D4.8

Lifecycle and System Modes Software Release Y1

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.8
Deliverable name	Lifecycle and System Modes – Software Release Y1
Date	December 2018
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 D4.8 *Lifecycle and System Modes Software Release Y1* 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 D4.8 *Lifecycle and System Modes Software Release Y1* 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://microros.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

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

- Git repository: https://github.com/microROS/system_modes
  Package name: system_modes
  Package path: ./system_modes
  Major (squashed) commit: c1e6794

Demo of system modes concept and implementation in same repository:

- Git repository: https://github.com/microROS/system_modes
  Package name: system_modes_examples
  Package path: ./system_modes_examples
  Major (squashed) commit: c1e6794

Markdown source of the microros.github.io/system_modes/ webpage:

- Git repository: https://github.com/microROS/microros.github.io
  Webpage path: ./system_modes/
  Major commits: d2d9ba7, 72ab07a

3 Annex 1: Webpage on System Modes

Content of https://microros.github.io/system_modes/ from 14th December 2018.

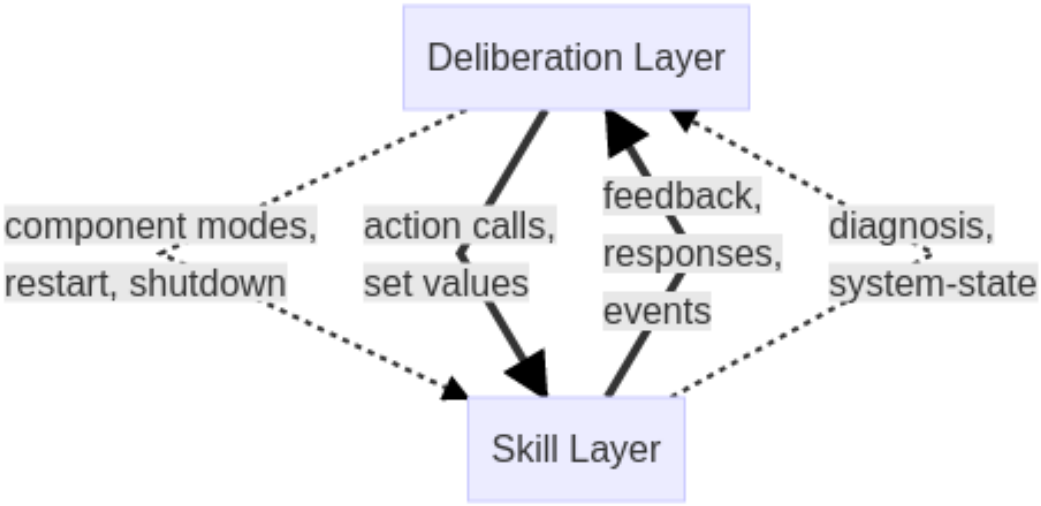
3.1 Introduction and Goal

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.).

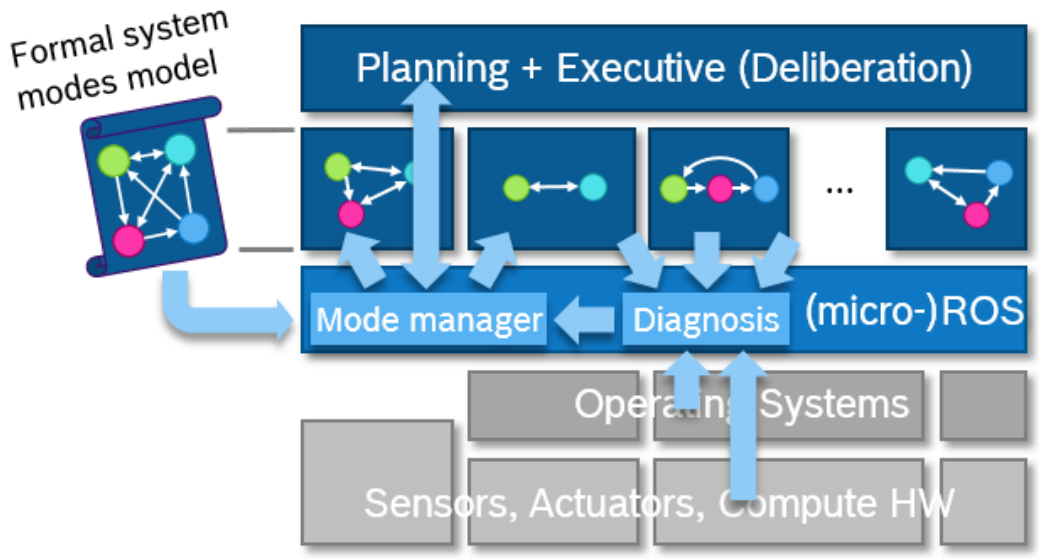


Analogously, we distinguish between *function-oriented notifications* from the skill layer in form a feedback on long-running service calls, messages on relevant events in the environment, etc. and *system-oriented notifications* about component failures, hardware errors, etc.

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 high-level architecture:



The envisioned key elements to achieve this goal are:

1. Extensible concept to specify the runtime states of components, i.e ROS 2 nodes.
2. Modeling approach for specifying system modes based on these component states.
3. Diagnosis module for deriving relevant information from the operating systems, the hardware and the functional components.
4. Mode manager module for system runtime configuration.

3.2 Requirements

The list of requirements is maintained in the doc folder of the micro-ROS system modes repository, at: https://github.com/microROS/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 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 rcl and rclcpp for ROS 2 and micro-ROS.

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

3.4.2 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 subsystems 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

3.4.3 Mode manager

The mode manager allows for runtime system adaptation based on such a system hierarchy and modes model. It parses the model and provides all services and topics to request state and mode changes and to monitor these changes.

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

3.5 Roadmap

2018

- Extended lifecycle concept and implementation for ROS 2 and micro-ROS.
- Modeling concept to specify system hierarchy as well as system modes of systems, subsystems, and their mapping along the system hierarchy down to nodes.
- Mode inference and mode manager in C++ for ROS 2.

2019

- Specific implementation of mode manager for micro-ROS as may be necessary.
- Diagnostics framework for micro-ROS, interoperating with ROS 2 diagnostics.
- MCU-specific diagnostics functions for resource usage on RTOS layer, latencies, statistics from middleware, etc.
- Integration of mode manager with real-time executor and/or roslaunch.

2020

- Lightweight concept for specifying error propagations between nodes and subsystems.

Note: The extension of the ACTIVE state by modes (substates) was originally planned for 2020 but brought forward in 2018.

3.6 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).

4 Annex 2: README.md from System Modes Package

Content of https://github.com/microROS/system_modes/blob/master/system_modes/README.md from 14th December 2018.

The system modes concept assumes that a robotics system is built from components with a lifecycle. It adds a notion of (sub-)systems, hiararchically grouping these nodes, as well as a notion of *modes* that determine the configuration of these nodes and (sub-)systems in terms of their parameter values.

A list of (current and future) requirements for system modes can be found here: requirements.

4.1 System Modes Package

The system modes concept is implemented as a package for ROS 2. This package provides a library for system mode inference, a mode manager, and a mode monitor.

4.1.1 System Modes Library

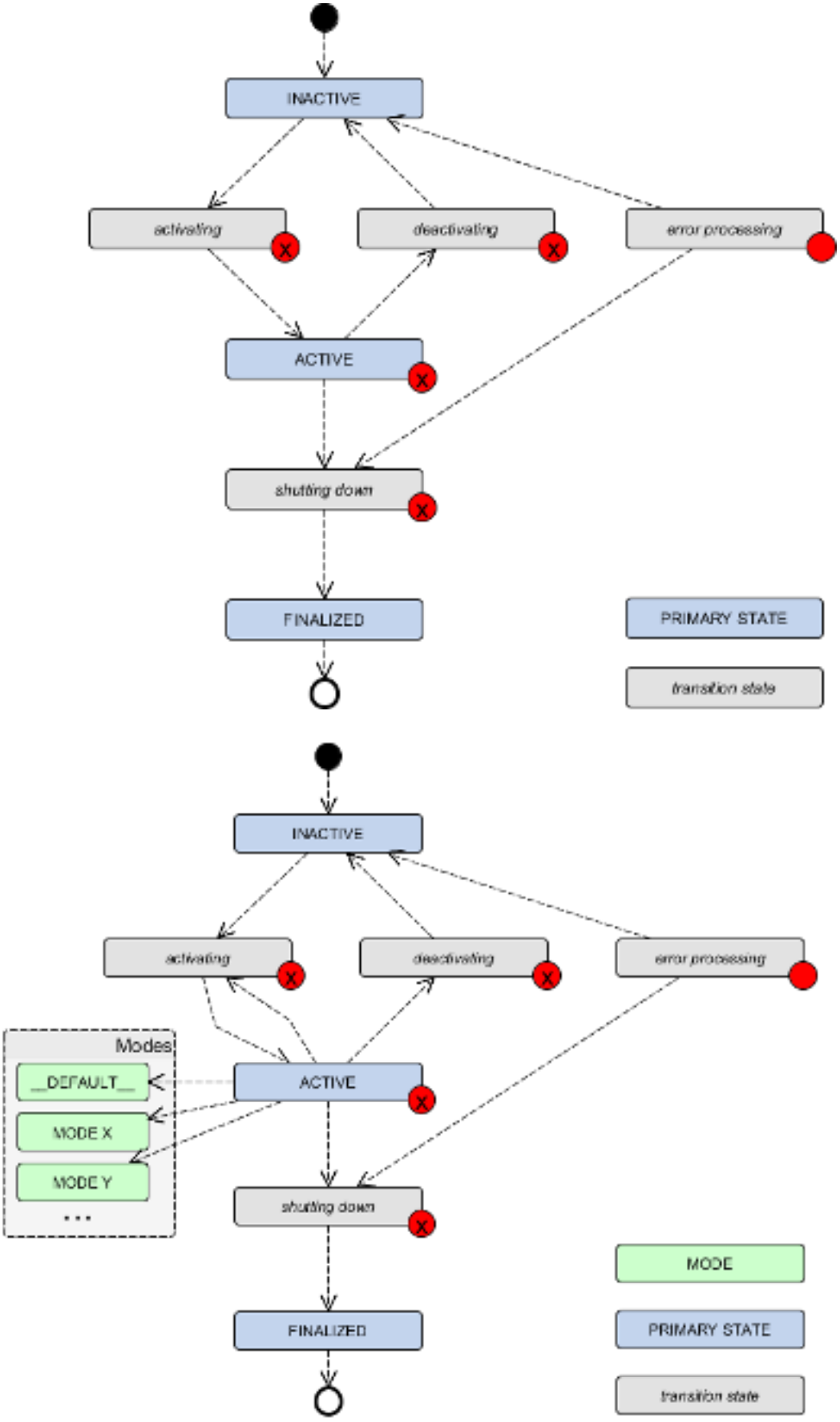
The library consists of the parser of the system modes model and the mode inference mechanism.

4.1.1.1 Hierarchal System Modeling We assume the system to comprise of loosely-coupled - potentially distributed - components with a runtime lifecycle; hereinafter referred to as '*nodes*'. We call semantic grouping of these nodes a *(sub-)system*. We assume that these (sub-)systems can again be hierarchically grouped into further (sub-)systems, see system-of-systems. All nodes and (sub-)systems that belong to a (sub-)system *S* are referred to as *parts* of (sub-)system *S*.

The introduced notion of (sub-)systems does not refer to a concrete software entity, but rather a *virtual* abstraction that allows efficient and consistent handling of node groups.

Note: In a first stage of this concept, we assume that the entire system can be specified up-front. Later revisions of this concept might have to take care of changing systems, i.e. further nodes and/or (sub-)systems joining at runtime.

4.1.1.2 Lifecycle Within this package, we assume that nodes are ROS 2 Lifecycle Nodes. We additionally establish the same lifecycle for the (sub-)systems introduced above. Hence, all *parts* of a system can be assumed to have the same lifecycle.


INACTIVE
activating
deactivating
error processing
ACTIVE
shutting down
FINALIZED
PRIMARY STATE
transition state
INACTIVE
activating
deactivating
error processing
Modes
__DEFAULT__
MODE X
MODE Y
...
ACTIVE
shutting down
FINALIZED
MODE
PRIMARY STATE
transition state

The illustration on the left shows a simplified version of the standard ROS 2 lifecycle. It is simplified by grouping the ROS 2 states *unconfigured* and *inactive* (both subsumed under *inactive*). We extend this lifecycle (illustration on the right) by the following aspects:

1. We introduce **modes** that are specializations of the active state, see System Modes.
2. We introduce an optional transition from *active* to *activating* to allow changing the mode without deactivating the node.

4.1.1.3 System Modes

System modes extend the *activate* state of the ROS 2 lifecycle and allow to specify different configurations of nodes and (sub-)systems:

- **Modes of nodes** consist of parameter values.
- **Modes of (sub-)system)** consist of modes of their *parts*.

For example, a node representing an actuator might provide different modes that specify certain maximum speed or maximum torque values. An actuation sub-system, grouping several actuator nodes, might provide modes that activate/deactivate certain contained actuator nodes and/or change their modes based on its own modes.

Both, the system hierarchy as well as the system modes are specified in a system modes and hierarchy model file (SHM file, yaml format) that can be parsed by the mode inference mechanism. The SMH file adheres to the following format:

```
{system}:
  ros__parameters:
    type: system
    parts:
      {node}
      […]
    modes:
      __DEFAULT__:
        {node}: {state}[.{MODE}]
        […]
      {MODE}:
        {node}: {state}[.{MODE}]
        […]
      […]
[…]

{node}:
  ros__parameters:
    type: node
    modes:
      __DEFAULT__:
        ros__parameters:
          {parameter}: {value}
```

```
        […]
      {MODE}:
        ros__parameters:
          {parameter}: {value}
          […]
      […]
[…]
```

(curly brackets indicate placeholders, square brackets indicate optional parts, ellipses indicate repeatability)

The system_modes_examples package shows a simple example consisting of modes for one system and two nodes. The model file of the example can be found here.

4.1.1.4 Mode Inference Since the introduced (sub-)systems are not concrete software entities, their state and mode has to be *inferred* from the states and modes of their parts. This inference mechanism is part of the system modes library and is used by the mode manager and mode monitor that are also included in this package. We can show that system states and modes can be deterministically inferred under the following conditions: 1. Nodes can be asked for their state, mode, and parameters
This is true, since the lifecycle nodes provide the according lifecycle state service (GetState) and the mode manager provides the according mode service (GetMode). 1. *Target* states and modes are known
Before attempting a state or mode change for a system or node, the mode manager publishes information about the request.
The according topics might need to be *latched* in order to allow nodes to do the inference after joining a running system.

4.1.2 Mode Manager

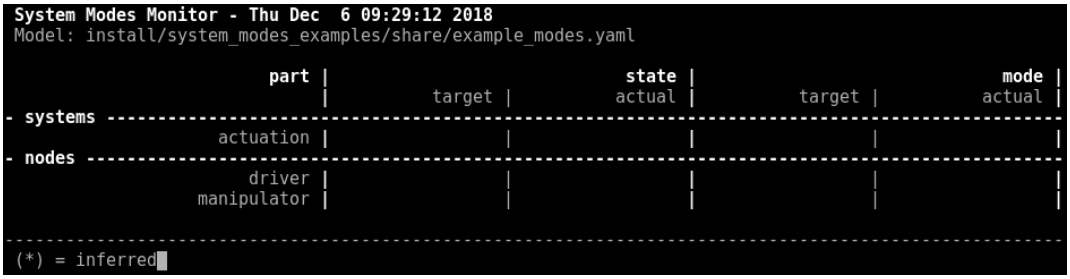
The mode manager is a ROS node that accepts an SHM file (see above) as command line parameter. It parses the SHM file and creates the according services, publishers, and subscribers to manage the system and its modes.

- For (sub-)systems, it mirrors the lifecycle services that are available for ROS 2 lifecycle nodes, i.e.
- `/{system}/get_available_states` - lifecycle_msgs/GetAvailableStates
- `/{system}/get_state` - lifecycle_msgs/GetState
- `/{system}/change_state` - lifecycle_msgs/ChangeState
- For (sub-)systems *and* nodes, it provides similar services for modes, i.e.
- `/{system_or_node}/get_available_modes` - system_modes/GetAvailableModes
- `/{system_or_node}/get_mode` - system_modes/GetMode
- `/{system_or_node}/change_mode` - system_modes/ChangeMode
- Service calls to these services publish information on the requested state change or mode change before attempting them. These are published on the following topics:
- `/{system_or_node}/transition_request_info` - lifecycle_msgs/TransitionEvent
- `/{system_or_node}/mode_request_info` - system_modes/ModeEvent

Running the manager:
$ `mode-manager path/to/modelfile.yaml`

4.1.3 Mode Monitor

The mode monitor is a ROS node that accepts an SHM file (see above) as command line parameter. It continuously monitors and displays the entire system state and mode based on the mode inference introduced above. It monitors the following topics: * `/{system_or_node}/transition_request_info` for all known (sub-)systems and nodes from the model file to monitor their target states * `/{system_or_node}/mode_request_info` for all known (sub-)systems and nodes from the model file to monitor their target modes * `/parameter_events` to infer the current modes for all known nodes based on their parameter values



Running the monitor:
$ `mode-monitor path/to/modelfile.yaml`

5 Annex 3: Requirements.md from System Modes Package

Content of https://github.com/microROS/system_modes/blob/master/system_modes/doc/requirements.md from 14th December 2018.

We discussed our first analysis results with the architect and developers of the above use-case as well as experts for robotics systems and software engineering and derived the following general requirements for the intended concepts and abstractions for (1.) system runtime configuration and (2.) system error and contingency diagnosis.

5.1 System Runtime Configuration

The envisioned approach shall provide concepts for modeling of the system hierarchy with regard to system configuration / modes as well as corresponding framework functionalities.

1. Model
   - **Hierarchical modeling of subsystems**
     - The envisioned approach shall allow to define subsystems consisting of multiple components (ROS nodes, micro-ROS nodes) in a hierarchical or at least two-staged (component - sub-system - system) manner.

  - TODO: Are such hierarchies also relevant for diagnosis/monitoring? If yes, explore whether the same hierarchy should be used.
  - Some robotic frameworks feature meta-components or hierarchical components. We prefer a more flexible approach allowing to define subsystems for each aspect (system modes, scheduling, distribution, …) individually. This approach gives more flexibility - e.g., to treat all device drivers in a uniform manner with regard to scheduling but put the camera driver and the feature and obstacle recognition into one sub-system with regard to system configuration. Also, it allows the application developer to only use mechanisms and framework features that are actually relevant for his system - e.g., distribution might be irrelevant for small robotic systems.
- **Modeling of system, sub-system and component modes**
  - Based on the definition the system, sub-systems and components, the envisioned system runtime configuration concept shall provide an approach for defining individual modes of each of these elements and specify the interconnection between them.
- **Standard but extensible component runtime lifecycle**
  - The component runtime lifecycle (component mode/state) model shall be based on the OMG standard used in ROS 2 and which also resembles the component states used in Smartsoft.
- **Modeling of error propagation and severity within subsystems**
  - Describe causal dependencies between components of a (sub)system with regard to errors. (Sources of inspiration are Fault-Tree Analysis and various academic works.)
- **Integrate component modes/states with component-specific set of parameters into one model**
  - Use of parameter model in style of dynamic reconfigure of ROS1.
  - However, this would introduce a dependency between component lifecycle and component parameters?
- **Timing/causality-aware switching between modes**
  - Two simple patterns in first step: (1.) Reconfigure components of a (sub)system sequentially according to a given list or (2.) reconfigure them simultaneously.

2. Implementation

- **Modeling of system, sub-system and component modes**
  - Strive for lightweight approach by a simple configuration file or even use of C++ API.
  - Should be well manageable (diffable, mergeable) with Git.
- **API primitives for defining preconditions and invariants with regard to system configuration**

3. Runtime/Execution

- **Distributed system state across multiple computing devices (uCs, uPs)**
  - The envisioned approach shall support the communication of the modes/states of subsystems or components located on different computing devices to always obtain a consistent system view.

- It shall also consider that a computing device may reconfigure its sub-systems or components (in a limited scope) independent of the sub-systems and components other computing devices.

5.2 System Error and Contingency Diagnosis

The envisioned approach shall also provide API primitives for detection of typical system errors and contingencies.

1. **Monitors for communication layers**
   - Check for receive rates, latencies, …
2. **Operating system monitors**
3. **Templates and APIs for hardware monitors**
   - Basic blocks for GPIOs (voltage, current)
   - Basic blocks for connected HW modules (via SPI, I²C)
4. **Templates and APIs for monitoring of functional properties**
   - No need of status messages here, but use of function-oriented topics
   - Building blocks/patterns for analysis of time-series (with a sliding window) for bounds, for delays, …

6 Annex 4: README.md from System Modes Examples Package

Content of https://github.com/microROS/system_modes/blob/master/system_modes_examples/README.md from 14th December 2018.

This ROS 2 package provides a simple example for the use of the system_modes package. It contains two ROS 2 LifecycleNodes, a *drive_base* node and a *manipulator* node, as well as simple a model file (yaml).

6.1 Example Model File

The SMH file example_modes.yaml specifies an *actuation* system consisting of the *drive_base* node and the *manipulator* node, system modes for the *actuation* system, as well as system modes for the two nodes: * The *manipulator* node has a default mode, a *STRONG* mode, and a *WEAK* mode, configuring different values for its *max_torque*. * The *drive_base* node has a default mode, a *FAST* mode, and a *SLOW* mode, configuring different values for its *max_speed* and its controller (*PID* or *MPC*). * The *actuation* system comprises of these two nodes. It has a default mode, a *PERFORMANCE* mode, and a *MODERATE* mode, changing the modes of its two nodes accordingly.

6.2 Running the Example

6.2.1 Setup

Until this package provies a proper launch configruation, open 3 terminals to set up your example system: 1. terminal 1: start the *drive_base* node:
$ `drive_base`
1. terminal 2: start the *manipulator* node:
$ `manipulator`
1. terminal 3: start the mode-manager with the provided example model file:
$ `mode-manager --help`
$ `mode-manager system_modes_examples/share/example_modes.yaml`
The mode manager parses the provided SHM model file and creates the necessary services and topics to manage the system modes of the two nodes as well as services and topics to manage the system modes *and* the lifecycle of the *actuation* system.

```
[INFO] [__mode_manager]: System Mode Manager, providing the following services:
[INFO] [__mode_manager]: - system 'actuation'
[INFO] [__mode_manager]:   - actuation/change_state
[INFO] [__mode_manager]:   - actuation/get_state
[INFO] [__mode_manager]:   - actuation/get_available_states
[INFO] [__mode_manager]:   - actuation/change_mode
[INFO] [__mode_manager]:   - actuation/get_mode
[INFO] [__mode_manager]:   - actuation/get_available_modes
[INFO] [__mode_manager]: - node 'driver'
[INFO] [__mode_manager]:   - driver/change_mode
[INFO] [__mode_manager]:   - driver/get_mode
[INFO] [__mode_manager]:   - driver/get_available_modes
[INFO] [__mode_manager]: - node 'manipulator'
[INFO] [__mode_manager]:   - manipulator/change_mode
[INFO] [__mode_manager]:   - manipulator/get_mode
[INFO] [__mode_manager]:   - manipulator/get_available_modes
```

In an additional fourth terminal, start the mode-monitor to see the system modes inference in action: * $ `mode-monitor --help`
* $ `mode-monitor system_modes_examples/share/example_modes.yaml` The monitor updates every second and displays the current lifecycle states and modes of the example system.

```
System Modes Monitor - Thu Dec  6 09:29:12 2018
Model: install/system_modes_examples/share/example_modes.yaml

                      part |                        state |                         mode |
                           |         target |       actual |         target |       actual |
- systems ------------------------------------------------------------------------------------
                 actuation |                |              |                |              |
- nodes --------------------------------------------------------------------------------------
                    driver |                |              |                |              |
                manipulator |                |              |                |              |

------------------------------------------------------------------------------------------------
(*) = inferred
```

Now that you set up the system and you are able to monitor it, play around with it.

6.2.2 Change System States and System Modes

In an additional fifth terminal, you may mimic a planning/executive component to change the state and mode of your system or its components.

1. Start by initializing your system to inactive. The ROS 2 command
   $`ros2 service call /actuation/change_state lifecycle_msgs/ChangeState "{node_name:`

`'actuation', transition: {id: 1, label: configure}}"`
will call the according service on the mode manager, which will change the state of the two nodes to *inactive* accordingly. Observe the console output of the mode manager and the two nodes as well as the mode monitor. The mode monitor should display the following system state:

```
System Modes Monitor - Thu Dec  6 11:12:21 2018
Model: install/system_modes_examples/share/example_modes.yaml

                        part |                          state |                          mode |
                             |         target |        actual |         target |        actual |
- systems ------------------------------------------------------------------------------------
                   actuation |       inactive |    inactive(*) |                |               |
- nodes --------------------------------------------------------------------------------------
                 driver_base |       inactive |       inactive |                |               |
                 manipulator |       inactive |       inactive |                |               |

------------------------------------------------------------------------------------------------
(*) = inferred
```

2. Activate your system with the following ROS 2 command:
   `$ ros2 service call /actuation/change_state lifecycle_msgs/ChangeState "{node_name: 'actuation', transition: {id: 3, label: activate}}"`
   To change the *actuation* system into active and its default mode (since no explicit mode was requested), the mode manager will set the *drive_base* to active and leave the *manipulator* inactive, as specified in the model file. The mode monitor should display the following system state:

```
System Modes Monitor - Thu Dec  6 11:13:06 2018
Model: install/system_modes_examples/share/example_modes.yaml

                        part |                          state |                          mode |
                             |         target |        actual |         target |        actual |
- systems ------------------------------------------------------------------------------------
                   actuation |         active |      active(*) |     __DEFAULT__ |  __DEFAULT__(*) |
- nodes --------------------------------------------------------------------------------------
                 driver_base |         active |         active |     __DEFAULT__ |  __DEFAULT__(*) |
                 manipulator |       inactive |       inactive |                |               |

------------------------------------------------------------------------------------------------
(*) = inferred
```

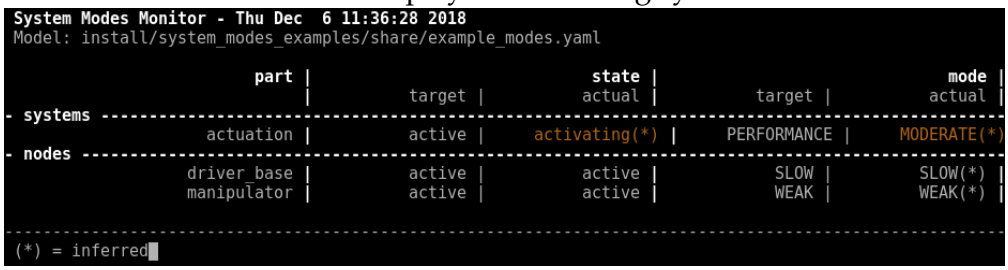
3. Set your system into *PERFORMANCE* mode with the following ROS 2 command:
   `$ ros2 service call /actuation/change_mode system_modes/ChangeMode "{node_name: 'actuation', mode_name: 'PERFORMANCE'}"`
   To change the *actuation* system into its *PERFORMANCE* mode, the mode manager will change the *drive_base* to *FAST* mode and activate the *manipulator* node in its *STRONG* mode. The mode monitor should display the following system state:

```
System Modes Monitor - Thu Dec  6 11:14:13 2018
Model: install/system_modes_examples/share/example_modes.yaml

                        part |                          state |                          mode |
                             |         target |        actual |         target |        actual |
- systems ------------------------------------------------------------------------------------
                   actuation |         active |      active(*) |    PERFORMANCE |  PERFORMANCE(*) |
- nodes --------------------------------------------------------------------------------------
                 driver_base |         active |         active |           FAST |        FAST(*) |
                 manipulator |         active |         active |         STRONG |      STRONG(*) |

------------------------------------------------------------------------------------------------
(*) = inferred
```

   Note, that the system state and mode as well as the node modes are indicated to be *inferred*, as explained in the mode inference section of the system_modes package.
4. You can further play around with the mode inference. For example, change the mode of the two nodes explicitly so that the target mode and actual mode of the *actuation* system diverge. Execute the following two ROS 2 commands:
   `$ ros2 service call /drive_base/change_mode system_modes/ChangeMode "{node_name: 'drive_base', mode_name: 'SLOW'}"`
   and

```
$ ros2 service call /manipulator/change_mode system_modes/ChangeMode "{node_name:
'manipulator', mode_name: 'WEAK'}"
```

The mode monitor should display the following system state:



Note, that the mode monitor is able to infer that the system's *actual* mode is now *MODERATE*. This is based on the fact that both its nodes are active, the *drive_base* is in its *SLOW* mode, and the manipulator is in its *WEAK* mode. However, the last requested mode for the *actuation* system is *PERFORMANCE*, so the monitor infers that the system is still transitioning into its target mode, indicating that the actual system state is *activating* (see lifecycle).