A Modeling Tool for Reconfigurable Skills in ROS

Bozhinoski, Darko; Aguado, Esther; Oviedo, Mario Garzon; Hernandez, Carlos; Sanz, Ricardo; Wasowski, Andrzej

DOI
10.1109/RoSE52553.2021.00011

Publication date
2021

Document Version
Final published version

Published in
Proceedings of the IEEE/ACM 3rd International Workshop on Robotics Software Engineering, RoSE 2021

A Modeling Tool for Reconfigurable Skills in ROS

Darko Bozhinoski*, Esther Aguado†, Mario Garzon Oviedo*,
Carlos Hernandez *, Ricardo Sanz †, Andrzej Wąsowski ‡

*Cognitive Robotics, TU Delft
†Universidad Politécnica de Madrid, Centre for Automation and Robotics
‡Software Quality Research, IT University of Copenhagen

2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE) | 978-1-6654-4474-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ROSE52553.2021.00011

***Abstract*—Known attempts to build autonomous robots rely on complex control architectures, often implemented with the Robot Operating System platform (ROS). The implementation of adaptable architectures is very often ad hoc, quickly gets cumbersome and expensive. Reusable solutions that support complex, runtime reasoning for robot adaptation have been seen in the adoption of ontologies. While the usage of ontologies significantly increases system reuse and maintainability, it requires additional effort from the application developers to translate requirements into formal rules that can be used by an ontological reasoner. In this paper, we present a design tool that facilitates the specification of reconfigurable robot skills. Based on the specified skills, we generate corresponding runtime models for self-adaptation that can be directly deployed to a running robot that uses a reasoning approach based on ontologies. We demonstrate the applicability of the tool in a real robot performing a patrolling mission at a university campus.**

Robots are expected to perform complex tasks autonomously, in dynamic environments under uncertainties that may influence execution. Known attempts to build autonomous robots rely on complex control architectures, often implemented on top of the Robot Operating System (ROS). Current designs for the deliberative layer are typically based on complex, distributed logic that requires application developers to be able to handle task coordination, contingency handling and system management. Most ROS robot software architectures follow a modular layered approach, where many of the components are standard libraries or software packages that can easily be reused. An example of that is the *System Modes* [1] library, which allows system architects to organize the deployment artifacts of the robot software architecture in (sub-)systems, hierarchically grouping ROS nodes into runtime modes to facilitate runtime configuration. However, while libraries like System Modes are an effective solution to deploy and configure the status of ROS nodes in use, it does not support complex runtime reasoning.

Robotics engineers need a reusable and maintainable design to support complex, runtime reasoning for robot self-adaptation, addressing contingencies and system management. This is especially important in safety-critical or mission-critical robotic systems where architectures and implementation artifacts undergo diligent quality and safety checks, and have to be maintained for extended periods of time. We have proposed the Metacontrol framework [2] to support runtime self-adaptation, this framework uses ontologies in order to represent and reason with runtime architectural models.

By using ontologies, it is possible to address most of the aforementioned challenges. They provide a formal naming and definition of concepts, properties and relationships to share a common knowledge of a domain. This contributes to a common understanding of complex phenomena in compound systems such as reconfiguration for mission completion. An example of this approach implemented in a group of robots sharing information can be found in [3].

The use of ontologies significantly increases code reuse and maintainability, as it separates and standardizes knowledge representation and reasoning from the rest of the system. Unfortunately, it also incurs a considerable cognitive cost for the developers, who need to formalize requirements into rules for a reasoner. To address this issue, we develop a language that allows a system architect to specify the architecture through the concept of a robot skill, which is then translated using model-to-model transformations into runtime models that can be directly deployed to a running robot. In this paper, we present a design tool that allows system architects to specify robot skills in a high level of abstraction. Based on the skills, we generate models for self-adaptation that are used by the Metacontrol framework to perform runtime reasoning.

In Section I, we introduce the Metacontrol framework and we illustrate the context for the tool. Section II, presents the tool, including a language to describe robot skills and the model-to-model transformations to generate the runtime models. Section III demonstrates the applicability of the language in a robot case study. Finally, Section IV discusses the technical aspects of the tool.

I. Metacontrol Framework

To develop self-adaptive solutions for robot software based on ROS we have created MROS2 (Metacontrol for ROS2 systems). MROS2 is based on three main components: i) The Metacontrol framework[4], which defines an architecture for self-adaptation at runtime based on models, ii) The RobMoSys[1] framework, which defines meta-models for robotic architectures, and iii) the System Modes concept [1], which provides abstractions for runtime system configuration and diagnosis in ROS2 systems.

The Metacontrol framework is the integrating component, it involves using the design-time architectural model of the

[1]https://robmosys.eu/wiki/

978-1-6654-4474-3/21/$31.00 ©2021 IEEE
DOI 10.1109/RoSE52553.2021.00011

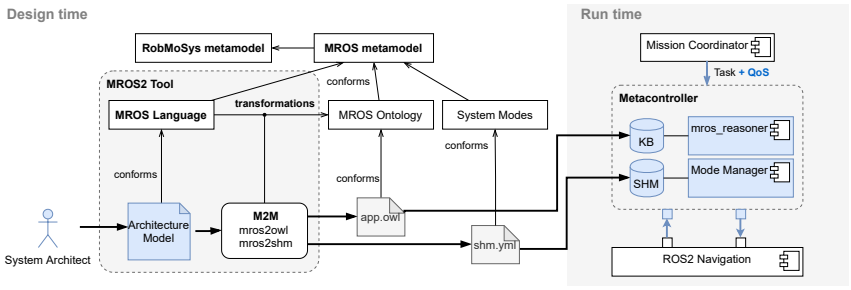

Fig. 1. Design Tool Overview

system to drive self-adaptation at runtime. This is achieved by defining two main elements: i) a platform and domain-independent metamodel, named Teleological and Ontological Model for Autonomous Systems (TOMASys), which models architectural variants in component-based systems, and includes semantics for architectural adaptation; and ii) the Metacontroller reference architecture for self-adaptation, which integrates ontological reasoning in a MAPE-K loop [2] to perform architecture analysis and adaptation decisions using the aforementioned metamodel.

In MROS2 we have mapped the architectural concepts in TOMASys to the ROS2 platform by creating a clear separation of the terminological part and general rules, the domain-specific knowledge of navigation concepts based on ROS2, and finally a set of application-specific individuals that model the functional and physical (component) architecture of the system. This structure for the knowledge base allows the reuse of both the ontology and the navigation-domain individuals. Further detail on the knowledge base structure can be found at [5]. The tool we developed allows users of the framework to leverage that knowledge without the necessity of mastering an additional, complex language.

To deploy a given configuration or to perform a reconfiguration, the Metacontrol framework uses the concept of System Modes. The System Modes extends the ROS2 node lifecycle concept[2] with the notion of (sub-)systems, hierarchically grouping nodes, as well as the notion of modes, to model architectural variants as different ROS2 node configurations. While System Modes are an effective solution to deploy and configure the status of the ROS2 nodes, it does not support complex runtime reasoning to relate the System Modes to the system task at runtime. For this, we have adapted the RobMoSys Skill metamodel and integrated it with the Metacontrol framework.

II. MROS Design Tool

In this section, we present an overview of the MROS2 design tool[3] to support the Metacontrol framework (Figure 1). It consists of: (i) a library of Skill definitions for navigation, (ii) the MROS Language and (iii) MROS transformations to runtime models for self-adaptation that can be directly deployed to a robot. *System architects* are the end-users of the tool. They can easily specify a robot system through a set of skill variants.

Domain-specific libraries facilitate the application of the Metacontrol framework to an application domain. We developed a library of Skill definitions for navigation, conformant to the MROS language and consistent with the ontologies in the Metacontrol framework. To extend the usage of skills beyond the navigation domain, other libraries need to be developed. Initially, a *domain expert* develops a set of conceptual rules for a domain, which are then encoded in an ontological form.

During the requirements phase, a *system architect* decides which skills should be used for the robot to be able to perform its tasks. He defines an architectural model of the application (Architecture Model in Fig. 1) using the MROS language. The language reduces the complexity of the application development process by minimizing the logical gap between the requirements specification and the system architecture.

Finally, the MROS transformations take as input the architecture model and automatically generate three runtime models that are used by the Metacontroller: the application-specific individuals (part of the ontology) used by the reasoner, the System Modes file (.yaml) used to deploy a suitable configuration at runtime and the observed components file (.yaml) used to monitor the status of the components at runtime.

A. MROS Language

The MROS Language is based on the MROS metamodel which is strongly aligned with the RobMoSys Metamodel[4]. In this context, for a *Task* to be executed, the robot needs to possess a certain set of skills. The language follows the conceptual decomposition of *Skills* in the RobMoSys Metamodel, which separates the specification of skill definitions with their corresponding skill realizations. In this direction, the MROS language has two main elements: (i) constructs to specify predefined skill definitions that are generic and platform-independent and (ii) skill realizations that are platform-specific elements (see Figure 2 for more details). The language was developed using the open-source Xtext[5] framework for domain-specific languages.

Conceptually, we link the concept of a *Task* that the robot needs to perform with the concept of a *SkillDefinitionSet*.

[2]https://design.ros2.org/articles/node_lifecycle.html
[3]https://github.com/MROS-RobMoSys-ITP/metacontrol_tooling
[4]https://robmosys.eu/wiki/modeling:metamodels:start
[5]https://www.eclipse.org/Xtext/

SkillDefinitionSet is the root language construct that consists of a number of *SkillDefinitions* - a set of functional system requirements that define what the system must be able to do. A *SkillDefinition* is characterized by a list of quality attributes relevant for the skill, optional input (*InAttribute*) and output (*OutAttribute*) attributes that depict the Skill dataflow and a result (*SkillResult*), which models the possible progress a skill can have at runtime (Figure 2(a)). The tool allows the architect to classify the progress of a *Skill* by categorizing results into one of the following: *INPROGRESS*, *SUCCESS* or *ERROR*.

It is important to note that MROS allows architects to model various SkillRealizations for each Skill. In this context, the system developer needs to provide ROS2 implementation of the skill realization. We define *SkillRealization* through: (i) the concept of a deployed system configuration (*SystemMode*); (ii) *QA Model* - a concept that provides information about the expected quality of a realization; (iii) *Context* - a concept that provides information about the settings in which the skill is realized.

A *SystemMode* is a concept that refers to a system configuration that can be deployed at runtime. It is a preexisting concept that comes from the integration of the System Modes library in the Metacontrol framework. It consists of a set of *Components* that can be realized as ROS2 nodes. Each *Component* is specified through a set of configurations where each *Configuration* has a number of properties (Figure 2(b)). To monitor a component during a mission execution, a tag *observable* should be assigned to it. This tag distinguishes the components that are subject to observation by the reasoner for adaptation. We consider that a component is observable if it satisfies the following requirements: (i) at each point of time, there is a clear separation between component states that correspond to failure, normal functioning and recovery; and (ii) the component provides an interface to detect failures/recoveries and trigger corresponding transitions.

Finally, a *QA model* is the quality performing a task expected from a skill realization. A Skill realization can contain a specification of a number of QA models (e.g. safety, performance etc.). Finally, a skill can be realized in different *Contexts*. It is important to specify a default value for the *Context* that corresponds to the normal system operation.

1) Language Guidelines: In this section we present guidelines on how to use the language. To monitor a component with an observer at runtime and instantiate it in the ontology, the system architect should assign the component a tag *observable*. Furthermore, in the MROS language, the concept of a component *Configuration* corresponds to the concept of a component *Mode* in System Modes [1]. Hence, the name of each configuration should be aligned to the ROS2 node lifecycle. Concretely, the name of a component *Configuration* must correspond to a state in the ROS2 node lifecycle (`inactive`, `unconfigured`, `finalized`, `configuring`, `cleaningup`, `shuttingdown`, `activating`, `deactivating`, `errorprocessing`), except when that state is active. In this case, the naming convention for a configuration is `active.*`, where * can be an arbitrary chosen name that corresponds to a mode.

Finally, it is important to note that each quality model (QA Model) in the Skill Realization should have a predefined type. The type must be listed in the Skill Definition quality attributes list. For example, if `safety` is not listed in the quality attributes of the skill definition, a corresponding skill realization can not have a quality model of type safety. This is why a system architect needs to provide an exhaustive list of quality attributes relevant for each skill definition.

B. MROS Model-to-Model transformations (M2M)

Using Model-to-model transformation techniques from the MROS2 system Architecture Model, we automatically generate three runtime models that can be directly deployed to a running system: the application-specific individuals deployed in the ontological Knowledge Base (KB), the System Modes file (.yaml) deployed in the System Modes library and the observed components file (.yaml) deployed in the system observers.

1) Application-specific individuals (KB): The application-specific individuals are part of the Metacontroller Knowledge Base, which is a runtime model specified in OWL. It consists of the following elements:

- *Components*: information on which running components influence the availability of skill realizations (function designs).
- *Function Designs*: Conceptually, it is equivalent to the definition of a skill realization. It contains information about the skill that is realizing and the expected quality.
- *Expected quality*: a concrete value of a quality attribute for a specific function design.

2) System modes: The System Modes file is a runtime model specified in .yaml format and consists of the following elements:

- *System structure*: contains the components (parts) that compromise the system
- *System Modes*: definition of the modes on a system level. Each system mode is defined through a set of components and their corresponding component modes.
- *Component modes*: definition of modes of the individual components. Component modes are defined through an assignment of different values to the corresponding component parameters.

3) Observed Components: An observer in the MROS framework needs information on components that should be monitored at runtime. As the last artifact that is generated through model-to-model transformation is a list of components defined in a .yaml file. This information is used to check for possible failures in the components of a running system.

III. Demonstration

To demonstrate the applicability of the language, we evaluated it in two case studies: a mobile robot that patrols the corridors of a university campus and an industrial consumer robot prototype (undisclosed). In this section, we focus on the

```
SkillDefinitionSet{
    Skill to_navigate {
        quality attributes [safety, performance, energy]
        input{DistanceToWall:float}
        result{
            success: "OK",
            error: "InternalError",
            inprogress: "IN_PROGRESS"
        }
    }
    Skill cover_area {
        quality attributes [safety, performance]
        input{AreaSize: float}
        result{
            success: "OK",
            error: "InternalError",
            inprogress: "IN_PROGRESS"
        }
    }
    Skill follow_wall {
        quality attributes [safety, performance]
        input{DistanceToGoal: float}
        result{
            success: "OK",
            error: "InternalError",
            inprogress: "IN_PROGRESS"
        }
    }
}
```

(a) Skill Definitions

```
Realization f_slow_mode {
    Component amcl {
        Configuration active.DEFAULT{
            transform_tolerance: 0.2,
            alpha1: 0.2, alpha2: 0.2,
            alpha3: 0.2, alpha4: 0.2, alpha5: 0.2
        }
    },
    Component bt_navigator {
        Configuration active.DEFAULT{
            default_bt_xml_filename: "navigate_w_replanning_and_recovery.xml"
        }
    },
    Component controller_server {
        Configuration active.SLOW {
            Path.max_vel_x: 0.1, Path.max_speed_xy: 0.1,
            Path.max_vel_theta: 0.5, Path.transform_tolerance: 0.2
        }
    },
    Component pointcloud_to_laser {
        Configuration inactive
    },
    Component laser_resender observable {
        Configuration active.DEFAULT{node_name: "laser_resender"}
    }
    Quality[energy: 0.3, safety:0.7,performance:0.4];
    context: Default
};
```

(b) A Skill Realization

Fig. 2. An example of skill definitions and a realization used in a mobile robot navigation scenario

architectural model for the robot that patrols the corridors at a university campus. The robot has to reach different locations, where each location corresponds to a different *Task*.

In this scenario, the architectural model of the robot consists of one SkillDefinition *toNavigate*. This SkillDefinition is implemented through 6 different SkillRealizations that differ in terms of the expected quality for three quality attributes: safety, performance and energy-efficiency. Each SkillRealization corresponds to a system variant that can be deployed at runtime. An example of a Skill Realization for *toNavigate* that has low performance and high safety is presented in Figure 2(b). The QA Models for each SkillRealization estimate the average score of an expected quality normalized in 0-1 range. In this demo, the safety level estimates the distance to obstacles, the performance level estimates the time to completion, and finally the energy level estimates the power needed to perform navigation in the selected configuration. The MROS2 tool automatically generates the 3 runtime models (Sect. II-B) ready for deployment in a running system: system modes file, application-specific individuals, observed components file.

While the robot performs the patrol, different contingencies might occur (e.g. laser errors, empty battery, reduced safety etc.). The Metacontroller observes the contingency type (if a component is broken or a quality attribute is violated) and selects a new configuration suitable for the current context.

IV. Technical Characteristics

Our tool aims to facilitate the specification of the system architecture. The Metacontrol framework (Sect. I) manages the knowledge base at three levels. The two top levels, the terminological and the domain-specific knowledge, are loaded as libraries, while the third level, the application-dependent ontology is output by our tool. This ontology could alternatively be created in any GUI-based tool like Protégé[6]. We reflect briefly on the difference in complexity of this alternative process to code directly the Metacontroller KB using ontology tools and our process using the MROS2 Tool, considering the effort required by a system architect implementing a metacontroller for a particular application.

In the navigation case study, we add eight *object property assertions*, three *data property assertions*, and four *class assertions* per each skill realization. Thus when an architect provides a design alternative for a skill, 15 relationships in Protégé would be created. If the designer prefers to edit the OWL XML file directly, 20 lines are needed per each skill realization. As the language is complex, these lines are added in several parts of the document. Our tool simplifies this task for systems architects dramatically. It allows the designer to add a skill realization using five lines of code, in a compact human-readable notation. Arguably, our language is also much easier to learn than the OWL language.

Acknowledgment: This work was supported by the RobMoSys-ITP-MROS (Grant Agreement No. 732410) project with funding from the European Union's Horizon 2020 research and innovation programme. Darko Bozhinoski acknowledges the support from the Belgian F.R.S.-FNRS.

[6]https://protege.stanford.edu/

References

[1] A. Nordmann, R. Lange, and F. M. Rico, "System modes – digestible system (re-)configuration for robotics." submitted to the 3rd International Workshop on Robotics Software Engineering, RoSE.

[2] C. H. Corbato, D. Bozhinoski, M. G. Oviedo, G. van der Hoorn, N. H. Garcia, H. Deshpande, J. Tjerngren, and A. Wasowski, "Mros: Runtime adaptation for robot control architectures," *arXiv preprint arXiv:2010.09145*, 2020.

[3] S. Niemczyk and K. Geihs, "Adaptive run-time models for groups of autonomous robots," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.

[4] C. Hernández, J. Bermejo-Alonso, and R. Sanz, "A self-adaptation framework based on functional knowledge for augmented autonomy in robots," *Integrated Computer-Aided Engineering*, vol. 25, no. 2, 2018.

[5] E. Aguado, Z. Milosevic, C. Hernández, R. Sanz, M. Garzon, D. Bozhinoski, and C. Rossi, "Functional self-awareness and metacontrol for underwater robot autonomy," *Sensors*, vol. 21, no. 4, 2021.