# Architecting Robot-Arm Applications with Robotic Manipulation Platforms and Behaviour Trees

Andrew K. Lui

RAS/REF, RI, QUT

January 2024

Version 2

## Table of Contents

# Architecting Robot-Arm Applications with Robotic Manipulation Platforms and Behaviour Trees

*Andrew Lui (RAS/REF, RI, QUT)*

*Jan 2024*
*Version 2*

## 1. Introduction

Robotic manipulation platforms are a critical component for supporting the development of robot arm applications. The stated purpose is to ease the development work required in mapping complex arm manipulation tasks into physical joint-space movements. For instance, *Armer*, an in-house developed platform, offers an intuitive cartesian world space perspective to developers and hides away the complex robot joint-space kinematics. Programming interface for the end-effector moving to a specific position and orientation and vector servoing at a specific speed is available. It also supports prototyping designs through visualization and integration with *RViz*.

A discussion among the REF-RAS team started near the end of 2023 on whether to switch from *Armer* to *Moveit*, which is a popular open-source platform of the same kind. The concerned issues include ongoing maintenance and development cost and support for our team's development work.

This report aims to provide insights into the discussion from the wider perspective of architecting robot arm applications. The report is structured as follows.

- Section 2 begins with a requirement analysis of developing robot arm applications and based on the technical gaps identified motivates solutions for various problems. The section concludes with a list of key components of a model architecture for implementing complex tasks on physical robotic platforms, which include robotic manipulation platforms, behaviour trees, task manager, and general commander interface.
- Section 3 reviews the effectiveness of the technologies used in the current CGRAS robot arm manipulation application.
- Section 4 introduces the model architecture and provides a brief overview of the structure and the layers.
- Section 5 illustrates the design of several selected components through a reference implementation of the robot arm manipulation application for CGRAS.
- The appendices include one that explains the structure of the reference implementation software stack, and another one that explains the implementation of sensor-based collision detection based on the reference implementation.

## 2. Robot-Arm Manipulation Applications

The purpose of a robot-arm manipulation application is to execute high-level manipulation tasks and to realize these tasks as control commands of the robot control hardware interface. The application usually operates to the business logic of a *mission* and entertains instructions from a top-level application layer to execute, monitor, and cancel *tasks*. The application interface supports non-trivial clients such as a ROS action server, which features goal cancellation and feedback. Figure 1 highlights its main purpose and relation with other components.

The transformation of a task into control commands is complex. To simplify the transformation into more manageable steps, a task is first divided into a logical sequence of behaviours or actions, each of which represents a well-defined robot arm manipulation outcome. Finally, the outcome is translated into robot-specific low-level commands with techniques such as inverse kinematics.

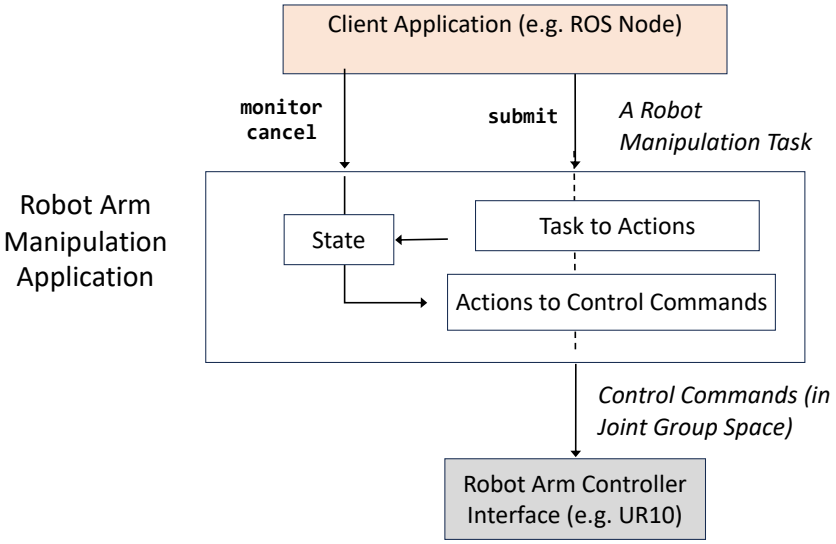


Figure 1. The major purpose of a robot arm manipulation application and its interaction with a client application.

The following lists a representative set of behaviours or actions. The abstraction level of behaviours is sufficiently sophisticated for describing part of a task and is sufficiently concrete for robot control.

- Move the end-effector to a pose in the cartesian space (i.e. the physical world).
    - The pose in the absolute frame or a relative frame to an object in the scene is acceptable.
    - Use a subset of the position, orientation, or part of the current pose in the target pose (e.g. rotation at the same position).
    - The path of movement can be linear.
    - The path of movement can be subject to constraints through setting the permissible orientation or physical space.
    - The pose can be a moving target.
- Move all parts (joints) of the robot arm to assume a pose (i.e. a shape).
- Move the end-effector along a vector in the cartesian space at a specified velocity.
- Manipulate the state of the end-effector (e.g. open or close the gripper or trigger a camera).

Task execution normally has a turn-around time from a few seconds to tens of seconds. Asynchronous interaction with robot-arm manipulation applications is preferrable so that the task can be monitored and cancelled. For managing tasks

through various intermediate and final outcomes, the applications maintain one or more states to inform the appropriate execution pathways.

### 2.1. Examples of Tasks and Behaviours

The representative set of behaviours are surprisingly sufficient as the building blocks for many robot manipulation tasks. The following shows the behaviours can be thoughtfully composed into some example tasks.

- Pick and place objects moving in a linear motion on a surface:
    - Move the end-effector to a *ready* pose in the world frame.
    - If a sensor detects an object on a surface, move to a *hunting* pose certain distance above the object (in the object's frame).
    - If it (the end-effector) is at the *hunting* pose, track the movement of the object and follow the object.
    - If it is following the object, move the vertical position (z-axis) downward.
    - If the object pick-up has failed, move to the *hunting* pose (and try again).
    - If the gripper has picked up the object, move to the *ready* pose.
    - If it is at the *ready* pose, move to the *dispose* pose (in the world frame).
    - If it is at the *dispose* pose, release the object.
- Restore to the pose for stowage (CGRAS project):
    - If the end-effector is in water, move to the *above-tile* pose (in the water tank's frame).
    - If it is at an *above-tile* pose, move to the *home* pose (in the world frame).
    - If it is at the *home* pose, move to the *stow* pose (in the world frame).
- Move and capture underwater images (CGRAS project):
    - If the end-effector is at the *stow* pose, move to the *home* pose.
    - If it is at the *home* pose, move to the *above-tile* pose of the first tile.
    - If it is in water, move to the *above-tile* pose of the current tile.
    - If the target tile requires a different orientation of the end-effector, move to the *above-tile* pose at the *middle planar position*.
    - If desired orientation of the end-effector is wrong and at the middle, rotate the end-effector.
    - If the desired orientation is correct, move to the *above-tile* pose of the target tile.
    - If it is at the *above-tile* pose of the target tile and the end-effector not tilted, rotate the end-effector to an oblique angle.
    - If it is at the *above-tile* pose of the target tile and the end-effector tilted, move into the water.
    - If it is in the water and the target pose is the current tile, move to the target pose.
- Move and calibrate the pose of the water tank (CGRAS project):
    - If the end-effector is at the *stow* pose, move to the *home* pose.
    - If it is in water, move to the *above-tile* pose of the current tile.
    - If it is at an *above-tile* pose, move to the *home* pose.
    - If it is at the *home* pose, calibrate.
    - If calibrate has failed, broadcast the abort signal.

Note that the last three tasks are actual examples derived from the CGRAS robot control application.

## 2.2. Support for Task Execution

The examples in section 2.1 serves the purpose of understanding the form of behaviours and the way they are executed in a task. The following essential characteristics of task execution in robot-arm manipulation applications are derived.

- Stateful execution. The behaviours/actions in a sequence are considered, executed, or skipped one at a time and the next behaviour to execute depends on the current behaviour and other conditions.
- Conditional execution. Generally, behaviours are relevant to a task only under certain conditions, such as whether the current pose is what is expected, or the sensor has found something desirable. All the behaviours listed in the above examples have an attached condition.
- Fallback execution. Execute alternative behaviours when a higher-priority event such as failure occurs, for example, the failure to calibrate can prompt another behaviour to broadcast an abort signal.
- Abortable execution. Task execution allows cancellation, which terminates the running behaviour.
- Asynchronous execution. Task and behaviour execution is non-blocking, enabling task pre-emption and execution of other processes.

Figure 2 below depicts the key components in robot arm manipulation applications that possess the essential characteristics. The task manager provides the application the capacity to manage the lifecycle of a task and to support abortable and asynchronous execution. It delegates task execution to the behaviour tree, which is a stateful engine to activate the appropriate behaviours under various conditions. The behaviours then delegate the physical actions to the robotic manipulation platform. State management in all three components as the activities involved are anything but atomic and error free.

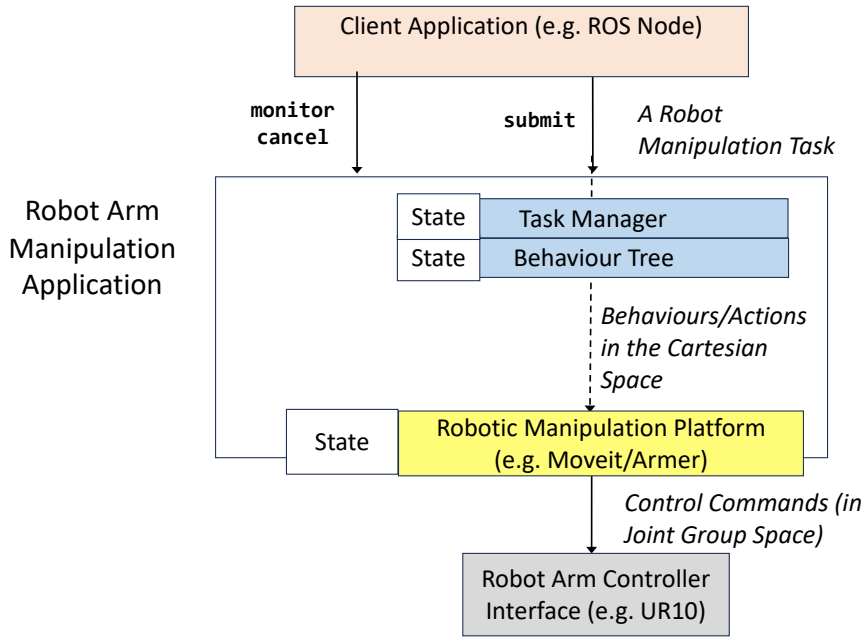


Figure 2. Motivating the need for task manager, behaviour tree, and the arm manipulation platform in robot arm manipulation applications.

## 2.3. Behaviour Trees and Finite State Machines

Behaviour trees and finite state machines (FSM) are effective models for handling stateful execution. The use of behaviour trees for robot-arm manipulation task execution is common because it can model action sequences very

effectively in coding, and the resulting application is resilience to change. The following table compares the suitability of behaviour trees and FSM in the support for task execution.

| | **Behaviour trees** | **Finite state machines** |
|---|---|---|
| Stateful Execution | Yes<br>• Supports sequential execution and no explicit transition links.<br>• Easy to maintain and make changes. | Yes<br>• Supports arbitrary execution pathways.<br>• Difficult to maintain and make changes if complex transitions are present. |
| Conditional Execution | No direct support in PyTrees 2.2.3<br>• Use the Task Guarded subtree of behaviours pattern, or,<br>• Program the behaviour to become conditional. | Support Implicitly conditional execution (the prior conditions before reaching the state), or,<br>• Program the in-state actions to become conditional. |
| Fallback Execution | Supported in PyTrees 2.2.3: the Selector pattern | No native support |
| Abortable Execution | Supported in PyTrees 2.2.3: the Eternal Guard pattern | No native support |
| Asynchronous Execution | Supported in PyTrees 2.2.3: the memory feature in the Composite classes | No native support |

The tree structure of behaviour trees enables a task, which is a logical sequence of behaviours, organized as a sub-tree. Reusing a sub-tree in other parts of the behaviour tree or that of another application is straightforward – just clone the sub-tree. Similarly, copy-and-paste a sub-sequence of behaviours under other sub-trees or tasks is easy. The implicit-transition feature of behaviour trees facilitates the reuse of the whole of a part of a sub-tree. The following figure illustrates how a task, represented as a sub-tree, is reusable across the same or a different application.

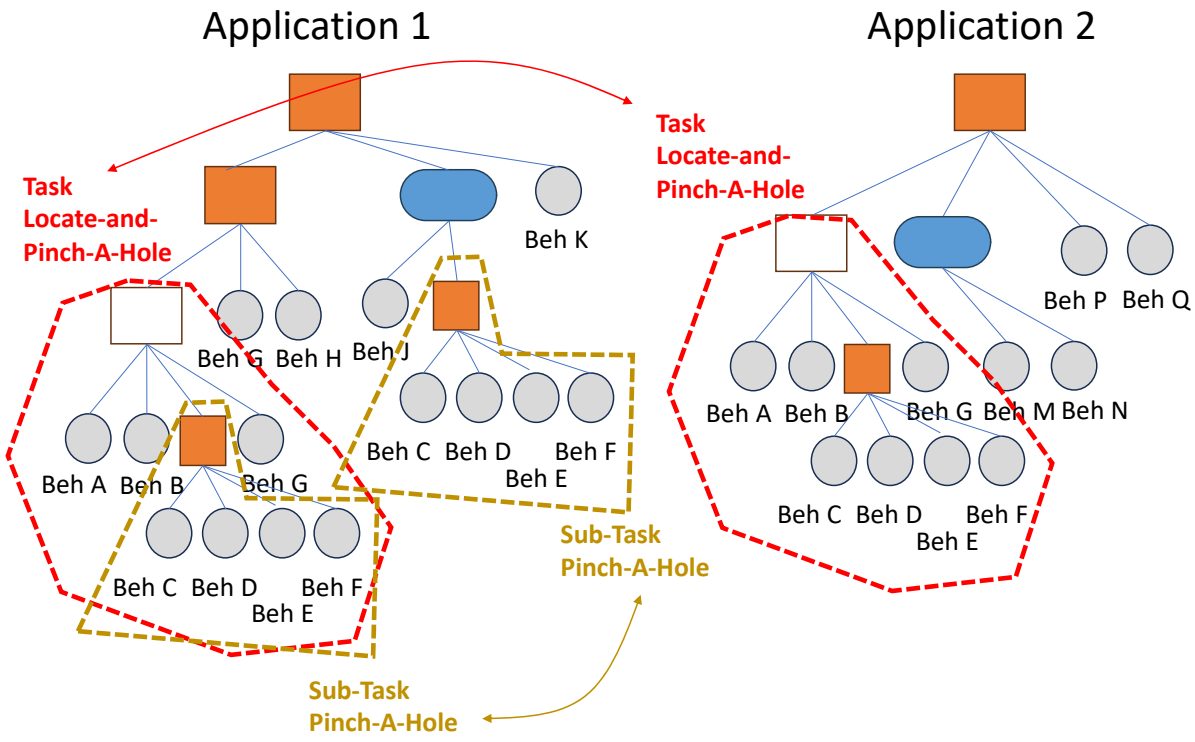


Figure 3. Reuse of tasks and sub-tasks within and across applications based on behaviour trees.

## 2.4. Separation of Conditions and Actions in Behaviours

The above figure illustrates that the reuse of tasks and sub-task can reduce the number of behaviour classes. Reusability at the behaviour level is also exploitable due to redundancies in the actions and conditions. From the following two behaviours extracted from section 2.1, note there are two different trigger conditions attached to the same action.

- If it is at the *above-tile* pose, move to the *home* pose (in the world frame).
- If the end-effector is at the *stow* pose, move to the *home* pose.

Note that the action part of the behaviour is redundant. Separating the trigger condition from actions can simplify the specification. The condition part becomes a pluggable unit of behaviour classes. One core behaviour of "move to the home pose" can turn into multiple instances of the following conditions.

- *If the end-effector is at the above-tile pose*.
- *If the end-effector is at the stow pose*, and
- Other conditions such as *If the sensor is activated*.

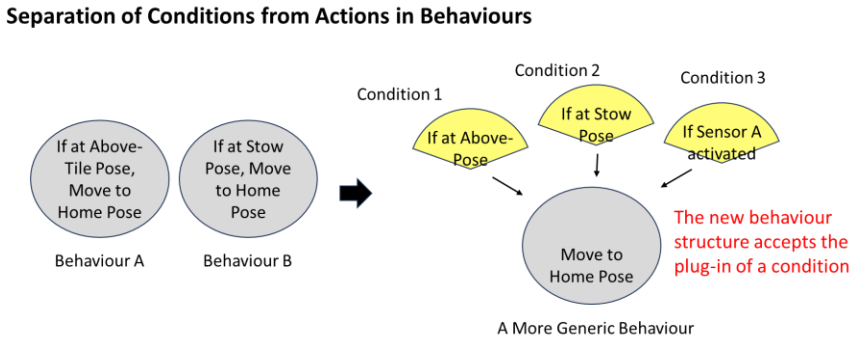


Figure 4. Conditions are separated from the action and turned into plug-in condition functions for behaviours.

Conditions have three semantic roles in behaviours:

- Determine if an action is relevant (e.g. move the arm if an object is visible).
- Determine if the outcome of the action is deemed success or attained (e.g. move to home pose only if not already here).
- Determine if a long-lasting action should be aborted or otherwise (e.g. move the arm if the object is still being tracked).

Separation of concerns is a powerful principle for simplifying software. The following figure illustrates how the number of behaviour classes can reduce significantly by pulling out the condition part as a pluggable unit. As the number of probable actions of an end-effector are mostly related to movement, the resulting behaviour classes are likely to be the primitives that are highly representative across various applications.

Condition units are usually simple to implement, as demonstrated in the examples above. The number of condition units can be reduced by breaking down composite conditions.

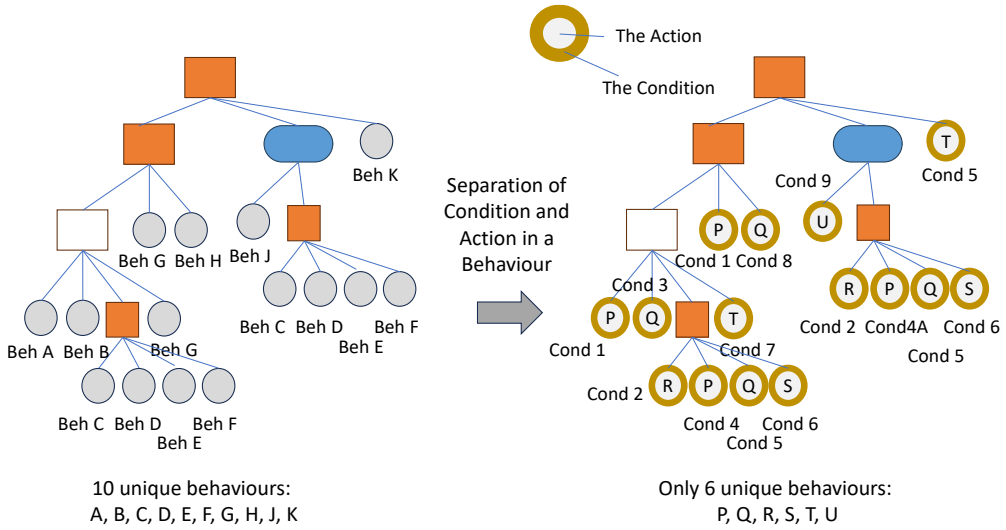


Figure 5. Significant reduction of the number of behaviour classes required due to de-coupling of conditions and actions.

## 2.5. Support for Passing Function as a Logical Pose Parameter in Behaviour Classes

Parameterization can further facilitate the reusability of behaviours and conditions. The following two behaviours, for example, can be combined into one with the parameter *logical_pose*, which determines the target of the move action.

- Move the end-effector to the home pose.
- Move the end-effector to the stow pose.

Similarly, conditions can be parameterized. The following two conditions, for example, can be combined into one with the parameter *time_lapse.*

- If no target is found after 10 seconds.
- If no target is found after 20 seconds.

Instances of behaviours are created at the behaviour tree building stage and the parameters are constants and determined at compile-time (e.g. the home and the stow poses). Tick-tock-time binding of parameters is significantly more useful in representing sophisticated behaviours, such as following a moving object or moving to a target according to input. To achieve this, the method of passing function as a parameter is effective. Applying this on the *logical_pose* parameter is particularly powerful.

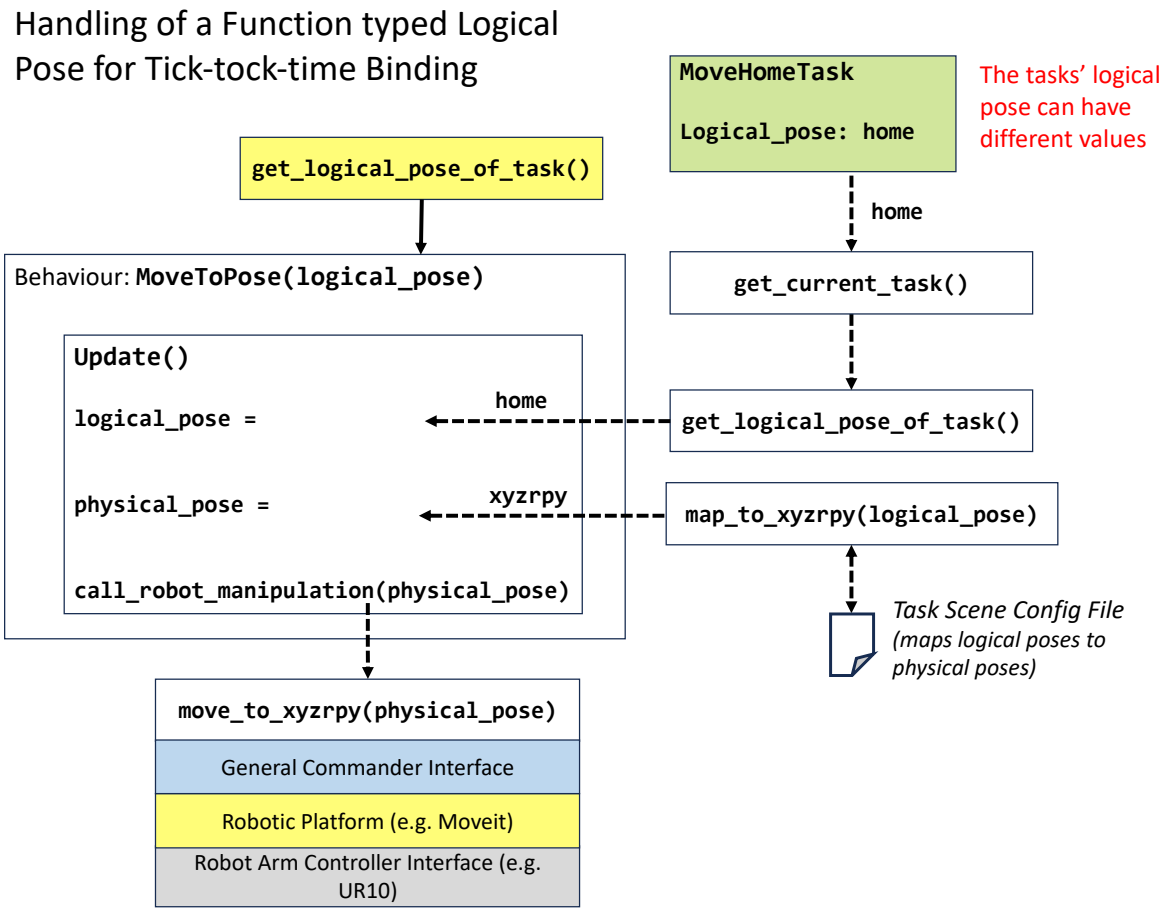


Figure 6. An illustration of how tick-tock time binding of logical poses works using function in behaviour parameters.

## 2.6. Support for Task Management

Robot-arm manipulation applications is client-facing. It will receive commands of task execution from a client application. To enhance functionality and reliability, the application should include the following asynchronous task management functions:

- Task submission and status tracking.
- Task cancellation.
- Thread-safe task management. To avoid race conditions during simultaneous execution of multiple tasks.

Task management is lacking in existing behaviour trees frameworks. It is necessary to program the following two additional components.

- Lifecycle management of tasks through pre-defined Behaviour classes (nodes) and service points for inspection. This component is project independent and reusable.
- Application specific task handling. This easy-to-develop component is project dependent.

## 2.7. Motivation for a Robotic Manipulation Platform

The role of robotic manipulation platforms is to provide service points for commanding robots with poses specified in the cartesian (world) space and the underlying core maps the commands to joint-space poses.

The following table compares the fit-for-purpose of two available platforms for the general robot arm manipulation actions. Moveit appears to support a wider range of actions than Armer.

| | Armer | Moveit (version 1) |
|---|---|---|
| **Move End-Effector to Pose** | | |
| Cartesian Space and Quaternion | Yes | Yes |
| Cartesian Space and Euler's Angles | No<br>Needs conversion with tools | Yes<br>Through the built-in `conversions` module |
| Absolute frame | Yes | Yes |
| Relative frame (to a scene object) | No<br>Needs transformation | Yes |
| Linear motion | No | Yes<br>Through the path planner and the trajectory executer |
| Path planning | Yes | Yes<br>OMPL path planners configurable |
| Constrained path planning | No | Yes<br>Supports path constraints based on permissible range of orientations, positions, and joint values. |
| Collision avoidance in path planning | Yes (only in the latest version) | Yes |
| **Move All Joints to Pose** | | |
| Support named Pose | Yes<br>Config file and run-time | Yes<br>Config file and run-time |
| **Servoing** | | |
| Linear and angular velocities | Yes | Yes |
| **Manipulate End-Effector** | | |
| Grasp Planning and Execution | No | Yes |
| **Robot Model Agnostic** | | |
| Specified through config files | Yes | Yes |
| # of validated arm models supported | 4 | 15[1] |

## 2.8. Motivation for a General Commander Interface

The task manager supports abortable and asynchronous task execution. The implementation requires the support of the underlying robotic manipulation platform. The purpose of a general commander interface is to abstract away the actual robotic manipulation platform, and in doing so, to provide the mediation so that the command execution is abortable, trackable, and asynchronous. It also enhances the API for interacting with the robotic manipulation platform, entertaining the following actions.

- Movement relative to the current pose (as opposed to the absolute pose in the world frame).
    - Displacement.
    - Move to a position and keep the same orientation.
    - Rotate to an orientation and keep the same position.
    - Linear motion.

[1] https://moveit.ros.org/robots/

- Movement in another frame of reference (as opposed to the world frame only).

The general command interface has a core that is specific to the underlying robot arm manipulation platform in the software stack. The interface provides a technology agnostic way to interact with different robot models through different robotic manipulation platforms (i.e., whether it is Moveit 1, Armer, or Moveit 2). As shown in the figure below, the upper layer software can remain unchanged.

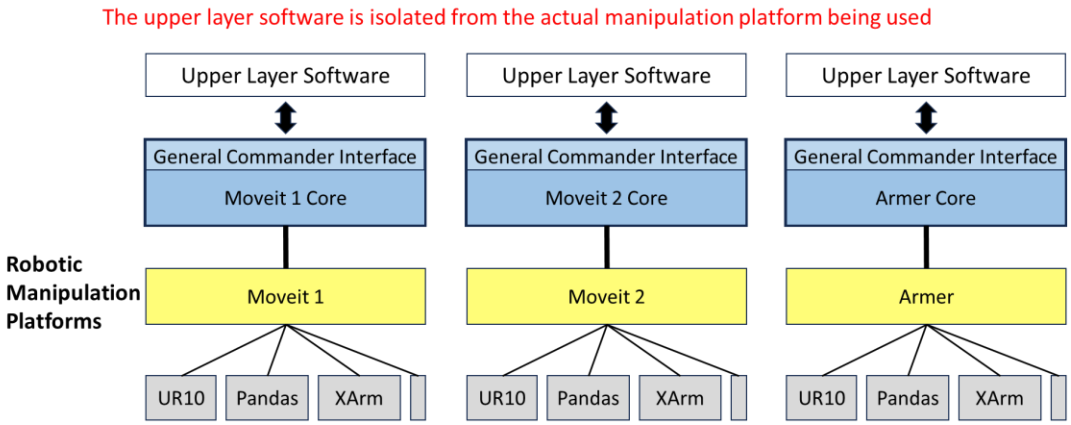


Figure 6. Layered architecture to hide away the specific underlying platforms and isolate the upper layer software from changes.

## 2.9 Layered Scenes and Poses Framework

The scene defines the environment in which the robot is operating. In robot arm manipulation applications, there are three abstraction levels of scene.

- The task or application scene. The behaviour trees and the behaviour classes operate on this level and deal with either locational concepts (e.g., the surface of the conveyor belt or a disposal bin) or logical poses (e.g. home pose and ready pose).
- The physical scene. The general commander interface and the robot arm manipulation interface operate on this level and deal with physical poses (e.g. xyzrpy or xyzqqqq).
- The robot configuration. The robot joint-group controller operates on this level and deals with joint-space poses.

The layered architecture delineates a structured realization of task concepts into physical robot arm manipulations. It helps keep the codebase stable and localizes code revision and troubleshooting. The design decouples task and behaviour implementation from changes in the scene, task parameters and robot parameters. A configuration file holds all the parameters relevant to the task and mapping between the task scene and the physical scene.

The following figure shows examples of poses represented at different abstraction levels.

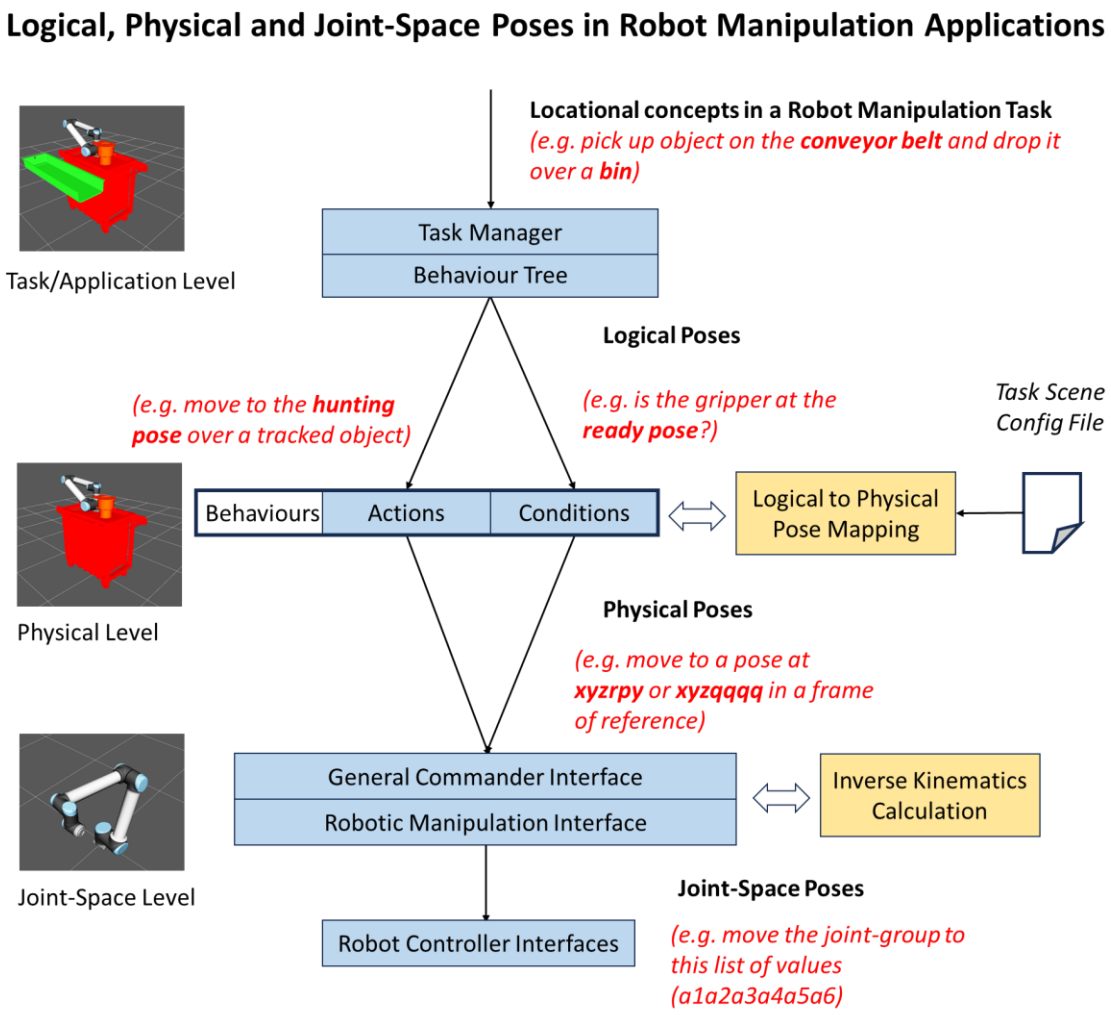


Figure 7. An illustration of the mapping between logical poses, physical poses, and joint-space poses.

## 3. Review of the Technologies adopted in the Current CGRAS Implementation

The REF-RAS group were engaged in the CGRAS project in 2023, and one work package was to develop a robot-arm manipulation application for capturing underwater images. The camera is inside a water-proof end-effector and mounted on a UR-10 robot arm. The application can execute three different tasks.

- Restore the robot arm to the stow pose for stowage and to the home pose for getting ready.
- Move the end-effector and capture underwater images at a logical pose.
- Move the end-effector to the home pose and calibrate the pose of the water tank.

The current version of the application is supporting a pilot data-collection experiment in the client's facility. The deployment is successful, which has proven its reliability and effectiveness.

The focus of this review is the effectiveness of the following enabling components during the development phase.

- Armer as the robot arm manipulation interface.
    - The inhouse-developed software is the outcome of an ongoing informal project. New features and bug fixes happen in bursts.
    - The existing Armer-supported project software code is a good mining source of reusable parts.
    - Armer is the supporting framework for the apriltag calibration routine, which is a critical existing component.
- ROSTrees + PyTrees 0.7.6 as the behaviour trees framework.
    - ROSTrees is another inhouse-developed software of which the goal is to enhance PyTrees with ROS capability and pose format conversion facilities.
    - There is a tight coupling between ROSTrees and PyTrees 0.7.6, which is a severely outdated version. The most recent release is version 2.2.3. The original developer has departed, leaving ROSTrees in a neglect state.

The following table compares the support for task execution from the two versions of PyTrees. The enforced use of an outdated version significantly widened the technical gap and increased the work of the developer.

| | PyTrees 2.2.3 | PyTrees 0.7.6 |
|---|---|---|
| Stateful Execution | Yes | Yes |
| Conditional Execution | No direct support | No direct support |
| Fallback Execution | Yes, through the Selector class | Yes, through the Selector class |
| Abortable Execution | Yes, through the Eternal Guard pattern | No direct support |
| Asynchronous Execution | Yes, through the memory feature in the Composite classes | No direct support |

There are at least three problematic issues with ROSTrees.

- The descriptions of ROSTrees have advertised its design intention to work with PyTrees, but the software library has introduced a programming approach and behaviour tree mechanism that is different from PyTrees. For example, the new entity Leaf as the behaviour trees' leaf nodes is entirely based on another lifecycle and signalling mechanism. The resulting behaviour tree is a mix of two programming approaches.
- A main feature of ROSTrees is the four built-in ROS Leaf classes for interfacing with ROS communication interfaces. The expose of ROS functionality on the behaviour tree level reduces programming effort. The interface has prescribed a particular protocol of interaction that represents only a portion of use cases. The

limitation is most obvious in the ActionLeaf class which has excluded pre-emption and feedback handling. The Subscriber Leaf and the Publisher Leaf are probably most useful as the one-way topic communication is straightforward anyway. The other two ROS Leaf classes are useful for the simple use cases.

- The different interaction protocol and programming approach induces additional cognitive loading on the developers.

Any developer must possess great skills, persistence, and perseverance to have successfully built robot arm manipulation applications based on Armer, PyTrees 0.7.6 and ROSTrees.

## 4. A Model Architecture

Software development is hard. Writing programs to do sophisticated tasks is harder. A good architecture illustrates an effective way to dissect the tasks and to organize the conceptual components. It enables pre-crafted software stacks to integrate with application specific components. It can reduce development and extension effort significantly.

The following table lists the major components in a model architecture for building robot arm manipulation applications. The problem analysis in the previous sections has already justified the need for these components.

| *Components* | *Pre-Existing* | *Post-Development Reuse Potential* |
|---|---|---|
| Robotic Manipulation Platform: Moveit + ROS + TF | Open source | |
| Scene Setup Assistant: Moveit | Open source | |
| Behaviour Trees: PyTrees 2.2.3 | Open source | |
| | | |
| General Commander Interface | No | Applicable to all applications |
| The Behaviour Tree for the Application | No | Similar Applications |
| The Base Conditional Behaviour Class | No | Applicable to all applications |
| The Base Commander Behaviour Class | No | (to be kept in the Behaviour Library) |
| Application Specific Conditional Behaviour Classes | No | Similar Applications |
| The Condition Units for the Behaviours | No | Similar Applications |
| Task Manager | No | The Lifecycle Management Part |
| Task Scene Model Interface | No | Yes (add new logical entities if required) |
| Task Scene Config File | No | No (except the format) |

The previous sections have discussed all the components except the following two:

- The base *Conditional Behaviour* class. The class is a specialization of the *Behaviour* class in PyTrees. It models the notion of separation of conditions from actions in behaviours and supports the development of application specific behaviours that exploit the notion.
- The base Commander Behaviour class. The class is a specialization of the *Conditional Behaviour* class. It has built-in support for managing the lifecycle of command execution in the background. It enables the development of application specific behaviours that use the general commander interface.

The below figure shows the layered structural view and the component view of the model architecture.

**Architecture for Robot Arm Manipulation Applications**
based on Moveit Commander (Moveit Noetic) and PyTrees (2.2.3)

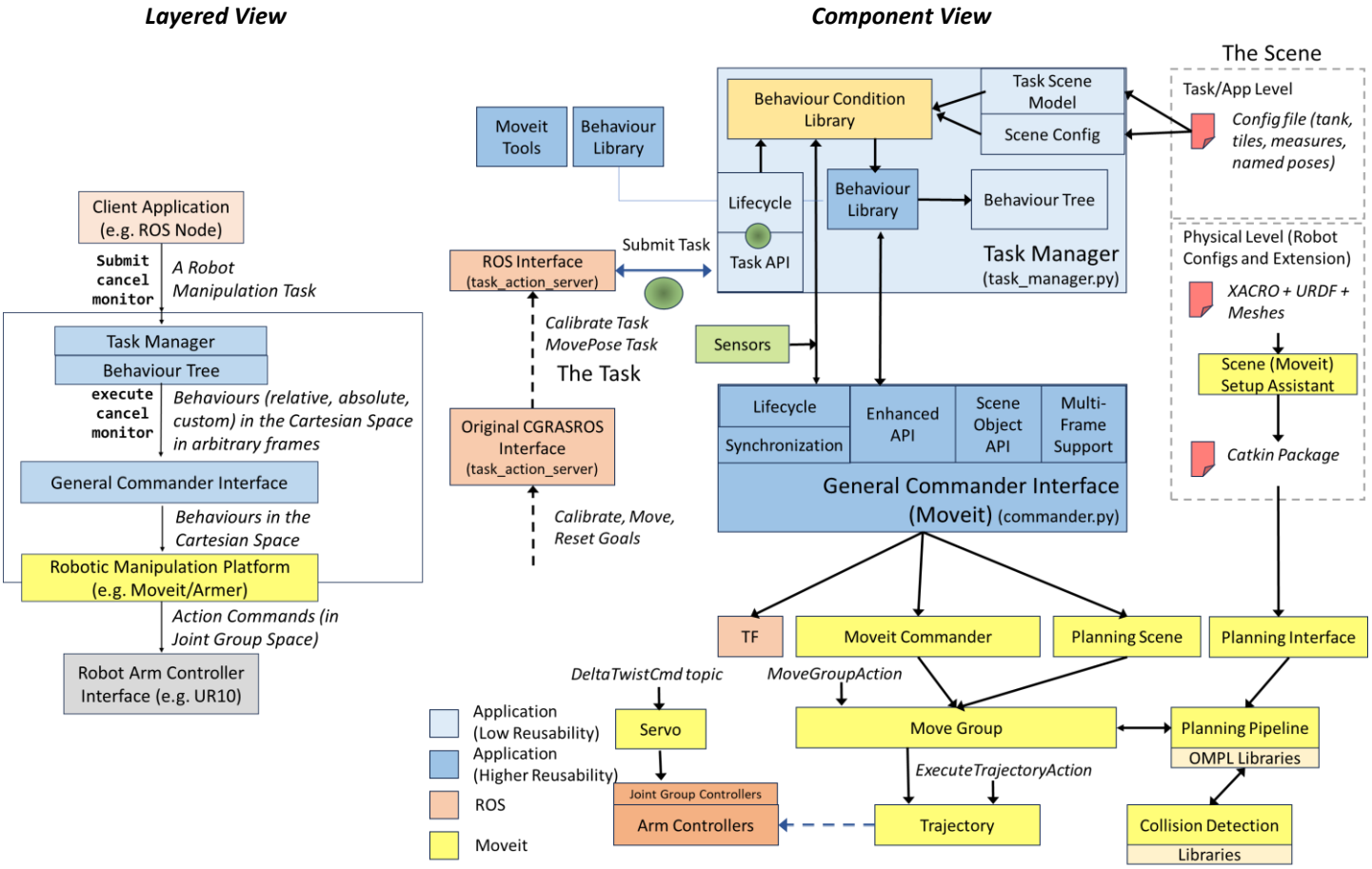


Figure 8. The model architecture for robot arm manipulation applications

The component view is an expansion of the layered view, including the constituents in the components and data/command flows between them. Revisit the previous sections for explanations.

The following discusses the key features of the model architecture.

- Separation of concerns. This most important feature ensures a logical organization of the components and decoupling of concepts that increases complexity, structural rigidity, and code redundancy. Examples include:
    - Conditions and actions in behaviours.
    - Behaviour trees and ROS.
    - Behaviour trees and robotic manipulation platform.
    - Behaviours and physical scene/poses.
- ROS agnostic. This feature ensures the portability of the application to various ROS versions and the reusability of the part of the application in a non-ROS environment.
- Robotic manipulation platform/robot model agnostic. This feature ensures the portability of the application to Armer, Moveit 1, newer versions of Moveit, and other platforms.
- Resilience to changes in the scene and the task. The loosely coupled architecture confines most changes to just one to two components. Examples:
    - Changes to the physical scene: changing the task scene configuration file.

  - Changes to the conditions of a behaviour: changing the condition argument in the behaviour tree building routine, and if needed, developing a new condition unit, or adapting a pre-existing one.
  - Adding or removing a behaviour in a sequence of behaviours: changing the behaviour tree building routine, and if needed, developing a new behaviour, or adapting a pre-existing one.
  - Changing a behaviour: changing the definition of the relevant behaviour class, and if needed, define new logical pose and the mapping in the task scene config file.
- Reusable primitives. Although the behaviour trees are application specific, a sub-tree or a sequence that represents a general logical task is useful across applications. The decoupling of conditions and actions promotes the development of increasingly primitive entities.

# 5. The Reference Implementation and Use Patterns

This section describes and discusses the reference implementation for the model architecture. The tangible outcome is a re-implementation of the robot arm manipulation application for the 2023 CGRAS project. The source code for the reference implementation will be available in a GitHub repository. This section will highlight selected aspects of the implementation. The purpose is to illustrate concepts of the architecture and to demonstrate the viability at the coding level.

## 5.1. States of the General Commander Interface

One goal of the general commander interface is to resolve the inadequacy of robotic manipulation platforms (i.e. Moveit or Armer) in lacking state management. The states of the general commander interface model the execution phases of control commands and regulate command cancellation and error handling. Figure 9 explains the transition of states in the general commander interface.

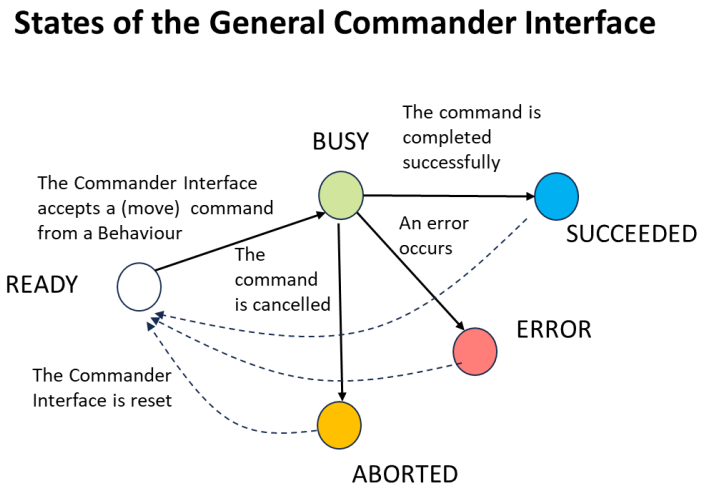


Figure 9. The lifecycle of the general commander interface

The states support thread-safe access, ensuring one command is active in the `BUSY` state. The eventual outcome is one of the three states: `SUCCEEDED`, `ERROR`, and `ABORTED`. A client application of the interface can then deal with the different outcomes appropriately.

The interface supports move control commands through both synchronous calls and asynchronous calls. The following is an example of an asynchronous call, specified through the **`wait`** parameter.

```
arm_commander = self.arm_commander

arm_commander.move_to_named_pose('stow', wait=False)

while True:
  the_state = arm_commander.get_commander_state()
  if the_state not in [GeneralCommanderStates.BUSY]:
    break
  rospy.sleep(0.1)

rospy.loginfo(f'Final state: {the_state} {the_state.message}')
rospy.loginfo(f'Final result: {arm_commander.get_latest_moveit_feedback()}')
```

### 5.2. The Enhanced API on the General Commander Interface

In addition to providing thread-safe access and lifecycle management to the robotic manipulation platform, enhanced API is another feature of the general commander interface. The enhanced API aims to expand the use cases of the original API offered by Moveit and potentially other platforms, and to reduce coding effort.

- Enhance the flexibility of pose specification:
  - Specify a component of a pose and assume the current values in the others (i.e. the default is the current value).
  - Specify a displacement from the current pose.
  - Specify pose in another frame of reference.
  - Accept different pose formats.
- Support both synchronous call and asynchronous call:
  - To manage an asynchronous call, functions for checking the state, aborting the move are provided.

| **Selected examples of the enhanced API calls: move commands** | |
|---|---|
| `def move_displacement(self, dx=0.0, dy=0.0, dz=0.0, wait=True)` | |
| | Move a displacement in cartesian motion.<br>`wait`: `True` (synchronous blocking call) or `False` (asynchronous) |
| `def move_to_position(self, x:float=None, y:float=None, z:float=None, cartesian=False, reference_frame:str=None, wait=True)` | |
| | Move to a position and the current orientation unchanged.<br>The default for `x`, `y`, `z` are the current values.<br>`cartesian`: Use cartesian motion or use the path planner.<br>`reference_frame`: The default is the current planning frame (usually the world frame). |
| `def rotate_to_orientation(self, roll:float=None, pitch:float=None, yaw:float=None, reference_frame:str=None, wait=True)` | |
| | Rotate to an orientation and the current position unchanged.<br>The default for `roll`, `pitch`, `yaw` are the current values. |
| `def move_to_pose(self, target_pose, reference_frame:str=None, wait=True)` | |
| | Move to a pose.<br>`target_pose`: Supported formats include a list of 6 floats (xyzrpy), 7 floats (xyzq), Pose, or PoseStamped. |

| **Selected examples of the enhanced API calls: query** | |
|---|---|
| `def current_joint_positons_as_list(self, print=False) -> list` | |
| | Returns the current joint positions as a list.<br>`print`: Also output the values to the log stream. |
| `def pose_in_frame_as_xyzq(self, link_name:str=None, reference_frame:str=None, ros_time:rospy.Time=None, print=False) -> list` | |
| | Returns the current pose as a list of 7 floats (xyzqqqq)<br>`link_name`: The pose of which is returned. The default is the end-effector.<br>`reference_frame`: The reference frame of the pose. The default is the current planning frame. |
| `def pose_in_frame_as_xyzrpy(self, link_name:str=None, reference_frame:str=None, ros_time:rospy.Time=None, print=False) -> list` | |
| | Returns the current pose as a list of 6 floats (xyzrpy) |
| `def pose_in_frame(self, link_name:str=None, reference_frame:str=None, ros_time:rospy.Time=None) -> PoseStamped` | |
| | Returns the current pose as a PoseStamped |

The general commander interface has also integrated some functions of scene management for path planning. The following lists the functions for customizing the scene programmatically.

| **Selected examples of the enhanced API calls: scene management** | |
|---|---|
| `def reset_world(self):` | |
| | Remove all scene objects that have been added through the commander |
| `def add_object_to_scene(self, object_name:str, model_file:str, object_scale:list, xyz:list, rpy:list):` | |
| | Add object to the scene and publish TF messages for the object<br>`object_name`: The name of the object<br>`model_file`: The path to the mesh file of the object<br>`object_scale`: The scaling factor in the x, y, z dimension of the object<br>`xyz`: The position of the object in the scene<br>`rpy`: The orientation of the object in the scene |
| `def list_object_names(self) -> list` | |
| | Returns the list of object names added through the scene management |
| `def get_object_pose(self, object_name:str, print=False) -> Pose:` | |
| | Returns the pose of an object<br>`object_name`: The name of the object |

A critical function missing in this section is the support for dynamic object modelling. In the future functions will be added to support a model of object movement.

### 5.3. Integration of the General Commander Interface with Behaviours

In typical robotic manipulation applications, the behaviours are, most of the time, the caller of general commander interface. The inevitable coupling between behaviour classes and the general commander interface prompts the design of a specialized base class for behaviours.

The class **`CommanderBehaviour`** contains the common logic of a behaviour in dealing with the states of the general commander interface. The following shows the structure of the **`update`** function of **`CommanderBehaviour`**. Subclasses of `CommanderBehaviour` will implement the three functions highlighted in the code snippet.

```
class CommanderBehaviour(Behaviour):
...
    def update(self):
        self.commander_state = self.arm_commander.get_commander_state()
        if self.commander_state == GeneralCommanderStates.READY:
             return self.update_when_ready()
        elif self.commander_state == GeneralCommanderStates.BUSY:
            return self.update_when_busy()
        elif self.commander_state == GeneralCommanderStates.SUCCEEDED:
            self.tidy_up()
            return Status.SUCCESS
        elif self.commander_state == GeneralCommanderStates.ABORTED:
            rospy.logerr(f'Task aborted: {self.arm_commander.get_latest_moveit_feedback()}')
            self.tidy_up()
            return Status.FAILURE
        elif self.commander_state == GeneralCommanderStates.ERROR:
            rospy.logerr(f'Task error: {self.arm_commander.get_latest_moveit_feedback()}')
            self.tidy_up()
            return Status.FAILURE
        else:
            return Status.SUCCESS
```

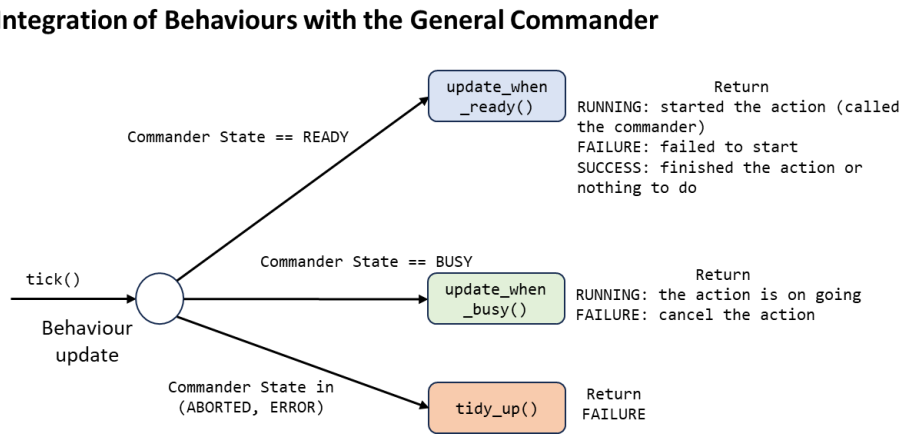


Figure 10. A framework for managing the states of the general commander interface in the **`CommanderBehaviour`** class.

The function **`update_when_ready`** is called when the general commander is READY, which means that generally the function makes a move call (i.e. move the robot). If the call is successful, then the commander will then switch to BUSY. The function **`update_when_busy`** is the place where the logic for managing the move call can be included. The function **`tidy_up`** is called when the commander has completed the move call with a success outcome or otherwise.

## 5.4. The behaviour tree and their behaviours in CGRAS

The figure below helps the readers appreciate the overall structure of the behaviour tree defined for the CGRAS robot-arm manipulation. The node labels are irrelevant to the discussion. The orange nodes are `Sequence`, the blue node is `Selector`, the white ones are `decorators`, and the grey nodes are `behaviours`.

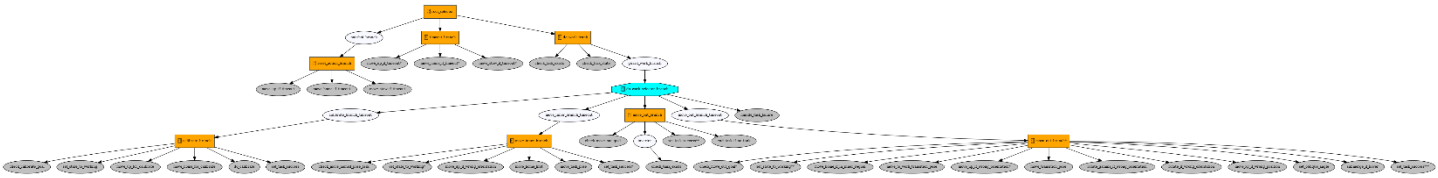

Figure 11. An impression of the size of the behaviour tree for the CGRAS application

The following tables summarize the behaviour classes and the condition units used in the above tree and the frequencies of these classes and units found in the tree.

| *Class Names* | # Instances | Remarks |
|---|---|---|
| **Base Class: `ConditionalCommanderBehaviour`** | | |
| `DoMoveNamedPose(named_pose)` | 8 | named_pose accepts a function or string |
| `DoMoveUpDown(z_level_pose)` | 5 | |
| `DoMovePoseAtTank(logical_pose, z_level_pose)` | 1 | logical_pose accepts a function or list |
| `DoMoveXYAtTankFixedYOrientation(logical_pose, z_level_pose)` | 3 | logical_pose accepts a function or list |
| `DoRotateAtTankFixedPosition(rotation_pose)` | 3 | rotation_pose accepts a function or list |
| **Base Class: `Behaviour`** | | |
| `DoCalibrate` | 1 | |

| Primitive Predicates for Conditions | # Usages | # Lines | Remarks |
|---|---|---|---|
| `task_is_timeout(duration)` | 3 | 1 | duration is a parameter |
| `in_a_region(logical_region)` | 9 | 2 | logical region is a parameter |
| `at_hover_level()` | 7 | 2 | |
| `below_hover_level()` | 5 | 2 | |
| `wrong_orientation()` | 4 | 3 | |
| `wrong_xy_at_tank()` | 1 | 3 | |
| `at_angle(rotation_pose)` | 2 | 7 | rotation pose is a parameter |
| `at_a_named_pose(named_pose)` | 3 | 3 | named pose is a parameter |

The architecture enables building a non-trivial behaviour tree with few behaviour classes and primitive predicates. All predicate definitions are between 2 to 3 lines of code that solicits the support of the general commander interface and the helper functions. Creating new predicates is usually a simple task.

### 5.5. Readability and maintainability of behaviour tree building code.

The following shows the code snippet that builds a sub-tree of the behaviour tree for the *move to named pose* task.

```
py_trees.composites.Sequence('move_named_pose_branch', memory=True, children = [
    py_trees.behaviours.CheckBlackboardVariableValue(name='check_move_named_pose_task',
          check=ComparisonExpression(variable='task.name', value=MoveNamedPoseTask, operator=operator.eq)),

    py_trees.behaviours.SetBlackboardVariable(name='set_state_to_working', variable_name='task.state',
                                              variable_value=TaskStates.WORKING, overwrite=True),

    DoMoveUpDown('move_up_if_in_water', condition_fn=[{'_fn': self.in_a_region, 'logical_region': 'work'},
                                                      self.below_hover_level],
                                    self.arm_commander, z_level_pose='hover'),
    DoMoveNamedPose('move_home_first', condition_fn= [{'_fn': self.in_a_region, 'logical_region': 'work'},
                                                      self.at_hover_level],
                                    self.arm_commander, named_pose='home'),
    DoMoveNamedPose('move_task_pose', condition_fn=True,
                                    self.arm_commander, named_pose=self.query_logical_pose_of_task),

    py_trees.behaviours.SetBlackboardVariable(name='set_task_success', variable_name='task.state',
                                              variable_value=TaskStates.SUCCEEDED, overwrite=True),
])
```

The application behaviour classes are in blue, and the condition functions are in orange. Note the two instances of the same behaviour class are created with two different conditions: the first is a composite condition of two predicates, and the second is the constant True. The semantics of the task is therefore visible, as each behaviour constructor call can be read from left to right as follows:

- The behaviour class name that describes the general action can be carried out.
- The behaviour name that gives away the specific role of the behaviour.
- The condition (refer to the next section for acceptable forms of condition).
- (For subclasses of **`ConditionalCommanderBehaviour`**) the commander to use (useful in multi-robot or multi-manipulator applications).
- The keyword parameters of the behaviour class. The actual ones depend on the individual behaviour classes.
    - In the above example, **`z_level_pose`** is a parameter of **`DoMoveUpDown`** and **`named_pose`** is a parameter of **`DoMoveNamedPose`**. Note that many of these parameters accept both a constant (e.g. **`'home'`**) and a function (e.g. **`self.query_logical_pose_of_task`**).

A tabulated representation of the same behaviour sub-tree is given below.

| Node Names | Behaviour Classes | Conditions | Remarks |
|---|---|---|---|
| check_move_named_pose_task | CheckBlackboardVariableValue | Nil | Check task.name==MoveNamedPostTask |
| set_state_to_working | SetBlackboardVariable | Nil | Set task.state = TaskStates.WORKING |
| move_up_if_in_water | DoMoveUpDown | in_a_region('work') and below_hover_level() | Move to z_level_pose='hover' |
| move_home_first | DoMoveNamedPose | in_a_region('work') and at_hover_level() | Move to named_pose='home' |
| move_task_pose | DoMoveNamedPose | True | Move to named_pose= query_logical_pose_of_task |
| set_task_success | SetBlackboardVariable | Nil | Set task.state = TaskStates.SUCCEEDED |

The parameters and conditions of behaviours are logically arranged to ease the effort in making changes. Adding a new behaviour to the task is also straightforward. The following shows how to ensure the end-effector is back to the *alpha* rotation pose before moving back to the home named pose. The coding effort is minimal as all the needed primitives already exist.

```
    ...
    DoMoveUpDown('move_up_if_in_water', condition_fn = [
                                                {'_fn': self.in_a_region, 'logical_region': 'work'},
                                                 self.below_hover_level],
                                    self.arm_commander, z_level_pose='hover'),
    DoRotateAtTankFixedPosition('rotate_to_alpha_if_needed', condition_fn = [
                                                {'_not_fn': self.at_angle, 'rotation_pose': 'alpha'},
                                                 self.at_hover_level],
                                    self.arm_commander, rotation_pose='alpha'),
    DoMoveNamedPose('move_home_first', condition_fn=[{'_fn': self.in_a_region, 'logical_region': 'work'},
                                                self.at_hover_level],
                                    self.arm_commander, named_pose='home'),
    DoMoveNamedPose('move_task_pose', condition_fn=True,
                                    self.arm_commander, named_pose=self.query_logical_pose_of_task),
    ...
```

## 5.6. The base class ConditionalBehaviour and behaviour conditions

The class **`ConditionalBehaviour`** is the base for specialized behaviours adopting the conditional behaviour notion. The **`condition_fn`** parameter is pre-processed at the class creation phase and converted into a form facilitating evaluation during the tick-tock phase.

```
class ConditionalBehaviour(Behaviour):
    SUCCESS_IF_FALSE = 0
    FAILURE_IF_FALSE = 1
    def __init__(self, name, condition_fn=True, policy=ConditionalBehaviour.SUCCESS_IF_FALSE):
        super(ConditionalBehaviour, self).__init__(name)

    def preprocess_condition_fn():
       ....

    def is_condition_satisfied():
       ....
```

In the reference implementation, the parameter **`condition_fn`** accepts the following different forms of a condition.

| Acceptable forms | Examples |
|---|---|
| A bool constant (**`True`**, **`False`**) | `DoMoveNamedPose('nodename', condition_fn=True, ...` |
| A function definition | `DoMoveNamedPose('nodename', condition_fn=self.func, ...` |

| | |
|---|---|
| - The function returns a boolean value. | `def func(self):`<br>`  ...`<br>`  return True` |
| A dictionary<br>- Specify a function definition through either the key `'_fn'` or `'_not_fn'` . Their difference is that `'_not_fn'` negates the boolean value returned from the function.<br>- Other key-value pairs are passed to the function as keyword parameters. | `DoMoveNamedPose('nodename', condition_fn = {`<br>`  '_fn': self.is_timeout, 'time_lapse': 10}`<br>`}`<br>`def is_timeout(self, time_lapse):`<br>`  ...`<br>`  return True` |
| A list<br>- Specify a composite condition of predicates connected by the logical AND operator.<br>- Each element can be a boolean constant, a function, or a dictionary as specified above. | `DoMoveUpDown('nodename, condition_fn = [`<br>`  {'_fn': self.task_is_timeout, 'duration': 40},`<br>`  {'_fn': self.in_a_region,'logical_region': 'work'},`<br>`  self.below_hover_level,`<br>`  {'_not_fn': self.wrong_orientation},`<br>`]` |

The parameter **`policy`** determines whether to return **`SUCCESS`** or **`FAILURE`** when the condition is **`False`**. Usually, returning **`SUCCESS`** is suitable for a **`Sequence`** parent with memory while **`FAILURE`** is for a **`Selector`** parent with memory.

### 5.7. Developing application behaviours based on the class ConditionalCommanderBehaviour

The base class **`ConditionalCommanderBehaviour`**, has merged the features of the class **`CommanderBehaviour`**, discussed in section 5.3, and the class **`ConditionalBehaviour`**, discussed in section 5.6. It offers developers the following:

- A framework for managing the interaction with the general commander interface.
- A framework for plugin of possibly complex condition.

The three functions that are to be specialized in subclasses of **`ConditionalCommanderBehaviour`** are given in the following.

```
class ConditionalCommanderBehaviour(ConditionalBehaviour):
    def __init__(self, name, condition_fn=True, arm_commander=None):
        super(ConditionalCommanderBehaviour, self).__init__(name, condition_fn)

    def update_when_ready(self):
       return Status.FAILURE

    def update_when_busy(self):
       return Status.RUNNING

    def tidy_up(self):
       self.arm_commander.abort_move()
```

Similar to the class `CommanderBehaviour`, a subclass can define specific actions to do in different commander states by overriding these functions:

- `update_when_ready()`: called if the state of the general commander is READY **and** `condition_fn` returns True.
- `update_when_busy()`: called if the state of the general commander is BUSY.
- `tidy_up()`: called if the state of the general commander is SUCCEEDED, ABORTED, or ERROR.

The following figure depicts how to implement the functions in the subclasses of **`ConditionalCommanderBehaviour`**.

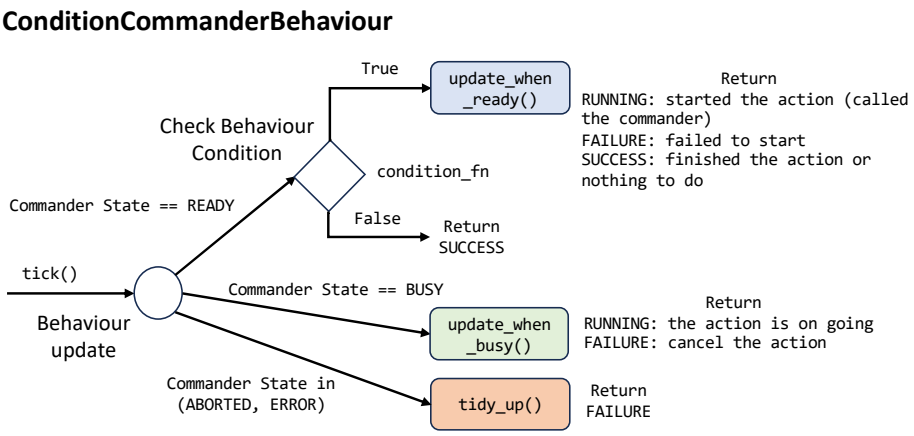


Figure 12. The possible pathways of a tick in the ConditionCommanderBehaviour, which includes a conditional divergence in the READY state.

Not that the implementations of these function should be non-blocking and returning as soon as possible.

The following shows the implementation of the **`DoMoveUpDown`** class. When the commander is READY, the function **`update_when_ready`** will convert the logical z pose into physical z pose and then instructs the commander to move in z-direction. Note the parameter **`wait`** is set to **`False`** for a non-blocking call.

```
class DoMoveUpDown(ConditionalCommanderBehaviour):
    def __init__(self, name, condition_fn, arm_commander, z_level_pose):
        super(DoMoveUpDown, self).__init__(name, condition_fn, arm_commander)
        self.z_level_pose = z_level_pose

    def update_when_ready(self):
        logical_scence: CGRASLogicalSceneModel = self.the_blackboard.scene
        physical_z = logical_scence.lookup_z_level_pose(self.z_level_pose)
        if physical_z is None:
            rospy.logerr(f'DoMoveUpDown ({self.name}): invalid z-level pose {self.z_level_pose}')
            return Status.FAILURE
        self.arm_commander.move_to_position(z=physical_z, wait=False)
        rospy.loginfo(f'DoMoveUpDown ({self.name}): started move to z level: {self.z_level_pose}')
        return Status.RUNNING

    def update_when_busy(self):
       # this is repeatedly called when the general commander is still BUSY
       # a potential use is real-time collision detection (i.e. return FAILURE before a collision)
```

Specific behaviours of **`ConditionalCommanderBehaviour`** are normally expected to execute actions **asynchronously** (calling the general commander with **`wait=False`**). The return value of the function is therefore either **`RUNNING`** (the call is made successfully) or **`FAILURE`** (failed to make the call).

Recall that the asynchronous call is handled by the base class **`ConditionalCommanderBehaviour`** . The update function of the base class will return **`SUCCESS`** status or others according to the final state of the general commander.

To ensure predictable operations, the parent **`Composite`** class (**`Sequence`** or **`Selector`**) should operate in memory mode (i.e. **`memory`** set to **`True`**).

## 5.4. Design Patterns involving Behaviour Conditions

**`ConditionalCommanderBehaviour`** can be the basis of many general repeatable solutions to common situations in robotic manipulation. The following lists three design patterns. It is worthwhile to develop more design patterns.

### *5.4.1. The Convergence Pattern*

The convergence pattern supports an orderly manipulation of the robot to a target through specific intermediate poses from various starting status.

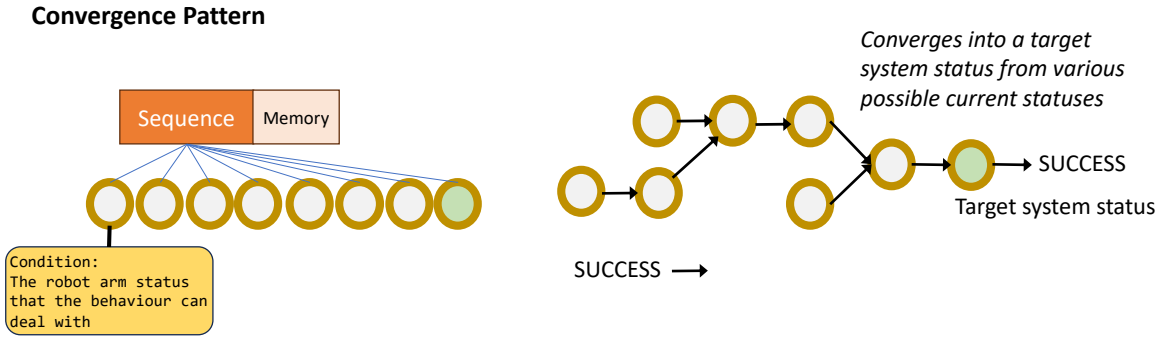


Figure 13. The convergence pattern.

The task of *move and capture underwater images* in the CGRAS project is an example problem that can exploit the convergence pattern. The target of the task is to move to a particular logical pose in the tank for image capture. It must cater for the various starting status of the robot. The end-effector may be in one of the possible poses: the stow or the home named pose, submerged or hover above water, or in various positions in the tank.

The sequence of behaviours represents a tree-like structure that brings the various starting points as leaves to the same target as the root. As formal descriptions of the starting points and the behaviour tree building code has explicated tied together the conditions and the corresponding behaviours for the traversal of the appropriate branches.

### *5.4.2. The Behaviour Exception Pattern*

This pattern supports the implementation of error handling during RUNNING state in a sequence of conditional behaviours. It is possible to add error checking code to the function **`update_when_busy`** in the class **`ConditionalCommanderBehaviour`**. It returns **`FAILURE`** to halt the sequence. The behaviour exception pattern offers behaviour specific error handling.

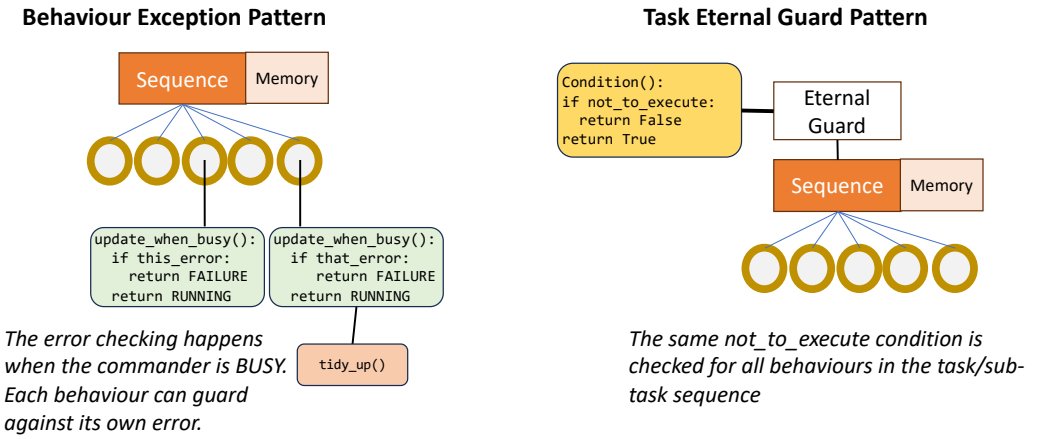


Figure 14. The behaviour exception pattern and for comparison the task eternal guard pattern

The use case of the behaviour exception pattern is different from that of Eternal Guard decorator of PyTrees. In the latter the condition is applicable to all the child behaviours.

### *5.4.3. The Sense-and-Move Cycle Pattern*

This pattern supports the implementation of sense-and-move cycle, a common problem in robotic manipulation.

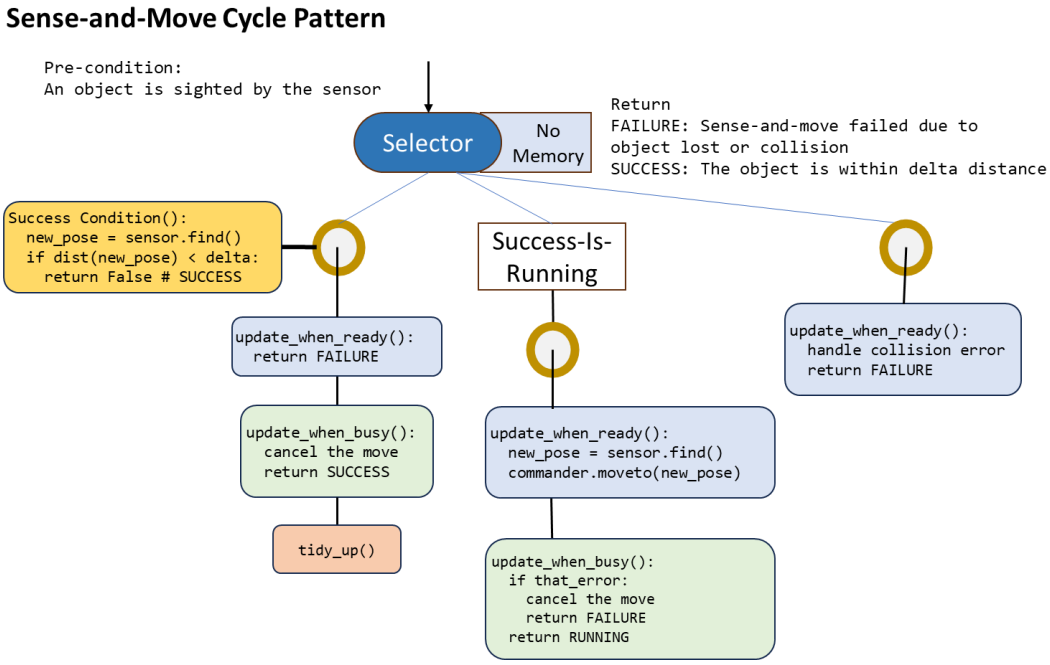


Figure 15. The sense-and-move cycle pattern

A sense-and-move cycle aims to approach a target object by moving the end-effector to its proximity. As an alternative to servoing, the solution runs in a cycle of updating the object pose and moving the end-effector to the pose. The cycle stops when either the approach is successful (e.g. within a certain distance) or failure to track the object.

### 5.5. Build-time binding and tick-tock-time binding of Behaviour Parameters

Parameter passing is exploited to specify variations in the action of a Behaviour class. Consider the following example where the parameter `named_pose` specifies the target of the behaviour **`DoMoveNamedPose`**. This is known as build-time binding because the constant parameter is fixed.

```
...
DoMoveNamedPose('move_task_pose', True, self.arm_commander, named_pose='home'),
...
```

Delaying the binding to tick-tock time enables more dynamic behaviours. The reference implementation shows that accepting functions as parameters can adapt the actions to the requirements of the executing task.

```
...
DoMoveNamedPose('move_task_pose', True, self.arm_commander, named_pose=self.query_logical_pose_of_task),
...
```

The function **`query_logical_pose_of_task()`** queries the current task the target named pose enable the execution of the task in a de-coupled manner.

### 5.6. Client Applications interacting with the Task Manager and the Task

A client application, such as a ROS action server, can interact with the task manager in the following manner.

- The client application creates a **`Task`** object of the desired class (e.g. **`CalibrateTask`**, **`MoveNamedPoseTask`**).
- It submits the **`Task`** object to the task manager through the **`submit_task`** function.
- It then has three options:
    - In a loop, monitors the state of the task using the **`get_state()`** function. The loop breaks at the completion of the task (successfully or otherwise).
    - Calls the **`wait_for_completion()`** function of the Task class to block until task completion.
    - Calls the **`cancel()`** function of the Task class to cancel the task.

The following example the callback function of a newly built CGRAS action server, which illustrates the structure of ROS action server interface for the task manager.

```
class TaskActionServer():
    ...
    def received_goal(self, goal):
        self.task = None
        result = TaskResult()
        if goal.name == 'calibrate':
            self.task_manager.submit_task(task:=CalibrateTask())
        elif goal.name == 'reset':
            self.task_manager.submit_task(task:=MoveNamedPoseTask(goal.data))
        self.task = task
        ...
        while self.task is not None:
            rospy.sleep(0.1)

            if task.get_state() in COMPLETION_STATES: # ABORTED, FAILED, INVALID, SUCCEEDED
                break
            if timeout:
                self.task.cancel()
                self.the_action_server.set_aborted(result)
        if self.the_action_server.is_preempt_requested():
            result.data = 'ABORTED_DUE_TO_CANCEL_GOAL'
```

```
            self.the_action_server.set_aborted(result)
        elif self.task.get_state() in [TaskStates.ABORTED, TaskStates.FAILED, TaskStates.INVALID]:
            result.data = self.task.result
            self.the_action_server.set_aborted(result)
        else:
            self.the_action_server.set_succeeded(result)
    ...
    def received_preemption(self):
        rospy.loginfo(f'preemption received')
        if self.task is not None:
            self.task.cancel()
    ...
```

## 5.8. The Task Scene, Logical Poses, and Physical Poses in Task Configuration

The layered structure of the scene helps isolate each layer from the complexity in other layers. For example, the logical poses considered in the application scene, such as the home pose, stow pose, and the grid poses are independent from their actual physical locations. Developers of behaviour trees can focus on high-level task execution, leaving the lower layers to oversee the mapping to physical poses and joint-space poses.

The following figure illustrates the task scene specified in the reference implementation. The application scene, including the logical poses and other task related object definitions, is defined in the task scene configuration file. In the CGRAS application, the tank is a task related object. The physical scene, which comprises the robot and task irrelevant objects, is defined in a set of xacro, urdf, and meshes files. Moveit provides a setup assistant that can solicit configuration settings and generate a catkin package from the file set.

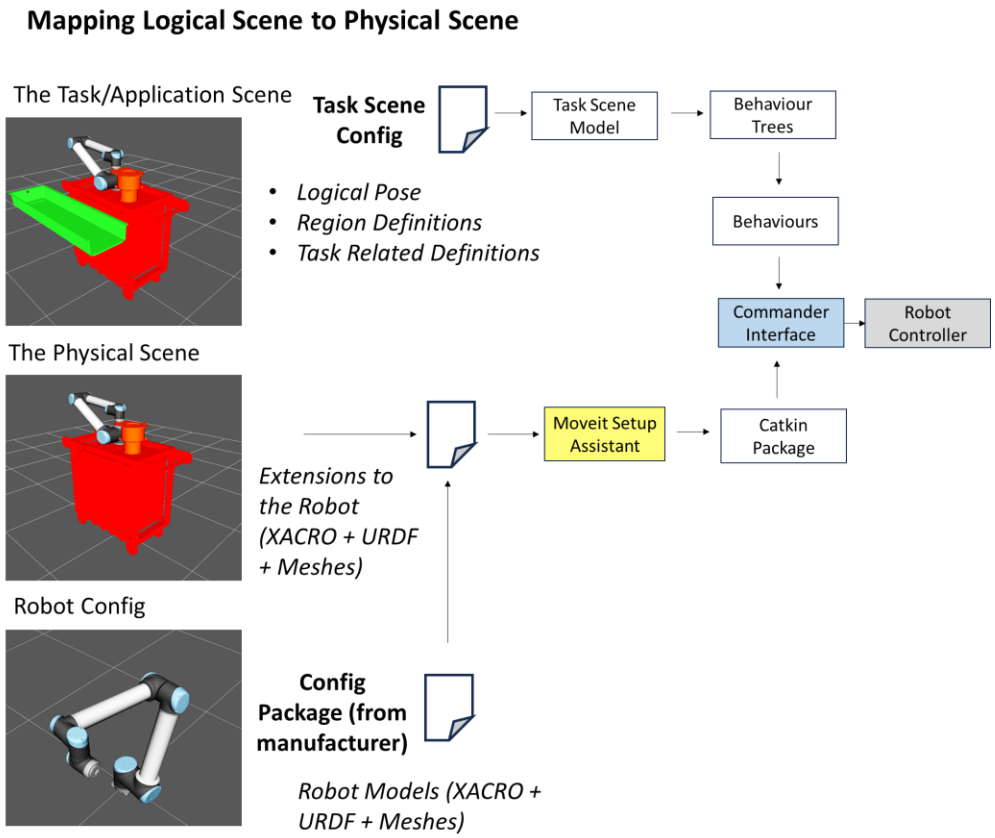


Figure 16. The specification of the task scene and the physical scene configurations

The CGRAS task scene configuration file has two parts.

- Definitions of the logical scene, including the workspace, named poses, and named regions.
- Definitions of the logical poses with reference to the tank, including poses and aliases for the image capture task. The diverse types of logical poses are shown in Figure 12.

The following shows example logical pose definitions in the task scene configuration (a yaml file).

```
  z_level_poses:
    default: 1.2
    hover: 1.2
    submerged: 1.02
  ee_rotation_poses:
    alpha: [-3.14, 0, 0]
    beta: [-3.14, 0, -3.14]
    delta: [-2.80, 0, null]
    gamma: [-3.14, 0, -1.58]
...
  tiles:
  - tile_x: 0
    tile_y: 0
    origin_position: [0.21, 0.0]
    ee_rotation: alpha
  - tile_x: 1
    tile_y: 0
    origin_position: [0.21, 0.45]
    ee_rotation: alpha
  - tile_x: 2
    tile_y: 0
    origin_position: [0.21, 0.90]
    ee_rotation: beta
```

Note that `null` represents the current value of the pose at use-time (i.e. when the pose is being used).

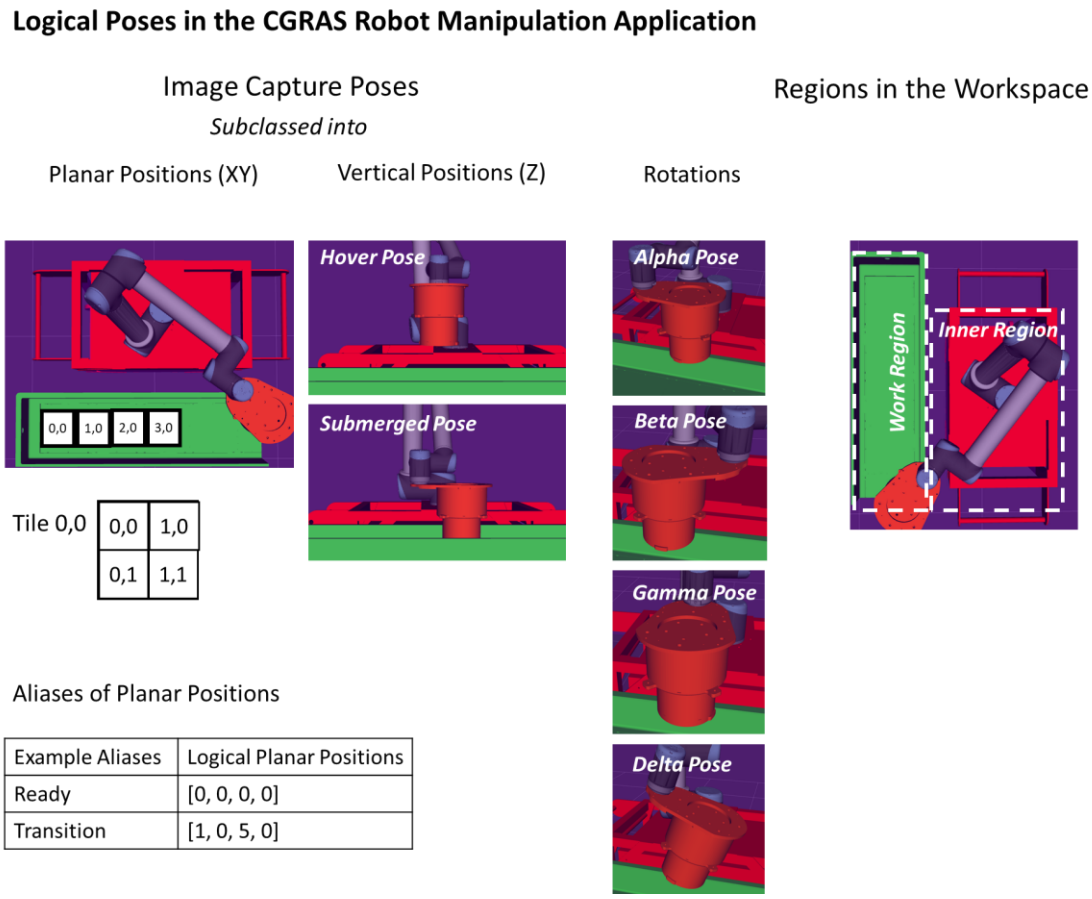


Figure 17. Illustrations of the logical poses defined for the CGRAS application

## 6. Acknowledgement

The author of this report thanks Timothy Morris for sharing with me inspiring insights in robotics and best practices in robotic manipulation programming. His effort in a thorough review on the first version of the report is also very much appreciated. The author is also grateful to Das Gunasinghe for the technical discussions proven to be especially useful for refining the architecture, and Josh Esplin for informing me his valuable knowledge in programming robots.

## A.1. File Structure of the Reference Implementation

The reference implementation stored in the repository follows closely the catkin package structure. The following figure organizes the files according to the layered architecture. The CGRAS specific files are to be enhanced or adapted to the goals of other applications.

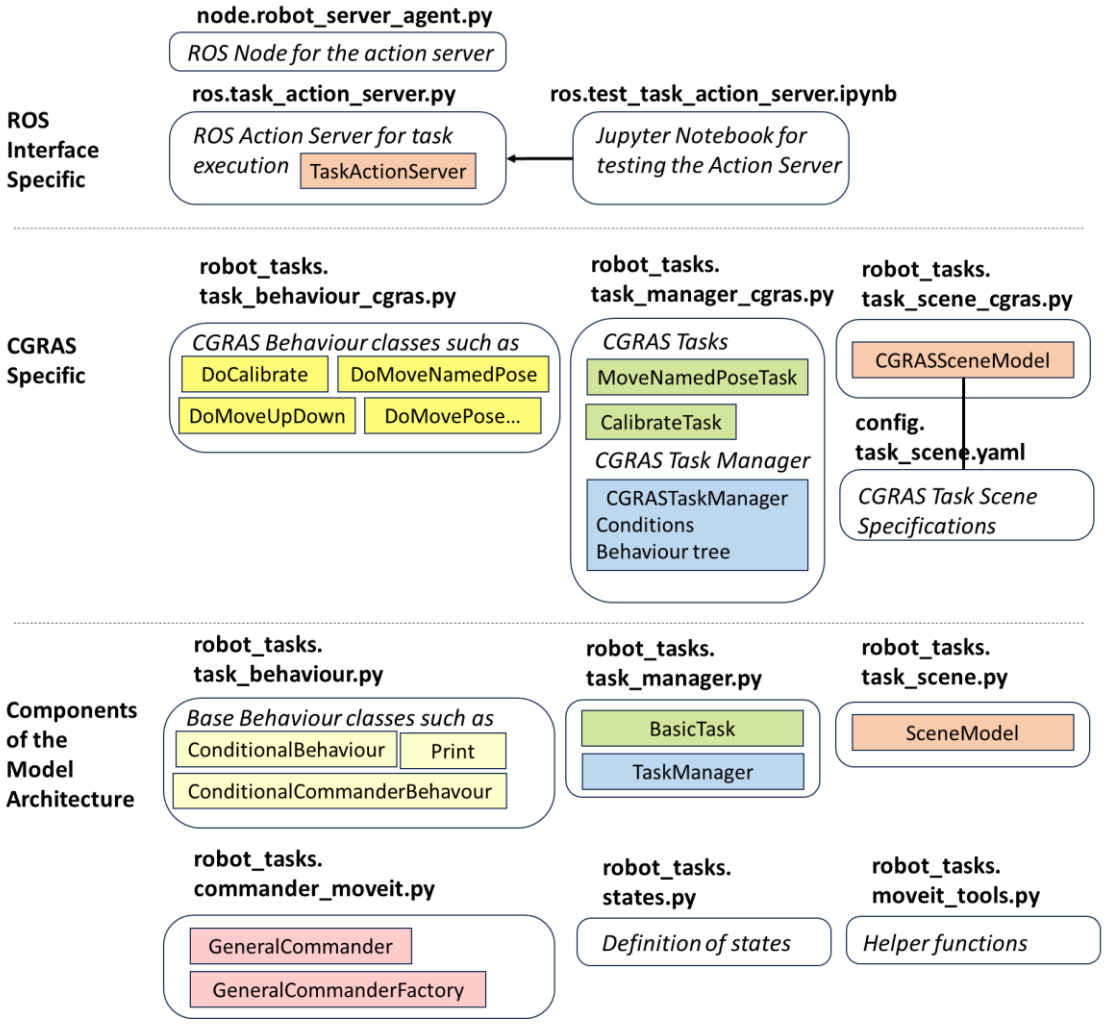


Figure A1. Structural view of the source files in the reference implementation of the mode architecture

In developing a new robot arm manipulation application, the bottom group of files and components should require no change. It is envisaged that many of the CGRAS specific files can be reused or can be the basis for the development of new behaviour trees and their components.

## A.2. Utilizing the Behaviour Exception Pattern: Integration with Sensor-based Collision Detection

This section aims to illustrate the use of the behaviour exception pattern for adding sensor-based collision detection to a behaviour class.

The movement behaviours within the tank area are prone to collision with the tank due to mis-calibration. The pose of the tank obtained from calibration does not match the reality.

An end-effector-mounted range finder can inform the distance from the bottom of the tank. If the distance is available in the ROS middleware, a sensor-based collision detection mechanism can exploit this data source. The below figure illustrates two similar methods for the enhancement.

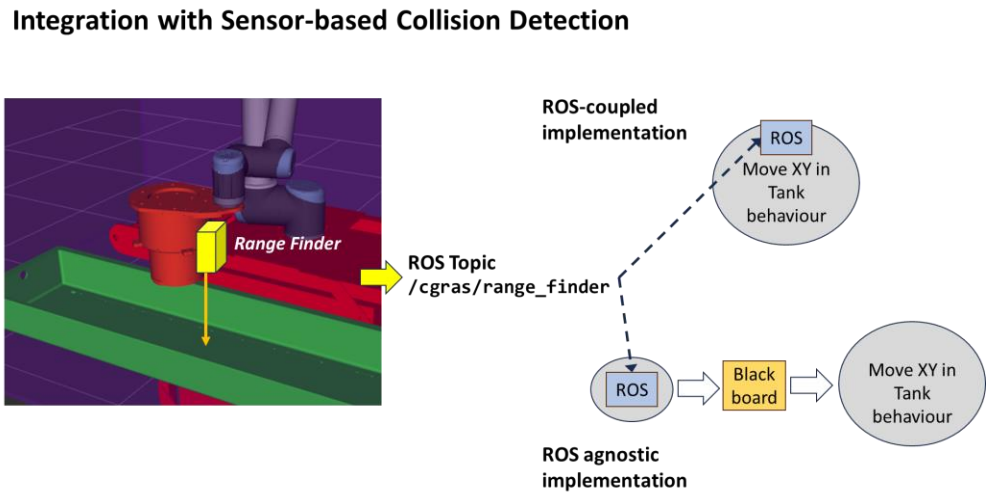


Figure A2. Two system designs for obtaining the range finder readings in a behaviour class for collision detection.

The behaviour exception pattern can significantly ease the design effort. The following show how to enhance the existing class `DoMoveXYAtTankFixedYOrientation` with sensor-based collision detection when the behaviour is RUNNING. The behaviour class may move the end-effector deep inside the tank. An enhancement to the class is to half the movement if the distance from the bottom is too close.

The following lists the content of `DoMoveXYAtTankVisualCDROS`, which is an enhanced version of the parent class `DoMoveXYAtTankFixedYOrientation`. The instance variable `current_dist` has the latest readings from the ROS topic `range_finder`. The function `update_when_busy`, called when the behaviour is `RUNNING`, checks if the distance is closer than the threshold. It called the general commander to abort the move immediately and return `FAILURE` to the parent of this node.

```
class DoMoveXYAtTankVisualCDROS(DoMoveXYAtTankFixedYOrientation):

    def __init__(self, name, condition_fn=True, policy=ConditionalBehaviour.SUCCESS_IF_FALSE, arm_commander=None,
                         logical_pose=None, z_level_pose=None):
        super(DoMoveXYAtTankVisualCDROS, self).__init__(name, condition_fn, policy=policy, arm_commander=arm_commander,
                                                        logical_pose=logical_pose, z_level_pose=z_level_pose)
        self.MIN_DISTANCE = 10.0
        self.range_finder_sub = rospy.Subscriber('/cgras/range_finder', Float32, self._cb_range_finder_update)
        self.current_dist = None

    def _cb_range_finder_update(self, msg:Float32):
        self.current_dist = msg.data    # update a local variable as the cache from ROS message

    def update_when_busy(self):
        too_close = False if self.current_dist is None else self.current_dist < self.MIN_DISTANCE
        rospy.loginfo(f'DoMoveXYAtTankVisualCDROS: too_close: {too_close}')
        if too_close:
            rospy.logerr(f'DoMoveXYAtTankVisualCDROS ({self.name}): visual collision detection alert (too close)')
            self.arm_commander.abort_move()
            the_task = self.the_blackboard.task
            the_task.result = f'INTERRUPTED RAISED BY VISUAL COLLISION DETECTION (distance too close:
{self.current_dist})'
            return Status.FAILURE
        return Status.RUNNING
```

The above implementation is ROS coupled. The following provides an alternative implementation that reads the distance from PyTree blackboard. The abstraction layer isolates the behaviour from code change if the data comes from another source.

```
Class DoMoveXYAtTankVisualCDBlackboard(DoMoveXYAtTankFixedYOrientation):

    def __init__(self, name, condition_fn=True, policy=ConditionalBehaviour.SUCCESS_IF_FALSE, arm_commander=None,
                         logical_pose=None, z_level_pose=None):
        super(DoMoveXYAtTankVisualCDROS, self).__init__(name, condition_fn, policy=policy, arm_commander=arm_commander,
```

```
                                                    logical_pose=logical_pose, z_level_pose=z_level_pose)
        self.MIN_DISTANCE = 10.0

    def update_when_busy(self):
        current_dist = self.the_blackboard.range_finder  # obtain the distance from the blackboard instead of cached ROS
        too_close = False if current_dist is None else current_dist< self.MIN_DISTANCE
        rospy.loginfo(f'DoMoveXYAtTankVisualCDROS: too_close: {too_close}')
        if too_close:
            rospy.logerr(f'DoMoveXYAtTankVisualCDROS ({self.name}): visual collision detection alert (too close)')
            self.arm_commander.abort_move()
            the_task = self.the_blackboard.task
            the_task.result = f'INTERRUPTED RAISED BY VISUAL COLLISION DETECTION (distance too close: {current_dist})'
            return Status.FAILURE
        return Status.RUNNING
```

The following simple ROS node captures the readings from the topic and stores them in the blackboard.

```
Class RangeFinderListener():
    def __init__(self):
        rospy.on_shutdown(self.cb_shutdown)
        # ros subscribe
        self.range_finder_sub = rospy.Subscriber('/cgras/range_finder', Float32, self._cb_range_finder_update)
        # pytrees blackboard
        self.the_blackboard = py_trees.blackboard.Client()
        self.the_blackboard.register_key(key='range_finder', access=py_trees.common.Access.WRITE)

    def cb_shutdown(self):
        rospy.loginfo('the ros node is being shutdown')

    def _cb_range_finder_update(self, msg:Float32):
        self.the_blackboard.range_finder = msg.data   # write the updated distance to the blackboard
```

## A.3. The Full Behaviour Tree

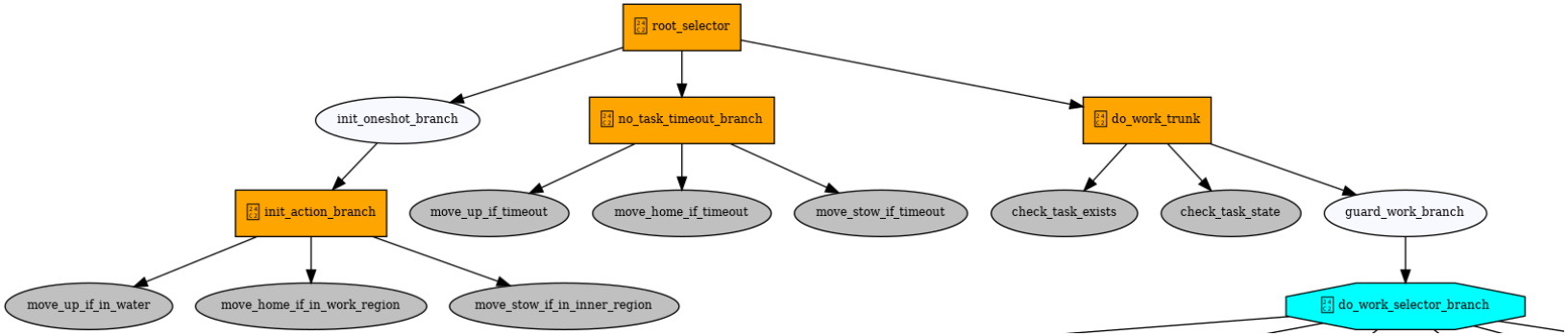


Figure A.3.1. The top-levels of the tree comprise three major branches: one for initialization of the arm pose, one for reset arm pose due to no new task timeout, and one for the handling of the three CGRAS tasks.

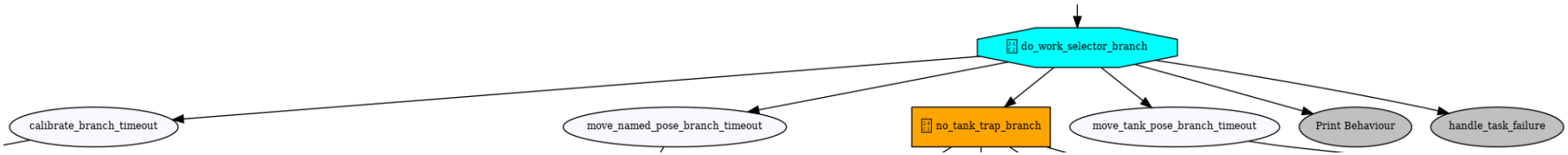


Figure A.3.2. The do_work_selector_branch comprises of the three CGRAS tasks, a branch the traps the case of no calibrated tank, and a behaviour that handles failure.

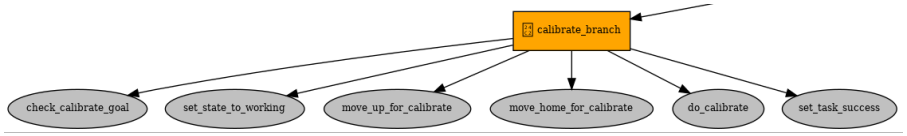


Figure A.3.3. The calibrate branch under the do_work_selector_branch

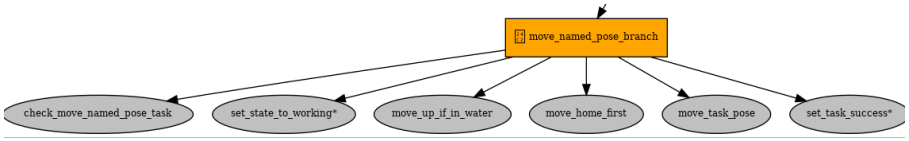


Figure A.3.4. The move_named_pose branch under the do_work_selector_branch

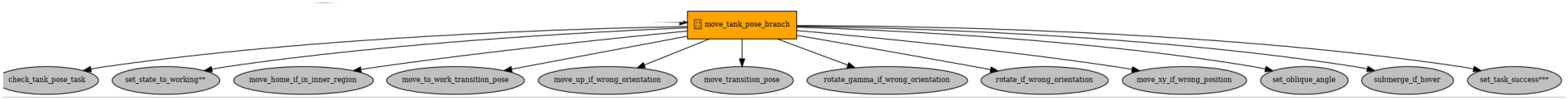


Figure A.3.5. The move_tank_pose branch under the do_work_selector_branch