MROS

A framework for robot self-adaptation

Silva, Gustavo Rezende; Garcia, Nadia Hammoudeh; Bozhinoski, Darko; Deshpande, Harshavardhan; Oviedo, Mario Garzon; Wasowski, Andrzej; Montero, Mariano Ramirez; Corbato, Carlos Hernandez

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

MROS: A framework for robot self-adaptation

Gustavo Rezende Silva*
Darko Bozhinoski†
Mario Garzon Oviedo*
Mariano Ramírez Montero*

Nadia Hammoudeh Garcia‡
Harshavardhan Deshpande‡
Andrzej Wasowski§
Carlos Hernandez Corbato*

ABSTRACT

Self-adaptation can be used in robotics to increase system robustness and reliability. This work describes the Metacontrol method for self-adaptation in robotics. Particularly, it details how the MROS (Metacontrol for ROS Systems) framework implements and packages Metacontrol, and it demonstrate how MROS can be applied in a navigation scenario where a mobile robot navigates in a factory floor. Video: https://www.youtube.com/watch?v=ISe9aMskJuE

KEYWORDS

Self-adaptive systems, Self-adaptation, Metacontrol, MROS, Robotics

1 INTRODUCTION

Autonomous robots are usually programmed by integrating and configuring individual robot capabilities in a governing static control architecture. 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. The demand for higher autonomy in robotic systems requires that robots can decide and switch between available alternative configurations, acting as a self-adaptive system. The addition of self-adaptation capabilities to robotic systems could be performed by including the adaptation logic into the robot's application logic. However, this leads to the problems: (1) the adaptation logic being tangled up with application logic, making it difficult to individually change any of the separate logic; (2) the lack of reuse of the adaptation reasoning, requiring it to be reprogrammed for each application. Thus, it is advantageous to use methods and tools that enable the separation of concerns and promote reuse. This paper demonstrate how Metacontrol [6] can be applied with the MROS [3] tooling to enable self-adaptation in robotic systems.

Metacontrol [6] is a framework that incorporates systems with the capability to self-adapt to maintain their functionalities at an expected performance. Metacontrol has the design goals of being reusable and extensible. This is achieved with the design principles: (1) separating the adaptation and application reasoning; (2) exploiting at runtime the engineering knowledge of how the system is

designed to reason how and when the system needs to adapt, i.e., by being model-based.

MROS [3] is a tooling that implements the Metacontrol framework for ROS-based [12] systems. This paper demonstrates how MROS[1] lowers the barrier for robot developers to add self-adaptation to a robot control architecture by leveraging Metacontrol's design principles to promote reuse and extensibility. In this paper, the novel contributions are:

- A methodology for robotic developers to effectively use the MROS tooling to add self-adaptation capabilities in a robotic system.
- A case study that demonstrates the value of the MROS tooling and the proposed methodology.

2 METACONTROL

Metacontrol [6] is a reference architecture that provides systems with the capability to self-adapt to maintain their functionalities at an expected performance, despite external disturbances, faults, and unexpected behavior. For this, it uses engineering knowledge of how the system is designed, in the form of a runtime model, to reason when and how the system needs to adapt. Metacontrol is further explored in the works [1, 3, 7–9, 11].

Metacontrol follows the standard model in self-adaptive systems (Figure 1) that separates the application into a Managed system, which consists of the robotic domain specific components, and a Managing subsystem, which is a Metacontroller component that closes a feedback loop with the Robot subsystem to monitor and adapt it when necessary.

The Metacontroller operates following the MAPE-K loop [10], which consists of a Monitor, Analyze, Plan and Execute steps, that are based on a Knowledge Base (KB). The Monitor step is responsible for measuring relevant quality attributes for the functionalities of the system. The Analyze step is responsible for deciding whether the Managed system needs to reconfigure. When it is necessary to perform reconfiguration, the Plan step is responsible for selecting a new configuration for the Managed system that satisfies its requirements. The Execute step is responsible for reconfiguring the Managed system.

The KB is the main difference of Metacontrol with other self-adaptation frameworks. It consists of a runtime model based on the TOMASys metamodel. TOMASys contains concepts to describe the functional and physical architecture of a system, and its variants, both at design time and at runtime. For the robotic developer using MROS, only TOMASys design time concepts are needed to create the Metacontrol KB, while the runtime elements are used

---

*Authors are with the Department of Cognitive Robotics, Delft University of Technology, Delft, Netherlands; Emails: g.rezendesilva@tudelft.nl, m.a.garzonOviedo@tudelft.nl, m.ramirezmontero-1@student.tudelft.nl, c.h.corbato@tudelft.nl

†Authors is with IRIDIA, University Libre du Bruxelles, Bruxelles, Belgium; Email: darko.bozhinoski@ulb.be

‡Authors are with Fraunhofer Institute for Manufacturing Engineering and Automation IPA, Stuttgart - Germany; Emails: nadia.hammoudeh.garcia@ipa.fraunhofer.de, harshavardhan.deshpande@ipa.fraunhofer.de

§Author is with Computer Science Department IT University Conpenhagen, Denmark; Email: wasowski@itu.dk

---

[1]This paper address the ROS1 version of MROS, which can be found at: https://github.com/meta-control/mc_mros_reasoner/tree/mros1-master

by an automatic reasoner in the Metacontroller. A more detailed description of TOMASys can be found in [1, 6].

The *Function* element represents an abstract functionality of the system, such as navigating from point A to B. A *Function Design* is an engineering design solution that solves a specific *Function* with an expected performance. *Quality Attribute Type* and *Quality Attribute Value* are used to capture the systems engineering meaning of QAs. A *Quality Attribute Type* represents a characteristic of the system that shall be observed, such as energy. And a *Quality Attribute Value* represents an amount of a *Quality Attribute Type*, e.g., 1 Joule.
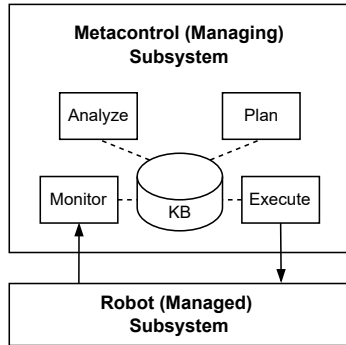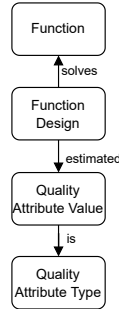

Figure 1: Metacontrol   **Figure 2: TOMASys**

3 MROS FRAMEWORK

MROS is a ROS-based implementation of Metacontrol that enables architectural self-adaptation in ROS-based robotic systems. MROS monitors the state of the ROS system and updates its KB according to it, then it uses ontological reasoning to Analyze and Plan the required adaptations, which are then executed by reconfiguring the ROS node graph. This section first describes how the MROS library implements the MAPE-K loop and the Metacontrol KB, and then it presents the methodology that robot developers can follow to use it to implement self-adaptation in ROS 1 applications[2].

3.1 MROS library

MROS follows the MAPE-K model for runtime adaptation, as shown in in Figure 3.

Monitor: This component is realized with the ROS node *rosgraph_monitor*, it provides system observers templates. The observer nodes operate at a fixed frequency and publish the monitored values to the Metacontrol reasoner using the standard ROS diagnostic mechanism (i.e., a specific channel and data structure in ROS systems). This means they easily integrate into an existing ROS system and might use or provide services already needed in the system, regardless of the presence of a Metacontroller.

Knowledge Base: The Knowledge Base consists of **(1)** the TOMASys metamodel that allows modeling the functional architecture of autonomous systems; **(2)** A TOMASys model of the Managed subsystem that defines its unique *Functions*, *Function Designs* and

---

[2]There is already a beta version of MROS supporting ROS 2, but it requires a different method

its expected *Quality Attributes*, in other words, all possible configurations for the Managed subsystem. The KB is implemented as an ontology using the Ontology Web Language (OWL) in combination with the Semantic Web Rule Language (SWRL).

Analyze and Plan: These components are realized with the ROS node *mros1_reasoner*, it integrates the OWL ontology, i.e. the KB, with ROS and it reasons over it to decide when and how to adapt the Managed subsystem. It consists of a ROS node implemented with Python that makes use of the library OwlReady2 to bridge the ontology with Python and ROS. And the reasoning is performed with the off-the-shelf ontological reasoner Pellet.

Execute: This step is realized with the ROS node *mc_rosgraph_manipulator*. It provides a ROS node that is responsible for killing and starting new ROS nodes and changing the necessary ROS parameters to fulfill the desired configuration.

3.2 MROS methodology

To use MROS to add self-adaptation to a ROS system, the activities in Figure 4 may be followed.

Step 1: Define the possible adaptations, i.e. set of architectural variants relevant for the robot to be able to perform a mission. For each architectural variant, the developer creates a ROS launch file for the automatic deployment of the corresponding configuration of ROS nodes.

Step 2: Create the KB by modeling the ROS system architectural variants with TOMASys. MROS provides an implementation of TOMASys in OWL, the developer needs to create the individuals for the application's *Functions*, *Function Designs* and its expected *Quality Attribute Values* and *Quality Attribute Types*.

Step 3: Create observers using the templates provided in the ROS package *rosgraph_monitor* to monitor the status of the active ROS nodes and to measure the relevant *Quality Attributes*.

Step 4: Configure the *mros1_reasoner* and *mc_rosgraph_manipulator* nodes by linking the system architectural variants (step 1) with TOMASys *Function Designs* (step 2), and connecting them to application-specific reconfiguration actions (e.g. to store required node states). This is done simply editing a template Metacontroller configuration file in YAML.

4 CASE STUDY

This section describes how MROS can be set up for a robot navigating on a factory floor following the MROS methodology described in Section 3.2. This case study comes for the previous work of Bozhinoski et al. [3]. The experiment code can be found at https://github.com/rosin-project/metacontrol_sim.

Case study description. The case study consists of a Clearpath Ridgeback mobile robot that navigates in a factory floor. The robot is equipped with two laser sensors, one IMU, and an odometry system. The navigation system is realized with the ROS1 navigation stack. As the robot navigates, unexpected obstacles may appear in it's path and it may get closer or further to objects, which causes its safety quality attribute level to change. When safety levels are high (lower risk of collision), the robot can navigate with higher speed and acceleration. When safety levels are low (higher risk of collision), it needs to use lower speeds. In addition, throughout the mission, the battery level diminishes and with this the robot must navigate

Figure 3: MROS framework

Figure 4: MROS Design time activities

with lower speed and acceleration to save energy. MROS is used to adapt at runtime the navigation parameters, such as maximum speed and acceleration, to satisfy safety and energy constraints.

4.1 Application to MROS to the case study

Step 1: Identify the architectural variants corresponding to different configurations (parameter values) of the main node of the ROS1 navigation stack, and create its corresponding launch files. In total, 27 different function designs were defined. A snippet of the ROS launch file of one *Function Design* is shown in listing 1. The parameters that have been specifically defined for this *Function Design* are: *max_vel_x*, *max_vel_y*, *acc_lim_x*, *acc_lim_y*, *qa_safety*, and *qa_energy*. The QA values are specified as ROS parameters to enable, if necessary, to change them at runtime. The QA values defined in the launchfiles override the ones defined in the ontology.
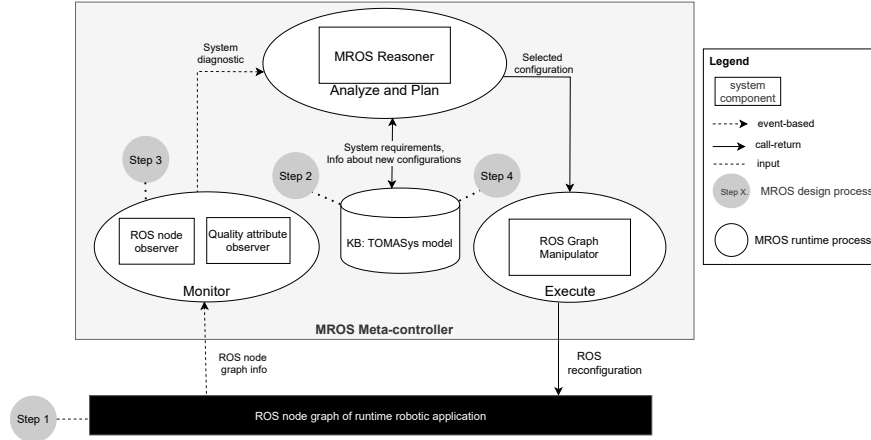
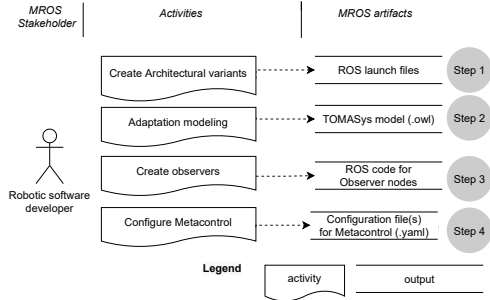**Listing 1: Function design 1 (out of 27)**

```
1  <launch>
2    <param name="qa_safety" value="0.7"/>
3    <param name="qa_energy" value="0.33"/>
4    <node pkg="move_base" type="move_base" name="move_base"
5      cwd="node" respawn="false" output="screen">
6      <param name="TrajectoryPlannerROS/max_vel_x"
7        value="0.3" />
8      <param name="TrajectoryPlannerROS/max_vel_y"
9        value="0.3" />
10     <param name="TrajectoryPlannerROS/acc_lim_x"
11       value="3.6" />
12     <param name="TrajectoryPlannerROS/acc_lim_y"
13       value="3.6" />
14   </node>
15 </launch>
```

Step 2: Model the architectural variants with the MROS TOMASys ontology. To create the OWL file for the KB, the graphical tool Protégé can be used to simplify the process. The TOMASys metamodel is available in the package mc_mdl_tomasys. It is only necessary to set up the application-specific ontology by creating individuals of the design time TOMASys classes (Figure 2).

For this case study, the following individuals are created:

- A *Function* individual for the navigation capability;
- *Quality Attribute Type* individuals for both safety and energy quality attributes;
- *Function Design* individuals for each variant that solves navigation, including object property individuals of *Quality Attribute Value* with their *Quality Attribute Type* and a data field with the expected QA value.

Step 3: Create observers with the templates provided in the ROS package rosgraph_monitor. These templates are two Python classes called *TopicObserver* and *ServiceObserver* that implement the general functionalities needed to monitor ROS topics and services, respectively. For each specific quality attribute that needs to be monitored, it is necessary to implement a new class that inherits from one of them.

For the use case, two Observers are implemented: *SafetyQualityObserver* and *EnergyQualityObserver*. A snippet of the implementation of the latter can be seen in listing 2. The class *EnergyQualityObserver* inherits from *TopicObserver*. In the initialization of the class, the topic to which the observer needs to subscribe and its

153

message type are defined. In line 4, it is defined that it needs to subscribe to the topic */power_load* to retrieve information about the battery, and that its message type is a float. For all observers, the method *calculate_attribute* must be overloaded, it is responsible for performing any necessary calculation with the data received via the topic defined in the initialization, it must return the final data as a key-value pair structured as a *diagnostic_msgs/DiagnosticArray* message. In the *EnergyQualityObserver*, in line 9, the method calculates the battery level as a normalized value, and returns it, for example, as {*energy*, 0.9}. The output of the observers is published in the topic *diagnostics*.

Listing 2: EnergyQualityObserver Observer snippet

```
1  class EnergyQualityObserver(TopicObserver):
2    def __init__(self, name):
3      #topic to observe and msg type
4      topics = [("/power_load", Float32)]
5      super(EnergyQualityObserver,
6        self).__init__(name, 10, topics)
7
8    def calculate_attr(self, msgs):
9      status_msg = DiagnosticStatus()
10
11     #normalized calculus for energy
12     attr =(msgs[0].data - 0.2)/(5.0-0.2)
13     print("normalized energy: {0}".format(str(attr)))
14
15     status_msg = DiagnosticStatus()
16     status_msg.level = DiagnosticStatus.OK
17     status_msg.name = self._id
18     status_msg.values.append(KeyValue("energy", str(attr)))
19     status_msg.message = "QA status"
20     return status_msg
```

Step 4: Configure Metacontrol through a yaml file. This is used to map each *Function Design* defined in the ontology (step 2) to its respective launch file (step 1). Additionally, to indicate which ROS nodes are killed and spawned during the reconfiguration process, as well, what actions and goals must be saved to be reset when the reconfiguration is performed. A snippet of the configuration file of this use case can be seen in listing 3.

Listing 3: Metacontrol configuration file

```
1  reconfiguration_action_name: 'rosgraph_manipulator_action_server
2  configurations:
3    f1_v1_r1:
4      command: roslaunch f1_v1_r1 f1_v1_r1.launch
5    f1_v1_r2:
6      command: roslaunch f1_v1_r2 f1_v1_r2.launch
7    f1_v1_r3:
8      command: roslaunch f1_v1_r3 f1_v1_r3.launch
9
10 kill_nodes: ['/move_base']
11 save_action: 'move_base'
12 goal_msg_type: move_base_msgs.msg.MoveBaseAction
```

4.2 Results

Bozhinoski et al. [3] show that by adding self-adaptation with MROS to this use case, the robot performance increases regarding the amount of time it violates its required safety and energy quality attributes, and the overall mission success. In average, safety violations decreases from 2.5% to 0.96%, energy violations from 2.98% to 1.86%, and mission success increases from 65.20% to 78.50%.

5 RELATED WORK

Aldrich et al. leverages predictive data models to enable automated robot adaptation to changes in the environment at run-time [2].

While the approach depicts the benefits of using models by capturing high-level artifacts, it makes it extremely challenging for a ROS developer to make use of them in robotic scenarios because: (1) it does not introduce models that can be reused for a different application; (2) it does not give insights on how to build similar models; (3) it does not provide infrastructure to leverage those models.

Cheng et al. propose a framework that uses GSN assurance case models to manage run-time adaptations for ROS systems [5]. The framework integrates assurance information from GSN models to ROS specific information to guide runtime monitoring and adaptation. It uses custom-developed libraries specific to the approach, rather than standard libraries in ROS (such as ROS Diagnostics) raising the entry barrier for ROS developers to effectively use it.

6 DISCUSSION AND FUTURE WORKS

This paper describes MROS - a tool that enables robots to perform self-adaptation at runtime based on ontological reasoning. MROS establishes generic self-adaptation mechanisms that drive self-adaptation through the MAPE-K reference feedback loop. This eases the process of designing self-adaptation for robots since it only requires users to define the proper observers, the Managed system ontological model conforming to TOMASys, a few configuration files, and the launch files for each architectural variant. Due to its reusability and extensibility, MROS has been used to handle different adaptation concerns in different robotic applications, such as reliable propulsion and motion control in underwater robots [1], contingency handling in mobile manipulators [3], and enhanced safety and energy saving in the navigation of a mobile robot [4].

REFERENCES

[1] Esther Aguado, Zorana Milosevic, Carlos Hernández, Ricardo Sanz, Mario Garzon, Darko Bozhinoski, and Claudio Rossi. 2021. Functional self-awareness and metacontrol for underwater robot autonomy. *Sensors* 21, 4 (2021), 1210.

[2] Jonathan Aldrich, David Garlan, Christian Kästner, Claire Le Goues, Anahita Mohseni-Kabir, Ivan Ruchkin, Selva Samuel, Bradley Schmerl, Christopher Steven Timperley, Manuela Veloso, et al. 2019. Model-based adaptation for robotics software. *IEEE Software* 36, 2 (2019), 83–90.

[3] Darko Bozhinoski, Mario Garzon Oviedo, Nadia Hammoudeh Garcia, Harshavardhan Deshpande, Gijs van der Hoorn, Jon Tjerngren, Andrzej Wąsowski, and Carlos Hernandez Corbato. 2022. MROS: runtime adaptation for robot control architectures. *Advanced Robotics* 36, 11 (2022), 502–518.

[4] Darko Bozhinoski and Jasper Wijkhuizen. 2021. Context-based navigation for ground mobile robot in semi-structured indoor environment. In *2021 Fifth IEEE International Conference on Robotic Computing (IRC)*. IEEE, 82–86.

[5] Betty HC Cheng, Robert Jared Clark, Jonathon Emil Fleck, Michael Austin Langford, and Philip K McKinley. 2020. 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*. 102–113.

[6] Carlos Hernández Corbato. 2013. *Model-based self-awareness patterns for autonomy*. Ph. D. Dissertation. Universidad Politécnica de Madrid.

[7] Carlos Hernández, Julita Bermejo-Alonso, and Ricardo Sanz. 2018. A self-adaptation framework based on functional knowledge for augmented autonomy in robots. *Integrated Computer-Aided Engineering* 25, 2 (2018), 157–172.

[8] Carlos Hernández, José L. Fernández, Guadalupe Sánchez-Escribano, Julita Bermejo-Alonso, and Ricardo Sanz. 2015. Model-Based Metacontrol for Self-adaptation. In *Intelligent Robotics and Applications*, Honghai Liu, Naoyuki Kubota, Xiangyang Zhu, Rüdiger Dillmann, and Dalin Zhou (Eds.). Springer International Publishing, Cham, 643–654.

[9] Carlos Hernandez Corbato, Zorana Milosevic, Carmen Olivares, Gonzalo Rodriguez, and Claudio Rossi. 2019. Meta-control and Self-Awareness for the UX-1 Autonomous Underwater Robot. In *Iberian Robotics conference*. Springer, 404–415.

[10] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

[11] Juliane Päßler, Esther Aguado, Gustavo Rezende Silva, Silvia Lizeth Tapia Tarifa, Carlos Hernández Corbato, and Einar Broch Johnsen. 2022. A formal model of metacontrol in maude. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 575–596.

[12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.