Outline

- ROS Basics
- Plan Execution
  - Very Simple Dispatch
  - Very Simple Temporal Dispatch
  - Conditional Dispatch
  - Temporal and Conditional Dispatch together
- Dispatching More than a Single Plan
  - Hierarchical and Recursive Planning
  - Opportunistic Planning

ROS Basics

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

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

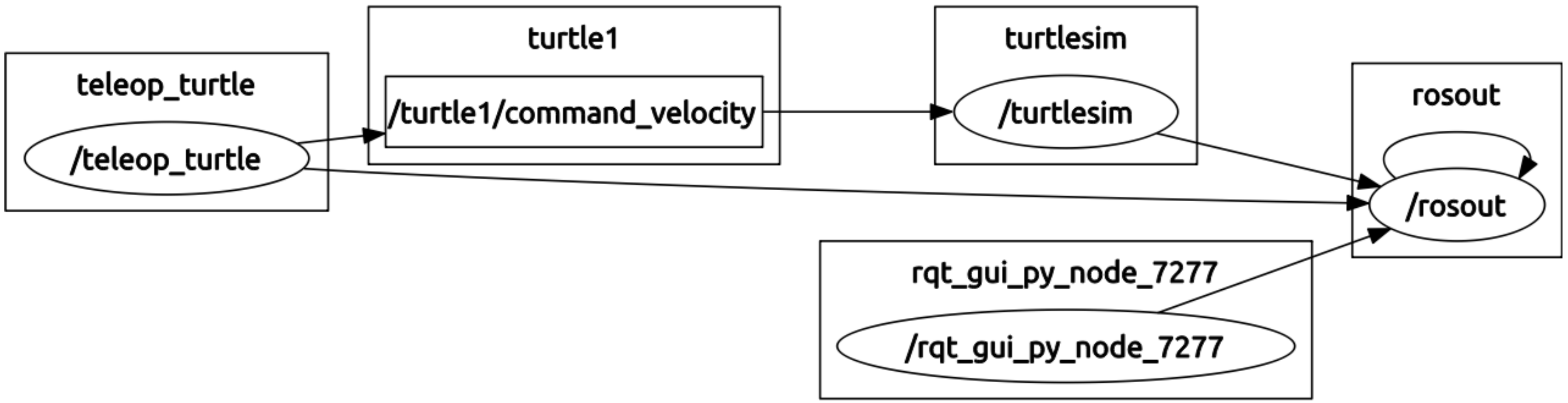
1. ROS messages are published on topics and are many-to-many.
2. ROS services are used for synchronous request/response.

ROS Basics

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

A ROS system is composed of nodes, which pass messages, 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, 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="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" />
    <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.

The actionlib package standardizes the interface for preemptable 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:

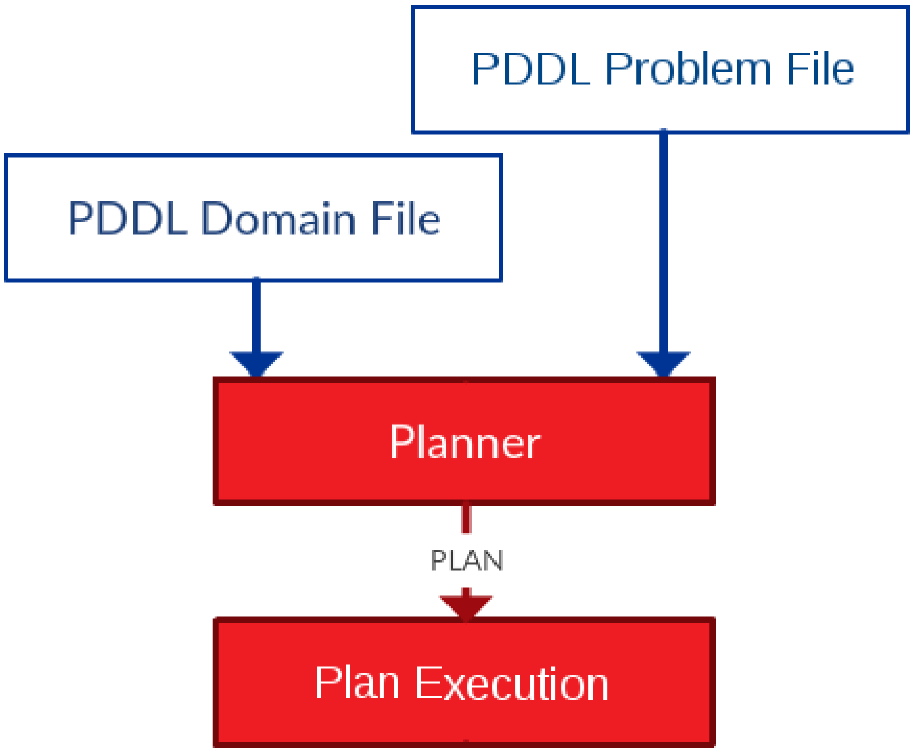
- goals
- feedback
- result

```
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
```

Plan Execution 1: Very simple Dispatch

The most basic structure.

- The plan is generated.
- The plan is executed.



Plan Execution 1: Very simple Dispatch

(Some) Related Work

McGann et el.C., Py, F., A deliberative architecture for AUV control. *In Proc. Int. Conf. on Robotics and Automation (ICRA),* 2008

Beetz & McDermott Improving Robot Plans During Their Execution. *In Proc. International Conference on AI Planning Systems (AIPS),* 1994

Ingrand et el. PRS: a high level supervision and control language for autonomous mobile robots. *In IEEE Int.l Conf. on Robotics and Automation,* 1996

Kortenkamp & Simmons Robotic Systems Architectures and Programming. *In Springer Handbook of Robotics, pp. 187–206,* 2008

Lemai-Chenevier & Ingrand Interleaving Temporal Planning and Execution in Robotics Domains. *In Proceedings of the National Conference on Artificial Intelligence (AAAI),* 2004

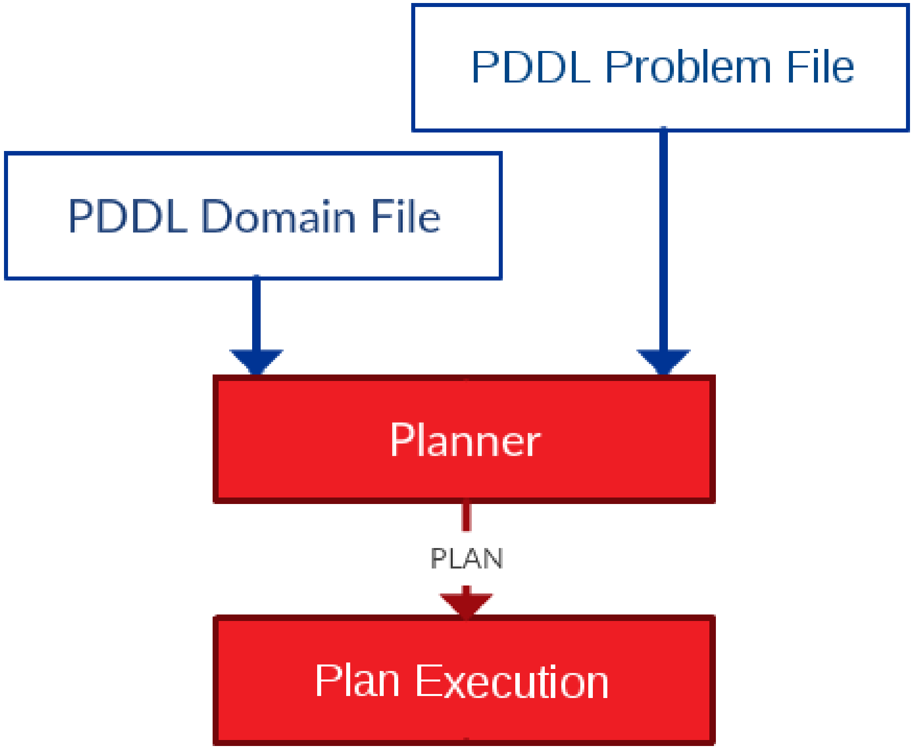
Baskaran, et el. Plan execution interchance language (PLEXIL) Version 1.0. *NASA Technical Memorandum,* 2007

Robertson et al. Autonomous Robust Execution of Complex Robotic Missions. *Proceedings of the 9th International Conference on Intelligent Autonomous Systems (IAS-9)*, 2006

Plan Execution 1: Very simple Dispatch

The most basic structure.

- The plan is generated.
- The plan is executed.



Plan Execution 1: Very simple Dispatch

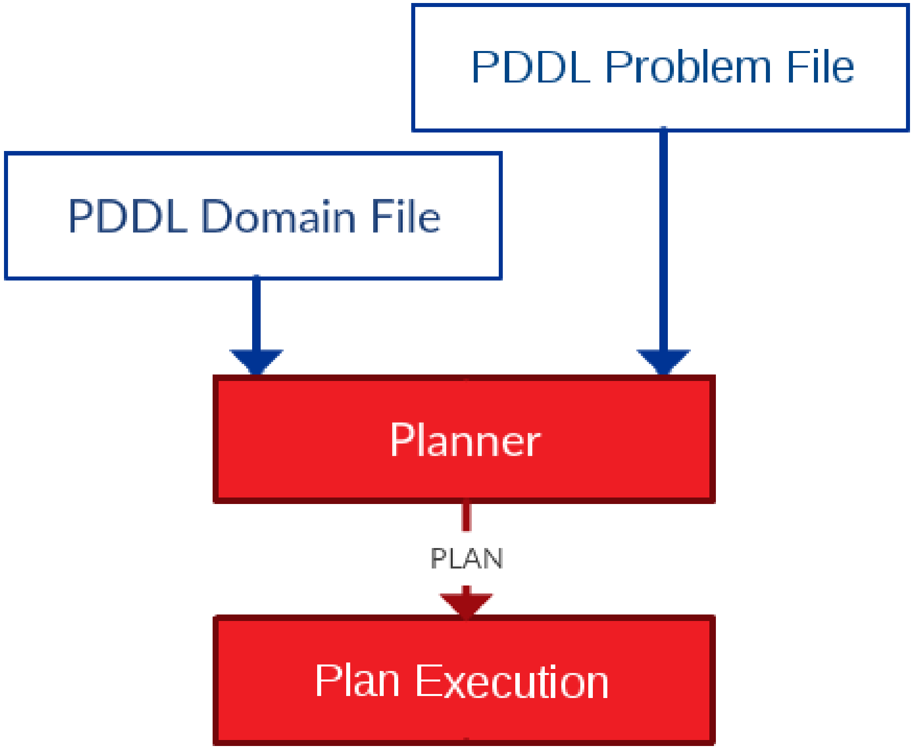
The most basic structure.

- The plan is generated.
- The plan is executed.

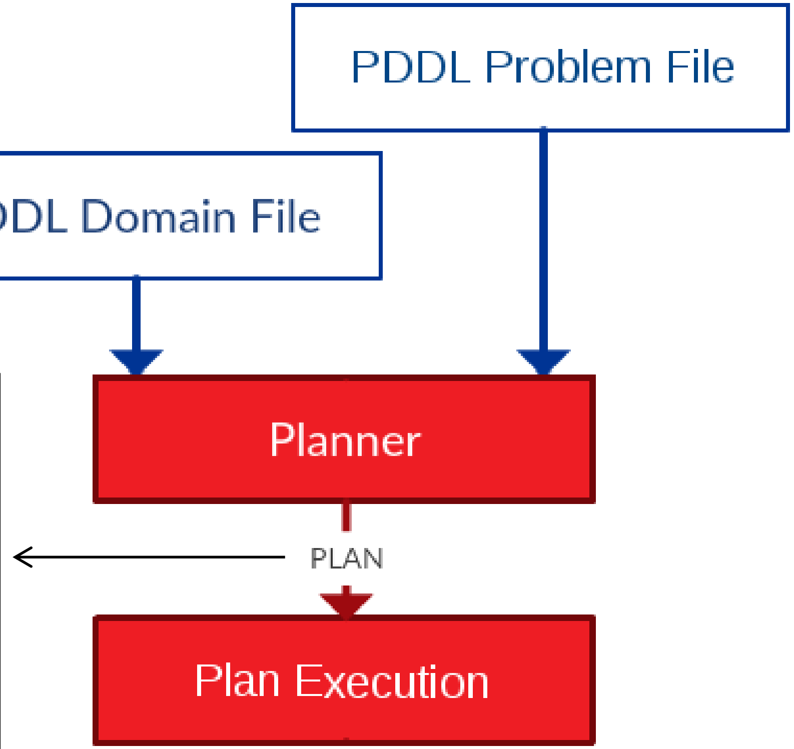
Red boxes are components of ROSPlan. They correspond to ROS nodes.

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

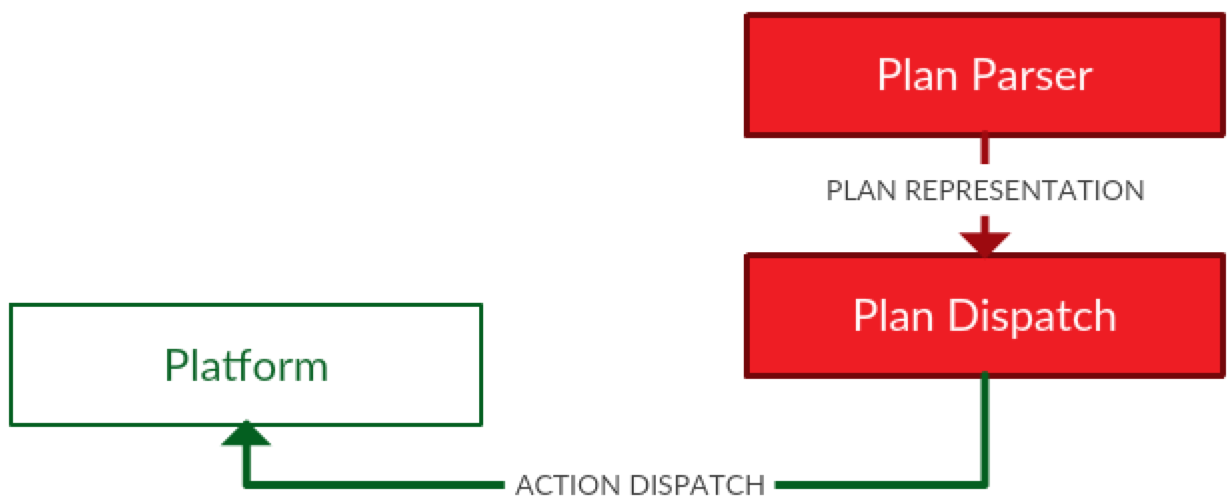
float32 duration

float32 dispatch_time

A dispatch loop without feedback

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

- simple sequential execution
- timed execution
- Petri-Net plans
- Esterel Plans
- etc.

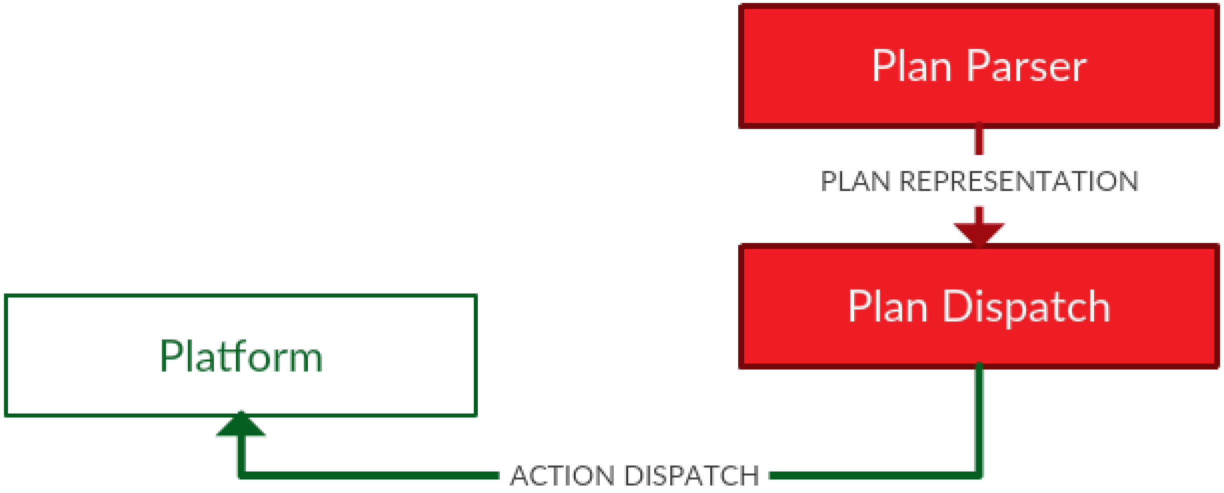


A dispatch loop without feedback

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.



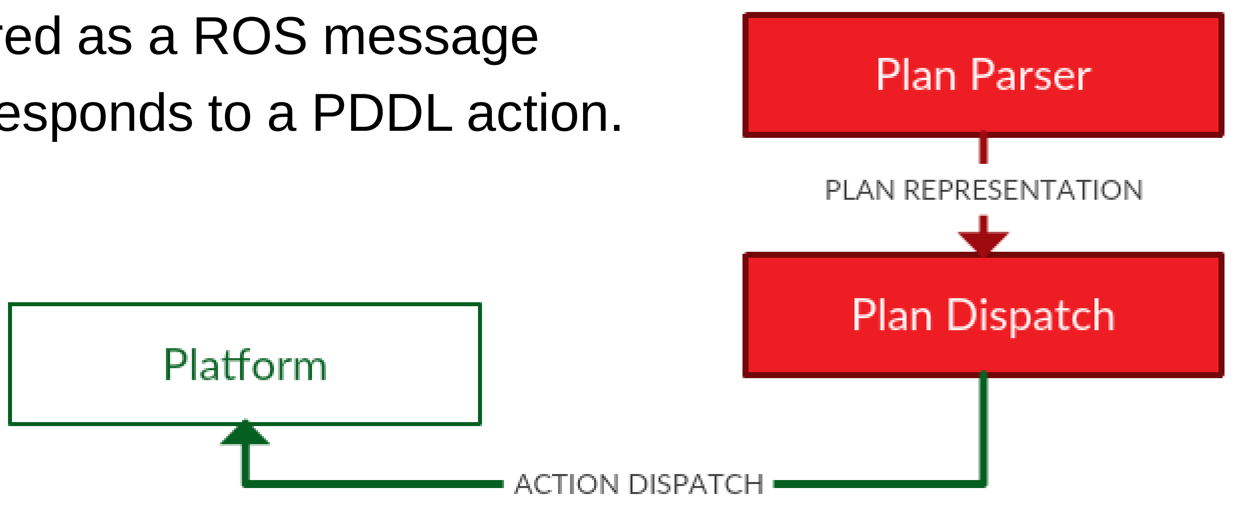
A dispatch loop without feedback

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.



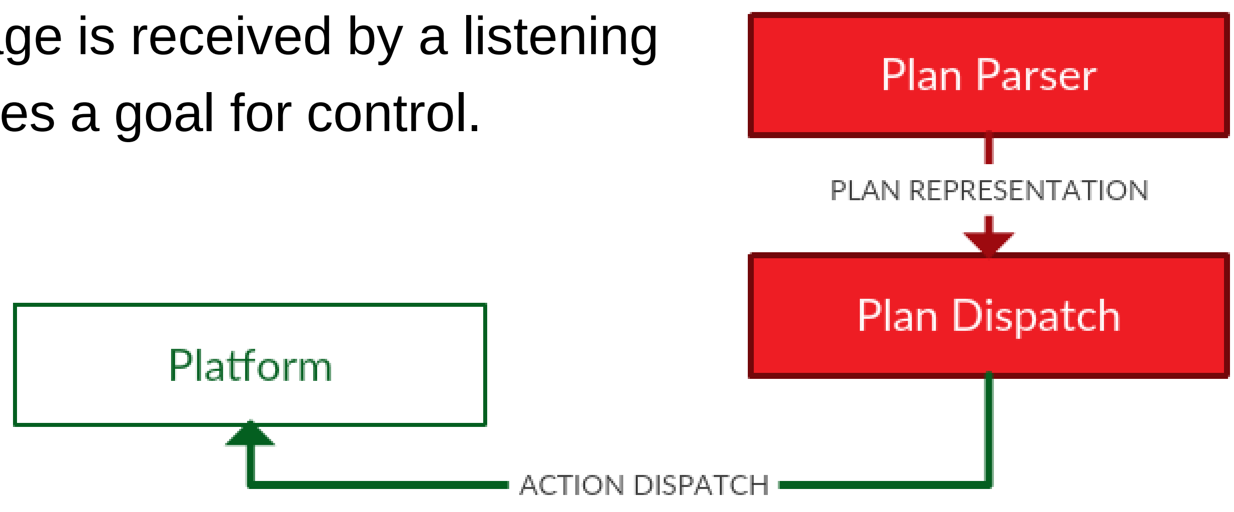
A dispatch loop without feedback

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.



A dispatch loop without feedback

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

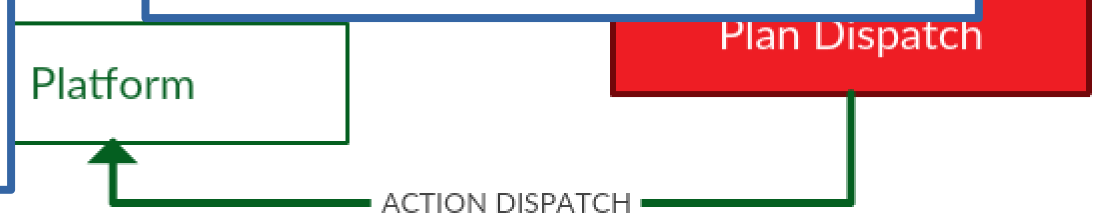
- simple sequential execution

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

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

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
      ...

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



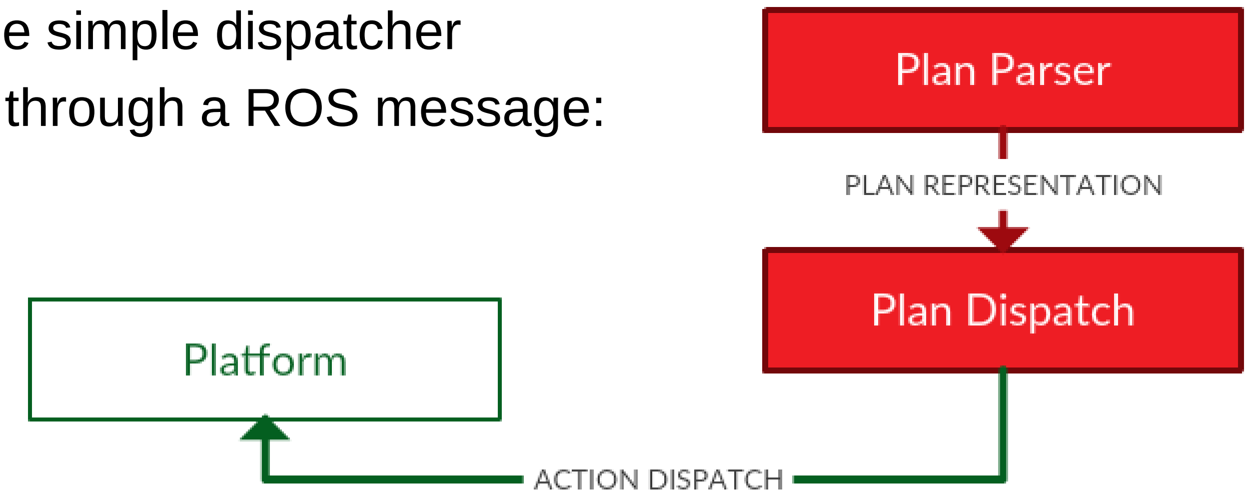
A dispatch loop without feedback

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.*

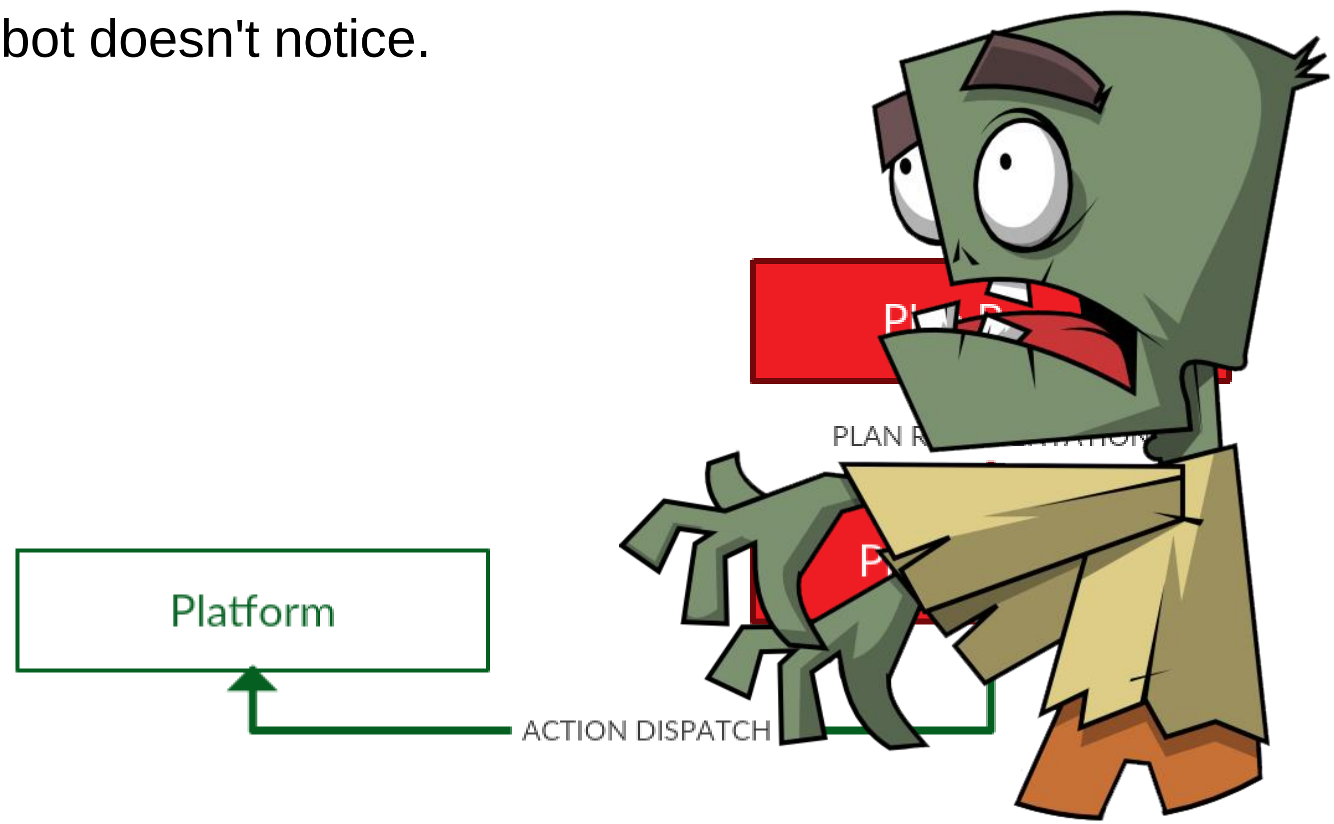


Plan Execution Failure

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.

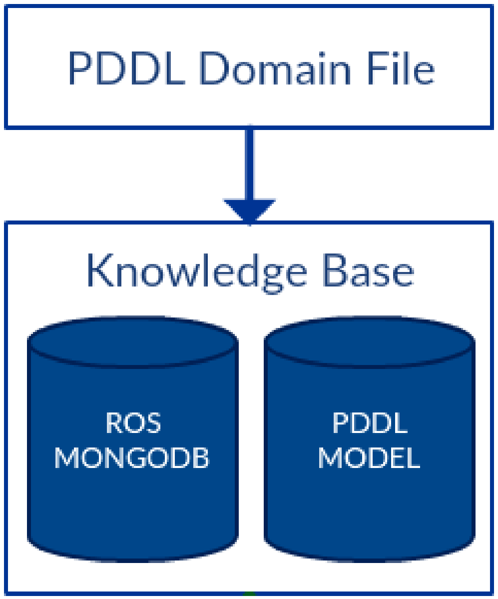
If we desire persistent autonomy, then the robot must be able to plan again, from the new current state, without human intervention.

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.

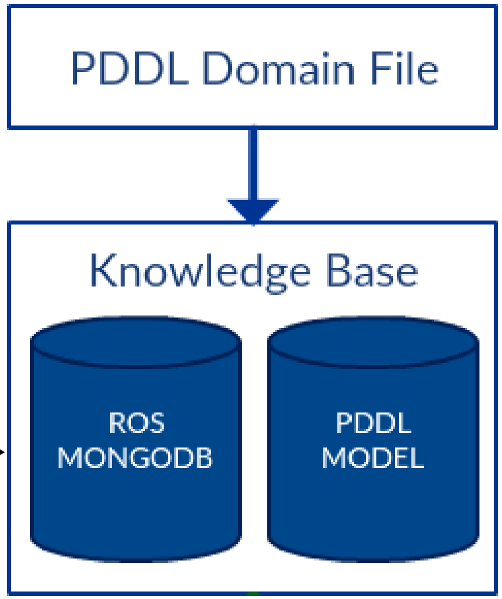


PDDL Model

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



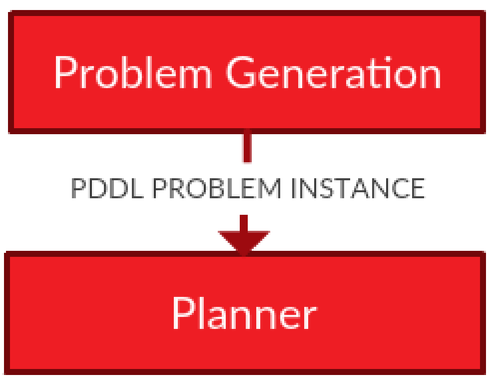
PDDL Model

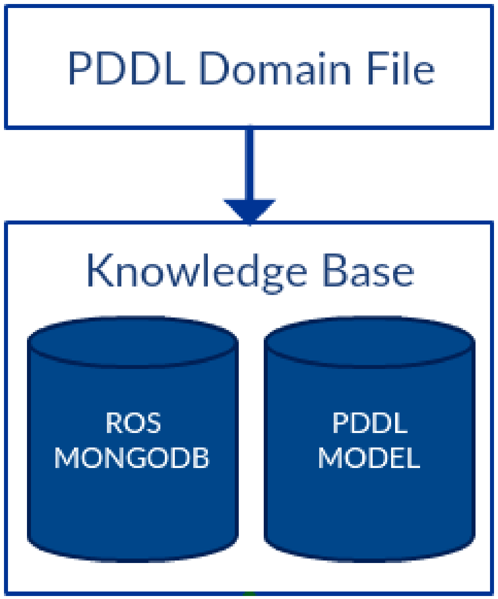
To generate the problem file automatically, the agent must store a model of the world.

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.



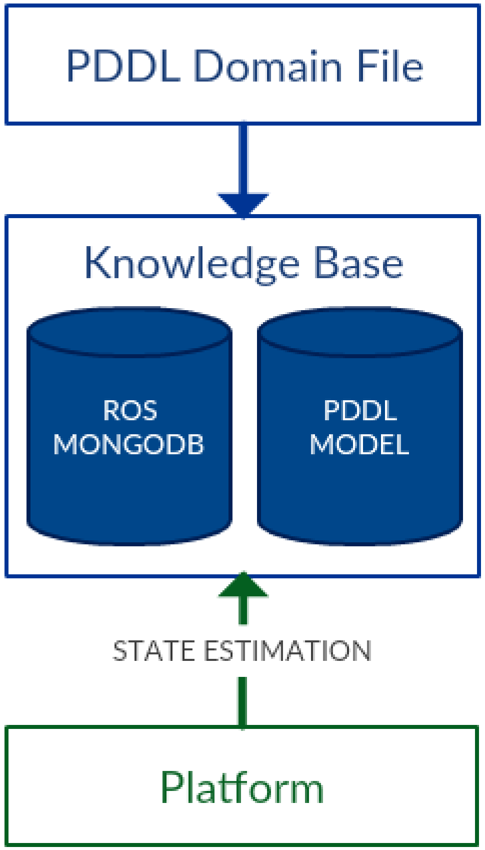


PDDL Model

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.

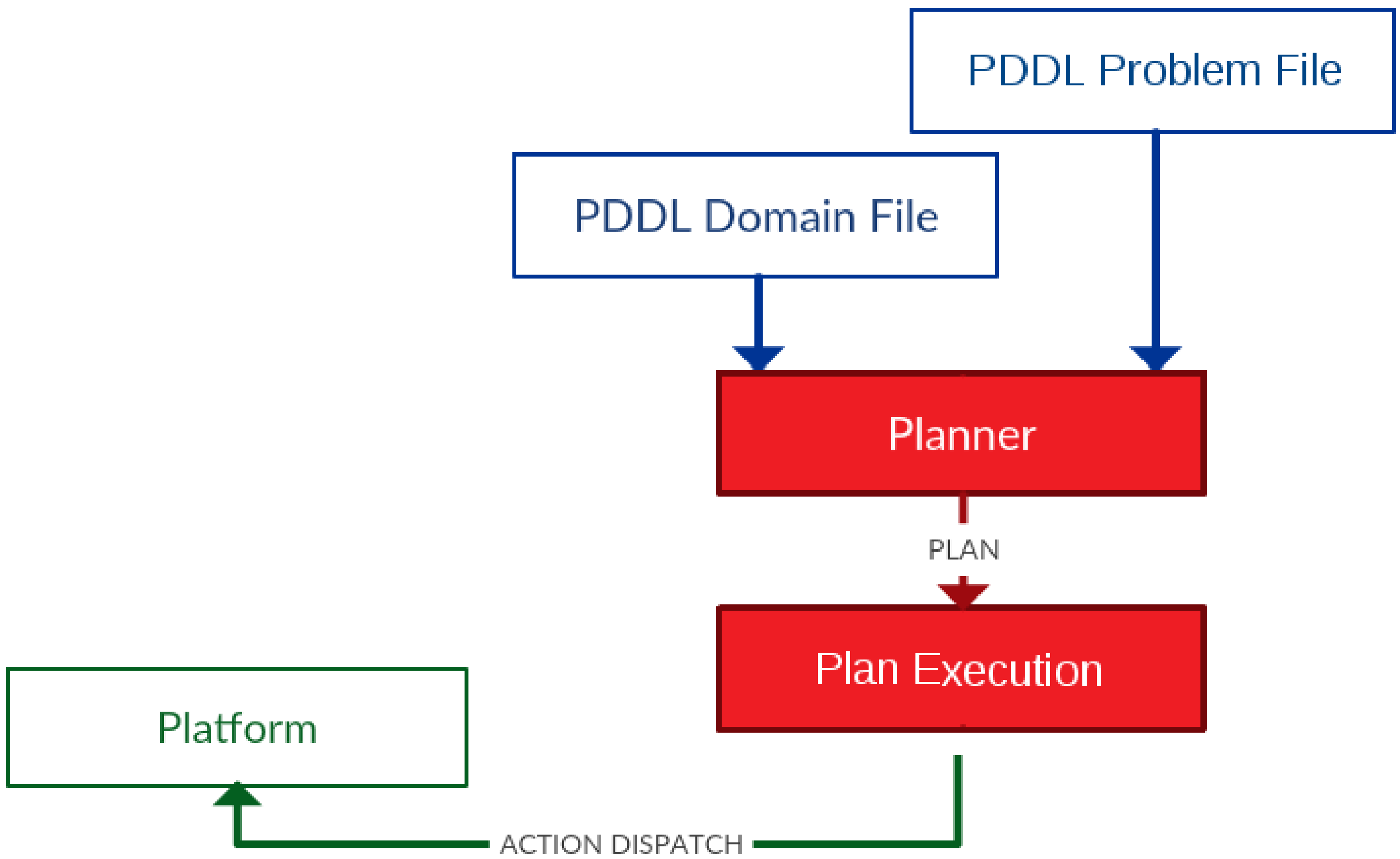
?wp)

PDDL Domain File

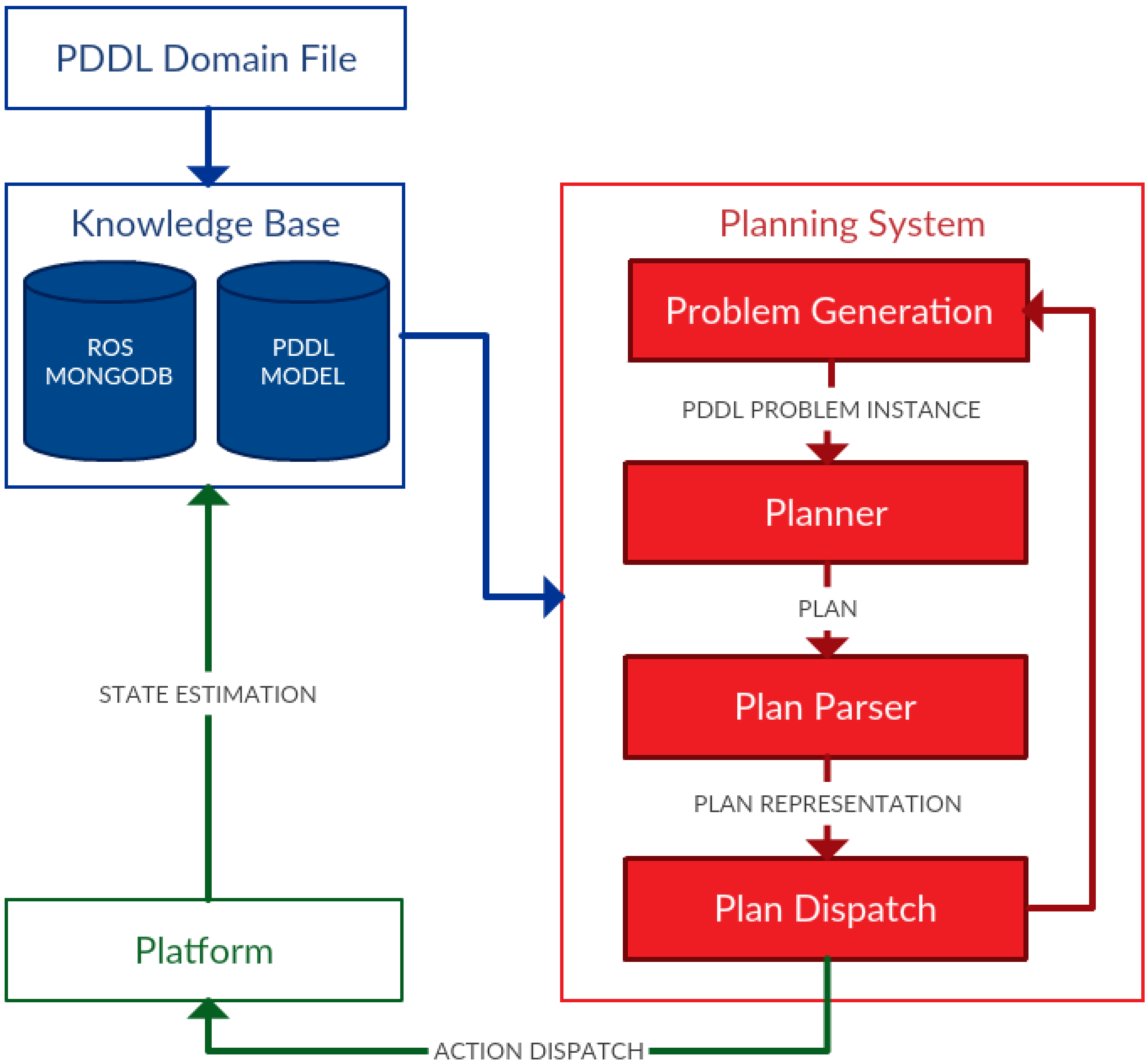
nav_msgs/Odometry
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3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

ROSPlan components



ROSPlan components



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.

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.

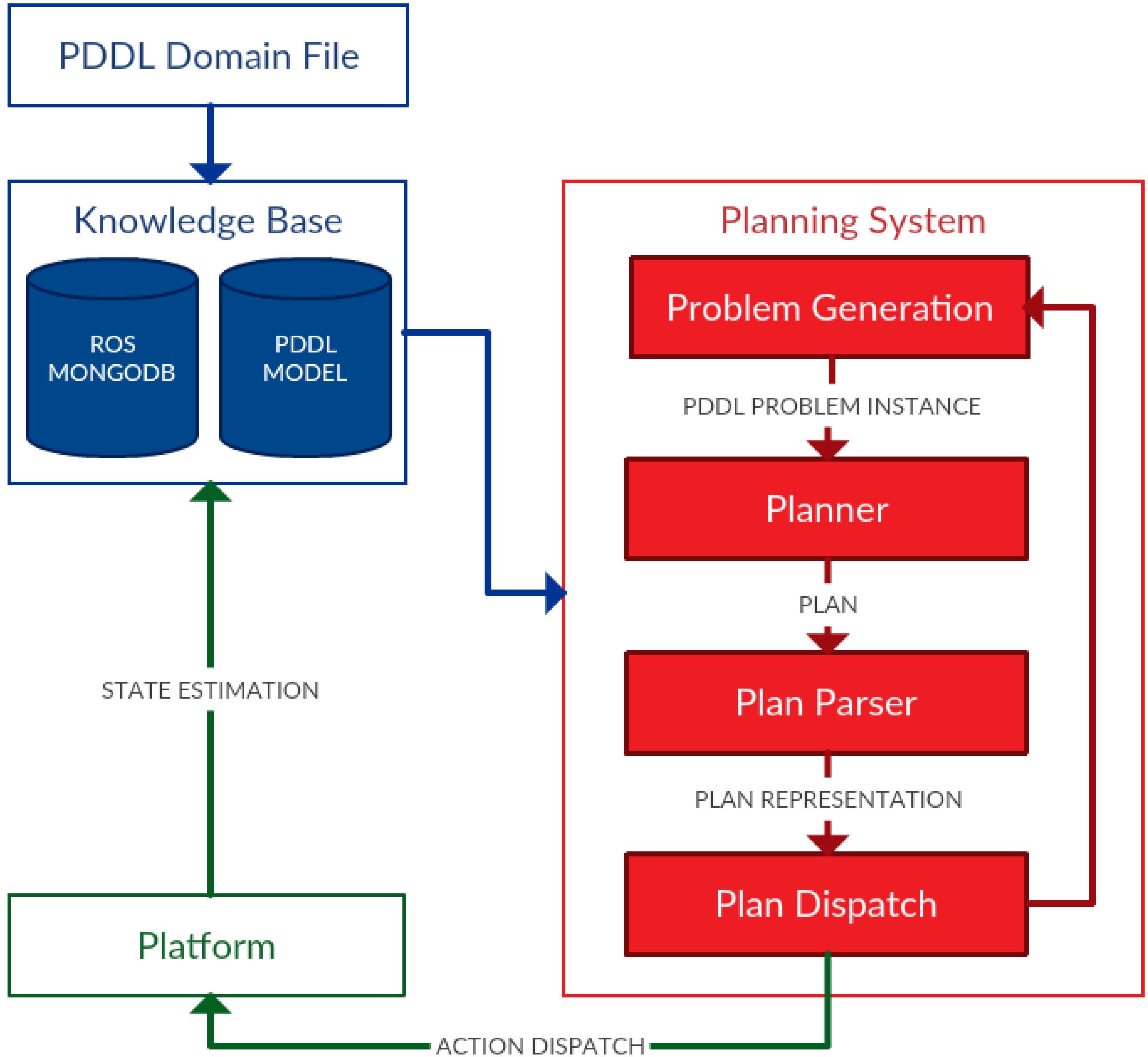
If the plan will fail in the future, the robot should not continue to execute the plan for a long time without purpose.

The success or failure of an action can sometimes not be understood outside of the context of the whole plan.

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.



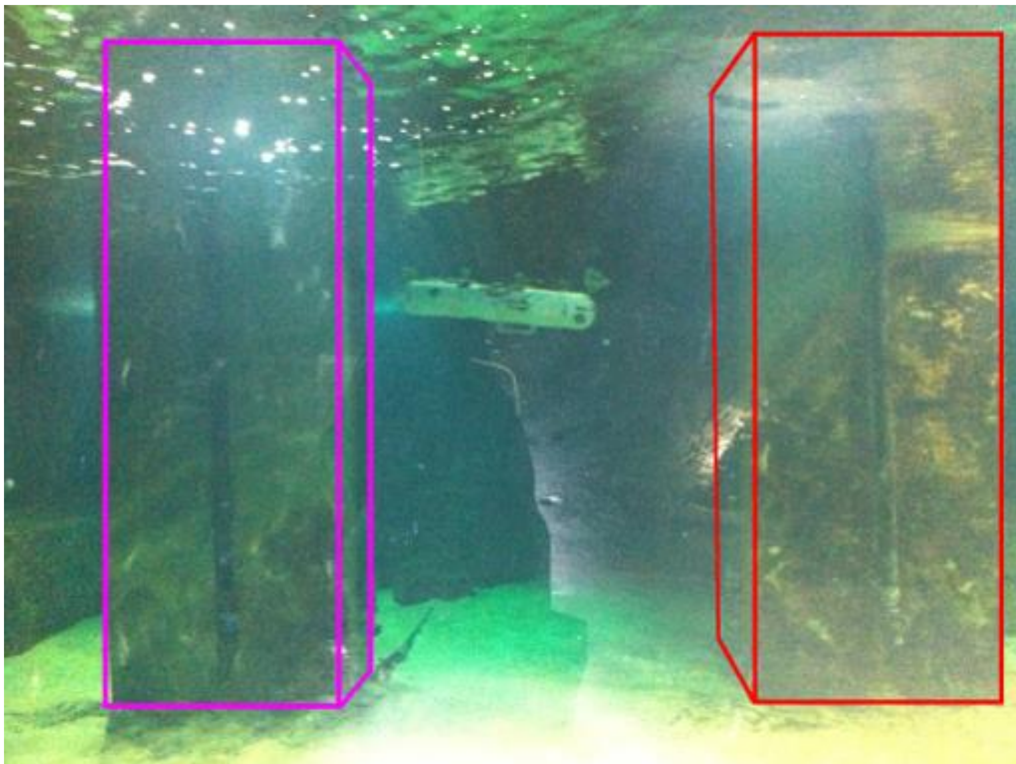
Bad Behaviour 2: Plan Failure

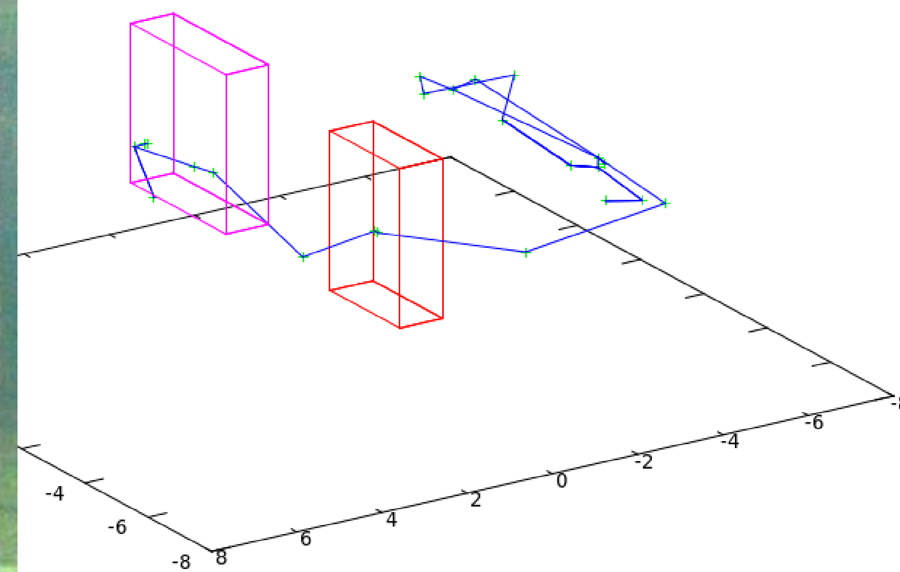
The AUV plans for inspection missions, recording images of pipes and welds.

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.



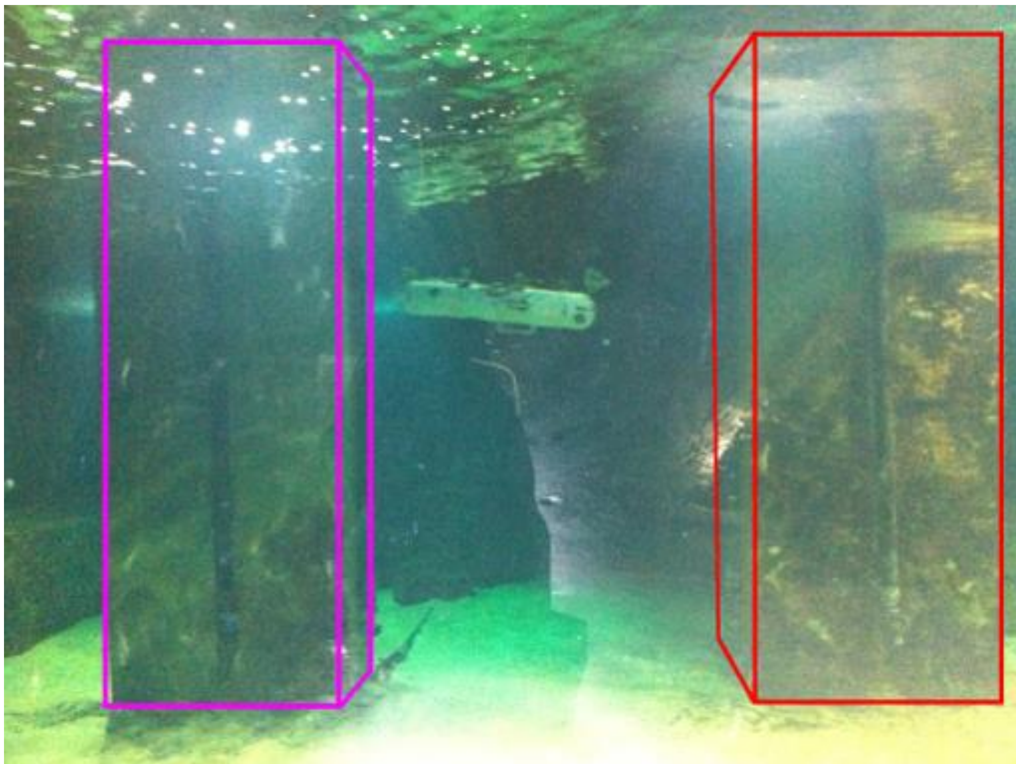


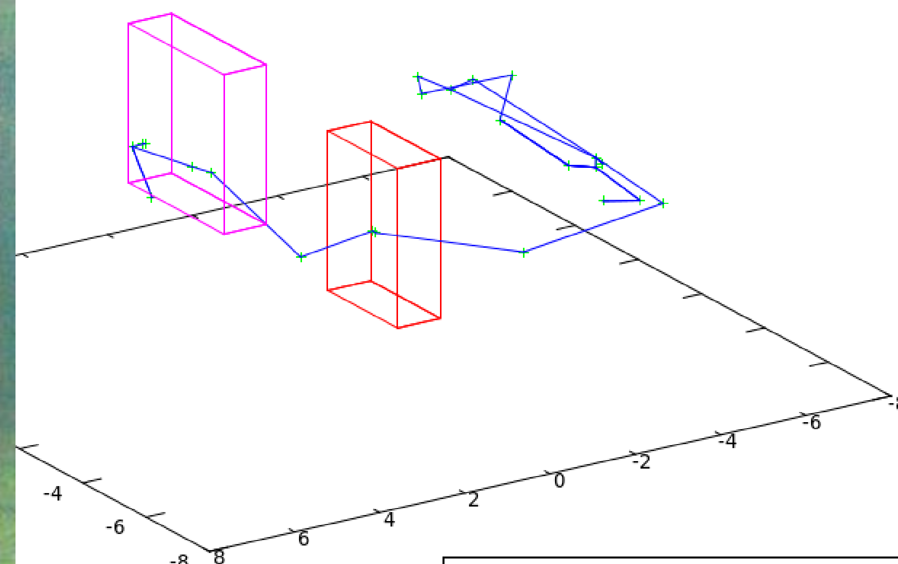
The planned inspection path is shown on the right. The AUV will move around to the other side of the pillars before inspecting the pipes on their facing sides.

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.





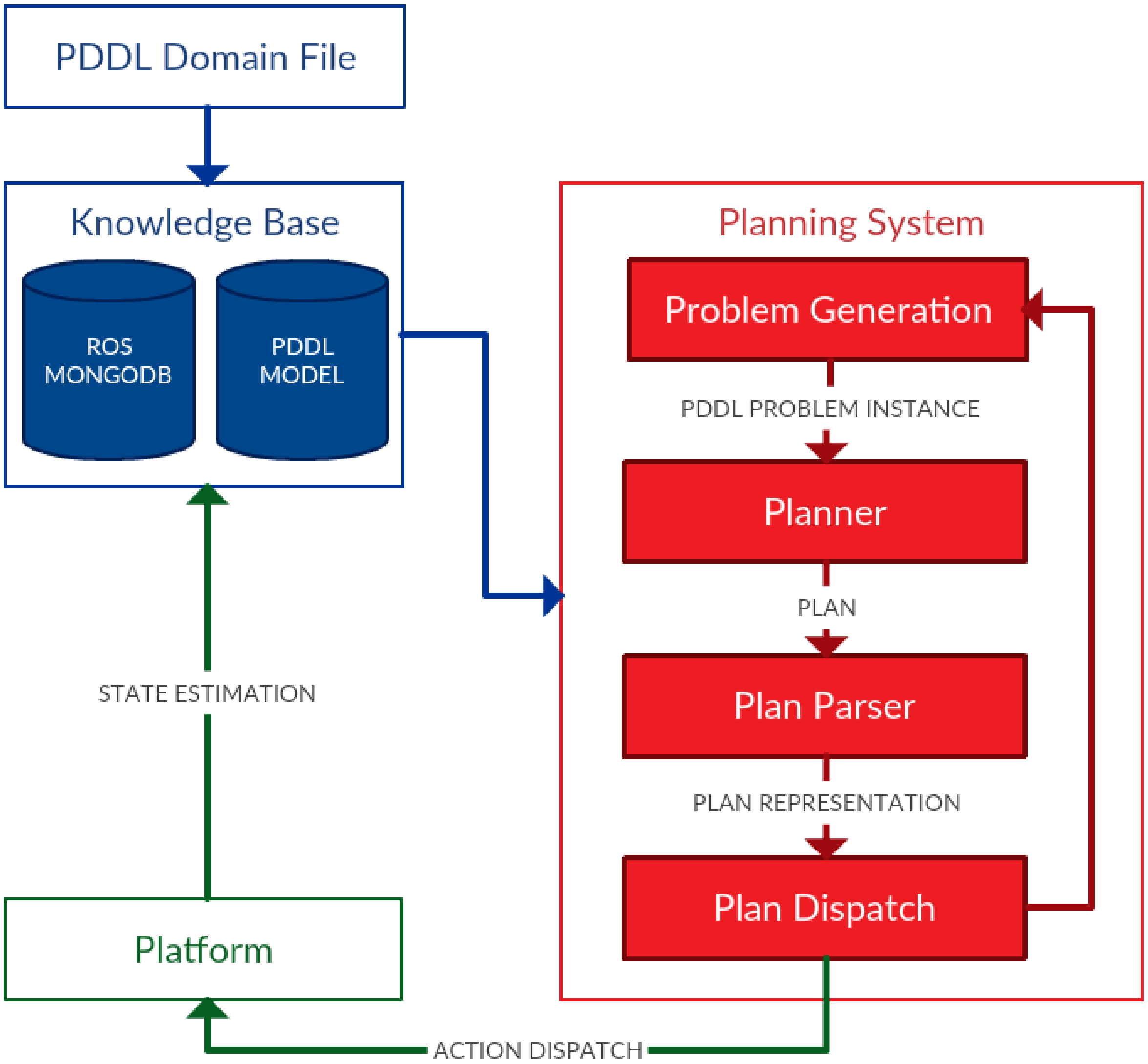
ROSPlan validates using VAL. [Fox et al. 2005]

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

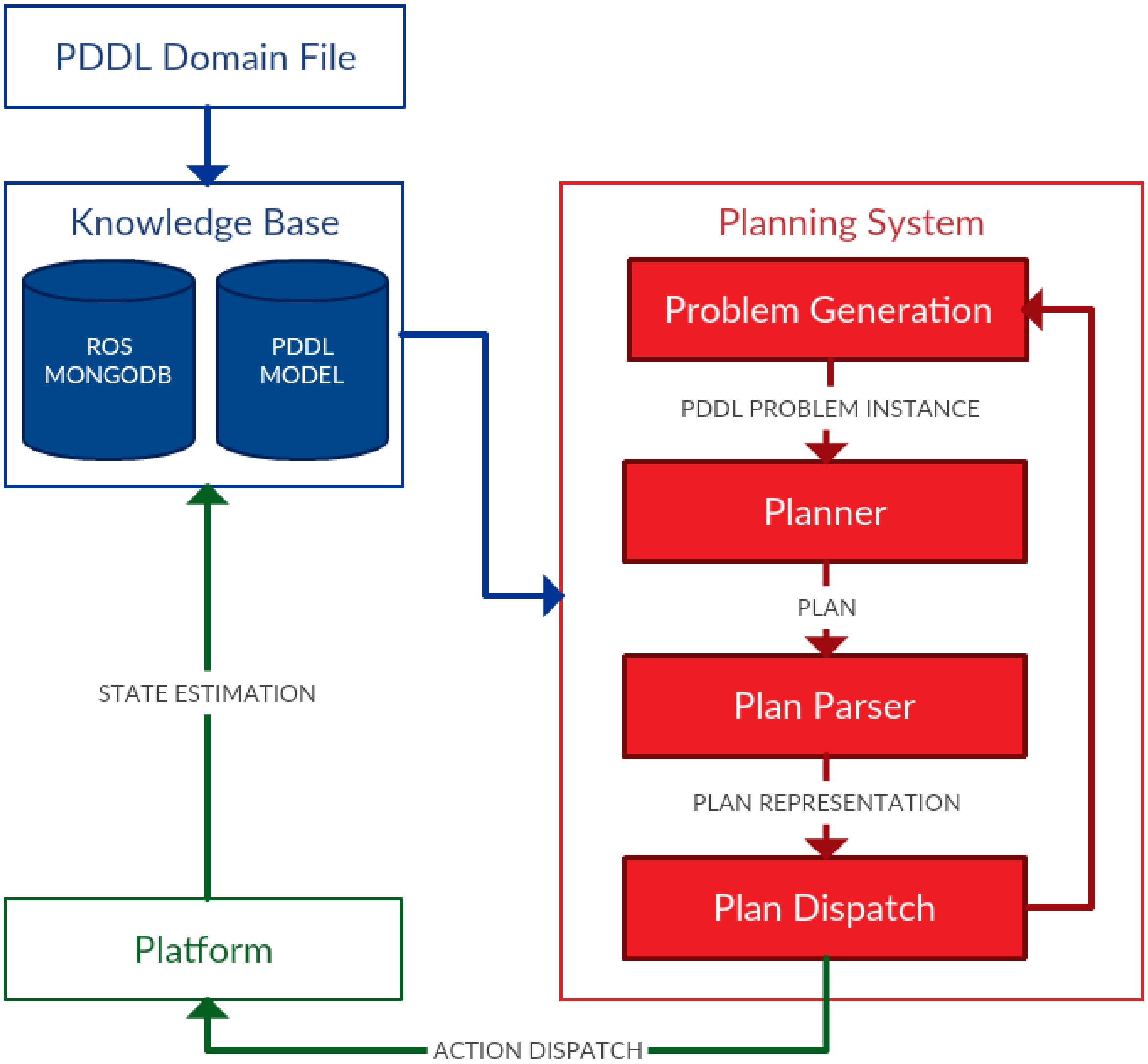
Now the system is more complex:

- 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.

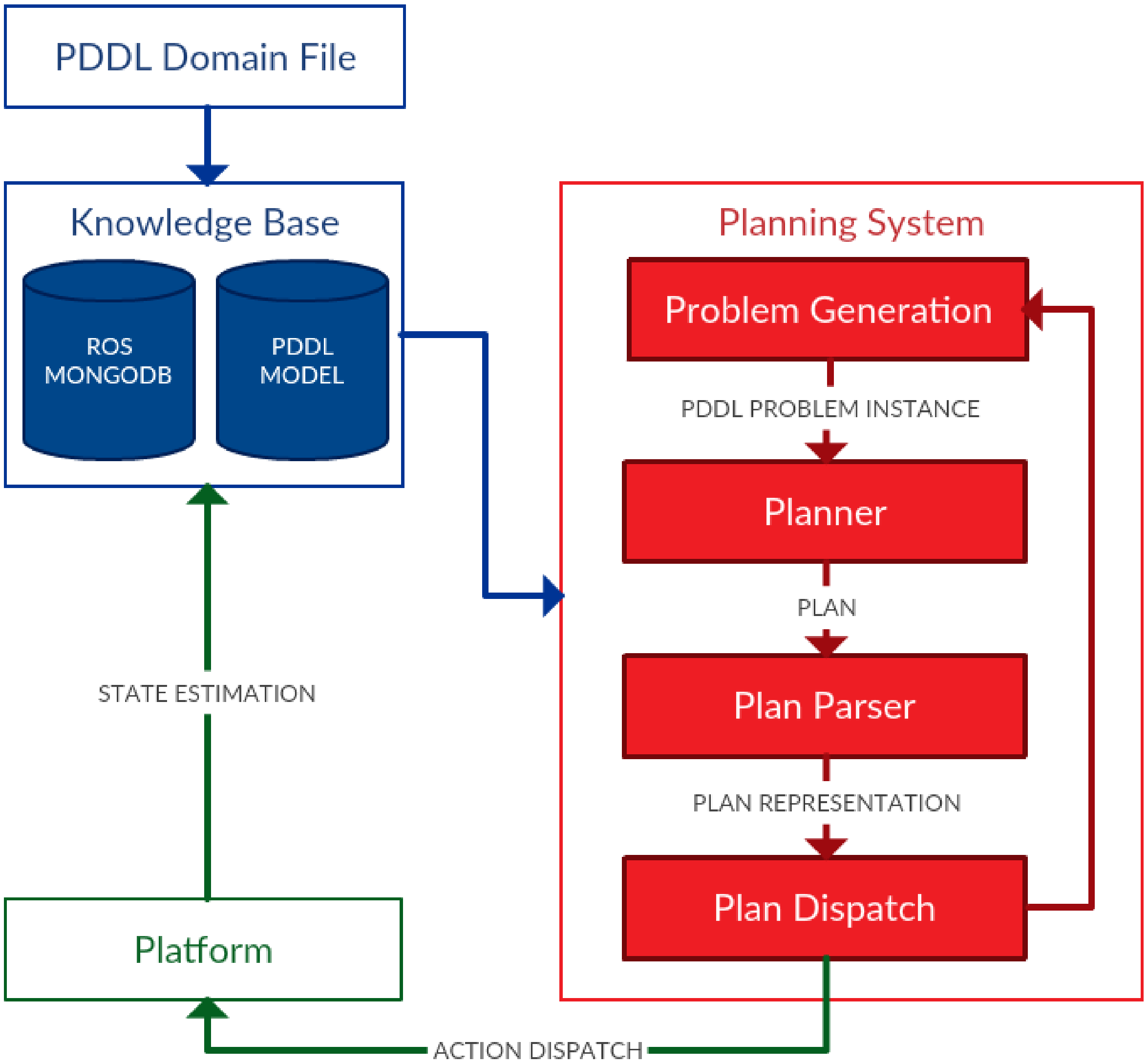
- problem file is automatically generated.

- 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.

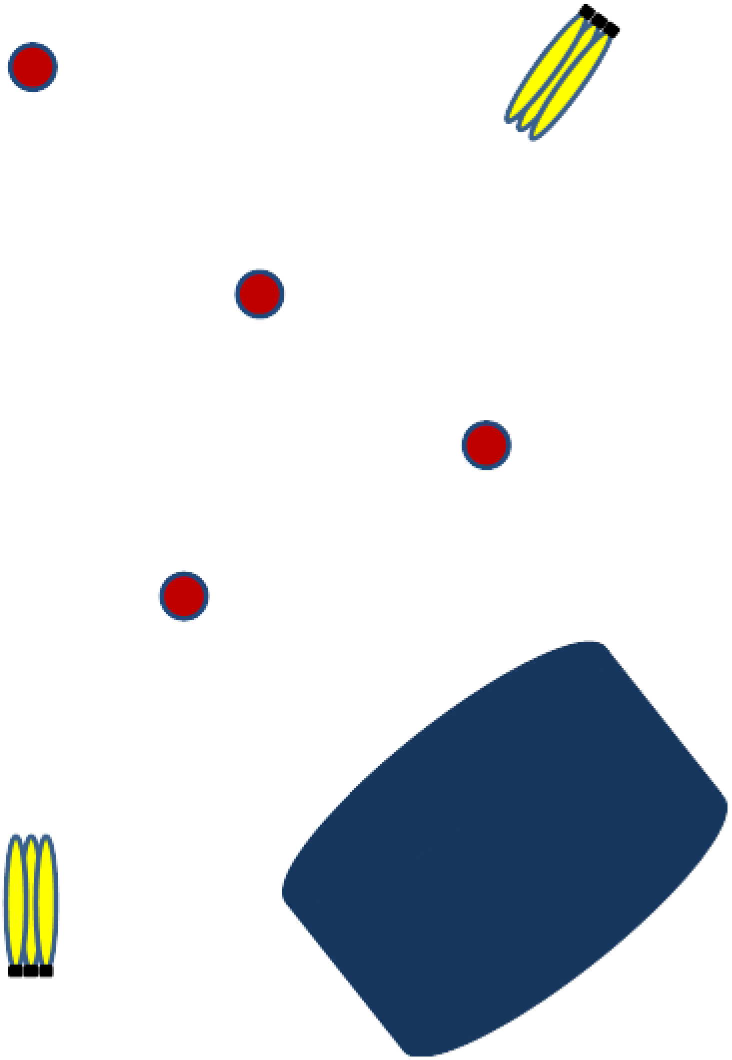


Plan Execution 2: Very Simple Temporal Dispatch

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?

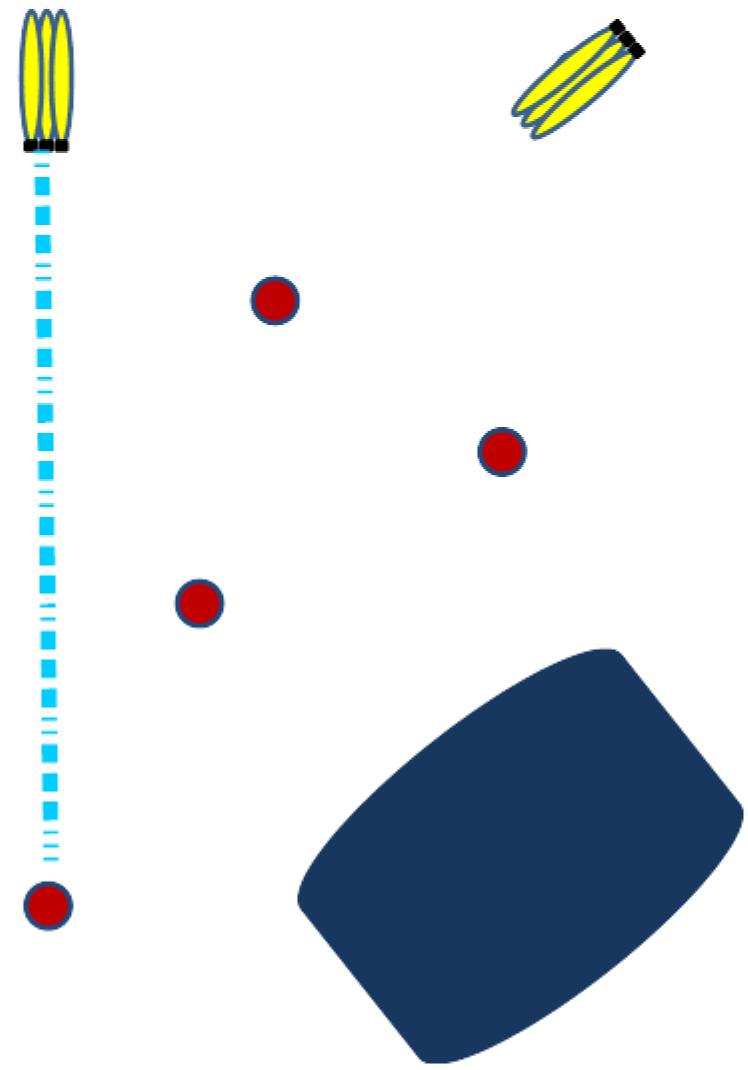


Plan Execution 2: Very Simple Temporal Dispatch

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?

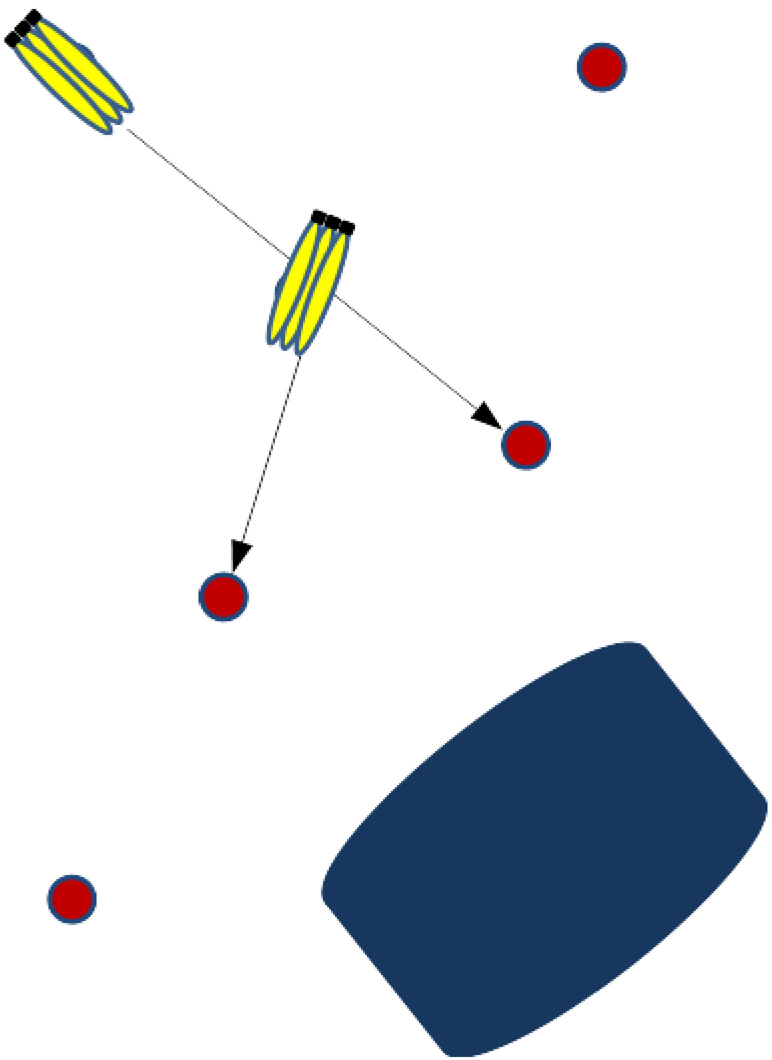


Plan Execution 2: Very Simple Temporal Dispatch

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?

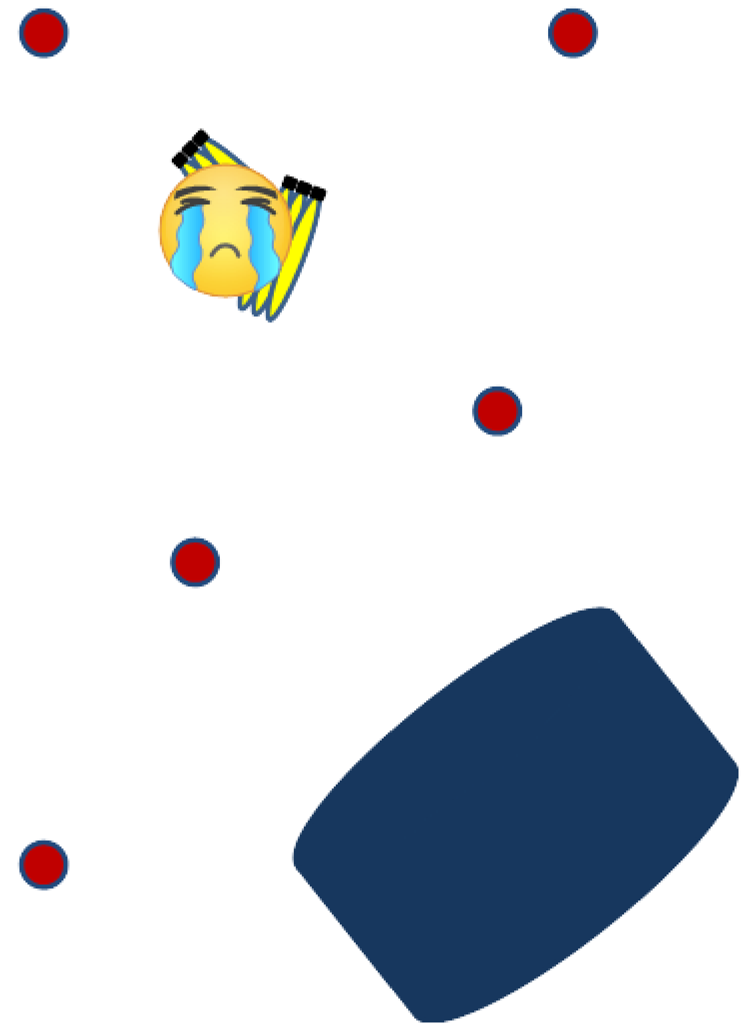


Plan Execution 2: Very Simple Temporal Dispatch

The real world requires a temporal and numeric model:

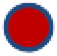
- time and deadlines,
- battery power and consumption,
- direction of sea current, or traffic flow.

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



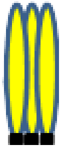
Temporal Constraints

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



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

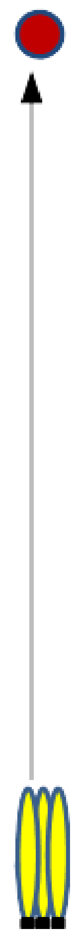
However, in the real world there are many uncontrollable durations and events. The estimated duration of actions is rarely accurate.



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.

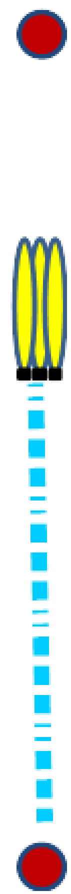


0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: (clean_chain wp2) [60.0]

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.

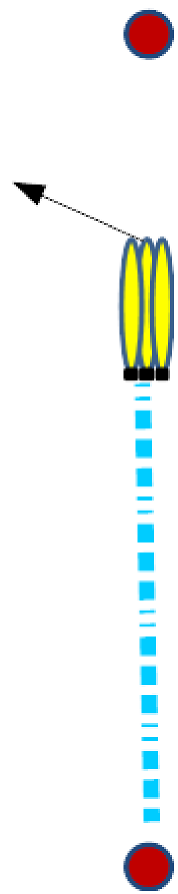


0.000: (goto_waypoint wp1) [10.0]
10.01: (goto_waypoint wp2) [14.3]
24.32: (clean_chain wp2) [60.0]

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.



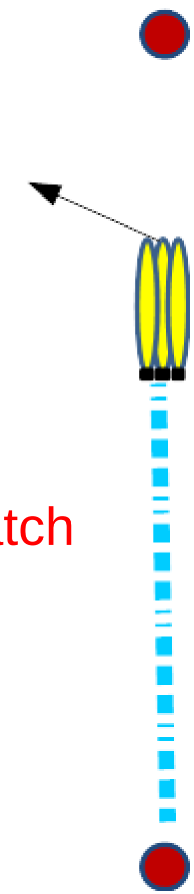
0.000: (goto_waypoint wp1) [10.0]	
10.01: (goto_waypoint wp2) [14.3]	
24.32: (clean_chain wp2) [60.0]	

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]

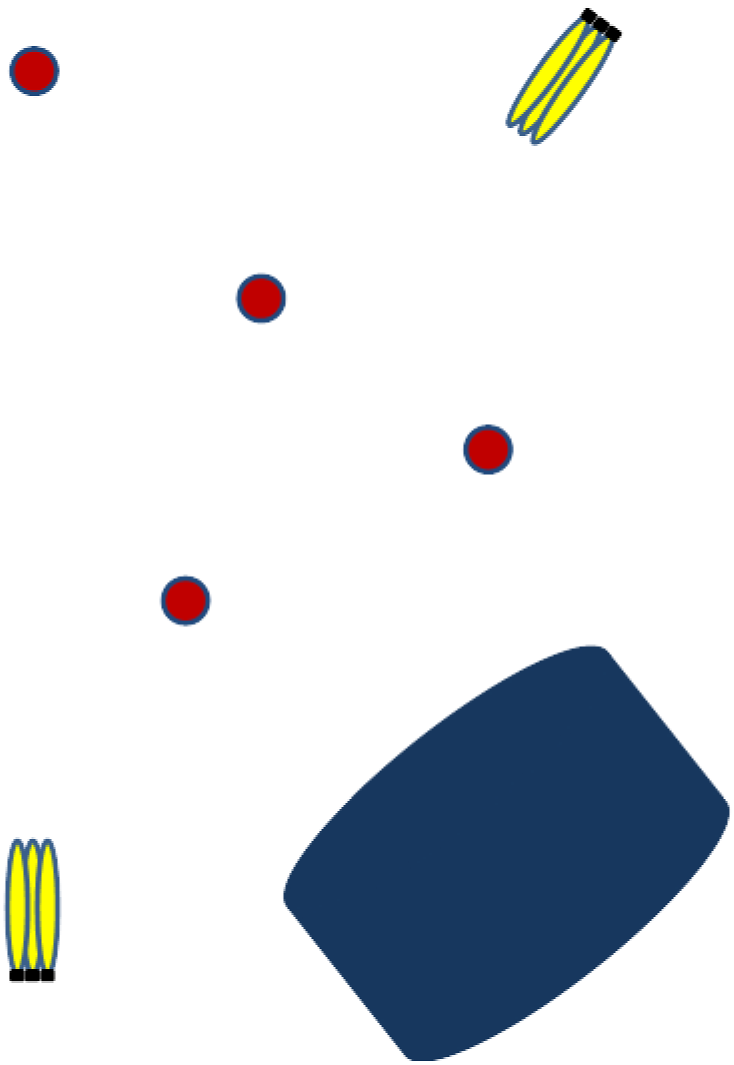
Temporal Constraints

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.



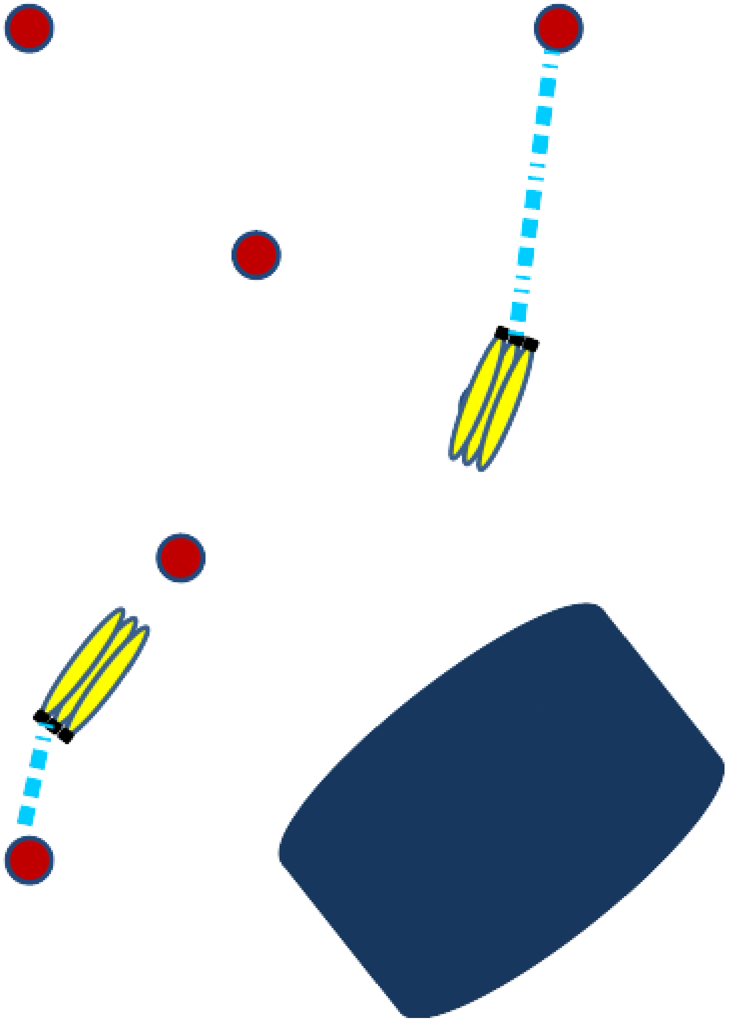
Temporal Constraints

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.



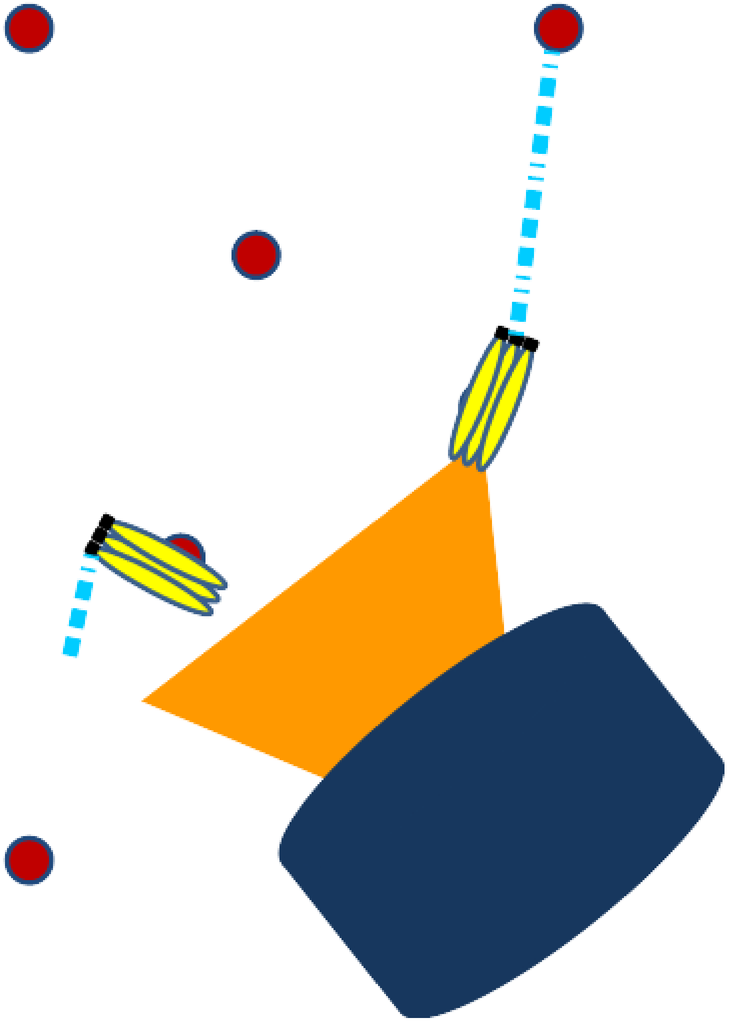
Temporal Constraints

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.



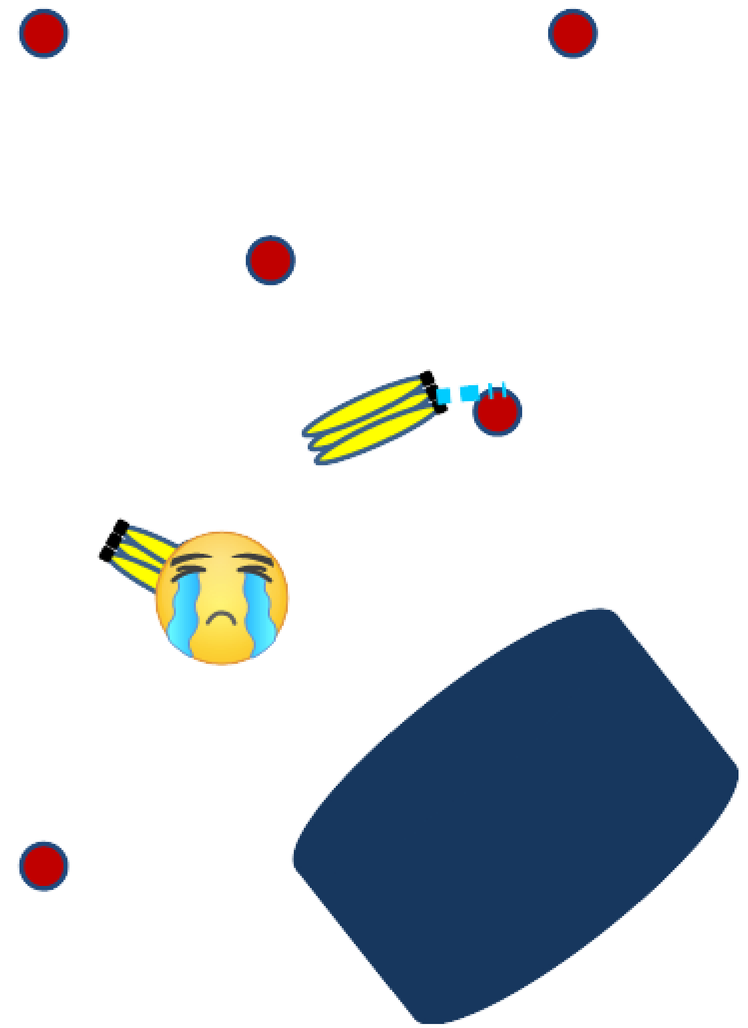
Temporal Constraints

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.

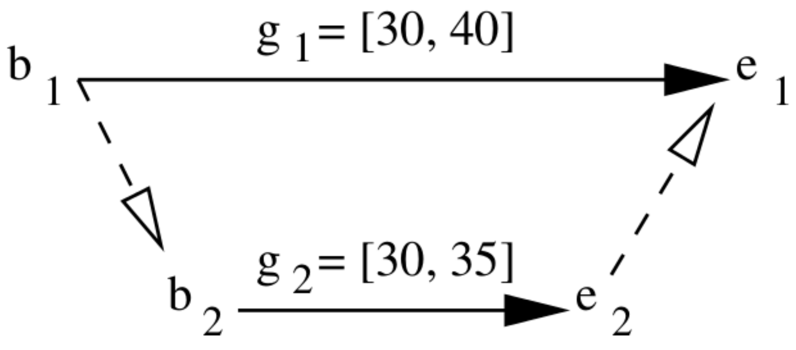
However, some plans require *temporal coordination* between actions, and the controllable durations might be very far apart.



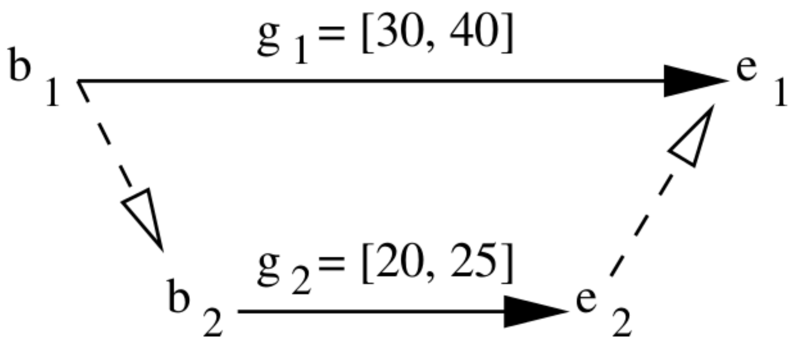
STPUs: Strong controllability

An STPU is strongly controllable iff:

- the agent can commit (in advance) to a time for all activated time-points,
- for any possible time for received time points, the temporal constraints are not violated.



(a)

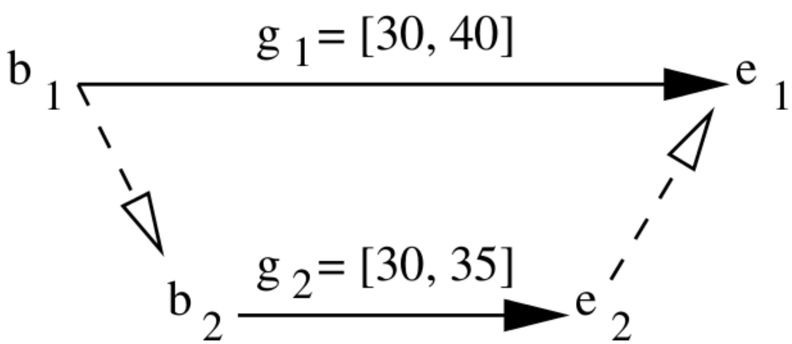


(b)

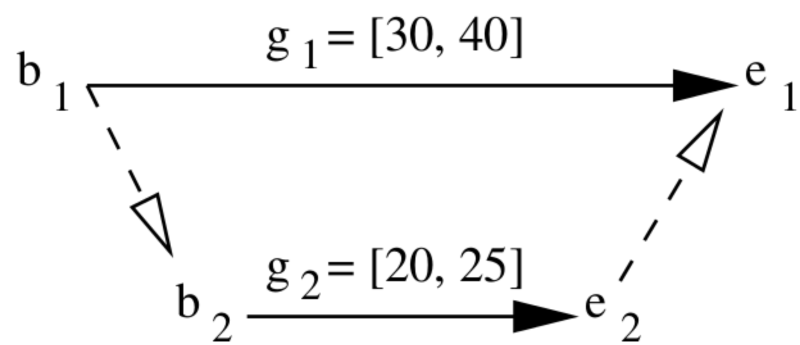
STPUs: Strong controllability

An STPU is strongly controllable iff:

- the agent can commit (in advance) to a time for all activated time-points,
- for any possible time for received time points, the temporal constraints are not violated.



(a)



(b)

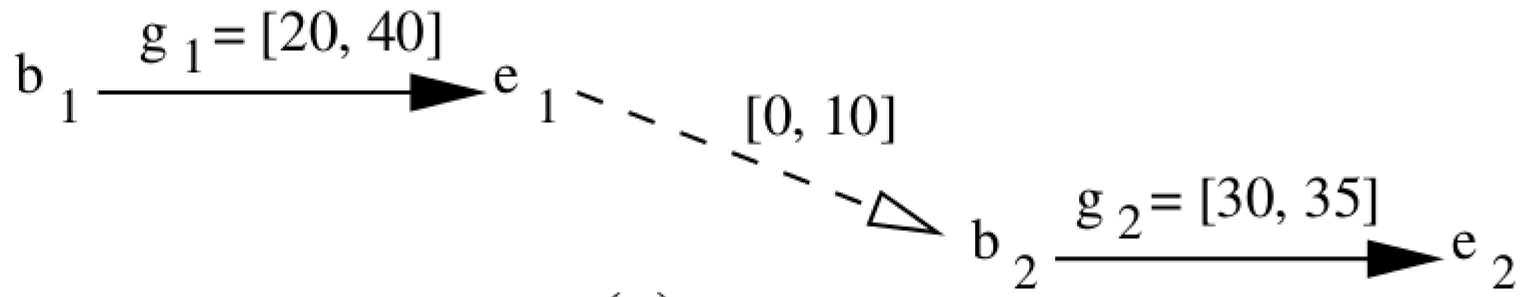
Setting *t(b1) == t(b2)* will always obey the temporal constraints.

STPUs: Strong controllability

An STPU is strongly controllable iff:

- the agent can commit (in advance) to a time for all activated time-points,

- 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.
It is dynamically controllable.

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.

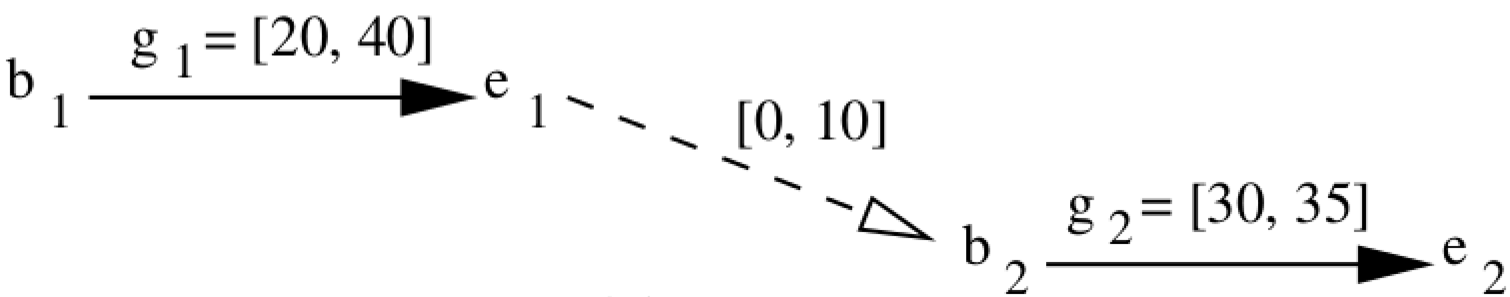
In this case, the agent does not have to commit to a time for any activated time points in advance.

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)

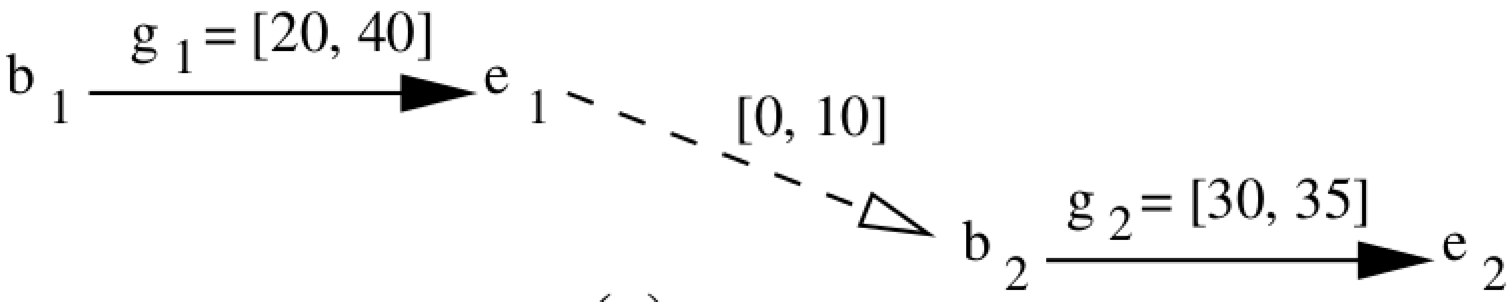
STPUs: Dynamic controllability

Not all problems will have solutions have any kind of controllability.

This does not mean they are impossible to plan or execute.

To reason about these kinds of issues we need to use a plan representation sufficient to capture

- the difference between controllable and uncontrollable durations,
- causal orderings, and
- temporal constraints.



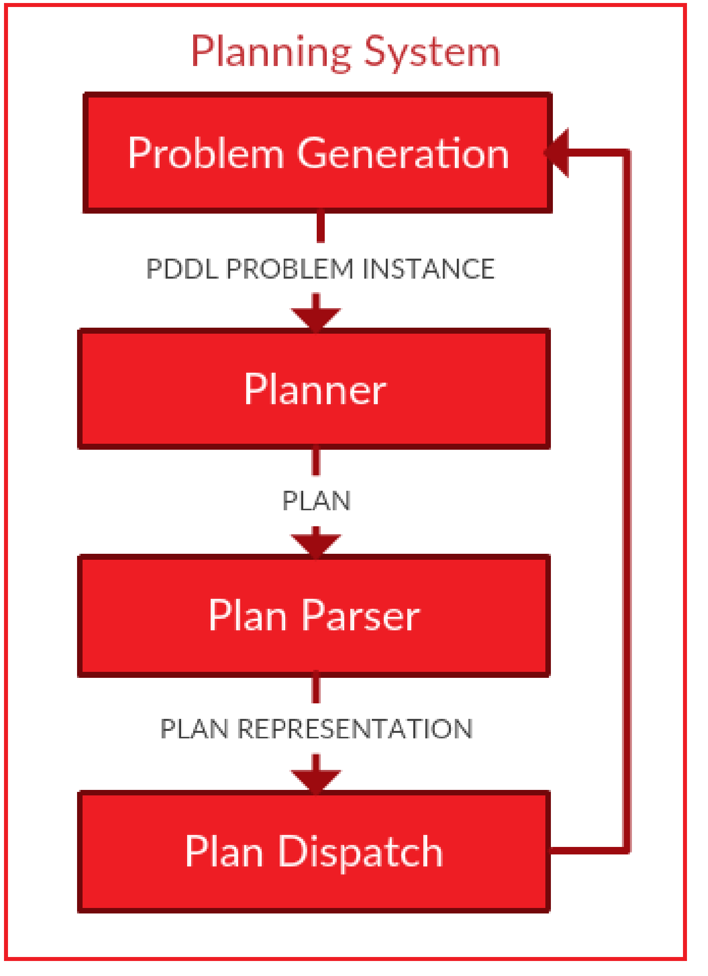
(c)

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.
- Humans are unpredictable.
- The environment is often initially full of unknowns.

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

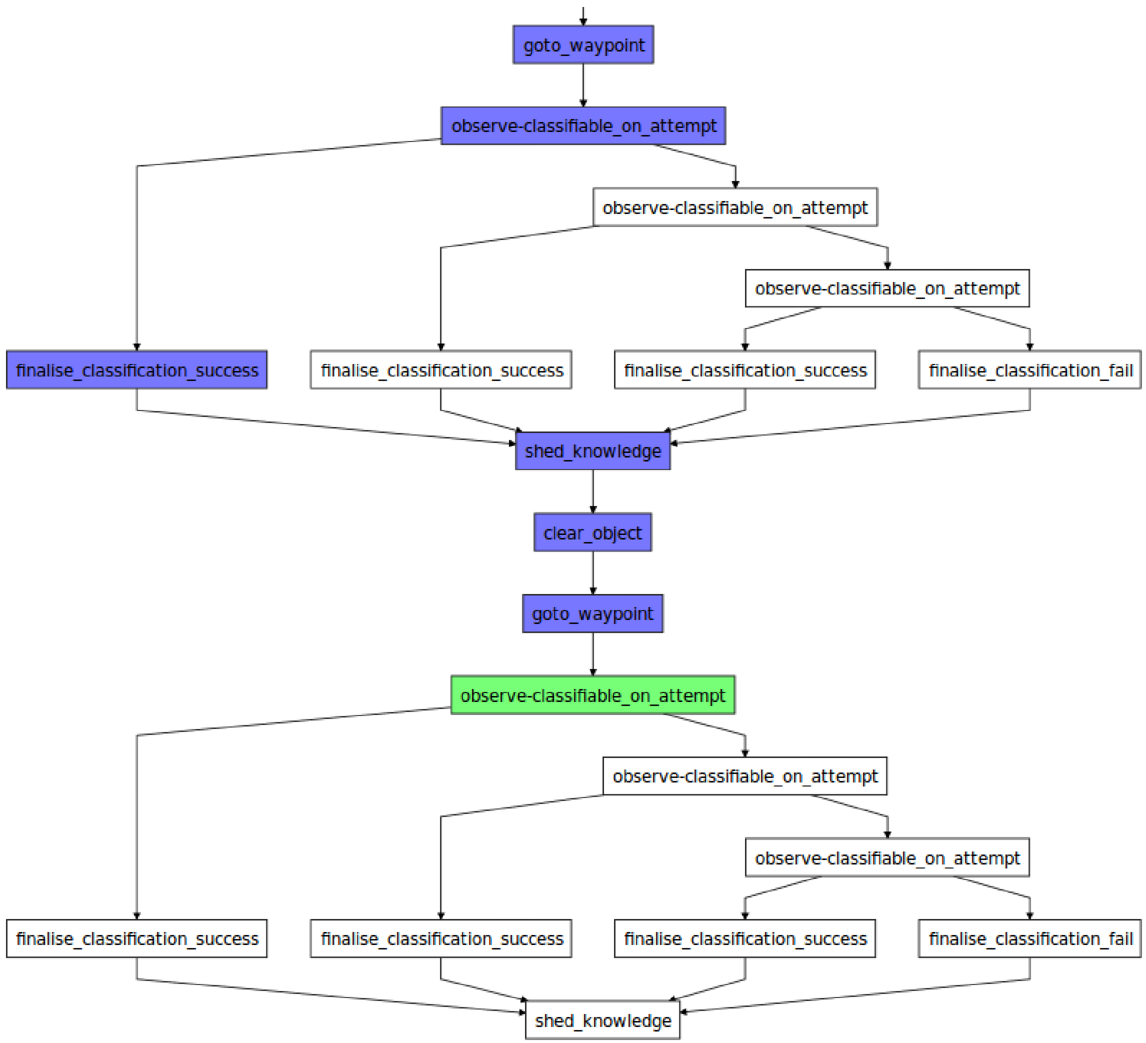
Uncertainty in AI Planning

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)



Plan Execution 4: Temporal and Conditional Dispatch together

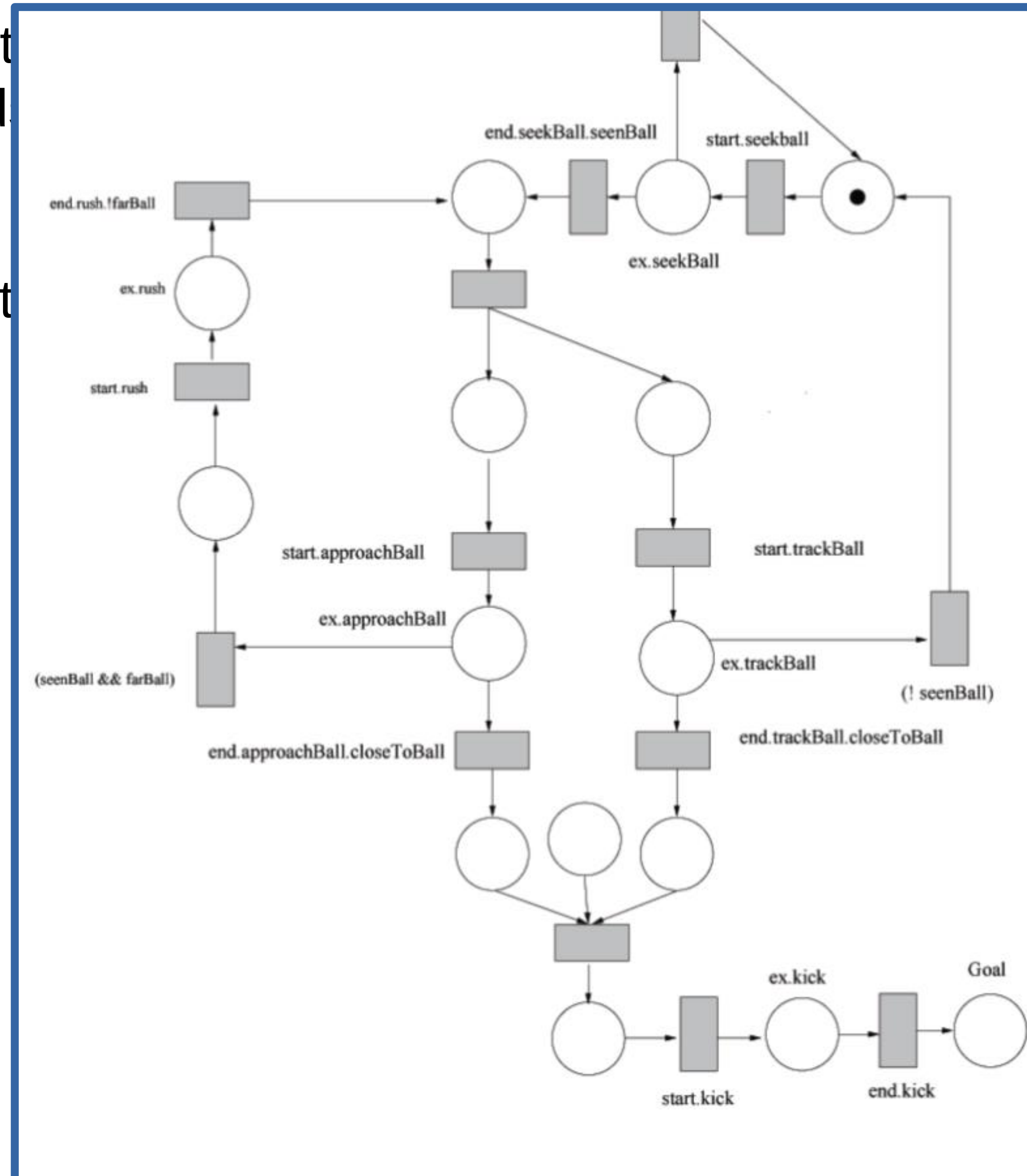
Robotics domains require a combination of temporal and conditional reasoning. Combining these two kinds of uncertainty can result in very complex structures.

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 combinat
reasoning. Combining these two kind
complex structures.

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



Plan Execution 4: Temporal and Conditional Dispatch together

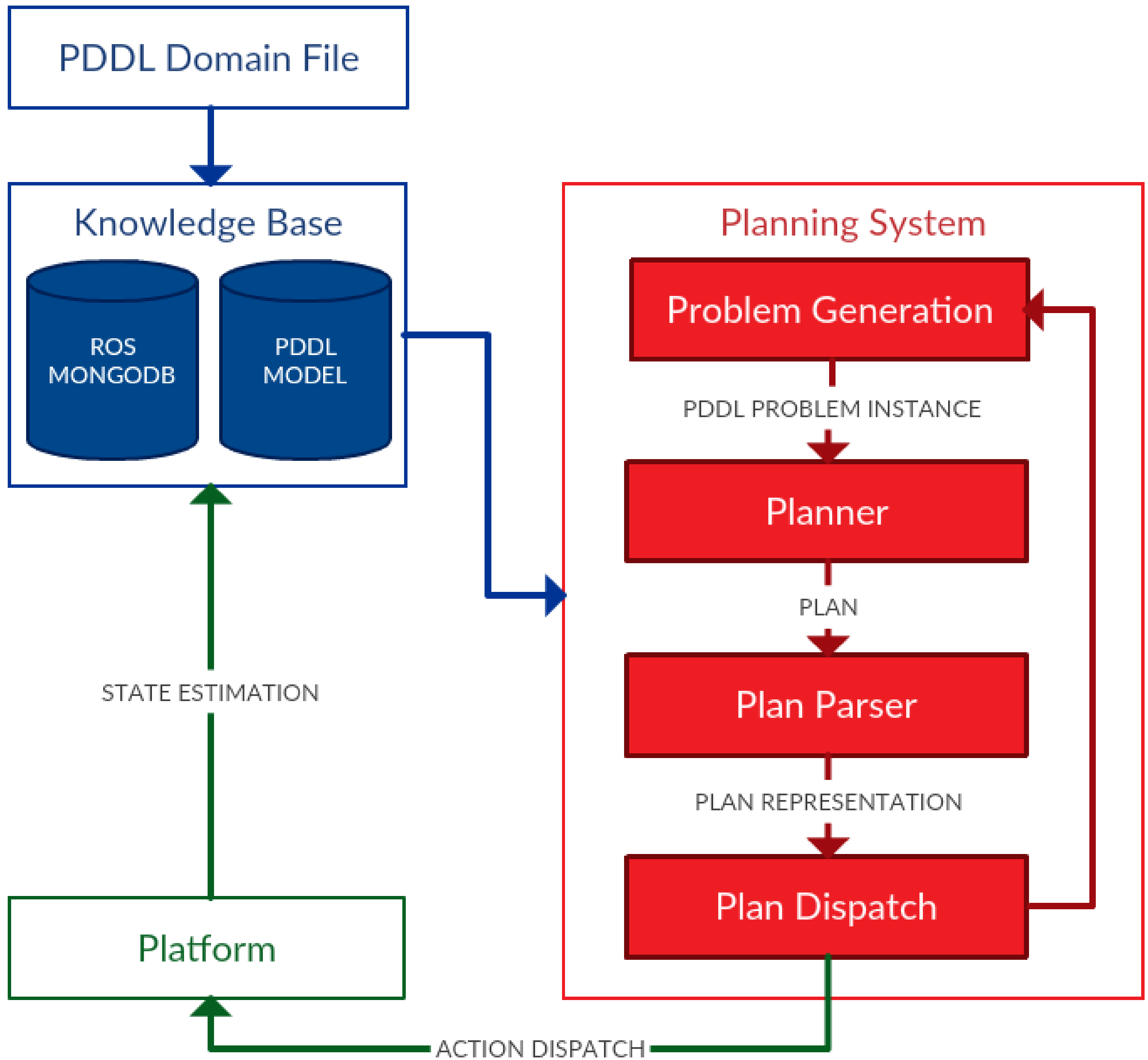
Robotics domains require a combination of temporal and conditional reasoning. Combining these two kinds of uncertainty can result in very complex structures.

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, Cashmore, Magazzeni, and Iocchi; 2017]*

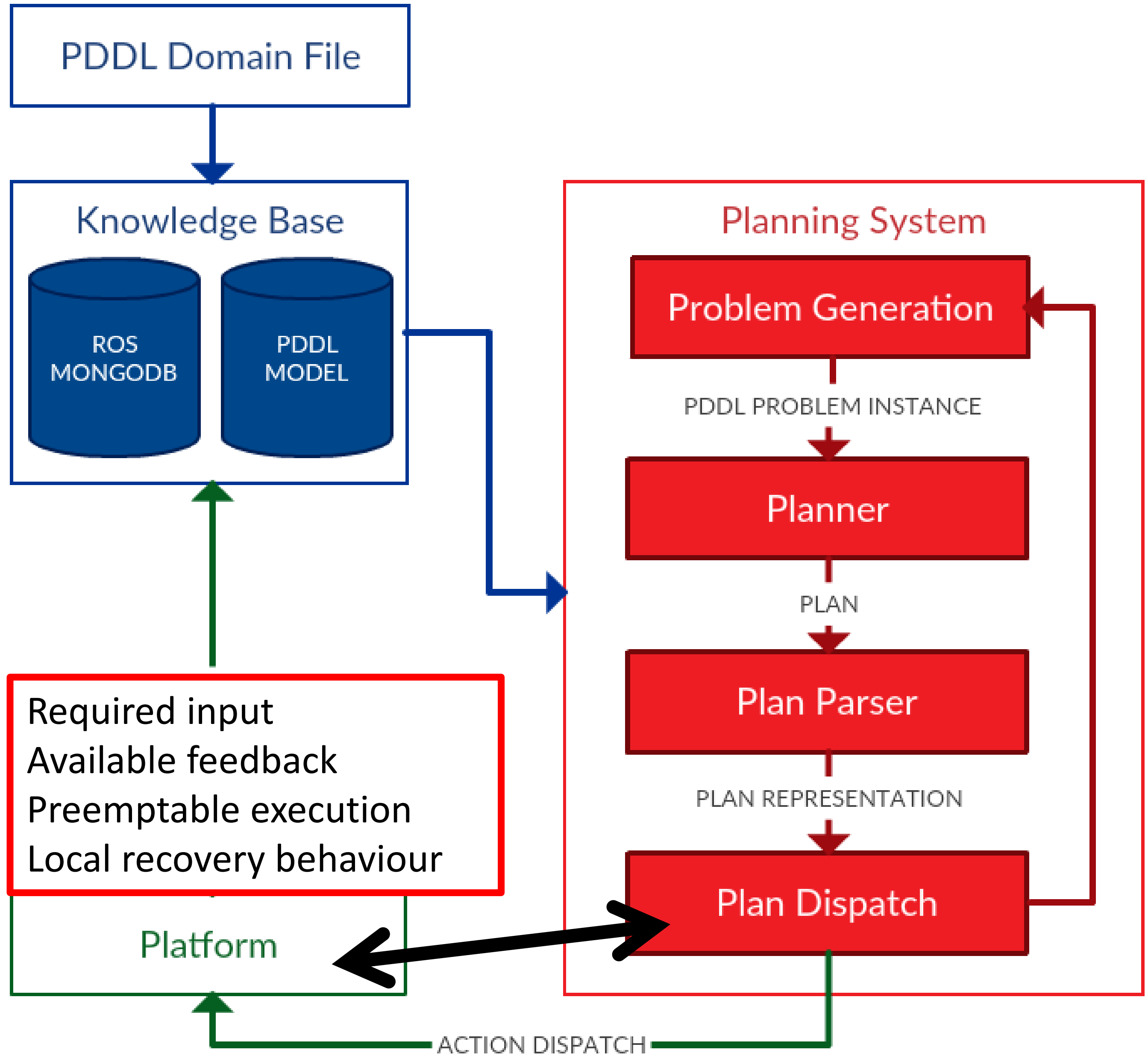
Summary of Very Simple Plan Execution

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.



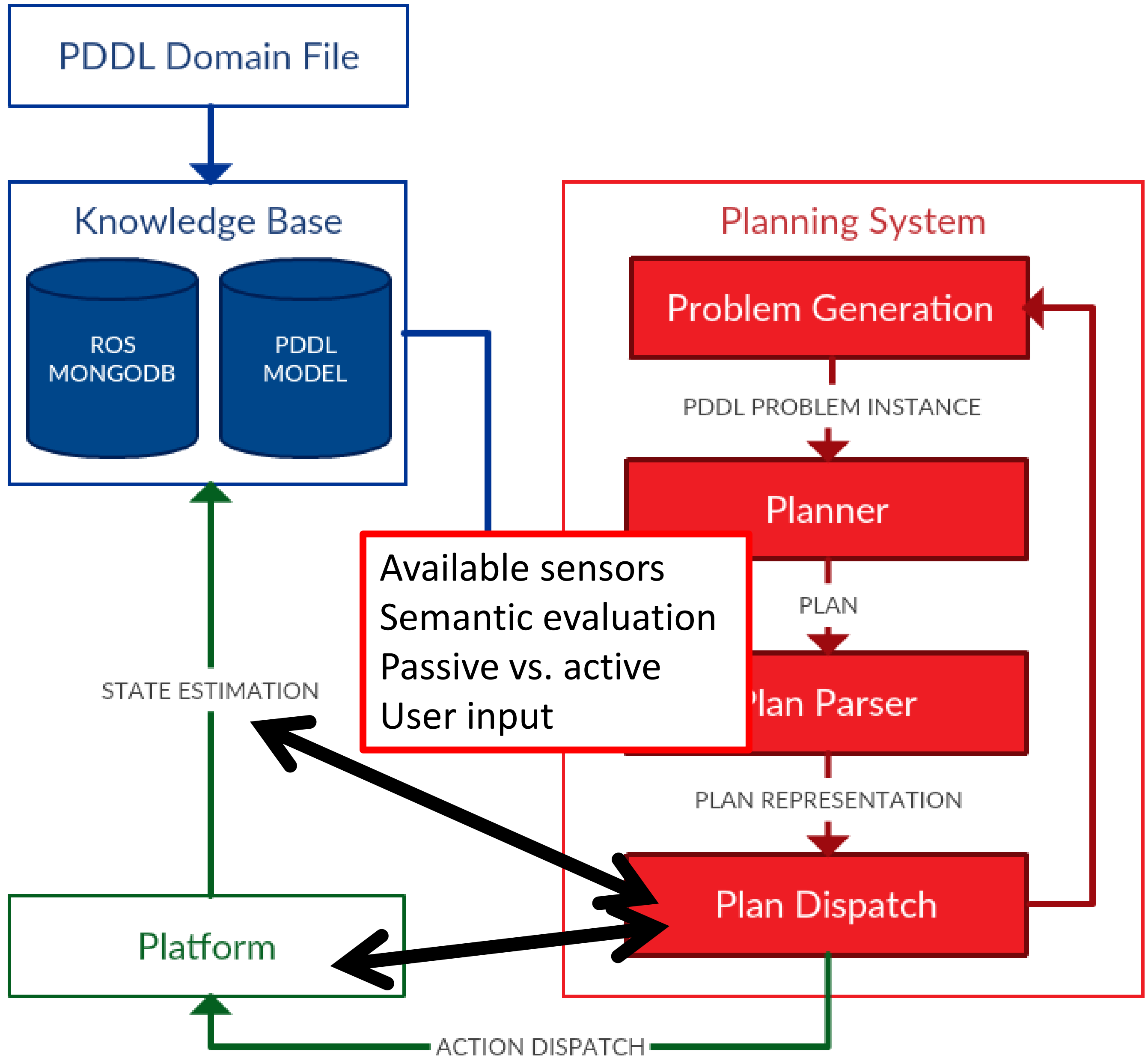
Summary of Very Simple Plan Execution

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.



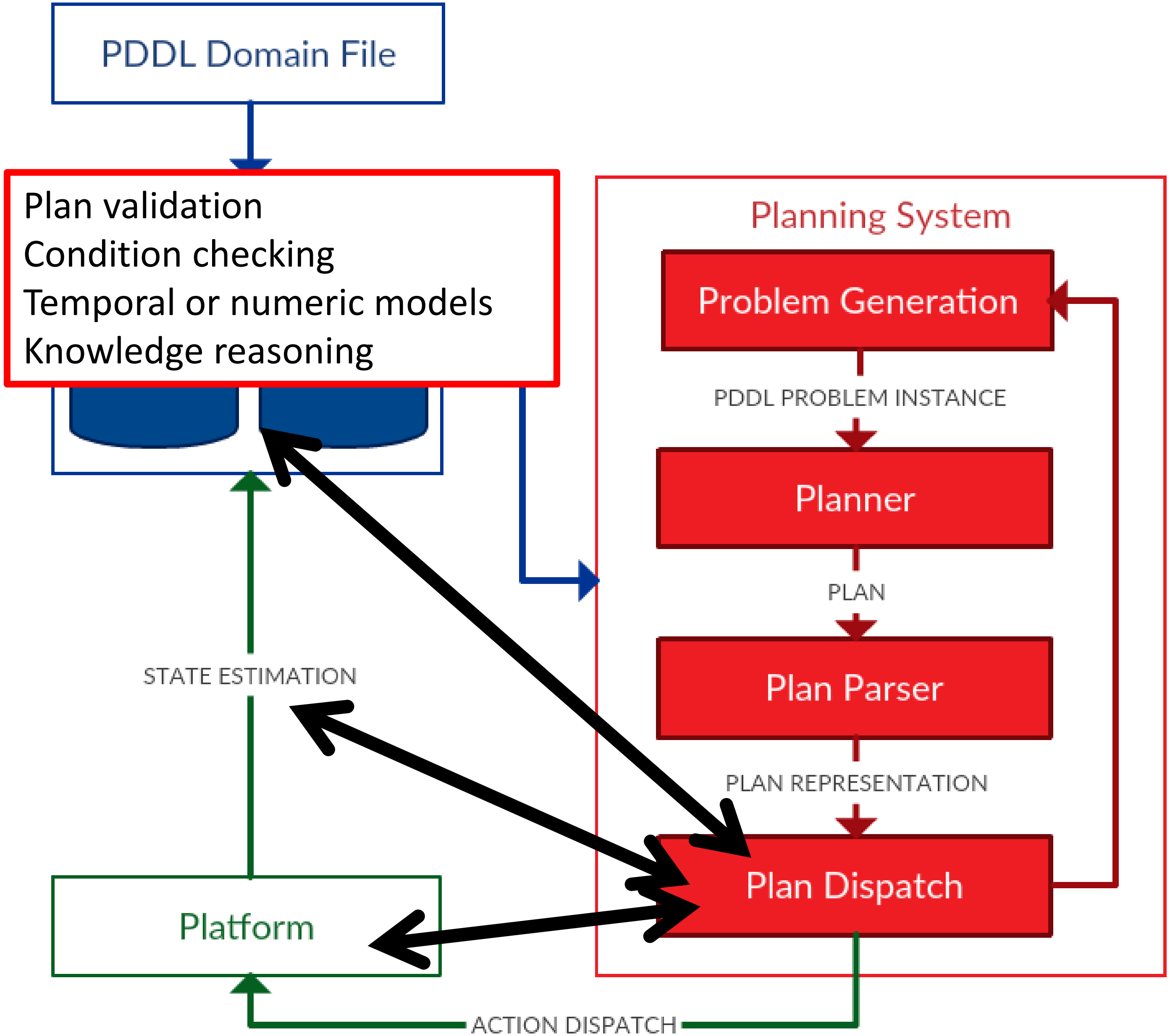
Summary of Very Simple Plan Execution

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.



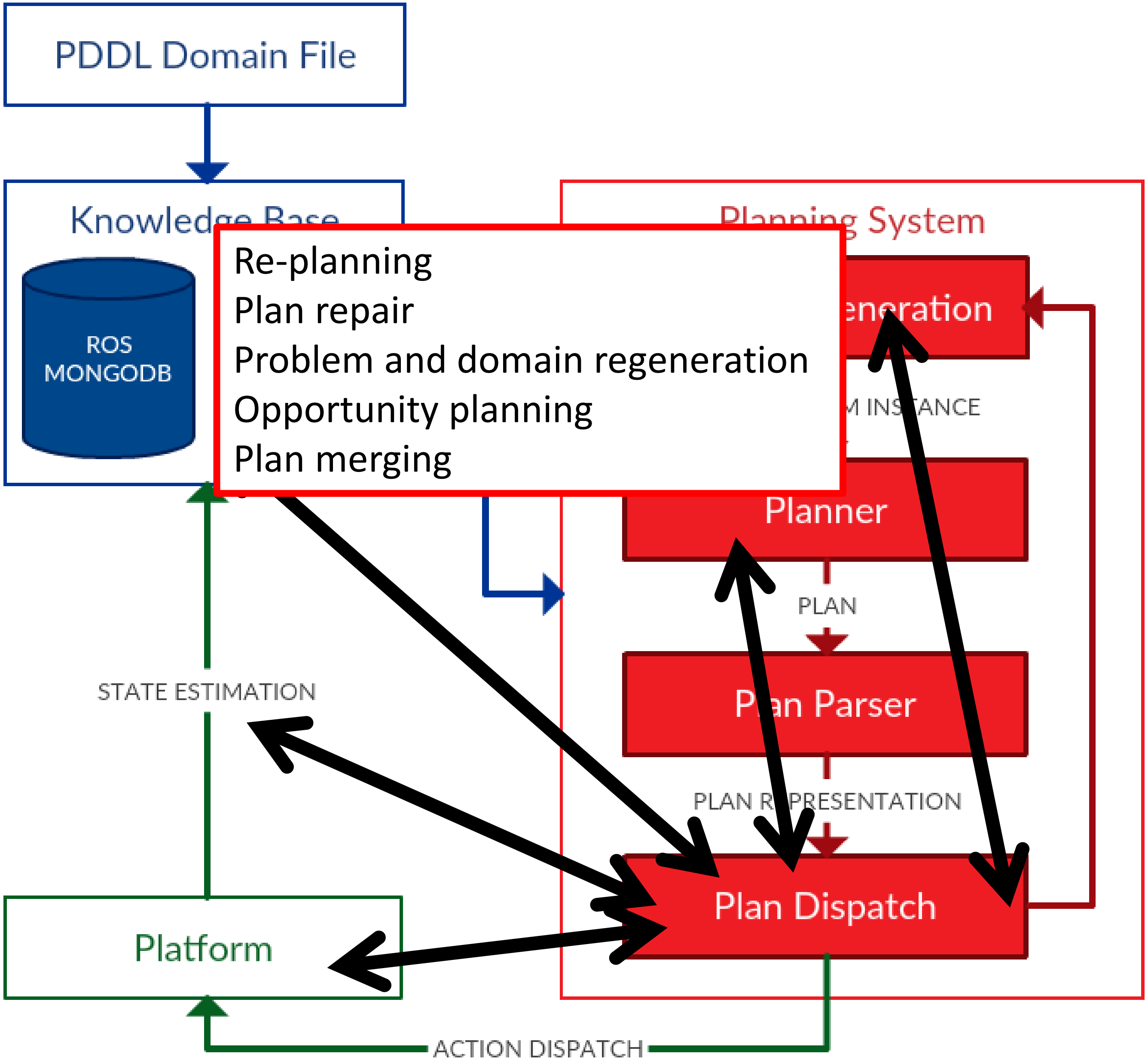
Summary of Very Simple Plan Execution

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

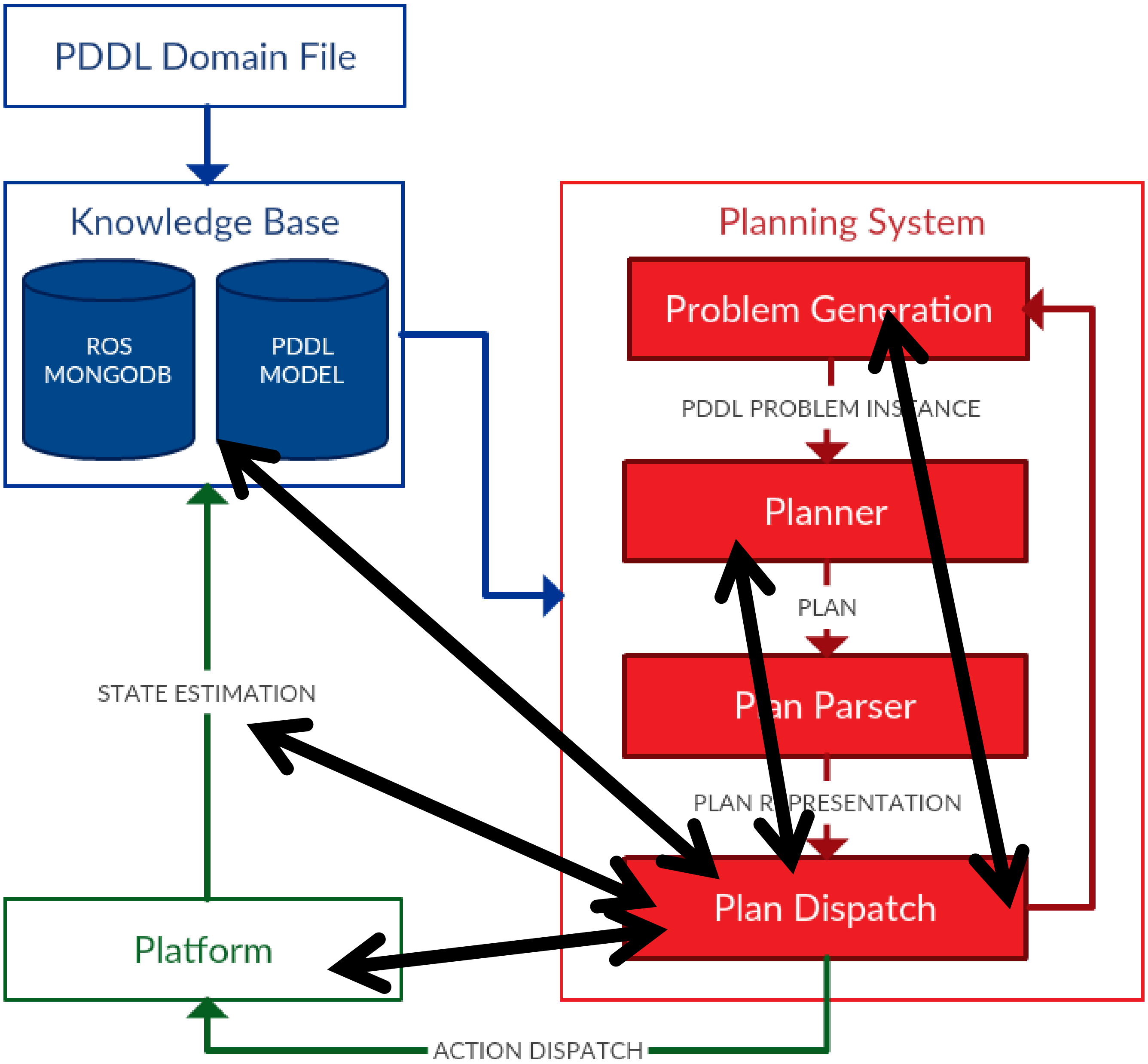
Plan Execution depends upon many components in the system. Changing any one of which will change the robot behaviour, and change the criteria under which the plan will succeed or fail.



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.

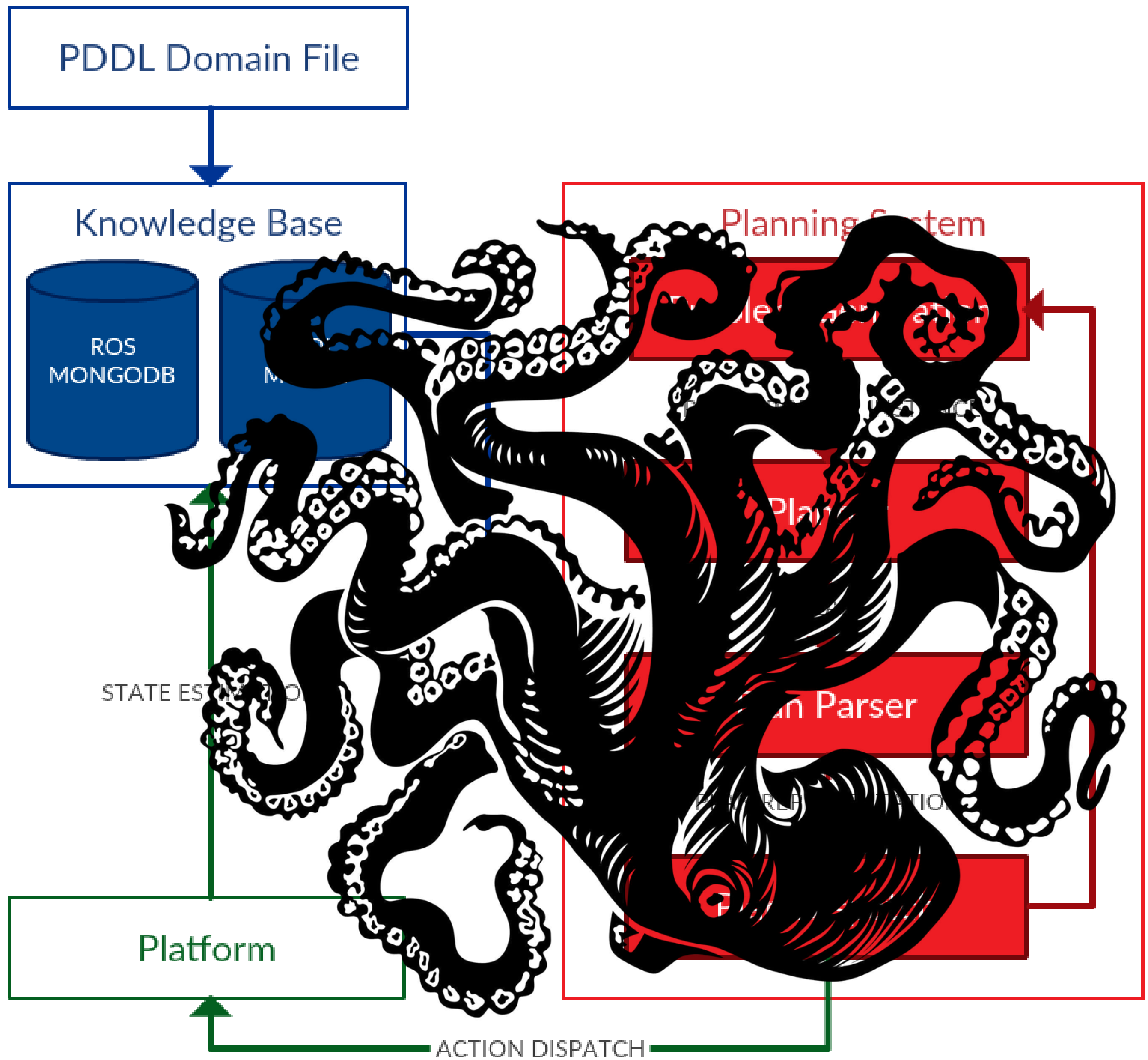
The execution of a plan is an emergent behaviour of the whole system.



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.

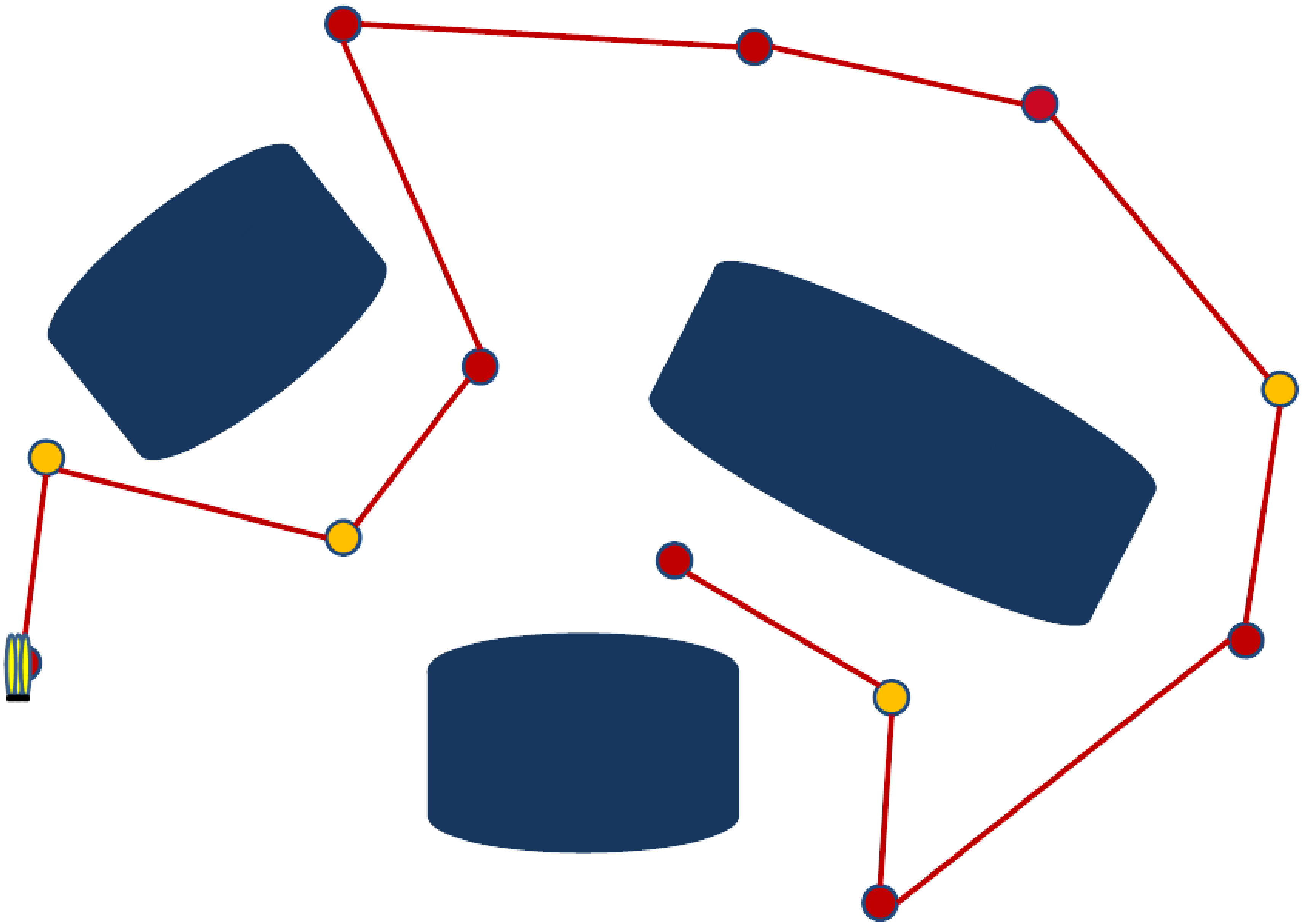
Dispatching more than a Single Plan

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

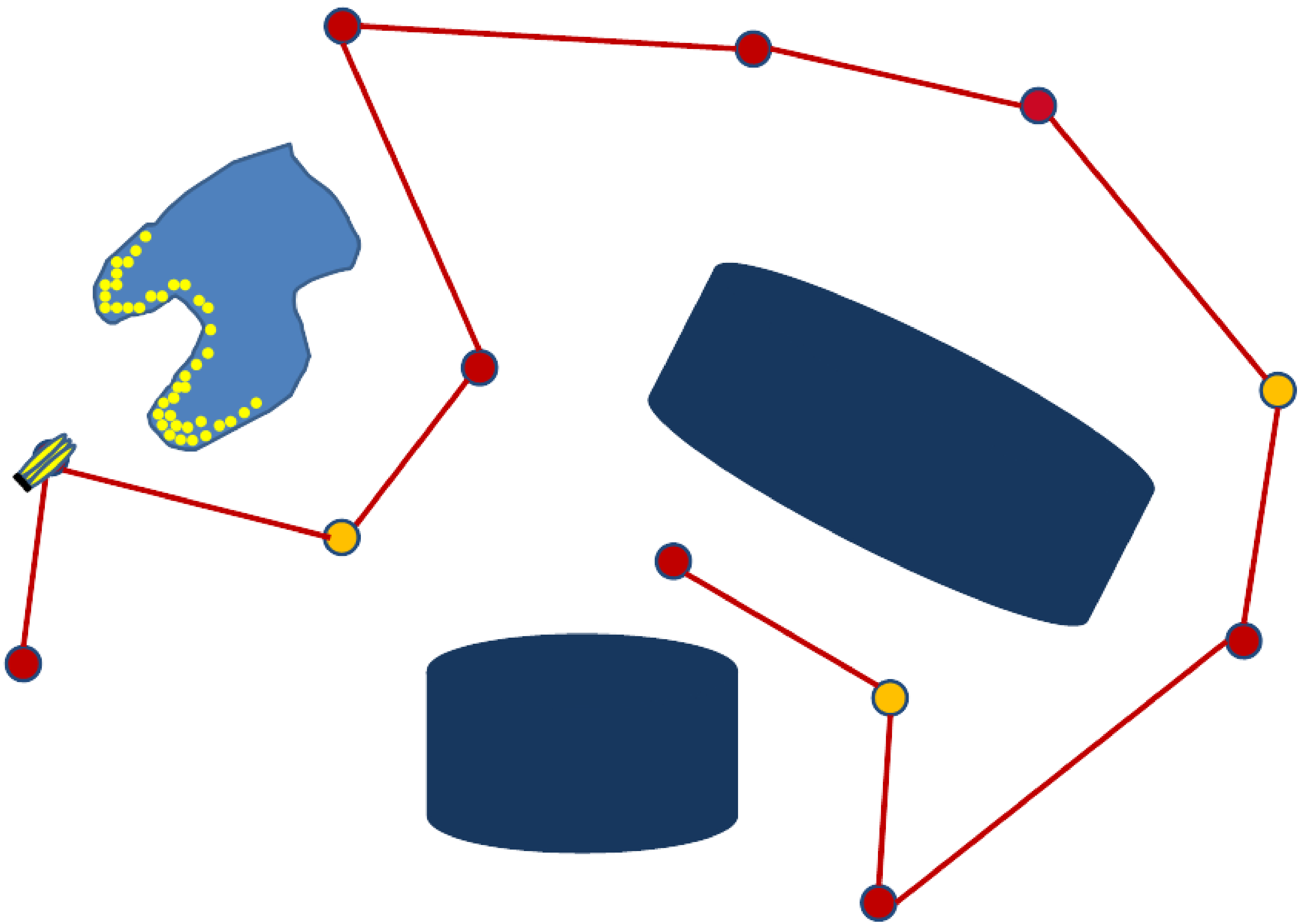
One very common and simple scenario with robots is planning a search scenario. For tracking targets, tidying household objects, or interacting with people.

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

Dispatching more than a Single Plan

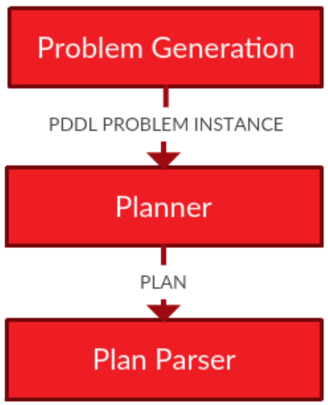


Dispatching more than a Single Plan



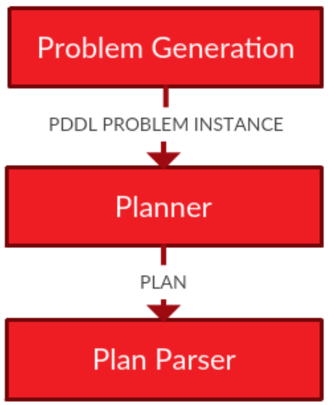
Hierarchical and Recursive Planning

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.

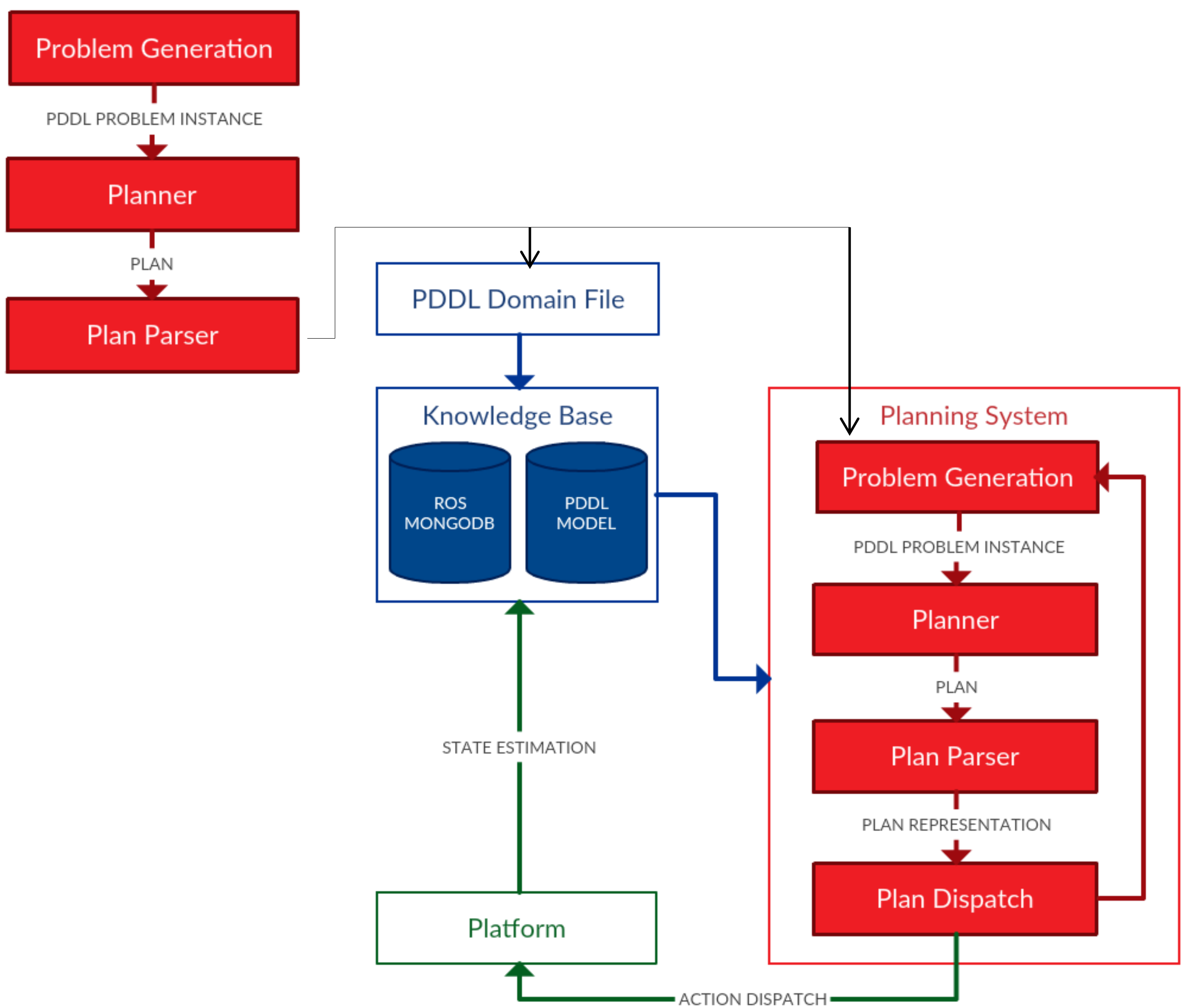



```
0.000: (correct_position auv0 wp_auv0) [3.000]
3.001: (do_hover_fast auv0 wp_auv0 strategic_location_7)
[11.403]
14.405: (correct_position auv0_strategic_location_78)
[3.000]
17.406: (observe_inspection_point auv0 strategic_location_7
inspection_point_2) [10.000]
27.407: (correct_position auv0 strategic_location_7)
[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]
...
```

Energy consumption = 10W
Duration = 86.43s

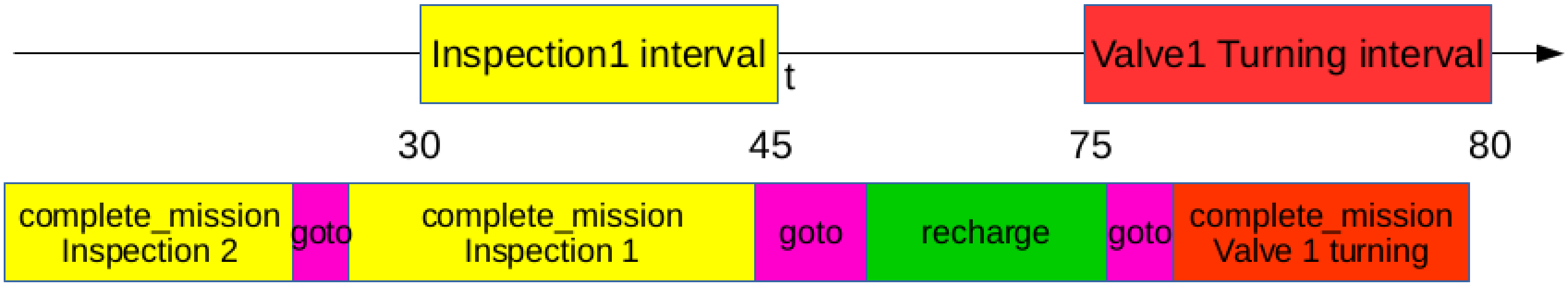
Hierarchical and Recursive Planning

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

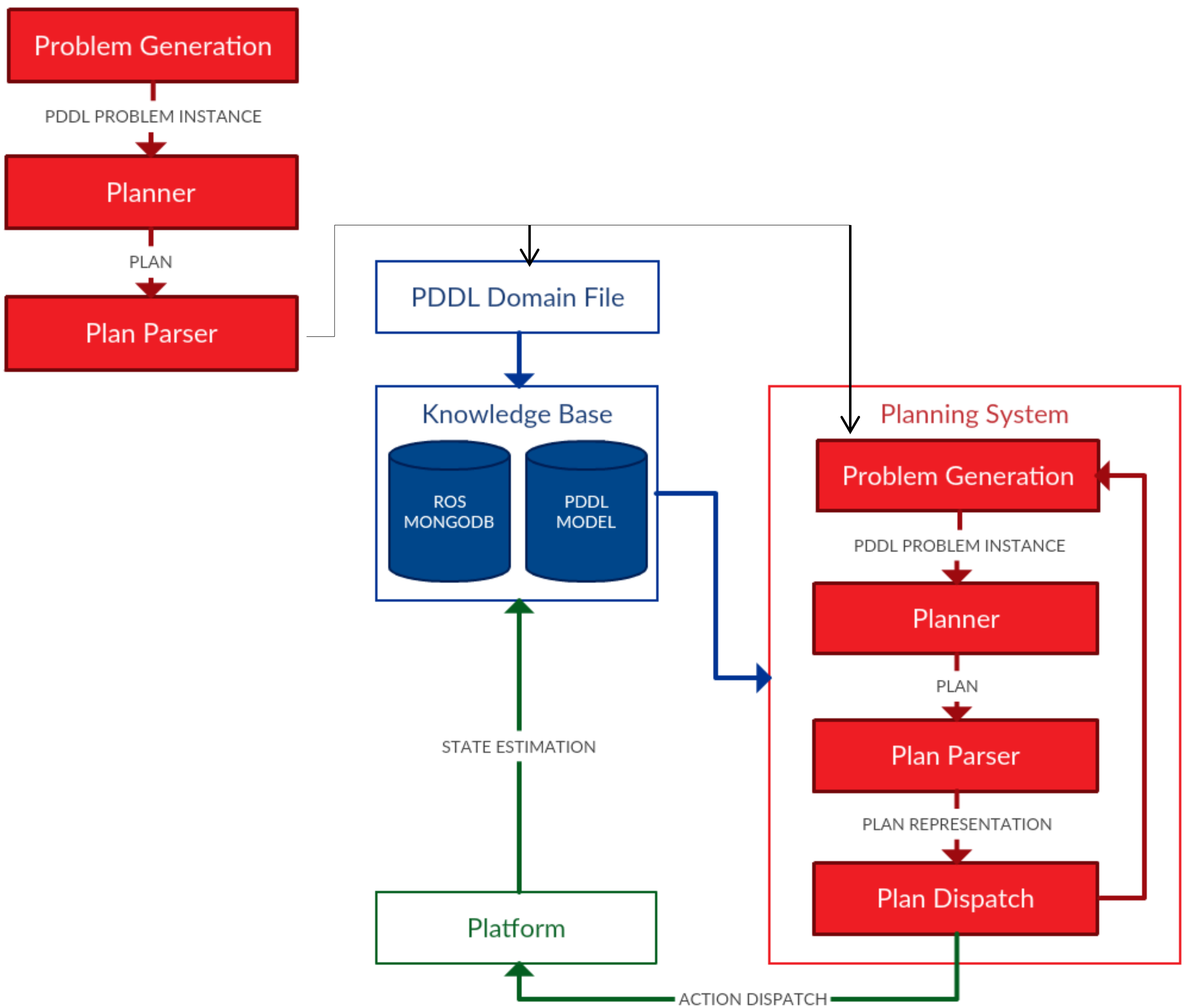
For each task we generate a *tactical plan*. The time and resource constraints are used in the generation of the strategic problem.



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

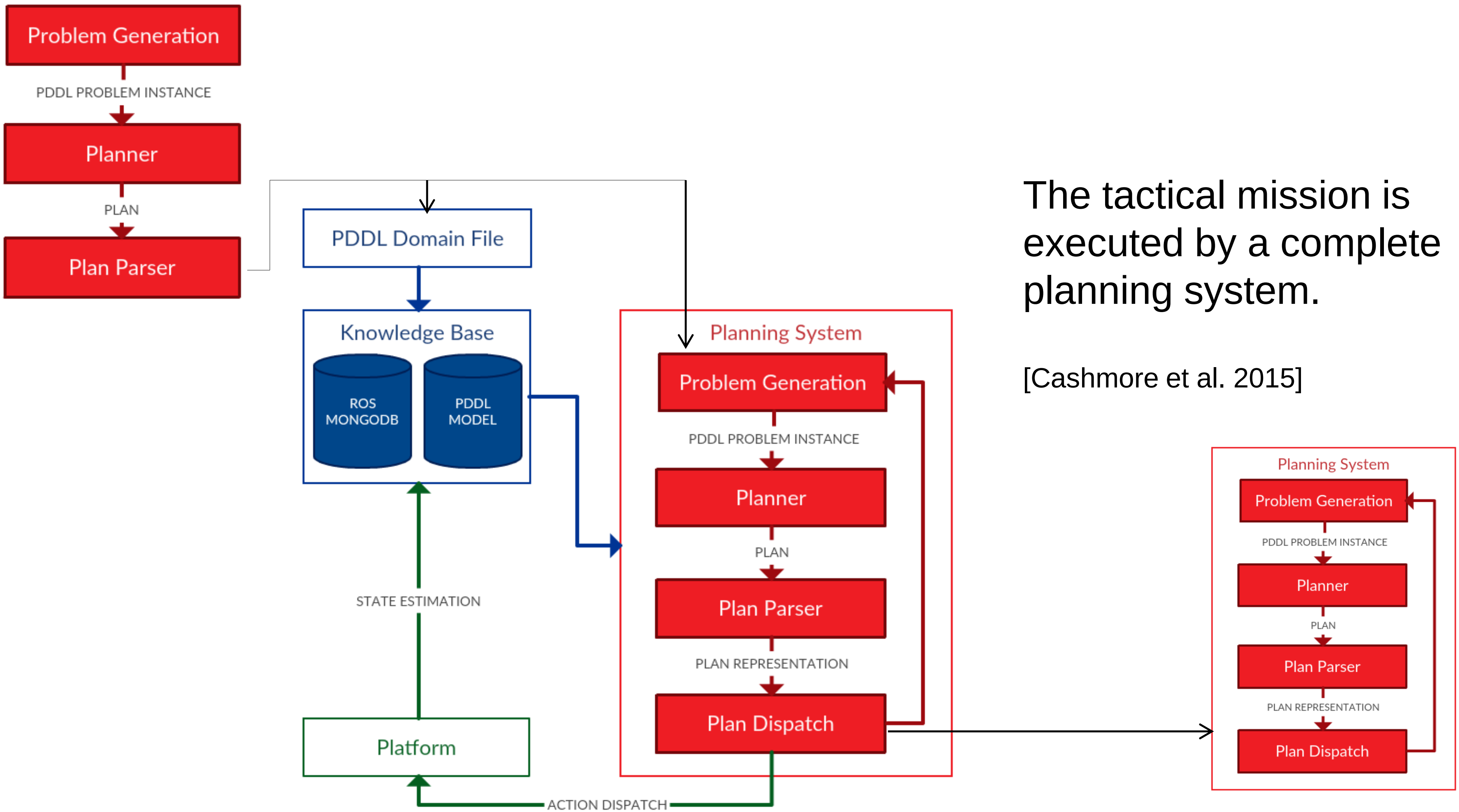
Hierarchical and Recursive Planning

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.

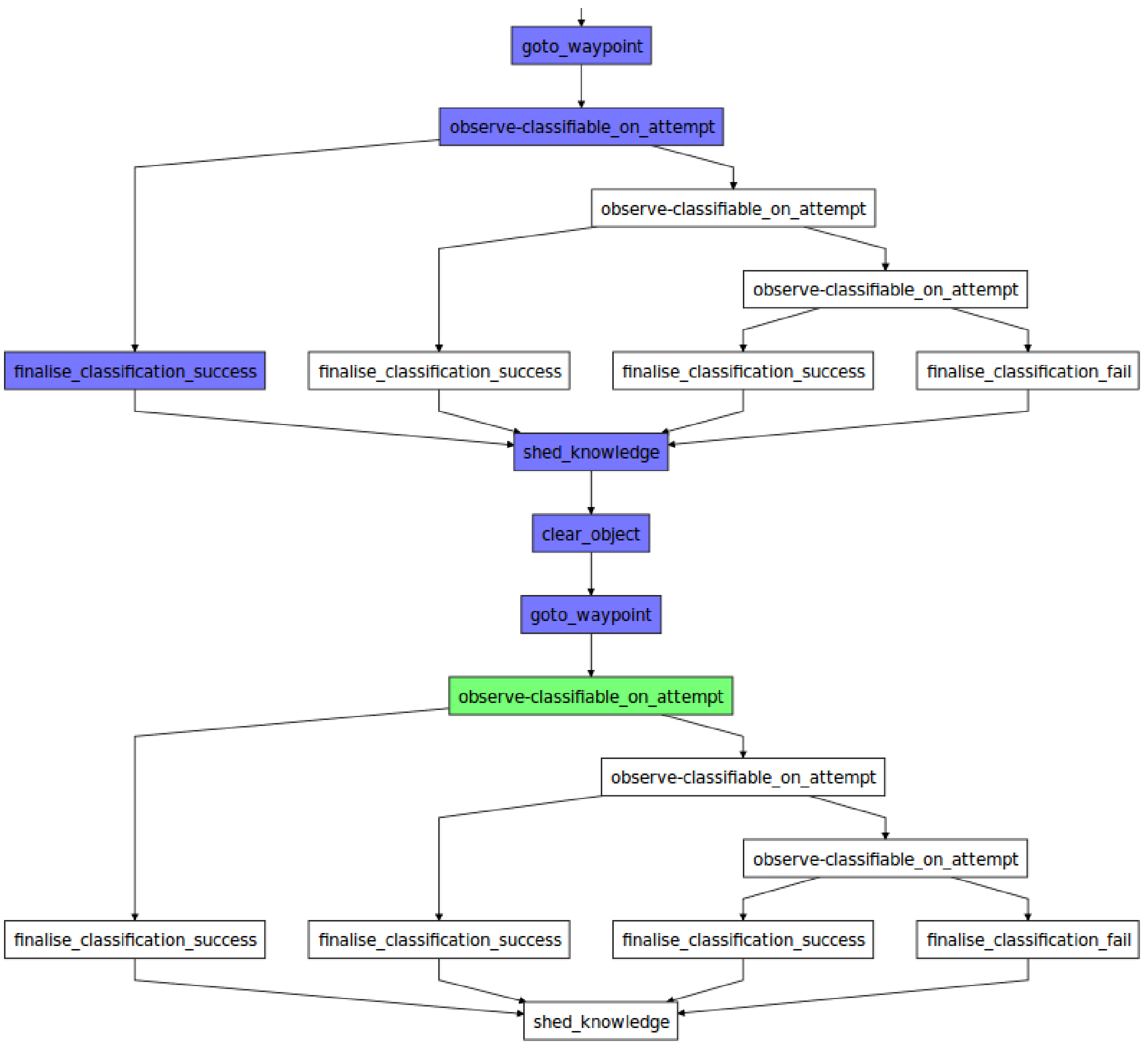
[Cashmore et al. 2015]

Hierarchical and Recursive Planning

Observing an object has two outcomes:

- Success. The object is classified or recognised

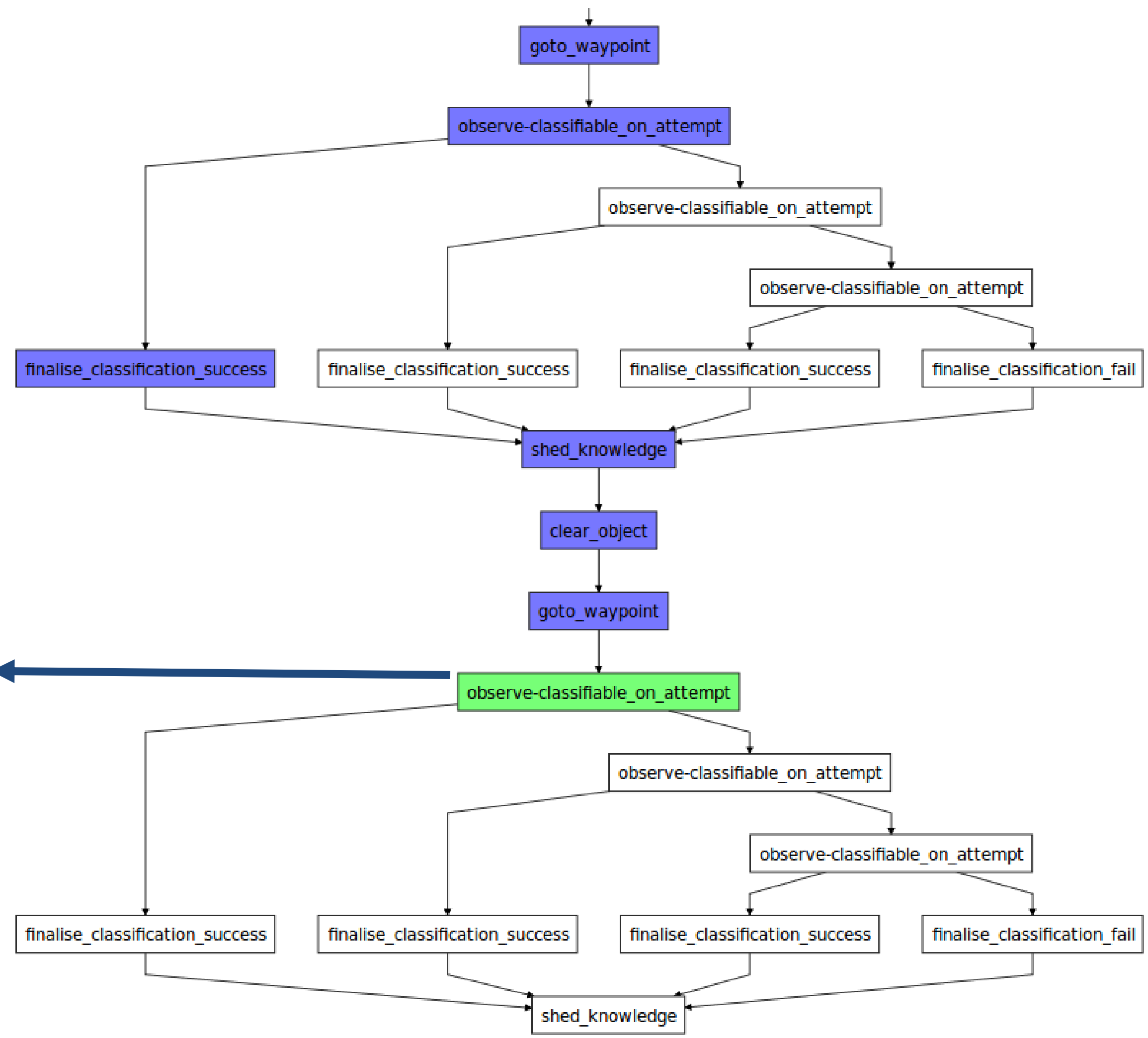
- Failure. The object type is still unknown, but new viewpoints are generated to discriminate between high-probability possibilities.



Hierarchical and Recursive Planning

The action corresponds to a short tactical plan to observe viewpoints.

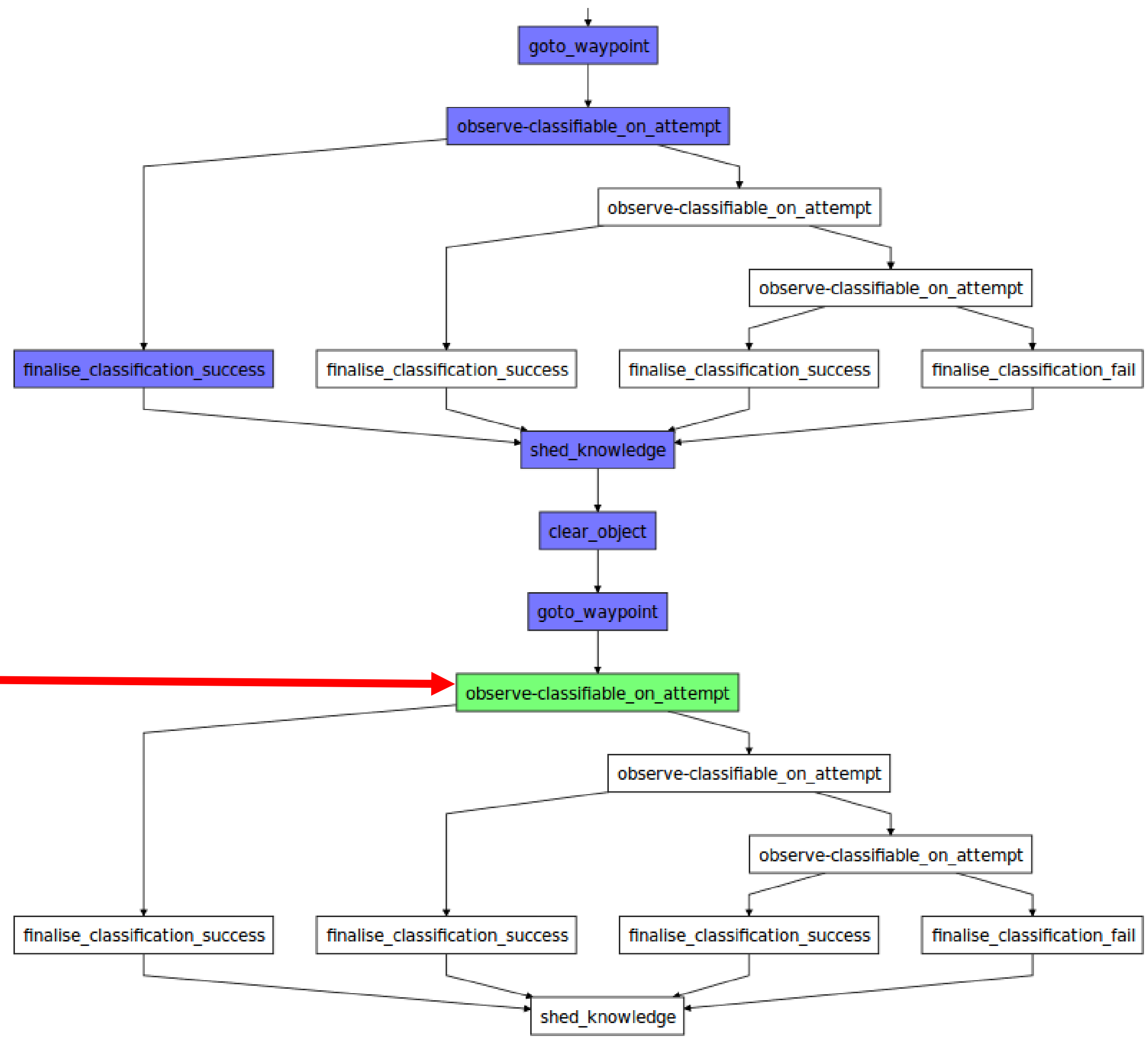
0.000: (goto_waypoint) [10.0]
0.000: (observe) [2.0]
0.000: (goto_waypoint) [10.0]
0.000: (pickup-object) [16.0]



Hierarchical and Recursive Planning

The action corresponds to a short tactical plan to observe viewpoints.

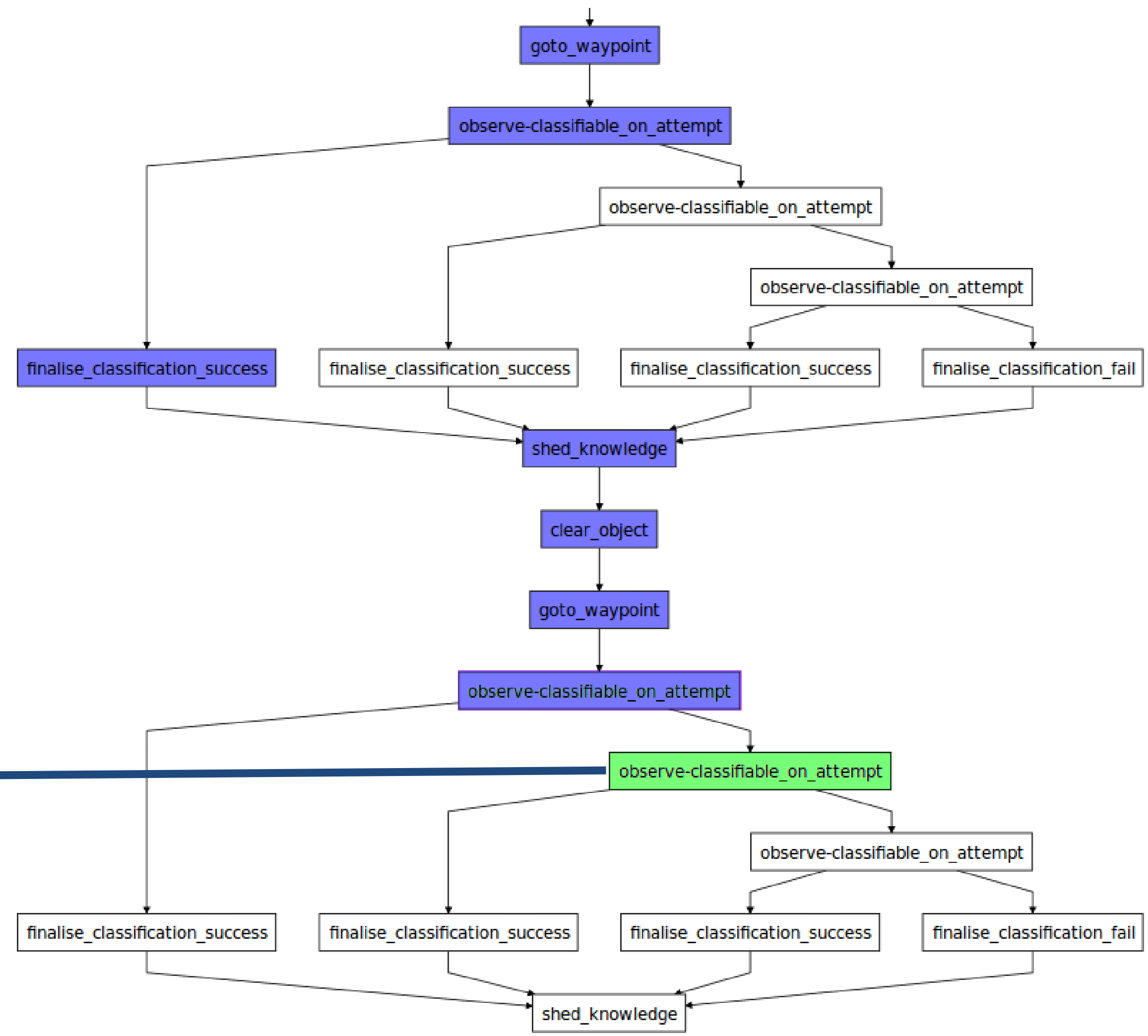
0.000: (goto_waypoint) [10.0]
0.000: (observe) [2.0]
0.000: (goto_waypoint) [10.0]
0.000: (pickup-object) [16.0]

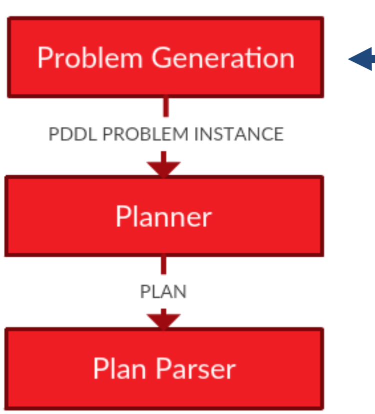


Hierarchical and Recursive Planning

The action corresponds to a short tactical plan to observe viewpoints.

The next tactical plan can only be generated once the new viewpoints are known.

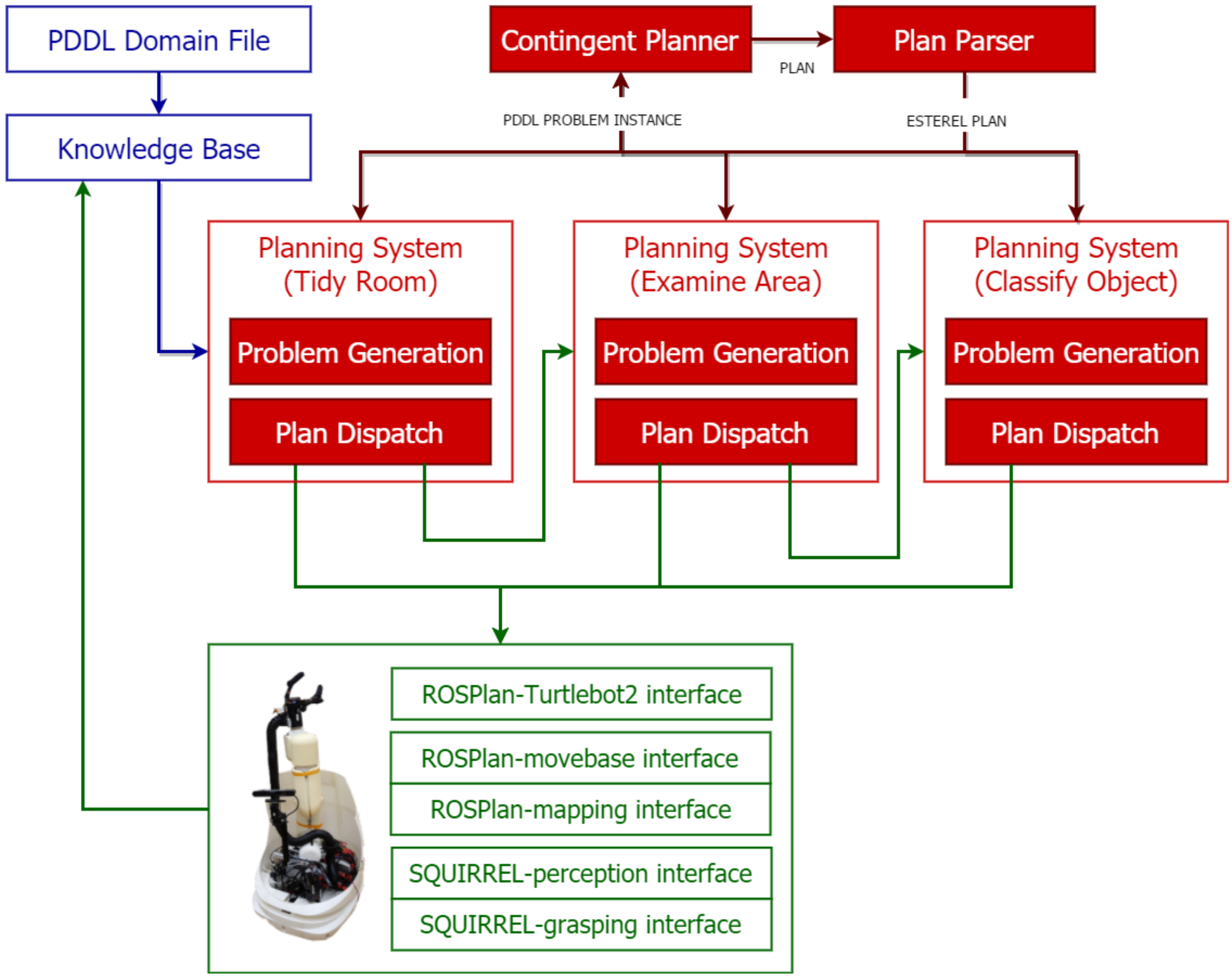




Hierarchical and Recursive Planning

The components of the system are the same as the very simple dispatch.

The behaviour of the robot is very different.



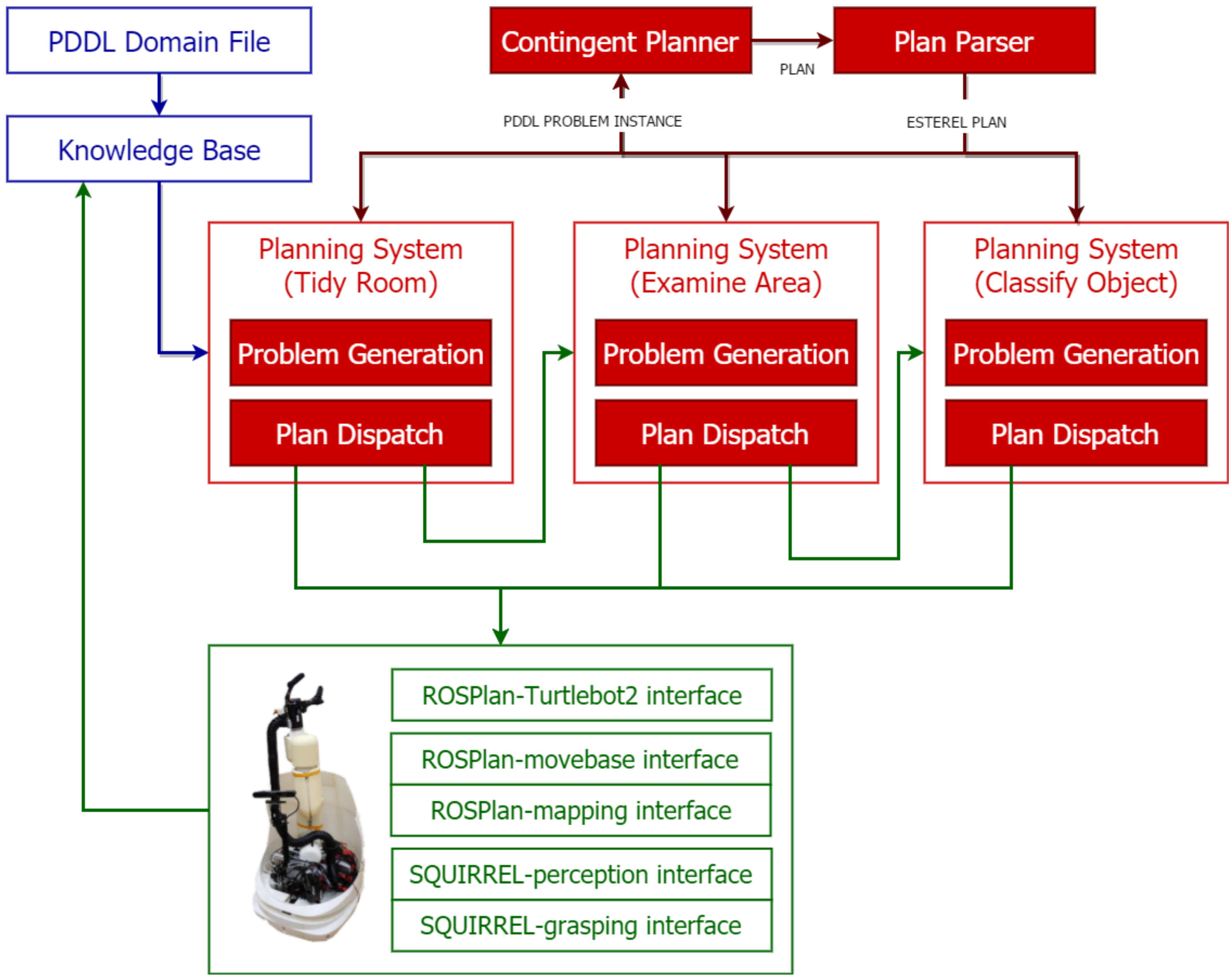
Hierarchical and Recursive Planning

The components of the system are the same as the very simple dispatch.

The behaviour of the robot is very different.

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

Both the components and how they are used.



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 more Plans: Opportunistic Planning

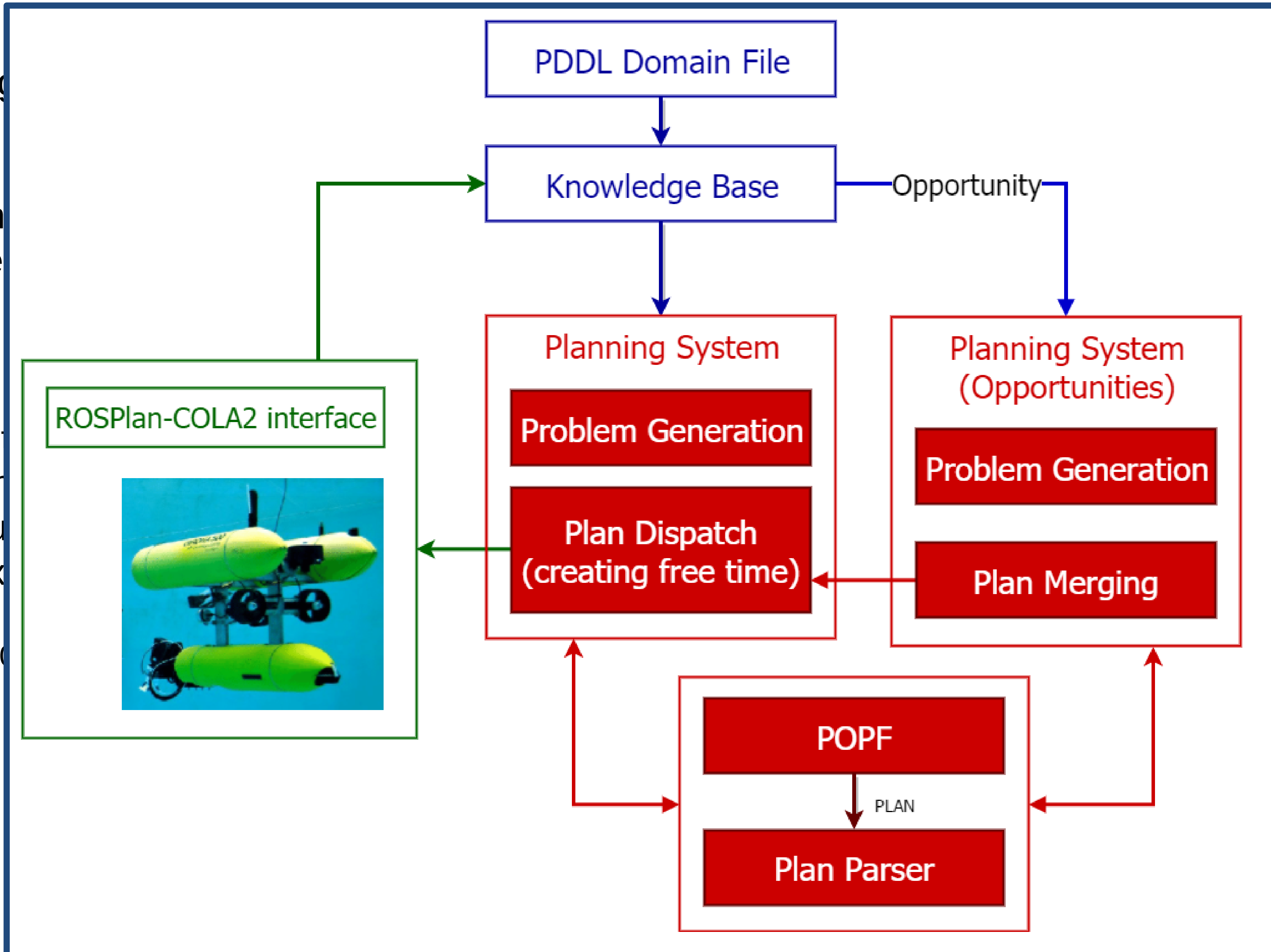
New plans are
goals and the
current plan.

If the new plan
then it is imme

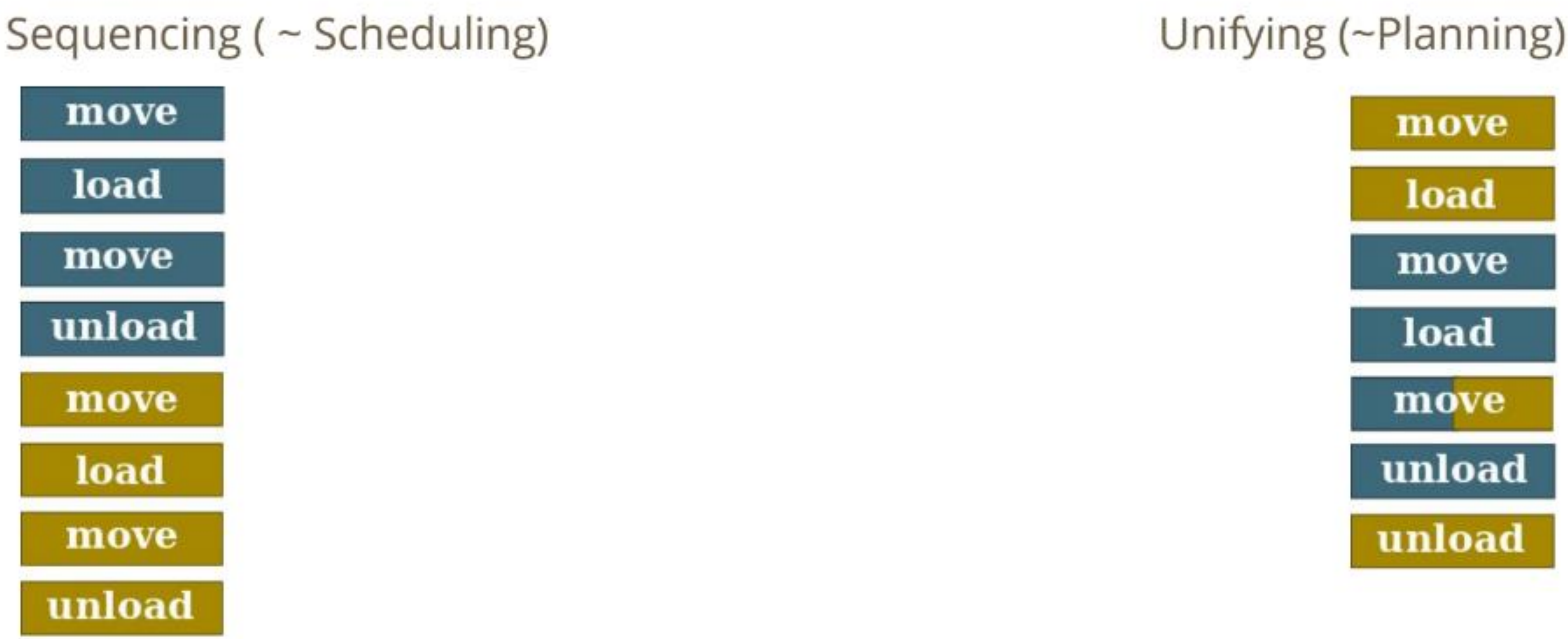
The approach

If an opportuni
of a plan fragm
plan can be pu
plan can be ex

[Cashmore et al. 2

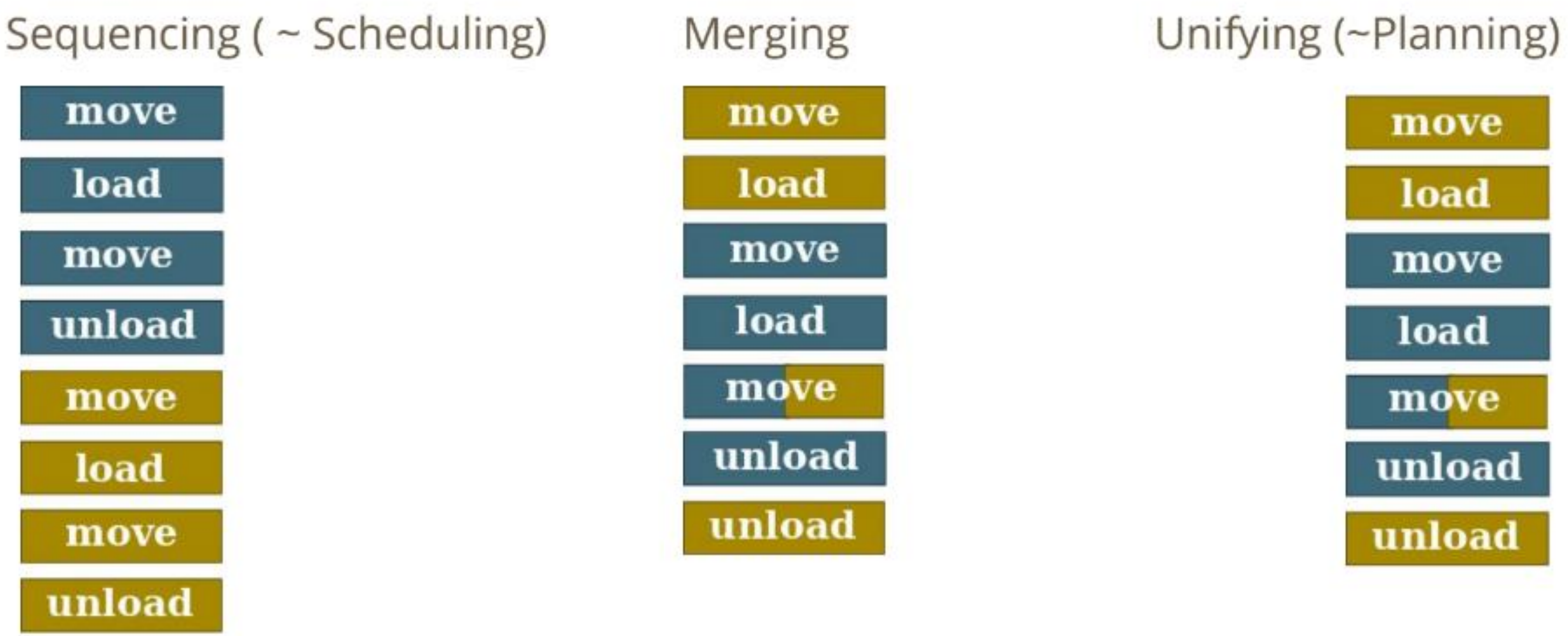


Dispatching Plans at the same time



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.

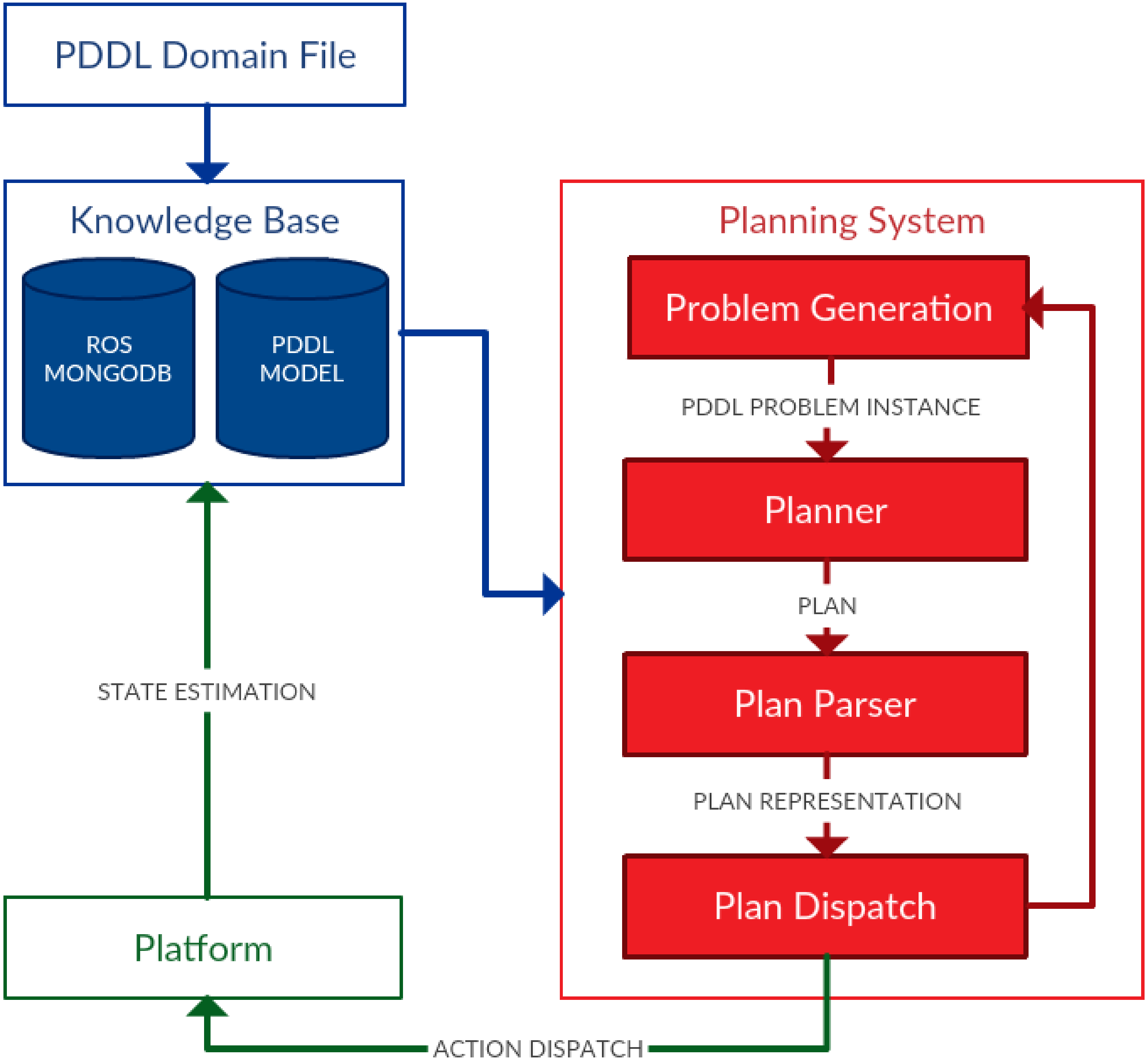
Plans can be merged in a more intelligent way. A single action can support the advancement towards multiple goals.

[Mudrova et al. 2016]

ROSPlan and PNP

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

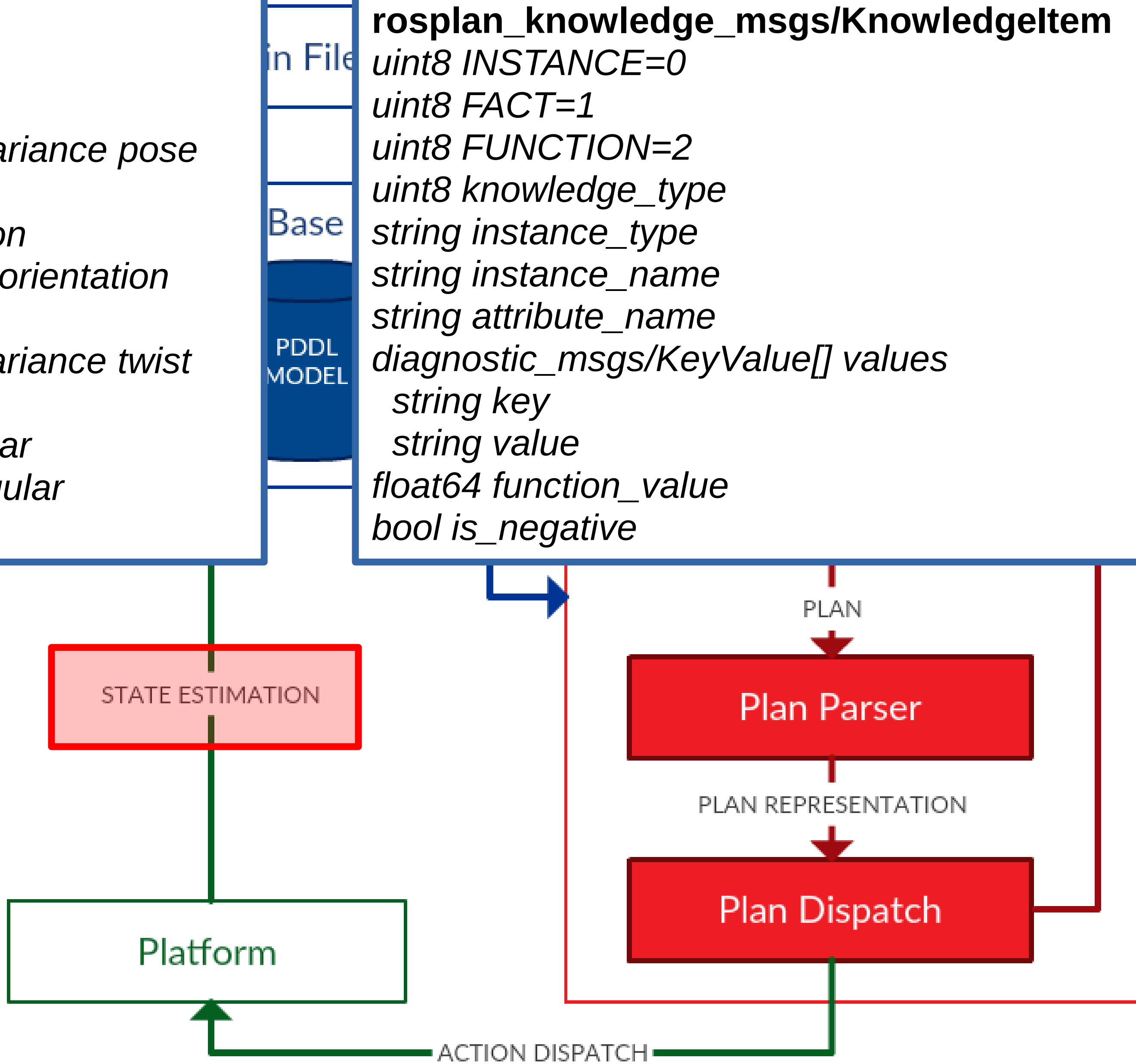
The plan is validated against a model that is continually changing and only partially sensed.



ROSPlan and PNP

```
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
```

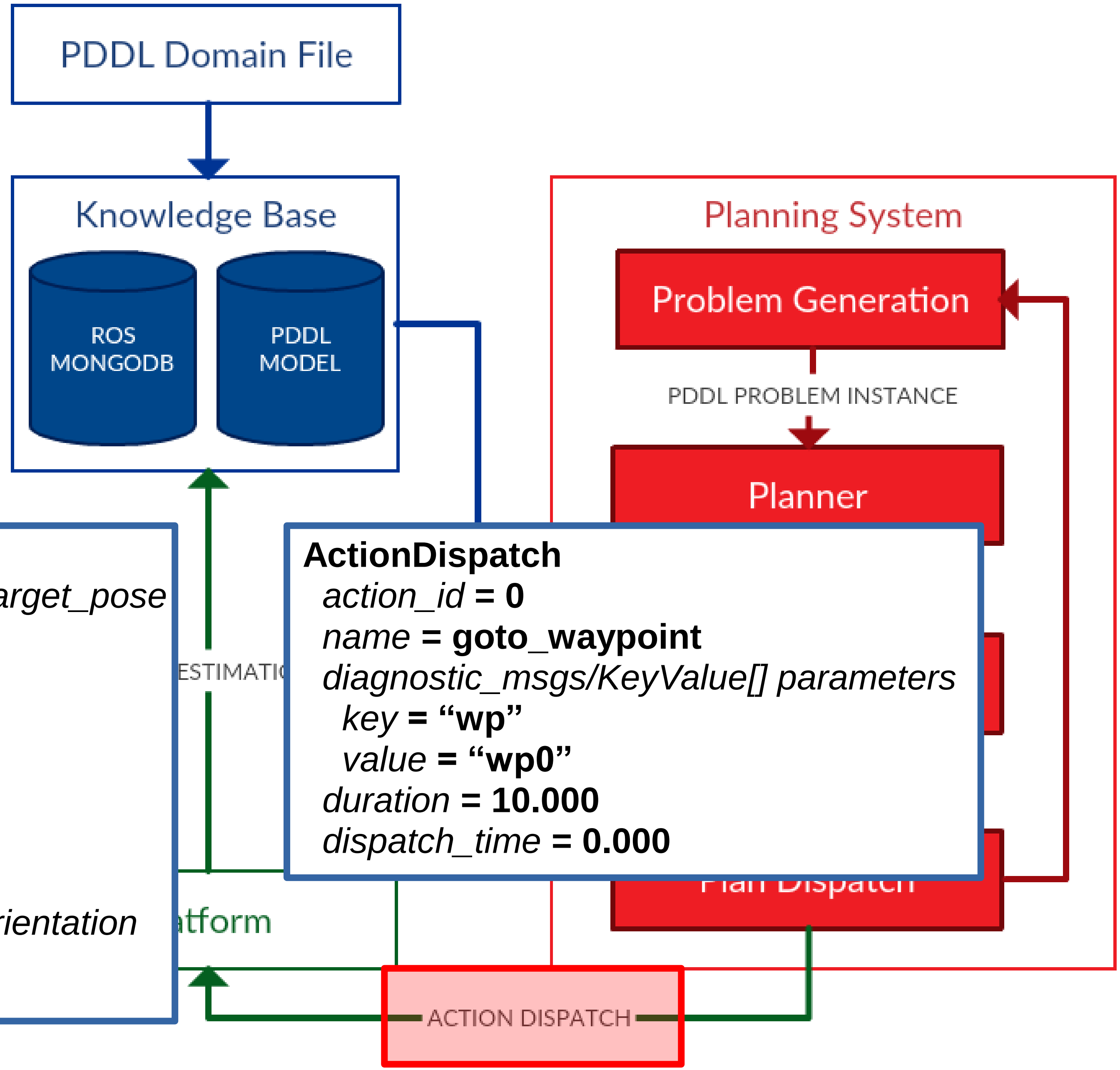
```
rosplan_knowledge_msgs/KnowledgeItem
uint8 INSTANCE=0
uint8 FACT=1
uint8 FUNCTION=2
uint8 knowledge_type
string instance_type
string instance_name
string attribute_name
diagnostic_msgs/KeyValue[] values
  string key
  string value
float64 function_value
bool is_negative
```



ROSPlan and PNP

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

The plan is validated against a model that is continually changing and only partially



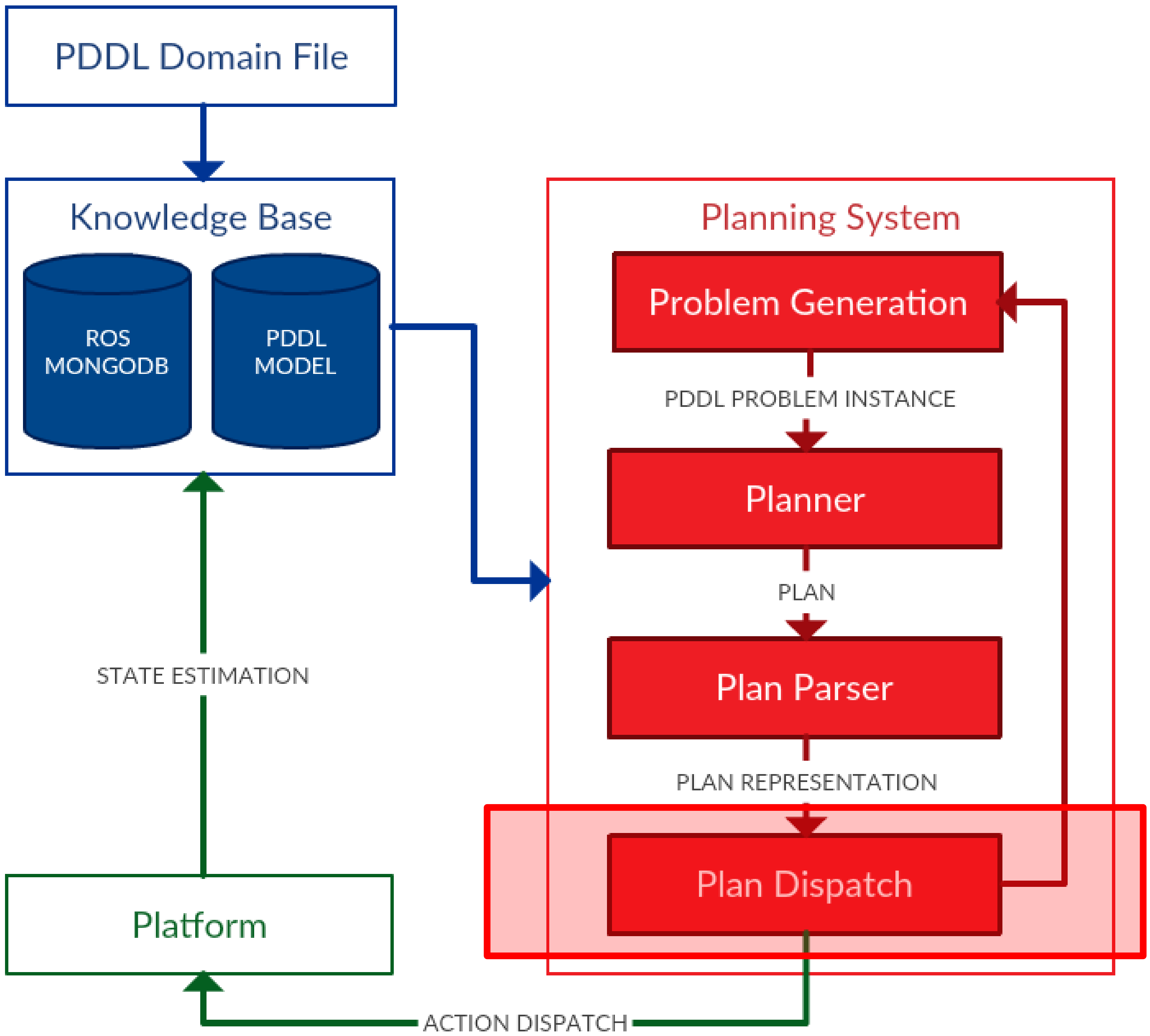

```
move_base/MoveBaseGoal
geometry_msgs/PoseStamped target_pose
  std_msgs/Header header
  ...
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      ...
```

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

ROSPlan and PNP

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

The plan is validated against a model that is continually changing and only partially sensed.



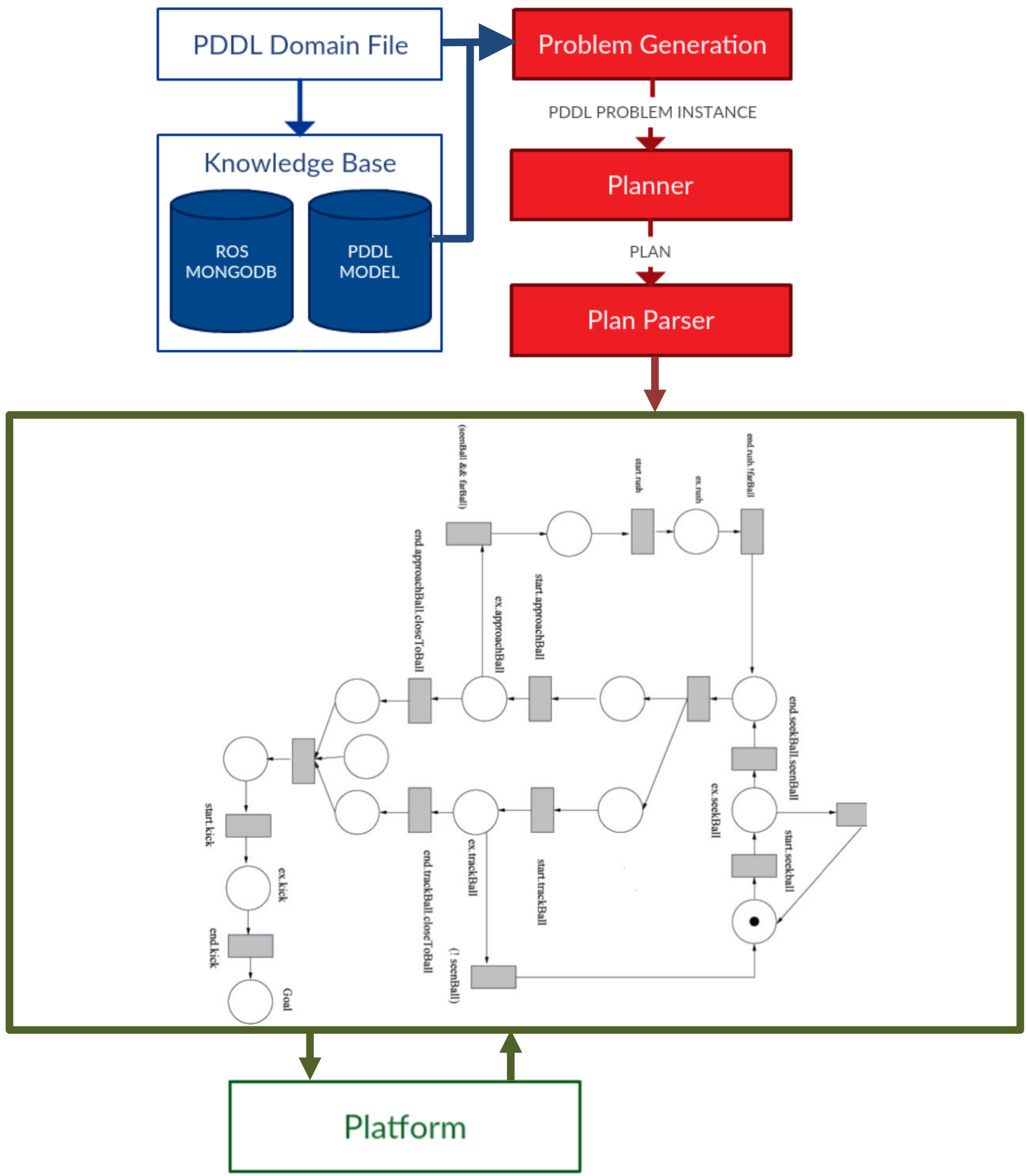
ROSPlan and PNP

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

The plan is validated against a model that is continually changing and only partially sensed.

The RosPNP Library encapsulates both action dispatch and state updates.

In a Petri Net plan the only state estimation performed is explicit in the plan.



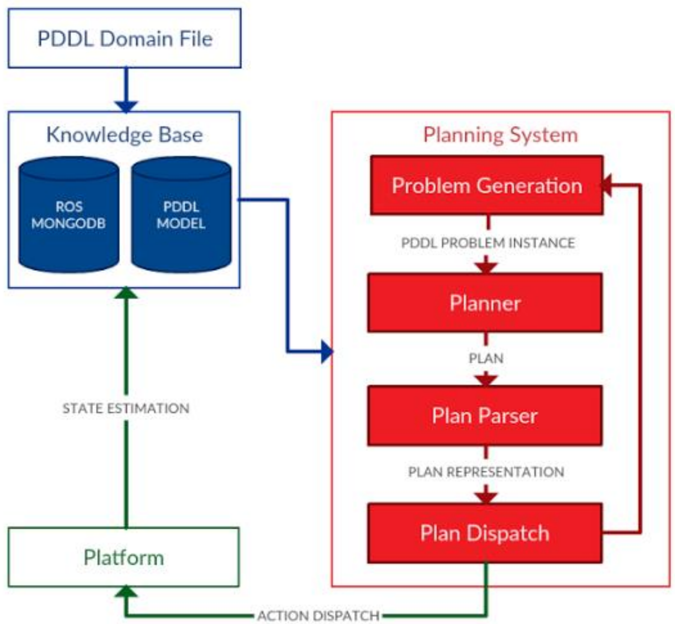
ROSPlan

ROSPlan | Documentation | Demos | Github Wiki | View on GitHub | Download .tar.gz | Contact

Documentation Home

What is ROSPlan?

The ROSPlan framework provides a generic method for task planning in a ROS system. ROSPlan encapsulates both planning and dispatch. It possesses a simple interface, and includes some basic interfaces to common ROS libraries.



What is it for?

ROSPlan has a modular design, intended to be modified. It serves as a framework to test new modules

Main
Documentation Home
ROSPlan Overview
List of Topics
List of Services

Planning System
Launching the Planning System
Using the Planning System
Generating a Problem Instance
Plan Representations
Plan Dispatch and Execution

Knowledge Base
Launching the Knowledge Base
Using the Knowledge Base
Fetching Domain Details
Fetching Problem Instance
Adding to the Knowledge Base

Working with ROSPlan
Replacing the planner
Replacing the problem generation
Replacing the plan dispatch
Adding an action
Adding state estimation

ROSPlan documentation and source:
kcl-planning.github.io/ROSPlan