

Session : Cognitive Architectures, Task Planning and PDDL

ACM SIGSOFT Summer School for Software Engineering in Robotics
Delft (Netherlands)

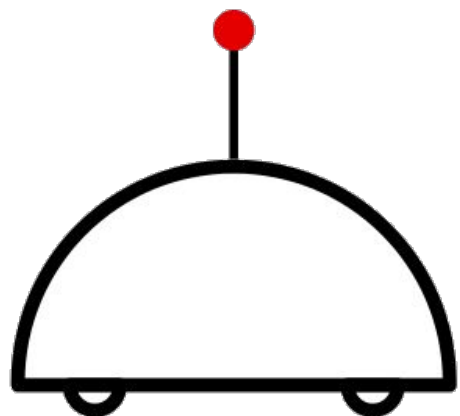
Francisco J. Rodriguez Lera



universidad
de león

Who I am

Researcher



GRUPO DE ROBÓTICA



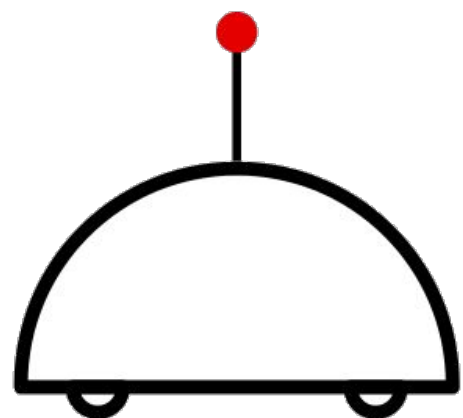
universidad
de león



<https://orcid.org/0000-0002-8400-7079>

Who I am

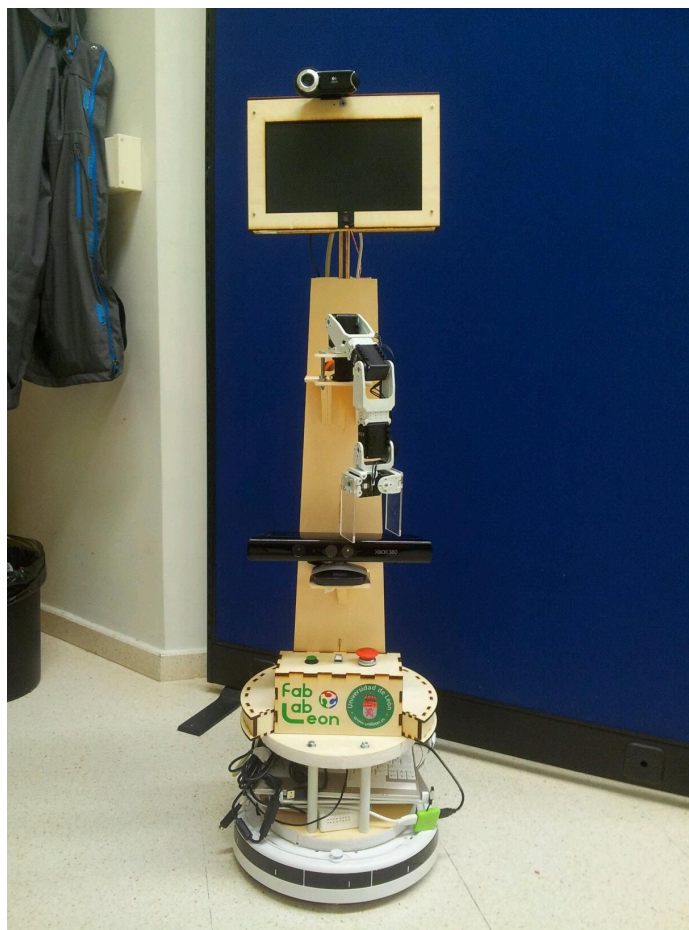
Researcher



GRUPO DE ROBÓTICA



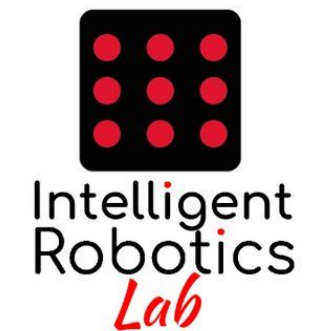
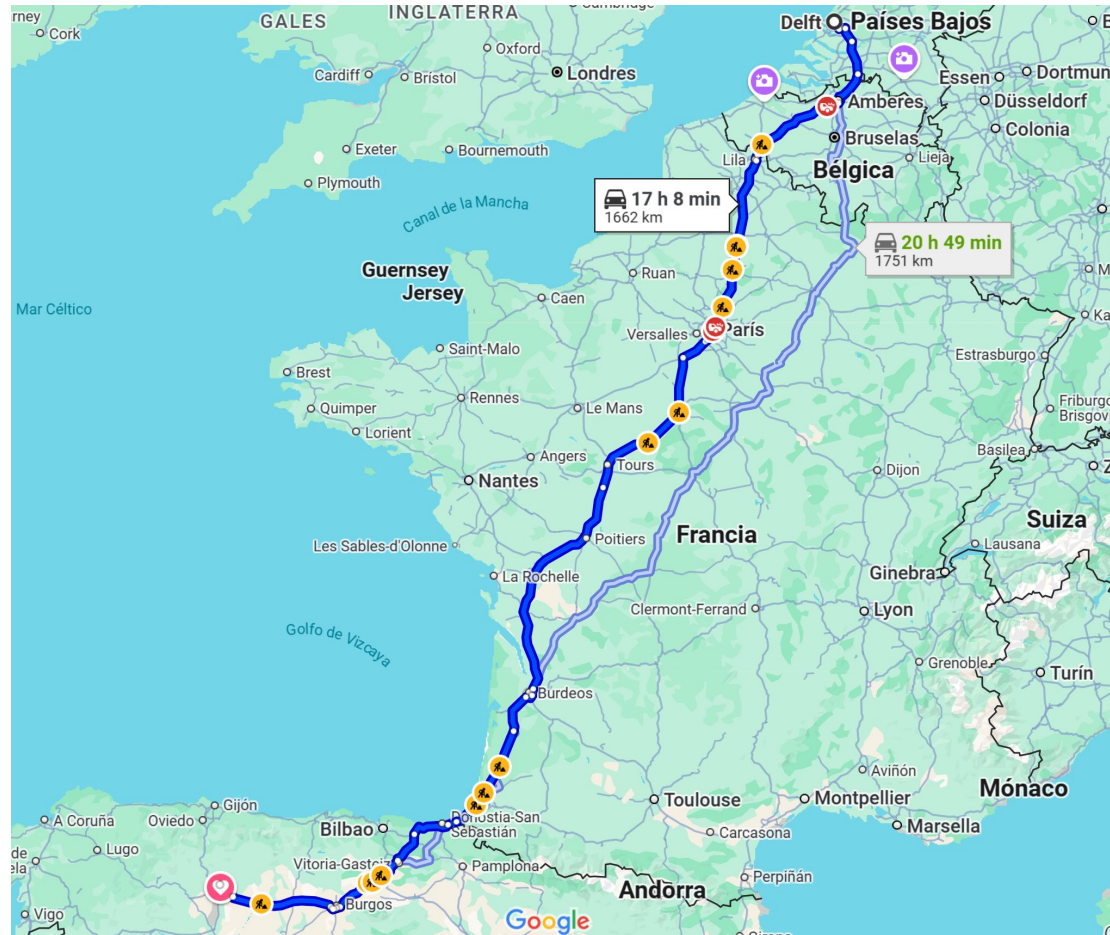
universidad
de león



<https://orcid.org/0000-0002-8400-7079>

Where I am

The team, collaborations, ...



Presentation Overview

Outline

1. Introduction
2. PDDL
3. PDDL Elements
4. Planners
5. Exercises
6. Takeaways
7. Acknowledgments

Introduction

Example

Someone wants to buy a mobile robot from the market and needs to choose which one to purchase.

.The decision maker knows how many robots are available but knows nothing about a specific one until they examine it.

.Upon examining specifications, the individual gains all the information needed to evaluate its utility.

.However, acquiring this information incurs a cost, such as time or mental effort.

Introduction

Example

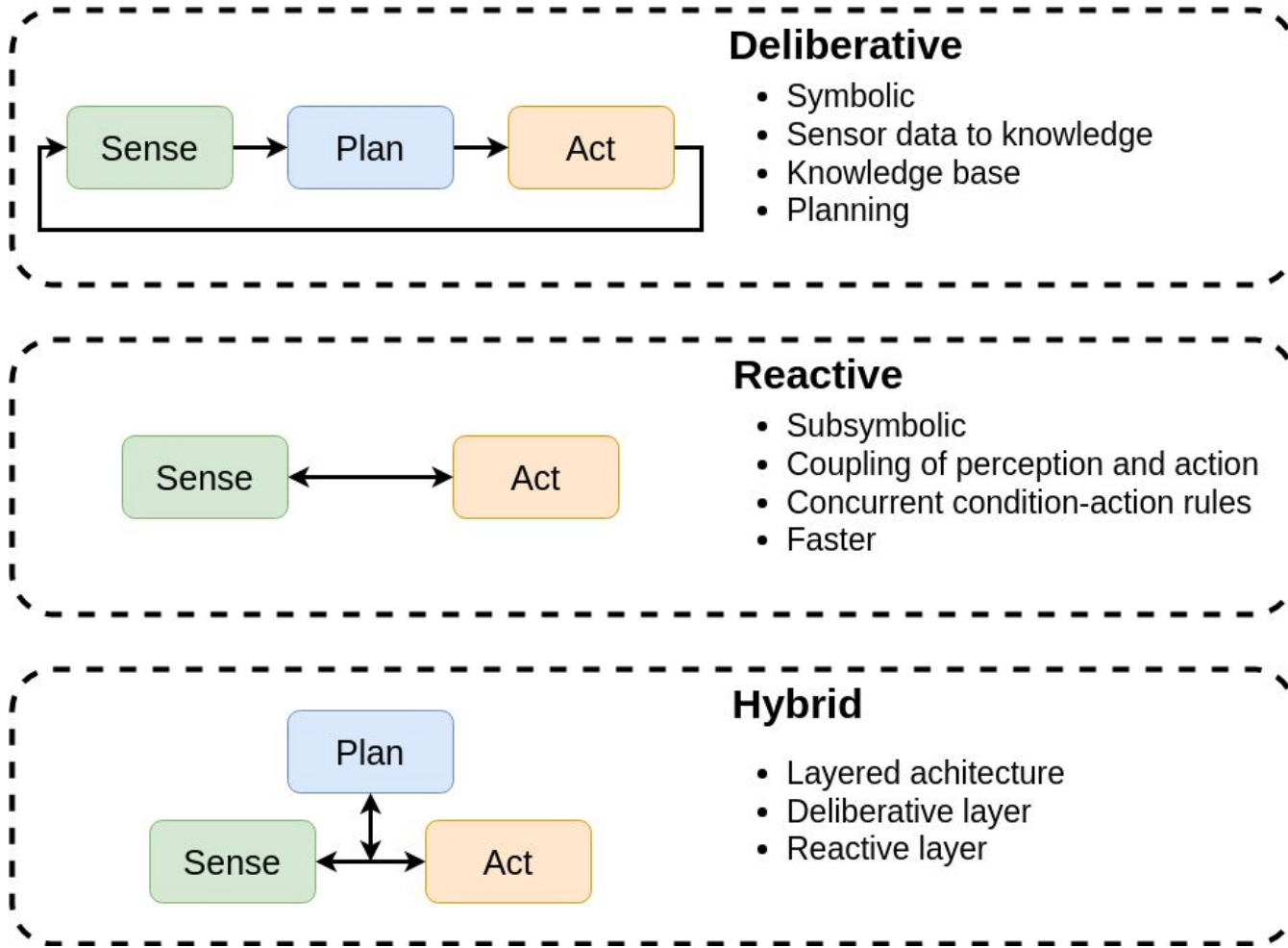
We can frame this situation as a simple optimal stopping problem.

The challenge **is to determine the optimal point at which the decision maker should stop searching and make a purchase.**

In each moment t_n , the decision maker is aware of the value of the best option they have seen so far, the number of remaining alternatives, and the cost of examining another option.

Introduction

Architectures



Deliberative → Fikes

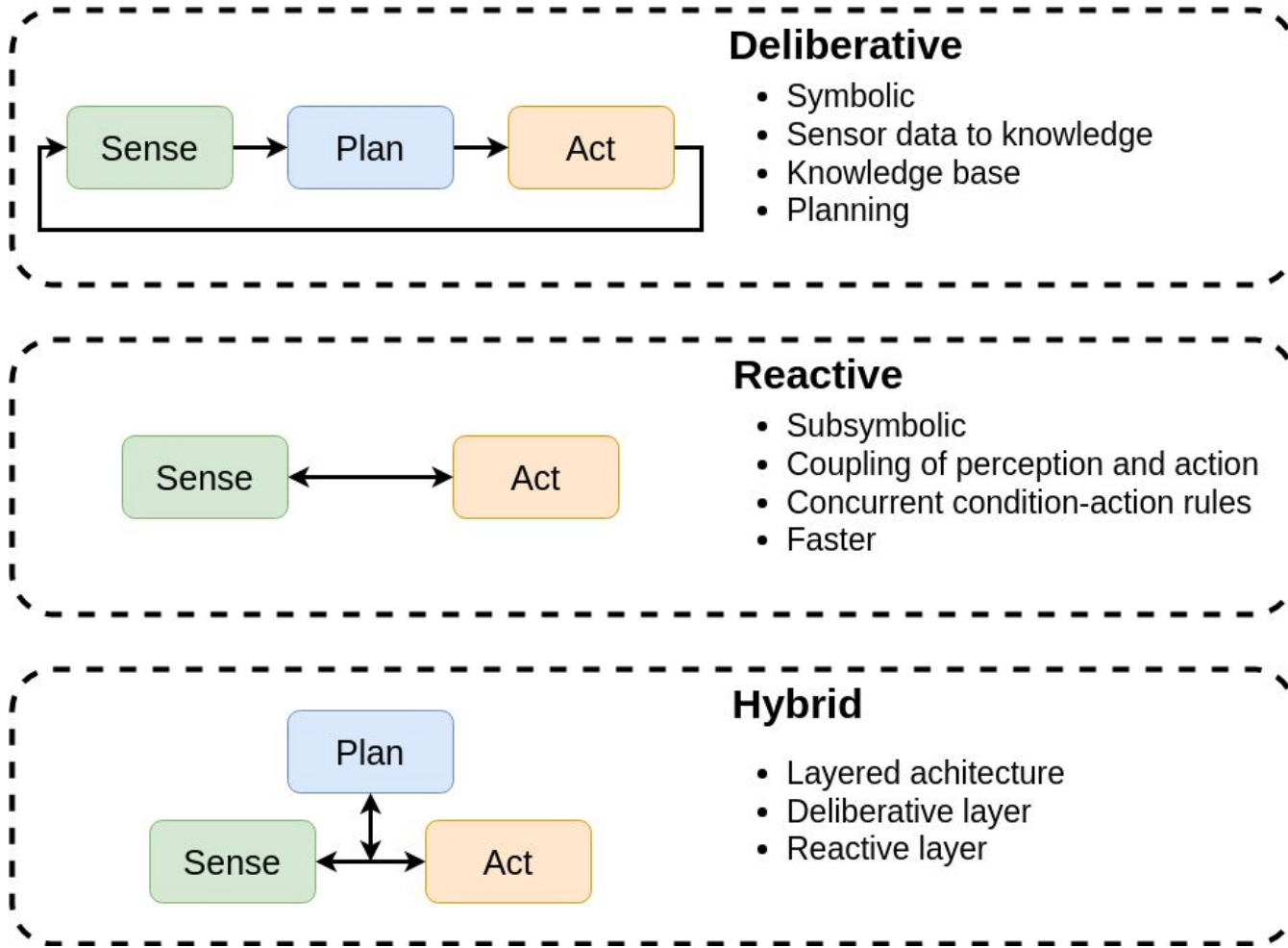
Reactive → Brooks

Hybrid → Arkin

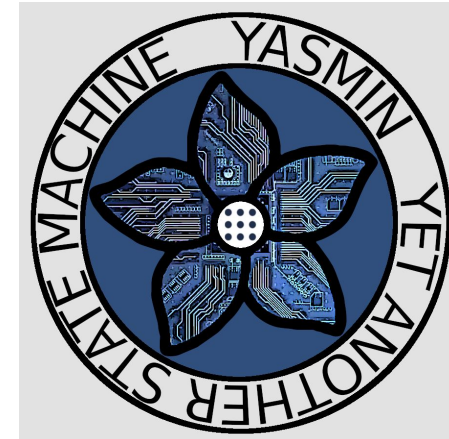
Three Layers → Gat

Introduction

Architectures



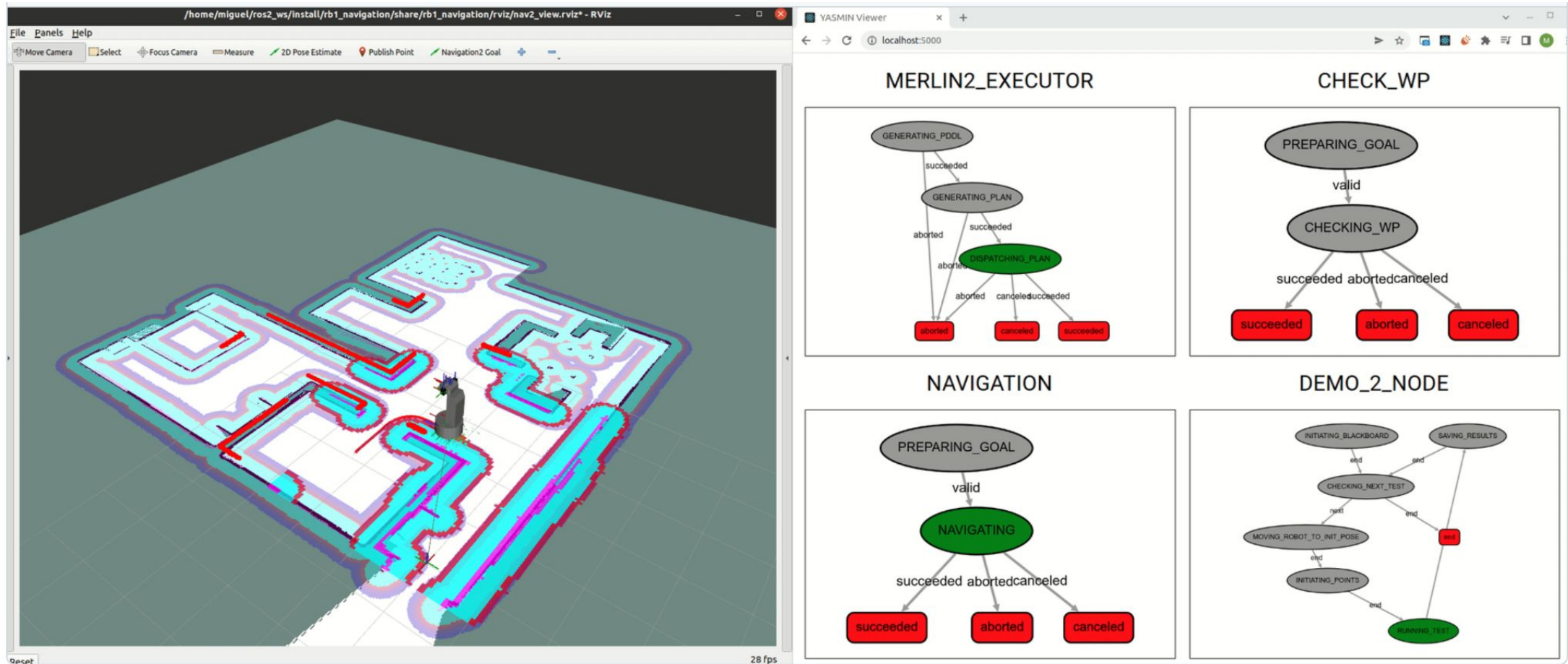
PlanSys
:::2



MERLIN 2
MachinEd Ros pLanINg

Introduction

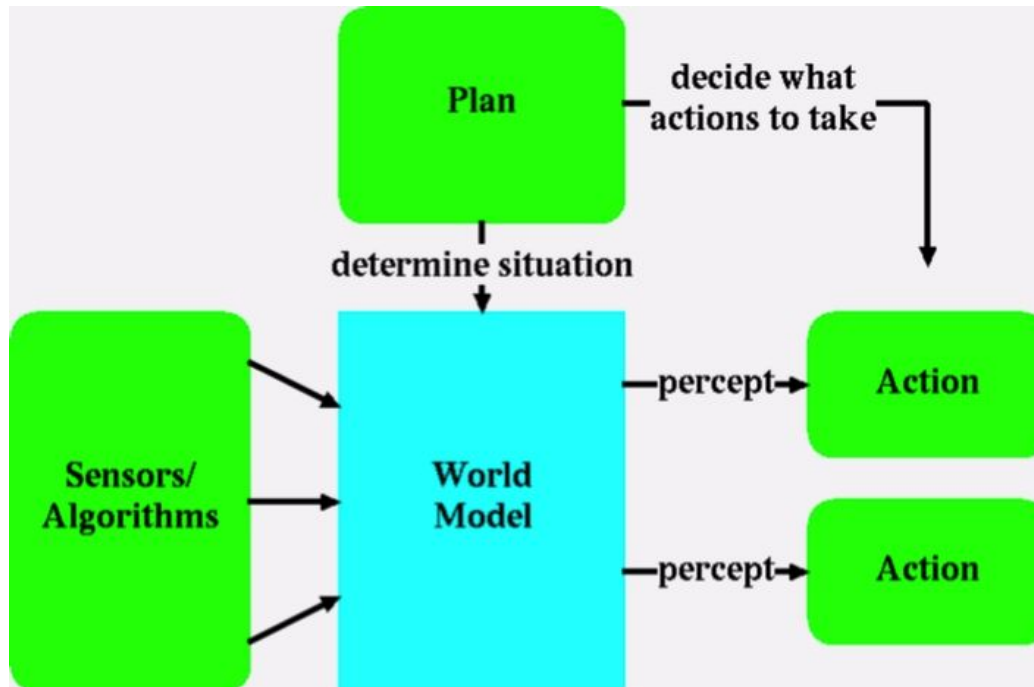
Architectures



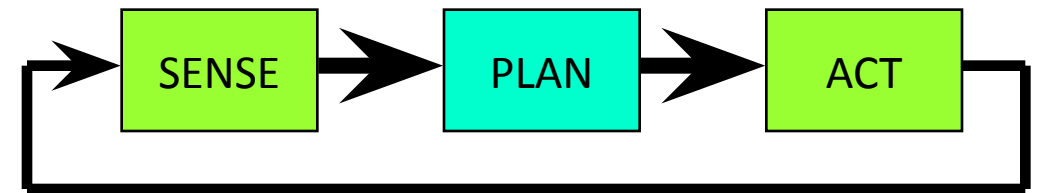
<https://github.com/uleroboticsgroup/yasmin>

Introduction

Hierarchical Architectures (aka: Deliberative, symbolic, classical...)



de Buy Wenniger, Gideon Maillette, and Attila Houtkooper. "GOAP." (2008).





The robot helps the operator to carry some luggage to a car which is parked outside.

Modelling the Problem

Carry My Luggage Example



- Main Goal
 - a. The robot helps the operator to carry a bag to a car parked outside.
- Optional Goals
 - a. Re-entering the arena
 - b. Following the queue on the way back to the arena
- Focus
 - a. Person following, navigation in unmapped environments, social navigation.
- Setup
 - a. Locations:
 - The test takes place both inside and outside the Arena.
 - The robot starts at a predefined location in the living room.
 - b. People: The operator is standing in front of the robot and is pointing at the bag to be carried outside.
 - c. Objects: At least two bags are placed near the operator (within a 2m distance and visible to the robot).

Modelling the Problem

Carry My Luggage Example

- Picking up the bag: The robot picks up the bag pointed at by the operator.
- Following the operator: The robot should inform the operator when it is ready to follow them. The operator walks naturally towards the car; after reaching the car, the operator takes the bag back and thanks the robot.
- Obstacles: The robot will face 4 obstacles along its way (in arbitrary order): (a) a small object on the ground, (b) a hard-to-see object, (c) a crowd of people obstructing the path outside, and (d) a small area blocked using retractable barriers.

Optional goals:

- **Re-entering the arena**: The robot returns to the arena, going back in through the entrance.
- **Following the queue**: After the robot has reached the car, a few of the people that formed the crowd obstructing the robot return to the arena in a queue. The robot can decide to join the queue on its way back to the arena, in a manner that appears natural to the people in the queue.

Modelling the Problem

Pednault's example

Using an example from Pednault (1988), let's consider a scenario with a single briefcase, B, that we want to use for transporting objects. Pednault models this straightforward domain with three operators:

- .MovB(l) for moving the briefcase along with its contents,
- .PutIn(x) for placing an item x into the briefcase, and
- .TakeOut(x) for removing an item from the briefcase.

Modelling the Problem

Pednault's example

TakeOut(x)

PRECOND : $x \neq B$

EFFECTS : $\neg \text{In}(x)$

PutIn(x,l)

PRECOND : $x \neq B \wedge \text{At}(B,l) \wedge \text{At}(x,l)$

EFFECTS : $\text{In}(x)$

MovB(m,l)

PRECOND : $\text{At}(B,m) \wedge m \neq l$

EFFECTS : $\text{At}(B,l)$

$\text{At}(z,l) \forall z \mid \text{In}(z) \wedge z \neq B$

$\neg \text{At}(B,m)$

$\neg \text{At}(z,m) \forall z \mid \text{In}(z) \wedge z \neq B$

Introduction

PDDL

PDDL ("Planning Domain Definition Language") is an attempt to standardize planning domain and problem description languages.

PDDL originated from the 1998 International Planning Competition (IPC) committee.

It aimed to promote empirical comparison between planning systems and benchmark diffusion.

PDDL has enhanced planning system evaluation and led to performance and expressivity improvements.

It has become a standard language for planning domain description in IPC, with its collection of domains serving as standard benchmarks.

It has facilitated the spread of planning techniques in various research and application domains, where modeling and solving decision problems are challenging

Introduction

AI Planning System vs Planner

- PDDL takes the problem's formalization, or model, as its input and employs various problem-solving techniques such as heuristic search or propositional satisfiability to derive a solution.
- Tasks such as transforming the model into a searchable space or logical reasoning problem and devising efficient heuristics to tackle it, are challenges for the planner's designer.
- The planner itself doesn't require knowledge of the specific problem description; it can operate on any problem expressed in its modeling language.
- Not every planner can solve every problem it's given. This characteristic of planners is termed domain-independence.

Introduction

PDDL

Components of a PDDL planning task:

- **Objects:** Things in the world that interest us.
- **Predicates:** Properties of objects that we are interested in; can be true or false.
- **Initial state:** The state of the world that we start in.
- **Goal specification:** Things that we want to be true.
- **Actions/Operators:** Ways of changing the state of the world.

Introduction

PDDL Versions

PDDL 1.2 (1998-2000):

Official language of the 1st and 2nd IPC(International Planning Competition).

PDDL 2.1 (2002):

Introduced functions, durative actions, and plan metrics.

Extensions of PDDL 2.1:

PDDL+ (continuous changes and predictable exogenous events).

Mapl (Multi-Agent Planning Language).

Opt (Ontology with Polymorphic Types).

NDDL (2003):

Proposed by NASA, based on activities and constraints rather than states and actions.

PDDL 2.2 (2004):

Introduced axioms and timed predicates.

Ppddl (Probabilistic PDDL) for probabilistic effects.

PDDL (2006):

Introduced state-trajectory constraints and preferences.

Appl (Abstract Plan Preparation Language) proposed.

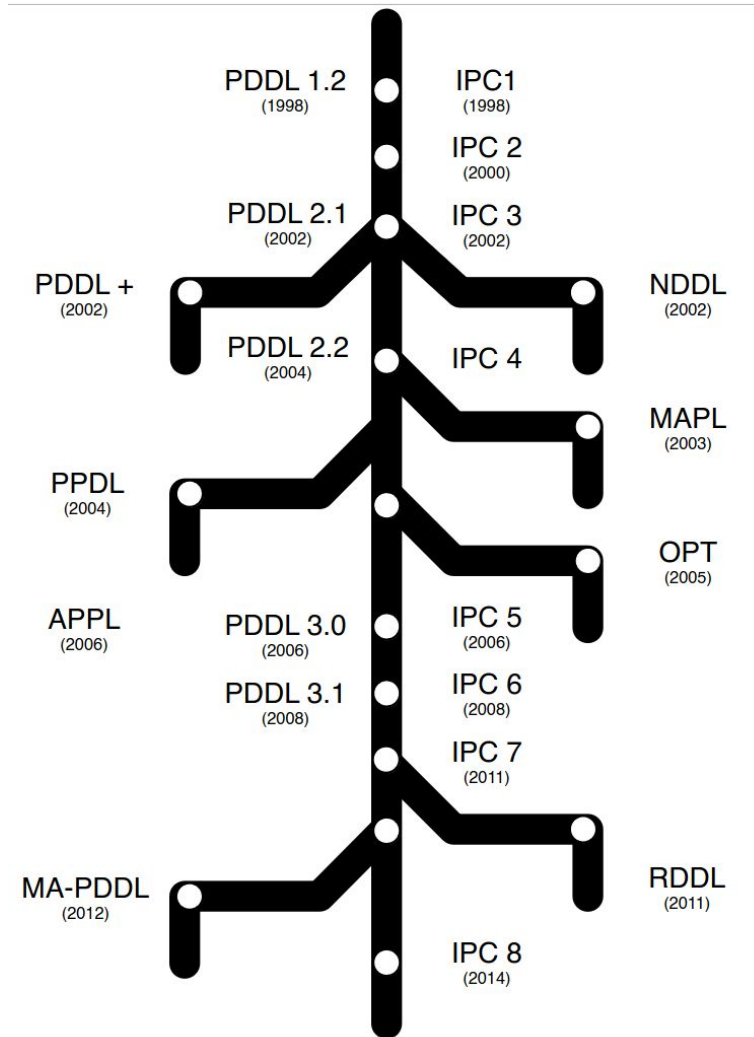
PDDL 3.1 (2008):

Introduces object-fluents.

Extensions: RDDDL (Relational Dynamic Influence Diagram Language) and MA-PDDL (Multi-Agent PDDL).

Introduction

History



Reference Pellier, Damien & Fiorino, Humbert. (2017). PDDL4J: a planning domain description library for java. Journal of Experimental & Theoretical Artificial Intelligence. 30. 1-34. 10.1080/0952813X.2017.1409278.

PDDL Notation

Origin

- Notation BNF/EBNF
 - It stands for Backus-Naur Form. It is a formal, mathematical way to specify context-free grammars.
 - It is precise and unambiguous
 - EBNF (Extended BNF) is widely used as the de facto standard to define programming languages
- Requirements
 - Each rule is of the form `<syntactic element> ::= expansion`.
 - Angle brackets (`<>`) delimit names of syntactic elements.
 - Square brackets (`[]`) surround optional material.
 - An asterisk (`*`) means “zero or more of”; a plus (`+`) means “one or more of.”
 - Some syntactic elements are parameterized. E.g., `<list (symbol)>` might denote a list of symbols, where there is an EBNF definition for `<list x>` and a definition for `<symbol>`. The former might look like `<list x> ::= (x*)` so that a list of symbols is just `<symbol>*`.
 - Ordinary parentheses are an essential part of the syntax we are defining and have no semantics in the EBNF meta language.

References:

<https://www.inf.ed.ac.uk/teaching/courses/propm/papers/ddl.html>

<https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf>

<http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/kovacs-pddl-3.1-2011.pdf>

PDDL Notation

Origin

Optional elements are enclosed in square brackets (`[]`)

Names, such as domain, predicate, action, etc., are typically comprised of alphanumeric characters, hyphens (`-`), and underscores (`_`), although some planners may impose restrictions.

Parameters of predicates and actions are identified by commencing with a question mark (`?`).

In predicate declarations (the `:predicates` section), parameters serve solely to specify the number of arguments for the predicate; hence, the specific parameter names are inconsequential as long as they are unique.

Predicates can encompass zero parameters; however, in such cases, the predicate name must still be encapsulated within parentheses.

PDDL Notation

Issues

Most planners do not fully support all elements of any version of PDDL.

Additionally, many planners have unique "features." For instance, they might misinterpret certain PDDL constructs or require slight syntax variations that deviate from the official language specification.

.Some planners implicitly require all arguments to an action to be distinct.

.Some planners mandate that action preconditions and/or effects be written as conjunctions (i.e., as **(and ...)**) even if the precondition/effect contains only one atomic condition or no condition at all.

.Most planners ignore the **:requirements** section of the domain definition. However, some planners may fail to parse a domain definition if this section is missing or contains an unrecognized keyword

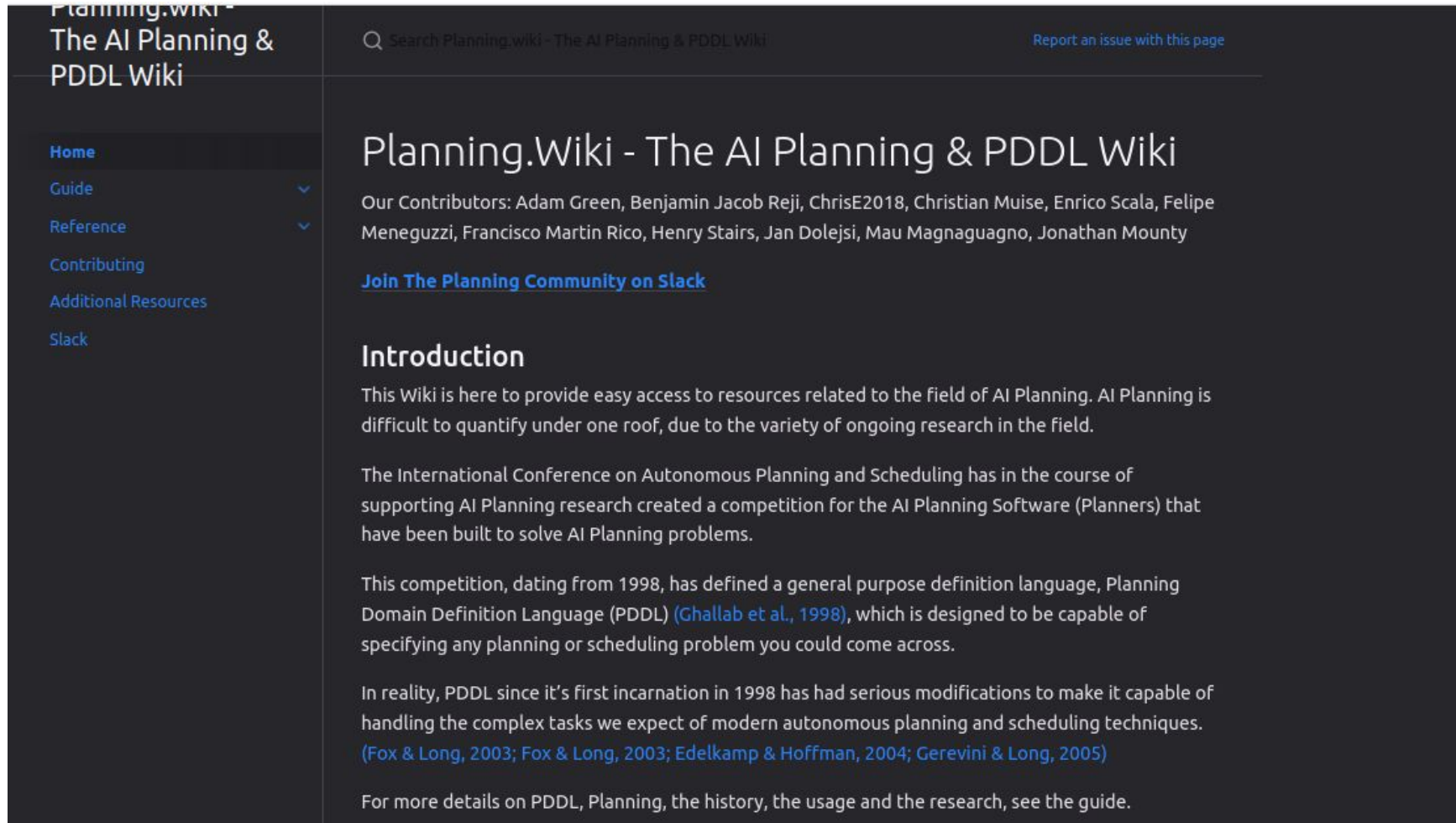
PDDL Notation

Tricks

- . Use the simplest constructs necessary to express the problem.
- . Read the documentation for the specific planner you intend to use.

References

Web Reference



The screenshot shows the homepage of Planning.Wiki. The left sidebar contains a navigation menu with links: Home, Guide, Reference, Contributing, Additional Resources, and Slack. The main content area has a search bar at the top, followed by the title 'Planning.Wiki - The AI Planning & PDDL Wiki'. Below the title, it lists contributors: Adam Green, Benjamin Jacob Reji, ChrisE2018, Christian Muise, Enrico Scala, Felipe Meneguzzi, Francisco Martin Rico, Henry Stairs, Jan Dolejsi, Mau Magnaguagno, and Jonathan Mounty. There is a link to 'Join The Planning Community on Slack'. The 'Introduction' section explains that the wiki provides easy access to resources related to AI Planning, which is difficult to quantify under one roof. It mentions the International Conference on Autonomous Planning and Scheduling's support for AI Planning research through a competition for AI Planning Software (Planners). It also describes the competition's history since 1998, defining a general purpose definition language, Planning Domain Definition Language (PDDL), and its design to be capable of specifying any planning or scheduling problem. The text concludes by stating that PDDL has had serious modifications since its first incarnation in 1998 to handle complex tasks, citing references like (Fox & Long, 2003; Edelkamp & Hoffman, 2004; Gerevini & Long, 2005). A final sentence directs users to the guide for more details on PDDL, Planning, history, usage, and research.

<https://planning.wiki/>

References

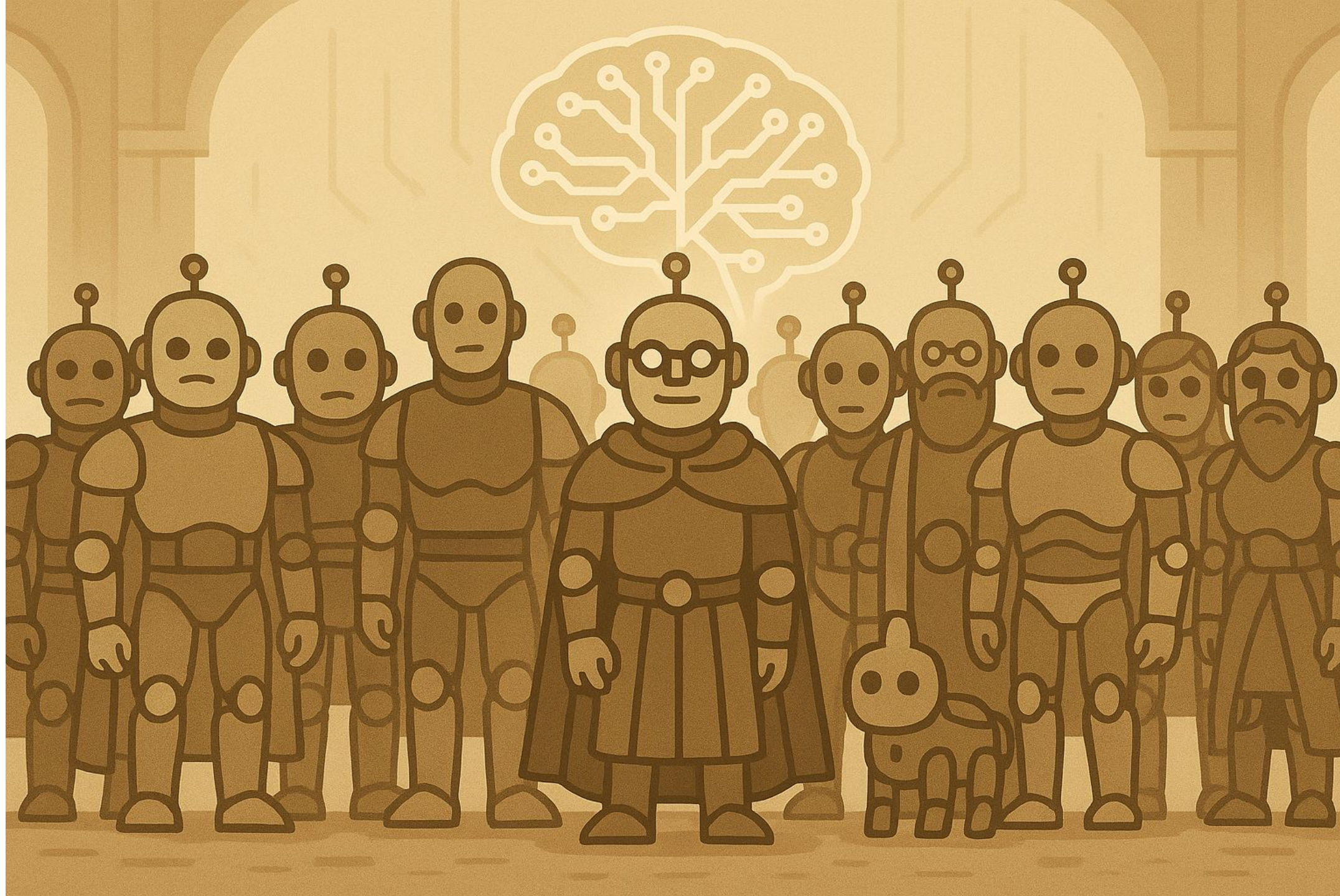
Writing and Debugging PDDL

The main purpose of modelling planning problems in PDDL is to apply automated planning systems to find solution plans.

<http://planning.domains> is an online repository of planning benchmark models, which also includes an on-line editor with PDDL-specific features such as syntax highlighting and semi-automatic instantiation of some common model patterns.

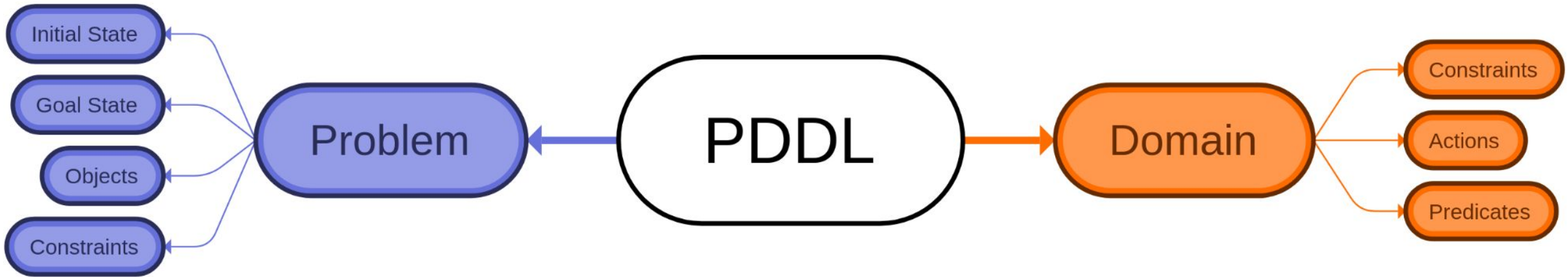
<http://icaps-conference.org/index.php/Main/Competitions> (the “deterministic” tracks). Each of the competitions since 2008 has provided links to the source code of participating planning systems.

<https://github.com/KCL-Planning/VAL> : The VAL tool suite includes a PDDL syntax checker and a plan validator. **A plan validator is a tool that takes as input a problem definition (in PDDL) and a plan, and determines if the plan solves the problem.** Validating manually written plans can be a useful approach to debug the problem definition. An alternative implementation of a plan validator for PDDL is INVALID (<https://github.com/patrikhaslum/INVALID>).



Using PDDL

Parts of a PDDL



PDDL elements

Domain

Domain

The domain definition includes the domain predicates and operators (referred to as actions in PDDL). It may also encompass types (see Typing below), constants, static facts, and various other elements. However, most planners do not support many of these features.

PDDL elements

Domain

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
               (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
               ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
  )

  (:action ACTION_2_NAME
    ...)

  ...)
```

PDDL elements

Domain - Definition

- The domain is defined with the `(define (domain <name>) ...)` construct, where `<name>` is the identifier for the domain.

PDDL elements

Domain - Requirements

- The **:requirements** field specifies the features and constructs the domain uses,
 - **:strips** This is the basic requirement for any PDDL domain. It stands for "Stanford Research Institute Problem Solver" and indicates that the domain uses basic STRIPS-style operators. Allows for simple actions with preconditions and effects that involve adding or deleting predicates.
 - **:typing** This requirement allows the use of types for objects, making the domain definition more expressive and structured. Enables the declaration of different types of objects, which can help in organizing the domain and making the preconditions and effects more readable.
 - **:adl** stands for Action Description Language, an extension of PDDL (Planning Domain Definition Language) that allows for more expressive planning problem definitions. ADL includes additional features that make it possible to describe more complex preconditions and effects for actions, going beyond the capabilities of basic STRIPS operators.
 - **:negative-preconditions** This requirement allows the use of types for objects, making the domain definition more expressive and structured. Enables the declaration of different types of objects, which can help in organizing the domain and making the preconditions and effects more readable.
 - **:disjunctive-preconditions** Allows the use of disjunctions (logical OR) in the preconditions of actions. This enables defining actions that can occur under multiple alternative conditions. Increases the flexibility in specifying when actions can be taken.
 - **:durative-actions** allow for the modeling of actions that take time to execute, with conditions and effects specified at different points in time
- Example: (**:requirements :strips :typing :negative-preconditions**)

PDDL elements

Domain - Predicates

Except for the special predicate `=`, predicates in a domain definition have no intrinsic meaning.

The `:predicates` section of a domain definition only specifies the names of the predicates used in the domain and their number of arguments (and argument types if the domain uses typing).

The "meaning" of a predicate—regarding which combinations of arguments it can be true for and its relationship to other predicates—is **determined by the effects that actions in the domain have on the predicate** and by which instances of the predicate are listed as true in the initial state of the problem definition.

A distinction is commonly made between static and dynamic predicates: a static predicate is not changed by any action.

PDDL elements

Domain - Actions

Actions are the primary elements that define the possible operations within a planning domain. Each action describes how the state of the world can change when the action is executed. Actions consist of parameters, preconditions, and effects:

- **Parameters:** The variables involved in the action.

- **Preconditions:** Conditions that must hold true for the action to be performed.

- **Effects:** Changes that occur as a result of the action.

All parts of an action definition except the name are optional (according to the PDDL specification).

An action that has no preconditions some planners may require an "empty" precondition, in the form `:precondition()` or `:precondition(and)`, and some planners may also require an empty `:parameter` list for actions without parameters).

PDDL elements

Domain - Actions

```
(:action move
  :parameters (?v - vehicle ?from - location ?to - location)
  :precondition (and (at ?v ?from) (not (= ?from ?to)))
  :effect (and (not (at ?v ?from)) (at ?v ?to)))
```


PDDL elements

Domain - Actions

STRIPS (Stanford Research Institute Problem Solver) is a simpler and more limited planning language. It only allows conditions and effects as **conjunctions** of positive literals or simple negations. It does not support disjunctions, quantifiers, or conditional effects. It is easy to implement and widely supported by classical planners.

```
(:precondition (and (at robot1 loc1) (clear loc2)))
(:effect (and (not (at robot1 loc1)) (at robot1 loc2)))
```

ADL (Action Description Language) extends STRIPS by offering **greater expressiveness**: it allows **disjunctions**, **complex negations**, **existential and universal quantifiers**, and **conditional effects**. This enables modeling more complex domains, although not all planners support ADL directly.

```
(:precondition (or (has-key ?r) (knows-code ?r)))
(:effect (when (has-key ?r) (unlocked ?door)))
```

PDDL elements

Domain - Actions

Feature

Conjunctions (and)

Disjunctions (or)

Complex negation (not, imply)

Quantifiers (forall, exists)

Conditional effects (when)

Object typing

Example planner

STRIPS

✓ Supported

✗ Not allowed

✗ Not allowed

✗ Not allowed

✗ Not allowed

✓ Optional

Fast Downward, popf

ADL

✓ Supported

✓ Supported

✓ Supported

✓ Supported

✓ Supported

✓ Optional

VHPOP

PDDL elements

Domain - Preconditions

Strips

An atomic formula:

`(PREDICATE_NAME ARG1 ... ARG_N)`

The predicate arguments must be parameters of the action (or constants declared in the domain, if the domain has constants).

A conjunction of atomic formulas:

`(and ATOM1 ... ATOM_N)`

ADL

A general negation, conjunction or disjunction:

`(not CONDITION_FORMULA)`
`(and CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
`(or CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`

A quantified formula:

`(forall (?V1 ?V2 ...) CONDITION_FORMULA)`
`(exists (?V1 ?V2 ...) CONDITION_FORMULA)`

PDDL elements

Domain - Effects

:effects describe how the state of the world changes as a result of an action being executed. They specify which facts become true and which facts become false after the action occurs. Effects can be simple (adding or deleting facts) or complex (conditional effects that depend on certain conditions).

The effects of an action are not explicitly categorized as "adds" and "deletes."

Negative effects (deletes) are indicated by negation.

PDDL elements

Domain - Effect

Strips

.An added atom:

`(PREDICATE_NAME ARG1 ... ARG_N)`

The predicate arguments must be parameters of the action (or constants declared in the domain, if the domain has constants).

.A deleted atom:

`(not (PREDICATE_NAME ARG1 ... ARG_N))`

.A conjunction of atomic effects:

`(and ATOM1 ... ATOM_N)`

ADL

.A conditional effect:

`(when CONDITION_FORMULA EFFECT_FORMULA)`

The interpretation is that the specified effect takes place only if the specified condition formula is true in the state where the action is executed. Conditional effects are usually placed within quantifiers.

.A universally quantified formula:

`(forall (?V1 ?V2 ...) EFFECT_FORMULA)`

PDDL elements

Domain - Example I

```
(define (domain logistics)
  (:requirements :strips :typing :negative-preconditions :disjunctive-preconditions)
  (:types robot location person)
  (:predicates
    (at ?obj - (either robot person) ?loc - location)
    (in ?person - person ?robot - robot))
  (:action move
    :parameters (?v - robot ?from - location ?to - location)
    :precondition (and (at ?v ?from) (not (= ?from ?to)))
    :effect (and (not (at ?v ?from)) (at ?v ?to)))
  (:action board
    :parameters (?p - person ?v - robot ?loc - location)
    :precondition (and (at ?p ?loc) (at ?v ?loc))
    :effect (and (not (at ?p ?loc)) (in ?p ?v)))
  (:action debark
    :parameters (?p - person ?v - robot ?loc - location)
    :precondition (in ?p ?v)
    :effect (and (not (in ?p ?v)) (at ?p ?loc)))
  (:action visit
    :parameters (?p - person ?loc1 - location ?loc2 - location)
    :precondition (or (at ?p ?loc1) (at ?p ?loc2))
    :effect (and (not (at ?p ?loc1)) (at ?p ?loc2)))
)
```


PDDL elements

Problem

The problem definition contains the objects present in the problem instance, the initial state description and the goal.

Key Components of a PDDL Problem File

Problem Definition: The problem file starts with a definition that includes the problem name and the associated domain.

Objects: Declares the specific instances of the types defined in the domain.

Initial State: Describes the facts that are true at the beginning of the planning problem.

Goal State: Specifies the conditions that must be true for the problem to be considered solved.

PDDL elements

Problem

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
)
```

<domain>	<pre> ::= ine (domain <name>) (def [<extension-def>] [<require-def>] [<types-def>]:typing [<constants-def>] [<domain-vars-def>]:expression- evaluation [<predicates-def>] [<timeless-def>] [<safety-def>]:safety- constraints <structure-def>*)</pre>
<extension-def>	::= (:extends <domain name>+)
<require-def>	::= (:requirements <require-key>+)
<require-key>	::= See Section 15
<types-def>	::= (:types <typed list (name)>)
<constants-def>	::= (:constants <typed list (name)>)
<domain-vars-def>	<pre> ::= ain-variables (:dom <typed list(domain-var-declaration)>)</pre>
<predicates-def>	::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>	::= (<predicate> <typed list (variable)>)
<predicate>	::= <name>
<variable>	::= <name>
<timeless-def>	::= (:timeless <literal (name)>+)
<structure-def>	::= <action-def>
<structure-def>	::= :domain -axioms <axiom-def>
<structure-def>	::= :action -expansions <method-def>

```

<action-def>      ::=      ion <action functor>
                    (:act
                      :parameters ( <typed list (variable)> )
                      <action-def body>)

<action functor>  ::= <name>
<action-def body> ::= [:vars (<typed list(variable)>)]
                    :existential-preconditions
                    :conditional-effects

                    [:precondition <GD>]
                    [:expansion
                      <action spec>]:action- expansions
                    [:expansion :methods]:action- expansions
                    [:maintain <GD>]:action- expansions
                    [ :effect <effect>]
                    [:only-in-expansions <boolean>]:action- expansions
  
```

PDDL elements

Problem

The initial state description (the `:init` section) is simply a list of all the ground atoms that are true in the initial state. **All other atoms are by definition false.**

The `:goal` description is a formula of the same form as an action precondition.

All predicates used in the initial state and goal description should naturally be declared in the corresponding domain.

The initial state and goal descriptions should be ground, meaning that all predicate arguments should be object or constant names rather than parameters.

PDDL elements

Problem

```
(define (problem logistics-problem-robot)
  (:domain logistics)

  (:objects
    robot1 robot2 - robot
    loc1 loc2 loc3 - location
    box1 box2 - object)

  (:init
    (at robot1 loc1)
    (at robot2 loc2)
    (at box1 loc1)
    (at box2 loc3))

  (:goal
    (and
      (at robot1 loc2)
      (at box1 loc2)
      (at box2 loc1)))
)
```

PDDL elements

Example



Exercises

Basic level

Description: Only PDDL exercises

Goal: Get familiar with PDDL formalism

Number: 1-5

Planners

Introduction

In the context of artificial intelligence and automated planning, **is a software tool or system designed to generate a sequence of actions to achieve a desired goal or set of goals in a given environment or domain.**

Planners are used in various applications where automated decision-making or task scheduling is required, such as robotics, scheduling, logistics, and process automation.

There are a plethora of planners, sourced from both the latest International Planning Competition (IPC), which have undergone rigorous testing and verification on standard planning competition domains.

It's important to note that these planners are primarily state-of-the-art research prototypes, and there is no absolute assurance of bug-free performance or flawless operation across all domains.

IPC labs serve as excellent environments for uncovering peculiar bugs within planners.

Planners

Introduction

Classical Planners: These planners operate on fully observable, deterministic planning problems. They typically use search algorithms to explore the state space defined by the initial and goal states and the available actions.

Probabilistic Planners: These planners handle uncertainty in the environment by incorporating probabilistic models. They often use techniques such as Monte Carlo methods or Markov Decision Processes (MDPs) to generate plans.

Temporal Planners: These planners handle temporal aspects of planning problems, such as action durations, deadlines, and concurrency constraints. They often use temporal reasoning techniques to generate plans.

Planners

Categories

Categories satisficing and optimizing

Optimizing planners aim at producing an optimal plan based on a predefined cost function, regardless of the time required to generate it. This could involve minimizing the number of actions or the total cost associated with executing the plan.

Satisficing planners prioritize efficiency, striving to generate a satisfactory plan within a reasonable timeframe.

While an **optimizing planner may prioritize speed over plan quality, this approach is typically undesirable**. A plan with a significantly higher cost but generated faster is rarely preferable to a more efficient alternative.

Satisficing planners seek to strike a balance between time efficiency and plan quality. Some employ heuristic methods to generate a single plan optimized for both criteria, while others offer more flexibility, initially generating a plan and then refining it until interrupted by the user.

Planners

Classic Automated planning

Planning involves the strategic selection and arrangement of actions aimed at altering the states within a system.

In modeling states and transitions, System Σ entails:

- A collection of states, denoted as S , which is recursively enumerable.
- A collection of actions, denoted as A , also recursively enumerable. These actions are under the control of the planner and may include a "no-op" option.
- A set of events, denoted as E , similarly recursively enumerable. Unlike actions, events are beyond the control of the planner and may include a neutral event, labeled as "e."
- A transition function, represented by

$$\gamma: S \times A \times E \rightarrow 2^S$$

mapping the combination of current state, action, and event to a subset of possible future states (2^S).

This function acknowledges that actions and events may sometimes be applied separately .

$$\gamma: S \times (A \cup E) \rightarrow 2^S$$

Planners

Plans

A plan in the context of PDDL is a sequence of actions that, when executed starting from the initial state, leads to the achievement of the specified goals. Plans are typically represented as a list of actions along with their parameters and conditions.

Remember!!:

- **Action Sequence:** The ordered sequence of actions that need to be executed.
- **Action Parameters:** The specific objects or entities involved in each action.
- **Preconditions:** Conditions that must be true before an action can be executed.
- **Effects:** Changes to the state of the world caused by executing an action.

Planners

Resolution

The resolution to a planning dilemma manifests as a plan.

The characteristics of a plan:

Let's denote **D** as a domain definition and **P** as a problem definition within **D**. We'll utilize the subsequent notation to reference the constituents of both the domain and the problem:

- **types(D)** signifies the collection of type names specified in the **:types** section of **D**.
- **predicates(D)** indicates the predicates defined within **D**.
- **actions(D)** denotes the action schemas outlined in **D**.
- For each action schema **a** in **actions(D)**, **name(a)** represents the action's name, and **param(a)** signifies the sequence **x1; ...; xk** of its parameters. For each parameter **xi**, **type-of(xi)** denotes the type name declared for the *i*-th parameter of the action. If the parameter lacks a declared type, **type-of(xi)** is designated as **object**. Similarly, **name(p)** and **param(p)** delineate the name and parameters, respectively, of each predicate **p** within **predicates(D)**.
- **objects(D)** denotes the collection of object names mentioned in the **:objects** section of **P**.
- **init(D)** encompasses the ground facts listed in the **:init** section of **P**.
- **goal(D)** represents the formula stated in the **:goal** section of **P**.

Planners

Plan Validity

Plan validity refers to whether a generated plan is feasible and correct for solving the given planning problem.

1. **Satisfaction of Preconditions:** Each action in the plan must have its preconditions satisfied in the current state before it can be executed.
2. **Consistency with Domain:** The actions in the plan must adhere to the constraints and definitions specified in the domain file, including types, predicates, and action definitions.
3. **Achievement of Goals:** The execution of the plan from the initial state should lead to the achievement of the specified goals in the goal state.
4. **Absence of Conflicts:** Actions in the plan should not lead to conflicts or inconsistencies, such as violating exclusivity constraints or causing state contradictions.
5. **Resource Constraints:** Plans should respect resource constraints, such as time, energy, or resource availability, specified in the domain file.

Planners

Type Correctness

- If the domain doesn't define any type names, all objects default to the type **object**. In such instances, there are no limitations on the objects that can be substituted for parameters within an action schema to generate a ground action instance.
- if the domain employs typing, a crucial requirement for a plan to be deemed valid is that the objects used to instantiate the ground actions correspond correctly to the types required by the action's parameters.
- **it's necessary to precisely define the relationship between objects and types.**
- Given that types in PDDL can establish a hierarchical structure, there exists a subtype-supertype relationship among types, along with a mapping from objects to their respective types.

Planners

State Transitions

- . The actions outlined in a plan instigate alterations to the state.
- . Upon the plan's successful execution, the resultant state satisfies the specified goal.
- . In the discrete and deterministic subset of PDDL, a state is characterized by the collection of facts that hold true within it.

Planners

Non-sequential plans

There are also less strictly ordered forms of plans.

Lifting restrictions on the order of actions is important for scheduling a plan, since it gives flexibility to place the actions in time.

Planners

Algorithms

- The algorithms utilized in planning, aim to find viable routes or series of actions enabling an agent to transition from an initial state to a desired goal state.
- These algorithms systematically explore the agent's environment, guided by a defined strategy to devise a plan.
- Exploration entails methodically searching for feasible plans throughout various states the agent may encounter.
- A state could represent the position and orientation of a robot or a specific arrangement of tiles in an eight-puzzle scenario.
- A state space encompasses the entirety of potential states an agent can occupy or reach when executing feasible actions from a given state.

Planners

Forward Search (Progression)

- Forward search is a method where the planner starts from the initial state and applies actions to progress towards the goal state.
- This is a state-space search technique where each node in the search tree represents a possible state of the world, and edges represent actions that transition between states.

Planners

Forward Search (Progression)

Advantages:

- Forward search is correct (if it returns a plan, that plan is a valid solution).
- Forward search is complete (if a plan exists, the search will eventually find it).

Disadvantages:

- High number of applicable actions at each state.
- Excessively large branching factor.
- Not feasible for plans with many steps.

Planners

Forward Search (Progression)

Key Concepts

1. **Initial State:** The starting point of the search, representing the current state of the world.

2. **Goal State:** The target state that the planner aims to reach.

3. **Actions:** Operators defined in the PDDL domain that can change the state when applied.

4. **Preconditions:** Conditions that must be true for an action to be applicable.

5. **Effects:** Changes that occur in the state after an action is applied.

Planners

Forward Search (Progression)

Steps in Forward Search

.Initialization:

- . Start from the initial state as the root of the search tree.

.Action Application:

- . For each current state, evaluate which actions are applicable based on their preconditions.
- . Apply applicable actions to generate successor states.

.State Transition:

- . Transition to successor states by applying the effects of the actions.
- . Add the new states to the search tree as child nodes of the current state.

.Goal Test:

- . Check if the current state satisfies the goal conditions.
- . If a goal state is reached, a plan (sequence of actions leading from the initial state to the goal state) is found.

.Search Strategy:

- . Use a specific search strategy to explore the search tree. Common strategies include breadth-first search, depth-first search, and heuristic-guided search (e.g., A*).

Planners

Forward Search (Progression)

PDDL Domain

```
(define (domain logistics)
  (:requirements :strips :typing)
  (:types robot location object)
  (:predicates
    (at ?obj - (either robot object) ?loc - location)
    (holding ?r - robot ?obj - object))
  (:action move
    :parameters (?r - robot ?from - location ?to - location)
    :precondition (and (at ?r ?from) (not (= ?from ?to)))
    :effect (and (not (at ?r ?from)) (at ?r ?to)))
  (:action pick_up
    :parameters (?r - robot ?obj - object ?loc - location)
    :precondition (and (at ?r ?loc) (at ?obj ?loc))
    :effect (and (not (at ?obj ?loc)) (holding ?r ?obj)))
  (:action put_down
    :parameters (?r - robot ?obj - object ?loc - location)
    :precondition (holding ?r ?obj)
    :effect (and (not (holding ?r ?obj)) (at ?obj ?loc)))
)
```

PDDL Problem

```
(define (problem logistics-problem-robot)
  (:domain logistics)
  (:objects
    robot1 - robot
    loc1 loc2 loc3 - location
    box1 - object)
  (:init
    (at robot1 loc1)
    (at box1 loc1))
  (:goal
    (at box1 loc2))
)
```

Planners

Forward Search (Progression)

Progression Search Execution

Initial State:

- . robot1 is at loc1
- . box1 is at loc1
- . No objects are being held

Possible Actions from Initial State:

- . (move robot1 from loc1 to loc2)
- . (move robot1 from loc1 to loc3)
- . (pick_up robot1 box1 loc1)

Apply (pick_up robot1 box1 loc1):

- . Preconditions: (at robot1 loc1), (at box1 loc1) (both true)
- . Effects: (not (at box1 loc1)), (holding robot1 box1)
- . New State:
 - . robot1 is at loc1
 - . box1 is being held by robot1

Planners

Forward Search (Progression)

Progression Search Execution

Possible Actions from New State:

- . (move robot1 from loc1 to loc2)
- . (move robot1 from loc1 to loc3)
- . (put_down robot1 box1 loc1)

Apply (move robot1 from loc1 to loc2):

- . Preconditions: (at robot1 loc1), (not (= loc1 loc2)) (both true)
- . Effects: not (at robot1 loc1), (at robot1 loc2)
- . New State:
 - . robot1 is at loc2
 - . box1 is being held by robot1

Possible Actions from New State:

- . (put_down robot1 box1 loc2)
- . (move robot1 from loc2 to loc1)
- . (move robot1 from loc2 to loc3)

Planners

Forward Search (Progression)

Progression Search Execution

Apply (put_down robot1 box1 loc2):

- . Preconditions: (holding robot1 box1) (true)
- . Effects: (not (holding robot1 box1)), (at box1 loc2)
- . New State:
 - . robot1 is at loc2
 - . box1 is at loc2

Goal Test:

- . Check if (at box1 loc2) is true in the current state.
- . Goal is satisfied. The plan is found.

Resulting Plan

- Action 1: (pick_up robot1 box1 loc1)
- Action 2: (move robot1 loc1 loc2)
- Action 3: (put_down robot1 box1 loc2)

This sequence of actions forms a valid plan to move **box1** from **loc1** to **loc2**.

Planners

Backward Search (Regression Planning)

This reasoning method starts from the final goal and moves backward through the rules and available knowledge to determine which facts or conditions are necessary to achieve that goal.

Backward search operates from a conclusion to identify the necessary premises that support it.

Planners

Backward Search (Regression Planning)

Advantages:

- Backward search is correct (if it returns a plan, that plan is a valid solution).
- Backward search is complete (if a plan exists, the search will eventually find it).

Disadvantages:

- Although the branching factor is typically lower than in forward search, the search space remains too large.

Planners

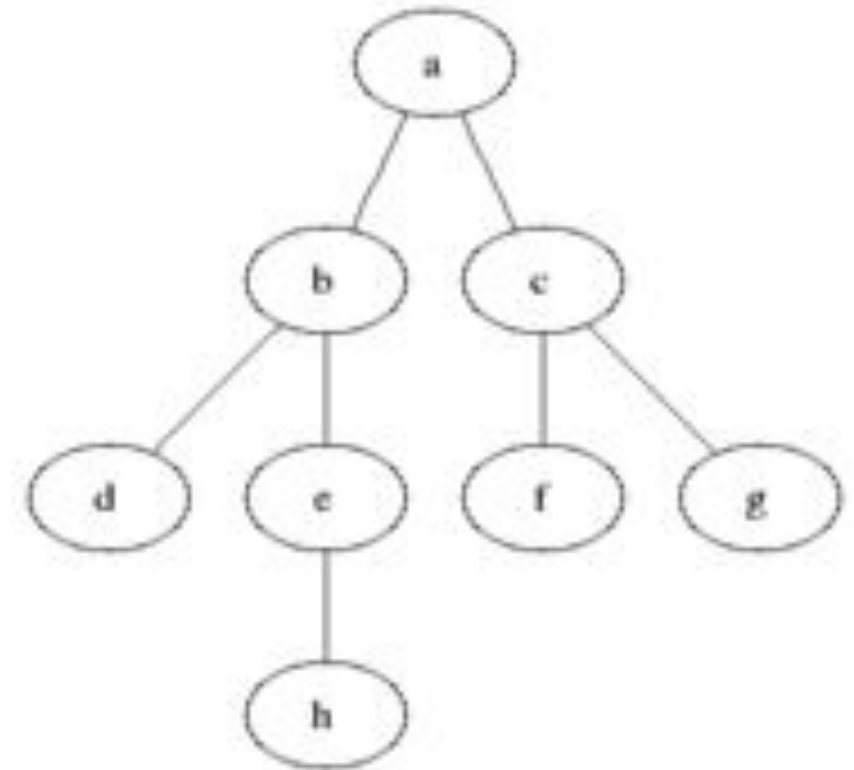
Comparison

Aspect	Forward Search (Progression)	Backward Search (Regression)
Start Point	Initial state	Goal state
Search Direction	From initial state toward goal	From goal state toward initial
Typical Use Case	Most search algorithms (DFS, BFS, A*, IDDFS)	Planning problems where goal is well defined
Action Application	Apply actions to generate successors	Apply inverse actions to regress from goal
Efficiency	Efficient when branching factor is low near the root	Efficient when goal is compact or has fewer preimages
Complexity	Can be high if many irrelevant paths	May be lower if goal constraints prune the space

Planners

Search Strategies

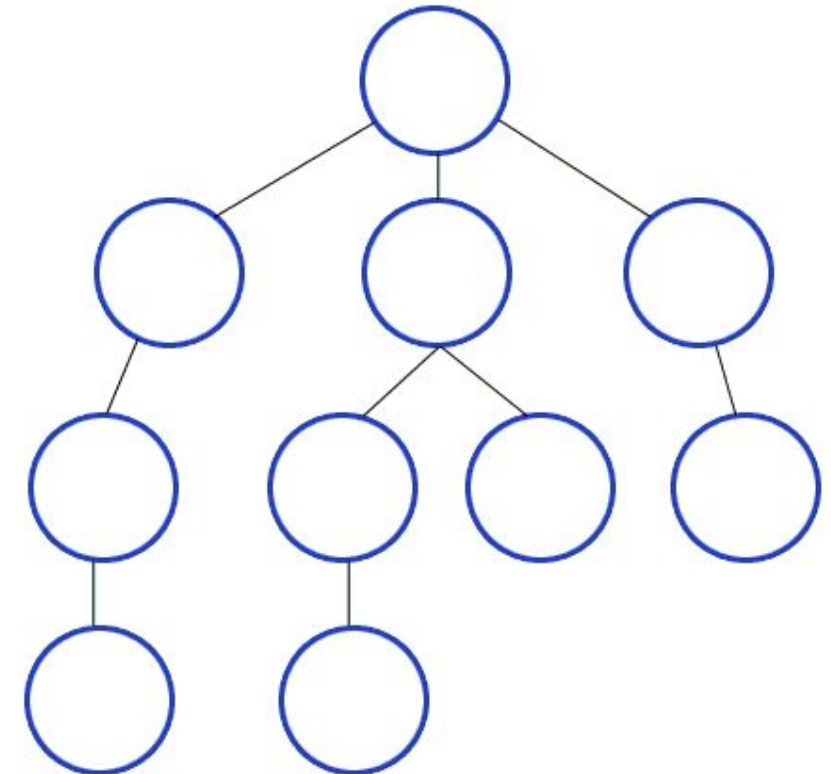
Breadth-First Search (BFS) explores and traverses a search graph or tree. The algorithm begins at a root node and explores all neighboring nodes at the same level before moving on to the next level of nodes.



Planners

Search Strategies

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It begins at a root node (or any arbitrary node in a graph) and uses a stack (explicitly or via recursion) to remember the path, enabling it to backtrack when it reaches a dead end.

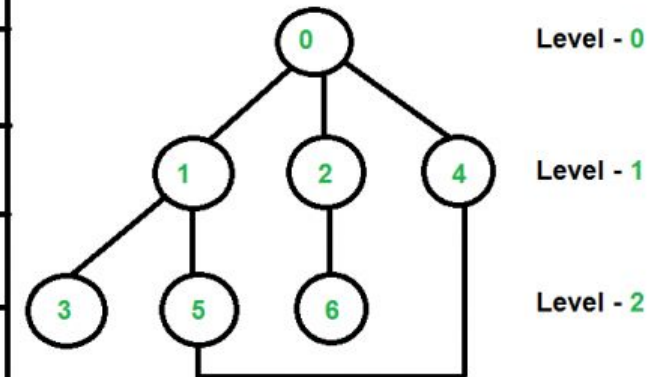


Planners

Search Strategies

Iterative Deepening Search (IDS) (aka **Iterative Deepening Depth First Search (IDDFS)**) involves conducting a series of limited-depth searches, starting with a depth limit of one and gradually increasing until a solution is found. In each iteration, the algorithm performs a limited-depth search on the search tree up to the current depth limit. If no solution is found, the depth limit is incremented by one, and the search is performed again.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Planners

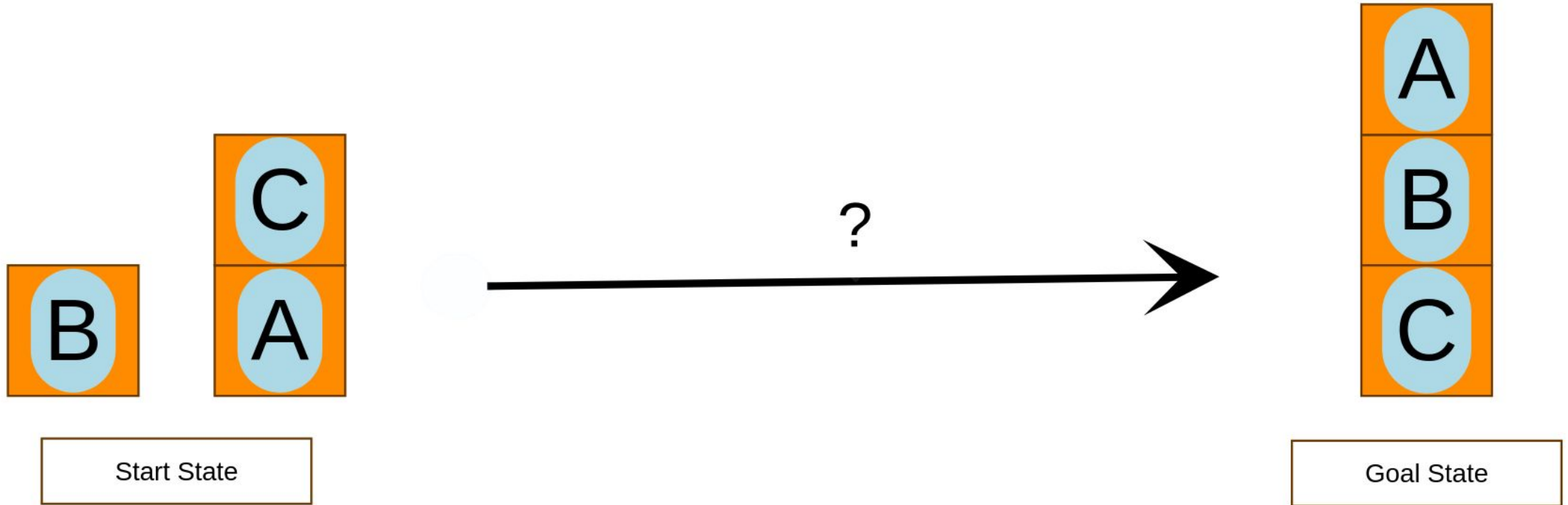
Comparison

Algorithm	Time Complexity	Space Complexity	When to Use
DFS	$O(b^d)$	$O(d)$	<ul style="list-style-type: none"> – When the solution is not necessarily close to the root – When the graph/tree is not very large or is finite
BFS	$O(b^d)$	$O(b^d)$	<ul style="list-style-type: none"> – When memory is not a constraint – When you need the closest solution to the root
IDDFS	$O(b^d)$	$O(b \cdot d)$	<ul style="list-style-type: none"> – When you want the benefits of BFS but have limited memory – Acceptable if slightly slower performance is okay

b = branching factor **d** = depth of the shallowest solution

Planners

Sussman Anomaly



Planners

Sussman Anomaly

Sussman Anomaly refers to a scenario in AI planning where **a seemingly straightforward problem-solving approach encounters unexpected complexity due to the order of operators' application.**

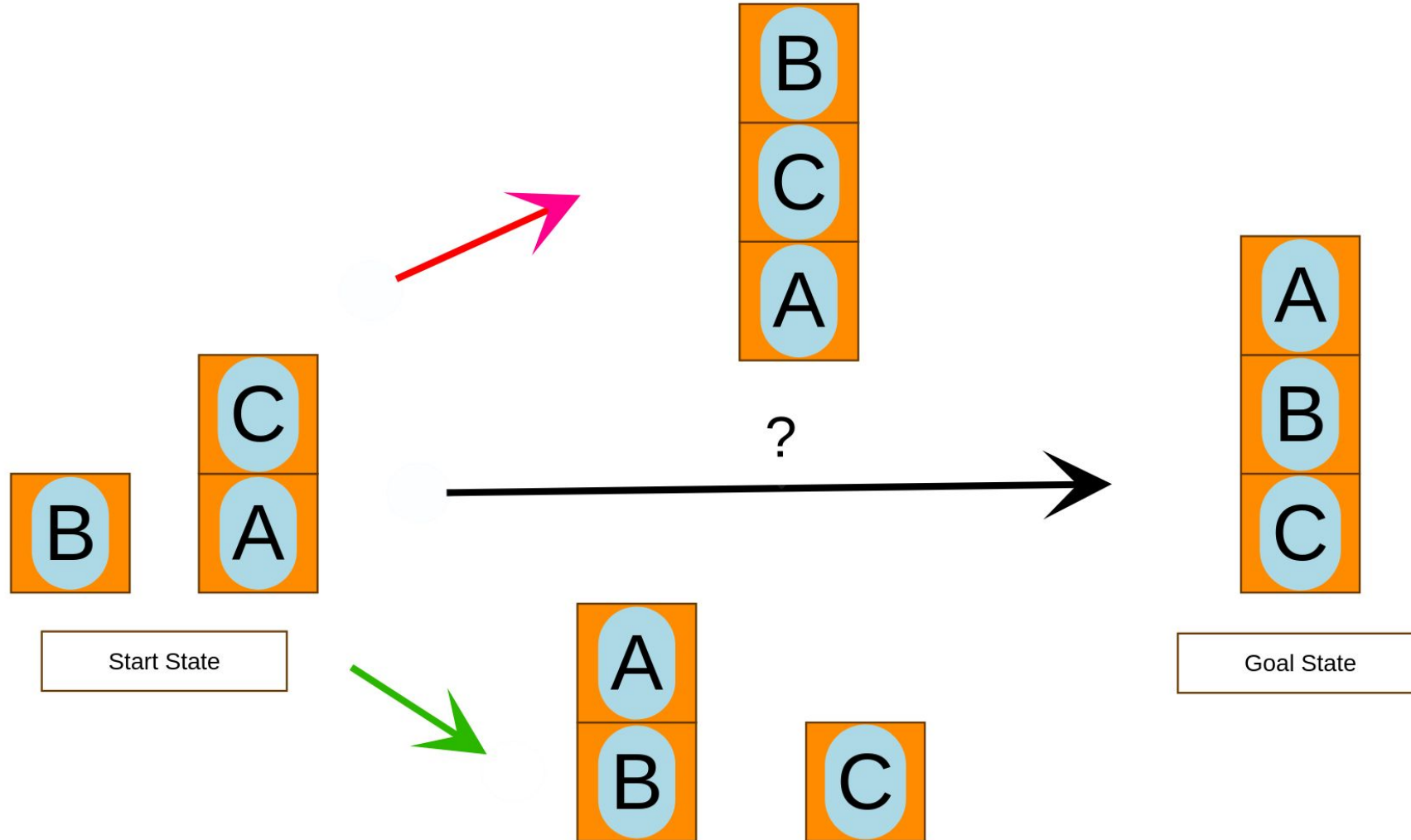
It was first described by Gerald Jay Sussman in his work on computer models of physical systems.

A problem that appears simple can lead to a significantly longer solution path than expected, mainly because of the interplay between parallel and sequential execution of actions.

This anomaly highlights the importance of understanding the interactions between different actions and their potential effects on the overall problem-solving process.

Planners

Sussman Anomaly



Session 4:

Task Planning and PDDL

ACM SIGSOFT Summer School for Software Engineering in Robotics

Exercises

Basic level

Description: Only PDDL exercises

Goal: Get familiar with PDDL formalism

Number: 1-5

Exercises

Medium level: Plansys 2

Description: using PDDL with ROS 2 using Plansys 2

Goal: Get familiar with the integration and deployment of PDDL in ROS 2 environments

Number: 6



PlanSys2

History: RosPlan

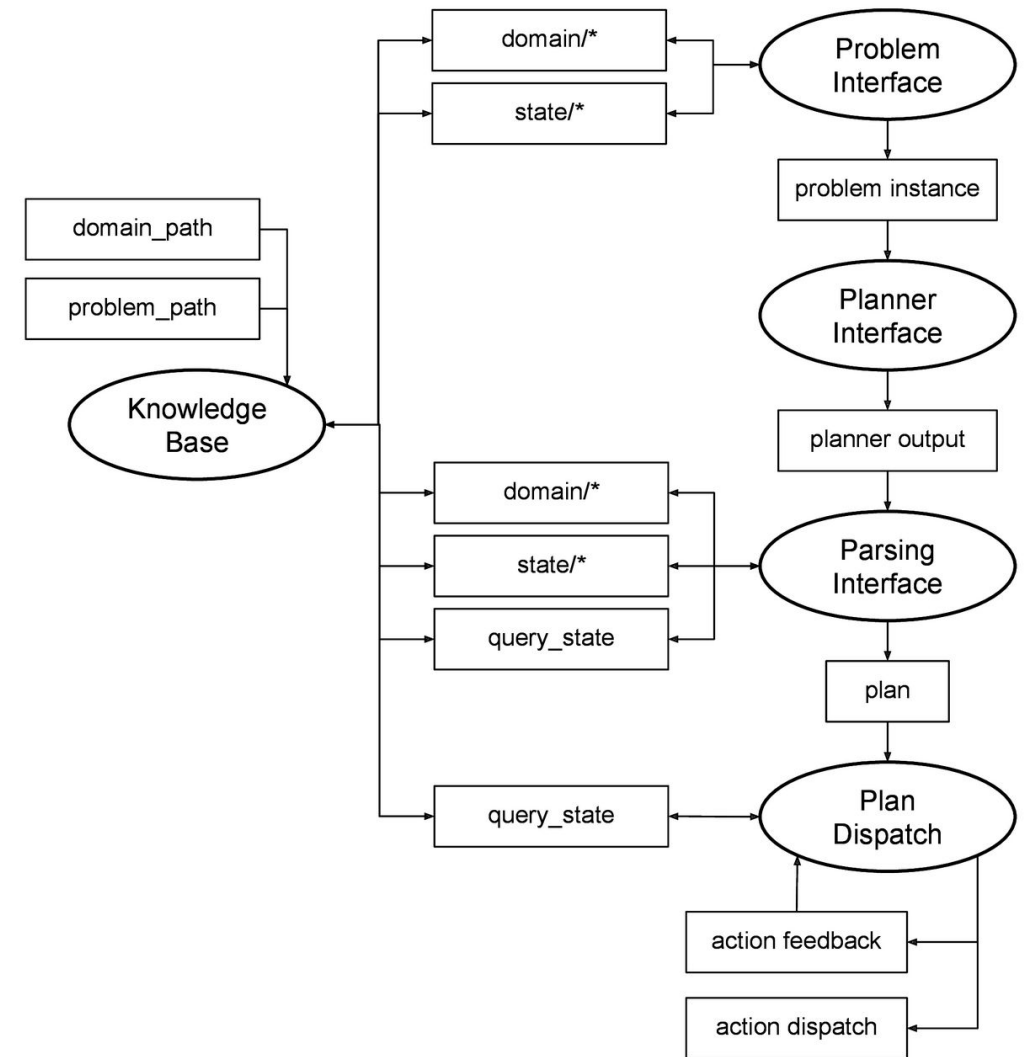
The **Knowledge Base** is used to store a PDDL model.

The **Problem Interface** is used to generate a PDDL problem, publish it on a topic, or write it to file.

The **Planner Interface** is used to call a planner and publish the plan to a topic, or write it to file.

The **Parsing Interface** is used to convert a PDDL plan into ROS messages, ready to be executed.

The **Plan Dispatch** encapsulates plan execution



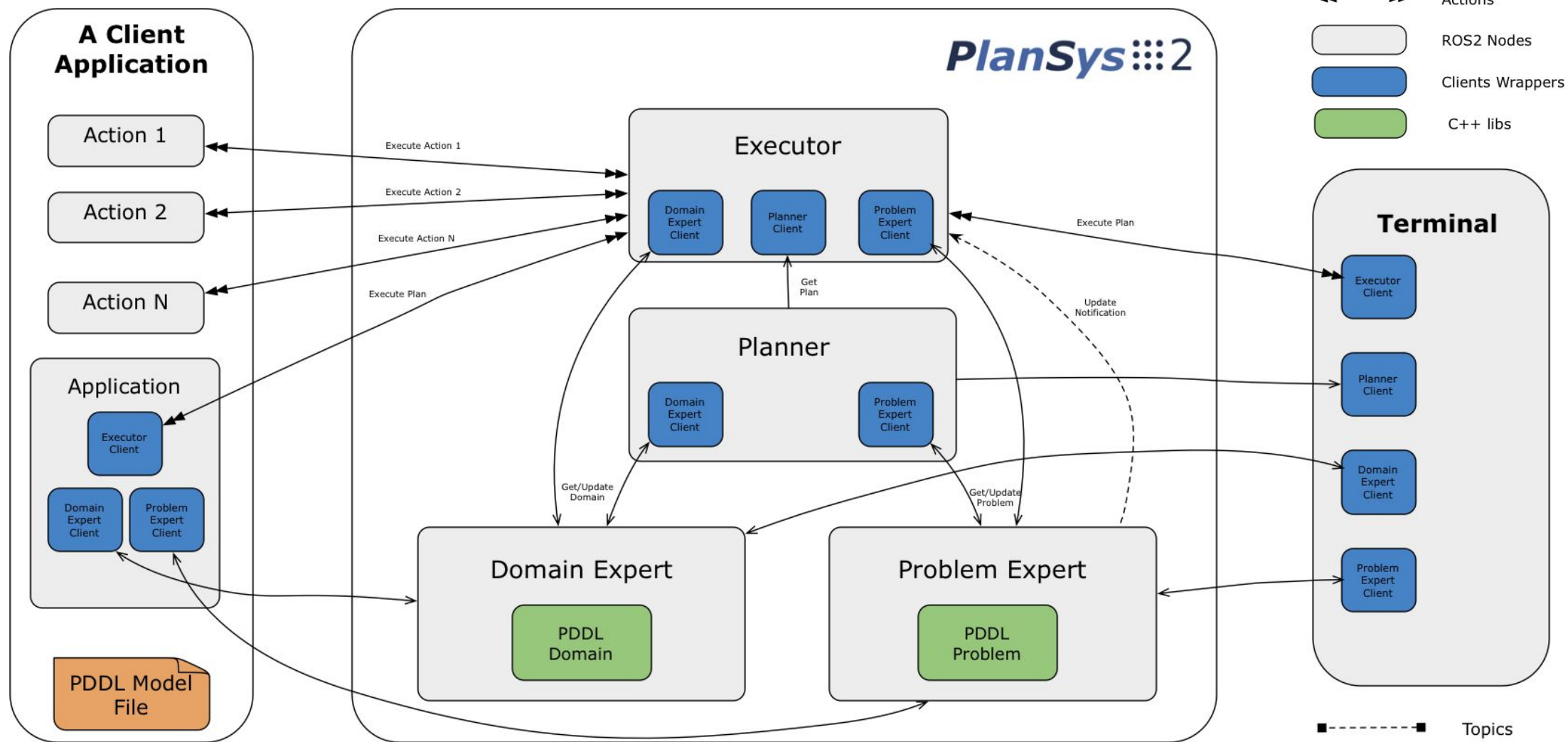
Architecture

RosPlan

- Loads a PDDL domain (and optionally problem) from file.
- Stores the state as a PDDL instance.
- Is updated by ROS messages.
- Can be queried.

PlanSys2

Architecture



PlanSys2

Elements

- **Domain Expert:** Contains the PDDL model information (types, predicates, functions, and actions).
- **Problem Expert:** Contains the current instances, predicates, functions, and goals that compose the model.
- **Planner:** Generates plans (sequence of actions) using the information contained in the Domain and Problem Experts.
- **Executor:** Takes a plan and executes it by activating the *action performers* (the ROS2 nodes that implement each action).

PlanSys2

PlanSys CLI

- Interactive CLI to test and monitor the planning system
- Not required in production — used for testing and debugging
- The state is stored in the PlanSys2 components, not in the terminal
- You can open/close multiple terminal sessions freely

https://github.com/PlanSys2/ros2_planning_system/blob/rolling/plansys2_docs/tutorials/tut_1_terminal.md

https://github.com/PlanSys2/ros2_planning_system_examples/tree/rolling

PlanSys2

CLI

0. Requisites:

Install PlanSys2 (see official docs)

Download the example domain:

```
wget -P /tmp
```

```
https://raw.githubusercontent.com/IntelligentRoboticsLabs/ros2\_planning\_system\_examples/master/plansys2\_simple\_example/pddl/simple\_example.pddl
```

1. Launch PlanSys2:

```
ros2 launch plansys2_bringup plansys2_bringup_launch_distributed.py  
model_file:=/tmp/simple_example.pddl
```

PlanSys2

CLI

2. Launch the terminal in a new shell:

```
ros2 run plansys2_terminal plansys2_terminal
```

You will see:

ROS2 Planning System console. Type "quit" to finish

Tips:

- Arrow keys = command history
- TAB = autocompletion
- Ctrl + D = quit

PlanSys2

CLI

Inspect the Domain

Check domain definition:

```
get domain
```

List elements:

```
get model types
get model predicates
get model actions
```

Get details:

```
get model predicate robot_at
get model action move
```

Define the Problem

Add instances:

```
set instance leia robot
set instance kitchen room
...
```

Add predicates (facts):

```
set predicate (connected kitchen dinning)
set predicate (robot_at leia entrance)
...
```

Define a goal:

```
set goal (and(robot_at leia bathroom))
```

PlanSys2

CLI

Generate the Plan

Compute a plan:

```
get plan
```

Example output:

(askcharge leia entrance chargingroom)

(charge leia chargingroom)

(move leia chargingroom kitchen)

...

Planner creates:

- /tmp/domain.pddl
- /tmp/problem.pddl

Optional – run manually:

```
ros2 run popf popf /tmp/domain.pddl /tmp/problem.pddl
```

PlanSys2

CLI

```
$ ros2 launch plansys2_bringup plansys2_bringup_launch_distributed.pymodel_file:=./domain.pddl
```

```
$ ros2 run plansys2_terminal plansys2_terminal --ros-args -p problem_file:=/problem.pddl
```

Exercises

Expert level: MERLIN 2

Description: using a complete Cognitive Architecture in ROS 2

Goal: Get familiar with Cognitive Architectures

Number: 7





The robot helps the operator to carry some luggage to a car which is parked outside.

Modelling the Problem

Carry My Luggage Example

- Main Goal
 - a. The robot helps the operator to carry a bag to a car parked outside.
- Optional Goals
 - a. Re-entering the arena
 - b. Following the queue on the way back to the arena
- Focus
 - a. Person following, navigation in unmapped environments, social navigation.
- Setup
 - a. Locations:
 - The test takes place both inside and outside the Arena.
 - The robot starts at a predefined location in the living room.
 - b. People: The operator is standing in front of the robot and is pointing at the bag to be carried outside.
 - c. Objects: At least two bags are placed near the operator (within a 2m distance and visible to the robot).

Modelling the Problem

Carry My Luggage Example

Picking up the bag: The robot picks up the bag pointed at by the operator.

Following the operator: The robot should inform the operator when it is ready to follow them.

The operator walks naturally towards the car; after reaching the car, the operator takes the bag back and thanks the robot.

Obstacles: The robot will face 4 obstacles along its way (in arbitrary order): (a) a small object on the ground, (b) a hard-to-see object, (c) a crowd of people obstructing the path outside, and (d) a small area blocked using retractable barriers.

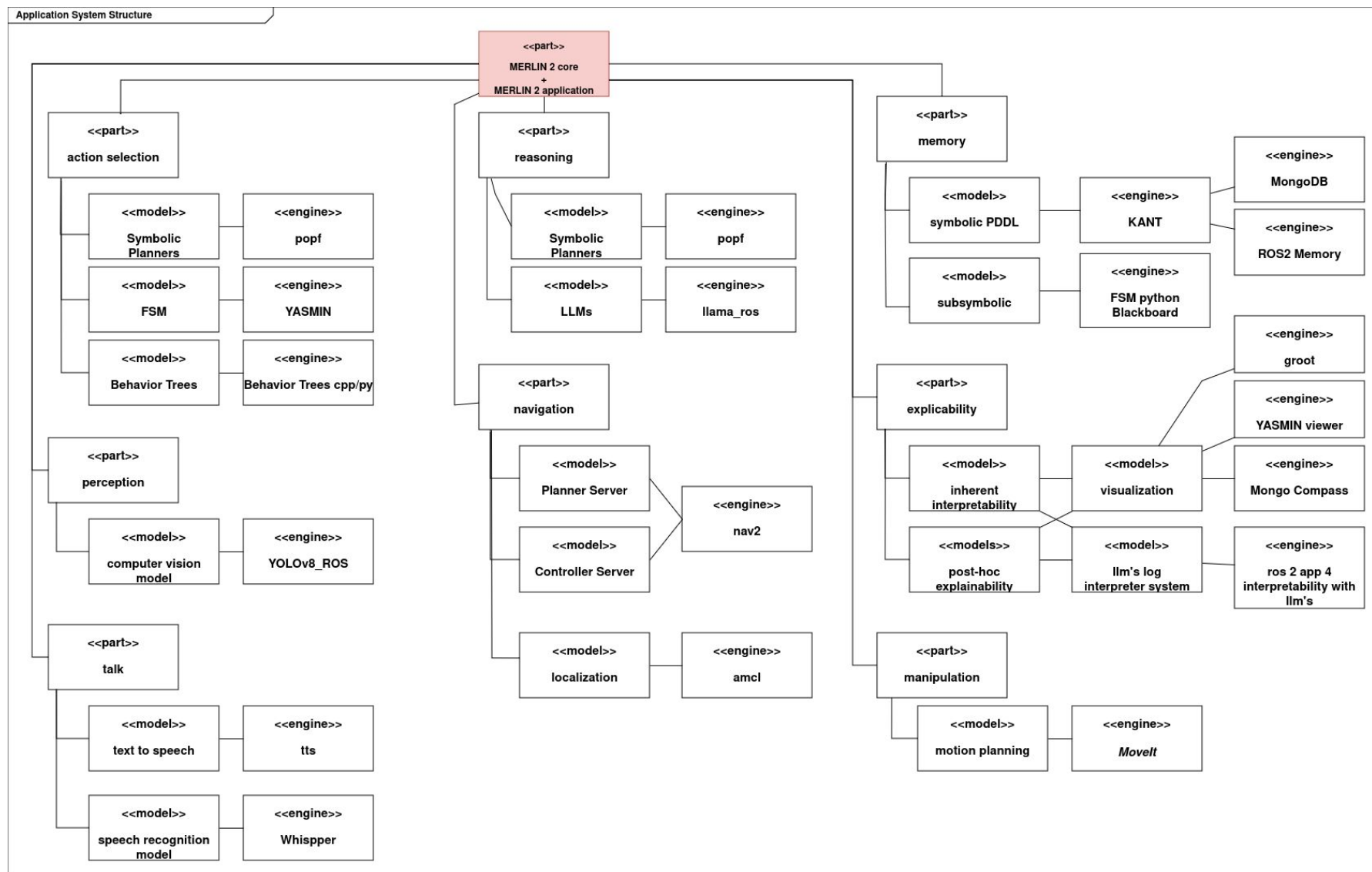
Optional goals:

4.1. Re-entering the arena: The robot returns to the arena, going back in through the entrance.

4.2. Following the queue: After the robot has reached the car, a few of the people that formed the crowd obstructing the robot return to the arena in a queue. The robot can decide to join the queue on its way back to the arena, in a manner that appears natural to the people in the queue.

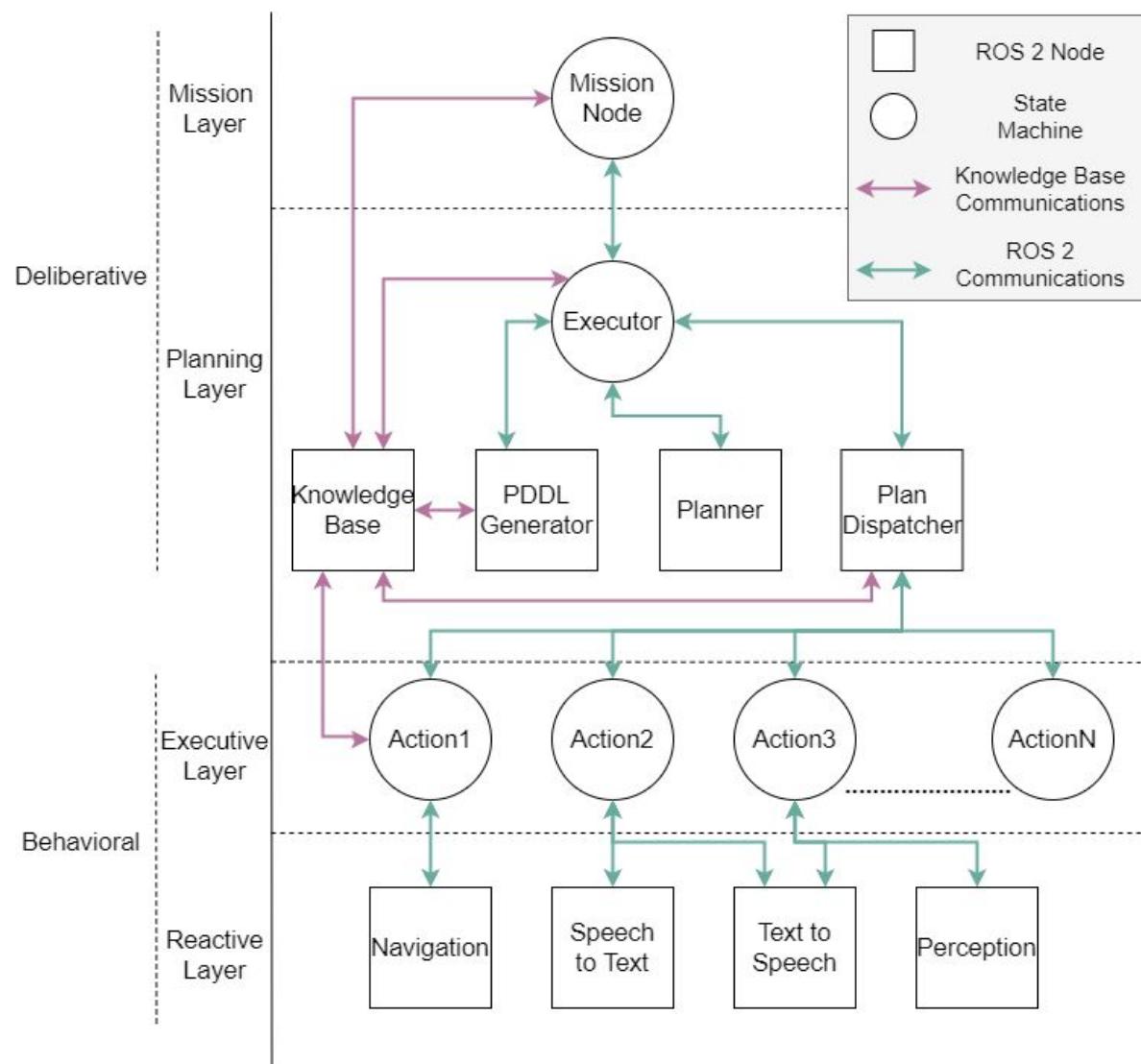
Architecture

MERLIN2



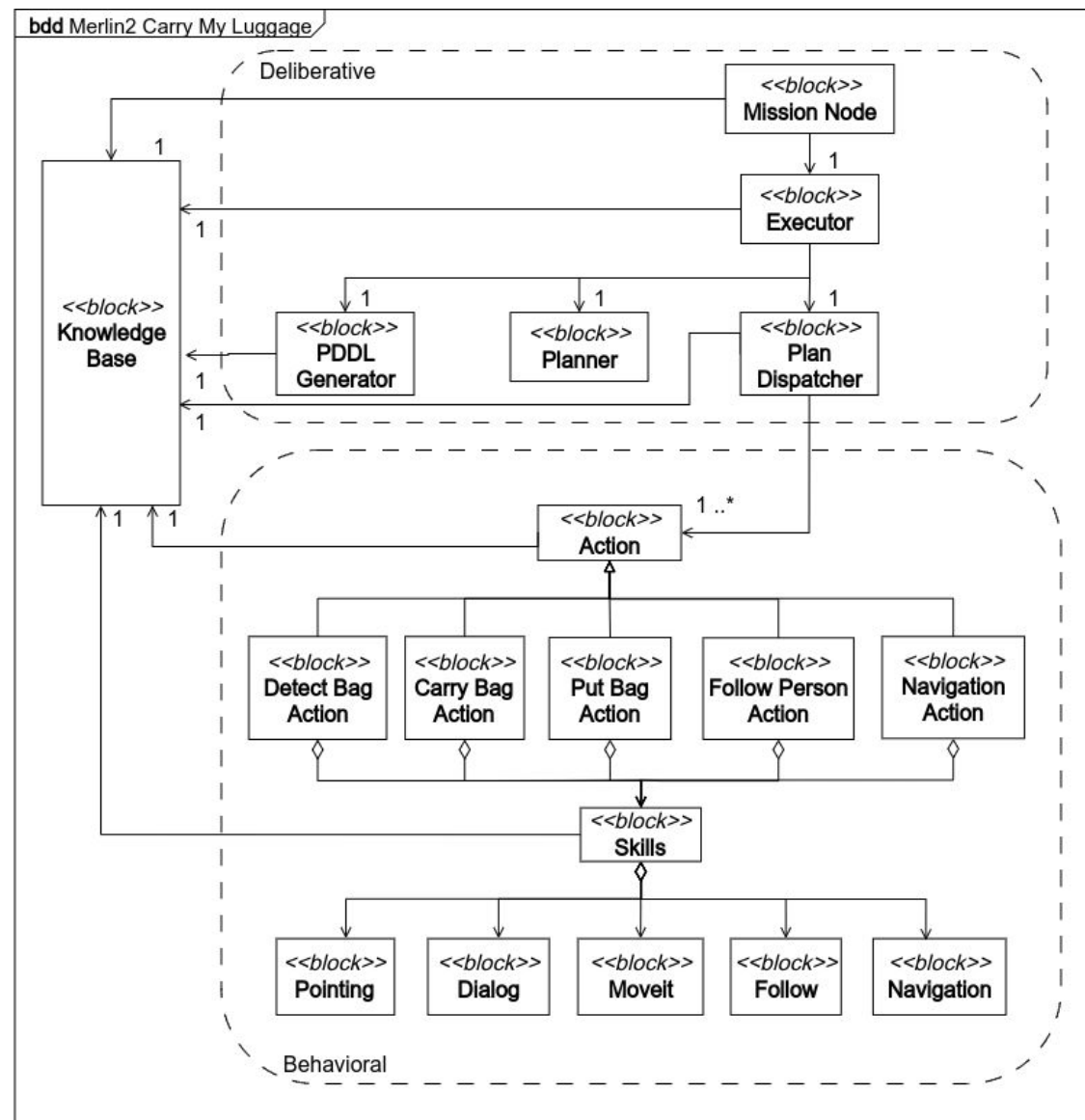
Architecture

MERLIN2



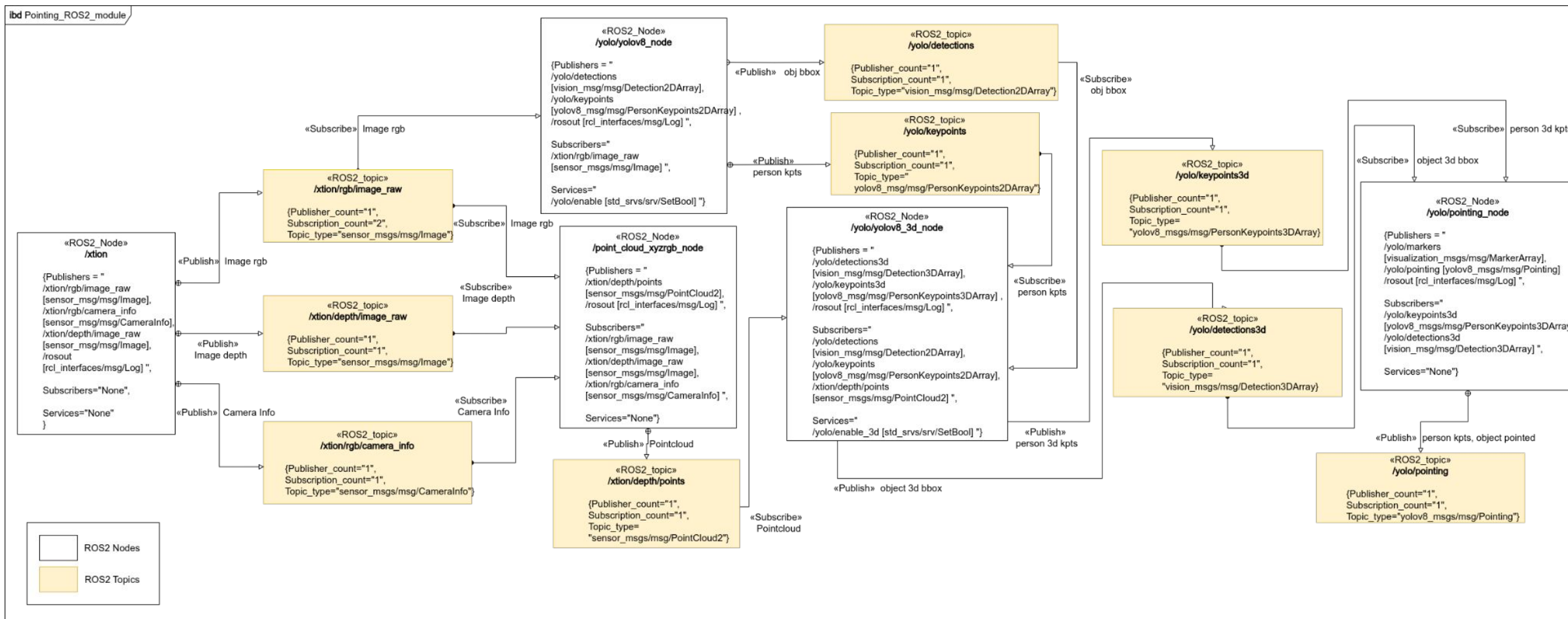
Architecture

MERLIN2



Architecture

Components



Acknowledgement

DMARCE



MINISTERIO
DE CIENCIA, INNOVACIÓN
Y UNIVERSIDADES



Cofinanciado por
la Unión Europea



AGENCIA
ESTATAL DE
INVESTIGACIÓN

DMARCE (EDMAR+CASCAR) Project PID2021-126592OB-C21 +
PID2021-126592OB-C22 funded by MCIN/AEI/10.13039/501100011033 and by ERDF A
way of making Europe

Acknowledgement

CORESENSE



CORESENSE Project

The Horizon Europe CORESENSE project GitHub organization

29 followers

<http://coresense.eu>

info@coresense.eu

README . md

CoreSense Project

Welcome to the CoreSense Project Github organization!

See documentation at [GitHub Pages](#).



**Funded by
the European Union**

The CoreSense Project is a research project funded by the European Commission Horizon Europe programme through grant #101070254. Views and opinions expressed in this organization/repositories are however those of the CoreSense partners only and do not necessarily reflect those of the European Commission or the Horizon Europe programme. Neither the European Union nor the granting authority can be held responsible for them.

Session : Cognitive Architectures, Task Planning and PDDL

ACM SIGSOFT Summer School for Software Engineering in Robotics
Delft (Netherlands)

Francisco J. Rodriguez Lera



universidad
de león