Specifying QoS Requirements and Capabilities for Component-Based Robot Software

Samuel Parra, Sven Schneider and Nico Hochgeschwender
Dept. of Computer Science, Hochschule Bonn-Rhein-Sieg, University of Applied Sciences
Sankt Augustin, Germany
samuel.parra@smail.inf.h-brs.de, sven.schneider@h-brs.de, nico.hochgeschwender@h-brs.de

***Abstract*—Assembling robotic multi-agent systems is becoming increasingly attractive due to the emergence of affordable robots. For coordinated missions such fleets usually have to communicate over unreliable channels and still achieve adequate performance. To support system designers in quantifying *adequateness*, in this paper we present a domain-specific language (DSL) that allows domain-experts to specify *(i)* quality of service (QoS) *requirements* of the communication channels; and *(ii)* QoS *capabilities* of the involved software components. Such QoS specifications complement the QoS management that has recently been introduced into ROS 2. To fully utilize this approach we have also developed an associated ROS 2 DSL which enables us to verify QoS specifications and provide feedback to the users already at design time. We have evaluated the developed language workbench following the Goal-Question-Metric (GQM) approach which demonstrates that the QoS DSL is complete with respect to ROS 2 and can be easily extended. Additionally, we generate a proof-of-concept implementation for a QoS monitor that can be seamlessly integrated into existing ROS 2 projects.**

I. Introduction

With the recent availability of low-cost, robust and commercial off-the-shelf robots such as unmanned aerial or ground vehicles it has become more attractive to assemble homogeneous or heterogeneous fleets of such robots. To achieve collaborative mission goals, the involved agents usually require mutual, intra-fleet communication. However, since the communication channels are usually established over the air, they are inherently unreliable and their quality changes over time. To let the robots' software cope with those challenges, designers introduce QoS management and configuration which enables the robots to autonomously *(i)* monitor the quality of communication channels; *(ii)* prioritize data streams; or *(iii)* reconfigure the producers and consumers, for instance, by compressing data. While those steps occur at the robots' runtime, they still rely on a design-time specification of the logical communication channels' *requirements* towards the physical network, e.g. the maximum tolerable delay, as well as the *capabilities* of the communicating end point, for instance, their buffer sizes.

The above description hints at two complementary views. First, the specification of QoS profiles and second, their application to a concrete communication framework. For the latter, we rely on established component-based software frameworks that have emerged as the de facto standard approach for implementing robotic software systems [1], [2]. Here, the components' implementation is agnostic about the communication middleware. One of the most popular frameworks is the Robot Operating System (ROS) [3] which targets multi-robot systems as those outlined above with its recent major upgrade named ROS 2. Due to its omnipresence in robotics software development, we have also chosen ROS 2 as the target platform in this paper.

Even if robotic software frameworks (RSF) have greatly contributed to managing the inherent complexity in robotic software systems, the development of robotic applications remains a complex and challenging task. One reason is that software component development and composition are only two — though important — of the many relevant domains in robots. Amongst the others one finds hardware design, control, kinematics or dynamics. Hence, robots are often developed by multidisciplinary teams of multiple members, each with possibly many development roles. From these circumstances many challenges arise, commonly leading to development mistakes and oversights, that cost time and effort to find and correct. Following fields such as the aviation or automotive industries, robotic developers have recently adopted model-driven methods and techniques to address such oversights. Here, models are first-class citizen that represents various views on a real system for documentation, analysis, verification or validation. In software engineering, another benefit is the transformation of models into software artifacts, ranging from individual configuration files to full code bases [4].

Related to models are *(i) meta-models* that define the available, yet abstract, concepts and constraints to create concrete models; as well as *(ii) domain-specific languages* (DSL) — concrete representations of meta-models, for example in textual or graphical notation, to be employed by domain experts.

One of the problems that we address in this paper is that ROS 2 is still inclined to the manual development of software and lacks formal, code-independent meta-models that facilitate model-driven approaches. To this end, we extend and adapt an existing meta-model for component-based software frameworks to also support ROS 2. As a second problem we have observed that QoS specifications in ROS 2 are only checked at runtime when the robots are already deployed. By leveraging a model-driven approach to formally represent QoS profiles, we can instead inform developers already at design time if certain QoS requirements are violated. Additionally, those models enable the generation of run-time monitors to detect QoS violations.

In summary, we make the following core contributions:

- We derive and implement the meta-models and DSLs to specify QoS profiles for component-based RSFs.
- We enrich an existing meta-model and DSL for modelling component-based software with ROS 2-specific concepts and constraints.
- We provide a language workbench that allows domain experts to specify models, perform constraint checks and generate code.

Following this introduction, Section II reviews the related work. Section III analyzes the required domains which are then realized as DSLs as explained in Section IV. After the evaluation in Section V we conclude the paper in Section VI.

II. Related Work

A. Modeling in QoS

Due to the ubiquity of wired and mobile phones, QoS management is predominantly found in the telecommunication domain where it targets communication and networking infrastructure in distributed systems. In an effort to abstract the performance engineers in those domains from the technical details of the underlying networks a considerable effort has been spent on the development of DSLs — also called QoS Modeling Languages (QML) — such as the HP QML [5], the QoS-MO ontology [6], the Contract Description Language CDL [7] or others as surveyed in [8]. The Data Distribution Service (DDS) QML (DQML) presented in [9] enables a model-based annotation of DDS [10] communication entities (e.g. publishers or subscribers) with specific QoS profiles. From such annotations configuration files can be generated. A limitation of the above DSLs is their lack of abstraction for component developers in robotic systems together with the general lack of tool support.

In the context of robotics, the RoQME project [11] focuses on a model-driven approach for the design-time specification of QoS metrics and their monitoring at runtime. Here, QoS metrics are system-level, non-functional requirements such as safety and performance.

B. Modeling of robotic software frameworks

Across various fields including control, electrical, mechanical or software engineering a recurring type of usually graphical models consists of *(i)* blocks; *(ii)* ports associated with those blocks; and *(iii)* connections between the ports. While they all feature the same representation, their semantics vary greatly from one field to another. NPC4 [12] is an effort to formalize the *structural* concepts and constraints of all those diagrams in the form of a domain-independent meta-meta-model. The Component-Port-Connector (CPC) meta-model [13] specializes the NPC4 meta-meta-model for the domain of component-based RSFs. There is a plenitude of RSF realizations that conform to the CPC meta-model with varying degrees of formalization: *(i)* ROS and OROCOS RTT [14] are examples of RSFs without formal models or formal models are just an afterthought; *(ii)* GenoM3 [15] or OpenRTM [16] are based on formal models; while *(iii)* V^{3}CMM [17] and RobotML [18] are RSF-independent modeling tools for component-based software derived from formal models.

State-of-the-art tools to realize DSLs and associated tooling, such as constraint checking or generators, are the Eclipse Modeling Framework (EMF) [19] or the JetBrains Meta Programming System (MPS)[1].

In this paper we employ the latter so to reuse the Component DSL[2] [20] for modeling framework-independent systems of software components. Here, components consist of ports, operations, and properties. They come in two "flavours", component types and component instances (similar to classes and their objects in object-oriented programming), to foster the reuse of component definitions. Only component instances can be connected with each other or receive concrete values for their properties.

While modeling a ROS 2 system we have observed two limitations of the Component DSL. First, it structurally constrains connections to a 1:1 cardinality i.e. exactly one source port can be connect to one target port. This limitation — which is, for example, also present in the CPC and the RobotML DSL — does not align with ROS' publish-subscribe messaging pattern that allows m:n connections between multiple sources and targets. The NPC4 meta-meta-model addresses this limitation by introducing the *connector* concept to connect an arbitrary number of ports. Second, the Component DSL lacks support for QoS settings. This is not merely a syntactic deficit, but instead also adds constraints that define valid compositions of components. For example, incompatible profiles will prevent ports from communicating, thus resulting in unwanted behaviour. In addition, the explicit declaration of QoS requirements enables the generation of code to monitor QoS conformance at runtime.

III. Domain analysis

Following the CPC meta-model, ROS allows to structure robotic software architectures by components (nodes), ports (publisher-subscriber or service client and server) and connections (topics or services). The original ROS version — "ROS 1" — uses a custom communication middleware that offers two types of transports to realize the connections, namely TCPROS and UDPROS, relying on TCP for reliable streams and UDP for best-effort datagrams, respectively. ROS 2 still conforms to the CPC meta-model, but replaces the custom communication middleware with DDS and also introduces backwards-incompatible API changes. The ROS middleware interface (RMW) acts as an abstraction layer to support DDS implementations from different vendors. This enables developers to choose an implementation that best suits their needs in performance.

One benefit of DDS is the support for QoS profiles, which enables a more fine-granular configuration of the communication middleware than the old transports in ROS 1. In particular, developers can choose the policies for each individual port, in

[1] https://www.jetbrains.com/mps/

[2] https://github.com/rosym-project/component-dsl

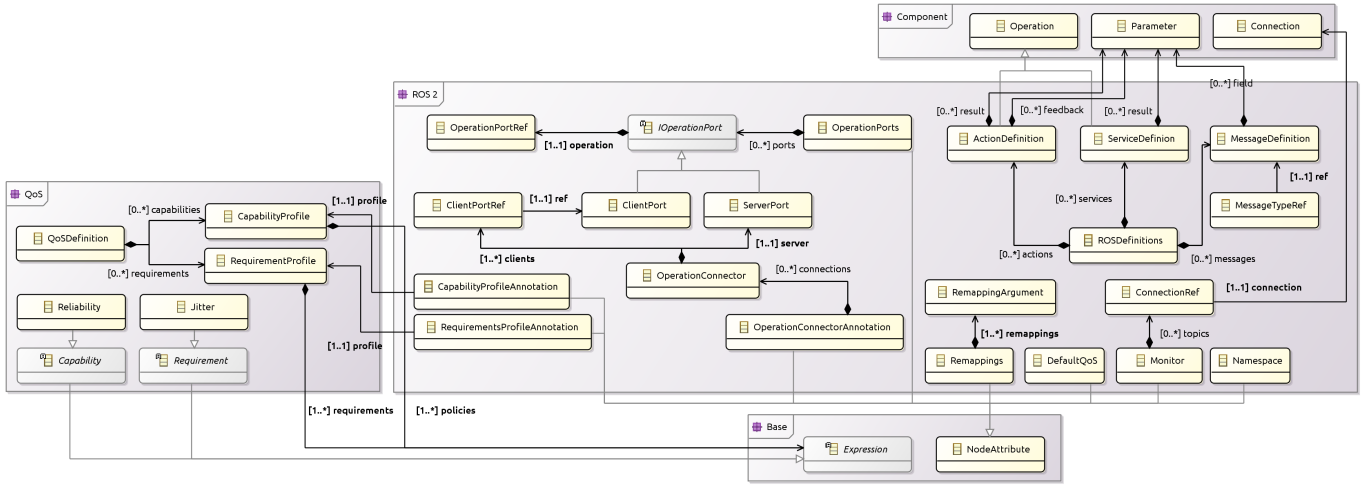


Fig. 1: Excerpt of QoS DSL (left) and ROS 2 DSL (center right) visualized in Ecore. Further capabilities extend the abstract *Capability* concept as exemplified by *Reliability* to be used in the profile model. Those profiles are annotated to ROS 2 *ClientPort*s, *ServerPort*s or *OperationPorts* via the *CapabilityProfileAnnotation*. The same holds analogue for requirements that are annotated to the *OperationConnector*. The ROS 2 DSL also adds concepts to represent ROS-specific interface definitions (messages, services and actions) and configurations such as *Remappings* or *Namespace*.

an effort to achieve better performance than with one global policy for all ports. The DDS standard [21] describes the supported QoS policies. However, ROS 2 only supports the following subset of those policies[3]: reliability, durability, history, depth, deadline, lifespan, lease duration and liveliness. If no policy is specified for a port, DDS falls back to the system's default which is however vendor-specific. To facilitate the communication between two ports, their QoS profiles must be compatible. The concrete definition of compatibility is defined in the standard, but roughly means that the policy requested by the subscriber must be equally or less restrictive than the policy offered by the publisher. Whenever a publisher and subscriber try to establish communication with incompatible QoS profiles, DDS will fire an event but ROS 2 will not notify the user by default. As a result, unless the developer configures a callback function for the specific event, there is no way to determine the reason why the connection could not be established.

In the next sections we will employ the following, non-normative definitions:

- A QoS *requirement* is specified on the logical connection between two components and has to be satisfied at runtime by the physical communication channel.
- A QoS *capability* is specified as a configuration setting on a communication channel's endpoint.

Those definitions reflect the protocol's view on the communication channels. However, it should be noted that there also exists a dual view (which we do *not* employ in the following) where capabilities are expressed on the physical channels and endpoints express requirements towards those communication channels.

IV. Domain-Specific Languages

The presentation of the developed DSLs follows MPS terminology, in particular a *concept* is a primitive in the MPS meta-meta-model similar to a class in Ecore or UML. We denote concepts by italic type. In the following we present our MPS workbench that consists of language modules for a generic QoS DSL, its specialization to DDS and a ROS 2 DSL that extends the Component DSL.

A. QoS DSL & DDS DSL

The QoS DSL is a minimal, yet extensible, declarative configuration language [22] for defining communication QoS profiles via QoS capabilities and requirements. While the language is designed to take advantage of the new QoS features in ROS 2, it is still DDS- and ROS-independent so that QoS profiles can be reused and specialized in different projects.

The *QoSDefinition* concept is composed of multiple *RequirementsProfile*s and *CapabilitiesProfile*s, as depicted in Fig. 1. These profiles are a collection of requirements and capabilities, respectively, which extend the expression concept from MPS' BaseLanguage[4], the meta-meta-model used to create all languages in the workbench. In this context, the collection has the semantics of a conjunction. One benefit of extending the expression concept is that the requirement definitions can be minimal with one requirement per statement, or complex with multiple requirements in a single statement using logical connectors, as shown in Fig. 2. At this point in time we do not yet check if the profiles themselves are consistent, for example, a requirement could be repeated with conflicting configuration values.

[3] https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/

[4] https://www.jetbrains.com/help/mps/base-language.html

```
QoS definition: turtle_qos_definition

Requirements profile:  wifi_connection
  Data rate > 300
  Jitter < 100 && Availability > 0.995

Capabilities profile: wifi_conn
  Depth == 20
  Durability == TRANSIENT_LOCAL
  Reliability == RELIABLE
  History == KEEP_LAST
```

Fig. 2: A simple QoS DSL model with the DDS DSL extension. The requirements profile consists of two statements: one with a single parameter, and one composed of two parameters connected with a logical connector. The capabilities profile exemplifies four policies from then DDS DSL.

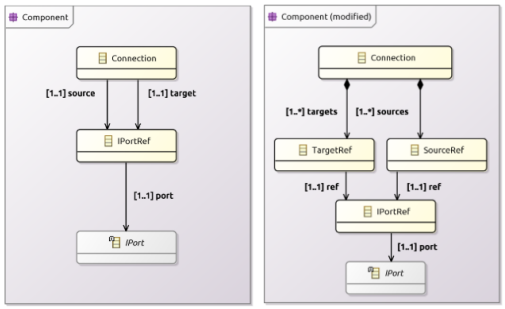


Fig. 3: Excerpt of the Component DSL showing the previous *Connection* concept (left) and our modification (right). The so-called smart references (i.e. the concepts with the suffix "Ref") are merely an artifact of a technical limitation in MPS that prevent 1:n cardinalities of references.

```
Component display
Package:
  simulations : Path to lib : /workspace/src/simulations

Ports:
  (InputPort) velocity_subsciber <TwistMsg> (QoS: wifi_conn )
  (InputPort) position_subscriber <PositionMsg>

Operations:
  boolean changeColorScheme ( new_color : string )

Properties:
  int width
  int height


<no lifeCycle>

Operation ports:
  (Server) color_scheme_server for: changeColorScheme (QoS: wifi_conn )
  (Client) move_to_point_client for: MoveToPoint
```

Fig. 4: Model of a component using the Component DSL with ROS 2 annotations. An input port (top) and an operation's server port (bottom) are annotated with a QoS profile called `wifi_conn`.

The different policies that compose a capability profile extend the concept *Capability*, and similarly the possible requirements extend the *Requirement* concept. This design choice facilitates the extension with further policies, as new languages can extend either of the two concepts. For instance, a new language can contribute other requirements by extending the *Requirement* concept. It is important that the additions extend the correct concept, as MPS constraints are used to ensure that a capability is not added to a requirement profile and vice versa. The current QoS DSL offers six pre-defined QoS requirements, namely: error rate, jitter, loss rate, data rate, availability, and delay. Those requirements cover most performance aspects of a connection [23] and align with the policies supported by ROS.

The DDS DSL conforms to the QoS DSL and adds DDS-specific constraints. First, where capabilities in the QoS DSL can be specified using arbitrary relational operators such as "equal to", "not equal to" or "less than", the DDS DSL instead only supports equality constraints (see Fig. 2). Second, it also constrains the numeric values associated with the profiles' capabilities.

B. ROS 2 DSL

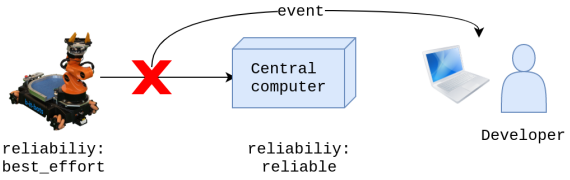
In order to model connections between multiple publishers and subscribers we slightly modify the Component DSL as visualized in Fig. 3. The changes conform to the NPC4 meta-meta-model with the addition of a directionality constraint as indicated by the names "source" and "target" in the new concepts. For RSFs such as OROCOS RTT, that only support 1:1 connections, a framework-specific MPS constraint must be added.

We employ MPS annotations to enrich the generic Component DSL with ROS 2 specific concepts and constraints. An annotation is one mechanism in MPS to extend a language by embedding concepts to models without introducing couplings between the original language and the extension [24]. The annotation concepts extend the *NodeAttribute* concept from BaseLanguage, as represented in Fig. 1. The annotation mechanism is used for attaching *(i)* structural ROS 2 elements (e.g. topics or services); and *(ii)* QoS profiles to the Component DSL. For the latter, the component's ports are annotated with the capabilities profiles as the QoS settings are configured in the publishers, subscribers, clients, and servers, whereas the connections — for topics, services, and actions — are annotated with the requirement profiles (see Fig. 4). Together, those languages enable us to model and validate ROS 2 applications.

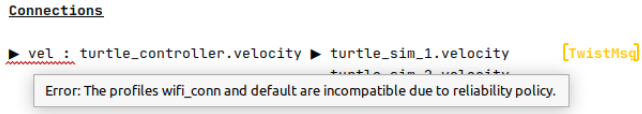
Beyond the graphical editors, we have also implemented code generators based on the MPS Text Generator plugin[5]. Those generators employ MPS' powerful template engine that extracts information from the developed models. In particular have we realized generators for three types of software artifacts: launch files in Python, configuration files in YAML, and monitor nodes in C++.

The monitor takes advantage of the topic statistics mechanism[6] in ROS 2 — a configurable ROS introspection mecha-

[5]https://jetbrains.github.io/MPS-extensions/extensions/plaintext-gen/
[6]https://index.ros.org/doc/ros2/Concepts/About-Topic-Statistics/



(a) With plain ROS 2 the incompatibility is identified at runtime, but the developer is only notified after *explicitly* registering a callback.



(b) With our toolchain the incompatibility is found at design time. The checking rule highlights the invalid connection to notify the user.

Fig. 5: Comparing development in ROS 2 and our toolchain

nism that provides certain metrics for topics. If monitors are specified, then per connection to monitor a node is generated and added to the launch description. The monitor currently supports the *jitter* and *delay* requirements expressed on the QoS requirement profile. It notifies the user whenever a violation of the requirement occurs. In its current state the monitor can only check for compliance of the requirements in topics.

Each monitor receives the statistics from the topic *<connection_name>/statistics*, and processes it to get the values for *jitter* and *delay*. These are obtained from the metrics message, which includes an average and a standard deviation for each metric measured[7]. Currently, the two metrics supported are message period and message age, both metrics are calculated by the subscribers and are measured in milliseconds. The delay is considered to be the average value of the message age, as the metric is calculated as the amount of time the message takes to reach the subscriber; the jitter is considered as the standard deviation of the message period, which is the time period between received messages.

As previously mentioned, most constraints on the composition of the system are already enforced by the Component DSL. One ROS 2 specific constraint is the compatibility of QoS profiles via an MPS checking rule that iterates through the sources and targets of a connection. The check is applied whenever the profiles for the source and target differ. Without the model-driven approach an incompatibility can only be found at runtime, rather than at design time as illustrated in Fig. 5a. Identifying this error is challenging because by default no notifications are raised. To determine if two profiles are compatible DDS uses a "Request vs Offered" model[3], where subscribers specify the policy values they will accept, and publishers specify the policy values they are able to offer. The two connect only if every requested policy is less or equally restrictive than the offered policy. The checking rule in the tooling resembles this mechanism by verifying that two

[7]https://github.com/ros2/rcl_interfaces/blob/master/statistics_msgs/msg/MetricsMessage.msg

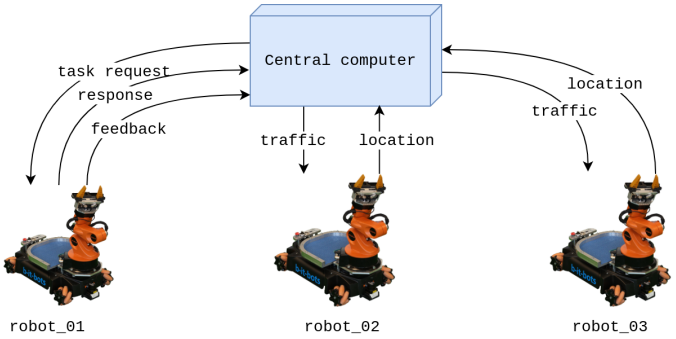


Fig. 6: Robots and their interaction in the case study.

policies do not share incompatible values. The only policies checked for, are those that can have incompatibilities; namely reliability, durability, liveliness, deadline, and lease duration. The system default value is considered as the value for any unspecified policy in the profiles, and can be modified with the *DefaultQoS* concept. If an incompatibility is found, the user receives an error message, as shown in Fig. 5b.

V. Evaluation

We employ the Goal-Question-Metric (GQM) approach [25] to evaluate the proposed DSLs and their corresponding tooling from a *language designer* and *language user* perspective. From the perspective of a *language designer* we evaluate attributes of the language, while from the perspective of the *language user* we evaluate the advantages and usefulness of the tooling. To this end, a case study (see Section V-A) is introduced which allows us to assess different evaluation objects such as models, constraint checks and generated artifacts.

A. Case Study

We consider a multi-robot system that manages the stock of a warehouse as our case study to exemplify the model-driven engineering support for ROS 2 (see Fig. 6). The system is composed of a central computer and a team of distributed mobile robots communicating with the central computer via a Wi-Fi network. The role of the central computer is to assign tasks to individual robots. Each robot shares its location and task status with the central computer. The setup implies two challenges with respect to timely and reliable information processing in such a distributed system. First, the strength of the Wi-Fi network throughout the warehouse will vary due to both the distance between the robot and the routers and the internal structure of the warehouse. Second, the messages exchanged between the robots and central computer vary with respect to their frequency, size and priority.

B. QoS Capabilities

To ensure a proper performance of the system in the presence of the challenges mentioned above, we associate capability profiles to several ROS topics in the envisioned system. An overview of the profiles employed in the case study is shown in Table I. The profile values shown in green are briefly discussed in the following paragraph.

Policy	default	task_action	location_robot	location_central	traffic_conn
Reliability	reliable	-	best effort	best effort	best effort
History	keep last	-	-	-	-
Depth	10	-	1	-	1
Durability	volatile	transient local	-	-	transient local
Deadline	+inf	-	10 s	10 s	30 s
Liveliness	automatic	-	-	-	-
Lease duration	+inf	10 s	10 s	10 s	30 s

TABLE I: Capability profiles for the case study. Dashes represent that the default value will be used.

The individual robots share their position in the warehouse with the central computer over the `location` topic. For this topic the `Reliability` policy is set to "best effort", since the location message is sent regularly by the robots and this configuration value ensures that the received messages are always the latest ones. Instead, a "reliable" policy could cause the retransmission of messages in non-ideal networks and hence result in outdated data. Since the location constantly changes the `Durability` is set to "volatile" to ensure that the central node does not assume a wrong location if it disconnects and reconnects. The central node is able to receive more messages (from different robots) that it can handle at a certain time, thus the robots and central node use different profiles in order to specify different depths of message history. The `Depth` in the profile for the central node is the default value, whereas the depth in the robot's profile is 1. The `traffic` topic is used to re-route and re-schedule robots in the warehouse. The `Deadline` and the `Lease duration` for this topic is 30 seconds and `Durability` is set to "transient local" such that reconnecting or newly-joining robots can receive some traffic information, even if it is not accurate. It is important that the central computer knows when a robot is unresponsive as soon as possible, hence the `Deadline` and `Lease duration` for the location topics is shorter.

C. QoS Requirements

We consider the QoS requirements in [26] for telemetry operations relevant for our case study as the robots continously share their position and other status information. According to Chen et al. [26] telemetry operations require real time performance, for which small delays are acceptable. Thus, for the location and traffic topic the maximum acceptable delay is 250 ms. The maximum acceptable jitter for the traffic topic is 100 ms. For the location topic the acceptable jitter has to be greater, as there will be more variability in the sending of the messages by the robots. Therefore, the jitter requirement is set to 300 ms.

D. Modeling languages

To model the component architecture considered in the case study we employed the modified Component DSL and the ROS 2 extensions through one system model, two component models, and one ROS definitions model. To evaluate the goal expressed in Table II we evaluate Q1 using the metrics M1 through M3. The ROS 2 extension consists of three languages

Goal	Purpose	Extend
	Issue	with ROS 2 structural aspects
	Object	the Component DSL
	Viewpoint	Language designer
Question	Q1	Which ROS 2 specific structural aspects were added in the extension?
Metric	M1	Number of changes to the Component DSL
	M2	Number of structural concepts added in ROS 2 DSL
	M3	Number of QoS annotations added to models
Question	Q2	Does the modifications to the Component DSL achieve a more general meta-model?
Metric	M4	Number of messaging patterns that can be modeled.

TABLE II: GQM analysis on the ROS 2 DSL.

and one modification of the Component DSL (M1). The ROS 2 DSL contains 22 additional structural concepts, such as the interface definitions and the operation ports (M2). Since one important addition in ROS 2 is the specification of QoS policies in publishers and subscribers, the QoS DSL and DDS DSL are included, which contain 19 and 12 structural concepts, respectively. The profiles defined with the QoS languages are referenced by two annotations in the component and system model (M3).

To evaluate the question Q2, we examine metric M4. The number of messaging patterns that can be modeled by the Component DSL without the ROS 2 extension remains one; however, with the modification the meta-model is less restrictive. Before the modification the Component DSL could model a publish-subscribe pattern with only two participants, which are the source and the target. With the proposed modification, connections with more participants can be expressed with the meta-model. This improves expressiveness concerning ROS without negatively impacting the modeling of connections in other frameworks, as models expressed with the original meta-model are still valid. Constraints can be implemented by framework-specific extensions to validate the number of participants in a connection.

Evaluating the goal for the QoS DSL is done through the analysis presented in Table III. For Q3 we consider the extensible concepts in the meta-model (M5). The QoS DSL was designed to be extensible so that language designers can expand the language whenever it is considered incomplete. To do so, the designers can extend either the *Capability* or the *Requirement* concept to add the relevant missing concepts. To answer Q4, we look into the metrics M6 and M7. The current

Goal	Purpose	Create
	Issue	an extensible
	Object	configuration language for QoS profile specification
	Viewpoint	Language designer
Question	Q3	How does a language designer extend the capabilities and requirements offered?
Metric	M5	Extensible concepts
Question	Q4	How complete are models with the current version of the QoS DSL?
Metric	M6	Number of capability concepts
	M7	Number of requirement concepts

TABLE III: GQM analysis the QoS DSL.

Goal	Purpose	Express
	Issue	a sound ROS 2 system with QoS settings
	Object	with the set of models created with ROS2 DSL and DDS DSL
	Viewpoint	Language user
Question	Q5	Which constraints and checking rules apply?
Metric	M8	Number of constraints and type rules checked for the models

TABLE IV: GQM analysis of constraints and type rules.

version of the meta-model offers eight capabilities (M6) and six requirements (M7). With the provided concepts a language user can model a complete QoS profile in ROS, but other frameworks and middleware may require more concepts for completeness. Similarly, the provided requirements may be insufficient to capture all performance related aspects.

E. Tooling

To assess the goal expressed in Table IV, we consider the question Q5 and the metric M8. The extension enforces five constraints on the models in addition to the constraints enforced by the Component DSL. These five constraints enable specifying a ROS 2 system with QoS settings that is sound, meaning that if implemented according to the models the resulting system will be error-free. The new constraints are:

1) Component instances must have a unique name.
2) Two distinct QoS capability profiles must be compatible for communicating.
3) The durability policy must not have “persistent” or “transient” as value.
4) The liveliness policy must not have “manual by participant” as value.
5) The policies deadline, depth, lease duration, and lifespan must have a positive integer value.

The last constraint (5) is enforced in the DDS DSL with four constraint concepts; whereas the constraints 1–4 are implemented in the ROS 2 DSL with one constraint concept and eight checking rules.

Code generation is evaluated with the GQM presented in Table V. From the models the toolchain generates three types of files (M9), namely the launch file, the configuration files, and the monitor. The number of files generated depends on the models: for each system model there is one launch file, for each component instance with properties there is one configuration file, and for each connection in the system to monitor there is a monitor component. Hence, for the case study six files are generated: one launch file, three configuration files, and two monitor components. The launch file is a Python script that returns a launch description with all the component instances and their configurations. The script is able to find the configuration files for the components when these are located in the `/config` directory of the ROS project.

Goal	Purpose	Generate
	Issue	a launch file, configuration files, and a monitor
	Object	from the models
	Viewpoint	Language user
Question	Q6	Which files are successfully generated?
Metric	M9	Generated files

TABLE V: GQM analysis of generated files.

Goal	Purpose	Generate
	Issue	a monitor to test at run-time the compliance
	Object	to the specified QoS requirements
	Viewpoint	Language user
Question	Q7	Which requirements can be monitored?
Metric	M10	Monitored requirements

TABLE VI: GQM analysis of the monitored requirements.

For evaluating the goal asserted in Table VI we consider the question Q7 and the metric M10. The monitor is a C++ ROS node that listens to the topic statistics generated by the subscribers and checks for each requirement that contains either the metrics jitter, delay or both (M10). Currently, the node only notifies the user whenever the requirements are violated. However, the developer is free to add additional logic to propagate errors through the system. The metrics are derived from the topic statistics mechanism of ROS, the developer is left to configure the subscribers correctly for the monitor to receive messages.

The performance of the generated node as a requirements monitor is acceptable with room for improvements. First, the measurements are imperfect since these values are calculated as a moving average, meaning that delays will not be caught immediately. Second, because of the moving average the values for delay and jitter will be maintained for a period of time, meaning that the monitor will still conclude that the requirements are being violated even if the bad network conditions are no longer present. Third, the monitor should support more metrics to give more information to the system and to the developer regarding network conditions. Finally, in its current state the monitor can only supervise the communication between publishers and subscribers, as the statistics mechanism is unavailable for clients and servers.

VI. Conclusion

We introduced an MPS language workbench for ROS 2 systems as an extension of the Component DSL which allows the user to specify and validate the QoS settings already at design time. The workbench comprises four main DSLs: *(i)* A

reused framework-independent Component DSL with slight modifications to support m:n connections between components. *(ii)* A ROS 2 DSL which enables developers to annotate models of the Component DSL with ROS 2 specific concepts. *(iii)* A newly-developed QoS DSL capturing framework- and middleware-independent QoS concepts and constraints. *(iv)* A new DDS DSL conforming to the QoS DSL by adding DDS-specific concerns. The workbench includes code generators associated with the ROS 2 DSL to generate launch files, configuration files and monitoring components. The meta-models introduced in this paper support different roles in the robotic development process. Still, the QoS DSL and the DDS DSL mainly support the responsibilities of the performance designer[8], who must configure performance related aspects of the system. By creating QoS profiles, performance designers can have an impact in the performance of the communication middleware, which is essential in a ROS application. Following the GQM evaluation method we have shown that our model-driven approach can benefit ROS 2 designers and developers by increasing the expressiveness of the Component DSL. By means of constraint checks and text generations, oversights can be identified and corrected at design time; and network conditions can be monitored at runtime. We are planning to extend the code generators to further ROS 2 artifacts (e.g. package description and interface definitions). In addition, it is worthwhile to investigate how round-trip engineering can support developers by updating the models based on changes in such ROS 2 artifacts. Even though the monitor is already helpful for developers to understand how the system performs we plan to extend the monitor to metrics such as availability, error rate, data rate, or loss rate. Finally, we plan to study the generality and completeness of the languages as well as the tooling in a more elaborate evaluation.

ACKNOWLEDGMENT

The authors gratefully acknowledge the on-going support of the Bonn-Aachen International Center for Information Technology. Sven Schneider has received a PhD scholarship from the Graduate Institute of the Hochschule Bonn-Rhein-Sieg. This work was supported by the European Union's Horizon 2020 project RobMoSys (grant agreement No 732410),SESAME (grant agreement No 101017258) and AAIP SafeMUV project.

REFERENCES

[1] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, 2009.

[2] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (Part II)," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 100–112, 2010.

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *IEEE Intl. Conference on Robotics and Automation. Workshop on Open Source Software*, 2009.

[4] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.

[8] https://robmosys.eu/wiki/general_principles:ecosystem:roles:performance_designer

[5] S. Frolund and J. Koistinen, "QML: A Language for Quality of Service Specification," Hewlett-Packard Software Technology Laboratory, Tech. Rep. HPL-98-10, 1998.

[6] G. F. Tondello and F. Siqueira, "The QoS-MO ontology for semantic QoS modeling," in *ACM Symposium on Applied Computing*, 2008.

[7] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, "Specifying and measuring quality of service in distributed object systems," in *Object-Oriented Real-Time Distributed Computing*, 1998.

[8] J. Jin and K. Nahrstedt, "QoS specification languages for distributed multimedia applications: A survey and taxonomy," *IEEE Multimedia*, vol. 11, no. 3, pp. 74–87, 2004.

[9] J. Hoffert, D. Schmidt, and A. Gokhale, "A QoS policy configuration modeling language for publish/subscribe middleware platforms," in *Intl. Conference on Distributed Event-Based System*, 2007.

[10] G. Pardo-Castellote, "OMG Data-Distribution Service: architectural overview," in *Intl. Conference on Distributed Computing Systems Workshops*, 2003.

[11] C. Vicente-Chicote, D. García-Pérez, P. García-Ojeda, J. F. Inglés-Romero, A. Romero-Garcés, and J. Martínez, "Modeling and Estimation of Non-functional Properties: Leveraging the Power of QoS Metrics," in *From Bioinspired Systems and Biomedical Applications to Machine Learning*, 2019.

[12] E. Scioni, N. Hubel, S. Blumenthal, A. Shakhimardanov, M. Klotzbucher, H. Garcia, and H. Bruyninckx, "Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain Specific Language NPC4," *Journal of Software Engineering for Robotics*, p. 20, 2016.

[13] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS component model: a model-based development paradigm for complex robotics software systems," in *ACM Symposium on Applied Computing*, 2013.

[14] H. Bruyninckx, P. Soetens, and B. Koninckx, "The Real-Time Motion Control Core of the Orocos Project," in *IEEE Intl. Conference on Robotics and Automation*, 2003.

[15] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *IEEE Intl. Conference on Robotics and Automation*, 2010.

[16] N. Ando, T. Suehiro, and T. Kotoku, "A Software Platform for Component Based RT-System Development: OpenRTM-Aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, 2008.

[17] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, "V^{3}CMM: A 3-view component meta-model for model-driven robotic software development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010.

[18] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, 2012.

[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, Second Edition*, E. Gamma, L. Nackman, and J. Wiegand, Eds. Addison-Wesley, 2008.

[20] D. L. Wigand, A. Nordmann, N. Dehio, M. Mistry, and S. Wrede, "Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case," *Journal of Software Engineering for Robotics (JOSER)*, vol. 8, no. 1, pp. 45–64, 2017.

[21] *Data Distribution Service (DDS) – Version 1.4*, Object Management Group (OMG) Std., 2015. [Online]. Available: http://www.omg.org/spec/DDS/1.4

[22] M. Voelter, *Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.

[23] I. Papaioannou, D. Tsesmetzis, I. Roussaki, and M. Anagnostou, "A QoS ontology language for Web-services," in *Intl. Conference on Advanced Information Networking and Applications - Volume 1*, 2006.

[24] M. Voelter, "Language and IDE Modularization and Composition with MPS," in *Generative and Transformational Techniques in Software Engineering IV: International Summer School*, R. Lämmel, J. Saraiva, and J. Visser, Eds. Berlin, Heidelberg: Springer, 2013, pp. 383–430.

[25] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, *Goal Question Metric (GQM) Approach*. John Wiley & Sons, Inc., 2002.

[26] Y. Chen, T. Farley, and N. Ye, "QoS Requirements of Network Applications on the Internet," *Information Knowledge Systems Management*, vol. 4, no. 1, pp. 55–76, 2004.