Course "Automated Planning: Theory and Practice"
Chapter 19: PlanSys2 - Robotic Planning System

Teacher: Marco Roveri - marco.roveri@unitn.it

M.S. Course: Artificial Intelligence Systems (LM)
A.A.: 2023-2024
Where: DISI, University of Trento
URL: https://bit.ly/3zOkGk8

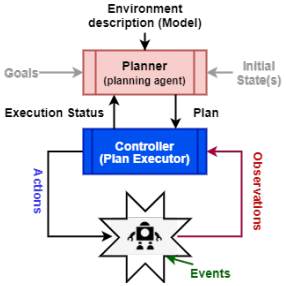
Last updated: Monday 4th December, 2023

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Marco Roveri and FranciscoM. Rico.

Introduction: Planning System

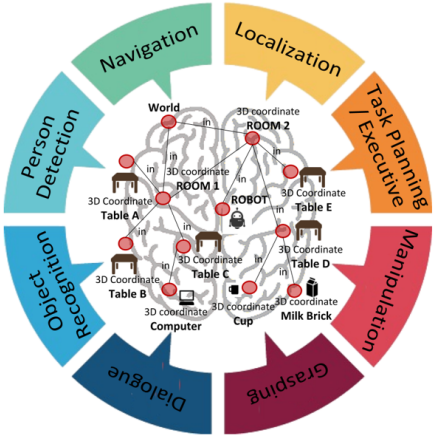
- A Planning System is a software platform to create, execute plans on an (robotic/software) agent and monitor their execution.
  - Relies on one or more models of the agents and of the environment where the agents operate.
  - Manages knowledge (instances, predicates, goals, functions)
    - Provides an interface to add/remove/update knowledge.
    - Provides validation between reality and the model.
  - Provides mechanisms to implement and execute actions
  - Provides mechanisms for monitoring execution at run-time by verifying compliance with preconditions/assumptions and action's effects.
  - Provides mechanisms to handle contingencies via fault-detection-identification and recovery (e.g. re-planning).



We are no longer in a closed world, and the knowledge must be validated with the reality

Planning System: Cortex

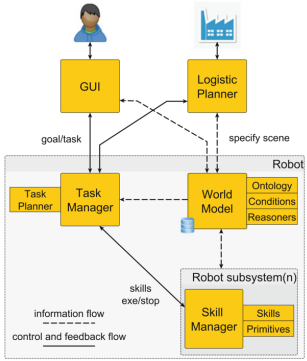
- CORTEX (by Bustos et al. [2]) is a cognitive architecture that leverages AI Planning.
- Maintains a representation of knowledge based on a graph
- Agents access the graph to fulfill their activities
  - Perception
  - Actuation
  - Deliberation/Planning
- Each agent generates task planning using Metric-FF (by Hoffmann [9])



https://robolab.unex.es/wp-content/papercite-data/pdf/luis-pablo-cortex.pdf

Planning System: SkiROS2

- SkiROS2 (by Polydoros et al. [11]) is a platform to create complex robotic behaviors through the composition of skills (software blocks modular) in Behavior Trees (by Marzinotto et al. [10]).
- Available in ROS and ROS2
- The user provides Skills, a scene and a goal
- Uses reasoning based on OWL ontologies
- SkiROS2 offers the following features:
  - A framework to organize the robot behaviors within modular skill libraries
  - A reactive execution engine based on Behavior Trees
  - An integration point for PDDL task planning using the "task planning" skill
  - A semantic database to manage environmental knowledge

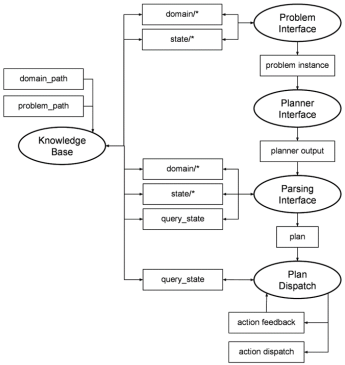


https://github.com/RVMI/skiros2

https://www.researchgate.net/publication/317753352_SkiROS-A_skillbased_robot_control_platform_on_top_of_ROS

Planning System: ROSPlan

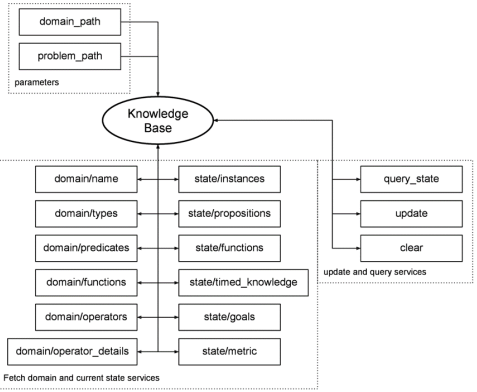
- ROSPlan (by Cashmore et al. [3]) is a framework that provides a collection of tools for AI Planning in a ROS system.
- ROSPlan has a variety of nodes which encapsulate planning, problem generation, and plan execution.
- It possesses a simple interface, and links to common ROS libraries.
  - The Knowledge 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.

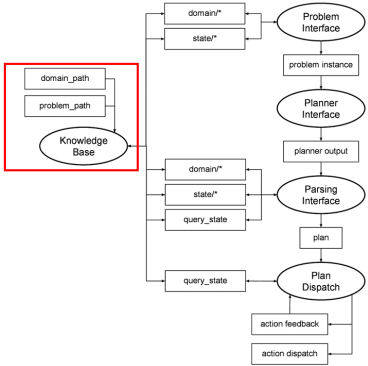


https://github.com/KCL-Planning/ROSPlan
https://kcl-planning.github.io/ROSPlan

https://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/download/10619/10379

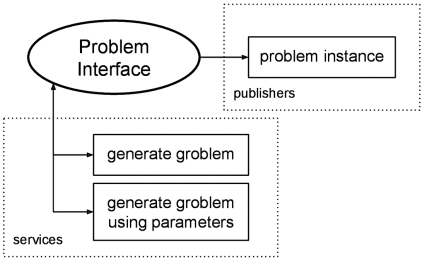
Planning System: ROSPlan (cont.)

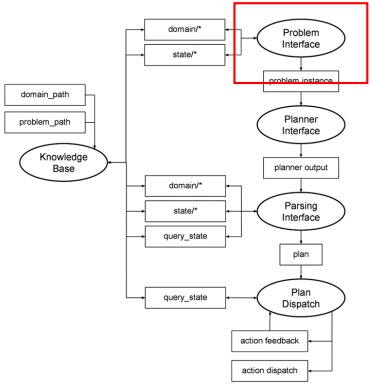




- The Knowledge Base stores the PDDL model: both a domain model and the current problem instance. It supports:
  - Loads a PDDL domain (and optionally problem) from file.
  - Stores the state as a PDDL instance.
  - Is updated by ROS messages.
  - Can be queried.

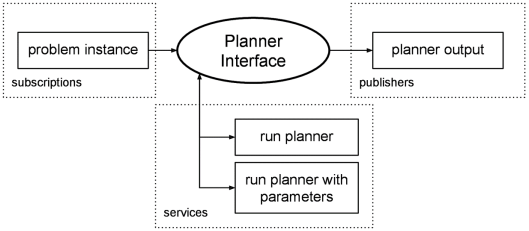
Planning System: ROSPlan (cont.)

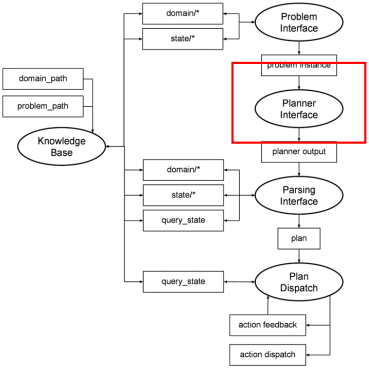




- The Problem Interface node is used to generate a problem instance.
- It fetches the domain details and current state through service calls to a Knowledge Base node and publishes a PDDL problem instance as a string, or writes it to file.

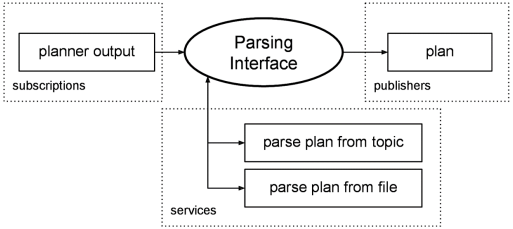
Planning System: ROSPlan (cont.)



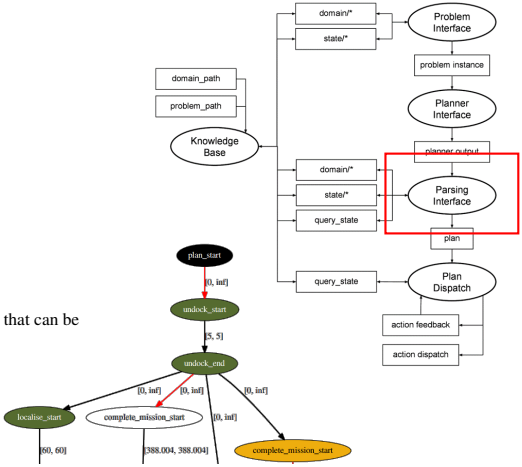


- The Planner Interface node is a wrapper for an AI Planner.
- The planner is called through a service, which returns true if a solution was found.
  - This interface feeds the planner with a domain file and problem instance, and calls the planner with a command line specified by parameter.

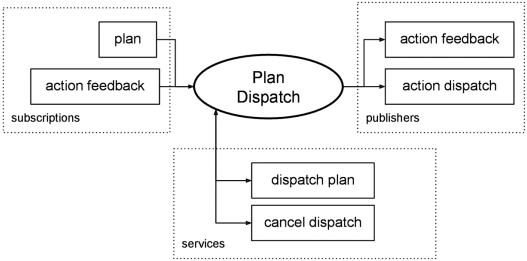
Planning System: ROSPlan (cont.)

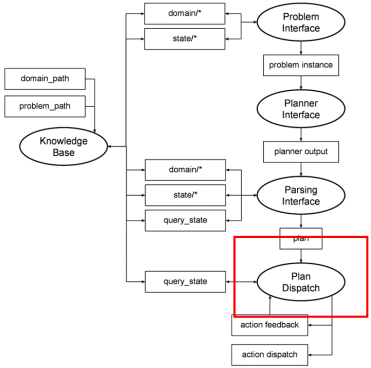


- The Parsing Interface node is used to convert planner output into a plan representation that can be executed, and whose actions can be dispatched to other parts of the system.



Planning System: ROSPlan (cont.)





- The Plan Dispatch includes plan execution, and the process of connecting single actions to the processes which are responsible for their execution.
  - An implementation of the Plan Dispatch node subscribes to a plan topic, and is closely tied to the plan representation of plans published on that topic.

Planning System: a lightweight comparison

- CORTEX and SkiROS2
  - Embed a symbolic planner, but their planners are not directly usable in other architectures, and their substitution with other planners not easy (tightly integrated).
  - No support for time based planning in CORTEX!

- ROSPlan
  - Is built on top of ROS1
    - No plans yet to move to ROS2
  - It has some limitations
    - There is no clean API for adding actions
    - There is no easy support for multiple robots
    - There is no clear interface to interact with the planning system interactively
    - It has not been designed with performance, reliability and flexibility in mind.
  - On the positive side
    - It is the first
    - It has been used in several projects in land, air, and marine applications
    - It has a reasonably good user base

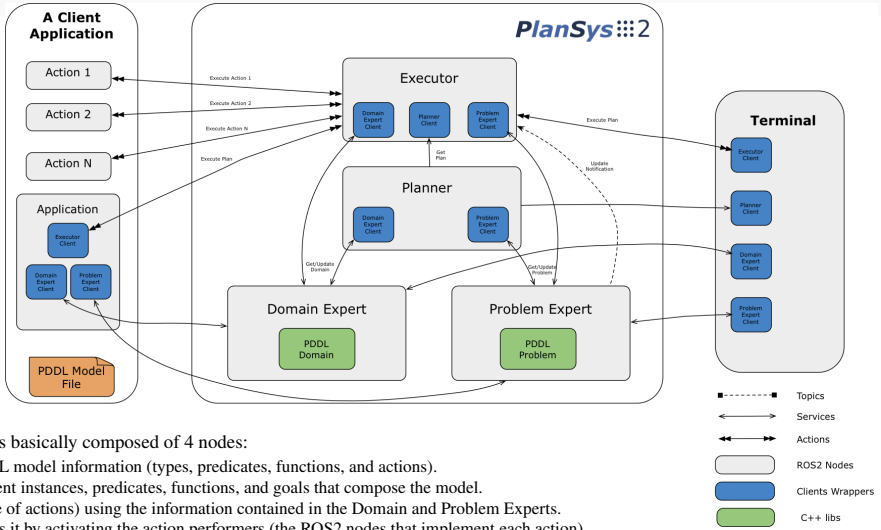
PlanSYS2: Overview I

- It has been inspired by ROSPlan
- It is an efficient, predictable, and reliable infrastructure
  - It is built on top of the ROS2 real-time capable meta-operating system, to address operational safety standards and determinism.
  - It uses the Data Distribution Service (DDS) communication standard adopted in critical systems such as spacecraft, airplanes, hyperloop, or next-generation automobiles.
  - ROS2 also introduces the concept of Managed Nodes (also known as LifeCycle Nodes):
    - These nodes have a clear life cycle based on the states and transitions from their creation.
    - The state are observable, and transitions can be triggered both internally and from outside the node.
    - They behave deterministically.
- It is modular, and each component clearly specify its interfaces, thus facilitating the substitution if needed.

PlanSYS2: Overview II

- Support for Multi-robot and Specialized Action Performers
  - The system is capable of executing plans on different robots and devices.
  - It relies on the ability of ROS2 nodes to communicate with zero-configuration (since DDS uses Multicast and in ROS2, all nodes of the same network can communicate with each other) and a new action auction protocol.
    - There can be multiple nodes in a network that implement the same PDDL action.
    - Each of them can be specialized in executing actions with specific parameters.
- It has been designed with Efficiency and Explainability in mind
  - The plans generated by the plan solver have room to be executed efficiently.
  - Executor's optimization is carried out on the generated plan by analyzing the plans to identify their execution flows, to allow for parallel executions.
  - Each update of the status of any component publishes comprehensible updates to which any visualization or monitoring application can subscribe.

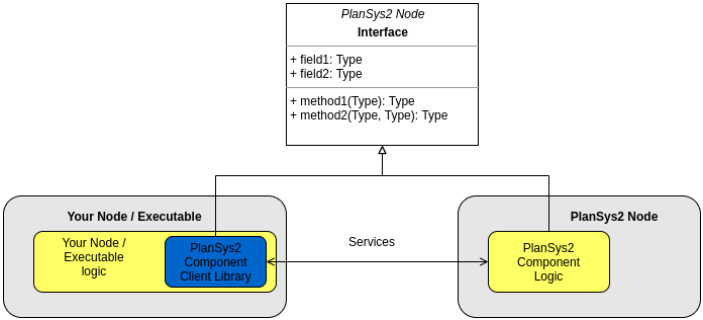
PlanSYS2: Design



- PlanSys2 has a modular design. It is basically composed of 4 nodes:
  - 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: Design (cont.)

- Each of the Domain/Problem Expert, Planner and Executor nodes exposes its functionality using ROS2 services.
- Although, PlanSys2 offers a client library that can be used in any application and hides the complexity of using ROS2 services.

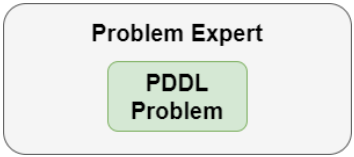


PlanSYS2: Domain Expert

- Its objective is to read PDDL domains from files and make them available to the rest of the components.
  - The PDDL files will be merged
  - Each component contributes with part of the PDDL and corresponding action implementation
    - e.g. one for movement,
    - another for object manipulation, ...

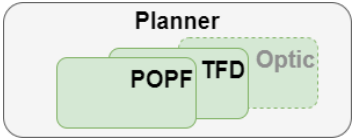
PlanSYS2: Problem Expert

- Contains the knowledge of the system: instances, grounded predicates and functions, and goals.
- Its objective is to make the knowledge available to the rest of the components.



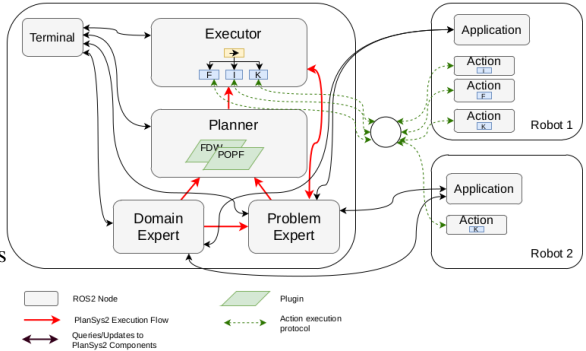
PlanSYS2: Planner

- This component computes the plan to achieve a goal.
  - A plan may be requested by providing a domain acquired from the Domain Expert and a problem acquired from the Problem expert.
  - The domains, problems and solutions are stored in proper name spaces to allow for multiple instances on the same hosting machine (E.g. `/tmp/<namespace>/`).
    - Useful for simultaneous simulation of multiple robots on the same hosting machine.
  - Run a PDDL solver (i.e. a planner) and returns the solution plan (if it exists).
- Each PDDL solver in PlanSys2 is a plugin.
  - Support for POPF by Coles et al. [4], and Temporal Fast Downward by Eyerich et al. [5]
  - POPF is the default
  - Other PDDL solvers can be used easily: OPTIC by Benton et al. [1] forthcoming



PlanSYS2: Executor

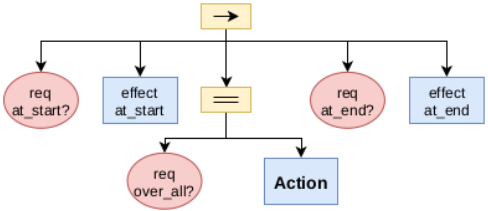
- This component is responsible for executing a provided plan.
- It is the most complex component
- Action execution involves activating the action performers.
  - This task is carried out with the following characteristics:
    - It optimizes its execution, parallelizing the actions when possible.
    - It checks if the requirements are met at runtime.
    - It allows more than one action performer for each action, supporting multirobot execution.



PlanSYS2: Executor - Behavior Tree Builder

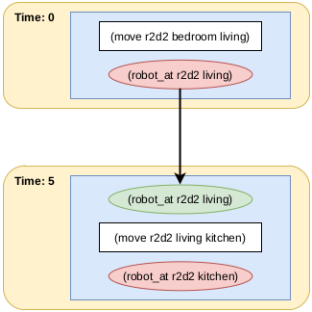
- Once a plan has been generated and it is ready for execution, the Executor converts the plan into a Behavior Tree (BT) [10] before the execution.
  - A "basic" node (corresponding to actions, blue rectangle) can be in NOTRUNNING, RUNNING, SUCCESS, and FAILURE state
  - A "condition" node to check truth of a condition: can be in SUCCESS or FAILURE state
  - There are composition nodes:
    - "=" for parallel composition
    - $\rightarrow$ for sequential composition

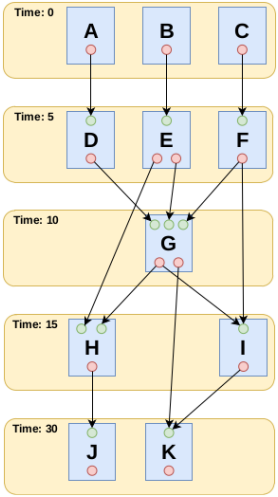
- Each (durative) action in the plan is transformed in the following subtree:



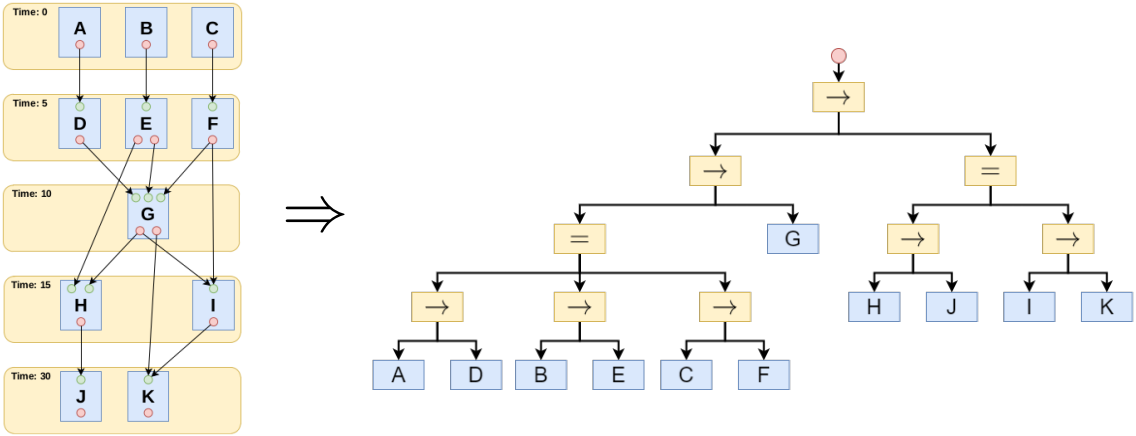
PlanSYS2: Executor - Behavior Tree Builder (cont.)

- The first step is building a planning graph that encodes the action dependencies that define the execution order.
- This is performed by:
  - pairing the effects of an action with the preconditions of a subsequent action
  - taking as reference the time of the calculated plan



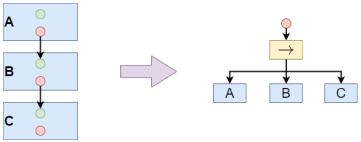


PlanSYS2: Executor - Behavior Tree Builder (cont.)

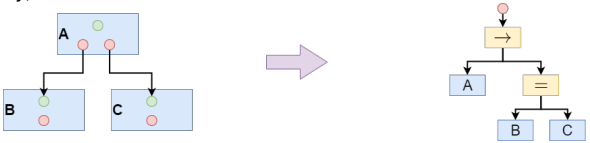


PlanSYS2: Executor - Behavior Tree Builder (cont.)

- If an action a_i in the planning graph is connected only to another action a_j and $|a_i \rightarrow| = 1$ and $|\rightarrow a_j| = 1$ (and $t_{a_i} < t_{a_j}$ is guaranteed by graph construction), this is translated to a sequence structure.

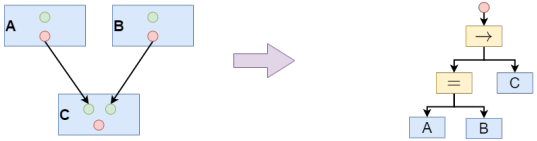


- If an action a_i such that $|a_i \rightarrow| > 1$ is connected to than actions $a_{j,k}$ such that $|\rightarrow a_{j,k}| = 1$, this is translated in a BT with one sequence that executes a_i before a parallel control node that contains all actions $a_{j,k}$.

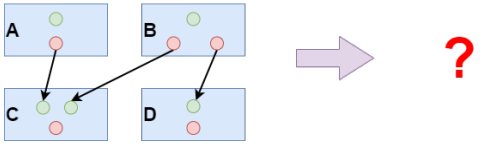


PlanSYS2: Executor - Behavior Tree Builder (cont.)

- If several actions $a_{j,k}$ such that $|a_{j,k} \rightarrow| = 1$, converge to the same action unit a_i, this is translated in a BT with one sequence that executes a parallel control node that contains action units $a_{j,k}$ before a_i.



- If an action unit a_i such that $|a_i \rightarrow| > 1$ is connected with an action a_j such that $|\rightarrow a_j| > 1$, the transformation to BT is not possible with no further information.

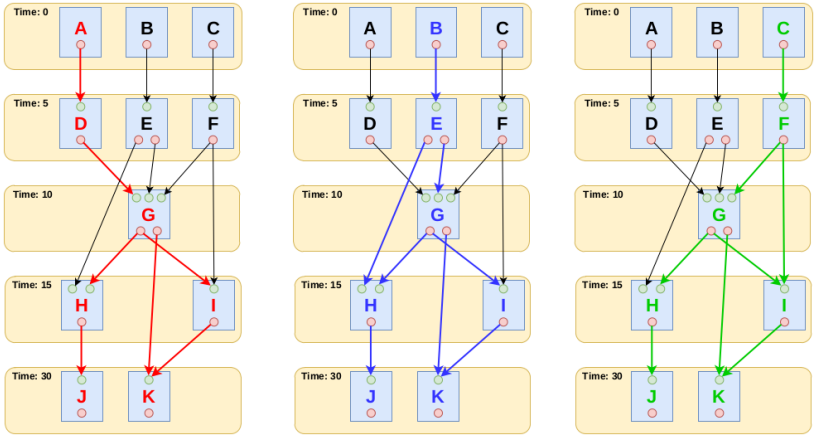


PlanSYS2: Executor - Behavior Tree Builder (cont.)

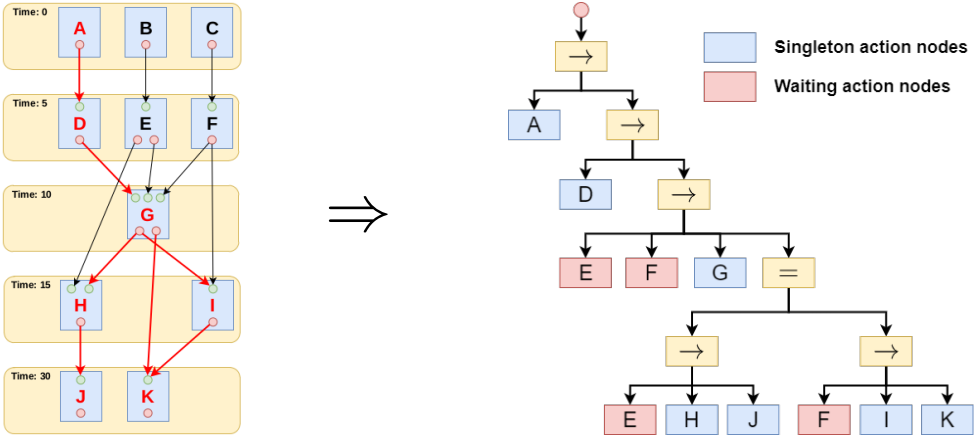
- To overcame limitations of the "simple" construction has been proposed an approach that leverages following concepts:
  - Execution flows: An execution flow starts from an action a_i such that $|\rightarrow a_i| = 0$, and add the actions in the path following the arc effect $\rightarrow$ requirement.
  - Singleton action: Each action unit a_i in the generated BT is a singleton, i.e., it can appear in different points in the BT, but it refers to the same action instance. If an action unit a_i has already returned SUCCESS when executed in one branch, it will return SUCCESS if it is ticked in any other branch.
  - Waiting nodes: This control node refers to an action instance a_i, and it returns RUNNING if a_i has never returned SUCCESS.

PlanSYS2: Executor - Behavior Tree Builder (cont.)

- Once created the planning graph, we identify the execution flows:



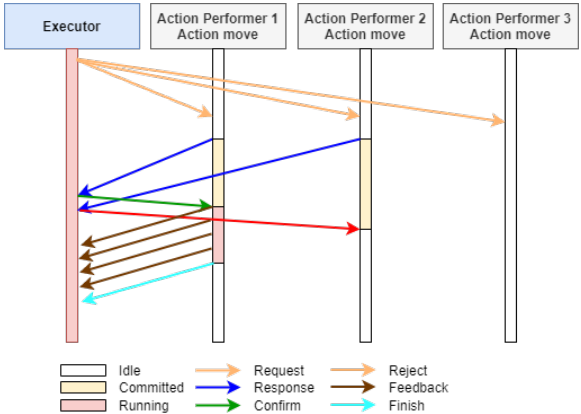
PlanSYS2: Executor - Behavior Tree Builder (cont.)



- Each flow is executed in parallel.
  - The flows can overlap, and it is not a problem since the BT that executes an action is a singleton.

PlanSYS2: Executor - Action delivery protocol

- Leverages a bidding-based delivery protocol
  - When the Executor must execute an action, it requests which action performer can execute it.
  - The Executor receives replies to his request.
  - The Executor confirms one of them (the first to answer), rejecting the others.
  - If none are found, the execution of the plan is aborted (may be after having tried some finite times to get a reply).



PlanSys2: Installation

- It relies on ROS2
  - Tested on Ubuntu 20.04.3 LTS
  - Tested on Ubuntu 22.04.1 LTS

- Pre-configured binaries:
  - `sudo apt install ros-<distro>-plansys2-*`
  - Example: `foxy` ROS2 distribution
    - E.g.: `command> sudo apt install ros-foxy-plansys2-*`
  - Works also for `humble` (replace `foxy` with `humble` in the following slides)

PlanSys2: Installation (cont.)

- Build from sources
  - Install ROS2 through build instructions for your distribution (e.g., `foxy`, `humble`):
    - `https://index.ros.org/doc/ros2/Installation`
  - Build sources
    - Create a workspace (e.g. `ps2-ws`):

```
command> mkdir ps2-ws
```

    - Clone the repositories (e.g. `master` or `foxy-devel`)

```
command> git clone https://github.com/PlanSys2/ros2_planning_system.git}
command> git clone https://github.com/PlanSys2/plansys2_tfd_plan_solver.git
command> git clone https://github.com/PlanSys2/ros2_planning_system_examples.git
```

    - Build the sources

```
command> rosdep install -y -r -q --from-paths src --ignore-src --rosdistro foxy
command> colcon build --symlink-install
```

PlanSys2: Using The Terminal

- Run:
  - `ros2 run plansys2_terminal plansys2_terminal`

```
command> ros2 run plansys2_terminal plansys2_terminal
[INFO] [1637759812.901738484] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
>
```

  - Available commands:
    - `get`
    - `set`
    - `remove`
    - `run`
    - `check`
    - `source`
- One may need at least to load a domain:

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

PlanSys2: Using The Terminal - commands

- `get domain|model|problem|plan`
  - `get domain`
  - `get model types|predicates|functions|actions`
    `get model predicate <pred_name>`
    `get model function <func_name>`
    `get model action <act_name>`
  - `get problem instances|predicates|functions|goal`
  - `get plan`
- `set instance|predicate|function|goal` (next slide)
- `remove instance|predicate|function|goal`
- `check actors`
- `run`
- `source <file_name> [0|1]` // 1 echo commands on standard output, 0 no echo!

PlanSys2: Using The Terminal - set command examples

- In the PlanSys2 terminal one can create the problem

```
set instance leia robot
set instance entrance room
...
set instance chargingroom room

set predicate (connected entrance dinning)
set predicate (connected dinning entrance)
...
set predicate (battery_low leia)
set predicate (robot_at leia entrance)

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

PlanSys2 Commands:
{ commands.txt }

PlanSys2: Using The Terminal - running planner

- One can run the planner

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

A simple Planning Package

- Types:
```
(:types  robot room )
```
- Predicates:
```
(:predicates (robot_at ?r - robot ?ro - room) (connected ?ro1 ?ro2 - room)
             (battery_full ?r - robot)        (battery_low ?r - robot)
             (charging_point_at ?ro - room))
```
- Actions:
  - move
```
(:durative-action move
  :parameters (?r - robot ?r1 ?r2 - room)
  :duration ( = ?duration 5)
  :condition (and (at start (connected ?r1 ?r2))
                  (at start (robot_at ?r ?r1))
                  (over all (battery_full ?r)) )
    :effect (and (at start (not (robot_at ?r ?r1)))
                 (at end (robot_at ?r ?r2)) )
)
```
  - askcharge
  - charge
- Simple PDDL Complete Domain:
  { simple_example.pddl }

A simple Planning Package (cont.)

- Clone the repository in your workspace (e.g. `ps2-ws`)

```
command> mkdir ps2-ws
command> cd ps2-ws
command> git clone -b foxy-devel
    https://github.com/PlanSys2/ros2_planning_system_examples.git
    example
```

- Visit directory `example/ros2_planning_system_examples/plansys2_simple_example`

```
command> tree
.
+-- CHANGELOG.rst
+-- CMakeLists.txt
+-- README.md
+-- launch
|   +-- plansys2_simple_example_launch.py
+-- package.xml
+-- pddl
|   +-- simple_example.pddl
+-- src
    +-- ask_charge_action_node.cpp
    +-- charge_action_node.cpp
    +-- move_action_node.cpp
```

A simple Planning Package: action implementation

- Action `move` (excerpt of file `move_action_node.cpp`)

```
class MoveAction : public plansys2::ActionExecutorClient
{
public:
  MoveAction() : plansys2::ActionExecutorClient("move", 250ms) {
    progress_ = 0.0;
  }
private:
  void do_work() {
    if (progress_ < 1.0) {
      progress_ += 0.02;
      send_feedback(progress_, "Move running");
    } else {
      finish(true, 1.0, "Move completed");
      progress_ = 0.0;
      std::cout << std::endl;
    }
    std::cout << "\r\e[K" << std::flush;
    std::cout << "Moving ... [" << std::min(100.0, progress_ * 100.0)
              << "%]  " << std::flush;
  }
  float progress_;
};
```

A simple Planning Package:: action implementation (cont.)

- Action `move` (excerpt of file `move_action_node.cpp`)

```
int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto node = std::make_shared<MoveAction>();

  node->set_parameter(rclcpp::Parameter("action_name", "move"));
  node->trigger_transition(lifecycle_msgs::msg::Transition::TRANSITION_CONFIGURE);

  rclcpp::spin(node->get_node_base_interface());

  rclcpp::shutdown();

  return 0;
}
```

A simple Planning Package: action implementation (cont.)

- `class MoveAction : plansys2::ActionExecutorClient("move", 250ms)` indicates that each 250ms (4Hz) the method `do_work()` will be called.
- `plansys2::ActionExecutorClient` has an API, with the following three relevant protected methods for the actions:

```
const std::vector<std::string> & get_arguments();
void send_feedback(float completion, const std::string & status = "");
void finish(bool success, float completion, const std::string & status = "");
```

where:

- `get_arguments()` returns the list of arguments of an action.
  - For example, when executing `(move leia chargingroom kitchen)`, it will return `{"leia", "chargingroom", "kitchen"}`
- `send_feedback(...)` sends feedback of percentage of completion.
- `finish(...)` indicates that the action has finished, and send back if it successfully finished, the completion value in [0-1] and optional info.
  Then, the node will pass to inactive state.

A simple Planning Package: ROS2 launcher

- The launcher must include the PlanSys2 launcher, selecting the domain, and it runs the executables that implements the PDDL actions:

```
def generate_launch_description():
    example_dir = get_package_share_directory('plansys2_simple_example')
    namespace = LaunchConfiguration('namespace')

    declare_namespace_cmd = DeclareLaunchArgument('namespace',
                                                  default_value='',
                                                  description='Namespace')

    plansys2_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(
            get_package_share_directory('plansys2_bringup'),
            'launch',
            'plansys2_bringup_launch_monolithic.py')),
        launch_arguments={
          'model_file': example_dir + '/pddl/simple_example.pddl',
          'namespace': namespace
          }.items())

    ...
```

A simple Planning Package: ROS2 launcher (cont.)

- The launcher must include the PlanSys2 launcher, selecting the domain, and it runs the executables that implements the PDDL actions:

```
def generate_launch_description():

    ... # as from previous slide

    move_cmd = Node(package='plansys2_simple_example',
                    executable='move_action_node', name='move_action_node',
                    namespace=namespace, output='screen', parameters=[])

    charge_cmd = Node(...) # similar to above
    ask_charge_cmd = Node(...) # similar to above

    ld = LaunchDescription()
    ld.add_action(declare_namespace_cmd)
    # Declare the launch options
    ld.add_action(plansys2_cmd)
    ld.add_action(move_cmd)
    ld.add_action(charge_cmd)
    ld.add_action(ask_charge_cmd)
    return ld
```

A simple Planning Package: action implementation (cont.)

- The action node, once created, must pass to `inactive` state to be ready for being executed.

```
auto node = std::make_shared<MoveAction>();

node->set_parameter(rclcpp::Parameter("action", "move"));
// Move the action node to inactive/idle state
node->trigger_transition(lifecycle_msgs::msg::Transition::TRANSITION_CONFIGURE);

rclcpp::spin(node->get_node_base_interface());
```

- The parameter `action` sets what action implements the node object.

A simple Planning Package (cont.)

- Compile the repository in your workspace

```
command> colcon build --symlink-install
command> rosdep install --from-paths example --ignore-src -r -y
command> colcon build --symlink-install
```

A simple Planning Package: Running the example

- Open a terminal and run PlanSys2 framework

```
command> ros2 launch plansys2_simple_example plansys2_simple_example_launch.py
```

- Open another terminal and run PlanSys2 terminal

```
command> ros2 run plansys2_terminal plansys2_terminal
```

A simple Planning Package: Running the example (cont.)

- In the PlanSys2 terminal create the problem

```
set instance leia robot
set instance entrance room
...
set instance chargingroom room

set predicate (connected entrance dinning)
set predicate (connected dinning entrance)
...
set predicate (battery_low leia)
set predicate (robot_at leia entrance)

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

PlanSys2 Commands:
{ commands.txt }

A simple Planning Package: Running the example (cont.)

- In the PlanSys2 terminal compute and run the plan

```
> get plan
...
> run
```

  The shell terminals show the current actions executed and respective level of completion.

- As soon as the plan execution finish, the PlanSys2 terminal command prompt will be available again to accept new commands.

Complete Example

- https://plansys2.github.io/tutorials/docs/simple_example.html

References I

[1] J. Benton, Amanda Jane Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI, 2012. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699. 19

[2] Pablo Bustos, Luis Jesús Manso, Antonio Bandera, Juan Pedro Bandera Rubio, Ismael García-Varea, and Jesus Martínez-Gómez. The CORTEX cognitive robotics architecture: Use cases. *Cogn. Syst. Res.*, 55:107–123, 2019. doi: 10.1016/j.cogsys.2019.01.003. URL https://doi.org/10.1016/j.cogsys.2019.01.003. 4

[3] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcís Palomeras, Natàlia Hurtós, and Marc Carreras. Rosplan: Planning in the robot operating system. In Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*, pages 333–341. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10619. 6

[4] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 42–49. AAAI, 2010. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1421. 19

References II

[5] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In Erwin Prassler, Johann Marius Zöllner, Rainer Bischoff, Wolfram Burgard, Robert Haschke, Martin Hägele, Gisbert Lawitzky, Bernhard Nebel, Paul-Gerhard Plöger, and Ulrich Reiser, editors, *Towards Service Robots for Everyday Environments - Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*, volume 76 of *Springer Tracts in Advanced Robotics*, pages 49–64. Springer, 2012. doi: 10.1007/978-3-642-25116-0_6. URL https://doi.org/10.1007/978-3-642-25116-0_6. 19

[6] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL https://doi.org/10.2200/S00513ED1V01Y201306AIM022.

[7] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[8] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB.

[9] Jörg Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artif. Intell. Res.*, 20:291–341, 2003. doi: 10.1613/jair.1144. URL https://doi.org/10.1613/jair.1144. 4

References III

[10] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 5420–5427. IEEE, 2014. doi: 10.1109/ICRA.2014.6907656. URL https://doi.org/10.1109/ICRA.2014.6907656. 5, 21

[11] Athanasios S. Polydoros, Bjarne Großmann, Francesco Rovida, Lazaros Nalpantidis, and Volker Krüger. Accurate and versatile automation of industrial kitting operations with skiros. In Lyuba Alboul, Dana D. Damian, and Jonathan M. Aitken, editors, *Towards Autonomous Robotic Systems - 17th Annual Conference, TAROS 2016, Sheffield, UK, June 26 - July 1, 2016, Proceedings*, volume 9716 of *Lecture Notes in Computer Science*, pages 255–268. Springer, 2016. doi: 10.1007/978-3-319-40379-3_26. URL https://doi.org/10.1007/978-3-319-40379-3_26. 5