Augmenting Robot Software Development Process with Flexbot

Paulius Daubaris
Department of Computer Science
University of Helsinki
Helsinki, Finland
paulius.daubaris@helsinki.fi

Juhana Helovuo
Atostek Oy
Tampere, Finland
juhana.helovuo@atostek.com

Niko Mäkitalo
Department of Computer Science
University of Helsinki
Helsinki, Finland
niko.makitalo@helsinki.fi

***Abstract*—Robot Operating System (ROS) and its successor ROS2 have significantly improved the state-of-the-art robot software development process. However, even with all the amenities offered by ROS2 to ease the development, research has shown that practitioners still encounter development issues making software a significant bottleneck. Therefore, in this paper, we discuss ROS-based software development challenges encountered during the development and identified in the existing literature, and introduce the new Flexbot framework that seeks to mitigate some of the identified challenges using model-driven engineering (MDE).**

***Index Terms*—robot operating system, ROS, ROS2, model-driven engineering, MDE**

I. Introduction

The development of robotic systems is a complex process. It often requires a collaborative effort from specialists of contrasting domains (e.g., mechanical engineering) [1]. To alleviate the complexity, various solutions to construct such software have been created throughout the years. One of the most prominent examples is the Robot Operating System (ROS) and its successor ROS2 [8]. ROS and ROS2 established development principles enabling to build the robotic systems collaboratively and rapidly. However, even with the luxury of having such a technology at the robot software developer's disposal, the development remains manual [6], and unsystematic [9].

Considering that robots are expanding to various domains [8] and becoming increasingly adapted to our daily needs, it is important to have the ability to produce software quickly that satisfies the requirements of its environment. So far, ROS and ROS2 have created the terms for rapid software development [5]. However, embracing quality demands continues to be problematic [9, 13]. Moreover, due to ROS2's nature (namely its architecture and design decisions), plain ROS2 development is also prone to various errors as well as incompatible and nonrobust designs. Therefore, in this paper, we introduce Flexbot – a new framework leveraging model-driven engineering that seeks to increase developer productivity and deliver high-quality software by addressing some existing ROS development challenges. The benefits and main reasons for using MDE in robotics have been studied by de Araújo Silva et al. in [2]. The main reasons for adopting MDE in robotics seem to be reducing complexity as well as improving reusability and variability aspects.

The remaining content of the paper is organized as follows. In Section II, we introduce ROS software development challenges that hinder the development process. In Section III, we introduce the Flexbot framework developed by Atostek Oy company that acts as an extension to ROS2 and seeks to assist in developing robot software. Section IV discusses what benefits it brings. We acknowledge related work regarding the existing approaches and future directions in Section V. Finally, we conclude the paper with final remarks in Section VI.

II. Development Challenges

In this section, we introduce some of the challenges ROS-based software developers typically encounter. Note that this is not an exhaustive list but rather based on the development experiences and reinforced by the available literature.

C1: Software robustness. Most ROS-based systems are written in C++ and Python programming languages [12], which are known to be permissive and provide a lot of freedom to the developer. Although beneficial, the enhanced capabilities provided by these languages may leave significant room for error (e.g., memory safety, data races). Such issues are especially relevant to the robotics domain, where one of the major requirements is reliability [9], and the systems are expected to be highly resilient to faults [4]. Unexpected crashes can cause harm to the environment and potentially humans if the system is supposed to interact with them.

C2: Development complexity. The development is further complicated by the wide range of specialists involved in the process [1]. Because of varying expertise and lesser experience in software engineering, unanticipated bugs and logical errors are more likely to be introduced. In ROS, these bugs can go unnoticed until the system is already running [13]. It is because ROS might not consider them defects and conceal the cause of the underlying issues [9]. For example, ROS enables the developer to adjust how nodes communicate using Quality of Service (QoS) policies. Each node can be assigned a policy to adopt a specific behavior based on the system's needs. Nonetheless, in case two or more communicating nodes have distinct policies, there is a risk that messages will not be delivered from the publisher to the subscriber. [3]. Considering

such circumstances, debugging can become burdensome and costly [11, 15].

C3: Technical debt. In addition, fixing the potential bugs later on throughout the life cycle of the project might have a negative influence on the developer productivity, considering they are introduced during the early stages of the development and noticed only later on. This is due to the constant need to reevaluate and reassess the system instead of embracing good practices and the quality of software from the very beginning. Furthermore, considering the distributed ROS-based system nature, the large selection of third-party packages, and their integration challenges [13], the development efficiency can be complicated even further.

C4: Insufficient documentation. Considering less experienced or new developers participating in the system's development, the information providing a general overview of the system is more than beneficial. Nonetheless, recently, Malavolta et al. [10] conducted an observational study where they investigated 335 open-source ROS-based projects and revealed that more than half of the inspected projects did not maintain documentation regarding the architecture. Such lack of information has the potential to become a bottleneck during the development.

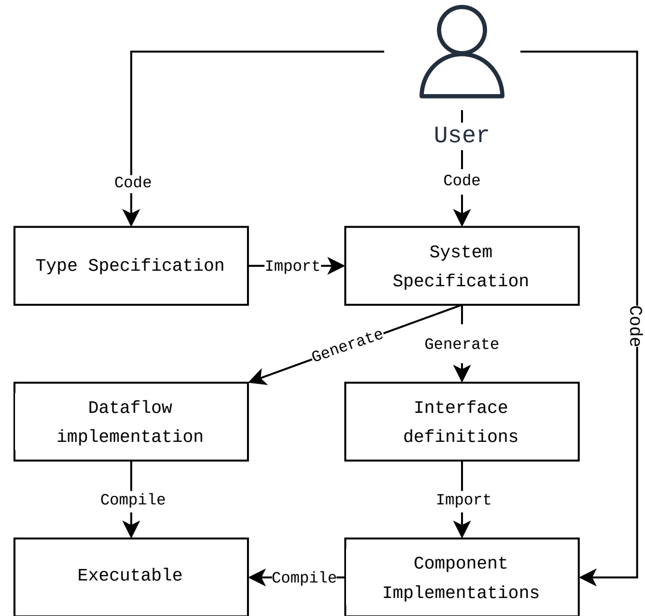

Fig. 1. A graph representing the Flexbot framework.

III. The Flexbot Framework

In this section, we introduce Flexbot – a software framework that can be thought of as an extension to ROS2, which seeks to mitigate robot software development issues identified in Section II. Figure 1 illustrates the high-level overview of the framework. In Flexbot, the user codes the type specification, which contains type declarations defined by the user, system structure specification, and component implementations. Flexbot then uses the system specification to generate data flow implementations and interface definitions. The interface definitions are used by the component implementations, which are compiled alongside the data flow implementations to produce the executable.

Flexbot is compatible with ROS2 and extends the idea of composing the system using nodes from the interprocess scale to a more fine-grained level (e.g., having a larger number of simpler nodes). It uses Haskell as its specification language and Rust as the default implementation language. The system is described by a system structure specification, which in essence, is a data structure written in Haskell and is responsible for listing all the nodes, their connections, and the data flow. It is important to note that the system structure specification does not include the implementations of the listed nodes but rather describes the network structure.

Each node has typed input and output ports that can be connected to typed many-to-many communication channels. These abstract ports and channels can be mapped to various communication mechanisms, such as ROS2 publishers, subscriptions, topics, inter-thread communication channels within a single process, or even plain FIFO data structures when transferring data within a single thread. Considering the fact that Flexbot is implemented in the Rust programming language, it also enables highly efficient and programming language-specific communication channels, which seek to minimize the communication overhead that might otherwise be unavoidable using ROS2 communication channels.

The framework abstracts different communication channel implementations. In case nodes are written in Flexbot, the same API is used by nodes to send and receive messages, regardless of the underlying mechanism. The selection of communication channels for individual nodes is specified at the specification level. As a result, the same node implementations can be reused among different applications.

In addition to nodes and channels, the Flexbot system specification prescribes the mapping of abstract nodes to different processes and threads. The mapping of nodes to threads can be changed by modifying the specification only. The specification is read by the code generator that creates the data flow skeleton code for each process. It is then possible to generate the network into a single event loop or split it into several. The programmer-written code consists of incoming event handlers that are then included in the appropriate processes.

The event handlers receive their input as call arguments and produce output by a framework-defined message send operations (similar to ROS). The framework can then route these messages to other components because it knows if the destinations are within the same process/thread or elsewhere. At the receiving end, the message is handed over to the next event handler as call arguments.

The code generator also employs static analysis to prevent common bugs introduced during the development. The generated code instantiates each node and moves data between the nodes by implementing an asynchronous I/O event loop, which listens for inputs from device drivers and inter-process

communication channels. The inputs are passed to input handler functions, and any produced outputs are forwarded into local node inputs and, or inter-process channels. The specification also allows specifying some nodes and processes as "external" which means that the framework assumes they are programmed outside of the Flexbot system, implying that only communication channels towards them are set up. Such an approach enables integration to, for example, common ROS2 components and tooling.

IV. Flexbot Benefits

Flexbot leverages automation to decrease redundant development and reduce the available room for error, thus mitigating some common robot development challenges.

Developing more robust software (C1, C2). With Flexbot, ROS2 nodes can be created using the Rust programming language. The addition of Rust creates the possibility of preventing common programming errors that occur while using programming languages such as C++ or Python, which are mainly used in ROS2. Rust mitigates memory or data race bugs prevalent in C++ by leveraging its ownership system, as explained in [7]. It is mostly achieved by the compiler, which prevents compilation in case a violation is detected. As a result, robot software developers can develop more reliable software and spend less time debugging undefined behavior.

To reduce the potential shortcomings even further, a system structure specification can be subject to various checks. For example, message type compatibility verification between message senders and receivers. Additional checks may include:

- Statically checking that all inputs and outputs are connected or are intentionally left unconnected.
- Statically specifying node input and output relationships (e.g. each message input produces exactly one output). In some cases knowing such relationships may allow message handler execution to be scheduled statically. Static scheduling improves performance and timing predictability. It may also uncover programming errors by failing to produce a valid schedule in the case of a deadlock or buffer overflow condition.
- Statically checking that there is only one sender in each channel. Technically, it is possible to have several senders per channel, but it is a suspicious structure, and a warning may be issued.

Less technical debt (C3). The foundation of Flexbot's features is the global, typed, and machine-readable system structure specification. It abstracts various parts of the system and is used to generate a majority of boilerplate code. Having the system described in a single source of truth increases the robustness of the system and the productivity of the developer because to change parts, it is no longer needed to go through various parts of the source code, but rather change in one location is sufficient. It makes a system with a large number of nodes, even up to several hundred, manageable. The individual nodes can then be made quite simple, which also means generic, facilitating reuse.

The assignment of nodes to execution threads is done at the specification level. Each node is only ever executed by its assigned thread. The node interacts with others only via channels. Because of these properties, mutexes or other synchronization devices are typically not needed inside the nodes. Such an approach helps in avoiding concurrency errors.

The specification also enables the developer to map execution nodes to specific operating system threads. This mechanism allows one to assign small tasks with low latency requirements to high-priority threads and longer computations to lower-priority ones. Moving nodes from one thread to another requires changes in the specification only.

The Flexbot framework is capable of logging messages between nodes in the same manner as ROS2. These logs can be used as test inputs for testing individual nodes or groups of nodes. Flexbot supports generating such test stubs by configuring the structure specification only.

The framework offers an alternative approach for implementing robot software: while it can improve software development productivity and reduce the technical debt, it is still an effort shift from the usual ROS2 software development.

Understanding the architecture (C4). It is not unusual for ROS packages to be poorly documented [9]. In Flexbot, the system structure specification representing the architecture is translated to a graph for visual inspection. Currently, the Flexbot tools can generate the specification into GraphML [14] file, which is always up-to-date because it is automatically updated at every software build. ROS2, on the other hand, has the RQt visualization tool to visualize the data flow graph, but it can only inspect a running ROS2 system.

V. Related and Future Work

The goal to alleviate development challenges is not new to ROS-based software. Santos et al. [13] described the HAROS framework as capable of statically analyzing ROS-based software and preventing common bugs introduced during development. Although HAROS and Flexbot overlap in the static analysis ideas, both frameworks serve different purposes. Flexbot seeks to aid in scaffolding the system where various static analysis checks apply. HAROS, on the other hand, is concerned only with the analysis.

Considering the issues that arise through manual code development, Garcia et al. [5] proposed using metamodeling for ROS in addition to the manual code development to sustain the possibility of rapid prototyping and leverage model-driven engineering practices to ensure the system robustness [2]. In essence, the approach is similar to Flexbot in that it seeks to improve the quality of the system. However, the proposed approach targets ROS systems, whereas Flexbot focuses solely on ROS2. Nonetheless, the authors indicate ROS2 support as their future goal [6].

As for future research work, we will seek to demonstrate a use case leveraging the Flexbot framework and show how it manages to maintain interoperability with ROS2. As a result, such work would provide insight into both the framework's benefits and shortcomings, and how it accomplishes to mitigate

challenges identified in Section II. In addition, since the communication in Flexbot is based on abstract channels that excel in highly scalable performance, we would like to provide empirical data elaborating on how Flexbot's means of communication compared to those of ROS2. To elaborate, if both ends of a channel are written in Rust, then the framework can generate a native Rust channel implementation between them. If they are different (e.g., one end in Rust and the other in Python), then ROS2 communication can be used instead. The channel implementation can be chosen on a per-channel basis from the structure specification. Tightly coupled nodes within the same process and thread can transfer data by a language-native reference-passing FIFO queue without serialization or synchronization. At the other end of the spectrum, ROS2 topics can be used with all the associated benefits and costs.

VI. Conclusions

In this paper, we discussed what issues hinder the ROS-based software development process based on the development experiences and available literature and introduced the Flexbot framework that seeks to increase robot software developer productivity and mitigate the identified challenges. The presented framework is most certainly not a universal solution to all the issues of the robotics domain. However, it offers a leap toward better software engineering practices alleviating developers from time-consuming tasks and enabling them to reason about the architecture of a system more conveniently. As a part of our future work, we consider constructing a use case and performing an in-depth analysis and evaluation that would describe to what extent Flexbot alleviates the identified challenges.

Acknowledgments

This work was supported by Business Finland.

References

[1] Giuseppina Lucia Casalaro et al. "Model-driven engineering for mobile robotic systems: a systematic mapping study". In: *Software and Systems Modeling* (2021), pp. 1–31.

[2] Edson de Araújo Silva et al. "A survey of Model Driven Engineering in robotics". In: *Journal of Computer Languages* 62 (2021), p. 101021. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2020.101021.

[3] ROS2 Documentation. *About Quality of Service settings*. URL: https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html.

[4] Sergio Garcıa et al. "Robotics Software Engineering: A Perspective from the Service Robotics Domain". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 593–604. DOI: 10.1145/3368089.3409743.

[5] Nadia Hammoudeh Garcia et al. "Bootstrapping MDE Development from ROS Manual Code - Part 1: Metamodeling". In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, pp. 329–336. DOI: 10.1109/IRC.2019.00060.

[6] Nadia Hammoudeh Garcıa et al. "Bootstrapping MDE development from ROS manual code: Part 2—Model generation and leveraging models at runtime". In: *Software and Systems Modeling* 20.6 (2021), pp. 2047–2070.

[7] Ralf Jung et al. "Safe Systems Programming in Rust". In: *Commun. ACM* 64.4 (Mar. 2021), pp. 144–152. ISSN: 0001-0782. DOI: 10.1145/3418295.

[8] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074.

[9] Ivano Malavolta et al. "How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2020, pp. 31–40.

[10] Ivano Malavolta et al. "Mining guidelines for architecting robotics software". In: *Journal of Systems and Software* 178 (2021), p. 110969. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110969.

[11] Samuel Parra, Sven Schneider, and Nico Hochgeschwender. "Specifying QoS Requirements and Capabilities for Component-Based Robot Software". In: *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. 2021, pp. 29–36. DOI: 10.1109/RoSE52553.2021.00012.

[12] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. 3.2. Kobe, Japan. 2009, p. 5.

[13] André Santos, Alcino Cunha, and Nuno Macedo. "Property-Based Testing for the Robot Operating System". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 56–62. ISBN: 9781450360531.

[14] GraphML Team. *The GraphML File Format*. URL: http://graphml.graphdrawing.org/.

[15] Thomas Witte and Matthias Tichy. "Checking Consistency of Robot Software Architectures in ROS". In: *2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE)*. 2018, pp. 1–8.