Enhancing Development of Modular Application-Specific Configurable Space Robots

H. Wiedemann[1], M. Schilling[2], P. Chowdhury[1], W. Brinkmann[2], I. Kien[1], J. Li[1], M. Langosz[2] and E. Michelson[2]

[1]University of Bremen, Robotics Research Group, Robert-Hooke-Str. 1, D-28359 Bremen
[2]German Research Center for Artificial Intelligence, Robotic Innovation Center, Robert-Hooke-Str. 1, D-28359 Bremen

E-mail: moritz.schilling@dfki.de

Abstract. In many space activities, existing robotic systems are highly mission-specific and cannot be reused. On the other hand, there are several highly modular system designs, that lack the specialized hardware, yet. The MODKOM (Modular Components as Building Blocks for Application-specific Configurable Space Robots) project, aims to create a toolbox that allows to configure and recombine a robot for certain tasks, out of specialized and standardized building blocks; this also includes commercial off-the-shelf components. Therefore, MODKOM also focuses on providing a software framework to compose and configure such systems. Based on the proposed system modeling, all entities (hard- & software) can be represented, handled, and also be stored to reuse sub-systems. By providing adequate (graphical) user interfaces and thereby lowering the need for manually-typed files, this process is simplified benefiting accessibility to non-experts. First tests in parallel projects show already the reduced necessity for manual configuration work and thus, a decrease in mistakes in formerly error-prone tasks. MODKOM also provides a set of modular hardware building blocks, which will be used in the upcoming final demonstration scenario to evaluate the advances made.

1. Introduction

Due to the challenging environment, robotic systems are crucial for space applications. In the past, space missions had few different mission goals, therefore highly specialized robots were used for the tasks. As complexity increases in planned space missions, modularity becomes more important.

Different levels of modularity have already been implemented in various spacecraft like the International Space Station, the Multimission Modular Spacecraft, or the Reconfigurable Operational Spacecraft for Science and Exploration [1]. By using standardized and easily replaceable units interconnected by multifunctional interfaces, serviceability and reusability are enabled [2]. Also for planetary rovers there are some concepts, like the modular robotic system presented by Hancher et al. (2006) [3], where many similar elements form one functional unit. Also other works as reviewed by Brunete et al. (2017) [4] show a likewise granularity and have been developed from scratch to be that modular, e.g. the SUPERBOT [5]. For such systems that rely on few building blocks Tosun et al. (2017) proposed already a computer-aided compositional design tool [6]. However, if even static shapes have to be formed from many actuated parts, complexity for control increases tremendously. Also integration of commercial

off-the-shelf (COTS) components becomes more difficult. As there is already a vast amount of robot parts from many different manufacturers, integrating those components in a toolbox instead of starting from scratch saves development work.

Therefore, in the proposed project the central aspect of the developed modular toolkit is to find a level of granularity of the modules that allows making use of existing COTS parts (see [7]). This encompasses not only the character and topology of the modules, but also the connectivity between them. By providing multi-functional interfaces (i.e. electrical, mechanical, data) and, if necessary adapters, this connectivity is also provided for COTS components.

Once added to this toolkit, systems can be composed out of atomic components as well as already composed subsystems. Specifically, this means that modular subsystems are scalable to a certain degree and according to the mission requirements.

To tackle the complexity issue of creating such systems, this paper presents methods and tools to support the user as much as possible to facilitate creation and operation of modular systems. Frameworks like Robot Operating System (ROS) [8] and Robot Construction Kit (ROCK) [9], which manage the interoperability of different software components, serve as foundation. The ROCK framework is highly designed based on such a modeling approach. It already supports the definition of a dataflow by a graph based representation of its software components. Moreover, ROCK uses Autoproj as package manager which allows various version control systems and sources for dependencies, what makes it easy to include third party software. Therefore, in the first implementation of the building block system the ROCK framework is used as backend to deploy and execute the final runtime software.

This paper presents the current state, developed methods and the first results of the project on enhancing the development of modular robots.

2. Software Framework

As the framework shall be able to handle components of all domains (mechanical, electrical, software etc.), appropriate system modeling is required as backbone of this approach. Based on these models all operations the user may perform in the frontend, e.g. the graphical user interfaces (GUI), are executed. By rendering the information stored in the system model to the necessary files, the created robot design can finally be deployed.

2.1. System Modeling

The modeling layer is the cornerstone of the entire software framework. It consists of the XTYPES-GENERATOR, the XTYPES, and the XDBI libraries and tools. The XTYPES-GENERATOR defines the *XType* base class (an entity storing properties and relations to other *XType*s) and the template specification out of which derived, project-specific *XType* implementations can be generated. As a result, a system of interdependent *XType*s forms a labeled graph.

The XTYPES library uses the *XType*s base class to implement the following classes which are necessary for modeling enitities in robotic systems (see Fig. 1): In general terms, this can be divided into parts and their connections. *InterfaceModel*s describe the general type of a connection from one part two another, while its instance, the *Interface*, specifies the connection possibilities of a part. Similarly, a *ComponentModel* defines the general type of a part and a *Component* is its specialized and configured instance in a network of parts with connections. Those networks form then again a new *ComponentModel* whose instances can be reused in other networks. Both, *ComponentModel*s and *Component*s can have *Interface*s. Also they might have DynamicInterfaces, which are basically *Interface*s, but their name and existance depend on the configuration of that *Component*. As soon as that, what a *ComponentModel* describes, is realized a *Module* is created. This *Module* represents a digital-twin of the real instance. By *ExternalReference*s, *ComponentModel*s can be annotated with a reference where to find further information about them, e.g. simulation models, manuals, etc.

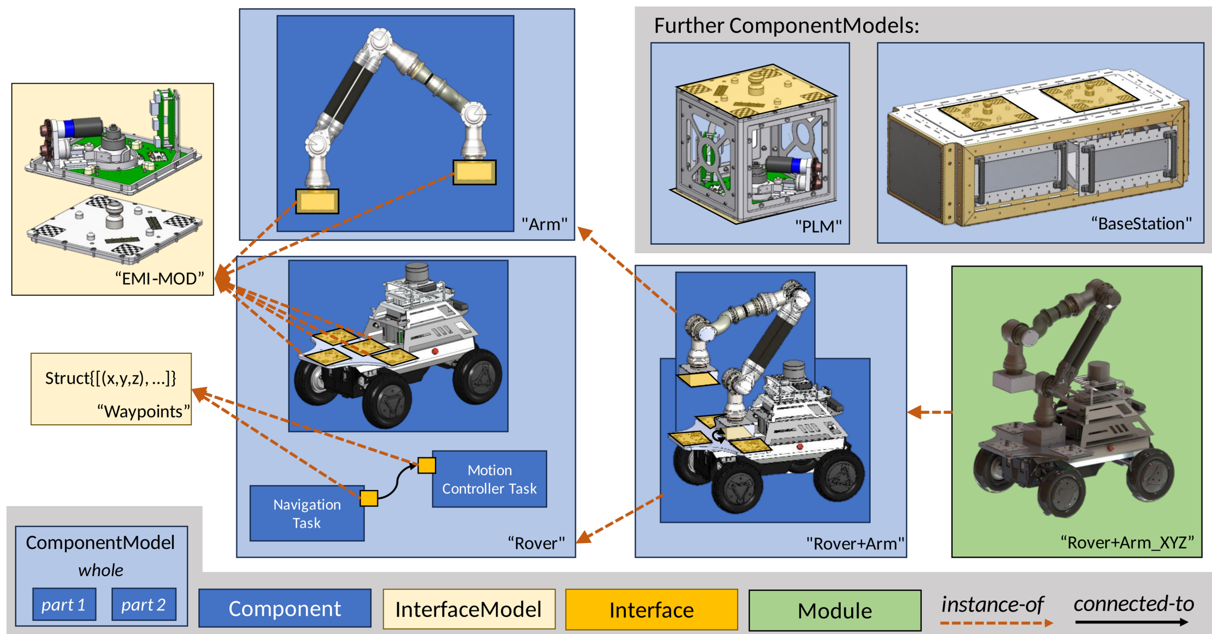

Figure 1: Example of the composition of *Components* by their *Interfaces* to a new *ComponentModel* of a mobile manipulator and finally its instantiation to a *Module*.

Relations defined on these derived *XTypes* store the dependencies between them. For example, the *instance-of* relation between a *Component* and a *ComponentModel* specifies that the *Component* has been instantiated from that *ComponentModel*. The *connected-to* relation between *Interfaces* encodes the connectivity between *Interfaces* of *Components* inside a *ComponentModel*. That relation would thus represent the flow of data/information between software components in the software domain. Fig. 1 depicts an example of how the XTYPES formalism can be used to model a robotic leg consisting of multiple instances of a certain robotic joint. The *part-of* relation specifies that a *Component* is part of some *ComponentModel* (whole).

XTypes also support the creation of abstract *ComponentModels*/*Components*, which enables the creation of template *ComponentModels* in which the abstract *Components* later on can be implemented by non-abstract *Components*. As *ComponentModels* can also have abstract ancestors these abstracts can be used to help the user find matching *Components* when instantiating templates.

To reuse components, persistently store and share the system design with other users, the XDBI layer implements an interface to the so called XRock-database. The database serializes/deserializes the *XTypes* and their adjacency lists (per relation) in a JSON format. It supports multiple users, local and remote access (via REST API), as well as mixed applications in which multiple databases can serve as information sources. Furthermore, it defines a set of functions for retrieval of as well as the consistent storage of (sub)graphs from/into the database.

2.2. User Interfaces

Several GUIs have been implemented to facilitate designing and planning modular systems. They shall help the user to visualize the different aspects of composing and configuring the components. This way it's not only made easier for non-expert users but also tedious tasks that are error-prone, when doing them manually, are simplified. There are tools for adding new atomic *ComponentModels*, assembling higher level *ComponentModels*, and configuring *Components* in such *Component* networks.

The **XDashboard** is a web-interface that gives the user a central cross-platform entry to the the different tools in the composition and configuration process. From this dashboard the user can access the different user interfaces associated with the current development step.

With the goal of having the complete workflow in the XDashboard represented, external tools like the XRock-GUI (Sec. 2.2.2) are highlighted according to the current step in the workflow and can be launched from XDashboard. New and existing tools can also be modularly integrated, like the Component Browser and the Deimos module (see Sec. 2.2.2). Those plugins are implemented as their own Python module which can be dynamically loaded, allowing for an easy way to integrate new tools.

The **Component Browser** acts as a high-level interface to an instance of a XTypes database. Besides inspecting *ComponentModel*s, their properties, and relations, as well as editing any of these properties or relations, it also allows to combine existing ones to new, yet unsophisticated *ComponentModel*s without connections. Thereby it is already one of several methods to add new building blocks.

2.2.1. Adding Building Blocks: Using the **Atomic-model-GUI** basically anything can be represented, stored, and later edited again as *ComponentModel* by defining its properties and interfaces. For some special cases this process is made easier by providing more specialized tools.

When a robotic part/system is not yet in the database, a proper representation of that hardware has to be created first. **Phobos** [10] supports the user in annotating the pure kinematic and visual robot model exported, e.g. from CAD (Computer-Aided Design) software, with all necessary annotations, i.e. sensors, collision shapes, interfaces, etc. Phobos is an open-source add-on to the open-source 3D modeling tool Blender, and thus allows the user to edit all this in a 3D environment. Once done, the model can simply be added via XDBI using the respective script included in the toolbox. This script collects all necessary information from the created model to a new *ComponentModel* and stores it to the database.

In the ROCK framework, a single software component is called task. ROCK already provides a model description of each task, which can be used by a command line (CL) tool to import it as new *ComponentModel* via XDBI. Using these CL-tools, the maintenance of models in the database can be automated. A CI-job (continuous integration) can run through a set of repositories containing ROCK tasks, or hardware models respectively and call the appropriate CL-tools to update the database. While versioning the created *ComponentModel*s by release tags is possible, mixing different versions of one software on the deployed system must be avoided.

2.2.2. Composing, Connecting and Configuring: The **XRock-GUI** is the main tool in the MODKOM toolbox for creating, composing, connecting, and configuring *Component*s inside a new *ComponentModel*. It is a graphical user interface that has database access via the XROCK-IO-LIBRARY-plugin that wraps XDBI. In the viewport all *Component*s with their *Interface*s are visualized as editable nodes where the user can view and edit the connections as edges between them (see Fig. 2). The GUI ensures that only valid connections can be drawn. For each *Component* the XRock-GUI can be used to edit its configuration, this includes the usage of global variables that are resolved at deploy-time. This minimizes the work of manually editing thousands of lines of configuration manually and ensuring everything is consistent.

When creating templates, abstract *Component*s can be used and later they can be specialized by their implementations. The **Abstract-GUI**-plugin can be used to define that one *Component* implements another and how its interfaces realize the abstract ones.

To facilitate assembling hardware modules, the XDashboard also provides access to the web-app **Deimos**. It allows three-dimensional composition of the parts of a *ComponentModel*, helping the user to ensure correct orientation of the modules with respect to each other.

2.3. Deployment

When the user is done describing the designed system in one or more *ComponentModel*s, these can be deployed using CL-tools. These tools are yet specialized for the use with ROCK [9] and Autoproj [11]. By just passing the respective *ComponentModel*, the tool will generate a buildconf[1] including local package sets[2] and also bundles[3] the CND files (Component Network Description) that define which ROCK-Tasks have to run and how they communicate. The buildconf including that bundle can then be installed on the robotic system.

During the deployment progress several things happen: The CL-tool instantiates first all *Component*s to *Module*s. As next step, also simulation models of assembled hardware will be assembled and provided. Then, all global variables are applied down through the tree of *Module*s, including paths to the simulation models, or dependent configuration settings. As final step, the CNDs and configuration files for the different ROCK-Tasks are exported. This automation ensures that essential elements are integrated into the robot's environment in an efficient and consistent manner. The deployed CNDs can then be used to start-up the task corresponding to the current robot configuration or mission goal [12].

3. Hardware Building Blocks

For the basic toolkit payload modules (PLM), base modules, and a modular manipulator arm are developed; Hunter SE from AgileX is involved as one COTS component (see Fig. 1) These individual subsystems can be combined with each other via the electro-mechanical interconnect MODKOM (EMI-MOD). The MODKOM toolkit includes active and passive EMI-MODs [13].

The mentioned PLMs are housings with an EMI-MOD and are intended for specific applications, like stereo camera, computation, and communication as well as environmental sensing or as power module. Thereby, they can e.g. expand the scope of tasks of the rover and the base module. [14]

The base module is a core structure with removeable compartments containing a set of relevant robotic sub-subsystems, like on-board computer, electrical power supply, and a communication module. The top of the base module can be equipped with up to two passive EMI-MODs.

The modular manipulator is also equipped with two active EMI-MODs as end effectors. By using the EMI-MOD, all mentioned systems can be combined with each other, also the Hunter SE. A platform fitted with at least four EMI-MODs will be mounted on the rover Hunter SE to provide the capability to interact with the other modules. With the help of the software framework described in this paper, autonomous (re-)configuration between the systems involved can be enabled by using the EMI-MOD. Other standard interconnects are also evaluated [15].

4. Markerless Visual Servoing for Docking

Online (re-)configuration requires reliable docking, that was performed by marker dependant visual servoing in the past. To bypass unrecognizable markers, i.e. when covered with dirt or in bad lighting conditions, markerless visual servoing is developed as an complementary approach. Introducing the use of additional cameras and algorithms, machine learning methods such as 2D object detection and 6D pose estimation are applied to detect the EMI-MOD passive side.

Pix2Pose [16] calculates the loss function using symmetric poses and predicts 3D coordinates from cropped images containing relevant parts from the 2D detection model. The machine learning pipeline is initially implemented in Python using the Perception for Autonomous Systems (PAZ) library [17], acquiring inference for quaternions and translation vector information relative to

[1] The buildconf is a configuration that describes for an Autoproj environment which package sets to include and which software to build.

[2] A package set describes where software repositories can be fetched and how to get their binary dependencies for the respective operating system.

[3] Bundles in Rock are collections of the files necessary to run the created setup, i.e. Component Network Description files (.cnd), OroGen task configurations, Ruby scripts for components, etc. They can be version controlled.

the camera, then providing it via a ROCK task. First experiments to evaluate feasibility and performance are carried out for the EMI-MOD on PLM.

Due to the absence of a PLM dataset, the training process employs a 3D PLM model from the Pyrender library [18]. The dataset is created by extracting objects from rendered images and blending them with various background images from the PASCAL Visual Object Classes dataset (VOC) and simulated environments. Furthermore, the images undergo several augmentations, like random occlusions, controlled blurring effects, and so on.

5. Dynamic Mission Planning

Modularization allows various module (re-)combinations, posing a combinatorical challenge in planning domain of determining how and when to (re-)configure a system before and during a space mission. TemPl [19] tackles this challenge by planning missions based on an ontological organization model for modular robots, called MoreOrg. Dynamic mission planning allows for online execution and knowledge updates during missions, enhancing flexibility in dealing with failures, changing conditions, and hierarchical goals, implemented in three steps.

(i) **Deployment of TemPl-Task**. TemPl is a planning system for reconfigurable multi-robot systems, available as a software *Component Model* for ROCK tasks, allowing easy integration into robot software stacks and online use by the system itself. To optimize the TemPl-task regarding performance and solution quality, methods such as A* and multi-objective optimization of the cost function are investigated.

(ii) **Development of Ontology-Editor-Task**. To update the knowledge base (either organization model or the mission constraints) during the mission, a second ROCK task is developed, which utilizes a C++ implementation [19] of the OWL-API [20].

(iii) **Integration of Ontology-Workflow to Software-Framework**. An ontology workflow was developed, that guides the framework user through the ontological modeling of composed robots. A GUI was developed to link *Component*s with ontological descriptions in an ontology-database, ensuring efficient usage of the organization model through automated instantiations and logic checks.

6. Current State & First Results

Fig. 2 shows an example application in the context of human-machine-interaction. In the scope of the application the wheeled robot has to react on the behavior of the human. If the human moves towards the legged robot, the wheeled robot predicts that the human wants to work on the legged one and knows that tools from the shelf are required. So, it autonomously moves to the shelf to pick up the tools and brings them to the human.

In this application the robot has to perform different behaviors, such as observing the human's intention, navigating through the hall, detecting object poses and manipulating/grasping them. All single behaviors are modeled based on the XRock tooling purely via graphical user interfaces. The `move_to` behavior is partly presented in Fig. 2. Only two *Component*s are used in the top-level network, which are the drivers of the simulated robot and the `move_to` behavior implementation. This way, the simulated driver *Component* can be exchanged by the driver *Component* of the real system without the need to modify the other layers of the software architecture. The `move_to` network consists of*Component*s, too, which represent other layers of the software architecture such as the navigation stack including a trajectory follower, obstacle detection, and avoidance. In this specific architecture mapper and path planner *Component*s are stored in another so called "base" network.

Overall, five different behaviors are setup in the example application. Each behavior is represented by a software component network with 20 different *Component*s in average. A single component network/behavior, represented in a human readable text file that is executed

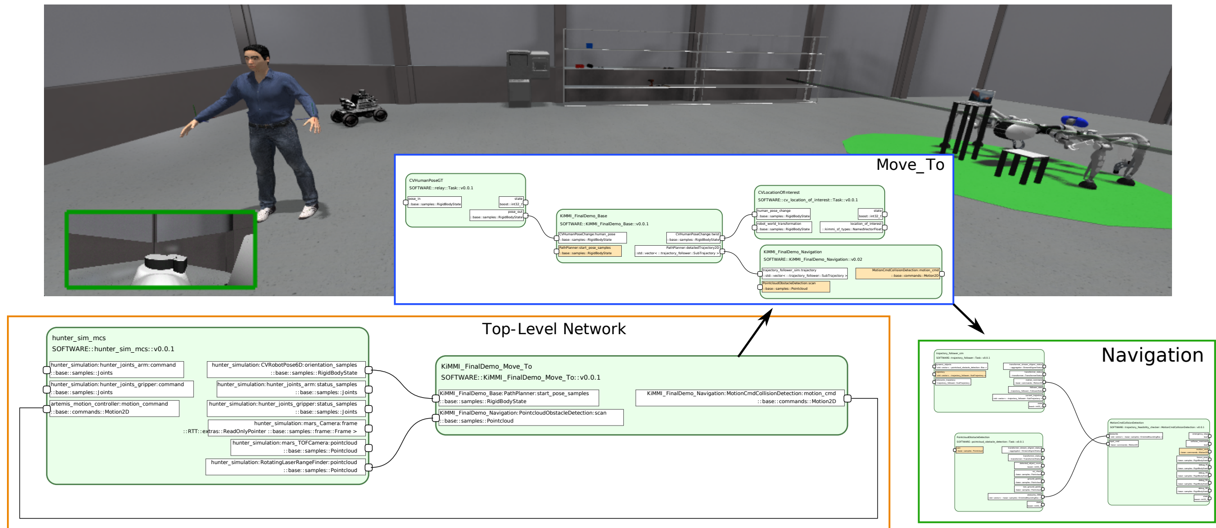

Figure 2: Example application setup based on the presented framework. Top/Image: A simulated environment where a mobile robot has to support a human in a lab. Bottom/Graph: The different layers of the modeled "move_to" behavior, which is one of several robot behavior to perform in this application. The highlighted *Interface*s of one layer represent the *Interface*s of the *Component*, if used on a higher layer. The shown networks are snapshots of XRock-GUI.

by the ROCK runtime environment, covers approximately 1200 lines of network definitions (tasks, connections, deployments, configurations and transformations). These text files were generated and maintained manually, before the XRock-GUI and the toolbox were introduced. However, some configuration sections still have to be written manually; for the behaviors of the example application approximately 100-200 lines of text must still be processed manually. This highly modular approach of designing the dataflow architecture reduces the overall complexity for the user and allows to share solutions/networks between different applications and robots. Additionally, due to the graphical user interfaces no syntax errors and almost no typos can be made by the user when designing new component networks.

7. Discussion & Outlook

By applying the here described methods (see Sec. 6) to other currently running projects, the amount of hand written ROCK task connections and configurations could already be reduced immensely; and thereby, the progress became less prone to mistakes. Providing these tools, MODKOM contributes to simplifying the development of space robots, especially modular ones. Some of the presented libraries have already been published open-source (Phobos, XTYPES-GENERATOR, XTYPES, XDBI, XROCK-IO-LIBRARY)[4]. Publishing the yet missing tools is planned later in the project, when their state has advanced. So, in the end the toolbox shall be publicly available.

The next steps are extending the *ComponentModel*s by further semantic relations amongst each other, to better represent ontology relations, e.g. to which hardware a driver relates or similar. This will also benefit integration with mission planning in TemPl. Furthermore, deployment tools and process of the software stack has to be refined and tested on the robotic hardware.

Using the example of a planetary mission scenario, the here presented functionalities will be put to test at the end of project. In this scenario a modular mobile robot shall reconfigure

[4] They can be found as repositories here: https://github.com/dfki-ric

itself to adapt to different tasks like exploration and placing a sensor station. Thus, not only the execution shall be evaluated but also how this robot can be designed, configured, assembled, and comissioned [15].

In general, the modeling approach can be used for any framework where software modules can be modeled by their interfaces and data exchange by connections between these interfaces. The adaptation of the yet ROCK-specific implementations to other robotic frameworks like ROS would be a possible subject to future projects.

Acknowledgements

The authors would like to thank the MODKOM and KiMMI-SF team and all supporting staff at University of Bremen Robotics Research Group as well as DFKI Robotics Innovation Center. The work presented is part of the project MODKOM and the current state was evaluated in project KiMMI-SF. Both are funded by the German Space Agency (DLR) with federal funds of the Federal Ministry for Economic Affairs and Climate Action in accordance with the parliamentary resolution of the German Parliament under grant no. 50RA2107 and 50RA2108 (MODKOM) and grant no. 50RA2021 and 50RA2022 (KiMMI-SF).

References

[1] Rossetti D, Keer B *et al.* 2015 Spacecraft Modularity for Serviceable Satellites *AIAA SPACE 2015 Conference and Exposition* (Reston, Virginia: American Institute of Aeronautics and Astronautics) ISBN 978-1-62410-334-6

[2] Post M A, Yan X T and Letier P 2021 Modularity for the future in space robotics: A review *Acta Astronautica* **189** 530–547

[3] Hancher M D and Hornby G S 2006 A modular robotic system with applications to space exploration *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)* (IEEE) pp 8–pp

[4] Brunete A, Ranganath A *et al.* 2017 Current trends in reconfigurable modular robots design *International Journal of Advanced Robotic Systems* **14** 1729881417710457

[5] Salemi B, Moll M and Shen W M 2006 Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE) pp 3636–3641

[6] Tosun T, Jing G *et al.* 2017 *Robotics Research: Volume 1* (Springer) pp 237–252

[7] Sonsalla R U, Brinkmann W *et al.* 2022 Towards modular components as building blocks for application-specific configurable space robots *Proceedings of the 16th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA 2022)* (Leiden, Netherlands: ESA/ESTEC)

[8] Stanford Artificial Intelligence Laboratory et al Robotic operating system URL `https://www.ros.org`

[9] DFKI GmbH Robotics Innovation Center The robot construction kit. accessed on 20.09.2023 URL `http://www.rock-robotics.org`

[10] von Szadkowski K and Reichel S 2020 Phobos: A tool for creating complex robot models *Journal of Open Source Software* **5** 1326

[11] Joyeux S autoproj. accessed on 20.09.2023 URL `http://www.github.com/rock-core/autoproj`

[12] Wirkus M, Arnold S and Berghöfer E 2020 Online reconfiguration of distributed robot control systems for modular robot behavior implementation *Journal of Intelligent & Robotic Systems* **100**(3) 1283–1308

[13] Brinkmann W, Yueksel M *et al.* 2023 Multifunctional interconnect for future modular planetary robots *74th International Astronautical Congress (IAC)*

[14] Brinkmann W, Cordes F *et al.* 2018 Modular Payload-Items for Payload-assembly and System Enhancement for Future Planetary Missions *2018 IEEE Aerospace Conference* (Big Sky, Montana, USA: IEEE Comput. Soc. Press) pp 1–10

[15] Sonsalla R, Wiedemann H *et al.* 2022 Toolbox design to demonstrate application-specific configurable space robots using modular components *International Astronautical Congress, IAC 2022* (IAF)

[16] Park K, Patten T and Vincze M 2019 Pix2pose: Pixel-wise coordinate regression of objects for 6d pose estimation *Proceedings of the IEEE/CVF International Conference on Computer Vision* pp 7668–7677

[17] Arriaga O, Valdenegro-Toro M *et al.* 2020 Perception for autonomous systems (paz) *arXiv preprint arXiv:2010.14541*

[18] Matl M 2019 Pyrender. `https://github.com/mmatl/pyrender`

[19] Roehr T M 2022 Active exploitation of redundancies in reconfigurable multirobot systems *IEEE Transactions on Robotics* **38** 180–196

[20] Horridge M and Bechhofer S 2011 The owl api: A java api for owl ontologies *Semantic web* **2** 11–21