To appear in *Advanced Robotics*
Vol. 00, No. 00, Month 201X, 1–20

FULL PAPER

MROS: Runtime Adaptation For Robot Control Architectures

Darko Bozhinoski[a*], Mario Garzon Oviedo[b], Nadia Hammoudeh Garcia[c], Harshavardhan Deshpande[c], Gijs van der Hoorn[b], Jon Tjerngren[d], Andrzej Wąsowski[e] and Carlos Hernandez Corbato[b]

[a]IRIDIA, Artificial Intelligence Lab, Université Libre de Bruxelles; [b]IRIDIA, Artificial Intelligence Lab, Université Libre de Bruxelles; [c]Fraunhofer Institute for Manufacturing Engineering and Automation IPA; [d]ABB Corporate Research Center Sweden; [e]IT University of Copenhagen;

Known attempts to build autonomous robots rely on complex control architectures, usually implemented with the Robot Operating System (ROS). Runtime adaptation is needed in these systems, to cope with component failures and with contingencies arising from dynamic environments—otherwise these affect the reliability and quality of the mission execution. Existing proposals on how to build self-adaptive systems in robotics usually require a major re-design of the control architecture and rely on complex tools unfamiliar to the robotics community. Moreover they are hard to reuse across applications.

This paper presents MROS: a model-based framework for run-time adaptation of robot control architectures based on ROS. MROS uses a combination of domain specific languages to model architectural variants and capture mission quality concerns, and an ontology-based implementation of the MAPE-K and meta-control visions for runtime adaptation. The experiment results obtained applying MROS in two realistic ROS-based robotic demonstrators show the benefits of our approach in terms of the quality of the mission execution, and MROS's extensibility and re-usability across robotic applications.

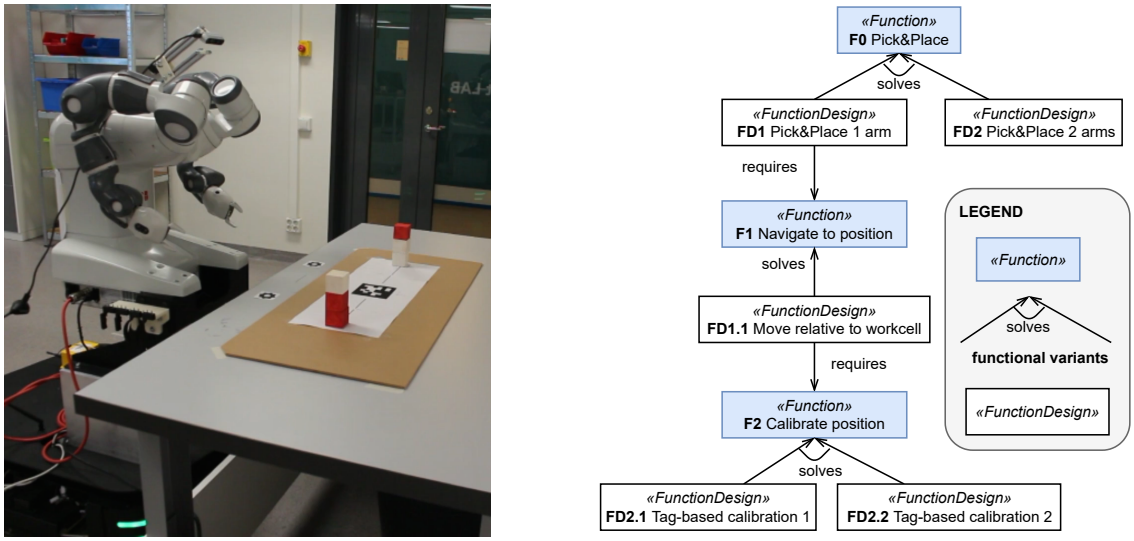
Keywords: self-adaptive systems, models-at-runtime, variability, autonomous robots, control architecture, ontologies

1. Introduction

Robotic systems are designed by integrating and configuring individual robot capabilities in a governing control architecture, most frequently based on the Robot Operating System (ROS) [1]. However, static control architectures fall short when addressing context variability in open-ended, dynamic environments, where internal errors also compromise the quality and autonomy of mission execution. Self-adaptive Systems methods offer various solutions to deal with the context variability resulting from uncertainty and contingencies during execution[2]. However, existing approaches to apply these methods in robotic systems are marginal, because they require a major re-design of the control architecture, they use complex tools unfamiliar to the robotics community, and they are hard to reuse across applications. Therefore, there is a pressing demand for solutions to integrate self-adaptation in robot control architectures. This paper demonstrates how model-based self-adaptation can be easily integrated in robot control architectures to increase mission reliability and quality.

Robotics researchers have come up with sophisticated control architectures that are able to perform very well a specific mission [3, 4]. However, these architectures are mission and robot

*Corresponding author. Email: darko.bozhinoski@ulb.be



(a) The mobile manipulator at the workcell where it has to build a pyramid with blocks.

(b) TOMASys model of the control architecture of the mobile manipulator.

Figure 1.: Mobile Manipulator based on an ABB YuMi robot.

platform specific and are unable to address other missions and robots without undergoing major modifications [5].

An example of a robot performing a complex task is the mobile manipulator shown in Figure 1a, which has capabilities to pick&place, navigate and calibrate its position (called functions (F) which are realized through function designs (FD) in the architecture model in Figure 1b, which will be explained in Section 2.2). The robot uses AprilTag[1] fiducial system to calibrate its position in the work cell, and the wooden blocks with which it builds a pyramid. The original ROS-based control architecture suffered from errors locating the fiducial tag in varying light conditions, and from task failures whenever an arm bumps lightly into any obstacle (the internal safety function prevents any further motion of that arm). However, the system offers redundancy that can be exploited for self-adaptation. The challenge is to integrate self-adaptation into its architecture without major development effort, and maintaining extensibility to other tasks (e.g. navigation between the workstations), so that it can be reused for other robots.

We claim that using a model-based solution to integrate self-adaptation in robot control architectures addresses this challenge and results in increased mission quality and reliability at a lower cost. Modeling has been successfully applied in various domains to solve issues of architectural composition both statically (software product lines [6]) and at runtime (dynamic software product lines [7], models-at-runtime [8, 9]). Using models of the robot control architecture raises the abstraction level in the implementation of robot control systems, improving reuse from system to system, which is presently rare and often *ad hoc*. While work has been done in this direction [10], given the complexity of robotics systems, it is unclear how to proceed to maximize the benefits, and to avoid redesigning and re-implementing the adaptation layer for each robot from scratch.

In this work, we propose MROS: a model-based approach for self-adaptation of robot control architectures, contributing two key insights. First, the use of different Domain Specific Languages to model different levels of abstraction allows for lean and extensible tools to design self-adaptation in robotics. Second, an explicit separation of concerns, between mission-specific logic and system management, and between general and mission-specific concerns, allows to reuse adaptation rules and artifacts across missions and robotic systems. For system design, MROS uses a combination of platform-specific Domain Specific Languages (DSLs) to model the robot architecture. These DSLs are close to the ROS ecosystem and effectively capture functional

[1] http://wiki.ros.org/apriltag_ros

and task quality concerns of a robotic system for run-time adaptation. For run time adaptation, MROS realizes the *meta-control* vision[11] using a novel implementation of the MAPE-K loop based on ontological reasoning to drive the run-time adaptations.

A parallel, but equally important, goal of this paper is to advance experiment design for model-based self-adaptation, to push the community from anecdotal evidence towards data. From methodological perspective, we propose to evaluate self-adaptation frameworks at two levels: (i) how well, and at what computational cost, a given framework improves mission performance and reliability, and (ii) how reusable (vs idiosyncratic) a framework is; to how large class of systems it applies, and how much specialized development is needed. We evaluate the former in a quantitative experiment, and the latter in a qualitative assessment based on engineering experience from the project, and its architectural qualities.

Our main contributions include:

- *A reusable implementation of MROS meta-controller,* ready to integrate self-adaptation in ROS systems.
- *A set of design tools and languages* to model and deploy the MROS solution in ROS systems.
- *An evaluation of MROS in two case studies (mobile manipulator, factory floor navigation)* exceeding prior methodological standards. The experiments show performance and robustness improvements, and quantify the computational cost of the general architecture.
- *An analysis of benefits of model-driven methods for this problem.*

Design methodologies for autonomy and self-adaptation remain an open discussion topic in the robotics community. With this paper, developed in a joint effort of robotics and modeling researchers, we hope to get more members in the modeling community interested in the problem. We also contribute a validated approach and artifacts to the discussion of systematic design methods and reusable architectural components for robotics. We believe that the integration of Meta-control with ROS may bring models-at-runtime and system modeling methods to broader groups of practitioners in robotics.

The rest of the paper is organized as follows. Section 2 gives background information for model-based development and ROS, and self-adaptation using meta-control. Section 4 presents the two robotic demonstrators developed to validate our solution. Section 5 defines our experimental setup and results regarding the run time feasibility of MROS on both demonstrators. Section 6 discusses how MROS supports system extensibility and reusability, Section 7 discusses related work and, Section 8 presents the conclusions and future research direction.

2. Background

2.1 *ROS and the RosModel tooling*

The Robot Operating System is a component-based, open-source platform considered the *de facto* standard for the development of robotics systems, and the platform chosen to develop this work. A key to the success of ROS lies in imposing few architectural constraints and in offering many tailored components (robotics-specific functional components and hardware drivers). A typical ROS system consists of many small and mostly independent distributed programs called *nodes* that implement the robotic capabilities. Nodes communicate via message passing, using a publish-subscribe mechanism, or service calls. Figure 2 shows as an example the runtime communication graph of the ROS Navigation package, the most used ROS library that implements the navigation capability.

To facilitate model-based development with ROS, Hammoudeh Garcia and colleagues developed the *RosModel tooling* framework which contains a set of languages to formalize relevant properties of a ROS system architecture, in a manner compatible with model-based ecosystem [12]. Our paper uses two of these languages:

- *RosModel* model: specifies the communication interfaces of each node (the ports offered by

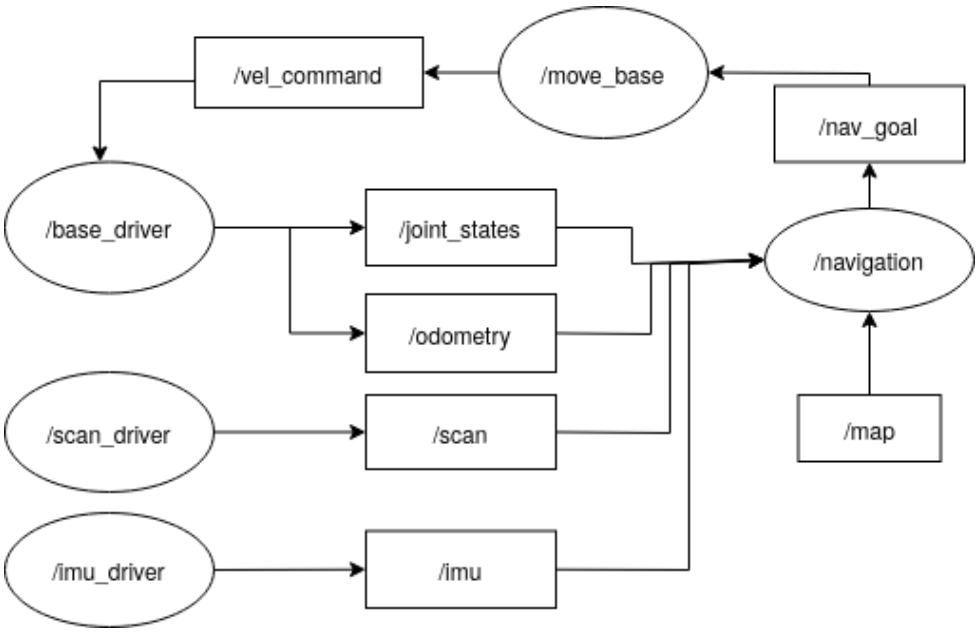


Figure 2.: ROS graph for a simple navigation application, extracted from a system at runtime by standard ROS tools. Ellipses denote nodes, boxes mark topics.

a node) and the deployment information (the ROS package of a node and the packages it depends on)

- *RosSystem* model: reflects the composition of the instantiated nodes, how they are wired and configured.

Figure 3 shows the (static) *RosSystem* for the (runtime) example of Fig. 2, in the *RosModel tooling* graphical editor. Nodes are defined as components that contain a set of interfaces, with properties *name* and a reference to the definition of the instantiated interface. The format also captures connections between the interfaces. Associated tools validate all the connections (i.e. the match of the type of the message sent by the publisher and expected by the subscriber) and automatically generate a *launch file*, an XML script used in ROS to start, configure, and connect all the nodes that constitute the robotics system. The tooling offers static analyzers and runtime introspectors that allow scaffolding the models from an existing software implementation.

2.2 *Meta-control and the TOMASys Meta-model*

Meta-control [11, 13] is a reference architecture for self-adaptive autonomous control systems inspired by MAPE-K (*Monitor-Analyze-Plan-Execute over a Knowledge base*, [14]) and supervisory control [15]. At its center is the *Teleological and Ontological Model for Autonomous Systems* (TOMASys) that allows to model the architecture of component-based systems, including architectural variations at both design-time and run-time [11]. Architectural models provide the appropriate level of abstraction to manage adaptation at runtime [16]. TOMASys captures the relation between the system's requirements and their allocation to the system components through the concept of functional and physical architectures from systems engineering [17, 18]. In the Meta-control vision, a Meta-controller component implements runtime adaptation using the TOMASys model of the system and automatic reasoning for the evaluation, selection, and deployment of architectural variants.

In TOMASys, a *function (F)* represents a capability that has been designed in the system, for example "Navigation" in a mobile robot. A *function design (FD)* is a possible realization of a function, an architectural variant able to deliver it. Internally, a function design prescribes a certain structure that delivers the function, i.e. it maps functionally to system structure, through specifications of components and their interconnection. TOMASys allows to model functional decomposition by designating *required* functions for a function design, capturing the dependencies

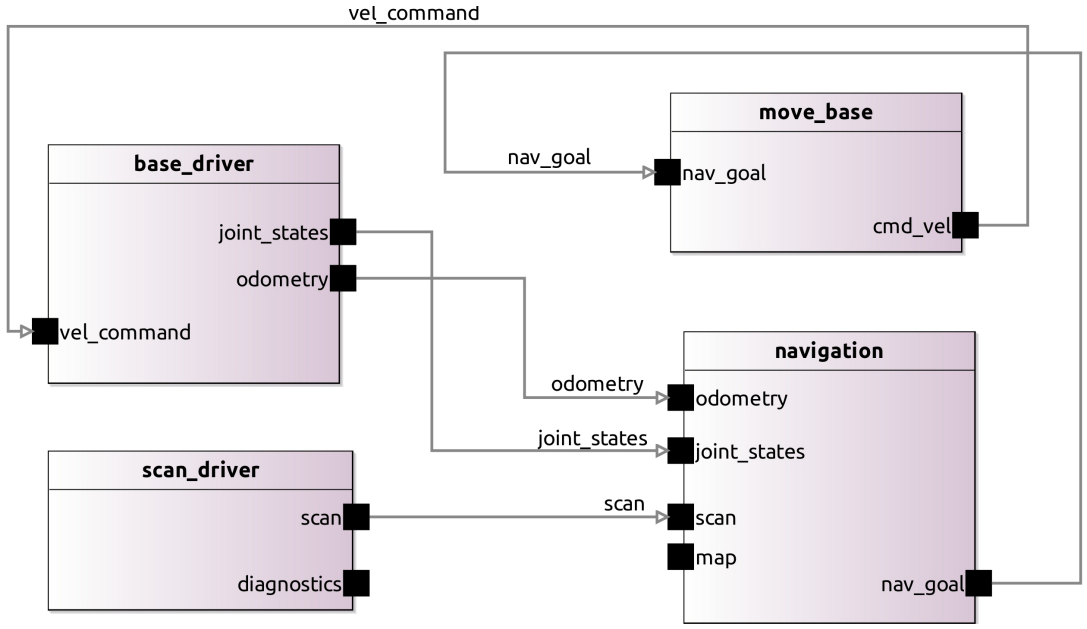


Figure 3.: Example RosSystem model for the system in Fig. 2

between lower- and higher-level functionality. At runtime, the functional requirements are represented as instances called *objectives*, which are solved by instances of the *function designs*, the active function designs called *function groundings*.

A TOMASys model example is shown in Fig. 1a, using UML notation extended with with stereotypes for TOMASys concepts. A high-level function F0 (Pick&Place) represents the capability of the ABB mobile manipulator to pick and place small objects. The model defines two function designs named FD1 and FD2 that solve this function. FD2 represents an architectural variant that solves Pick&Place using both arms of the robot (which jointly can reach the entire workspace), and it is implemented as a monolithic program in the ABB programming language. FD1 represents a variant that solves the function using only one arm and the ability of the mobile base to move and approach objects if they are out of reach of the arm. It is implemented as a set of ROS nodes specifically parametrized to solve F0, and it depends on function F1 (Move position) implemented by function design FD1.1 (Move relative to work cell). FD1.1 in turn requires the function F2 (Calibrate position) delivered by two designs, FD2.1 and FD2.2, which detect the fiducial tag to calibrate the position of the robot in the work cell. FD2.1 and FD2.2 use different camera settings configured for different light conditions. The instantiation or grounding of a FunctionDesign during mission execution is called function grounding. Each function design discussed above can be instantiated during mission execution with corresponding parameters. For example, FD1.1. has a parameter *relative_ distance* that is assigned at runtime.

3. MROS Model-Based Meta-control

MROS is a model-based solution for meta-control that integrates easily in a typical ROS development workflow. It provides a general reusable and extensible meta-control node that extends any ROS system with runtime self-adaptation. Reuse is maximized by separating the general reasoning principles, the adaptation related to system configuration, and error handling from task-related adaptations. Extensibility is achieved by using ontological reasoning to implement the decision-making in the meta-controller. At design time, MROS greatly simplifies the development effort by extending the *RosModel tooling* with a series of plugins that allow ROS developers to model architectural variants and their quality attributes, and to automate the generation and deployment of the meta-controller.

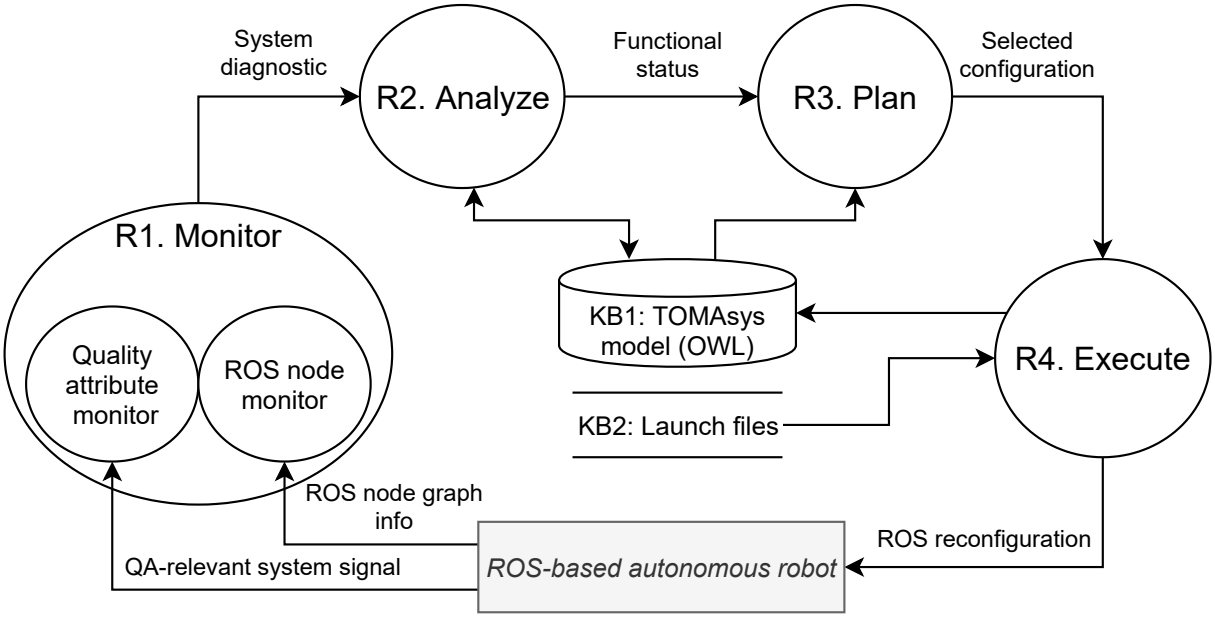


Figure 4.: Data flow diagram for the run-time operation of the MAPE-K loop implemented by the MROS Meta-controller.

3.1 *Runtime Adaptation*

MROS implements a MAPE-K loop using a runtime model conforming to the TOMASys meta-model serving as a knowledge base (KB1). The realization of MAPE-K with ontological reasoning is a core contribution of this work. It allows to separate: (i) the TOMASys model of the robotic system, coded with the Ontology Web Language (OWL), (ii) the adaptation rules, coded with the Semantic Web Rule Language (SWRL), and (iii) the self-adaptation process—a ROS node `mros1_reasoner`. Figure 4 summarizes the main data processing steps in the MROS MAPE-K loop. We discuss them in order below.

Step R1. Monitor

Two monitoring processes track the status of active ROS nodes and the degree of fulfillment of non-functional requirements using quality attributes as a proxy. The goal is to (i) detect and identify parameters misconfigured by developers in ways that lead to sub-optimal robot behavior, (ii) identify erroneous node behavior, and (iii) observe violation of system constraints (i.e. level of quality attributes). The monitors operate at a fixed frequency and produce diagnostics using the standard ROS diagnostic mechanism [19]. This means they easily integrate into an existing system, and might use or provide services already needed in the system, regardless of the presence of a meta-controller.

The *ROS node monitors* leverage *RosModel* tools. The existing `ros_graph_parser` node is used to create a *RosSystem* model of the running ROS-based system. A newly developed `rosgraph_monitor` compares that model with the desired *RosSystem* model of base application (cf. Table 1). It publishes a diagnostic error message in case a difference is detected. The *Quality attribute monitors* are implemented by a separate *observer* node per each quality attribute. They report normalized values for the quality attributes: zero means that the quality attribute is at the required level, one denotes maximum deviation from the required level. The skeleton of these nodes is automatically created from the *RosSystem* model, for the attributes and the system signals (ROS topics) specified by the system architect. Developers need to implement the logic in the observers to obtain the quality attribute level from the given signals. For the evaluation cases presented below, we implemented observers for two attributes: safety (risk of collision with humans, using proximity sensors) and energy (using the runtime battery consumption and a battery model).

Table 1.: Key steps in development of MROS meta-controller.

Step	Input	Design-time activity	Output
1	ROS source code packages	Model the application (reverse engineering and graphical design tools)	*RosSystem* model of base application
2	*RosSystem* model of base application	Model architectural variants (manual)	*RosSystem* models for each variant
3	*RosSystem* model of base application	Generate meta-controller deployment configuration (**automatic**)	(YAML)
4	*RosSystem* models	Generate deployment configuration for variants (**automatic**)	ROS Launch files for each variant
5	*RosSystem* models	Generate observers of the quality attributes (**automatic**)	Observers boilerplate code
6	*RosSystem* models	Generate runtime model for the meta-controller (**automatic**)	*TOMASys model (OWL)*

Step R2. Analyze

The system diagnostic data produced in Step R1 is used to infer the status of the functional architecture of the system. Any new facts, for instance a new component status when a node has reported a fatal error, or a change in the level of a quality attribute, are asserted in the TOMASys model (KB1). Then, ontological reasoning and application-independent rules (declared in SWRL, as proposed in [20]) are applied to the model to infer the current status of the functional architecture, including the status of the TOMASys objectives and the applicability of the alternative function designs. Finally, the status is updated in KB1.

Step R3. Plan

If the meta-controller discovers in Step R2 that any objectives are violated, an architectural adaptation is proposed by searching for alternative function designs in the TOMASys model (KB1). If several available designs meet the required quality attribute levels, the one maximizing an application-specific utility function is selected. The `mros1_reasoner`, used in R2–R3, is implemented with the Owlready2 library [21], the off-the-shelf reasoner Pellet to infer the functional status, and a custom logic for the search and selection of system configuration in the Plan step.

Step R4. Execute

Enforce the adaptation selected in Step R3—stop nodes no longer needed, start new nodes needed, reconfigure the nodes remaining. It is implemented by a general `rosgraph_manipulator`, which follows an adaptation tactic suitable for ROS1, re-deploying the nodes by using launch files created by the *RosModel* tools.

3.2 *Design Time Activities*

The application developer uses the MROS extension of the *RosModel* tools to describe the architecture of the domain application by specifying the set of possible configurations of components and their quality attributes. Table 1 summarizes the steps involved.

Step 1. Model the application

We first create a *RosSystem* model of the base application, starting with individual nodes. Two tools for automatic model generation from source code are available from the *RosModel* project [22]: a built-in plugin that generates the models for ROS packages stored locally, and a

```
RosSystem { Name 'system_a' RosComponents (
 ComponentInterface { name move_base
  RosParameters{
  RosParameter 'max_vel_x'{value 0.5},
  RosParameter 'max_vel_y'{value 0.5},
  RosParameter 'inflation_radius'{value 0.5},
  RosParameter 'observation_sources'{value scan}}})
 Parameters {
 Parameter{name 'qa_safety' type Double value 0.41},
 Parameter{name 'qa_energy' type Double value 0.48}
}}
```

```
RosSystem { Name 'system_b' RosComponents (
 ComponentInterface { name move_base
  RosParameters{
  RosParameter 'max_vel_x'{value 0.3},
  RosParameter 'max_vel_y'{value 0.3},
  RosParameter 'inflation_radius'{value 0.8},
  RosParameter 'observation_sources'{value point_cloud}}})
 Parameters {
 Parameter{name 'qa_safety' type Double value 0.71},
 Parameter{name 'qa_energy' type Double value 0.33}
}}
```

Figure 5.: Fragments of *RosSystem* models for two variants of a navigation system, specifying configurations of the `mode_base` node and expected values for the quality attributes. Left: base variant using a laser scanner, Right: alternative using a camera.

web interface, available at [1], for ROS packages publicly available. Once the *ROSModel* models describing single nodes are obtained, the developer can compose them, using either the graphical or the textual editor, to define the *RosSystem* model of the entire base application. Optionally, if MROS is used for an existing ROS application, the *RosModel* tools support automatic analysis of the complete existing system. This step results in a *RosSystem* model (a `.rossystem` file) which is validated by our tools and used to automatically generate executable artifacts (launch files to start the ROS system).

Step 2. Model architectural variants

Next, the developer models the adaptation possibilities—the architectural variants. Most adaptations can be sufficiently captured by parameter variations, but larger architectural variations are also possible like switching to a different hardware module). Figure 5 shows the representation of two different configurations of the same system. In this case, we have two different configurations of the node `move_base`. This node is in charge of interpreting the actions from the navigation module and translating them into commands to the base. Notice, how the maximum speed, inflation radius, and the type of sensor data used to adapt to the environment varies. The base variant `system_a` uses the output of a laser proximity scanner, while the alternative variant `system_b` uses the point cloud obtained from a 3D camera. Additionally global parameters are added to define the value of the quality attributes (abbreviated `qa`), in the case of this example for safety and energy.

Step 3. Generate meta-controller deployment configuration

We provide a *RosModel* plugin that automatically generates the application-specific configuration for the `mros1_reasoner` node to be integrated on top of the base application.

Step 4. Generate deployment configurations for variants

We automatically generate executable artifacts (KB2: ROS launch files in Figure 4), which can be called to execute runtime adaptations when a new configuration is selected by the reasoner (R4 in Figure 4).

Step 5. Generate observers of quality attributes

Another *RosModel* wizard is used to generate the scripts to observe the system. The GUI allows to select the quality attributes to be monitored. MROS generates a description of the ROS node as a *RosModel* and the skeleton of the Python code for monitoring as shown in listing 1. The class *QualityObserver* inherits from *TopicObserver*. In the initialization, we define the topic (*observed_ topic*) to which the Observer needs to subscribe and its message type. For all Observers, the method *calculate_ attribute* must be overloaded, as it is responsible for performing

[1] http://ros-model.seronet-project.de/

intelligent calculation with the data received via the topic defined in the initialization. It returns the final data as a key-value pair structured as a *diagnostic_ msgs/DiagnosticArray* message.

```
class QualityObserver(TopicObserver):
def __init__(self, name):
  #topic to observe and msg type
  topics = [(observed_topic, Float32)]
  super(QualityObserver,
      self).__init__(name, 10, topics)

def calculate_attr(self, msgs):
    status_msg = DiagnosticStatus()

    #normalized calculus for energy
    attr =normalize_value(msgs[0].data)

    status_msg = DiagnosticStatus()
    status_msg.level = DiagnosticStatus.OK
    status_msg.name = self._id
    status_msg.values.append(KeyValue("quality_attribute", str(attr)))
    status_msg.message = "QA status"
    return status_msg
```

Listing 1: Quality Observer snippet

Step 6. Generate runtime model for the meta-controller

Finally, MROS provides a script that given the *RosSystem* models of different adaptation strategies, automatically generates the TOMASys model that feeds the meta-controller at runtime.

In summary, the provided tools offer a holistic model-driven solution for the adaptation problem, exploiting models and automation, with a high degree of reuse both at design time *and* at runtime. Since many artifacts are needed , generating all of them offers not only productivity gains: the tools ensure consistency of the input artifacts and guarantee consistency of all the generated artifacts.

4. MROS Demonstrators

We use two different robotic systems to validate MROS and the developed artifacts.[1] Both are based on realistic applications for autonomous robots, concretely manipulation (MROS Demonstrator 1 (*MROS-D1*): Dual-Arm Manipulator discussed in 4.1) and navigation (MROS Demonstrator 2 (*MROS-D2*): Mobile robot navigating in a factory discussed in 4.2) [23]. Mobile manipulation and navigation robots are designed to operate in semi-structured environments, such as a warehouse, a factory floor, and a healthcare facility. These environments change dynamically (e.g. the floor plan can change from day to day, some areas may be shared with humans and with other robots, etc). This context variability demands varying configurations of the robot control architecture and appropriate methods to support successful mission completion.

4.1 *Dual-Arm Manipulator (MROS-D1)*

In *MROS-D1*, the robot is a YuMi dual-arm manipulator standing on a Clearpath Ridgback mobile platform and equipped with an Intel Realsense RGBD camera (Fig. 1a). Its task is to build a pyramid with wooden blocks at the work station. The camera and the fiducial tags are used to calibrate the robot's position in the work cell, and to locate the bricks and the target position of the pyramid. This information and prior knowledge about the blocks sizes, is used

[1]The source code of the meta-controller and the experiment scripts is released under the open source license and can be found in the following repos: https://github.com/rosin-project/metacontrol_sim (Metacontrol) and https://github.com/rosin-project/metacontrol_experiments (experimental scripts)

to compute the required manipulation motions. To improve robustness, we use a YuMi dual-arm manipulator. Its two arms provide architectural variability: the task can be performed using both or one arm. Each arm has limited range of operation, so when working with one arm, the robot uses the mobility of the Ridgeback platform to reach all the bricks.

4.2 *Mobile Robot Navigating in a Factory (MROS-D2)*

The mobile robot demonstrator (Fig. 6a) is concerned with navigation on a factory floor, where the robot moves between workstations to perform different manipulation tasks. The robot is the Ridgeback platform, equipped with two Hokuyo lasers, odometry and an inertial measurement unit, and is augmented for this demonstrator with an indoors localization system. The baseline control architecture for navigation is based on the ROS1 navigation stack [24], which uses off-the-shelf control and AI algorithms for planning and reactive obstacle avoidance. The robot navigates to a target location using odometry and sensor data and sending velocity twist commands to the Ridgeback.

The context variability includes: (i) new obstacles on the factory floor affecting the quality attribute values attained in the default configuration of the navigation stack (ii) variations in power consumption, resulting in changing battery depletion.

5. Feasibility of Runtime Adaptation

We analyze the implementation of MROS with respect to its management of component failures and runtime quality attribute levels, by using the two demonstrators and asking the following research questions:

RQ1. *How does MROS improve system robustness in the presence of component failures?*

RQ2. *How does MROS affect system and mission performance in the presence of component failures and system quality concerns?*

With **RQ1**, we demonstrate *qualitatively* how MROS addresses component failures on a manipulation mission. **RQ2** investigates *quantitatively* how the MROS reconfiguration affects the task and system performance of a robot by addressing functional and non-functional concerns in a navigation mission.

5.1 *Manipulation Mission (RQ1)*

Experiment setup

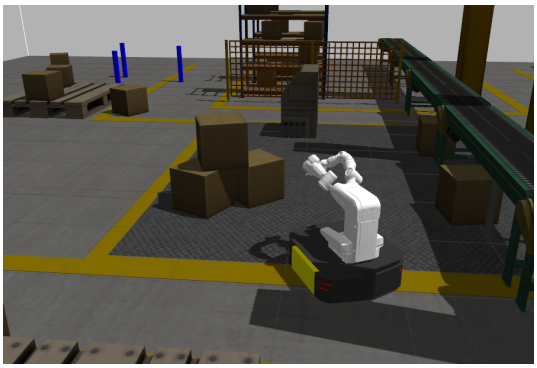
To address **RQ1**, we conducted a study on *MROS-D1* dual-arm manipulator (Section 4.1) in two scenarios:

(1) System does not detect the workstation tag.
(2) System cannot use both arms.

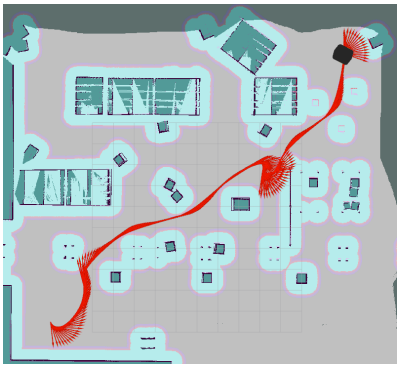
The first scenario aims to demonstrate the MROS ability to recover a function that requires multiple components (increased mission robustness). The second scenario demonstrates the MROS controller's ability to recover from an error by realising an alternative set of functions to perform the task goal with degraded performance (increased system robustness).

Results and discussion

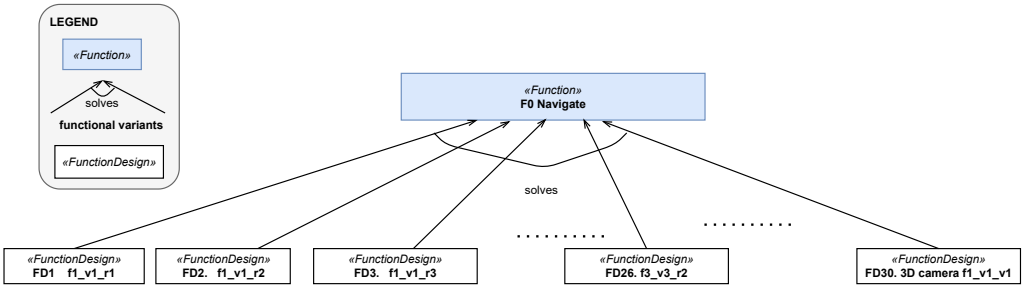
In the first scenario, we dim the lights in the room. At startup, the robot activates and grounds function design FD2.1 to use the default camera configuration. However, the fiducial tag for calibration is not detected, due to the light conditions. In this case, the function grounding that



(a) Simulation of the mobile manipulator in a factory.



(b) An example route of the mobile manipulator navigating.



(c) TOMASys model of the control architecture of the mobile manipulator navigating.

Figure 6.: Mobile Manipulator in a factory

realizes FD2.1 is unavailable (Step R2. Analyze). Hence, the meta-controller uses the TOMASys model Fig. 1a to switch to the alternative configuration of the camera parameters defined by FD2.2 that allows to detect the tag in low lighting conditions (Step R3. Plan).

In the second scenario,the robot is initially using both arms for manipulation (FD2 in Figure 1b) . We then inject a failure by blocking one of the robot's arms. MROS identifies the error in the variant currently deployed (FD2) (Step R2. Analyze), and uses the functional architecture model to find the alternative variant FD1 that only uses the available arm for manipulation (Step R3. Plan). This variant requires the robot to also navigate to different positions for the base (F1) to reach blocks that are not initially at the reach of the available arm, so the meta-controller deploys the required function groundings (FD1.1 and F2.1) (Step R3. Plan).

> We conclude that the MROS meta-control augments the reliability of the overall system, delivering the expected behavior in real-time in the context of component failures.

5.2 *Navigation Mission (RQ2)*

Experiment setup

To address **RQ2**, we systematically compared a system with the *MROS* meta-controller to a benchmark system (*Baseline*) for mission success, component failure, and quality concerns in the navigation scenario of Section 4.2.[1] The robot is placed at a starting location and it navigates to a goal location. An example of the route during one run is shown in Fig. 6b; the starting position is at the bottom-left of the map and the goal is at the top-right. We considered six pairs of initial and goal positions.

[1]The data from this experiment can be found in https://doi.org/10.5281/zenodo.5340886 released under the Creative Commons Attribution 4.0 International license.

Baseline

We developed 30 variants of a control architecture, with different quality expected performing the navigation mission. The TOMASys model of the control arhchitecture for the navigation mission is presented in Figure 6c where each of variant is represented as a function design. Each of these 30 function designs realize one function that is F0 Navigate and constitutes the *Baseline* system for the corresponding experiment run. We obtained different function designs by selecting between two different components: a laser or a 3D camera as sensor input for obstacle avoidance, and different values of three parameters that govern the behavior of the `move_base` node that generates the motion commands: `inflation_radius` (minimal distance allowed to the closest obstacle), `max_vel` (maximum velocity allowed for the robot), `controller_frequency` (the frequency of the control loop).

MROS-D2

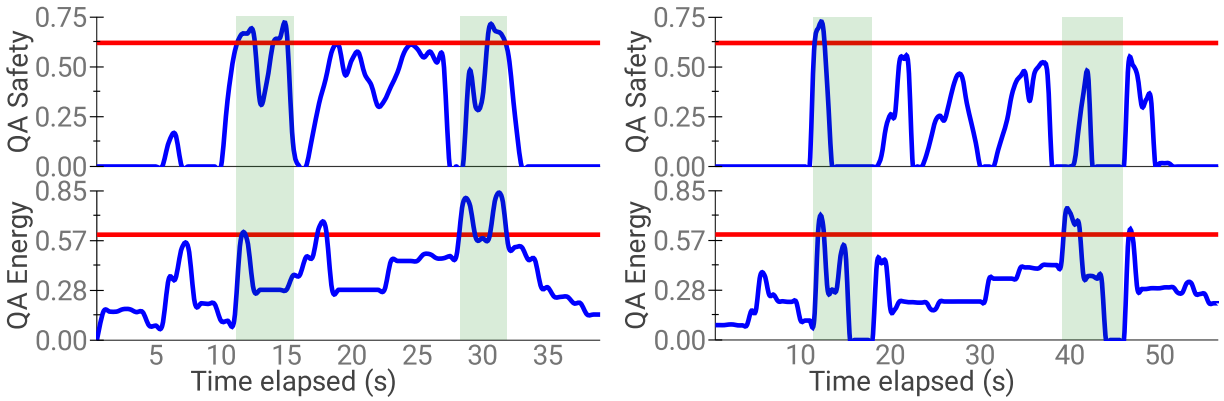
In *MROS-D2*, the meta-controller runs on top of one of the architectural variants, performing run-time analysis on component failures and quality violations, and making decisions on when and how to switch to a different navigation configuration. *MROS-D2* calculates safety at run-time by measuring the braking distance of the robot before hitting an obstacle `d_break` [25] and the distance from the current location of the robot to the closest obstacle in its direction P. We define runtime safety to be 1 if the robot is at least `d_break` distance from the closest obstacle. As the robot approaches the closest obstacle, safety at run-time is computed as $^{P}/_{\texttt{d\_break}}$. Furthermore, *MROS-D2* computes the energy at run-time based on an energy model that simulates the power consumption of a robot. The energy model computes the instantaneous `power_load` of the robot using current controller frequency, velocity, acceleration, etc. We specify 0.6 as a required threshold for safety and 0.62 for energy. If the robot breaches these thresholds, *MROS-D2* initiates self-adaptation.

The independent variables in our experiment are: (i) the architectural variant initially deployed for the control architecture, and (ii) the contingencies during the mission:
For each run, we compare the MROS and *Baseline* systems in terms of violation of system qualities thresholds, component failures handling, and mission performance. A run consists of an initial *system configuration* C and a *perturbation* P. We selected nine initial configurations (out of 30) that show the best-expected performance in a set of preliminary tests. A perturbation models a sequence of contingencies within a mission. Each perturbation P is represented by a tuple $\langle iv_1, iv_2, iv_3 \rangle$, where each element represents a value for each contingency type:

(1) *Cluttered environment:* the environment contains a set of new obstacles unknown to the robot (not in the map used by the navigation) in randomized positions. We quantify the level of clutter depending on the number of new obstacles as *low* (7), *medium* (14) and *high* (20).
(2) *Unexpected increase in power consumption:* This could be the result of a sticky surface, a slope, some problem with the wheels, or activation of a new sensor that consumes more power. We quantify this contingency in three levels: 20%, 40%, and 60% increase in power consumption.
(3) *Component failure:* We inject a failure in the laser sensor by killing its driver ROS node at run-time.

To estimate the effects of MROS on adaptation and mission performance we measure the following dependent variables on both systems:

- The *percentage of time the mission is above the threshold for safety risk and energy consumption.*
- *Mission success.* We defined a demanding metric for mission success that combines reaching the goal with performance and safety requirements. A mission is considered successful if (i) the robot reaches its goal, (ii) the average safety risk level for a run is < 0.4, (iii) the safety risk is below its threshold for more than 95% of the mission time; (iv) the



(a) Baseline without MROS meta-control.

(b) The system with MROS Meta-control.

Figure 7.: Safety (top) and Energy (bottom) quality attribute values for a single run. The red line indicates the maximum allowed value in each case. The highlighted areas mark the contingencies where a reconfiguration occurs.

Table 2.: Average time a mission is above the safety threshold.

How cluttered the environment is	*MROS*	*Baseline*
no clutterness contingency	0.99%	2.60%
low clutterness	0.98%	2.84%
medium clutterness	0.88%	1.73%
high clutterness	1.00%	2.70%
Total	**0.96%**	**2.50%**

energy-consumption is below its threshold level for more than 90% of the time.

- *Time to complete* the mission. We measure the time the robot needs to complete a single run.

Results and Discussion

We collected data points for 6727 runs of both systems. Figures 7a and 7b illustrate the management of safety and energy at run-time during a single run for the Baseline system and for MROS-D2. The red line represents the threshold for each quality, while the highlighted area is a time period when contingencies are introduced. For the *Baseline* system (Fig. 7a), the total run-time is 39 seconds, the safety quality attribute value has an average of 0.47 and is above the defined limit for 4 seconds (10% of the run-time), and the QA Energy has an average of 0.38 and is above the required limit for 3 seconds (8% of the run-time). For the *MROS* system (Fig. 7b), the total run-time is 56 seconds. In this case, the average of the safety quality attribute value is 0.37 and it is above the defined limit for only 1 second (2% of the run-time). The average for QA Energy is 0.325 and it is above the required limit for 3 seconds (5% of the run-time). Moreover, when the MROS system is used, two reconfigurations occur, which can be clearly seen in Figure 7b where, in the highlighted areas, both QA Energy and Safety values drop down immediately after the limit is crossed and then they restart with a different trend. Each reconfiguration takes about 4.3 seconds, contributing to the MROS run being longer than the *Baseline*.

Table 2 shows the average percentage the *Baseline* and *MROS-D2* are above the safety threshold during the mission. On average *MROS-D2* is 2.6 times more effective in reducing the safety violation from 2.5% to 0.96% on average in a single mission. *MROS-D2* always outperforms the *Baseline*, independently of the level of clutter (number of obstacles). Furthermore, *MROS-D2* manages to keep the safety violation below 1% for all contingency levels related to a cluttered environment. Considering these contingency levels, the *MROS-D2* meta-controller re-configures

Table 3.: Average time a mission is above the energy threshold.

Power consumption	*MROS*	*Baseline*
no power contingency	1.26%	2.63%
power increase 20%	1.16%	2.22%
power increase 40%	2.47%	2.63%
power increase 60%	2.63%	4.57%
Total	**1.86%**	**2.98%**

Table 4.: Mission success rates under various contingencies.

Contingency Type	*MROS*	*Baseline*
no clutter contingency	78.16%	67.03%
low clutterness	77.92%	62.03%
medium clutterness	79.13%	70.16%
high clutterness	77.77%	62.06%
no power contingency	81.63%	59.51%
power increase 20%	79.38%	68.48%
power increase 40%	76.47%	71.01%
power increase 60%	76.35%	60.18%
Total	**78.50%**	**65.20%**

the system once every 2-3 runs.

Table 3 shows the average percentage the *Baseline* and *MROS-D2* are above the energy threshold during the mission. Again, *MROS-D2* outperforms the *Baseline* for all contingency cases related to the increase in power consumption. The biggest difference is the 60% energy increase contingency, where the result shows that MROS reduces the average violation of the energy constraint from 4.57% to 2.63% during the mission.

Table 4 shows the mission success for each of the *MROS-D2* and *Basline* systems under various contingencies. For example, in all missions where high clutterness was introduced the *MROS-D2* had 77.77% success rate, while the *Baseline* had only 62.06% success rate. Table 4 shows that *MROS-D2* outperforms the *Baseline* for all contingency types. *MROS-D2* improves the mission success by 13.3% on average. In this calculation, we decided not to include the data for component failure, because the *Baseline* will always fail the mission when a laser failure is introduced (so the 13.3% reported is biased against *MROS-D2*). In contrast, *MROS-D2* effectively adapts when laser failure is introduced, completing the mission in 73% of the mission runs. Beyond the component failure contingency, the biggest difference in mission success can be seen in the case when there is not any increase in power consumption. In this case, the *Baseline* has 59.51%, while *MROS-D2* has 81.63% mission success.

Finally, MROS increases the mission completion time by 7.5, as a result of the Meta-controller choosing a slower configuration to satisfy the energy and safety quality concerns. As expected in MROS-D2, adaptations were triggered more often for higher contingency levels in terms of clutter and energy peaks.

It is important to note that one can engineer a more resilient robot, with more redundant hardware and control strategies, and then the reported numbers for *MROS* will improve further, as they are indeed a function of a more adaptive strategy, which the baseline system does not have. The experiment demonstrates however that the model-based MROS meta-controller is capable to exploit the hardware and software redundancy to increase robustness. Thus we conclude:

> Model-based intelligent reconfiguration does improve mission and system performance for robot systems, effectively managing component failures and quality concerns.

6. Qualitative Analysis of MROS Design

MROS contributes two tools: the *RosModel* MROS plugins for ROS developers at design time, and the Meta-controller ROS packages to deploy it in a ROS system. In order to understand the generality of our contribution, we discuss the re-usability and extensibility of these artifacts.

6.1 *Reusability*

We prototyped the MAPE-K loop of the meta-controller for the *MROS-D1* dual-arm manipulator demo without generative programming. Following a common practice in the field, we then generalized it for *MROS-D2* using model-driven development.

For *MROS-D1*, we designed the OWL implementation of the TOMASys meta-model (`tomasys.owl`) to support the specification of MROS models, and the ontological reasoning for Steps R2 and R3 (`mros1_reasoner.py`). The TOMASys model was manually developed. We first analyzed the existing software for the system following the *ISE&PPOOA* MBSE methodology [17], obtaining the conceptual representation of its functional architecture and possible variants (Fig. 1b). Then we modeled it in OWL, using `tomasys.owl` and the Protege editor. For *MROS-D1*, we used application specific solutions for steps R1 and R4 at runtime.

To exploit the commonality in the domain, we consequently settled to use the model-driven methods and the *RosModel* bridge tools. The resulting MROS tools offer:

- *Extracting models from code*, allowing to add meta-control to existing projects and using new nodes with MROS. (corresponds to Design Step 1. in the application design process in Table 1)
- *Modeling languages and tools for ROS architectures:* (a) linking ROS and TOMASys via *RosSystem* (b) with consistency checking and syntax feedback (corresponds to Design Step 2. in the application design process in Table 1)
- *Automatic generation of the ontological knowledge base* (otherwise a tedious and low-level task) - corresponds to Design Step 6. in the application design process in Table 1.
- *A reusable monitor of node status* `rosgraph_monitor` (developed as part of the MROS generic infrastructure and is used in Step R1 of the adaptation process)
- *A generator of boilerplate code for observers* (step R1). (supports Step 5 in the application design process in Table 1). Observers boilerplate code is shown in Listing 1.
- *A reusable meta-controller*, including tactics to detect and address violations of quality attributes and contingencies (developed as part of the MROS generic instracture and is used in steps R2 and R3 of the adaptation process).
- *A reusable navigation ontology* based on `tomasys.owl` including minimal domain-specific model of safety, energy and performance quality attributes in navigation, along with the associated SWRL adaptation rules (This is developed as part of the MROS generic infrastructure with an update on domain-specific details that are developed as part Design Step 6. in the application design process in Table 1). It is used in steps R2, R3 of the adaptation process).
- *Automatic generation of executable reconfiguration actions for the system architecture* (launch files) used by the `rosgraph_manipulator` to reconfigure the system (corresponds to Design Step 3. in the application design process in Table 1 and it is used in Step R4 in the adaptation process).

For demonstrator *MROS-D2*, we used this infrastructure to model the architectural variants with 30 different *RosSystem* models of the possible configurations of the navigation stack.

It is a well known fact that effective deployments of model-driven engineering require a very good domain implementation. It is somewhat surprising thus that often model-driven technology is demonstrated in isolation, or on small examples and use cases. The Robot Operating System provides an excellent platform, a rich and highly customizable implementation of the robotics domain. ROS with model-driven engineering is a "match made in heaven." Using ROS was an enabling factor for this research: it opened access to standard formats, architectures, and interfaces. We benefited from the rich tooling and experience ecosystem, which allowed us to efficiently execute rich and realistic experiments, rarely seen in model-based self-adaptation research. Furthermore, using generative model-driven solutions with ROS meant that the meta-controller technology is readily accessible to unprecedented number engineers and researchers working on self-adaptation in robotics and software engineering using ROS.

6.2 *System Extensibility*

We analyze how the MROS framework supports system extensibility in the context of a mobile robot navigating in a factory scenario with respect to which aspects should one extend in the application: (i) to capture new components added to the system; (ii) to capture new quality attributes to be monitored.

Adding new components

A 3D camera component was added so the mobile robot would be able to continue the task even if the laser fails. To add a new component we only needed to add the new configurations in the *RosSystem* models. We extended the number of system variants by adding new configurations that contain the camera component as shown in Figure 5. We used the *RosModel* tools to model 3 new configurations (Step 2. Model architectural variants in Table 1 and generate ROS Launch files for each variant (Step 3. in Table 1) which corresponds to 30 different variants in total. For monitoring, we did not need to add new observer models, because the current tooling allows component failures to be captured. For analysis and planning, we used the model-to-model transformation to automatically generate the updated knowledge base. Finally, the *RosModel* tools automatically generate the new reconfiguration actions (Step 6. Generate runtime model for the meta-controller in Table 1).

Adding new quality attributes

We extended the application scenario *MROS-D2* with energy-efficiency. For monitoring, we developed an observer for energy-efficiency that calculates energy at run-time as shown in Listing 1. The energy level (function calculate_attr) was computed using an energy model that simulates the power consumption of a robot using controller frequency, velocity, acceleration, etc. For analysis (R2) and planning (R3), we extended the knowledge base with new information about the energy-efficiency of each system configuration (Step 2. in Table 1). Manual work is unavoidable, but the artifacts are modular and reusable across robotic systems with the same quality attributes. Again as a final step, the *RosModel* tools automatically generate the reconfiguration actions (Step 6. in Table 1).

In conclusion, in both cases the cost of development of the new features of the robot, greatly outweighs the cost if incorporating it into the meta-control system, which is mostly automatic, barring modeling the new configurations (easy) and new quality attributes (reusable).

7. Related Work

Dynamic architectures and self-adaptation have been an active topic of research in model-driven software engineering for a decade. The so called *models-at-runtime* paradigm [9, 26] has been exploring the use of abstractions to reduce complexity and cope with increasingly complex runtime adaptation problems. One group of prior works related to the use of models in robot software

is based on variability management and feature models that can represent the functionalities provided by a software system symbolically [27, 28]. The previous efforts have focused on the solution space of architecture design, that is the possible configurations of components. The discrete variants in the architecture are modeled separately from properties and quality attributes (the problem space). However, formal semantics, a common meta-model, for automated reasoning about the relation between both the solution and the problem space is missing. We address this shortcoming by extending the TOMASys meta-model to represent quality attributes and implement it as an ontology for automated reasoning.

Non-functional requirements, such as performance, reliability or safety, have been considered in the dynamic reconfiguration of robot software architecture [29]. In a newer work, Brugali et al. have integrated feature models, standard modelling languages (UML-MARTE) with queuing network models to support reconfiguration in order to maintain an adequate level of performance [10]. These solutions require intensive modelling efforts, MDE skills and completely new toolsets. In contrast, we chose a less invasive path, in that we exploit existing codebase and practices in the ROS ecosystem.

Iftikhar and Weyns [30] propose Active FORmal Models for Self-adaptation (ActivFORMS). The adaptation goals that are verified offline are guaranteed to be observed at runtime. The method uses timed automata for modeling the behavior of the multi-robot system, a method unlikely to be accepted by a broader robotics community. Its expressiveness is lower than of linear hybrid control systems, which means they require careful modeling and abstraction for any typical robotics problem. Our ontology-based approach presented offers more natural models of trade-offs and requirements, including non-linearities, thanks to the use of description logics.

Aldrich et al. [31] leverage predictive data models to enable automated robot adaptation to changes in the environment at run-time. While the approach clearly depicts the benefits of using models by capturing high-level artefacts (combining application and system logic), it makes it extremely challenging for a developer to make use of them in robotic scenarios because: (i) it does not introduce models that can be reused for a different application; (ii) it does not give insights how someone can build similar models; (iii) it does not provide automated infrastructure to leverage those models.

The BRICS[32] and RobMoSys projects [33] have proposed model-based approaches to develop robotic systems. Our work is inspired by the meta-models resulting from those efforts, but goes beyond realizing their proposed future roadmap to use the models at run time for adaptation and autonomy.

Cheng et al. propose [34] a framework that uses GSN assurance case models to manage run-time adaptations for ROS-based systems. While the framework clearly shows the value of systematically integrating assurance information from GSN models to ROS specific information to guide runtime monitoring and adaptation, it has couple of drawbacks: (i) it uses custom developed libraries specific to the approach, rather than standard libraries in ROS (such as ROS Diagnostics); (ii) it does not reuse preexisting practises developed in the ROS ecosystem raising the entry barrier for ROS developers to effectively use it. Also, thanks to the tight integration of the ROS framework and model-driven automation, we are able to execute extensive experiments, beyond simple demos. We quantify the improvement of the self-adaptation, and its computational cost.

As discussed in section 2, our work exploits of two existing streams of contributions. Hernández et al. [11, 20] proposed principles for knowledge-based self-adaptation and the concept meta-control based on off-the-shelf reasoners, but did not provide a solution to obtain the architectural models or to realize their approach on robotic systems. Hammoudeh et al. [12] created the model-based RosModel tooling to support development ROS systems, but it could not model architectural variants or quality concerns. Our work leverages both works to create a complete solution to develop self-adaptation in ROS systems.

8. Conclusions and Future Work

This paper introduces the MROS framework to support the integration of self-adaptation in robot control architectures based on ROS. MROS promotes separation of system management decision making from the mission logic, using ontological reasoning and design models.

MROS uses platform specific DSLs to represent the architectural models of variants of the robotic system. We have developed extensions of the *RosModel* tools that lower the modelling effort for developers and allow to automate the implementation and deployment of the self-adaptation mechanisms. The use of an ontology-based knowledge based and reasoning for the MROS MAPE-K loop facilitates reusing MROS adaptation rules across robotic applications, and extend them to address new quality concerns beyond safety and energy, which have been addressed in this work.

We developed two MROS tools: *RosModel* tooling extensions to support the development phase, and a ROS package to deploy the MROS meta-controller for run time adaptation, and applied in two different complex robotic systems for validation. Our results from systematic experiments in the mobile robot demonstrator show how MROS can be used to increase mission reliability and safety and energy concerns. The implementation also illustrates how the MROS DSL approach at design time and ontological approach at run time allowed to easily reuse the adaptation rules from component failure, and extend them to safety concerns and then to energy.

The architecture of MROS provides a framework to explore the integration of methods from robotics, software engineering and artificial intelligence. For example, currently we are investigating deep learning methods for quality prediction, and the integration of MROS meta-controller with different solutions to manage the mission logic, such as behaviour trees or planning methods. Moreover, we are working on a version of MROS for ROS2, which offers better support for component and system management than ROS1, and more industrial traction. This will open new opportunities and challenges for self adaptive systems and modelling in robotics.

Acknowledgement

This work is supported by the RobMoSys-ITP-MROS (Grant Agreement No. 732410) and ROSIN (Grant Agreement No. 732287) projects with funding from the European Union's Horizon 2020 research and innovation programme. Darko Bozhinoski acknowledges support from the Belgian Fonds de la Recherche Scientifique–FNRS [No:40001145].

References

[1] Quigley M, Conley K, Gerkey BP, Faust J, Foote T, Leibs J, Wheeler R, Ng AY. ROS: an open-source Robot Operating System. In: Icra workshop on open source software. Vol. 3. 2009. p. 1–6.

[2] Weyns D, Bencomo N, Calinescu R, Camara J, Ghezzi C, Grassi V, Grunske L, Inverardi P, Jezequel JM, Malek S, Mirandola R, Mori M, Tamburrelli G. Perpetual assurances for self-adaptive systems. In: de Lemos R, Garlan D, Ghezzi C, Giese H, editors. Software engineering for self-adaptive systems iii. assurances. Cham: Springer International Publishing. 2017. p. 31–63.

[3] Thrun S, Montemerlo M, Dahlkamp H, Stavens D, Aron A, Diebel J, Fong P, Gale J, Halpenny M, Hoffmann G, Lau K, Oakley C, Palatucci M, Pratt V, Stang P, Strohband S, Dupont C, Jendrossek LE, Koelen C, Markey C, Rummel C, van Niekerk J, Jensen E, Alessandrini P, Bradski G, Davies B, Ettinger S, Kaehler A, Nefian A, Mahoney P. Winning the darpa grand challenge. Journal of Field Robotics. 2006;Accepted for publication.

[4] Correll N, Bekris KE, Berenson D, Brock O, Causo A, Hauser K, Okada K, Rodriguez A, Romano JM, Wurman PR. Lessons from the amazon picking challenge. CoRR. 2016;abs/1601.05484. Available from: http://arxiv.org/abs/1601.05484.

[5] Hernandez Corbato C, Bharatheesha M, van Egmond J, Ju J, Wisse M. Integrating different levels

of automation: Lessons from winning the amazon robotics challenge 2016. IEEE Transactions on Industrial Informatics. 2018;PP(99):1–1.
[6] Pohl K, Böckle G, van der Linden F. Software product line engineering. Springer Verlag. 2005.
[7] Capilla R, Bosch J, Trinidad P, Cortés AR, Hinchey M. An overview of dynamic software product line architectures and techniques: Observations from research and industry. J Syst Softw. 2014;91:3–23. Available from: https://doi.org/10.1016/j.jss.2013.12.038.
[8] Bencomo N, France RB, Cheng BHC, Aßmann U, editors. Models@run.time — foundations, applications, and roadmaps. Vol. 8378 of Lecture Notes in Computer Science. Springer. 2014.
[9] Blair G, Bencomo N, France RB. Models@ run.time. Computer. 2009 Oct;42(10):22–27.
[10] Brugali D, Capilla R, Mirandola R, Trubiani C. Model-based development of qos-aware reconfigurable autonomous robotic systems. In: 2018 second ieee international conference on robotic computing (irc). 2018 Jan. p. 129–136.
[11] Hernández C, Bermejo-Alonso J, Sanz R. A self-adaptation framework based on functional knowledge for augmented autonomy in robots. Integrated Computer-Aided Engineering. 2018;25(2):157–172.
[12] Hammoudeh Garcia N, Lüdtke M, Kortik S, Kahl B, Bordignon M. Bootstrapping mde development from ros manual code - part 1: Metamodeling. In: 2019 third ieee international conference on robotic computing (irc). 2019 Feb. p. 329–336.
[13] Aguado E, Milosevic Z, Hernández C, Sanz R, Garzon M, Bozhinoski D, Rossi C. Functional self-awareness and metacontrol for underwater robot autonomy. Sensors. 2021;21(4). Available from: https://www.mdpi.com/1424-8220/21/4/1210.
[14] Kephart J, Chess D. The vision of autonomic computing. Computer. 2003 January;36(1):41 – 50.
[15] Blanke M, Kinnaert M, Lunze J, Staroswiecki M. Diagnosis and fault-tolerant control. Springer-Verlag Berlin. 2006.
[16] Garlan D. Software architecture: A travelogue. In: Proceedings of the on future of software engineering. FOSE 2014. Hyderabad, India. New York, NY, USA: ACM. 2014. p. 29–39. Available from: http://doi.acm.org/10.1145/2593882.2593886.
[17] Fernandez-Sánchez JL, Hernández C. Practical model based systems engineering. Artech House. in print 2019.
[18] Hernandez C, Fernandez-Sanchez JL. Model-based systems engineering to design collaborative robotics applications. In: 2017 ieee international systems engineering symposium (isse). 2017 Oct. p. 1–6.
[19] ROS Diagnostics Package. ????. Available from: http://wiki.ros.org/diagnostics.
[20] Hernandez Corbato C, Milosevic Z, Olivares C, Rodriguez G, Rossi C. Meta-control and self awareness for the ux-1 autonomous underwater robot. In: Silva MF, Luís Lima J, Reis LP, Sanfeliu A, Tardioli D, editors. Robot 2019: Fourth iberian robotics conference. Springer International Publishing. 2020. p. 404–415.
[21] Lamy JB. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. Artificial Intelligence in Medicine. 2017;80:11 – 28. Available from: http://www.sciencedirect.com/science/article/pii/S0933365717300271.
[22] Hammoudeh García N, Deshpande H, Santos A, Kahl B, Bordignon M. Bootstrapping mde development from ros manual code: Part 2—model generation and leveraging models at runtime. Software and Systems Modeling. 2021 Apr;Available from: https://doi.org/10.1007/s10270-021-00873-2.
[23] ABB. Abb demonstrates concept of mobile laboratory robot for hospital of the future. https://new.abb.com/news/detail/37279/hospital-of-the-future. 2019 Oct.
[24] Marder-Eppstein E, Berger E, Foote T, Gerkey B, Konolige K. The office marathon: Robust navigation in an indoor office environment. In: International conference on robotics and automation. 2010.
[25] Chung W, Kim S, Choi M, Choi J, Kim H, Moon Cb, Song JB. Safe navigation of a mobile robot considering visibility of environment. IEEE Transactions on Industrial Electronics. 2009;56(10):3941–3950.
[26] Bencomo N, Götz S, Song H. Models@run.time: a guided tour of the state of the art and research challenges. Software & Systems Modeling. 2019 Jan;Available from: https://doi.org/10.1007/s10270-018-00712-x.
[27] Kang et al. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-21. 1990. Tech Rep.
[28] Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wasowski A. Cool features and tough decisions: a comparison of variability modeling approaches. In: Eisenecker UW, Apel S, Gnesi S, editors. Sixth international workshop on variability modelling of software-intensive systems, leipzig, germany, january 25-27, 2012. proceedings. ACM. 2012. p. 173–182. Available from: https://doi.org/10.1145/2110147.

2110167.

[29] Lotz A, Inglés-Romero JF, Vicente-Chicote C, Schlegel C. Managing run-time variability in robotics software by modeling functional and non-functional behavior. In: Nurcan S, Proper HA, Soffer P, Krogstie J, Schmidt R, Halpin T, Bider I, editors. Enterprise, business-process and information systems modeling. Berlin, Heidelberg: Springer Berlin Heidelberg. 2013. p. 441–455.

[30] Iftikhar MU, Weyns D. Activforms: Active formal models for self-adaptation. In: Proceedings of the 9th international symposium on software engineering for adaptive and self-managing systems. 2014. p. 125–134.

[31] Aldrich J, Garlan D, Kästner C, Le Goues C, Mohseni-Kabir A, Ruchkin I, Samuel S, Schmerl B, Timperley CS, Veloso M, et al.. Model-based adaptation for robotics software. IEEE Software. 2019; 36(2):83–90.

[32] Bruyninckx H, Klotzbücher M, Hochgeschwender N, Kraetzschmar G, Gherardi L, Brugali D. The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC '13. Coimbra, Portugal. New York, NY, USA: ACM. 2013. p. 1758–1764. Available from: http://doi.acm.org/10.1145/2480362.2480693.

[33] RobMoSys Project. RobMoSys - Composable Models and Software. ????. Available from: https://robmosys.eu/.

[34] Cheng BH, Clark RJ, Fleck JE, Langford MA, McKinley PK. AC-ROS: Assurance Case driven Adaptation for the Robot Operating System. In: Proceedings of the 23rd acm/ieee international conference on model driven engineering languages and systems. 2020. p. 102–113.