Abstract—High-level deliberation in robotic systems has to handle three different but closely interwoven aspects simultaneously: intended tasks, contingencies, and system-level errors. To reduce the complexity, we propose the system modes concept, to abstract runtime state information and reconfiguration of the software components of the underlying layers by a model-based approach. The proposed concept introduces a notion of hierarchically composed, virtual subsystems as well as a notion of modes that determine their configuration. It features an inference engine to deduce the modes of the subsystems from the components and top-down reconfiguration mechanisms. Repetitive and fine-grained communication between high-level deliberation and underlying software components can thereby be reduced, decreasing unwanted coupling between system parts.

I. INTRODUCTION

Modern robotic software architectures follow a layered approach. The layer with the core algorithms for SLAM, motion planning, navigation, object recognition, 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 often referred to as executive layer or task layer, possibly with a planning layer or mission layer on top. In the following, we use the term deliberation layer and do not consider any substructure of this layer. We observe three aspects to be handled on the deliberation layer:

1) Intended tasks: The straight-forward, error-free control flow to achieve a goal or to accomplish a mission.
2) Contingencies: Handling of expectable contingencies in the interaction with the real world such as blocked paths in navigation, loss of localization, low battery, and poor light conditions.
3) System-level errors: System-related errors and exceptions such as sensor failures, actuator dropouts, and software component crashes.

These aspects are closely interwoven and have to be treated simultaneously, which is a root cause for the complexity in robot deliberation.

A key to reduce this complexity and to avoid complicacy is to hide or abstract technical aspects of the skills and details of their implementation in software components as far as possible. The middleware mechanisms used in modern robotic software frameworks play a vital role for such transparency and abstraction. The deliberation layer can invoke services and actions or communicate set values to the skills without any knowledge about the component structure. The same applies to listening on feedback on long-running service calls or on messages on events in the environment.

However, as illustrated in Fig. 1, there is a gap when it comes to activating/deactivating of skill-level functions, their parameterization, runtime states, and diagnostics. This applies to the Robot Operating System (ROS) in particular, but also to other robotic software frameworks.

The goal of the system modes concept is to close this gap by providing suitable abstractions and framework functions for (1.) system runtime configuration and (2.) system error and contingency diagnosis. In detail, our contributions are:

- Requirements analysis: We report on the analysis of the deliberation layer of an industrial robotics project and derive requirements to the system modes concept.
- Concept: We propose a suitable concept based on the runtime component lifecycle model of ROS 2, also known as Managed Nodes.
- Implementation: We present an open-source implementation of the System Modes concept for ROS 2.

The remainder of this paper is structured as follows: We discuss requirements in Sect. II and related work in Sect. III followed by the presentation of the System Modes concept in Sect. IV and its implementation for ROS 2 in Sect. V. We report on a practical evaluation with the ROS 2 Navigation stack on a Tiago robot in Sect. VI before concluding the paper in Sect. VII.
II. REQUIREMENTS ANALYSIS

We analyzed the hierarchical task network developed in a robotics project at Bosch as a deliberation layer for an autonomous intralogistics platform. In this project, a common runtime lifecycle similar to the lifecycle proposed by the OMG Robotic Technology Component Specification [1] had been already defined for all software components of the system. We discovered a couple of repeating patterns, including (1.) activating/deactivating fixed subsets of the components, (2.) handling errors or contingencies that affect dependent components, (3.) reconfiguring multiple components simultaneously, and (4.) reacting on failures during reconfiguration.

In close cooperation with the developers, we derived the following requirements to the system modes concept to reduce the complexity of the deliberation layer in general and the occurrence of such repeating patterns in particular.

A. Model

- **System hierarchy:** The model shall allow defining subsystems consisting of multiple components in a hierarchical or at least two-staged manner.
- **Runtime states:** The model shall allow defining individual states of each of these elements (subsystems, components) and specifying the relation between them.
- **Standard but extendable states:** These states shall follow a common standard for consistency and homogeneity but shall at the same time be extendable to application-specific needs.
- **Error propagation:** It shall be possible to model causal dependencies between the elements of a subsystem and in particular the propagation of errors. Sources of inspiration are Component Fault Trees that allow a Fault Tree Analysis for component-based systems [2].
- **Timing/causality-aware switching between states:** Two common patterns in first step: (a) Reconfigure components of a subsystem sequentially in a fixed order or (b) reconfigure them simultaneously.

B. Implementation

- **Textual model:** The model should be well manageable with standard version control systems.
- **Transparency:** The System Modes concept should be transparent for the components, i.e. not require any changes to existing (legacy) components.
- **Compatibility:** The APIs shall be compatible with existing component runtime lifecycle mechanisms.

C. Runtime

- **Distribution:** The System Modes approach shall support distributed setups with multiple micro controllers or microprocessors.
- **Tolerance to communication faults:** The approach shall support (limited) local error handling in case of communication faults between these devices.
- **Introspectability:** The states shall be introspectable for developers, ideally with standard tools.

III. RELATED WORK

In the Architecture Analysis & Design Language (AADL), a mode is defined as a "visible operational state" of a component with "mode-specific properties and configurations of subcomponents and connections that are active in specific modes" [3][Ch. 7]. It further distinguishes between operational modes and fault-tolerance modes. NASA's Remote Agent [4] proved that mission flexibility and resilience can be improved by using declarative models of modes to reconfigure systems for failure diagnosis and recovery. Continuation of that work, successfully flying during a 17 years mission, confirms the benefits of using explicit models for run-time configuration and adaptation [5], explored in this work.

Hernández et al. [6] propose a self-adaptation framework, modeling functional knowledge for augmented autonomy in an ontology termed Teleological and Ontological Model for Autonomous Systems (TOMA5ys). In TOMA5ys, Function Designs model the potentially many different robot’s capabilities that map to different Function Groundings that realize these capabilities. The System Modes concept proposed in this work is capable of serving as realization of Function Groundings as demonstrated in Sect. [VI].

Leng et al. [7] introduced the DyKnow framework with similar concepts for ROS 1. DyKnow supports reconfiguration by introducing so-called Transformation Specifications that describe different usages of ROS 1 nodelets based on different configurations. A managing component keeps track of transformation specifications, in addition to instantiated nodelets and their connections to allow knowledge based (self-)adaptation in ROS-based robotic systems.

IV. SYSTEM MODES

In this section, we present the System Modes approach for the introduced requirements.

A. Assumptions

We assume the system to be comprised of loosely-coupled – potentially distributed – components with a runtime lifecycle, hereinafter referred to as nodes. We call semantic grouping of these nodes a system. We assume that these systems can again be hierarchically grouped into further (sub-)systems (system-of-systems). All nodes and systems that belong to a certain system are referred to as parts of this system.

We further assume that nodes can be asked for their current state and current parameter values. Target states of nodes and systems can be known as well, either by requesting those or caching according requests.

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

B. Model

1) **Hierarchical System Modeling:** The System Modes approach adds a notion of systems, hierarchically grouping these nodes, as well as a notion of modes that determine
the configuration of these nodes and systems in terms of their parameter values.

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

2) Lifecycle: For the sake of illustration, we assume that nodes are ROS 2 Lifecycle Nodes. We then extend the ROS 2 default lifecycle by the following aspects:

- We introduce modes that are specializations of the ACTIVE state, see Fig. 2.
- We additionally establish the same lifecycle for the systems introduced above. Hence, all parts of a system can be assumed to have the same lifecycle.

3) Modes: Concrete system modes (or simply: modes) extend the ACTIVE state of the ROS 2 lifecycle and allow to specify different configurations of nodes and systems:

- modes of nodes specify parameter values.
- modes of systems specify 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.

C. Mode Inference

Since the introduced systems are not concrete software entities, their state and mode has to be inferred from the states and modes of their parts according to the model. This inference mechanism is part of the system modes library and is used by the mode manager and mode monitor. System states and modes can be deterministically inferred under the following conditions:

- Nodes can be asked for their state, mode, and parameters.\(^1\)
- 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.\(^2\)

We used the SCODE-ANALYZER [8] to verify that these rules are complete and consistent, thus resulting in an unambiguous mode inference.

Fig. 3 shows an example, where parameters of nodes are observed and the according modes of these nodes can be inferred. This is true for exact matches of parameter values as well as parameter ranges, if these don’t render mode specifications ambiguous. Based on the inferred modes of the nodes, the modes of the systems can be inferred.

D. Mode Management

While the mode inference allows to infer the current actual state of the system, mode management allows to manage and change the system state. To be able to work with standard components as much as possible, per our requirements (Sect. II), we don’t require nodes to know about system modes, i.e., nodes don’t need to know their modes and provide according mode change services.

Instead, a mode management instance maintains the model of the system and its modes based on the SMH file and manages nodes for all specified systems and nodes. As depicted in Fig. 3, if the system is changed into a certain mode, the mode management will i) first set the modes of the sub-systems accordingly, then ii) set set the modes of the nodes accordingly, and last iii) set parameters of the nodes in accordance to these.

\(^1\)True for ROS 2, since lifecycle nodes provide the according lifecycle service and the mode manager provides the according mode service.

\(^2\)In ROS 2, the according topics might need to be latched in order to allow nodes to do the inference after joining a running system.
Fig. 4. Screenshot of the mode monitor, showing the observed resp. inferred (see asterisk (*) modes of one system and its two parts (nodes).

E. Error Handling and Rules

In case of contingencies or system errors, recovery strategies are often similar: if a component crashes, either a restart of the component is attempted or the system is put into a degraded mode, which can operate without the crashed component or failed sensor/actuator.

Mode inference and mode management allow to detect and react to these situations under the conditions stated in Sect. IV-C. We propose a simple syntax to specify common reactions: 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 == (specific state/mode)
then:
 system.target := (specific state/mode)

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

This simple rule set allows reactions bottom-up reactions and recoveries from low-level system failures, e.g., sensor failure leading to degraded mode of the perception systems leading to slow mode of the entire system.

V. IMPLEMENTATION AND AVAILABILITY

A first version of the system modes concept was implemented in C++ and is available for ROS 2. While the algorithmic core of the system modes is not ROS-specific, the library uses software concepts and data structures provided by ROS 2, introducing a software dependency to ROS 2. The system modes implementation consists of three main parts: the core library, a mode monitor, and a mode manager.

A. System Modes Library

The library consists of the mode inference mechanism, as well as basic mode management and mode monitoring capabilities. It relies on the software concepts and data structures provided by ROS 2, introducing a dependency to ROS 2, but its algorithmic core is not ROS-specific.

Both, the system hierarchy as well as the system modes are specified in a System Modes and Hierarchy (SMH) model file (yaml format) that is parsed by the mode inference.

The SMH file adheres to the following format, curly brackets indicating placeholders, square brackets indicating optional parts, ellipsis indicating repeatability (shown without error handling rules for the sake of brevity):

```yaml
(system):
  type: system
  parts:
    [node]
    [...
    modes:
      __DEFAULT__:
        [node]: [state][{MODE}]
        [...
        [MODE]:
          [node]: [state][{MODE}]
          [...
          [...
          [...
          [...
    [node]:
      type: node
      modes:
        __DEFAULT__:
          [parameter]: [value]
          [...
          [MODE]:
            [parameter]: [value]
            [...
            [...
            [...

In this model, each node is specified with its named modes. Each mode of a node is defined by a concrete node configuration, i.e., a set of parameter values. A node can have an unlimited number of modes.

Additionally, an arbitrary number of named systems can be defined, each defined with a list of its parts (other systems or nodes) and its named modes. Each mode of a system is defined by the configuration of its parts, i.e., a set of target state/mode pairs of its parts.

The SMH file can be conveniently generated by the MROS modeling tool to specify reconfigurable skills in ROS [9].

B. Mode Monitor

The mode monitor is realized as a ROS 2 node, including the mode inference, observing relevant topics, and providing a visual overview of the system and its modes, shown in Fig. 4.

The mode monitor subscribes to all state change topics, mode change topics, and parameter change topics of all systems and nodes modeled in the SMH file. Information gathered from these topics are handed to the mode inference, which maintains a consistent, up-to-date model of the current system state. The mode monitor additionally provides a simple user interface to inspect this current state.

C. Mode Manager

The mode manager is realized as a ROS 2 node, including the mode inference and providing services to manage the extended lifecycle. It subscribes to the same topic as the mode monitor to maintain a consistent, up-to-date model of the current system state based on the mode inference. It additionally provides the following services for the extended lifecycle of all systems and nodes from the SMH file:

1) Lifecycle services (get state, get available states, change state) for all systems from the SMH file. It thereby
exposes the same lifecycle interface for systems that standard ROS 2 nodes expose.

2) Mode services (get mode, get available modes, change mode) for all systems and nodes from the SMH file.

By providing the standard ROS 2 lifecycle services for systems and providing the same mode service for systems and nodes, we expose the exact same extended lifecycle interface for systems and nodes, providing a consistent abstraction for the deliberation layer. I.e., the deliberation layer configures a skill, it no longer needs to know if it is implemented by a node or a system of nodes.

D. Packaging and Availability

System modes are available in the ROS 2 system_modes package, complemented by an examples package (system_modes_examples). It was developed in the context of micro-ROS, where it is integrated with a C-based implementation of the ROS 2 lifecycle for microcontrollers.

VI. CASE STUDIES AND METACONTROL

Apart from their usage in the micro-ROS project, integration of System Modes with an ontology-based control framework called metacontrol was successfully demonstrated in the course of the MROS project. System Modes were evaluated in two case studies: a mobile robot navigation scenario as well as a Bosch consumer robot prototype (undisclosed).

A. Scenario

Fig. 6 shows the software architecture of the first scenario software architecture, composed of the following elements:

- At the Mission level, a behavior tree is used to command the robot to navigate various waypoints sequentially. In MROS, the mission level does not directly send the navigation action to the level that implements the navigation skill, but instead an intermediate element called MetaController. This action contains the desired quality of service for the task.

- The MetaController receives the action and forwards it to the navigation level, extracting the required quality of service. This level contains a reasoner, which establishes an appropriate system configuration depending on the required quality of service, observation of the system’s state, and further factors.

- The navigation level uses Nav2, the ROS 2 navigation system. The bt_navigator orchestrates the navigation process, asking the Planner to generate a route to the waypoint, sending the route to the controller to send the speed commands to the robot, and asking the Recovery Server to act in case of errors or contingencies. AMCL implements localization on the map.

- Nav2 requires information from a Laser to locate and avoid obstacles. This reading can be provided by a laser, which offers a 360° range. Alternatively as a fall-back, a virtual laser reading can be obtained by processing the information from an RGBD Camera with a 58° range.

B. System Modes and Contingencies

In the evaluated scenario, two contingencies could occur that the MetaController solves by requesting system reconfiguration through a change of modes. Fig. 6 shows the available modes of the system, and further factors. The contingencies handled based on these modes are:

- Battery low: When the battery level is low, the MetaController requests that the energy_saving_mode is activated. This mode slows the robot down (see Fig. 7) via controller parameterization and uses a behavior tree to navigate. In addition, it raises a battery status warning and informs the mission level about alternative plans, like aborting the itinerary and navigate to a recharge point.

- Laser failure: If the laser fails, the MetaController requests that the degraded_mode is activated. This mode activates the node that processes virtual laser readings from the RRGBD camera. As the range of readings is narrower, the robot’s speed must decrease to avoid encountering obstacles during turns. Also, being less precise than the laser, AMCL parameters are reconfigured to rely more on odometry than the laser perception.

In addition to these contingencies, further system configurations are associated with the quality of service that may be required from the mission level.

https://github.com/micro-ROS/system_modes
https://micro-ros.github.io/docs/concepts/client_library/system_modes/
https://github.com/micro-ROS/rclc/tree/master/rclc
MROS - Metacontrol for ROS 2 systems: https://robmosys.eu/mros
The presented scenario demonstrates System Modes’ capability to adapt a system to contingencies and quality of service by allowing a reconfiguration of the system at runtime, including the activation or deactivation of nodes, Fig. 8 shows the evaluation carried out on a professional Tiago robot.

VIII. DATA AVAILABILITY

The ROS 2-based source code for the mobile robot navigation case study discussed in Sect. VII is publicly available for replication, tagged with ‘rose2021’ in the following repositories:

- https://github.com/micro-ros/system_modes

ACKNOWLEDGMENTS

This work was supported by the EU-funded projects OFERA (micro-ROS) under Grant Agreement No. 780785 and RobMoSys ITP MROS under Grant Agreement No. 732410.

REFERENCES


https://www.youtube.com/watch?v=j67xXNIdRkQ