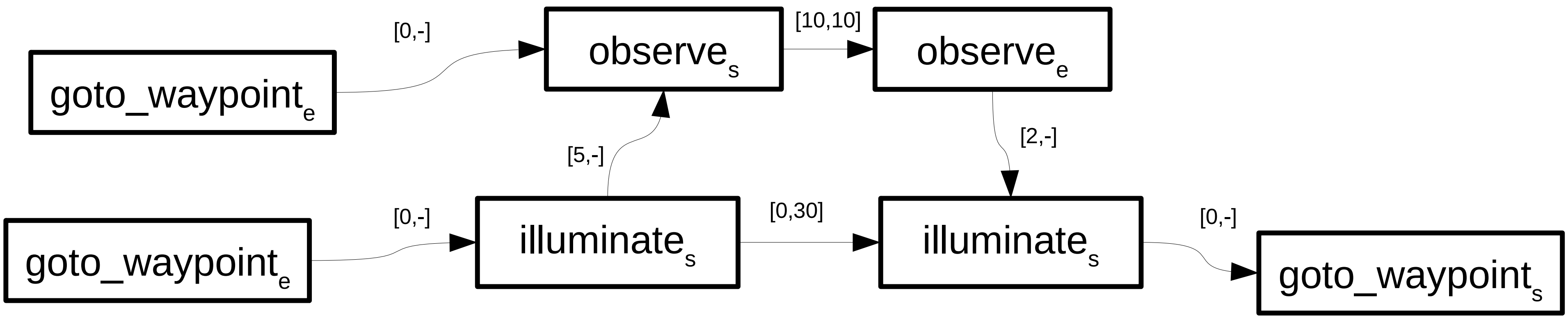
With this representation, the system is able to dispatch actions at times to maintain the consistency of the STN.

Temporal Constraints



The temporal plan with uncontrollable durations can also be represented as a *Contingent Temporal Constraint Network* (C-TCN). *[Vidal & Fargier 1999]*

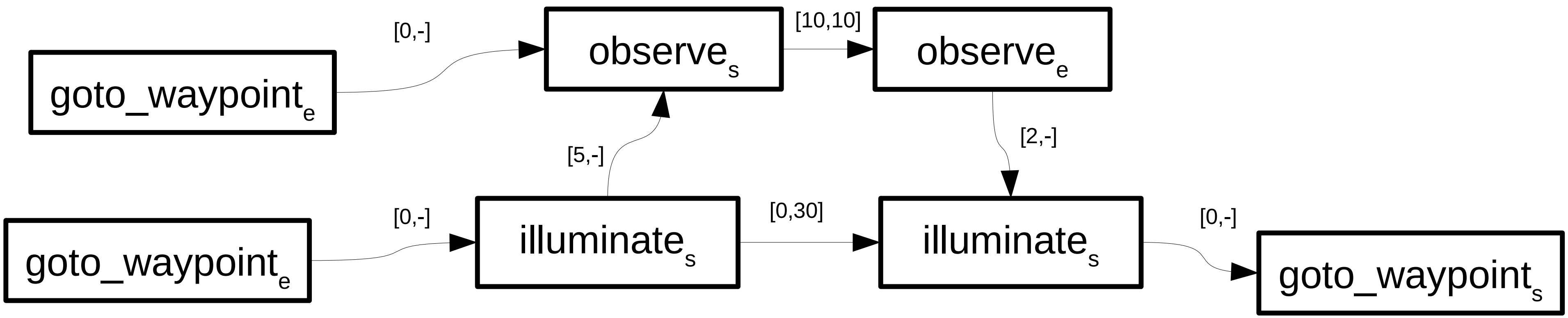
Temporal Constraints



The temporal plan with uncontrollable durations can also be represented as a *Contingent Temporal Constraint Network* (C-TCN). *[Vidal & Fargier 1999]*

The time-points of a C-TCN are divided into *activated* time-points whose dispatch time can be chosen by the agent, and *received* time-points whose time is unpredictable.

Temporal Constraints



The temporal plan with uncontrollable durations can also be represented as a *Contingent Temporal Constraint Network* (C-TCN)*. [Vidal & Fargier 1999]*

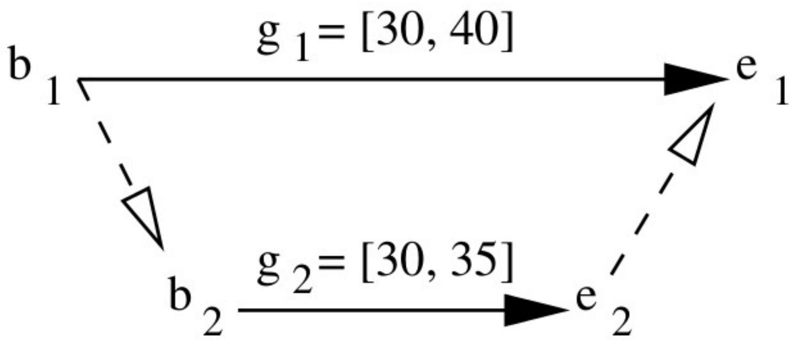
The time-points of a C-TCN are divided into *activated* time-points whose dispatch time can be chosen by the agent, and *received* time-points whose time is unpredictable.

The *Simple Temporal Problem under Uncertainty* (STPU) described by a C-TCN might be strongly, weakly, or dynamically controllable.

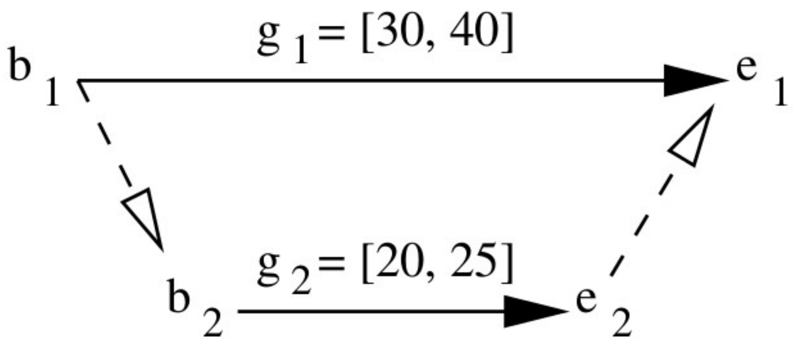
[Ciamatti, Micheli et al. 2016]

STPUs: Strong controllability

An STPU is strongly controllable iff:
- the agent can commit to a time for all activated time-points,
- such that for any possible time for received time points,
- the temporal constraints are not violated.



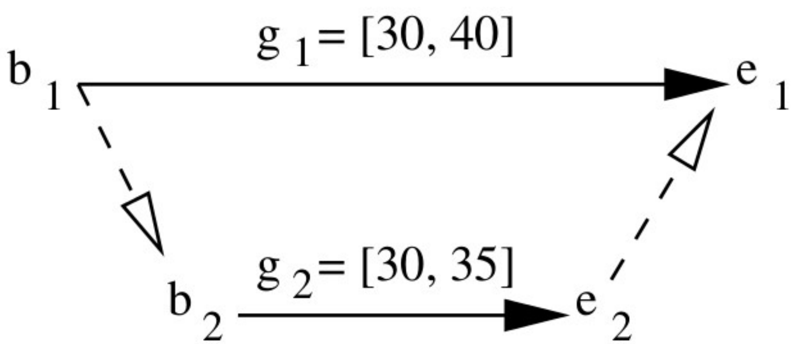
(a)



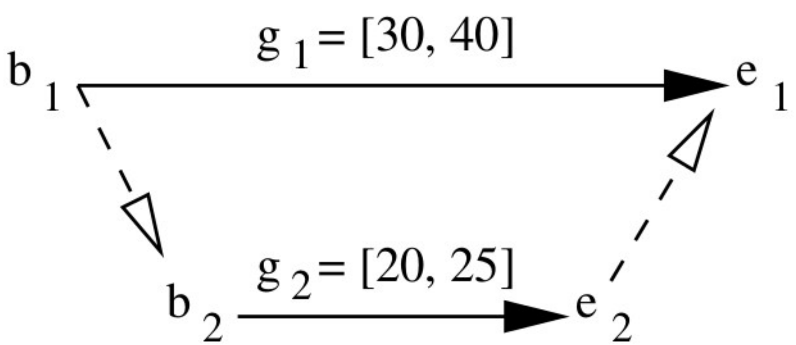
(b)

STPUs: Strong controllability

An STPU is strongly controllable iff:
- the agent can commit to a time for all activated time-points,
- such that for any possible time for received time points,
- the temporal constraints are not violated.



(a)

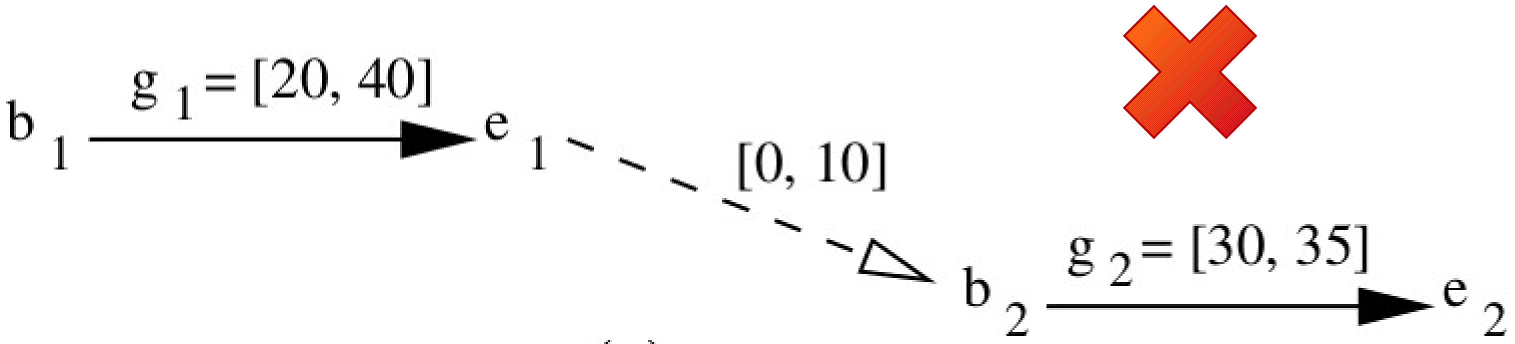


(b)

Setting *t(b1) == t(b2)* will always obey the temporal constraints.

STPUs: Strong controllability

An STPU is strongly controllable iff:
- the agent can commit to a time for all activated time-points,
- such that for any possible time for received time points,
- the temporal constraints are not violated.



(c)

The STPU is not strongly controllable, but it is obviously executable.
We need dynamic controllability.

STPUs: Dynamic controllability

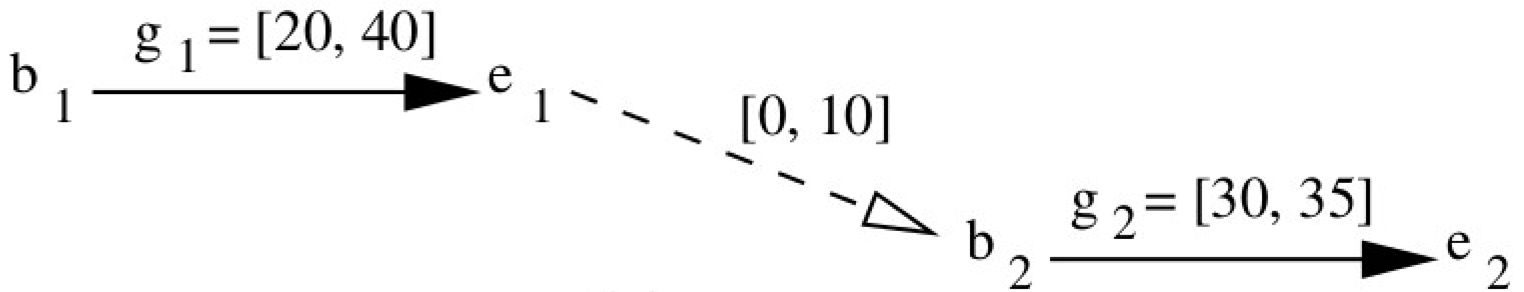
An STPU is dynamically controllable iff:
- at any point in time, the execution so far is ensured to extend to a complete solution such that the temporal constraints are not violated.

In this case, the agent does not have to commit to a time for any activated time points in advance.

STPUs: Dynamic controllability

An STPU is dynamically controllable iff:
- at any point in time, the execution so far is ensured to extend to a complete solution such that the temporal constraints are not violated.

In this case, the agent does not have to commit to a time for any activated time points in advance.

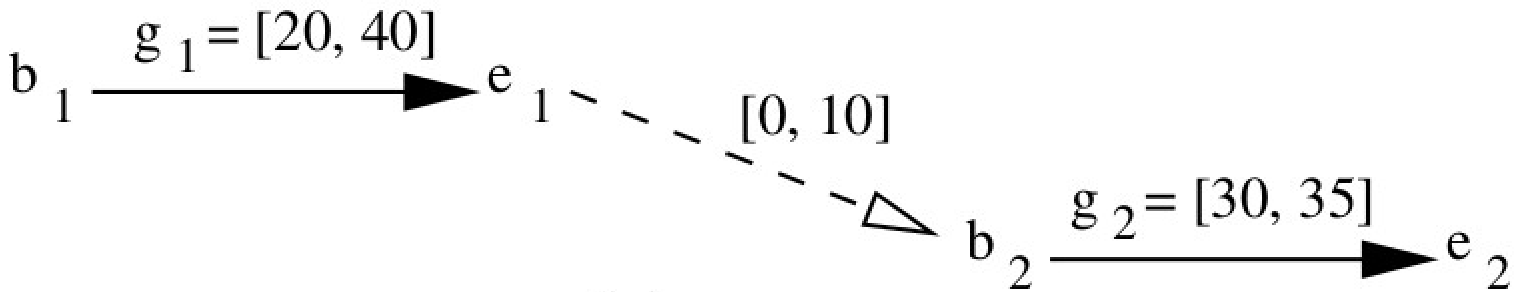


(c)

STPUs: Dynamic controllability

Not all problems will have solutions which have any kind of controllability. This does not mean they are impossible.

To reason about these kinds of issues we need to use a plan representation sufficient to capture the controllable and uncontrollable durations, causal orderings, and temporal constraints.



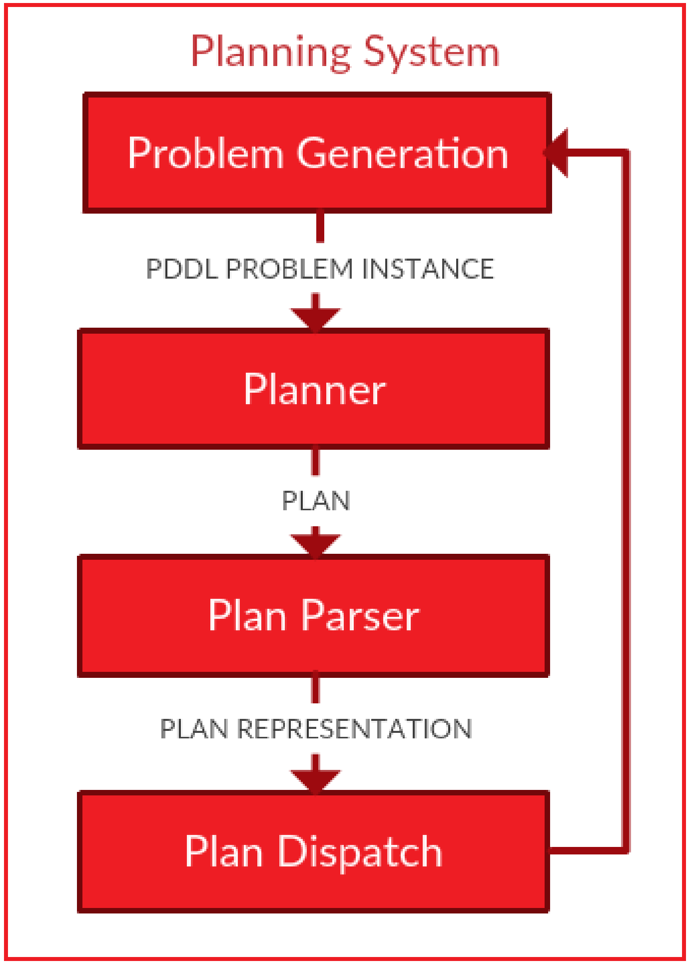
(c)

Plan dispatch in ROSPlan

To reason about these kinds of issues we need to use a plan representation sufficient to capture the controllable and uncontrollable durations, causal orderings, and temporal constraints.

The representation of a plan is coupled with the choice of dispatcher.

The problem generation and planner are not *necessarily* bound by the choice of representation.



Plan Execution 3: Conditional Dispatch

Uncertainty and lack of knowledge is a huge part of AI Planning for Robotics.

- Actions might fail or succeed.
- The effects of an action can be non-deterministic.
- The environment is dynamic and changing.
- The environment is often initially full of unknowns.

The domain model is *always* incomplete as well as inaccurate.

Uncertainty in AI Planning

- The environment is dynamic and changing.
- The environment is often initially full of unknowns.

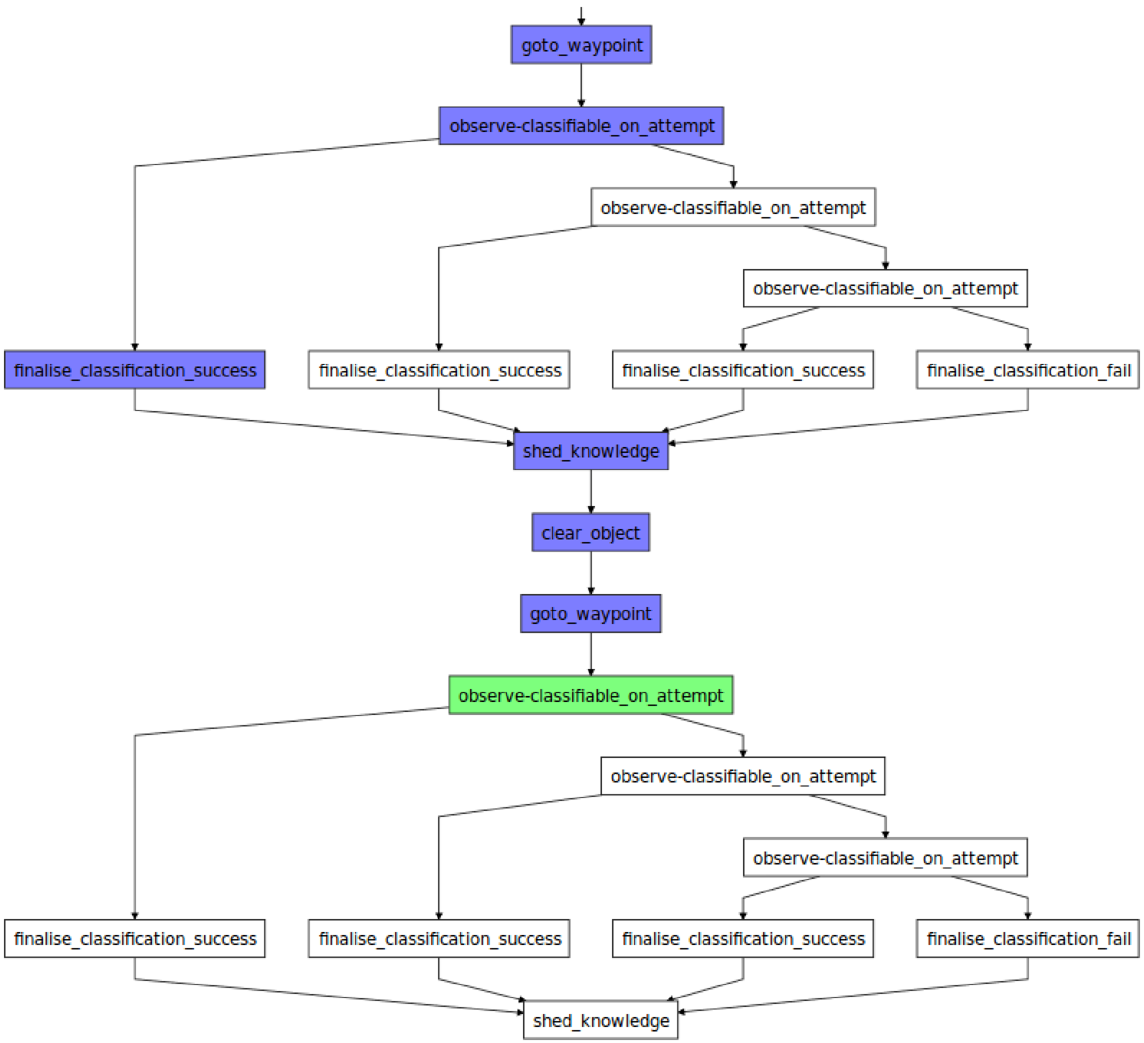
Uncertainty in AI Planning

Some uncertainty can be handled at planning time:

- Fully-Observable Non-deterministic planning.

- Partially-observable Markov decision Process.

- Conditional Planning with Contingent Planners. (e.g. ROSPlan with Contingent-FF)



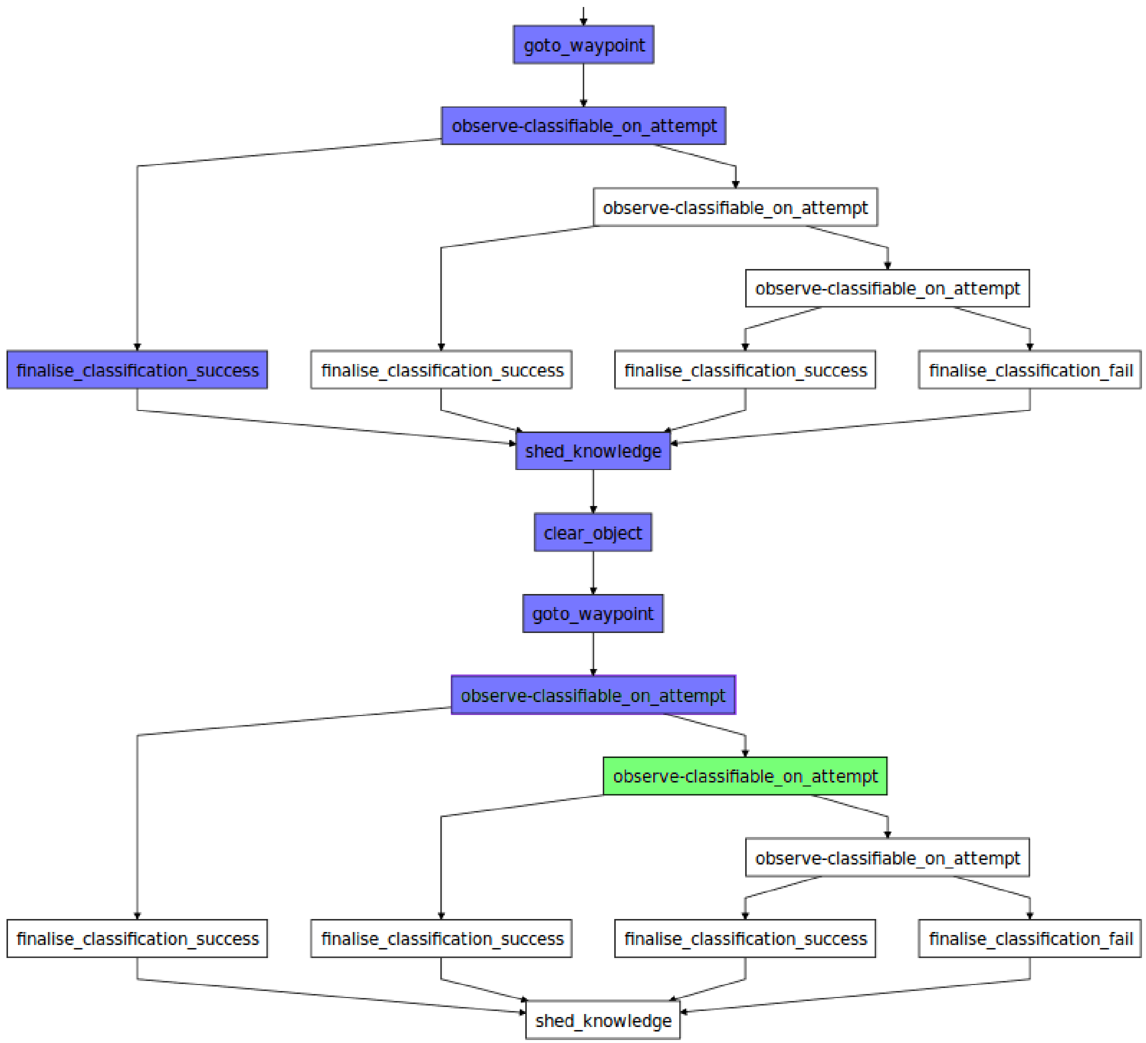
Uncertainty in AI Planning

Some uncertainty can be handled at planning time:

- Fully-Observable Non-deterministic planning.

- Partially-observable Markov decision Process.

- Conditional Planning with Contingent Planners. (e.g. ROSPlan with Contingent-FF)



Uncertainty in AI Planning

Some uncertainty can be handled at planning time:

- Fully-Observable Non-deterministic planning.

- Partially-observable Markov decision Process.

- Conditional Planning with Contingent Planners. (e.g. ROSPlan with Contingent-FF)

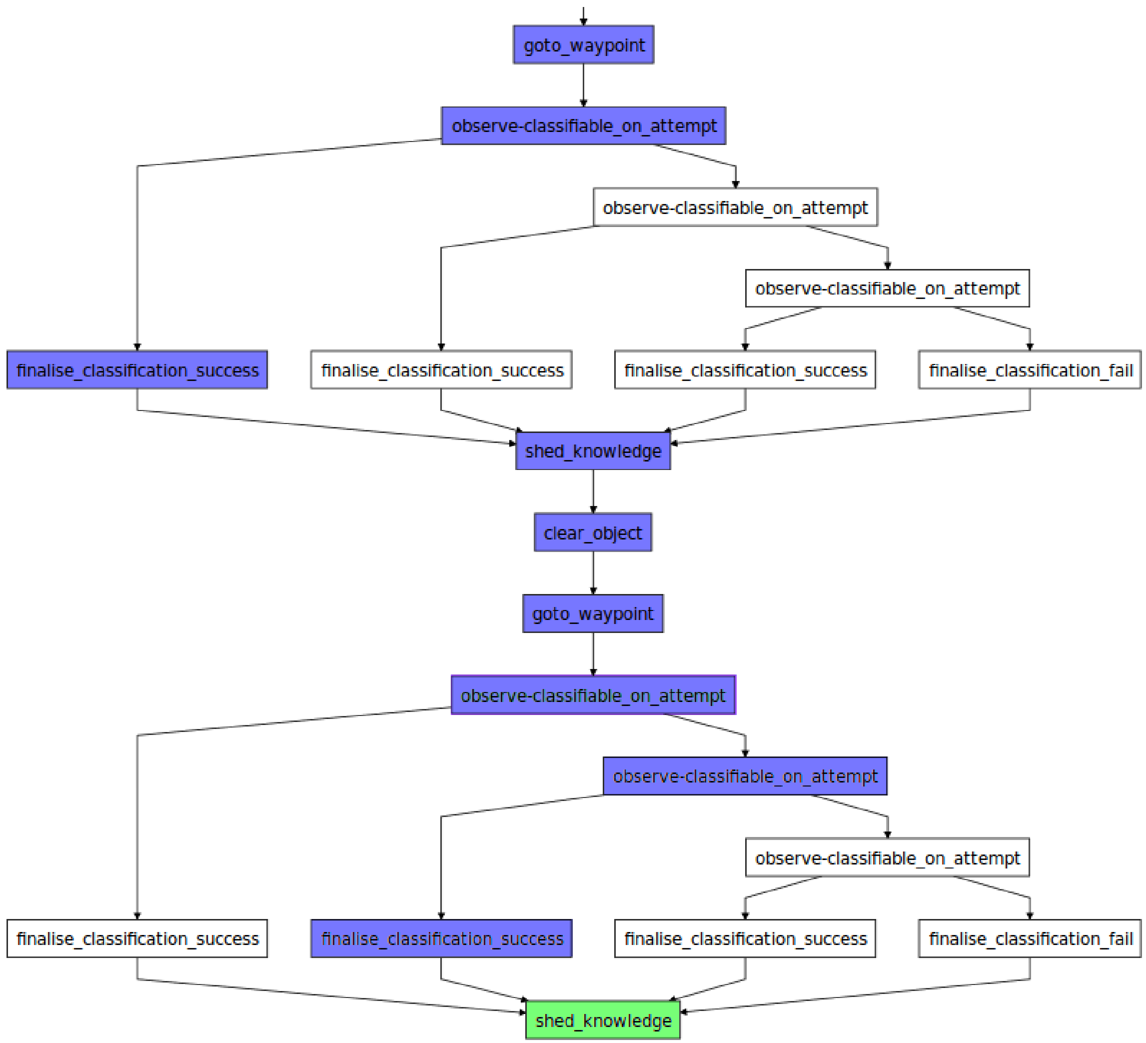
Uncertainty in AI Planning

Some uncertainty can be handled at planning time:

- Fully-Observable Non-deterministic planning.

- Partially-observable Markov decision Process.

- Conditional Planning with Contingent Planners. (e.g. ROSPlan with Contingent-FF)



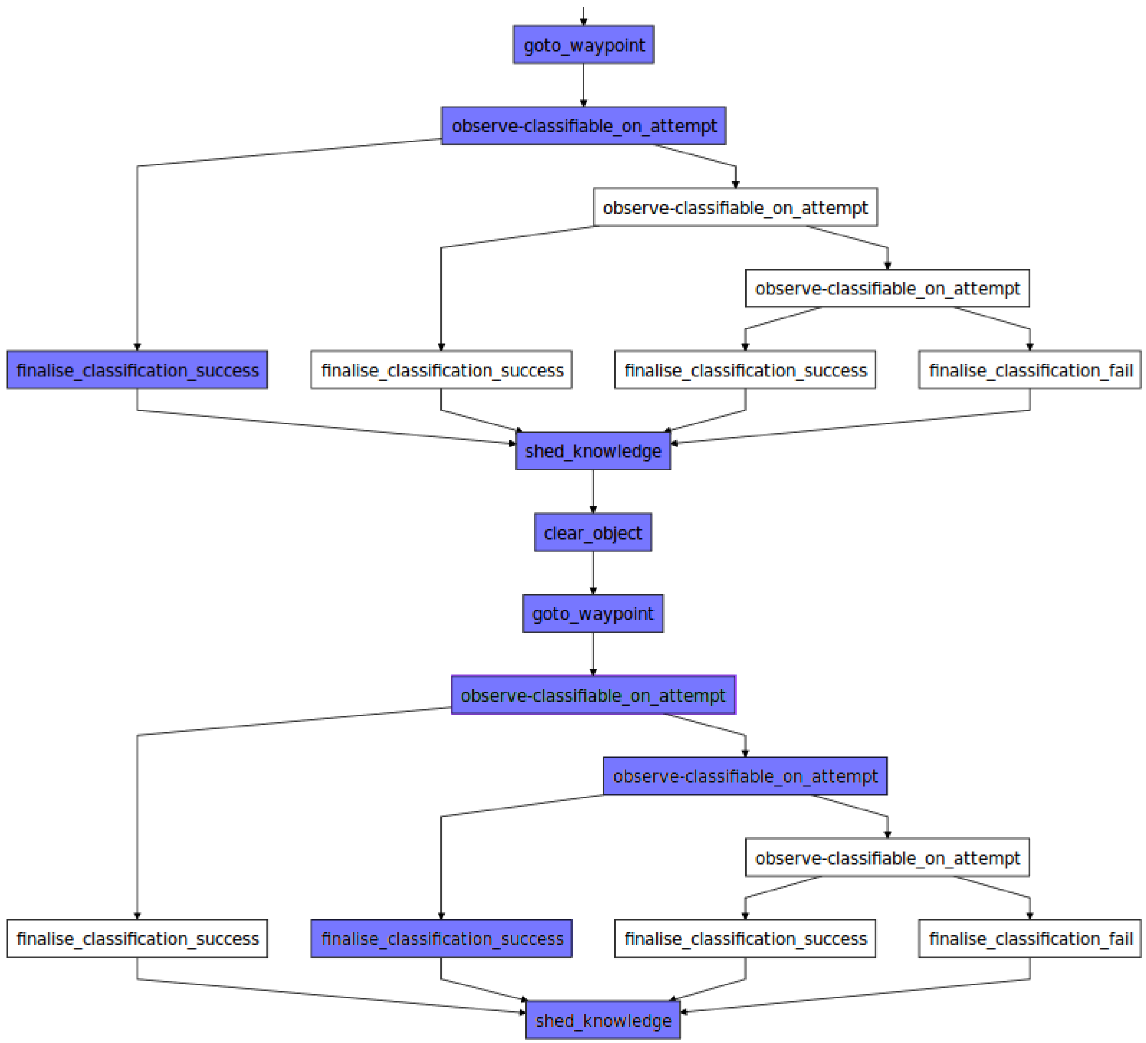
Uncertainty in AI Planning

Some uncertainty can be handled at planning time:

- Fully-Observable Non-deterministic planning.

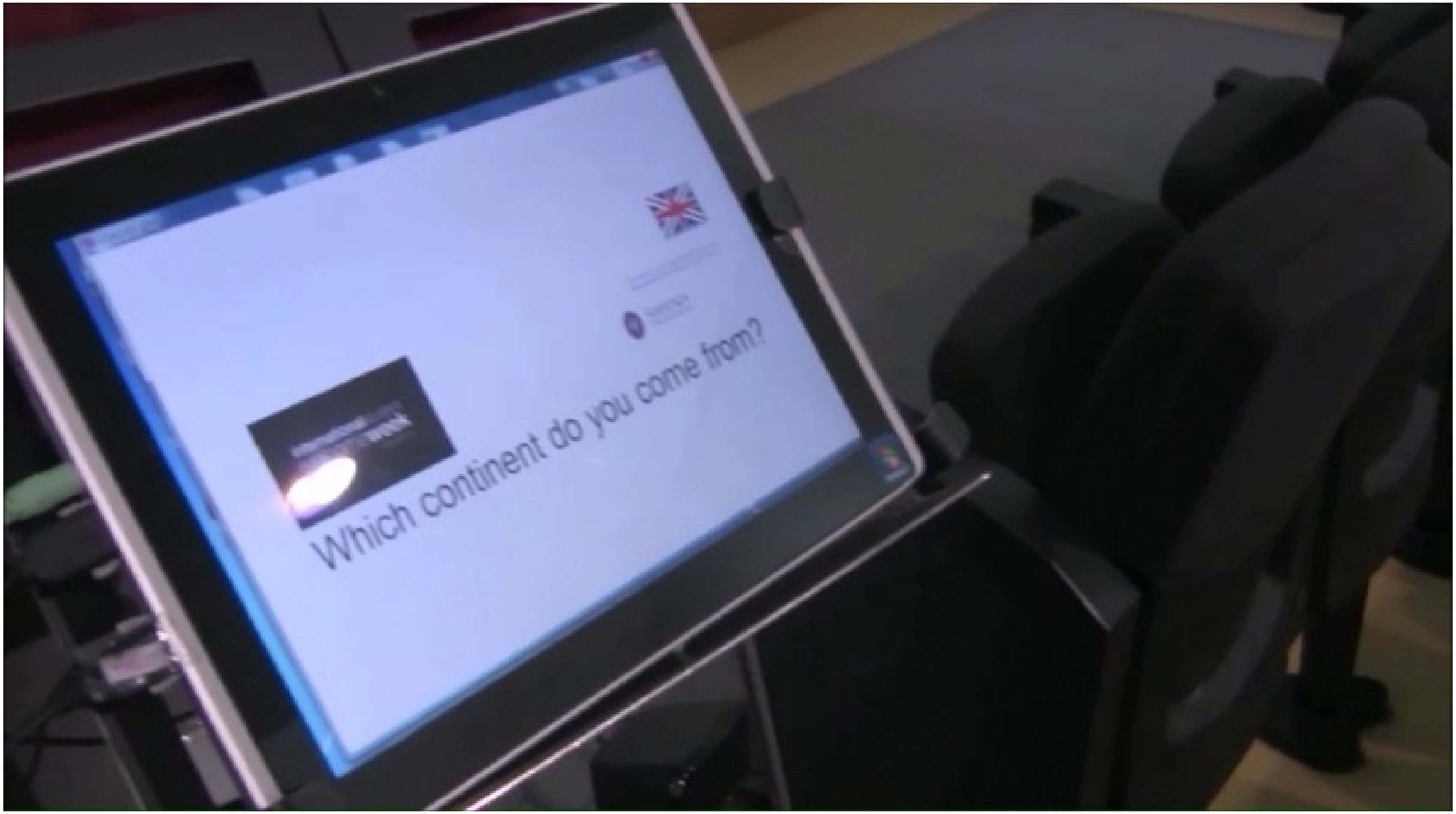
- Partially-observable Markov decision Process.

- Conditional Planning with Contingent Planners. (e.g. ROSPlan with Contingent-FF)



Uncertainty in AI Planning

Human Robot Interaction is filled with uncertainties.



Plan Execution 4: Temporal and Conditional Dispatch together

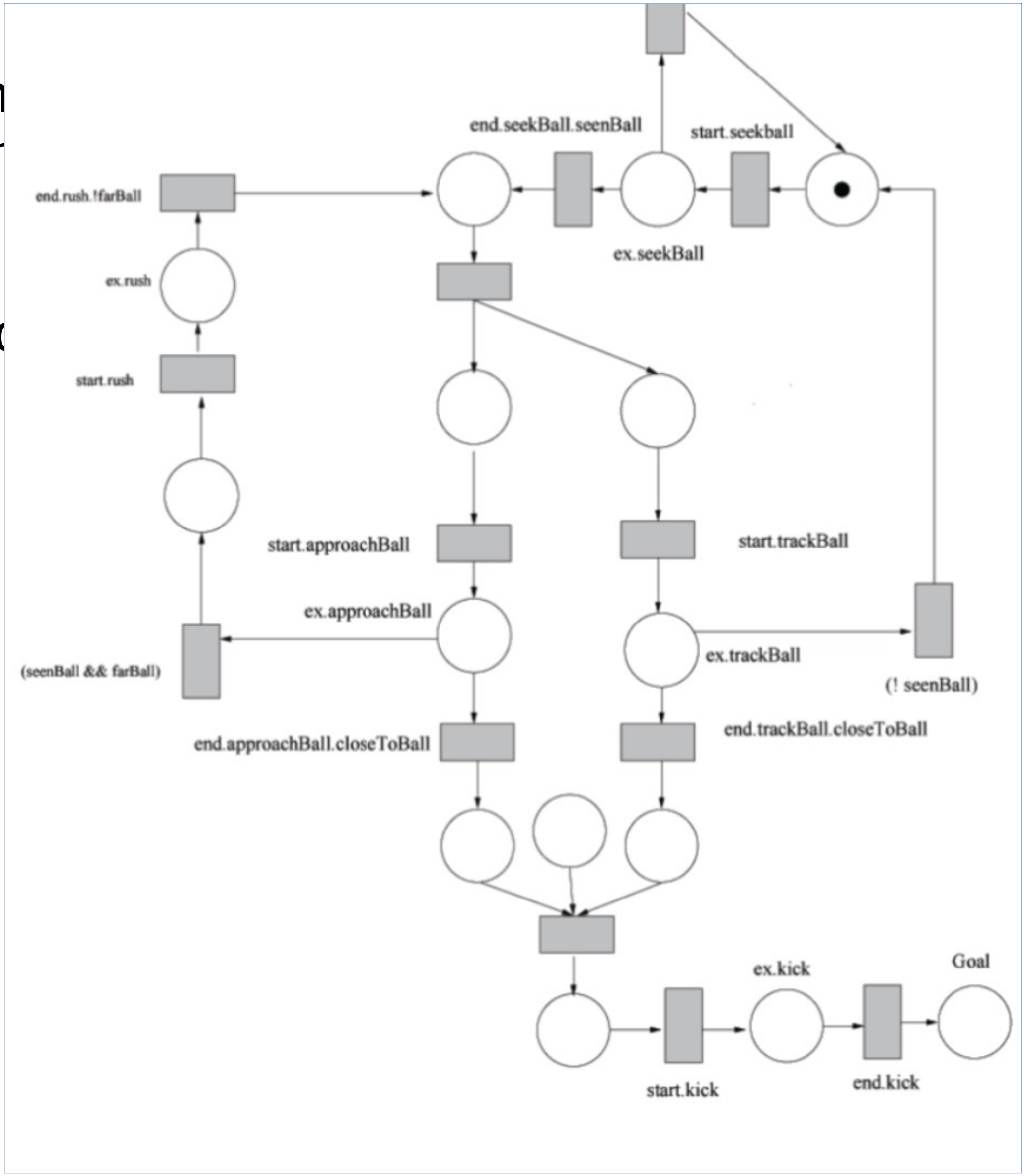
Robotics domains require a combination of temporal and conditional reasoning. Combining these two kinds of uncertainty can result in very complex structures.

There are plan formalisms designed to describe these, e.g.:
- GOLOG plans. *[Claßen et al., 2012]*
- Petri-Net Plans. *[Ziparo et al. 2011]*

Plan Execution 4: Temporal and Conditional Dispatch together

Robotics domains require a combin
reasoning. Combining these two kir
complex structures.

There are plan formalisms designed
- GOLOG plans. *[Claßen et al., 2012]*
- Petri-Net Plans. *[Ziparo et al. 2011]*



Plan Execution 4: Temporal and Conditional Dispatch together

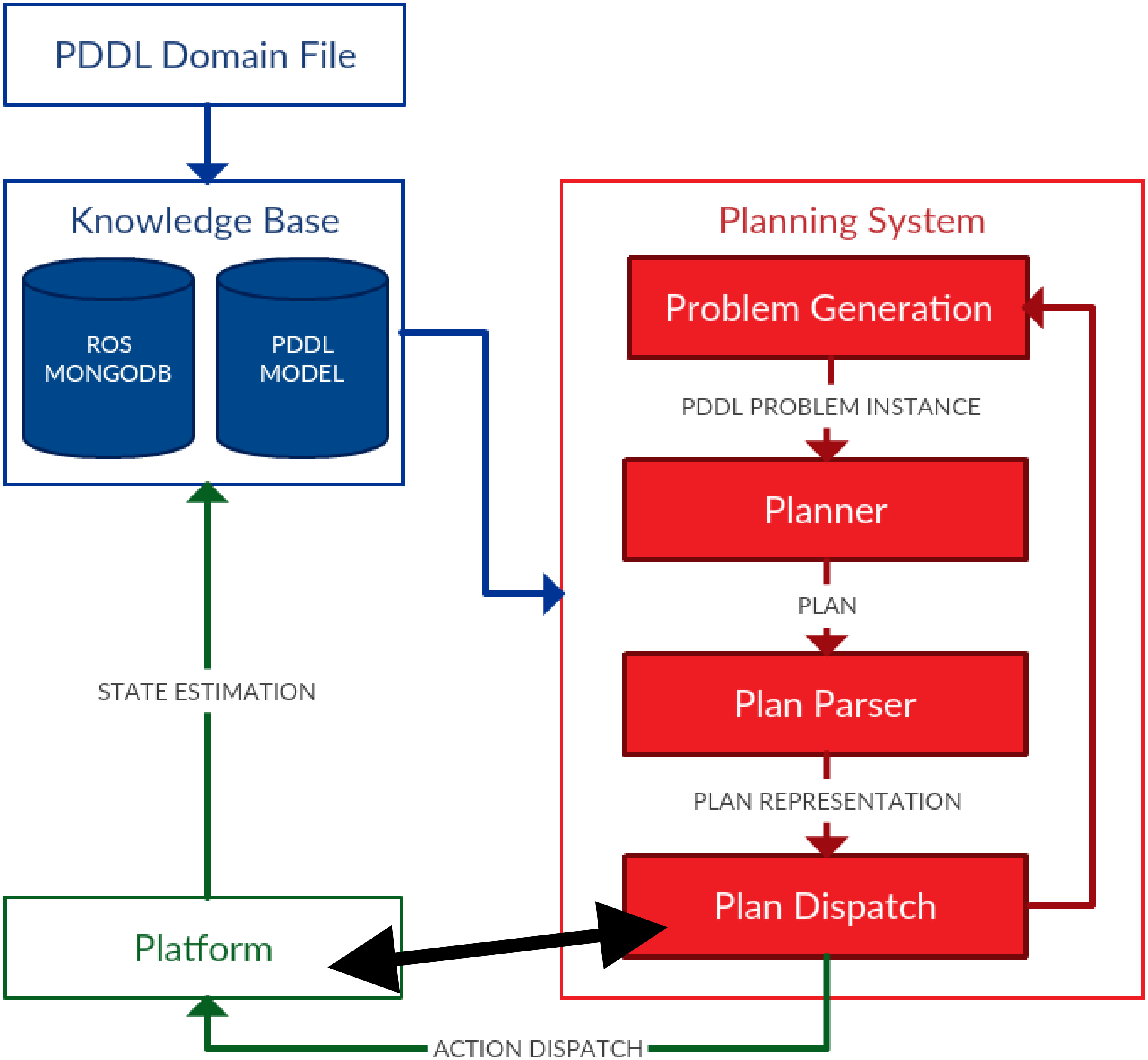
Robotics domains require a combination of temporal and conditional reasoning. Combining these two kinds of uncertainty can result in very complex structures.

There are plan formalisms designed to describe these, e.g.:
- GOLOG plans. *[Claßen et al., 2012]*
- Petri-Net Plans. *[Ziparo et al. 2011]*

ROSPlan is integrated with the PNPRos library for the representation and execution of Petri-Net plans. *[Sanelli et al. 2017]*

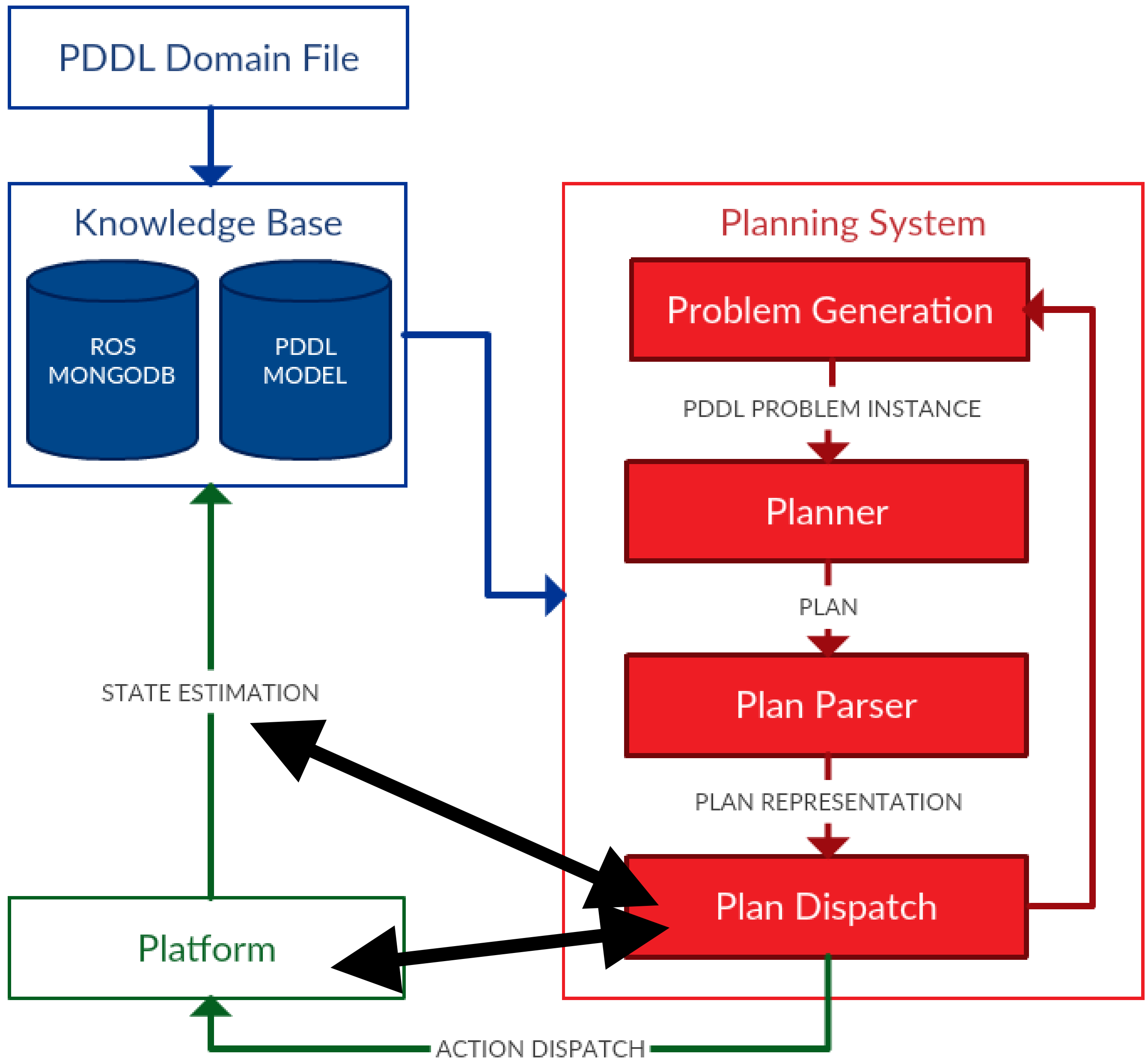
Summary of Very Simple Plan Execution

Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.



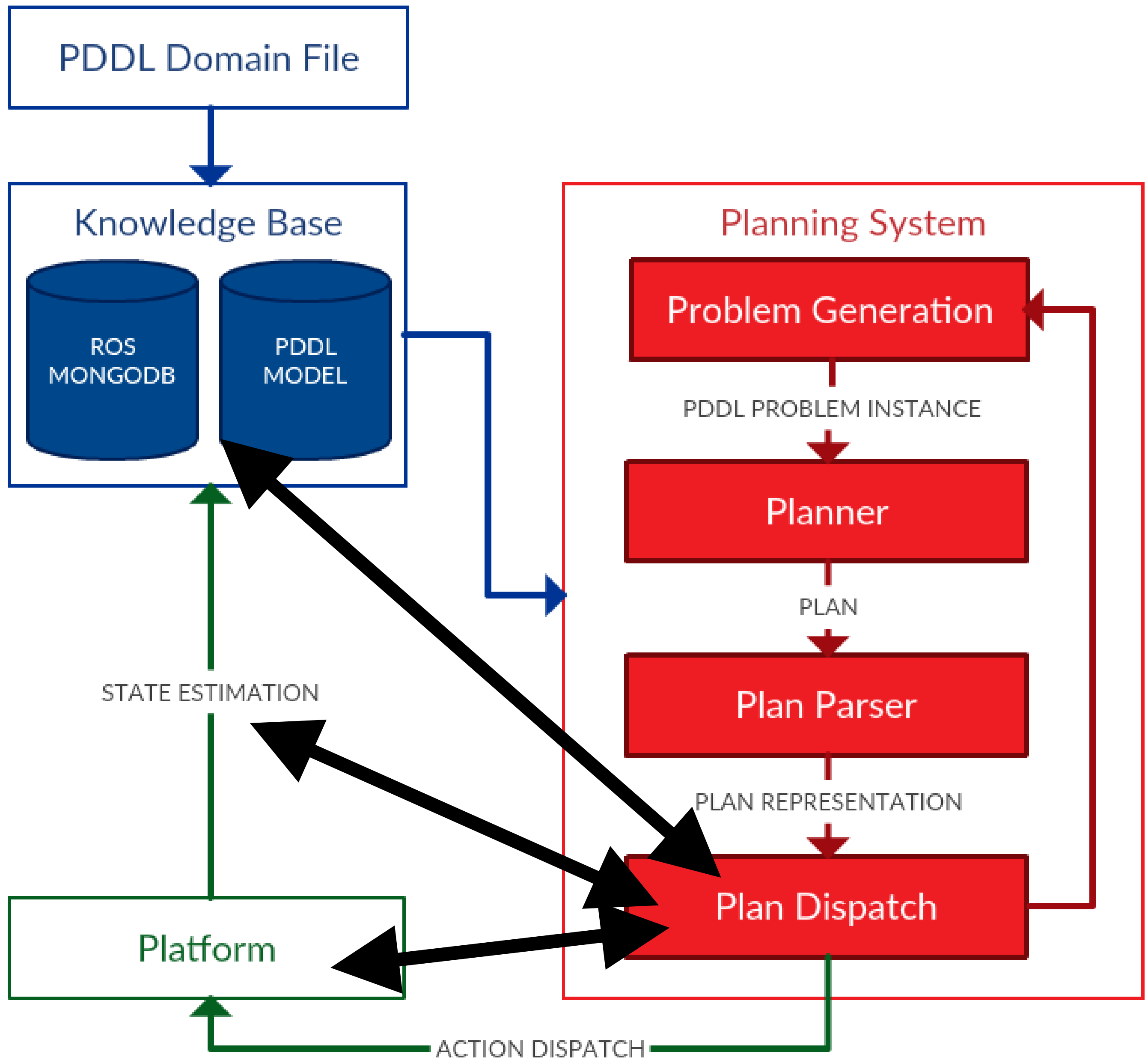
Summary of Very Simple Plan Execution

Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.



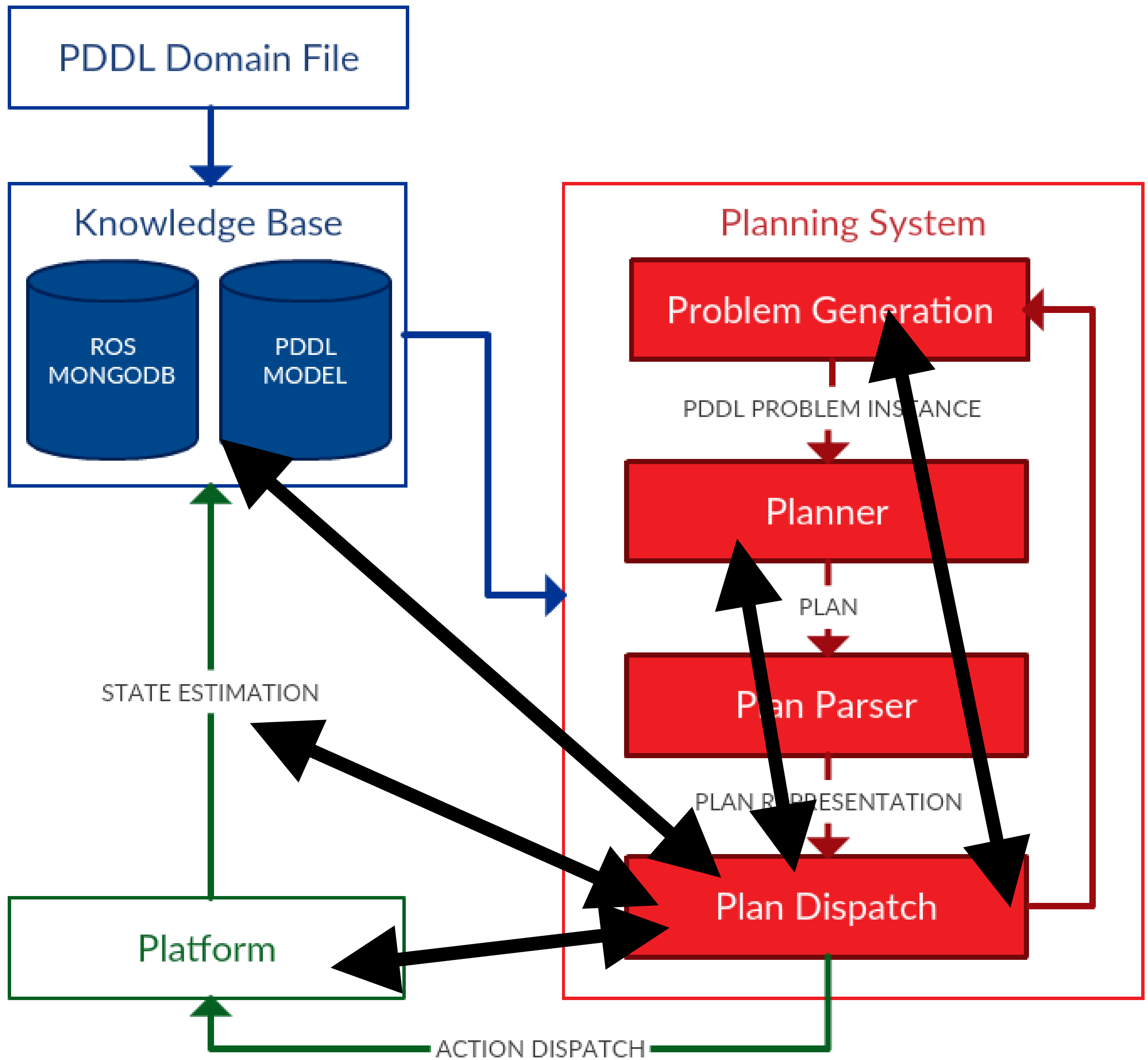
Summary of Very Simple Plan Execution

Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.



Summary of Very Simple Plan Execution

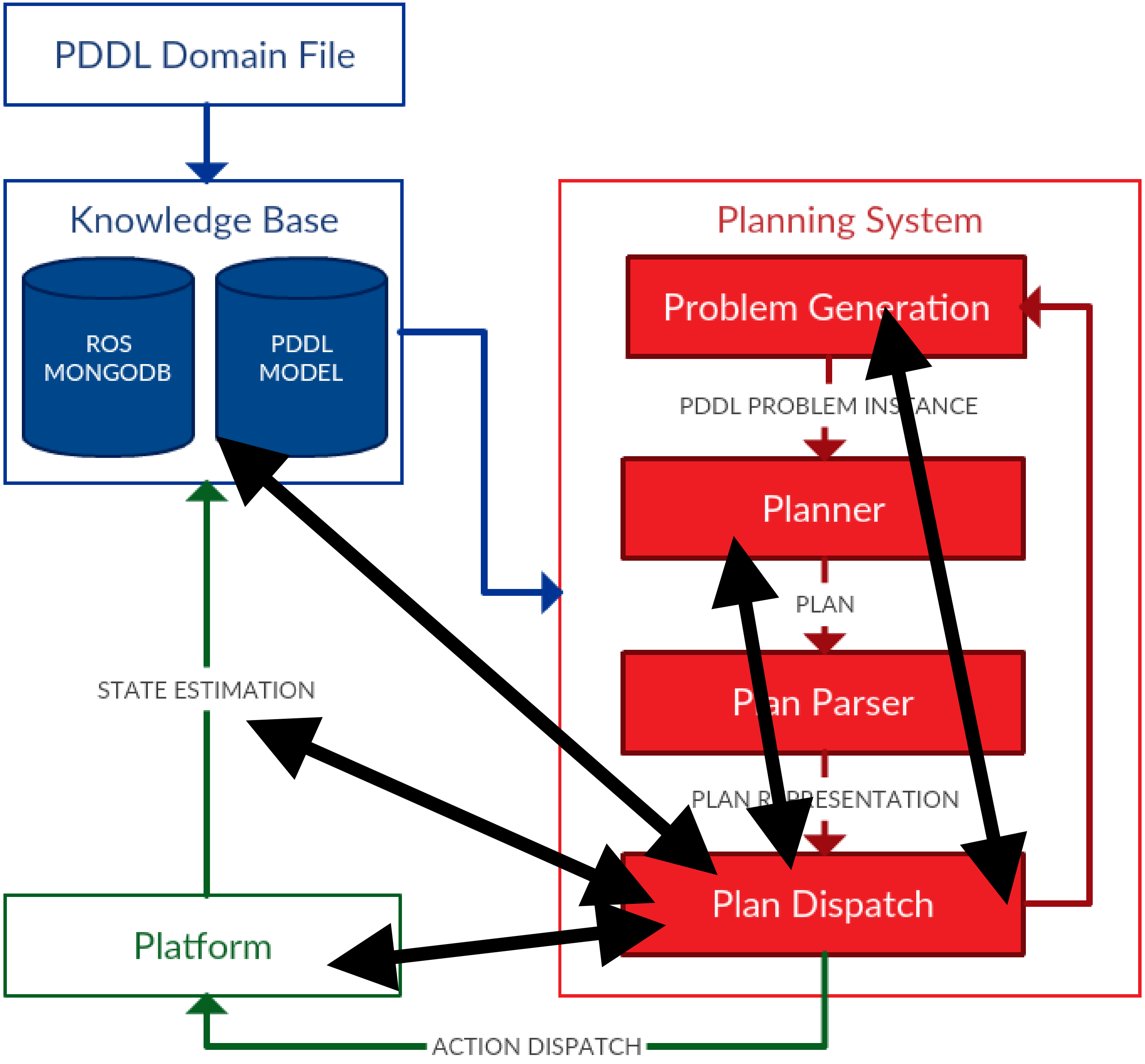
Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.



Summary of Very Simple Plan Execution

Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.

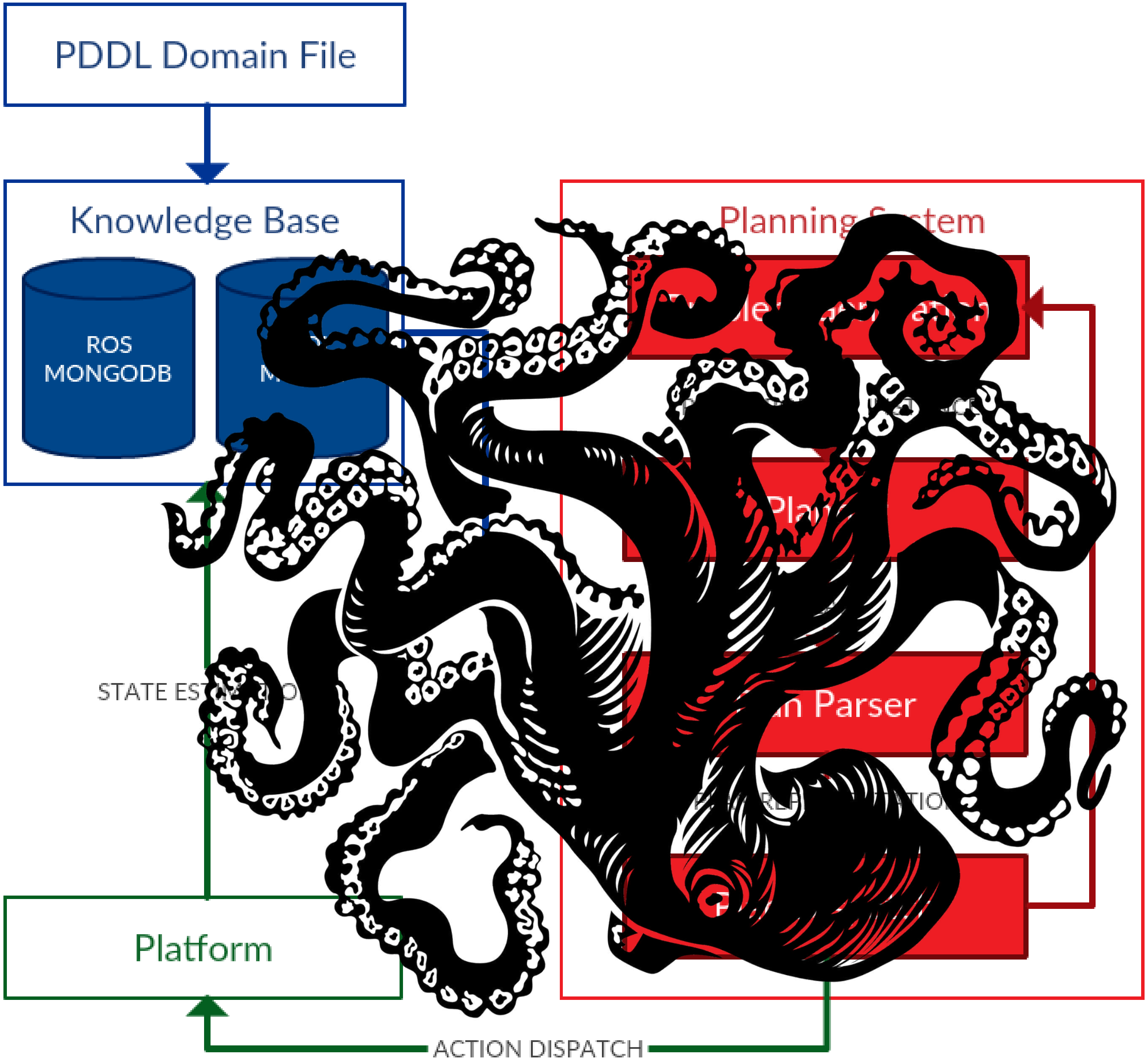
The execution of a plan is an emergent behaviour of the whole system.



Summary of Very Simple Plan Execution

Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.

The execution of a plan is an emergent behaviour of the whole system.



Dispatching more than a Single Plan

The robot can have many different and interfering goals. A robot's behaviour might move toward achievement of multiple goals together.

Dispatching more than a Single Plan

The robot can have many different and interfering goals. A robot's behaviour might move toward achievement of multiple goals together.

The robot can also have:
- long-term goals (plans are abstract, with horizons of weeks)
- but also short-term goals (plans are detailed, with horizons of minutes)

Dispatching more than a Single Plan

The robot can have many different and interfering goals. A robot's behaviour might move toward achievement of multiple goals together.

The robot can also have:
- long-term goals (plans are abstract, with horizons of weeks)
- but also short-term goals (plans are detailed, with horizons of minutes)

The behaviour of a robot should not be restricted to only one plan.

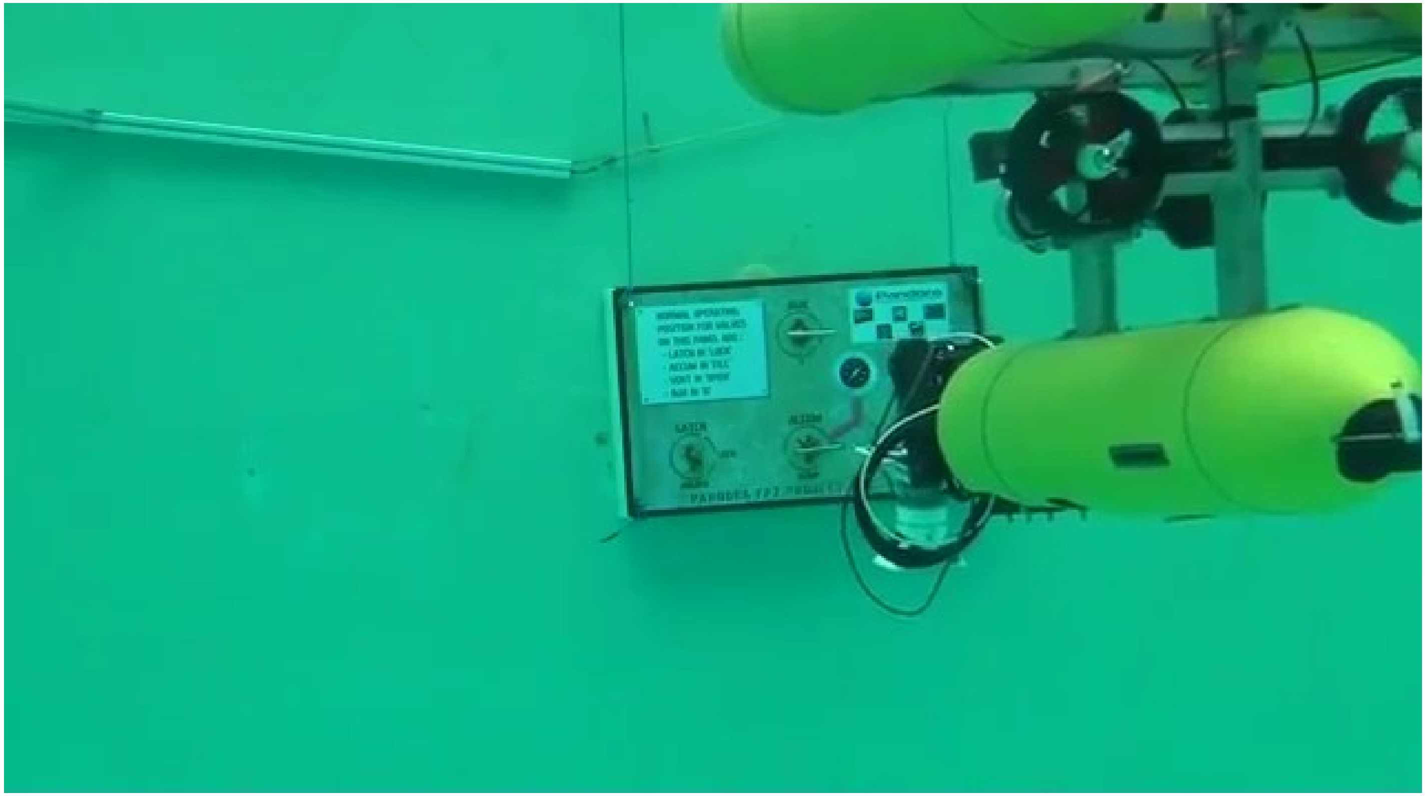
In a persistently autonomous system, the domain model, the planning process, and the plan are frequently revisited.

There is no “waterfall” sequence of boxes.

Dispatching more than a Single Plan

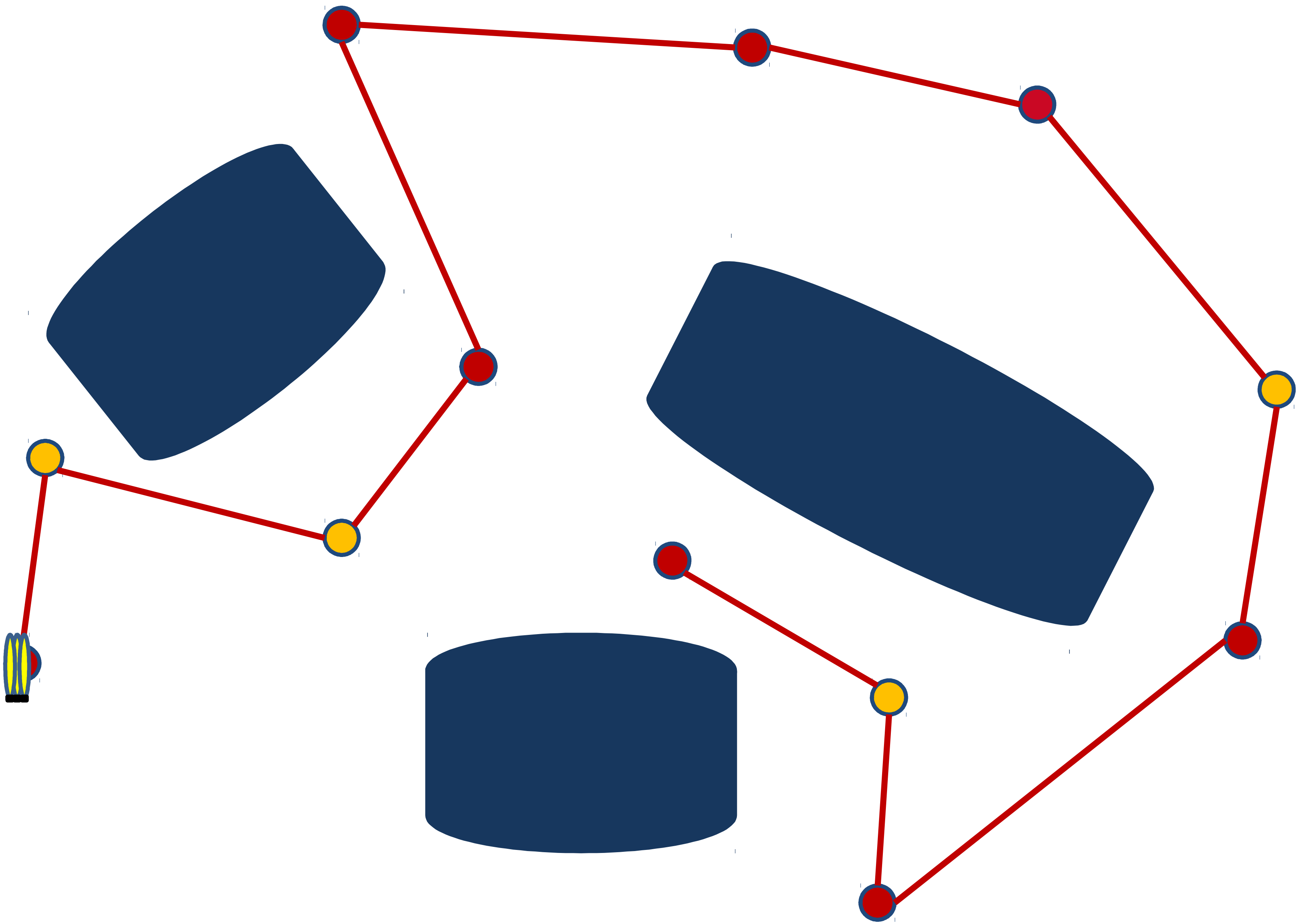
Example of multiple plans: What about unknowns in the environment?

One very common and simple scenario with robots is planning a search scenario. For tracking targets, tidying household objects, or interacting with people.

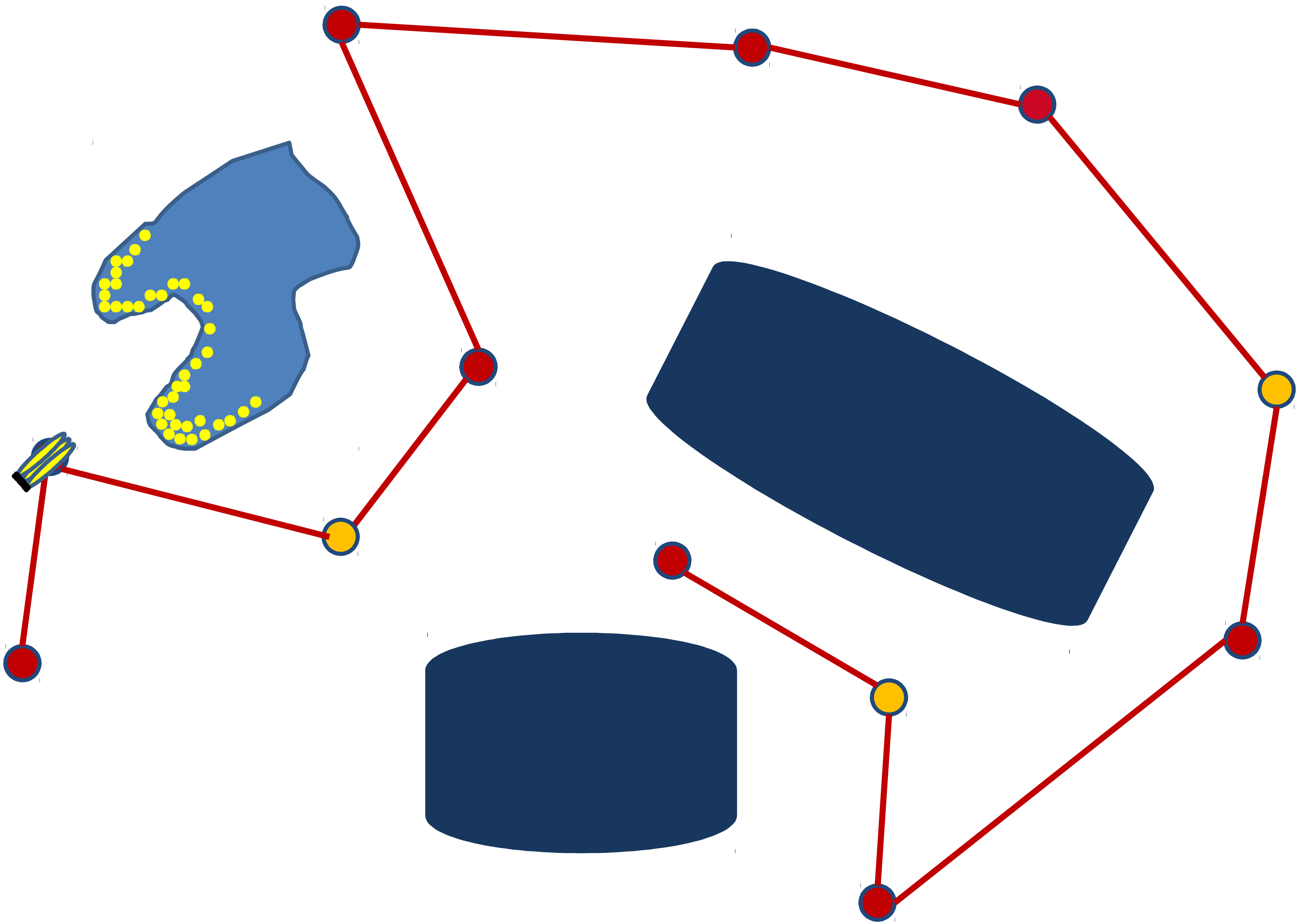


How do you plan from future situations that you can't predict?

Dispatching more than a Single Plan

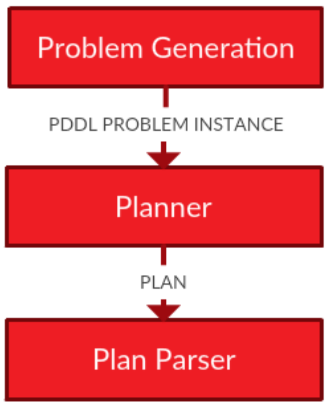


Dispatching more than a Single Plan



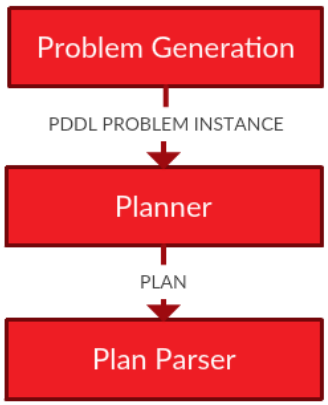
Hierarchical and Recursive Planning

For each task we generate a *tactical plan*.



Hierarchical and Recursive Planning

For each task we generate a *tactical plan*. The time and resource constraints are used in the generation of the strategic problem.



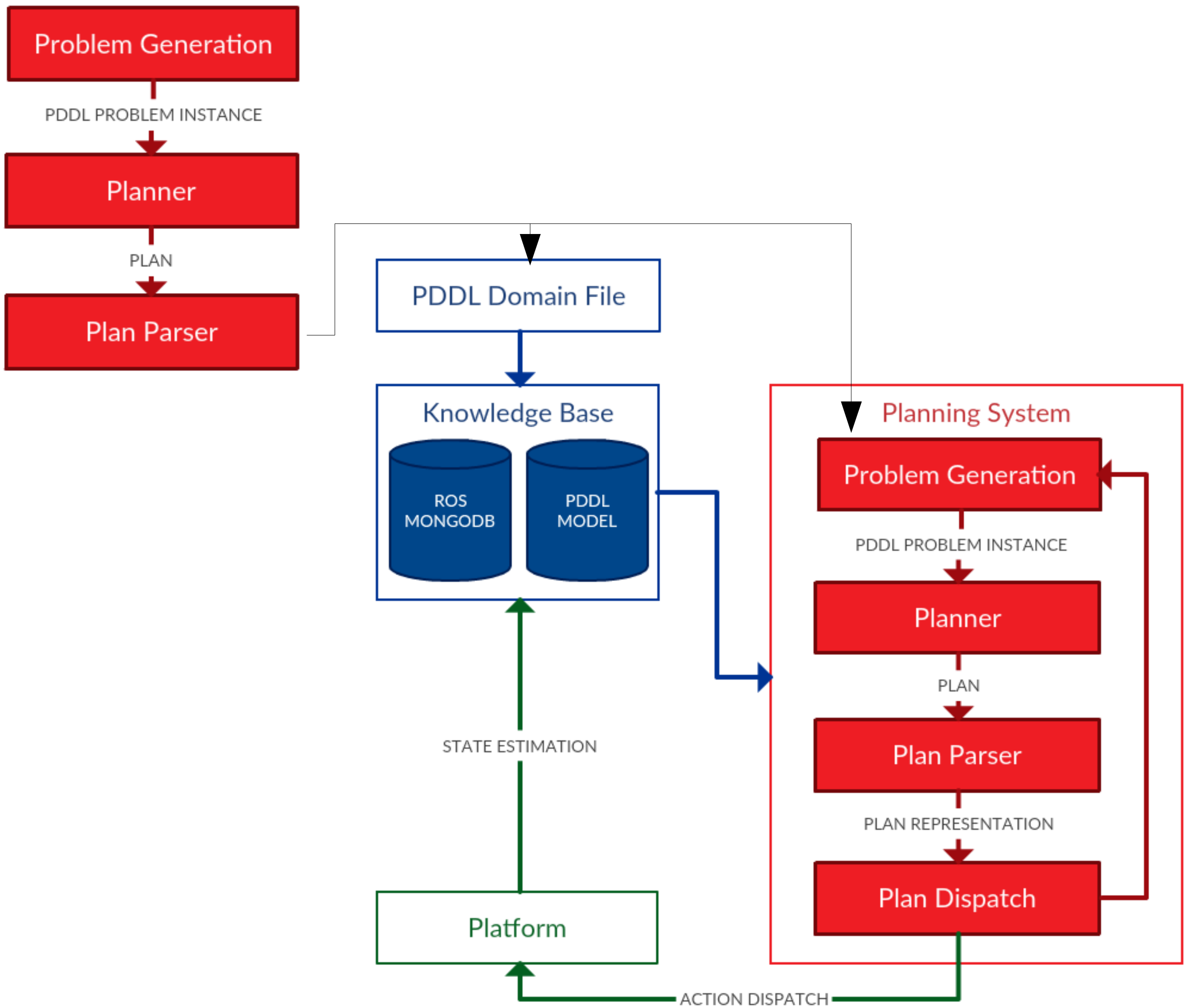

```
0.000: (correct_position auv0 wp_auv0) [3.000]
3.001: (do_hover_fast auv0 wp_auv0 strategic_location_7)
[11.403]
14.405: (correct_position auv0_strategic_location_78)
[3.000]
17.406: (observe_inspection_point auv0 strategic_location_7
inspection_point_2) [10.000]
27.407: (correct_position auv0 strategic_location_7)
[3.000]
45.083: (do_hover_controlled auv0 strategic_location_5
strategic_location_5) [4.000]
49.084: (observe_inspecetion_point auv0
strategic_location_5 inspection_point_4) [10.000]
...
```

complete_mission

Energy consumption = 10W
Duration = 86.43s

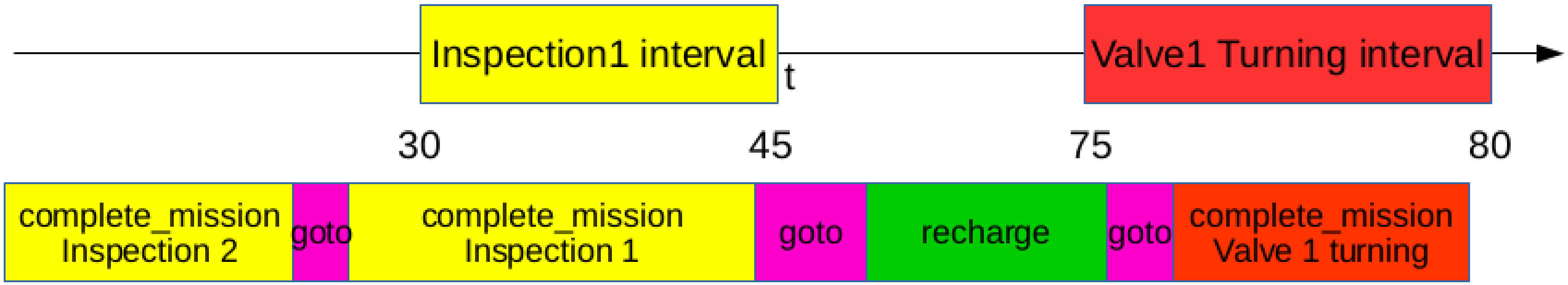
Hierarchical and Recursive Planning

For each task we generate a *tactical plan*. The time and resource constraints are used in the generation of the strategic problem.



Hierarchical and Recursive Planning

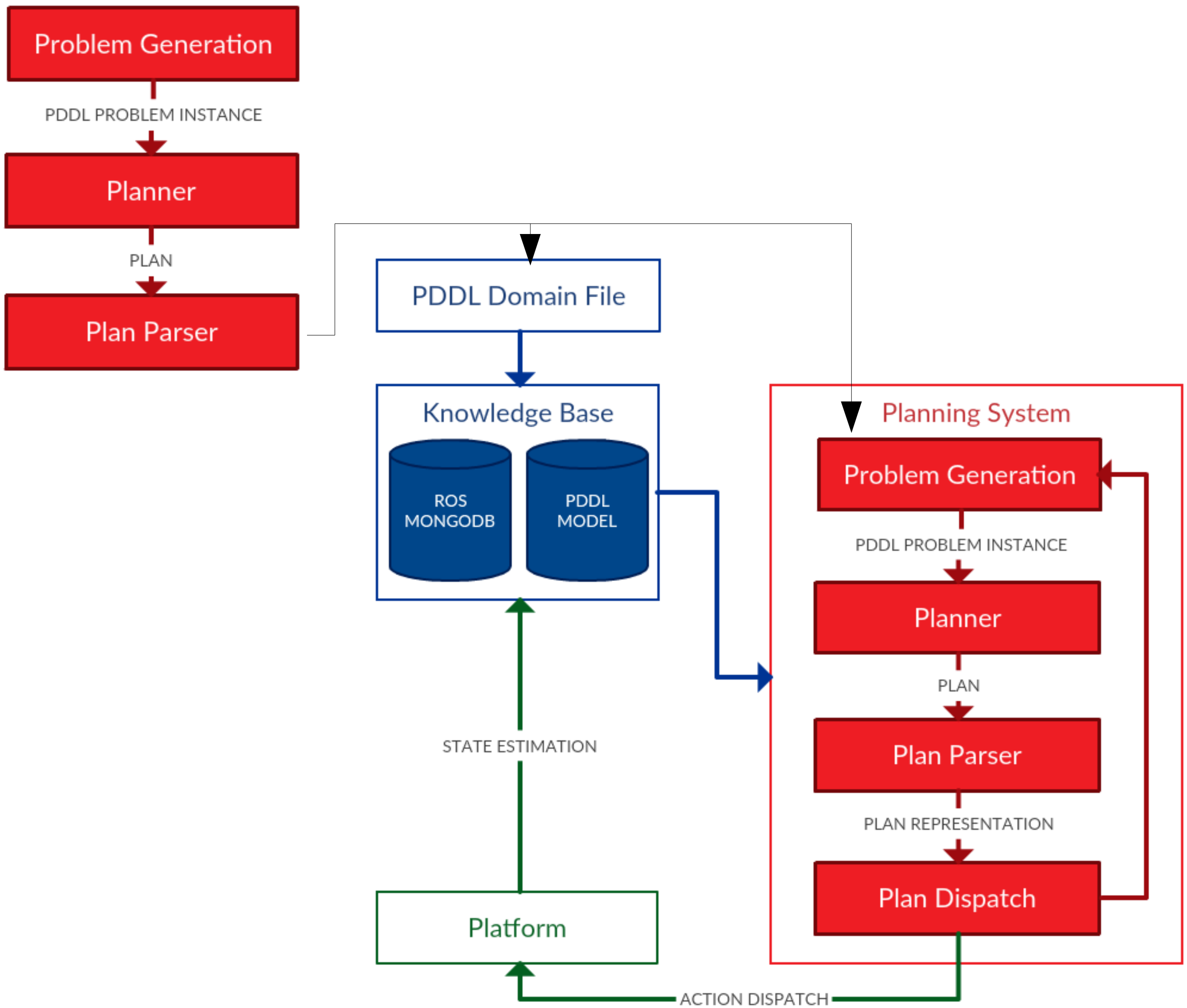
For each task we generate a *tactical plan*. The time and resource constraints are used in the generation of the strategic problem.



A strategic plan is generated that does not violate the time and resource constraints of the whole mission.

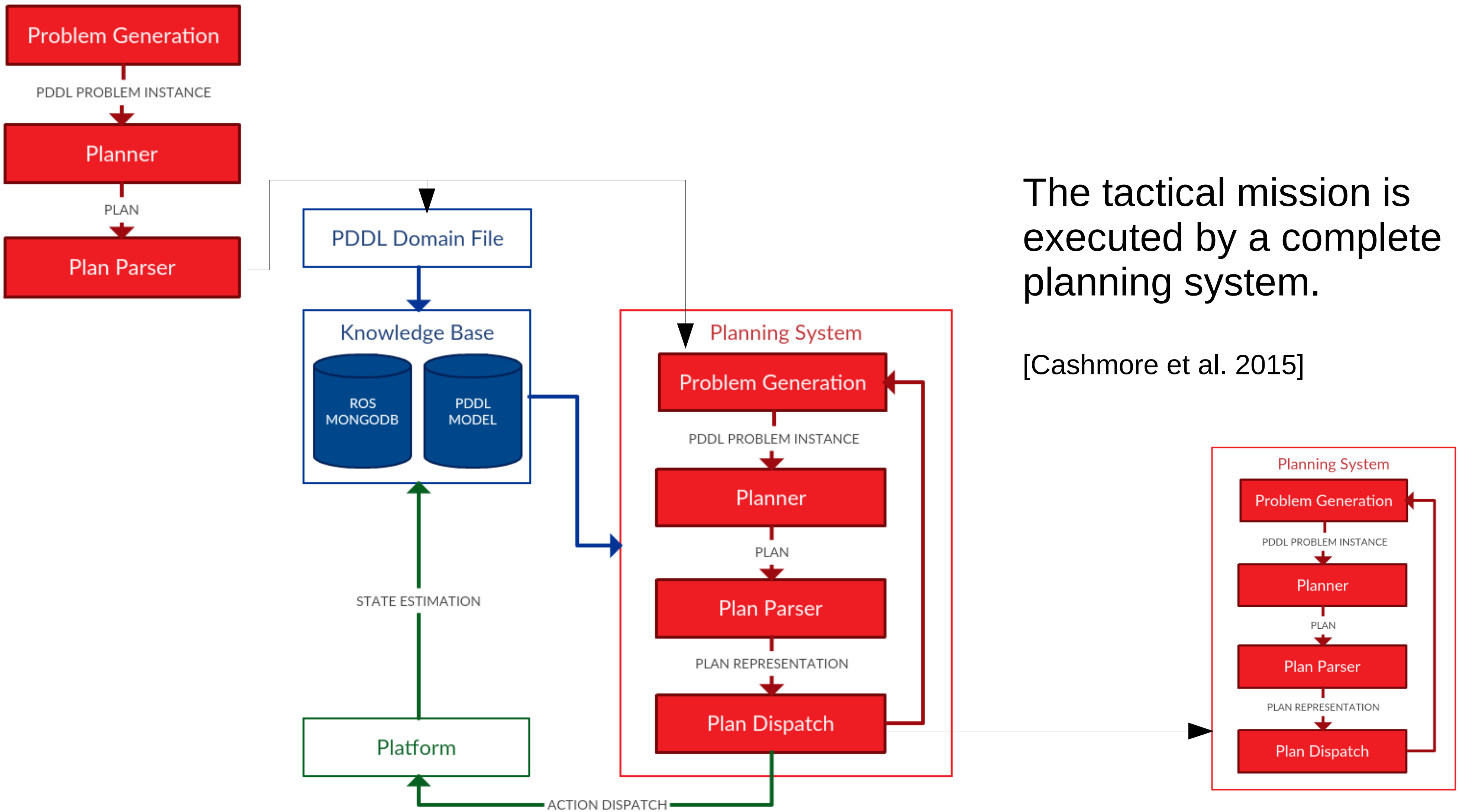
Hierarchical and Recursive Planning

When an abstract “complete_mission” action is dispatched, the tactical problem is regenerated, replanned, and executed.



Hierarchical and Recursive Planning

When an abstract “complete_mission” action is dispatched, the tactical problem is regenerated, replanned, and executed.



The tactical mission is executed by a complete planning system.

[Cashmore et al. 2015]

Dispatching more Plans: Opportunistic Planning

There might also be unknowns that we don't expect to discover.

For example, new opportunities are found during execution, and the robot should exploit them.

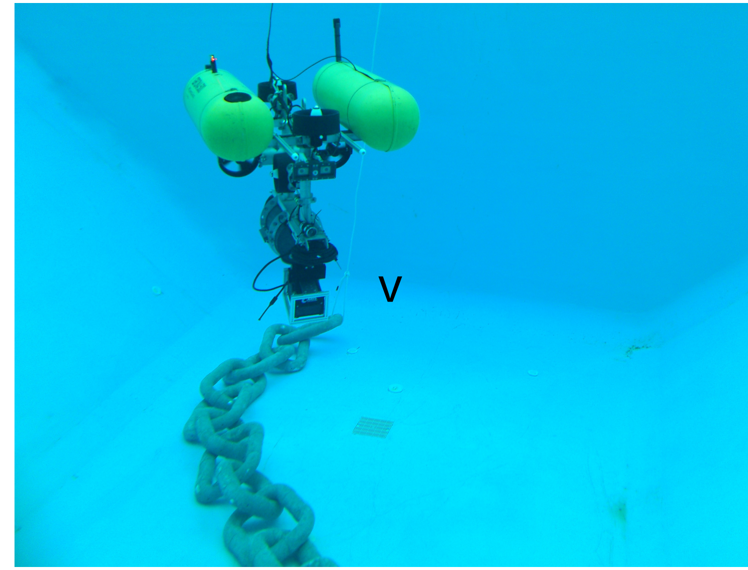
Dispatching more Plans: Opportunistic Planning

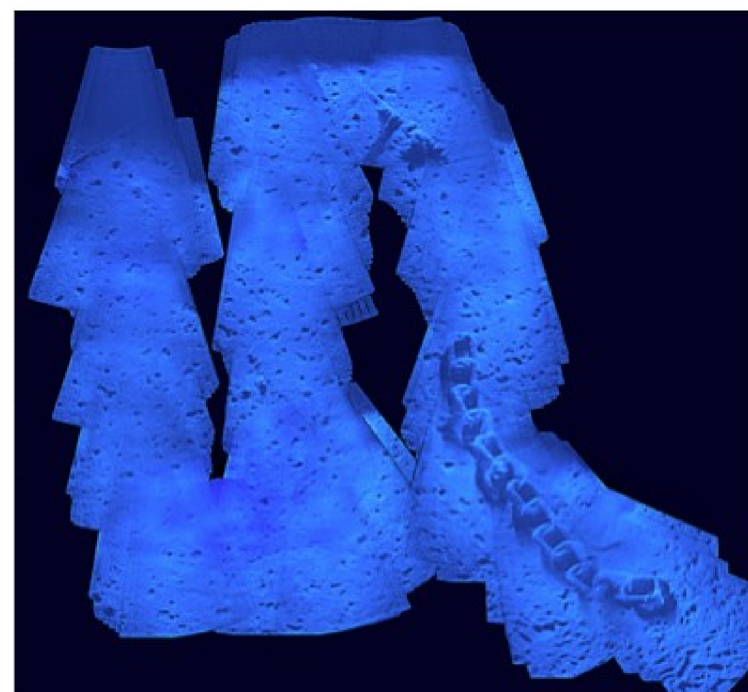
High Impact Low-Probability Events (HILPs)

- the probability distribution is unknown
- cannot be anticipated
- **our example is chain following**

If you see an unexpected chain, it's a good idea to investigate...

2011 Banff	5 of 10 lines parted.
2011 Volve	2 of 9 lines parted
2011 Gryphon Alpha	4 of 10 lines parted, vessel drifted a distance, riser broken
2010 Jubarte	3 lines parted between 2008 and 2010.
2009 Nan Hai Fa Xian	4 of 8 lines parted; vessel drifted a distance, riser broken
2009 Hai Yang Shi You	Entire yoke mooring column collapsed; vessel adrift, riser broken.
2006 Liuhua (N.H.S.L.)	7 of 10 lines parted; vessel drifted a distance, riser broken.
2002 Girassol buoy	3 (+2) of 9 lines parted, no damage to offloading lines (2 later)



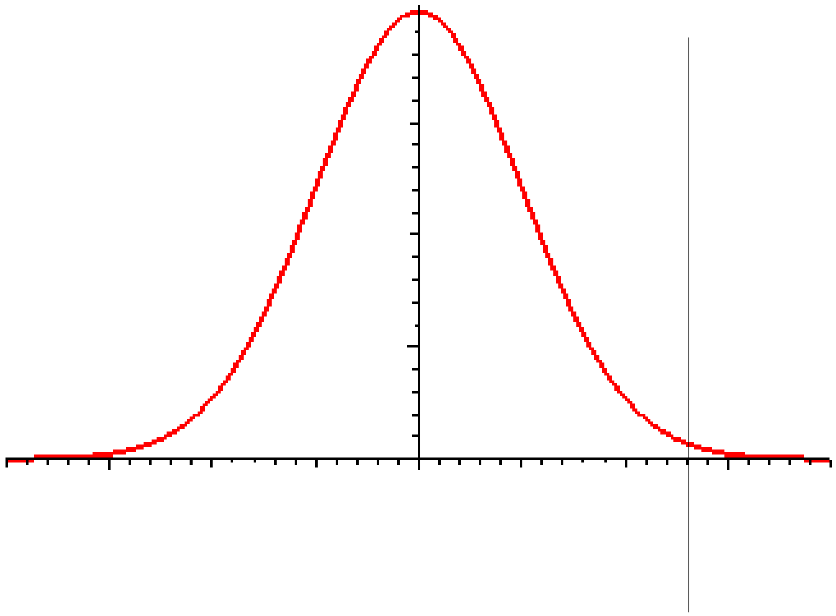


Dispatching more Plans: Opportunistic Planning

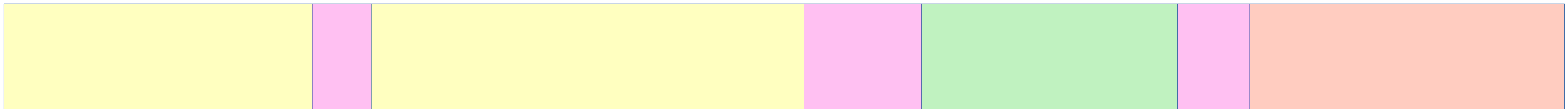
In PANDORA we planned and executed missions over long-term horizons (days or weeks)

Our planning strategy was based on the assumption that actions have durations normally distributed around the mean.

To build a robust plan we therefore used estimated durations for the actions that were 95th percentile of the normal distribution.



The resulting overestimation of actions builds a **free time window**

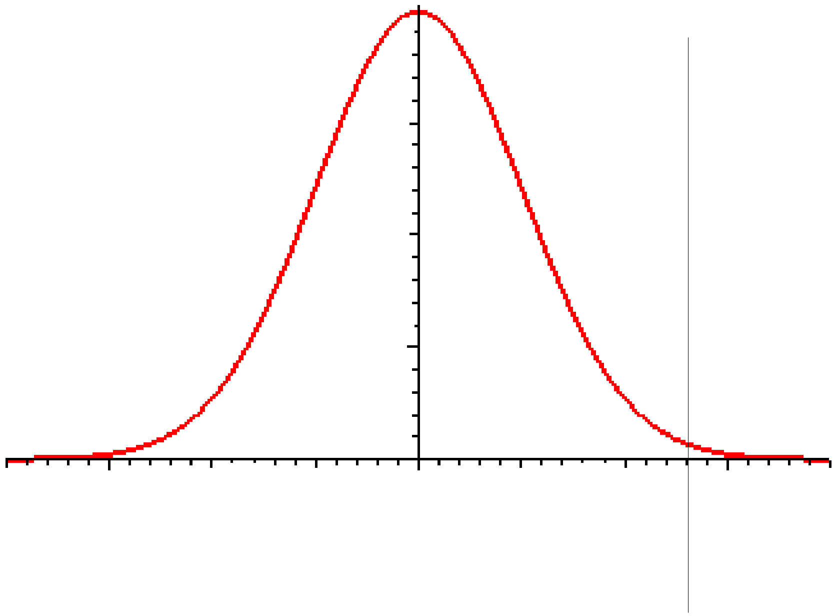


Dispatching more Plans: Opportunistic Planning

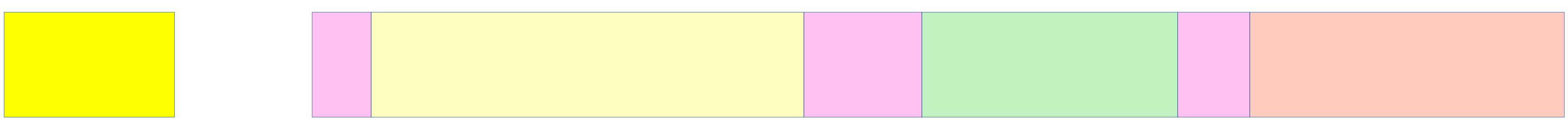
In PANDORA we planned and executed missions over long-term horizons (days or weeks)

Our planning strategy was based on the assumption that actions have durations normally distributed around the mean.

To build a robust plan we therefore used estimated durations for the actions that were 95th percentile of the normal distribution.



The resulting overestimation of actions builds a **free time window**

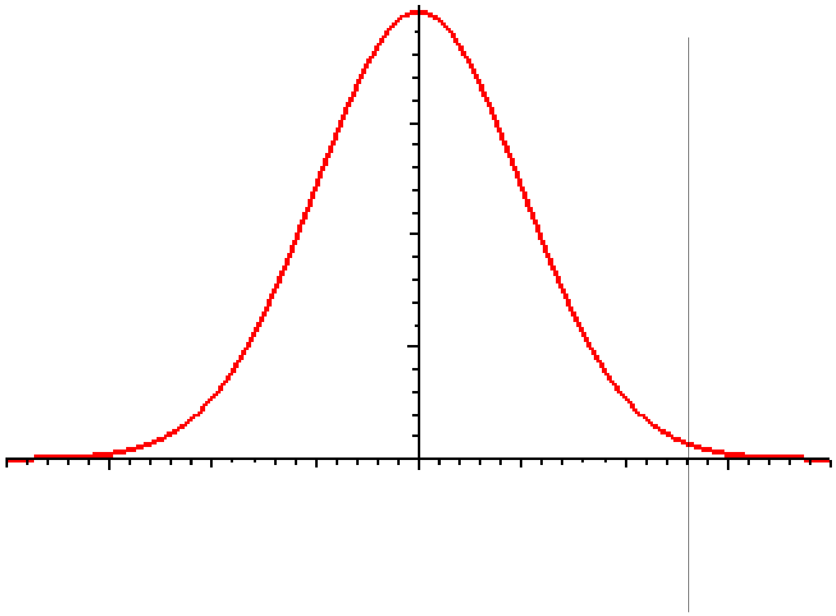


Dispatching more Plans: Opportunistic Planning

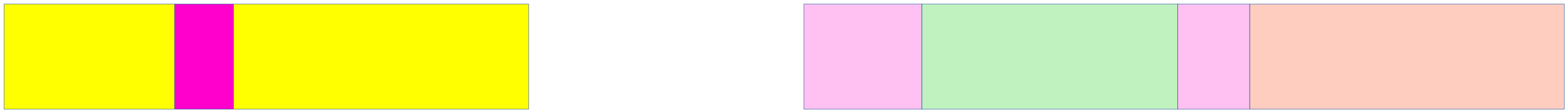
In PANDORA we planned and executed missions over long-term horizons (days or weeks)

Our planning strategy was based on the assumption that actions have durations normally distributed around the mean.

To build a robust plan we therefore used estimated durations for the actions that were 95th percentile of the normal distribution.



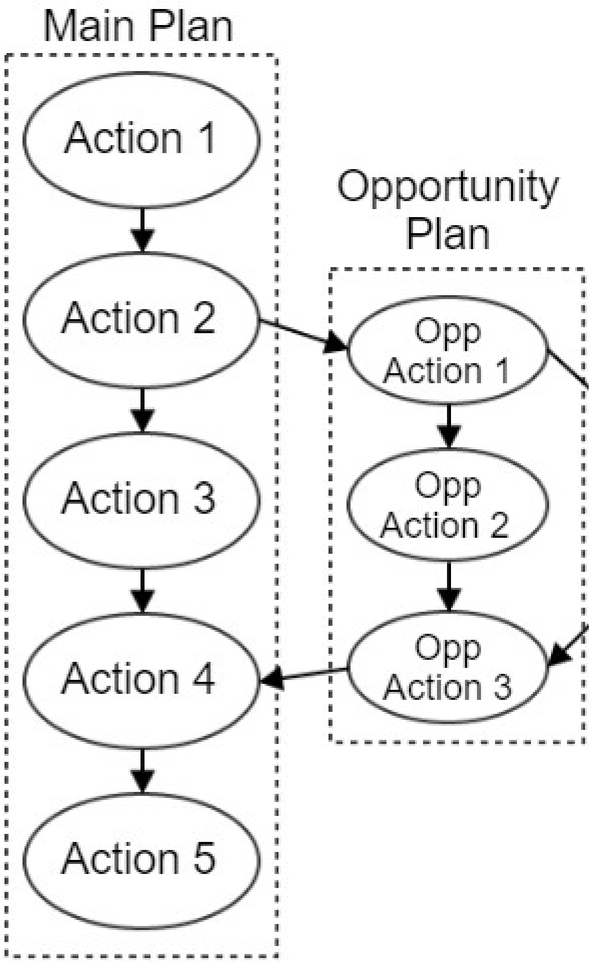
The resulting overestimation of actions builds a **free time window**



Dispatching more Plans: Opportunistic Planning

New plans are generated for the opportunistic goals and the goal of returning to the tail of the current plan.

If the new plan fits inside the free time window, then it is immediately executed.



Dispatching more Plans: Opportunistic Planning

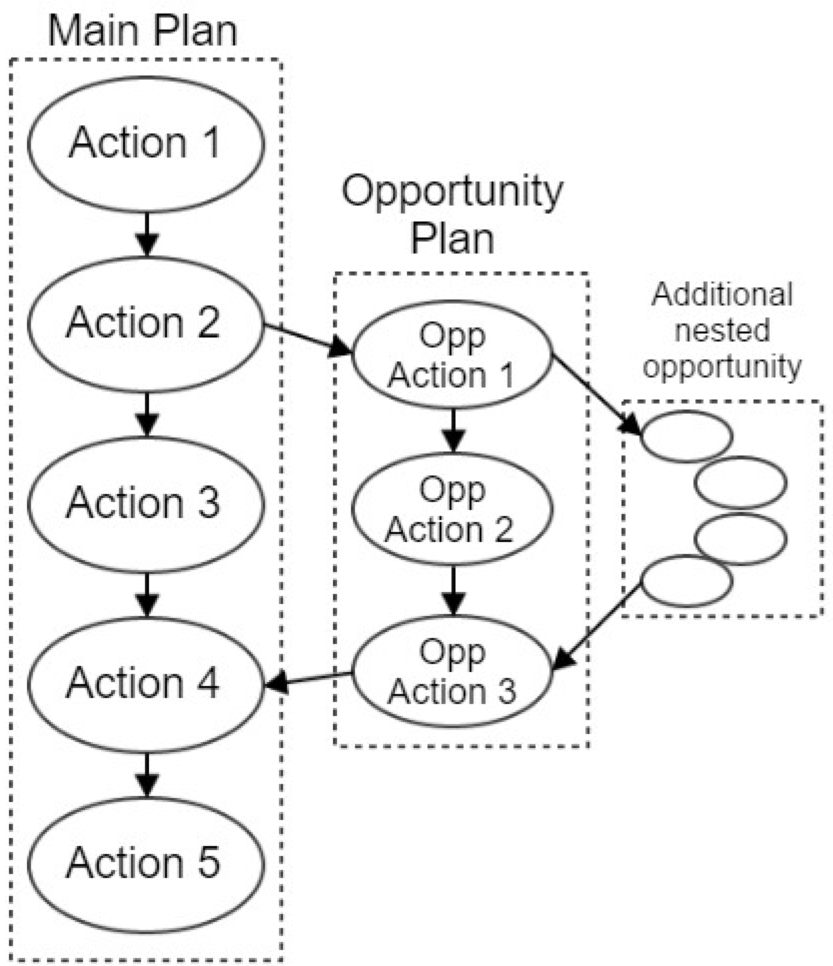
New plans are generated for the opportunistic goals and the goal of returning to the tail of the current plan.

If the new plan fits inside the free time window, then it is immediately executed.

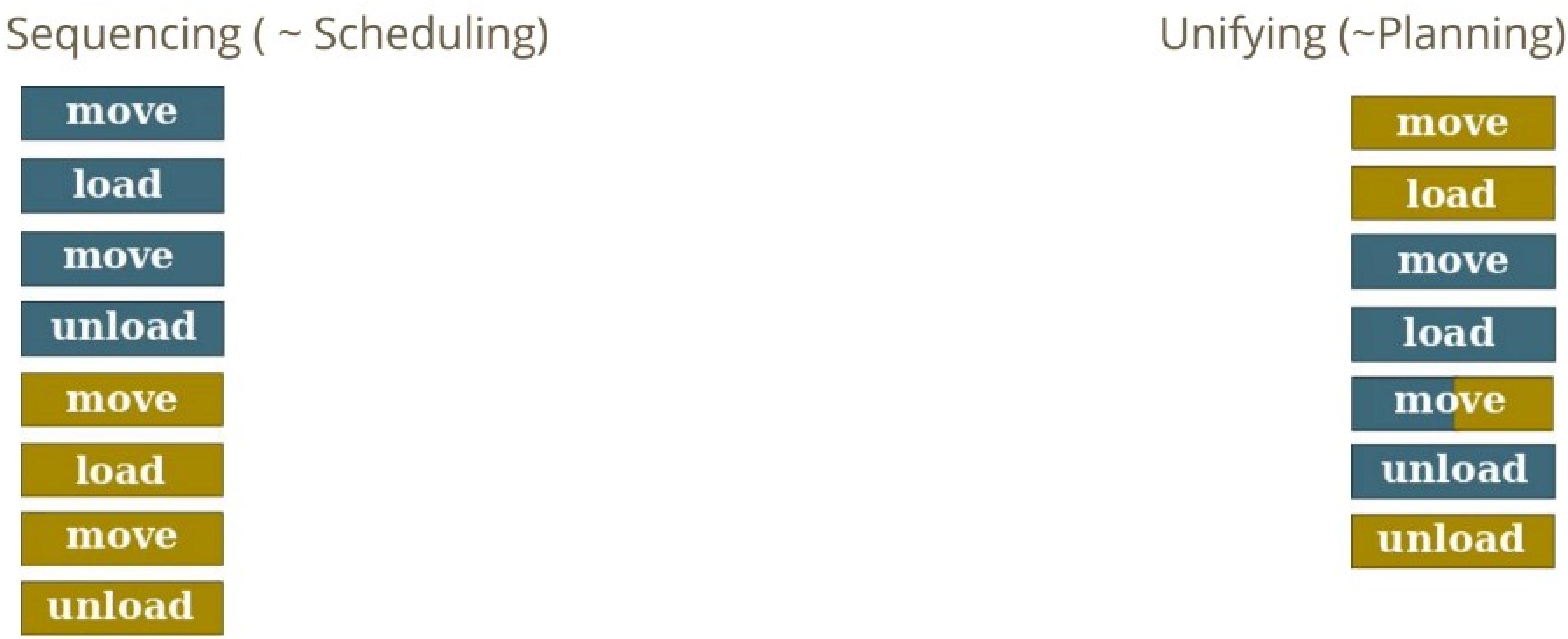
The approach is recursive:

If an opportunity is spotted during the execution of a plan fragment, then the currently executing plan can be pushed onto the stack and a new plan can be executed.

[Cashmore et al. 2015]

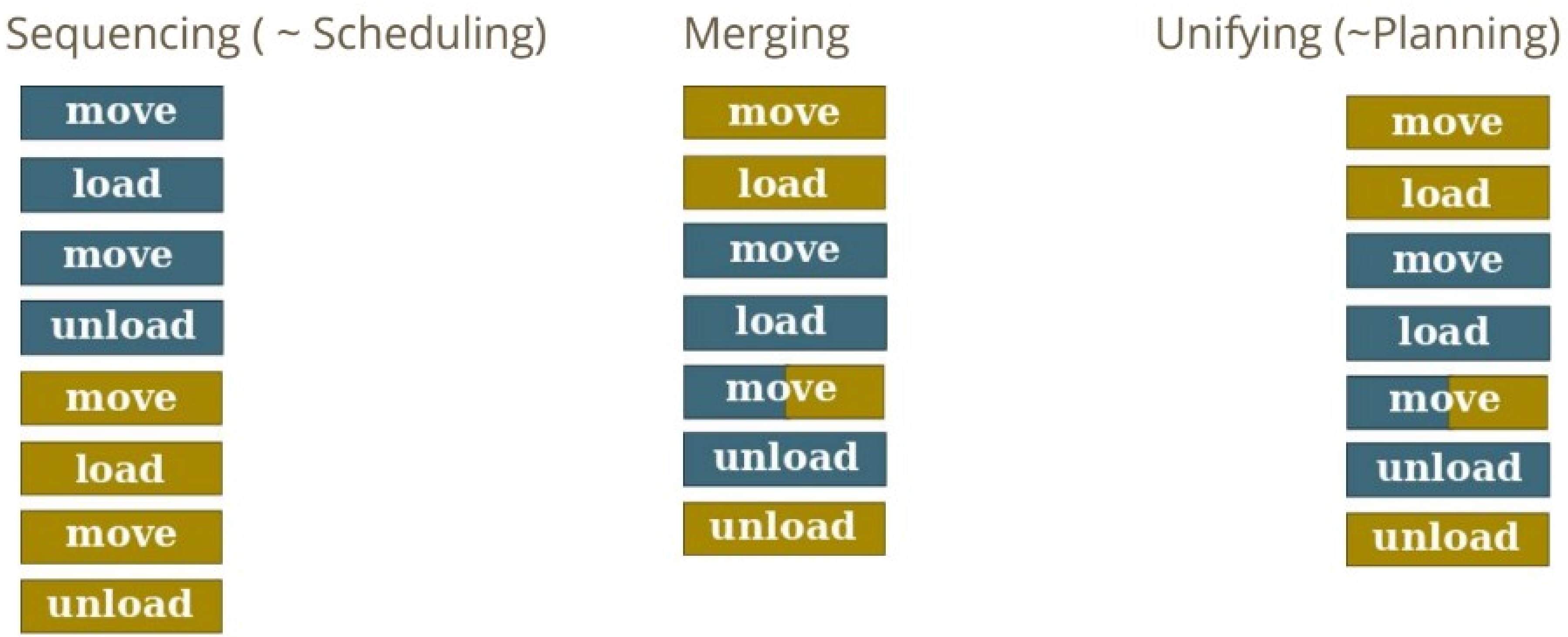


Dispatching Plans at the same time



Separating tasks and scheduling is not as efficient.
Planning for everything together is not always practical.

Dispatching Plans at the same time



Separating tasks and scheduling is not as efficient.
Planning for everything together is not always practical.

Plans can be merged in a more intelligent way. A single action can support the advancement towards multiple goals. [Lenka et al. 2016]

Questions?

What is the glue in a Plan Execution framework that is *always* required?

How do we modify a domain model during execution?

Which parts of a domain model are transferable to other tasks?

Which parts of a domain model can be generated automatically

- From a description of the robot?
- From a source ontology?

How can we get rid of the planning expert?

- Can a description of a task be written by a non-expert, and a generic domain extended?