ROSSi a graphical programming interface for ROS 2

Constantin Wanninger, Sebastian Rossi, Martin Schörner, Alwin Hoffmann, Alexander Poeppel, Christian Eymüller, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Wanninger, Constantin, Sebastian Rossi, Martin Schörner, Alwin Hoffmann, Alexander Poeppel, Christian Eymüller, and Wolfgang Reif. 2021. "ROSSi a graphical programming interface for ROS 2." In *2021 21st International Conference on Control, Automation and Systems (ICCAS), 12-15 October 2021, Jeju, Republic of Korea*, 255–62. Piscataway, NJ: IEEE. https://doi.org/10.23919/ICCAS52745.2021.9649736.

ROSSi
A Graphical Programming Interface for ROS 2

Constantin Wanninger[1],
Sebastian Rossi[2], Martin Schörner[3], Alwin Hoffmann[4], Alexander Poeppel[5], Christian Eymueller[6], Wolfgang Reif[7]
Institute for Software and Systesm Engineering
University of Augsburg
Augsburg, Germany
Email: {wanninger[1], rossi[2], schoerner[3], hoffmann[4], poeppel[5], eymueller[6], reif[7]}@isse.de

***Abstract*—The Robot Operating System (ROS) offers developers a large number of ready-made packages for developing robot programs. The multitude of packages and the different interfaces or adapters is also the reason why ROS projects often tend to become confusing. Concepts of model-driven software development using a domain-specific modeling language could counteract this and at the same time speed up the development process of such projects. This is investigated in this paper by transferring the core concepts from ROS 2 into a graphical programming interface. Elements of established graphical programming tools are compared and approaches from modeling languages such as UML are used to create a novel approach for graphical development of ROS projects. The resulting interface is evaluated through the development of a large project built on ROS, and the approach shows promise towards facilitating work with the Robot Operating System**

I. Introduction

The degree of automation has grown increasingly in recent years, which is clearly illustrated by the statistics of the VDMA [1]. Robotics in particular forms an essential part of this automation, which opens up new possibilities, for example, in production [2]. The field of robotics includes not only the typical industrial robots such as jointed-arm robots in the automotive industry, but also mobile ground robots or flying robots, e.g. for the transport of goods. Due to the variety of types of robots as well as the large number of companies that manufacture them, there are many programming languages as well as tools, such as simulation environments or graphical user interfaces. This is also leading to the emergence of standards and frameworks that support robots from different manufacturers and of different types, since the underlying concepts, such as geometric description of space and positions, do not differ. One example of those frameworks is the RoboticsAPI [3]. This was developed by the *Institute for Software and Systems Engineering* (ISSE) of the University of Augsburg and provides both real-time control as well as a 3D visualization. However, only a handful of robots are supported, and the development has all but ceased due to emergence of more comprehensive frameworks. Today, the prevalent robotics framework in research is the Robot Operating System, or ROS [4]. This open source project supports a variety of different robots and programming languages and additionally provides visualization and simulation tools such as RVIZ [5] and Gazebo [6]. However, due to the required level of abstraction and the comprehensive node architecture, the learning curve for developers new to ROS is comparatively steep. In addition, the ability to debug the node associations of ROS systems is still only rudimentary. This paper presents the extent to which the principle of model-driven development can be applied to the domain of ROS to facilitate the application development process. Model-driven development is a concept that has long been used in other industries, such as the automotive industry, and has led to very powerful programs like Simulink from Mathworks citeSimulinkBook. But there are also great examples of model-driven development in the field of game development that transcend the boundary between game designer and developer, speeding up the entire development process. The result of this work is a program that provides tools for the graphical development of ROS 2 program code.

II. Related Work

For an overview of related topics, concepts for model-driven development from different domains are presented below. Special focus was given to the efficiency of the concepts and their relation to ROS or the possibility of applying them to the domain of ROS.

A. RoboticsLanguage

The *RoboticsLanguage* is a domain-specific language for ROS, which is developed by Robot Care Systems B.V. [7]. The goal is a modeling language that is based on mathematical notation. In addition, it is possible to describe state machines with the help of a so-called *Mini Abstraction Language*. The combination of these two concepts allows a very abstract but expressive definition of ROS nodes. Another interesting aspect are temporal logic statements, which can be defined to evaluate boolean values over time, similar to if statements. This enables queries like, change state if there was no ”true” on a given topic in the last n seconds.

The RoboticsLanguage divides a ROS node into four phases as shown in fig. 1. In the first phase *Definitions* the callback functions and state machines of, for i. a., Publisher, Subscriber

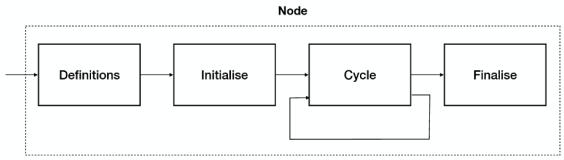

Fig. 1: Four phases of a node

are defined. To send messages via the publishers regularly, the *Cycle* phase is needed. Before and after this phase there are the *Initialise* and the *Finalise* phase, respectively. These are executed once at the start or at the end before the node is destroyed [7].

Even though this is not a graphical tool, the individual phases and also the state machines could easily be represented graphically. However, the project is not yet advanced enough to replace all nodes written in C++ or Python, since program logic beyond state machines or temporal logic statements cannot be created. It also appears that the project will not continue, as the most recent changes were made over a year ago.

B. Unreal Engine: Blueprints

The Unreal Engine is a game engine from Epic Games that is free up to a certain revenue cap. Starting with version 4, game developers can use blueprints to create program logic using only graphical components [8]. Blueprints consist of different colored blocks for events, functions and operations connected with lines representing the program and data flow. In addition, each data type also has its own color in order to enable intuitive visual distinction [8]. Blocks have input pins on the left and output pins on the right, to which lines can be connected. The program flow is initiated from event blocks and proceeds through the function blocks.

Blueprints can be viewed at runtime, allowing developers to monitor object and program flow through animations of the lines between the blocks. For debugging purposes, breakpoints can be defined or the execution can be paused manually in order to examine the current state and inspect the entire graph. Nevertheless, game engines are generally not suitable for evaluating sensors or controlling actuators in the real world.

C. Matlab Simulink

Simulink is an extension of the commercial software Matlab by MathWorks [9]. Similar to the aforementioned blueprints, this graphical programming environment also uses blocks, which can be combined into a complex system by interconnecting them. With the integrated code generators, these blocks can be converted into C program code, for example. Simulink is mainly used in research and industry to easily create simulations of mathematical or physical systems such as circuits or signal processing [9]. There are numerous libraries that contain, for example, mathematical and physical blocks. Also there are built-in blocks to directly support the creation of ROS and ROS 2 nodes.

Simulink simulates step-by-step, there is no data or program flow comparable to blueprints. Instead, there are sources, such as constants, waves, or the (simulated) time, from which the simulation runs through all intermediate operations to endpoints (sinks).

Although in Simulink ROS nodes can be created and executed directly, in the open source project ROS a tool that is also publicly available and that can be integrated directly into ROS would be beneficial. In addition, only individual nodes can be started in Simulink and the aspect of ROS launch files is completely missing.

III. Objectives

The goal of this research is to apply concepts of model-driven software development to ROS 2 in order to lower the barrier to entry and facilitate the development of systems based on it. The goal of this research is to provide a methodology for facilitating the development of systems based on ROS 2. Typically, such applications are developed in a bottom-up fashion, which can quickly lead to confusion about the overall architecture and hamper further development. In order to alleviate this, we propose inverting the design process of robotics applications in ROS 2 by utilizing model-driven development methods. To this end, the core principles of such approaches from other disciplines examined in chapter II are transferred into a graphical development tool for the domain of ROS 2. To demonstrate the efficacy of the presented approach, two projects previously developed for ROS 1 are converted to ROS 2 with its support (cf. chapter V) and evaluated in chapter VII.

IV. Concepts

To cover the full spectrum of the ROS architecture, two separate modeling languages were designed:

One for the design of nodes, and the other for the generation of launchfiles. The following section gives a brief overview over the core concepts of these languages.

A. Domain-specific modeling language for ROS nodes

Initially, all essential components of a ROS node must be mapped to corresponding components of the newly defined modeling language. Since completely dispensing with written program code exceeded the scope of this work but is nevertheless desirable, this is treated as a future extension but still considered during the conception of this model. Hence placeholders are defined in place of future graphical implementations of those parts of a node where complex program logic would be defined.

Thus the current result of the compiled modeling language consists only of the skeleton code of a ROS Node, which must be completed by the developer with a concrete implementation of logic.

In the following, the structure of components in the modeling language is presented before the individual phases of a node's lifecycle are subsequently explained.

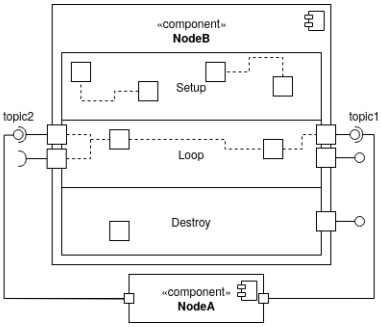

Fig. 2: Inner structure of a node

1) Modeling a ROS node: Every ROS node is modeled as a component in the proposed modeling language. This base component internally models lifecycle of a node by dividing it into three different phases and providing predefined areas for the insertion of the appropriate program code for processing in each of these phases. The graphical representation of this structure within a node *NodeB* is shown in fig. 2. It is largely inspired by the modeling of a component in UML, however the UML specification had to be deviated from in some areas, for example, to be able to represent the phases of a node within a component. Additionally, in order to enable future extensions of the modeling language, the component specification is easily extendable with custom classes modeling the desired behavior and lifecycle specification.

The square *function blocks* seen in fig. 2 define arbitrary functionality in different parts of a ROS node. Function blocks within a node component define an internal function, which can be helpful in structuring and reusing implementation fragments within a node. External function blocks define communication interfaces with a node. In ROS communication with a node can be realized in one of three ways:

- **Topics**: Asynchronous communication following the publish / subscribe principle
- **Services**: Synchronous communication following the server / client principle
- **Actions**: Used for long running operations. Asynchronous invocation and status updates.

Each of these communication interfaces can be modeled with function blocks, which define a node's actions and reactions to specific events and messages. As such, a publisher is defined as a function block, which responds to the update of a variable by publishing it to the predefined topic. A subscriber on the other hand responds to the reception of a message on the desired topic by invoking the node's corresponding callback function.

Fig. 2 also shows the connection of two nodes via topics as well as an abstract representation of the internal interconnection of function blocks in order to realize the required functionality.

The implementation of the actual logic of a function block can again be performed using the modeling language, which effectively produces nested components within the overall model. As of now, the implementation of function blocks must be performed manually, since, as mentioned above, the complete elimination of manually written code was outside of the scope of this work. However, the developer is aided by the generation of skeleton code from the model as further described in chapter VI.

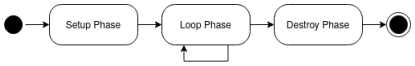

Fig. 3: Activity diagram of the phases of a node

2) Lifecycle of a node: The lifecycle of any program can loosely be divided into three steps: setup, looped execution and destruction. By modeling the lifecycle of a node with these distinct phases, the developer of a node need not care about the manual definition of these phases and can concentrate solely on the implementation of the actual logic that needs to be executed within each of them. Later, when translating the model into skeleton code, this can aid the developer by generating method definitions for each of the three phases within the node definition (cf. VI). An activity diagram of the lifecycle of a ROS node and the three different phases is shown in fig. 3.

a) Setup phase: Compared to the RoboticsLanguage this initial phase combines the *Definitions* and the *Initialise* part [**?**], as both are run only once and thus the distinction is unnecessary in this context. This phase is intended for the proper initialization of the node, its data members and all required communication channels. Additionally, the node must be registered in the overall system. After setup is completed, the node is transitioned to the loop phase.

b) Loop phase: In this phase, the actual functionality of the node is executed in every iteration of a global main loop. The loop phases of all existing components are executed consecutively by the system within each time step.

c) Destroy phase: This phase is responsible for correctly destroying the node and all of its data members. After completing this phase, the node is terminated and no longer present in the system.

In summation, the model definitions of nodes, their lifecycles and their external communication interfaces allow the automatic generation of skeleton code for the implementation of fully fledged ROS nodes, as well as the automatic execution and lifecycle management of thusly defined components.

B. Domain-specific modeling language for ROS launchfiles

Launchfiles are a useful part of ROS, as they can be used to simultaneously start multiple nodes from one central point. The modeling language defined for the generation of launchfiles does not add any additional concepts or functionality, but is required in order enable automation of the entire deployment and execution process of the generated ROS nodes described in the previous section.

The basic concepts of launchfiles, and thus the defined modeling language are as follows:

a) Node / Executable component: A node in the roslaunch model is equivalent to those shown in 2. Within a launchfile, however, nodes may be parametrized according to a specific application. The only required information when including a node in a launchfile are its name and the enclosing package.

b) Launchfile component: Launchfiles can also be included in other launchfiles and passed a set of parameters, which those in turn pass down enclosed launchfiles and nodes. As with nodes, the only required information is the name and containing package.

c) Namespace component: For better organization of a ROS application, launchfiles also support the concept of namespaces, which consequently must also be represented in the modeling language.

Together with the model for nodes and the automation of their execution, the abstract modeling language for launchfiles allows the automatic parametrization and execution of nodes and, consequently, automatic execution of entire ROS applications.

V. Case Study

At the Institute for Software & Systems Engineering, there is an experimental setup that takes place in an indoor flying hall. Quadrocopters (flying robots with four rotors) can be moved through space only using sensor-based manipulation via an external tracking system. Virtual walls keep the flying robots from causing damage to people and objects in the room. A lightweight collision avoidance strategy prevents quadrocopters from colliding within the flight area. Teaching paths for a quadrocopter is also possible using gesture control only. This allows repetitive tasks to be set up intuitively and flown repeatedly by the robots. If there is a risk of collision, the aforementioned collision avoidance strategy ensures that, if necessary, the taught path is left and then flown to again. By stabilizing the quadrocopter, the built-in flight controller makes it possible to move objects such as a water glass, which is placed unattached on the robot, through space without spilling any liquid. This experiment shall serve as a case study for the graphical development environment for ROS 2 programs developed in this thesis. It is currently only available as ROS 1 project and shall be rewritten to an executable ROS 2 project using the development environment. Nevertheless, a simplified version, which was developed for this work as ROS 1 project, will be used to explain the underlying concepts of the indoor flying hall and the external tracking system and will be converted from ROS 1 to ROS 2 as well. The sequence of the example experiment consists of a quadrocopter that is first activated (arm) and then automatically ascends to a point exactly one meter above its starting position. There the quadrocopter remains for a short time and then land again and deactivate itself (disarm). After that, the actual experiment will be used to cover the use case of a larger project as well.

VI. Implementation/Realization

In view of already existing plugins of the *RQT* package for ROS and ROS 2, especially the *rqt_graph*, the program developed for this work should also be able to be used as RQT plugin. For this the name ROS - Simple was chosen, in short ROSSi. It is implemented exclusively with Python 3, the second programming language directly supported by ROS 2 besides C++. In total ROSSi supports three different diagram editors, the *Node Editor*, the *Launchfile Editor* and the *Live Diagram*, which can be managed in tabs. There is also a library window which, like Simulink's, is detached from the diagram editor windows. All diagram editors derive from the same parent class, which in turn builds directly on *QT* elements. It brings some methods that must be overridden and their information is used to create menu items appropriate to the particular diagram. ROSSI is then able to automatically find all subclasses and display them as an option the user can choose from. This makes the main ROSSi window easily extendable by new editors and free of menu items irrelevant to the selected diagram editor.

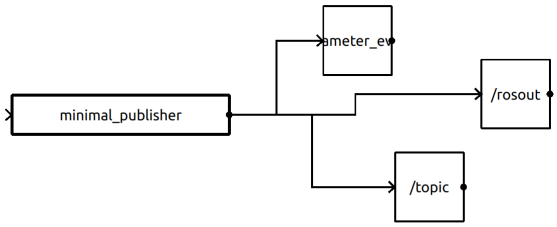

Fig. 4: Live diagram of the system after starting an example program provided by ROS 2

A. Live Diagram

The live diagram provides an overview of all running nodes and their served topics. This functionality is also provided by the *RQT* plugin *rqt_graph*. However, the live graph additionally reacts to changes over time in the system and removes components or adds new ones. Three different types of boxes are used to distinguish between nodes, namespaces and served topics. In order to prevent that newly added boxes cover others, all components distribute themselves with the help of a so-called *Spring Embedders* similar to that of Fruchterman and Reingold [10] independently in the space.

In the topic boxes you can see all corresponding messages that are published at runtime. To distinguish whether a node publishes to a topic or subscribes to it there are left and right ports. Topic components, such as those shown in figure 4, can also be intuitively identified as publishers or subscribers by the placement of the ports.

B. Node Editor

The node editor is the realization of the domain-specific modeling language for ROS 2 nodes presented in the chapter IV. A small example can be seen in figure 5. Due to the component-based architecture of the diagram editors presented here, additional extensions can easily be added to the node editor later on.

The generation of program code strictly follows the three phases defined in chapter IV. All of the components developed for the node editor implement an interface function that has program code relevant to themselves in each phase as a return value. Thus, by simply querying each component placed in the diagram, the node editor is able to generate the entire program code of the node developed with it.

1) Base component: The base component is representative of the node to be created and provides the executable basis of a node, which all other components build upon. Two of these basic components and their interaction can be seen in the figure 2. The component is implemented by the *RosBaseNodeGraphEntity* class and contains, in addition to the name configurable in the options menu, with which the node registers itself in the ROS system by default at startup, also the template for the structure of the program code of the class generated from the modeled node. A base component is automatically created with the creation of a new node editor diagram and cannot be removed from it.

2) Function component: This component serves as a placeholder for the initially described extension possibility of the model by a program logic generating component. Depicted are the functions as squares within the different phases of the node in figure 2. The dashed lines between the functions are meant to indicate mutual calls. It is added to the diagram by the *PythonFunctionGraphEntity* class and can currently only be used in conjunction with other components. By the name given to it by connected components, the function header can be created during program code generation. After inserting the extension, it should be possible with the function component to display any program logic within the node editor. This means that functional components can be chained and supplied with required parameters by other components.

3) Publisher component: This component allows developers to create a publisher in his node via drag-and-drop. In the options menu of the component they can choose from all known message definitions. Additionally, a clock rate at which messages should be published and the name of the topic through which the messages should be sent can be set. Internally, this component holds a reference to a function component that is also inserted into the node editor with each publisher. There, the sending behavior of the publisher can be modeled directly.

4) Subscriber component: Similar to the publisher component, subscribers require the message definition and topic name to be selected. However, an obvious difference is the resulting program code of this component, which defines a subscriber in the setup phase of the node and adds a callback function to the generated class. Together with the publisher component just mentioned, interfaces between nodes can be realized as in figure 2.

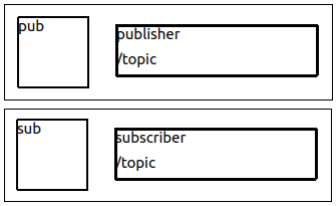

Fig. 5: Node editors of a simple publisher / subscriber pair

5) Parameter component: ROS nodes can be configured by values from the so-called *parameter server* of the ROS system from outside. The parameter component allows the user to use this aspect of nodes within the node editor. Only a name and a default value must be specified.

6) Other components: By inheriting the *CodeHolder* class, it is easy to integrate your own program code into the individual phases of a node. By inheriting the *StandardGraphEntity* only a few steps to a visual representation in the diagram are necessary.

7) Generation of program code: Dividing a node into phases helps to model the individual components. Each of them can represent its own implementation details in the desired phases. That is, depending upon need certain behavior of a component can be implemented in one of the three following phases. From the four phases presented in the RoboticsLanguage, three are extracted for the model developed in this work [**?**]. These can be seen in the figure 3 as well as indicated in the component diagram of the figure 2. The setup phase combines the *Definitions* and the *Initialise* part of the RoboticsLanguage, as both are run only once each [**?**]. At this point, the node is parameterized and registered in the system and necessary initialization processes of all components added to the model are performed. This includes, for example, registering a publisher in the ROS system.After completion of this phase, the system switches directly to the loop phase. There the actual functionality of the node is processed in the loop phase at a clock rate specified in the base component. The loop is executed from the start time of the node until the node or the whole ROS system is terminated. After completing the destroy phase, the node is terminated and no longer present in the system. The program code defined in the function components can be added as methods within the generated class. All references from other components to a function component can now be realized by calling the method with its name.

C. Launchfile Editor

An elementary part in the development of a ROS project is the development of launch files, as they can be used to simultaneously start multiple nodes in a grouped and parameterized manner. In addition, they can be nested, i.e. one launchfile can launch other launchfiles. This can become unclear and often requires a significant amount of time. With the help of the launchfile editor, which is integrated in ROSSi by default, this process shall be accelerated and at the same time facilitated. Since for ROS 2 projects launchfiles are implemented as Python 3 scripts, they offer significantly more functionality than in ROS 1. With respect to the limited time frame of this work, only the functionality known from ROS 1 is therefore mapped within the launchfile editor. Specifically, this means

that the user can use namespaces, arguments, executables, and external launchfiles in the diagram editor to create custom launchfiles.

Figure 6 shows an example of a launch file configuration from ROSSi. Three arguments are created and an executable is launched within a namespace. One argument is the namespace name in which the executable *publisher_old_school* from the package *examples_rclpy_minimal_publisher* should be launched. The other two are in sequence and their compound value is passed to the executable as parameters named *arg1*. The four basic components of a launchfile diagram are now briefly explained.

1) Parameter component: In ROS there are launch arguments - consisting of a name and a default value. These can be usually overridden by command line parameters when the launch file is started. A parameter component consists of three parts whose combination results in the value stored by the component. These parts are the prefix, the suffix and the actual value of the parameter to which a default value is assigned. Prefix and suffix can be activated or deactivated. If there is a connection to another parameter, the own default value is ignored and replaced by the value of the other parameter. In figure 6 you can see how a compound value is created using two parameters, which can then be used, for example, as a value for a parameter of an executable or as the name of a namespace. By concatenating multiple parameter components it is possible to compose complex values which can be overwritten in whole or in part when the launchfile is started. This is a convenient functionality, since for example namespaces can be named correctly just by combining some arguments.

2) Node component: A complete list of all executables of all packages of the current ROS environment can be found in the library window. They are graphically represented by node components. These have an arbitrary extensible list of parameters, which each require a name and a value.A parameter is represented by instances of the class *ParameterGraphEntity* and its value must be obtained from a *RosArgumentGraphEntity*.

3) Launchfile component: Just as the executables of a package are listed in the library window, the launchfiles contained in each package are also provided as entries. They, too, can have a list of paramaters attached to them that behave exactly like those of a node component, although they must be dealt with slightly differently during program code generation.

4) Namespace component: The special feature of this component is that it can be scaled in size as desired. If this component completely encloses another one, the latter is placed inside the namespace in the program code generation. This provides an intuitive way to assign a namespace to executables and launchfiles.

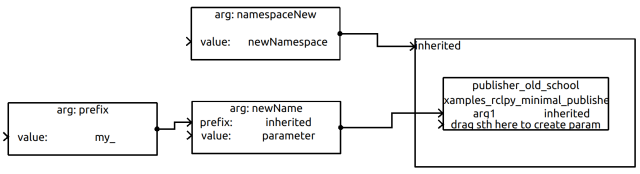

Fig. 6: Example of a launch diagram

5) Additional information: In chapter IV the idea of finding and displaying publisher / subscriber pairs between all included nodes during the creation of a diagram within the launchfile editor was already mentioned. Since simple analysis of the program code is not sufficient for finding such pairs, additionally a black box test was used to try to get more information about the behavior of a node. For each new node component placed in the diagram, a subprocess was started, which in turn starts the corresponding node in a separate namespace. Subsequently, all topics used by the node were examined with the help of the interfaces offered by ROS. Hence, even after a certain amount of time, the complete registration of all topics used in a node is not necessarily complete if it was started alone in a encapsulated system. One way to get around the error-prone automated topic discovery would be to include a self-description of each node within its own package. This could be provided as a simple *JSON* [11] or *YAML* [12] file by developers.

6) Generation of program code: In contrast to generating program code from node editor diagrams, this procedure is comparatively easy for launchfiles. The same procedure can be used recursively for each namespace. All components of the diagram within a namespace can be appended to the resulting *LaunchDescription* in any order. The only exception to this are chained launch arguments. These must first be sorted by order of appearance in the diagram. Since they are not affected by the namespaces, but are available throughout an entire launchfile, they are initially appended to the *LaunchDescription*, now put in correct order. After that the generated program code of all other components is appended to the resulting launchfile in the way described above.

The plugin for *RQT* for the graphical modeling of ROS 2 projects called ROSSi developed in this thesis currently consists of the three different editors, but its architecture allows the connection of new ones. While the Live Diagram supports the development process by providing a visual representation of the current system, the node editor can be used to lay the foundation for the nodes to be developed. Once these have been compiled into executable program code in a package and added to the environment of the terminal that runs the *RQT* plugin, the launch file editor can be used to intuitively create all the necessary launch files and then export them in a fully functional form.

VII. Proof of concept

In this chapter, ROSSi is used to translate the two technical demonstrators of the ISSE chair presented in chapter V from ROS 1 to ROS 2 and thus prove the functionality of the plugin. For this purpose, the underlying technical details of both projects will be discussed first, followed by the steps necessary to translate the projects to ROS 2 using ROSSi.

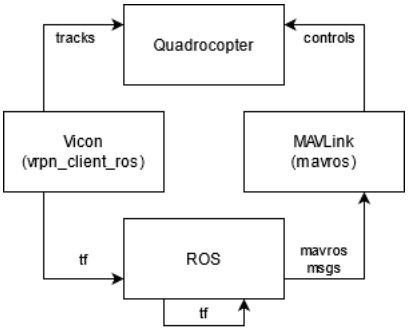

Fig. 7: Main components of the minimal example

A. Test setup

Both technical demonstrators take place in a flight area, with a Vicon [13] infrared camera system mounted around its borders. With its help, movements of objects within the area visible to the camera system can be tracked. The Vicon system has an average measurement error of 0.15 millimeters for standing and less than 2 millimeters for moving objects [14]. In the case of our experimental setup, this is used to measure the positions and orientations of quadrocopters located in the flight area. To make them visible to the camera system, reflective markers must be attached to them. If the markers are distributed asymmetrically and in an unique way, the orientation of each individual quadrocopter can be determined. The quadrocopters used are equipped with a flight controller from Autoquad [15], which can be controlled with *MAVLink* [16] commands.

Another component of the experimental setup is a wand equipped with markers. With its help, it is possible to select or deselect individual quadrocopters or even navigate through space using gestures such as pointing to a target position.

1) Minimal example for quadrocopter control: The minimal example was created for both ROS Melodic and ROS 2 Eloquent and will subsequently be developed using only the tools provided by ROSSi to demonstrate any differences that may arise.

Figure 7 shows the technical structure of the experiment. In addition to the program code of the experiment itself, components, or executables, of the packages shown in the figure are also running on the ROS side, which communicate with the Vicon camera system and the flight controller on the quadrocopter. The Vicon camera system periodically publishes the position of the quadrocopter equipped with reflectors into designated topics within the *TF graph*. These messages are now used to determine the next target position to approach and, if available, to send it to the flight control of the quadrocopter.

Figure 8 illustrates the concept of the minimal example for quadrocopter control as a launchfile diagram in the launchfile editor of ROSSi. Due to space constraints, the parameters and the rest of this launchfile diagram had to be separated. The corresponding parameters can be seen in figure 9. The component, in this case another launchfile, that connects the Vicon camera system to ROS is already present and can be included directly in the diagram. This is in contrast to *mavros*, a package that is currently not directly available in ROS 2. It must be attached to the project using the *ros1_bridge* executable and a parallel running ROS 1 system when using this launchfile. On the ROS 1 system all missing components (here only the interface to *MAVLink* with the name *mavros*) have to be started. The *ros1_bridge* will then find and connect all publisher / subscriber pairs regardless of the ROS version.

Finally, for the minimal example for quadrocopter control, there is a namespace within which the missing components from figure 7 are placed. The namespace name is set using the *mav_id* parameter and the *namespaceName* for the quadrocopter with that same id. This includes, on the one hand, the conversion between the coordinates given by the Vicon system to the ROS system and, on the other hand, those sent to the Autoquad flight control software via *mavros*. This is done in a dedicated node called *autoquad_vicon_bridge*, which is supplied with the necessary information by the four parameters seen in it. The node named *autoquad_pos_vel_target*, which also sends messages to the flight control of the quadrocopter whenever the target frame intended for the flying robot changes within the *TF-graph*. This target frame is given to all relevant nodes via the parameter *copter_target*. Changing the target frame is the task of the last node in the diagram and is only changed twice in this example. Once to give the quadrocopter the signal to fly to one meter above the ground and the second time to get back to the ground.

2) Large technical demonstrator: In the flight area there are two quadrocopters, which are brought into two different positions via the wand with gesture control. During this process, the flown path of the first quadrocopter can be recorded via a gesture. Meanwhile, the collision avoidance of the quadrocopters can now be demonstrated by steering the second flying robot with the wand into the path of the recorded trajectory.

In addition to the components already known from the minimal example, this experiment also has nodes for gesture control of the wand and collision avoidance. All these com-

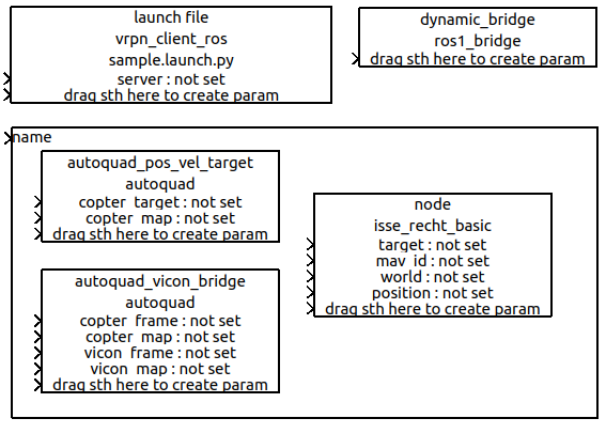

Fig. 8: Launchfile diagram of the small demonstrator (Part 2)

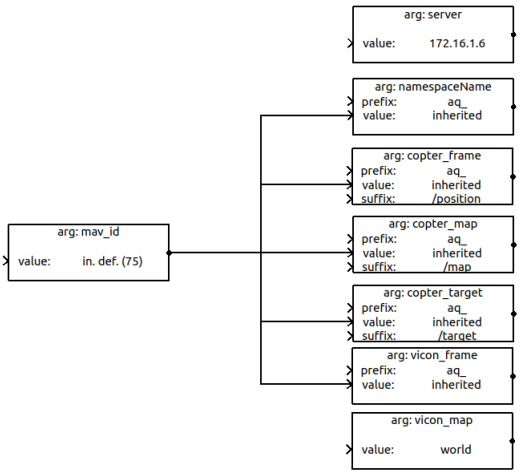

Fig. 9: Launchfile diagram of the small demonstrator (Part 1)

ponents obtain the necessary information from the *TF-graph* and in the end jointly manipulate the next target position of the respective quadrocopter.

Although the structure of the large technical demonstrator is the same, other nodes are involved in finding the next target frame of a quadrocopter. One of these is the control wand, whose position is used as the target point for one of the two quadrocopters, depending on the state set in advance by a gesture. The position of the wand is artificially increased so that the flying robot comes to a stop at some distance in front of it. Also, the collision avoidance strategy implemented in the demonstrator operates as a high-priority instance on the target frames of the two robots, adjusting them as necessary. These additional components are also placed within the particular namespace for one quadrocopter in the large demonstrator and are consequently executed simultaneously for both without interfering with the other. Only the collision avoidance strategy obviously needs to know both quadrocopters in order to compare and adjust their target frames.

Through ROSSi, complex demonstrators could be easily modeled in ROS and components from other demonstrators could be reused. Also the conversion from ROS to ROS 2 could be facilitated.

VIII. Conclusion

ROSSi includes different modeling tools that can accelerate new developments or the maintenance of existing ROS applications. Repetitive implementation tasks can be reduced by using reusable components and their clear and editable relationships. The launchfile editor prepares the otherwise easily confusing files in a visually appealing way and offers a quick insight into the structures of a project. The live diagram also helps to detect errors during runtime by visualizing the real time message flow within the node components. In the future, the live diagram can be further developed in the sense of the UML deployment diagram in order to better reflect the often highly distributed systems of ROS applications.

Due to the extensibility of ROSSi, new specialized components can be added to the already existing diagram editors in the future. But also the integration of completely new diagram editors is intentionally supported by the architecture. These include, above all, the modeling of program logic within the node editor and the possibility of including self-descriptions in ROS packages to provide developers with tools such as the launchfile editor deeper insights into the behavior of a node within a specific context. Simultaneous development of launchfiles and nodes that have not yet been converted to program code is another useful extension of ROSSi. Specifically, it would be conceivable to create a launch file based on a previously made conceptualization of a project and to create missing nodes in the launch file diagram using the node editor. This would even allow ROSSi to export a complete ROS package at once. With ROSSi, a foundation has been laid for testing the still little-used concept of model-driven development on state-of-the-art frameworks such as the Robot Operating System, and it has been shown that promising tools can be developed with it.

References

[1] Aktuelle branchendaten - robotik und automation auf rekordniveau. VDMA. (last visited 05.07.21). [Online]. Available: https://www.vdma.org/v2viewer/-/v2article/render/15372372

[2] Produktionswert von mehrzweck-industrierobotern in deutschland in den jahren 2009 bis 2019. (last visited 05.07.21). [Online]. Available: https://de.statista.com/statistik/daten/studie/445206/umfrage/produktionswert-von-mehrzweck-industrierobotern-in-deutschland/

[3] A. Angerer, A. Hoffmann, A. Schierl *et al.*, "The robotics api: An object-oriented framework for modeling industrial robotics applications," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010, pp. 4036–4041.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[5] Ros rviz. Open Source Robotics Foundation, Inc. (last visited 05.07.21). [Online]. Available: http://wiki.ros.org/rviz

[6] Ros gazebo. Open Source Robotics Foundation, Inc. (last visited 05.07.21). [Online]. Available: http://gazebosim.org/

[7] Roboticslanguage. Robot Care Systems. (last visited 05.07.21). [Online]. Available: https://github.com/robotcaresystems/RoboticsLanguage

[8] B. Sewell, *Blueprints Visual Scripting for Unreal Engine*. Packt Publishing Ltd, 2015.

[9] D. K. Chaturvedi, *Modeling and simulation of systems using MATLAB and Simulink*. CRC press, 2017.

[10] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Softw.: Pract. Exper.*, vol. 21, no. 11, pp. 1129–1164, Nov 1991.

[11] F. Pezoa, J. L. Reutter, F. Suarez *et al.*, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 263–273.

[12] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml) version 1.1," *Working Draft 2008-05*, vol. 11, 2009.

[13] Vicon. Vicon Motion Systems Ltd UK. (last visited 05.07.21). [Online]. Available: https://www.vicon.com/about-us/what-is-motion-capture/

[14] P. Merriaux, Y. Dupuis, R. Boutteau *et al.*, "A study of vicon system positioning performance," *Sensors*, vol. 17, no. 7, p. 1591, 2017. [Online]. Available: https://www.mdpi.com/1424-8220/17/7/1591

[15] Autoquad website. Autoquad. (last visited 05.07.21). [Online]. Available: http://autoquad.org/?lang=de

[16] Mavlink website. dronecode. (last visited 05.07.21). [Online]. Available: https://mavlink.io/en/