D4.10

Lifecycle and System Modes Software Release Y3

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.10
Deliverable name	Lifecycle and System Modes – Software Release Y3
Date	December 2020
Dissemination level	public
Workpackage and task	4.3
Author	Arne Nordmann (Bosch)
Contributors	Ralph Lange (Bosch)
Keywords	micro-ROS, robotics, ROS, microcontrollers, runtime lifecycle, system modes
Abstract	This document provides links to the released software and documentation for deliverable D4.10 *Lifecycle and System Modes Software Release Y3* of the Task 4.3 *Lifecycle and system modes*.

Contents

1 Overview to Results

This document provides links to the released software and documentation for deliverable D4.10 *Lifecycle and System Modes Software Release Y3* of the Task 4.3 *Lifecycle and system modes*.

As an entry-point to all software and documentation, we created a dedicated webpage on the micro-ROS website: https://micro-ros.github.io/system_modes/

The annex includes a copy of this webpage and copies of the major documentation files of this software release.

2 Links to Software Repositories

Documentation of the system modes package on the micro-ROS website was updated to cover the latest developments:

- Git repository: https://github.com/micro-ROS/micro-ROS.github.io

2.1 Basic Lifecycle and System Modes

2.1.1 ROS 2 Lifecycle for micro-ROS

The ROS 2 Lifecycle was implemented for micro-ROS as part of the C programming language client library *rclc*:

- Git repository: https://github.com/ros2/rclc
  Package name: rclc
  Package path: ./rclc_lifecycle
- Git repository: https://github.com/ros2/rclc
  Package name: rclc
  Package path: ./rclc_examples

The rclc_lifecycle package and the rclc examples package were bloomed and released for the ROS 2 distributions Dashing, Eloquent, Foxy, and Rolling:

- Git repository: https://github.com/micro-ROS/rclc-release
  Releases: ROS 2 Dashing (0.1.4-1), ROS 2 Eloquent (0.1.4-1), ROS 2 Foxy (0.1.4-1), ROS 2 Rolling (0.1.4-1)

2.1.2 System Modes

The parser for the system modes model and the mode inference mechanisms (including the extended lifecycle) are provided in a repository named system_modes, together with two ROS nodes for mode management and mode monitoring.

- Git repository: https://github.com/micro-ROS/system_modes
  Package name: system_modes
  Package path: ./system_modes
  Fixed bugs: #9, #11, #13, #14, #24, #25, #26, #42, #44, #45

Demo of system modes concept and implementation in same repository:

- Git repository: https://github.com/micro-ROS/system_modes
  Package name: system_modes_examples
  Package path: ./system_modes_examples

The system modes package and the system modes example package were bloomed and released for the ROS 2 distributions Dashing, Eloquent, Foxy, and Rolling:

- Git repository: https://github.com/micro-ROS/system_modes-release
  Releases: ROS 2 Dashing (0.4.1-1), ROS 2 Eloquent (0.4.1-1), ROS 2 Foxy (0.4.1-1), ROS 2 Rolling (0.4.1-1)

2.2 Diagnostics

The ROS diagnostics updater and diagnostics aggregator package were ported to ROS 2 as a precondition for micro-ROS diagnostics and integration with system modes:

- Git repository: https://github.com/ros/diagnostics
  Package name: diagnostic_updater
  Package path: ./diagnostic_updater
- Git repository: https://github.com/ros/diagnostics
  Package name: diagnostic_aggregator
  Package path: ./diagnostic_aggregator

For micro-ROS, a simple diagnostics framework built against rclc was created, comprising of:

- Diagnostic messages - Diagnostic messages and services suited for micro-ROS, e.g., no dynamic message field

- Diagnostic Updater - rclc convenience functions for diagnostic updaters and tasks, publishing micro-ROS diagnostic messages
- Common Diagnostics - Micro-controller specific monitors and diagnostic updaters

The respective repositories are:

- Git repository: https://github.com/micro-ROS/micro_ros_diagnostics
  Package name: micro_ros_diagnostics
  Package path: ./micro_ros_diagnostic_msgs
- Git repository: https://github.com/micro-ROS/micro_ros_diagnostics
  Package name: micro_ros_diagnostics
  Package path: ./micro_ros_diagnostic_updater
- Git repository: https://github.com/micro-ROS/micro_ros_diagnostics
  Package name: micro_ros_diagnostics
  Package path: ./micro_ros_common_diagnostics

2.3 Integration with Real-Time Executor

Integration with the real-time executor does not require additional software extensions. Instead, the available feature of running multiple executors in addition to the fact that system modes map to standard ROS 2 node parameters allow this integration already:

If nodes are run with multiple executor instances, their callbacks can be mapped to differently prioritized threads, task, etc. As system modes map to node parameters, i.e. mode changes map to parameters changes, these can easily be monitored to map callbacks to different executors in order to adapt callback priorities depending on system mode changes.

This concept will be validated in the domestic outdoor robot use-case in 2021 (Task 6.3).

2.4 Error Propagation

The system modes package was extended to incorporate a lightweight concept and implementation for specifying error handling and recovery. It was recently merged to the main branch and released for ROS 2 dashing, eloquent, foxy, and rolling.

The error handling and recovery feature was recently validated and successfully demonstrated in the EU-funded MROS project (an Integrated Technical Project (ITP) in RobMoSys), e.g., within a navigation sub-system of a mobile robot.

3 Annex 1: Webpage on System Modes

Content of https://micro-ros.github.io/system_modes/ from December 10th 2020.

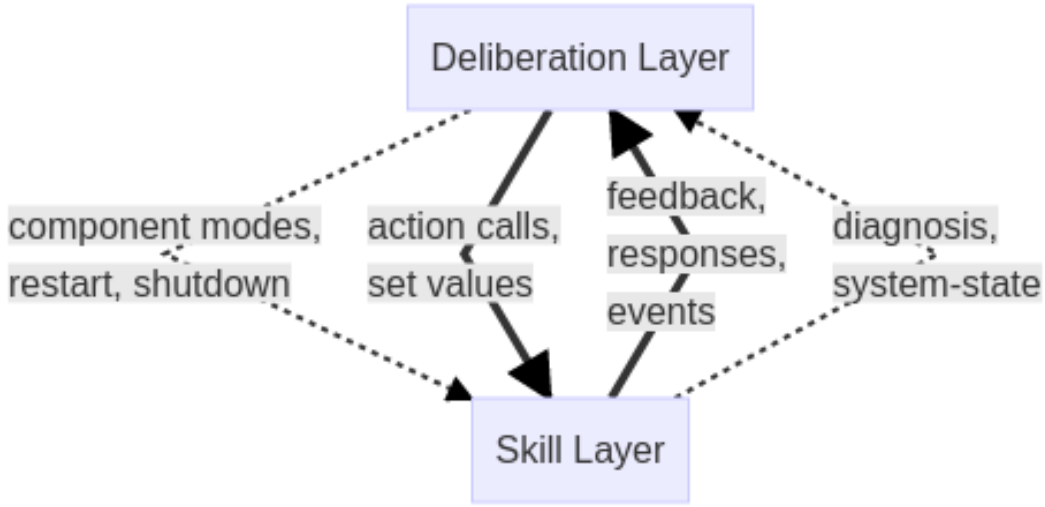
3.1 Introduction and Goal

Modern robotic software architectures often follow a layered approach. The layer with the core algorithms for SLAM, vision-based object recognition, motion planning, etc. is often referred to as *skill layer* or *functional layer*. To perform a complex task, these skills are orchestrated by one or more upper layers named *executive layer and planning layer*. Other common names are *task and mission layer* or *deliberation layer(s)*. In the following, we used the latter term.

We observed three different but closely interwoven aspects to be handled on the deliberation layer:

1. **Task Handling**: Orchestration of the actual task, the *straight-forward*, *error-free* flow.
2. **Contingency Handling**: Handling of task-specific contingencies, e.g., expectable retries and failure attempts, obstacles, low battery.
3. **System Error Handling**: Handling of exceptions, e.g., sensor/actuator failures.

The mechanisms being used to orchestrate the skills are service and action calls, re-parameterizations, set values, activating/deactivating of components, etc. We distinguish between *function-oriented calls* to a running skill component (set values, action queries, etc.) and *system-oriented calls* to individual or multiple components (switching between component modes, restart, shutdown, etc.).

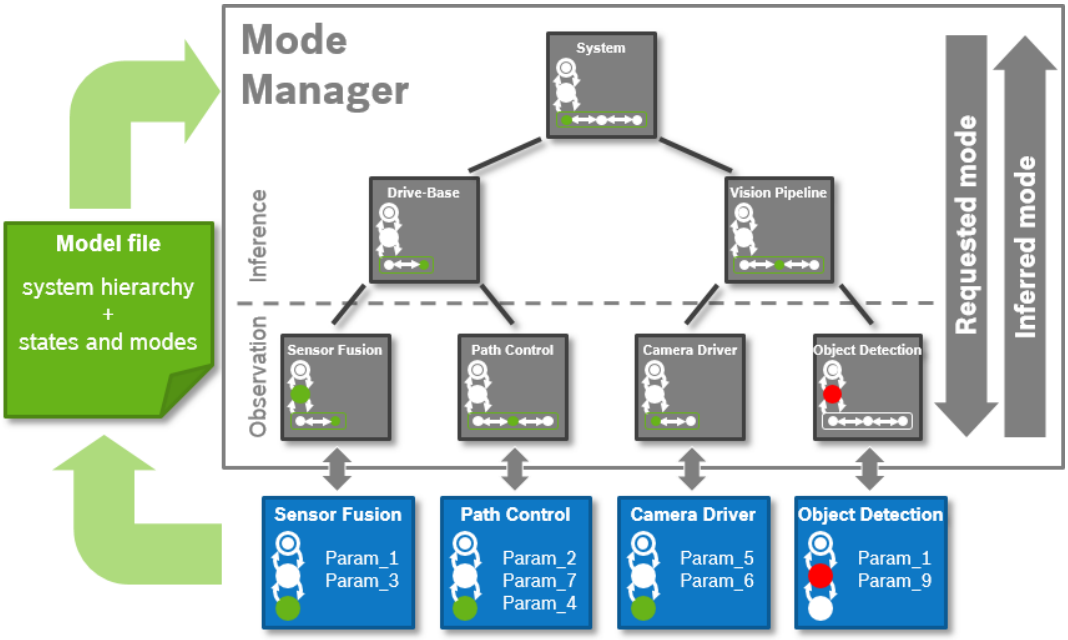


Analogously, we distinguish between *function-oriented notifications* from the skill layer in form a feedback on long-running service calls, messages on relevant events in the environment, etc. and *system-oriented notifications* about component failures, hardware errors, etc.

Our observation is that interweaving of task handling, contingency handling, and system error handling generally leads to a high complexity of the control flow on the deliberation layer. Yet, we hypothesize that this complexity can be reduced by introducing appropriate abstractions for system-oriented calls and notifications.

Therefore, our **goal** within this work is to provide suitable abstractions and framework functions for (1.) system runtime configuration and (2.) system error and contingency diagnosis, to reduce the effort for the application developer of designing and implementing the task, contingency and error handling.

This goal is illustrated in the following example architecture, which is described and managed based on a model file:



The main features of the approach are (detailed in the remainder):

1. *Extended Lifecycle*: Extensible concept to specify the runtime states of components, i.e ROS 2 lifecycle nodes.
2. *System Hierarchy and Modes*: Modeling approach for specifying a ROS system in terms of its system hierarchy and *system modes*, i.e. different (sub-)system configurations.
3. *Mode Manager*: A module to manage and change the system runtime configuration.
4. *Mode Inference*: A module for deriving the entire system state and mode from observable system information, i.e. states, modes, and parameters of its components.
5. *Error Handling*: Lightweight concept for specifying an error handling and recovery mechanism.

3.2 Requirements

The list of requirements is maintained in the doc folder of the micro-ROS system modes repository, at: https://github.com/micro-ROS/system_modes/blob/master/system_modes/doc/requirements.md

3.3 Background: ROS 2 Lifecycle

Our approach is based on the ROS 2 Lifecycle. The primary goal of the ROS 2 lifecycle is to allows greater control over the state of a ROS system. It allows consistent initialization, restart and/or replacing of system parts during runtime. It provides a default lifecycle for managed ROS 2 nodes and a matching set of tools for managing lifecycle nodes.

The description of the concept can be found at: http://design.ros2.org/articles/node_lifecycle.html The implementation of the Lifecycle Node is described at: https://index.ros.org/doc/ros2/Managed-Nodes/.

3.4 Main Features

3.4.1 Basic Lifecycle

The ROS 2 Lifecycle has been implemented for micro-ROS as part of the C programming language client library *rclc*, see rclc_lifecycle for source-code and documentation.

The rclc_lifecycle package is a ROS 2 package that provides convenience functions to bundle a ROS Client Library (rcl) node with the ROS 2 Node Lifecycle state machine in the C programming language, similar to the rclcpp Lifecycle Node for C++.

An example, how to use the rclc Lifecycle Node is given in the file `lifecycle_node.c` in the rclc_examples package.

3.4.2 Extended Lifecycle

In micro-ROS, we extend the ROS 2 lifecycle by allowing to specify modes, i.e. substates, specializing the *active* state based on the standard ROS 2 parameters mechanism. We implemented this concept based on rclc_lifecycle and rclcpp_lifecycle for ROS 2 and micro-ROS.

Documentation and code can be found at: github.com:system_modes/README.md#lifecycle

3.4.3 System Hierarchy and Modes

We provide a modeling concept for specifying the hierarchical composition of systems recursively from nodes and for specifying the states and modes of systems and (sub-)systems with the extended lifecycle, analogously to nodes. This system modes and hierarchy (SMH) model also includes an application-specific the mapping of the states and modes along the system hierarchy down to nodes.

The description of this model can be found at: github.com:system_modes/README.md#system-modes A simple example is provided at: github.com:system_modes_examples/README.md#example-mode-file

3.4.4 Mode Inference

The mode inference infers the entire system states (and modes) based on the lifecycle states, modes, and parameter configuration of its components, i.e. the ROS 2 lifecyle nodes. It parses the SMH model and subscribes to lifecycle/mode change requests, lifecycle/mode changes, and parameter events.

Based on the lifecycle change events it knows the *actual* lifecycle state of all nodes. Based on parameter change events it knows the *actual* parameter values of all nodes, which allows inference of the *modes* of all nodes based on the SMH model. Based on the SMH model and the inferred states and modes of all nodes, states and modes of all (sub-)systems can be *inferred* bottom-up along the system hierarchy. This can be compared to the latest *requested* states and modes to detect a deviation.

The documentation and code can be found at: github.com:system_modes/README.md#mode-inference The mode inference can be best observed in the mode monitor, a console-based debugging tool, see: github.com:system_modes/README.md#mode-monitor

3.4.5 Mode Manager

Building upon the *Mode Inference* mechanism, the mode manager provides additional services and topics to *manage and adapt* system states and modes according to the specification in the SMH model.

The documentation and code can be found at: github.com:system_modes/README.md#mode-manager A simple example is provided at: github.com:system_modes_examples/README.md#setup

3.4.6 Error Handling and Rules

If the *actual* state/mode of the system or any of its parts diverges from the *target* state/mode, we define rules that try to bring the system back to a valid *target* state/mode, e.g., a degraded mode. Rules work in a bottom-up manner, i.e. starting from correcting nodes before sub-systems before systems. Rules are basically defined in the following way:

```
if:
 system.target == {target state/mode} && system.actual != {target state/mode} && part.actual ==
then:
 system.target := {specific state/mode}
```

If *actual* state/mode and *target* state/mode diverge, but there is no rule for this exact situation, the bottom-up rules will just try to return the system/part to its *target* state/mode.

Note: This feature is suited for simple, well-defined rules according to the depicted syntax. For more complex orchestration, integration of system modes with ontological reasoning (*metacontrol*) has been validated and successfully demonstrated in the MROS project, e.g., within a navigation sub-system of a mobile robot.

3.5 Acknowledgments

This activity has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement n° 780785).