Lifting ROS to Model-Driven Development: Lessons Learned from a bottom-up approach

Nadia Hammoudeh Garcia*, Harshavardhan Deshpande†, Ruichao Wu‡ and Björn Kahl§
Fraunhofer Institute for Manufacturing Engineering and Automation (IPA)
Nobelstr. 12, 70569 Stuttgart, Germany
*nadia.hammoudeh.garcia@ipa.fraunhofer.de,
†harshavardhan.deshpande@ipa.fraunhofer.de,
‡ruichao.wu@ipa.fraunhofer.de, §bjoern.kahl@ipa.fraunhofer.de

Andreas Wortmann
Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW)
University of Stuttgart
Seidenstr. 36, 70174 Stuttgart, Germany
wortmann@isw.uni-stuttgart.de

2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE) | 979-8-3503-0184-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/RoSE59155.2023.00010

***Abstract*—The benefits of using model-based technologies are well proven in different sectors. However, in robotics, where the dominant framework is one whose nature is to create the code by hand, MDD approaches are struggling to gain a foothold. Our approach is distinctly different from traditional MDD solutions, we have created models and tools with a bottom-up perspective, i.e. analyzing " everyday" code and encapsulating this existing code in models. In this paper we present the lessons learned from (1) designing the models to encapsulate ROS hand-written code, (2) creating tools to ease the production and debugging of complex systems using the models, and (3) applying them to real use cases. Through this work we have identified in which situations it is worth using model-based technologies and in which situations the traditional approach is more meaningful. Our aim is to give a broad view of our conclusions and inspire the design of new model-based tools for the assistance of roboticists.**

***Index Terms*—MDD, robotics, ROS**

I. Introduction

In many established domains, such as the automotive sector, model-based technologies are used on a daily basis to optimize software development. Among other features, model-driven efforts facilitate software portability and interoperability and can automate some steps by generating code, as well as validate the implementation while designing and reducing the number of defects of the final code.

In robotics, since its release in 2007, the Robot Operating System (ROS) [19] has dominated software development. ROS has expanded very quickly, in part due to its low barrier to entry, rapid prototyping characteristic, and the use of popular programming languages such as Python or C++. Unfortunately, within the ROS community, the typical developer codes manually for rapid deployment [21]. There is no culture of modeling the design before the implementation phase, an approach more adequate for industrial applications.

Traditional software development, assisted by model-based tools, has a top-down lifecycle, i.e., firstly, the models are created and, secondly, from these models new code is created. Several initiatives, like the EU-project BRICS (Best Practice in Robotics) [3], have tried to introduce Model-Driven Development (MDD) in robotics, but the low acceptance of the ROS community to adopt them has resulted in little success.

After an experience with ROS by collaborating on projects together with industrial partners in various sectors such as logistics or production, and being aware of its potential and its incursion into industrial domains [22], we have analyzed the factors behind the community's displeasure of using models and worked on a solution, a way of introducing models, that would make the models more user-friendly for the typical ROS developer.

That is why we have opted for a bottom-up approach, i.e., starting from existing code and transforming it into models. In this way we enable the developer to continue implementing the code manually, but all the relevant information is translatable into models that can be used, e.g., for testing, composition or to generate new packages.

In this paper, we convey the lessons we have learned during the development process of our models and the tools to operate them. This includes both the problems we have encountered in trying to represent ROS concepts in models and the benefits we have been able to gain from MDD to improve ROS. Our findings may serve to inspire future work in this field, in particular, to establish where MDD may be most valuable and where it may not be worth the effort.

II. Background

A. ROS

ROS [8] is characterized by its little architectural constraints, and a federated development model.

The most well known aspect of the ROS architecture is the *computation graph*. ROS systems are composed by a set of processes called *nodes*. The nodes run all at the same time and can communicate among them with three types of mechanisms: the *topic*, *service* and the *action* patterns.

The objects of the communication (e.g., messages, services and actions types) are defined using language-independent data structures composed of common data types (String, Double,

979-8-3503-0184-7/23/$31.00 ©2023 IEEE
DOI 10.1109/RoSE59155.2023.00010

Int, Boolean ...). *Launch files* are used to start a group of nodes, as well as to set global parameters and their values.

Since its launch in 2007, this framework has been widely supported by the robotics community. In 2016 the new version, ROS 2, was released, with the main goal of adapting ROS to new needs, such as improving the quality and the security of the systems.

B. Kinematics Modeling

Unified Robotics Description Format (URDF) is an XML specification used to model multi-body systems.The specification covers the kinematic and dynamic descriptions of the robot, the visual representation of the robot, and the collision models of the robot. The advantage of having a non-interpreted data exchange format is that the robot system can be described independent of the code, which can then be reused and consumed in many different softwares like collision checking and dynamic path planning. A parser parses this URDF file and populates C++ or Python URDF data structures. There are some limitations with URDF. The most notable ones are

- robot description cannot be changed (immutability),
- only tree structures (no loops),
- no sensor models, and
- low reusability of URDF files (composability).

The first three limitations are out of the scope of this paper. The problem of composability is solved with the widely used Xacro, an XML macro language. The Xacro file has to be dynamically converted into a URDF file before being consumed by the target software. Syntax and parser errors happen for a variety of reasons during this conversion process resulting in an exception thrown by Xacro tools. Many other non-interpreted data exchange formats like SDF [9] (Simulation Description Format, an XML format that describes objects and environments for robot simulators, visualization, and control) and COLLADA [2] (COLLAborative Design Activity, an interchange file format for interactive 3D applications) and also interpreted forms (Python or Ruby based) have been proposed, though they have not been accepted by the wider community.

III. Related Work

A. MDD approaches in robotics

The BRICS (Best Practices in Robotics) [14] project joined a model-driven approach with a separation of concerns paradigm. Also, they introduced the BCM (BRICS Component Model) which serves as a high abstraction model to describe the characteristics of the robotics software independently of the actually deployed framework. The BRICS Integrated Development Environment (BRIDE [15]), bridged BCM and ROS using model-to-text (M2T) transformations to generate ROS skeleton code.domain expert.

There are other examples of how we can generate ROS applications, starting from modeling languages. A previous effort uses AADL to represent and generate ROS code [13], a case study of this approach shows the complexity of this type of solution because of the high degree of expertise required. Along the same lines of system architecture representation but with a more sophisticated approach [12], domain-specific M2M transformations can be used to optimize code generation, and more easily scale the robotic application to be implemented.

All of these efforts follow a traditional MDE top-down approach with the generation of boilerplate code from the models, but they do not support the import of existing code created manually.

The most recent community attempt is the RobMoSys project [6]. This project promoted the use of MDE techniques through cascaded funding initiatives. The RobMoSys project enables the use of their conceptual models through two toolchains:

- SmartSoft [20], a service-oriented component-based approach for the definition of the full robotic application. It provides code generators, validators, and textual and graphical editors for robot system designers, component developers, and others.
- Papyrus for Robotics [5], apart from being a solution to design, simulate and deploy robot applications, it checks safety measures by performing dysfunctional analysis on the components.

The RobMoSys project has made great improvements in interoperability between frameworks, as well as in the separation of roles by offering different perspectives of its tools. However, it has not accomplished major results in the integration and re-utilisation of existing code and systems.

IV. Context and practical use of our approach

All the work presented in this section is open-source[1]. To get a deeper understanding of our metamodels please check our previous publication in the topic [18]. Concrete experiments and examples applying the technology are also publicly available [17][2].

A. Bottom-up approach. From code to models.

In our case, we divided the development into two parts. Firstly we implemented the metamodels and secondly tools that allow us to automatically convert existing software into models, called extraction tools.

1) Metamodels: The first step was to determine which parts of the information in a ROS package we want to cover with our models. In our case we relied on three aspects:

1) The information represented in a ROS package containing the source code of a node.
2) The information of a running system once the launch file has been started.
3) The description of the robot's kinematics.

Apart from that, we have to cover two dimensions, on the one hand, the basic concepts that characterize a ROS package (node, topic, message, service, etc.) and on the other hand,

[1]https://github.com/ipa320/ros-model and https://github.com/ipa320/kinematics-model

[2]https://github.com/ipa-nhg/ros-model-examples

the generic concepts that we want to represent (interface, port, component, systems, etc.).

For the ROS aspects, it was sufficient to analyze the most representative ROS packages as well as the existing documentation and Wiki to identify all the information that should be represented in the models.

The kinematics in ROS is a special case since it uses the URDF format (Sec. II-B), which is already a model. But the specification for the model exists only as outdated Wiki pages [10] and as hardcoded data structures along with validation functionality in the C++ and Python parsers. Even though an attempt has been made to formalize URDF [11], it has never been utilized.

For generic concepts, we looked for existing standards to take as a basis and inspiration. Among them, we opted for the OMG standard about "Deployment and Configuration of component-based Distributed Applications" [16], in addition to it, we studied the models created by other initiatives like the BRICS and RobMoSys.

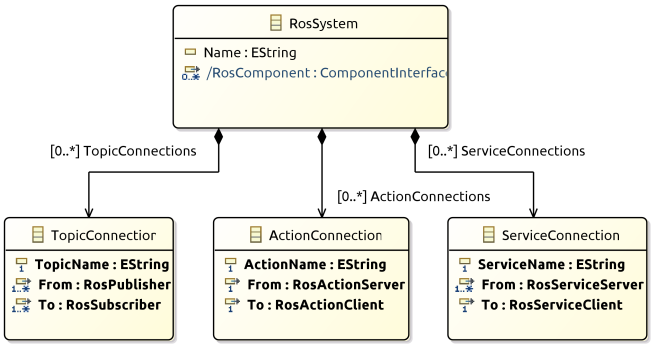

Fig. 1: Simplified representation of the RosSystem metamodel

We decided then to design three types of models:

1) Representation of the information in a package, our **ROS model**. The filesystem information, the definition of the communication objects (i.e., messages, services, and action types) as well as the description of the nodes, their names, which topics, services, and actions they use, and the parameters they consume and set.
2) Representation of a system, our **RosSystem model**. A system is a composition of components, where a component can be one or several nodes. This model references the nodes of the ROS model. Fig. 1 shows a simplified version of its metamodel.
3) Representation of kinematics information, our **Kinematics model**. The kinematic (links and joints) and the dynamic description (inertia) of the robot, the visual representation of the robot, and a collision model of the robot.

2) Extraction technologies: In our view, one of the reasons behind previous MDD efforts were not well adopted in the ROS community is because they are based on generating new code from the models, while in ROS there is already a large catalog of open-source components. The usual practice in this framework is to reuse existing code. Therefore we put a significant effort into the automatic extraction of models from existing code.

We do this in three different ways: (1) Using static code analysis for C++ and Python, (2) with observers to the ROS *rosgraph* during the execution time, and (3) parsing existing models like URDF format files.

B. Practical applications

In this section, we will give a brief overview of the most representative tools we have created based on our models.

1) Composability / Integration: We have created tools for the composition of both ROS nodes and physical components at the kinematic level. A graphical interface allows to import of models of nodes and kinematics components and easily builds the connections between them. In addition, a compiler takes care of validating that the connections are correct (e.g., the same type of message by the publisher and the subscriber) and finally automatically generates the artifacts of the composition, i.e., launch files for the system and xacro files for the kinematics.

This approach has been successfully tested with different robot systems with different levels of complexity[3].

Furthermore, the use of abstract component models in the system composition process easily allows to handle of multiple software frameworks in one application. As proof of concept, we developed a code generator for interfacing native ROS nodes and SmartSoft components[4].

2) Gather best practices: Our models are accompanied by their respective DSLs (Domain Specific Languages). This approach allows us to create a textual editor that, among other features, integrates an auto-complete function. We use it to suggest to the user the objects to use, e.g., common message types while defining a publisher. Apart from that the validators of the DSLs contain rules to follow the ROS conventions, for example, the naming conventions defined in the REP 144[5], as it is showed in Fig. 2.

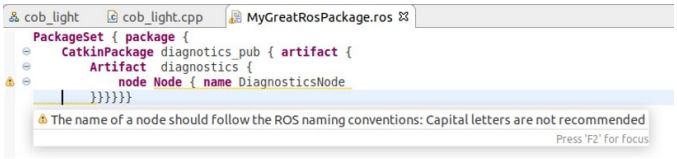

Fig. 2: Warning message shown when the designed components do not follow the naming conventions.

3) Identification of design patterns: We used the static code analysis tool to get the models of all the packages listed on the *ROSDistro*. Then, using a comparison algorithm we were able to identify common design patterns.

For example, in our analysis of the ROS Noetic Distro[6] we found a total of 384 useful models automatically, passing our

[3]https://github.com/ipa-nhg/ros-model-examples/tree/master/RosSystems
[4]https://github.com/seronet-project/SeRoNet-examples/tree/master/SeRoNet-Tooling-ROS-Mixed-Port
[5]https://ros.org/reps/rep-0144.html
[6]https://github.com/ipa320/RosCommonModels

comparison algorithm this resulted in a total of 44 common design patterns.

As a long-term feature, this can be the base of a catalog of components categorized by functionalities, as well as it could be a good impulse for the definition of common specifications.

4) Runtime diagnosis: Our introspection tool called **ROS-Graph monitor**[7] uses the *rosgraph* node to obtain information about the running system, this is compared to the designed system model and publishes an error message in case one of the desired nodes is not present at the execution time. This tool is completely generic and uses the standard diagnostic message type from ROS. Fig. 3 represents an example of the use of these tools. The desired system (as a .rossystem model) is created using the system designed and compared at runtime with the actual running system. In case of unexpected node shutdown, a diagnostic message was published with "Missing node" error. Also we can set specific observers like for the battery component. This module observes the current battery level of the robot and in case it dropped below the prescribed threshold, a diagnostic message will be published with an error "Battery level below threshold".

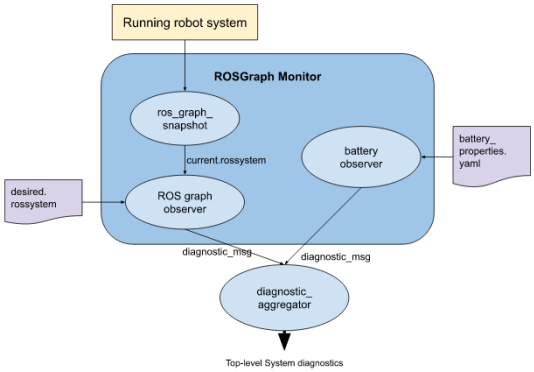

Fig. 3: Overview of the ROSGraph monitor tool.

V. Lessons Learned

A. Models design

In this section, we analyze the lessons we learned by designing our models from ROS existing code.

1) Level of abstraction: In ROS there are no developer roles. In general, the same person is in charge of developing new packages, integrating, testing, and finally deploying them. The consequence of this is that there is no clear distinction between the software related to each phase of the integration process. All code packages are treated in the same way, regardless of whether they are specific (like a concrete arm driver), generic (like the robot manipulation platform *Moveit!*), low-level or high-level, and there is usually no division into subsystems or blocks. This is a major drawback when creating models and deciding on their level of abstraction. Many of the classic concepts of component, module, or system are undefined in ROS, the developer has the freedom to decide the type of modularity of the resulting system. Thus, we decided to define a component as an entity with a name and a set of interfaces (i.e., input and output ports). In this way, a component can be either a single node as a system or a union of nodes, independent of how it is created internally. This type of component can be combined to create new systems. With this approach, we give the integrator all the freedom that she has in ROS to define the granularity of the system. Besides that, we can highlight two disadvantages of this approach:

- The systems models created as compositions of sub-systems are complicated sets of model references. This resulting model is not very readable and practically impossible to create by hand, so we had to create a series of tools to help the user to create the systems in a graphical way.
- In the generator part, where for one system we generate ROS launch files, all the structure is lost, and we can only create a large launch file with a list of nodes that at most if the user has created it correctly, will be divided according to their namespaces.

In this case, it would probably be preferable to prescribe certain rules when defining components and subsystems, adding more rigidity to the definition and distinguishing between systems made up of simple components and systems made up of sub-systems. In this way, the definition of conventions and best practices can help to unify the models, and, in addition, would make the systems much more modular and these modules reusable.

2) Applicability of standards and framework-independent concepts: As previously mentioned Sec. IV-A1, to create the models we were inspired by an OMG standard that defines the concept of a system as a composition of configurable components. Thanks to this mechanism we enable the M2M (model-to-model) transformation with other MDD framework-independent approaches at the component level. To demonstrate the advantages of this concept we developed a model transformer between ROS and the SmartSoft Model Driven Software Development(MDSD) [20] to be able to auto-generate glue-code between ROS native components and OPC-UA or ACE middlewares. This approach is compatible with the RobMoSys [6] concept.

3) Overcoming ROS models limitations: To design the kinematics, as mentioned above in Sec. II-B, there has already been an effort to create an XML-based specification of kinematics that can be used as a meta-model description. One of the reasons XML-based specification of kinematics has not been accepted in the ROS community is that it cannot describe URDF elegantly. In this regard, two re-design aspects can be highlighted:

a) Types and attributes poorly supported: An example clause from URDF that is difficult to describe through an XML schema is *The "limit" element of the "joint" element can only have "lower" and "upper" attributes IF the "type" of the joint is not "continuous"*.[8]. To overcome this problem, we have to modify the models to comprehend four different sub-classes

[7]https://github.com/ipa320/rosgraph_monitor

[8]Paraphrased from http://wiki.ros.org/urdf/XML/joint.

for the joint types: fixed, continuous, revolute, and prismatic since each class has attributes that are specific to its type.

b) Unclear split of concepts: The current URDF specification combines three different aspects of the robot hardware description: kinematics, dynamics, and geometry. We decided to modify the model to split it into three sub-models. The reason for this is to enable different developers (model users) to work on the domain of their expertise. This allows to extension of any of the sub-models for special cases without affecting the overall specification. Over time if it is accepted by the wider community, it can be integrated into the corresponding sub-model. In addition to URDF, a meta-model for composition is defined, which is currently one-to-one mapped to Xacro. This allows for the logging of accurate error messages.

> Many of the classic concepts like components, module,s or systems are undefined in ROS. Thus, we created a component entity whose description is independent of how it works internally, i.e., it can be a single node or a composition of several nodes.

B. Model simulations: Use-cases application

In this section, we analyze the learnings from applying the models in real use cases.

1) The limitations of extraction technologies: The success rate of static code analysis tools varies considerably in ROS. There are too many code styles that cannot all be supported. In previous experiments [18] we were capable of successfully finding 66% of the information with this technique. Conversely, extracting the full model from code at runtime is, on the one hand, dangerous, and on the other hand, it is not possible to reliably access all information, e.g., all information about the file system level is lost. Even for ROS1 we also cannot distinguish at runtime which node set which parameter, and we cannot get the data about the services, i.e., they are only visible if they are called. A redesign of our current tools is necessary.

2) Inconsistency between design-time and runtime information: We found three specific peculiarities that break the coherence of the models at design time with respect to runtime and that are hard to overcome:

a) The name of an interface can be given by using input arguments: We separate the information coming from a node (from its source code (C++ or Python) with the system-level information (what is in a launch file). However, in ROS a very common technique is to pass the name of an interface (e.g., of a topic) as an argument in the launch file or even as a parameter. This means that this information, which for us should be static, is only given when the node is instantiated.

b) The type and name of the interfaces are parsed as parameters: A very clear and well-known example of this particularity is ros_control. In this case, the parameters specify the type of controller that will be started (i.e, a joint trajectory controller, a velocity controller, or a position controller...). This node at runtime checks the parameters and, only then, starts the interfaces particular to the selected controller type.

c) The use of nodelets in ROS1: Nodelets provide a way to compose nodes into a single process, that allows zero-copy passing of data between nodelets in order to reduce the network traffic. During design time, nodelets can be composed in a launch file using a specific language construct. However, nodelets provide the same interfaces as nodes at runtime, therefore, our models treat nodelets as the same as nodes.

3) Lack of conventions: In ROS there are a set of recommendations available in the REPs (ROS Enhancement Proposals) [4], some of them are used as common conventions but they are not mandatory. Furthermore, these REPs do not cover all aspects. When using our models for system integration we encountered two problems related to this issue:

a) Structure of the launch files: There is no description of good practice on how to organize or split a launch file. In ROS1 the options are limited so, apart from deciding where to set the parameters and how to create launch file groups or include, we can handle most cases with our system models and code generators. However, in ROS2, the options get wider because a launch file can be implemented in Python, XML, and YAML. Especially, when a launch file is implemented in Python, apart from different coding styles, there is no clear guide as to when and how nodes and processes should be launched. For example, there are big differences between launch files of MoveIt! configuration packages for the panda[9] and Universal Robots[10] manipulators.

b) URDF/xacro: Certain REPs exist to specify naming conventions and semantic meaning for coordinate frames of mobile platforms, industrial arms, and cameras (for example [7]). These conventions provide a specification to develop better integrable and reusable software components that can be used with a variety of robots. Even though most OEMs follow the conventions, it is not always the case. Also, these conventions are broken in multi-robot scenarios, where is it necessary to add a ”prefix” to distinguish similarly named coordinate frames.

> The lack of conventions or well-defined good practices is a big drawback to implementing an MDD solution as there are no guidelines for the code to be generated. In our case, we had to put effort into analyzing existing code and interview experts to build our solution on a best-practices basis.

VI. Leveraging the traditional benefits of model-based technologies

In the Tbl. I we summarise, based on our experience, to what extent the use of MDD has been beneficial or disadvantageous compared to ROS. The first column lists the

[9]https://github.com/ros-planning/moveit_resources/blob/ros2/panda_moveit_config/launch/demo.launch.py.

[10]https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver/blob/2.2.6/ur_moveit_config/launch/ur_moveit.launch.py.

TABLE I: Table analyzing advantages and disadvantages of using Models compared to ROS in various dimensions

MDD benefit/disadvantage	How?
Reduce the gap between experts and developers	Models are easier than code to be understood for all the different project and process profiles.
Empowers domain experts	There isn't a clear split of domain experts within the ROS developers community. But our models make the only division between low-level code developers and system integrators.
Enforce Architecture	The graphical editors and the system models are based on the architecture representation, this is missed by the ROS traditional approach.
Introduce rigidity	The use of models introduce always rigidity, however so far with our bottom-up approach, we didn't find restrictions on what is allowed in ROS.
Programmers can focus on code logic	The code generators make automatically the tedious and repetitive work.
Modularity of the software architecture	System models help to structure a system and its modules.
Interoperability with other frameworks	M2M transformations automatize the bridges generation between frameworks.
Portability to new technologies	This was not yet demonstrated with our models.
Unification of specifications	The use of models facilitates the identification and use of design patterns.
Documentation equals Implementation	The type of models we provide servers as ROS nodes and systems documentation.
Up-to-date documentation	The models describing the code serve as documentation, every model update comes together with documentation update.
Increase productivity and efficiency	Validation and code generators avoid typo errors and repetitive work. Significantly reduces debug tasks, which in ROS are typically trail-error methods.
Automatization of tasks	Code generators and M2M techniques automatize tasks.
Faster Development Time	For the case of large systems, the time is significantly reduced.
More cost-effective	The tests made so far do not demonstrate it. But we expect that this will be the case for large systems once the user is familiar with the use of our tools.
Less error prone	All the URDF and launch files created manually are highly error-prone. Thanks to code generators we avoid this.
Validation at design time	The DSLs associated with the models include rules that are checked while editing the model.
Meaningful validation	At design time ROS does not allow validation at all, while the DSLs implementations take care of it.
High quality software	The generation of code does not ensure high-quality software.
VCS not efficiently supported	The code generated automatically is not for all the features well handled by VCS systems.
Friendly-development environment	MDD tools enable graphical editors, however, this is not necessarily more friendly for a ROS developer.
Lower the entry barrier	ROS entry barrier is very low, due to the use of general-purpose programming languages. MDD requires learning the use of new tools and domain-specific languages.

typical virtues [1] and shortcomings that are often mentioned when talking about model-based technologies. In green are represented those that we believe have been evident when using our approach, in red those that have not, where the manual solution used in the community is preferable. We have left blank those where we have not been able, through our experiments, to demonstrate. This overview is intended to give only an insight into our findings, clearly, these points need to be studied in detail with practical examples in order to be accurately evaluated and scored.

VII. Conclusion

In this paper, we present what we have learned from our efforts to combine ROS with MDD using a bottom-up, code-to-model approach. Thanks to this experience we have formed an initial picture of where MDD can improve the development of robotic systems with ROS and where ROS is much more powerful. Our next step is to evaluate quantitatively to what extent this improvement is significant, and in which cases and at what level of system complexity it is worthwhile to use MDD techniques. In addition, we will continue to develop our tools to make them more user-friendly to the ROS community and to mitigate further known shortcomings.

References

[1] "15 reasons why you should start using Model Driven Development." [Online]. Available: http://www.theenterprisearchitect.eu/blog/2009/11/25/15-reasons-why-you-should-start-using-model-driven-development/

[2] "COLLADA," https://www.khronos.org/collada/.

[3] "Eu research results: Best practice in robotics (brics)." [Online]. Available: https://cordis.europa.eu/project/id/231940

[4] "Index of ROS Enhancement Proposals (REPs)," http://www.ros.org/reps.

[5] "Papyrus for robotics - A graphical development environment for robotic applications." [Online]. Available: https://www.eclipse.org/papyrus/components/robotics/

[6] "RobMoSys - Composable Models and Software." [Online]. Available: https://robmosys.eu/

[7] "ROS Enhancement Proposal for Coordinate Frames for Mobile Platforms," https://www.ros.org/reps/rep-0105.html.

[8] "ROS: Robot Operating System," http://www.ros.org/, accessed: 2022-11-25.

[9] "SDFormat," http://sdformat.org/.

[10] "URDF: Unified Robot Description Format," http://wiki.ros.org/urdf/XML, accessed: 2022-11-25.

[11] "URDF XML specification," https://github.com/ros/urdfdom/blob/master/xsd/urdf.xsd, accessed: 2022-11-25.

[12] K. Adam, K. Holldobler, B. Rumpe, and A. Wortmann, "Engineering robotics software architectures with exchangeable model transformations," 04 2017, pp. 172–179.

[13] G. Bardaro, A. Semprebon, and M. Matteucci, "A use case in model-based robot development using AADL and ROS," in *Proceedings of the 1st International Workshop on Robotics Software Engineering*, ser. RoSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 9–16. [Online]. Available: https://doi.org/10.1145/3196558.3196560

[14] 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 (SAC)*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480693

[15] A. Bubeck, F. Weisshardt, and A. Verl, "BRIDE - A toolchain for framework-independent development of industrial service robot applications," in *International Symposium on Robotics (ISR/Robotik)*, June 2014, pp. 1–6.

[16] "Deployment and configuration of component-based distributed applications," 2016, version 4.0.

[17] N. Hammoudeh García, H. Deshpande, A. Santos, B. Kahl, and M. Bordignon, "Bootstrapping MDE development from ROS manual code - Part 2: Model generation and leveraging models at runtime," *Software and Systems Modeling*, Apr 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00873-2

[18] N. Hammoudeh Garcia, M. Lüdtke, S. Kortik, B. Kahl, and M. Bordignon, "Bootstrapping MDE development from ROS manual code - Part 1: Metamodeling," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, Feb 2019, pp. 329–336.

[19] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *IEEE International Conference on Robotics and Automation (ICRA) - Workshop on Open Source Software*, 2009.

[20] C. Schlegel and R. Worz, "The software framework SMARTSOFT for implementing sensorimotor systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 1999, pp. 1610–1616 vol.3.

[21] L. J. D. L. Yoonseok Pyo, Hancheol Cho, *ROS Robot Programming (English)*. ROBOTIS, 12 2017.

[22] L. Zhang, R. Merrifield, A. Deguet, and G.-Z. Yang, "Powering the world's robots—10 years of ROS," *Science Robotics*, vol. 2, no. 11, 2017, http://robotics.sciencemag.org/content/2/11/eaar1868.