# Modular Software for an Autonomous Space Rover

Sylvain Joyeux**, Jakob Schwendner*, Thomas M. Roehr*

*Robotics Innovation Center, DFKI, Germany
e-mail: firstname.lastname@dfki.de

** e-mail: sylvain.joyeux@m4x.org

## Abstract

The increasing difficulty of space exploration missions, human support and maintenance of infrastructure push the expectations for future robotic systems. Complex software systems are required to autonomously perform mobile manipulation and interaction tasks in a robust fashion. Component based software development and software frameworks provide a way for efficient implementation. This paper introduces the Rock framework, which uses a model driven approach to handle the complexity of component networks which can reconfigure at run-time. The Artemis Rover, which participated at the SpaceBot Cup competition is used as an example where the Rock framework has been successfully applied to a mobile manipulation system.

## 1 Introduction

Robots are an effective tool for the exploration of our solar system. They can be scaled to the mission, and are much more resilient when it comes to the harsh environments of planetary surfaces and interplanetary travel. The only resource they require for ongoing operation is energy, which can either be taken with them using a Radioisotope Thermoelectric Generator (RTG) or generated using solar power. On-board instruments can be used for in-situ science. All in all, robots have a number of advantages over human exploration. What they currently lack is the intelligent decision making abilities of humans. While artificial intelligence research has come a long way and software systems can beat humans in certain restricted settings, AI systems are currently no replacement for direct human intervention. However, the communication latencies and visibility restrictions involved for most space exploration targets often make it difficult to tele-operate the systems. Autonomy is a key enabling factor to improve the abilities and ultimately usefulness of robots in space exploration.

As the number of elements in a software system grows, the number of potential interactions between these elements increases at least quadratically. Thus software complexity is a significant time and cost factor which does not scale linearly with mission requirements. On the other hand, software has the potential for cost savings as well. Unlike hardware, parts of the software can be reused over different missions. The key is to use appropriate software engineering methods which allow breaking down the complexity into manageable parts, which can be reused in different contexts, and even be shared with different domains of robotic applications.

One of the defining factors of software for space is reliability. Strict development, testing and qualification guidelines are currently used for flight software. The question is if these procedures scale well with the complexity of the system, and if this approach can be held up with increasing amount of AI that comes into the system. Even when the components are modularized, the integration needs to take the interactions and the different states of the system and its modules into account.

Frameworks can help to manage the complexity of software. By imposing standards for development and providing tools, it becomes easier for software developers to design and implement software modules that work well together. These software frameworks have so far mostly been used in the development process of robotic systems for space, and not for actual flight systems. This is not exclusively true anymore [1], and frameworks and software component libraries are actively evaluated for their suitability for flight systems. The research on software frameworks is very active, and mobile outdoor robots pose an interesting problem set. One could argue that most parts of robotics itself is the art of integration. Frameworks are the tools to do this on a software level.

The first part of this paper presents a short overview on existing frameworks for the integration of robotics software. Then an overview of the design rationales for the Robot Construction Kit (Rock) is given. Frameworks are developed to improve application development and the second part of this paper illustrates the benefit of Rock using the Artemis rover as an example. This rover was developed for the Spacebot Cup [9] – a robotic competition organized by the DLR, which took place in 2013.

## 2 State of the Art

The idea of component systems dates back to 1968, where [15] argues for strong similarities between hardware and software industry. Although, object-oriented programming significantly improved the reusability of software, in 2002 [23] stated that "modelling of component-based systems is still a largely unsolved problem". This situation has changed and component-based development and model-based engineering found broad application including robotics research.

The main driver for building systems from components is efficiency. Clear encapsulation along with clear interfaces allows developers to be more productive by reusing already existing software modules or components in general [5].

Developing a component system is not trivial and even reuse of components cannot be considered straight forward [5]. The identification of generic and thus reusable components can be challenging and even involves significant additional cost – three to four times compared to developing components for a single application context [23]. However, the benefit of a component-based system should become clear when integrating a complex system. The effort spent should be minimal and focused on linking components to form assemblies with high-level functionality. Nevertheless, there is no free lunch for building a system from components and several processes need to be in place to support the development of components.

A workflow has to account for different development phases, i.e. design, deployment and runtime phase. [11] suggests further that in the design phase components should be templates which are applied in the deployment phase to construct new so-called composite components. In addition to this staged development process they expect tools to support this process. [11] limit their focus on the first two phases and assign reconfiguration of a component-based systems to the runtime phase. Existing visual programming environments such as LabVIEW give additional insight of what is required in a component-based system such as a component catalogue as well as standardized interfaces and standardized datatypes. While such tools allow modelling with components they are also limited to design and deployment phases – management and reconfiguration of the component-based system at runtime is not considered in the modelling.

These days, Robot Operating System (ROS)[17] is the most popular development framework in robotics research community. Though it is a component-based system its development workflow does not rely on the usage of a well defined component model specification. ROS provides a general interface for components (aka nodes), but it does not provide an explicit specification – the specification can only be inferred when running a node. [4] intends to close this modelling gap, but at the same time shows that current practices of developing with the ROS lack modelling and follow a rather ad-hoc development approach. ROS offers a low-resistance workflow and a fast-path to a component based systems, but the system design capabilities remain less scalable – without explicit modelling ROS does not provide tooling to actively manage an increasing number of components during the runtime phase.

In the following, this section will present, but not discuss, the main paradigms in modern robotic control software. We will use some of the major names in robotic software frameworks to illustrate these paradigms and concepts:

- ROS [17], the leading robotic software framework in the research community
- Orocos Real-Time Toolkit (RTT) [22, 21], a middleware-agnostic component implementation which is the basis for the Rock toolchain
- GenoM versions 2 [6] and 3 [13], a component development tool developed at LAAS/CNRS that sees some general use, including in the space domain, especially at ESA

It is interesting to note that Robonaut 2 [1] - the only system that has flown in an actual space mission with a state-of-the-art robotic framework – runs a mix of ROS and RTT.

## 2.1 Middlewares, Software and Control Architectures

In literature a distinction is often made between the concepts of middleware, software architecture and control architecture. However, while mentioned, the distinction is rarely explained. This section will try to characterize what is meant by each, and why – in our opinion – this is a somewhat artificial distinction.

The *Middlewares* implement high-level APIs and protocols that allow to exchange information (messages, procedure calls), inter-process, in a network transparent way[1]. Examples abound in the software engineering world, such as CORBA, DDS, AMQP, or ZeroMQ. The *Robot Software Architecture* defines software components as well as a way to use a middleware (or multiple middlewares) to interface them. In effect, it is the definition of the component in a system. Here comes a confusions: some middlewares, such as CORBA, already have a concept of components, and somebody's robotic software architectures (such as ROS) can be used as another's middleware. The *Robot Control Architecture* defines how to use components to control a robot – the components being implemented in a Robot Software Architecture.

In theory, a given robot control architecture can apply to more than one robot software architecture (and vice-versa). In practice, however, software architectures define *how* to interact to components, but define it loosely.

[1]`http://en.wikipedia.org/wiki/Middleware`

The control architecture is what defines how to write these components in practice. The components therefore end up being specific to the control architecture, even though the software architecture is not.

In other words, in practice, control and software architectures are often tied (one could say that the control architecture is the model for the software architecture). The middlewares is what can be reused in various implementations.

## 2.2 Data Flow

What has long been specific to robot software architectures, when compared to more "software engineering" component-based systems is that the main interaction paradigm is the data stream: robot software is a real-time processing system, in which algorithms process data as it is sensed in order to perform planning and actuator commands. One of the main issue is to let data flow between processing algorithms. A central question is therefore how one defines the way which data stream goes into which component. The two leading paradigms are publish/subscribe and point-to-point. Another less applied method is the tuplespace (or blackboard), which is used e.g. in Cougaar [20].

In *publish/subscribe* systems, components **separately** announce that they publish and/or subscribe to some data stream, the data streams being globally identified. Identifiers are commonly human-readable names such as `/left_hand/camera`. The rationale behind this scheme is that one should reason in terms of which data a certain component needs, not how this data gets generated. ROS is a well-known implementation of this scheme, which is also the basis for widely used middlewares such as DDS or RabbitMQ.

In *point-to-point* systems, the data flow is established explicitly between the outputs and inputs of components, i.e. one would link the camera driver to the image processing component. This is the paradigm used by RTT. It is interesting to note at this point that one can easily build a publish/subscribe system on top of a point-to-point one, (OpenDDS is for instance built on top of CORBA), but emulating a point-to-point system on top of a publish/subscribe one is not fully feasible: a topology where $A$ feeds data to $X$, $B$ to $X$ and $B$ to $Y$ for instance cannot be represented. Such topologies have already been encountered in practice by the authors.

## 2.3 Remote Procedure Calls

In parallel to the dataflow, one needs to control the components' execution. The oldest and most widely used method for this is to implement some form of Remote Procedure Call (RPC). An RPC is a mechanism to perform procedure calls (in the programming language sense) in an inter-process, network-transparent way. This is a very common paradigm to allow cross-process interaction. CORBA is for instance built on this, and it is also the underlying paradigm for ROS services and RTT operations.

## 2.4 Task-State Pattern

The task-state pattern [12] could be summarized as *asynchronous remote procedure call with progress tracking*. This is the pattern that GenoM requests and ROS actionlib implements, and is present as well in other major frameworks. When using a task-state, one first sends a request to start a given task (with parameters). Said task will then be started asynchronously, and the remote component that implements it will report about the execution progress. Additionally, the caller often has the ability to request an interruption of the task. The core idea behind the task-state pattern is to make the component layer provide an interface that high-level coordination paradigms (state machines, planning) know about. In other words, task-state implementations are meant as a way to glue dataflow-based paradigm (components) with task-oriented control (planning, state machines).

## 2.5 Configuration

All major robotic frameworks offer a configuration interface. ROS for instance has a "parameter server" and RTT is using properties. Configuration is separated from the dataflow in that the configuration values are mostly static (discussed later in this section), and are read at startup by the components. Separating the configuration interface from the dataflow interface also makes clear what is being processed by the component (the dataflow) and what defines how the processing should be done (the parameters).

## 2.6 Dynamicity

When looking at state of the art frameworks, one sees that the ability to reconfigure, i.e. change the shape and parametrization of a component network is often an afterthought. Changing parameters is commonly handled by using RPC mechanisms creating two parallel ways to configure a component, e.g., ROS' functionality for dynamic reconfiguration works by standardizing a certain service that gets called when a dynamic parameter is set.

The ability to change the dataflow is often non-standard, and ends up being embedded in the components themselves. In GenoM with pocolibs and ROS, one can implement hand-written requests that allow to change which streams the component is subscribed to or publishes. Because of the practical complexity of the technique, as well as its non-standard nature, it is seldom used (and cannot be embedded in tooling). RTT is different in this respect, as components' ports can be disconnected and reconnected at will, even while the component is running.

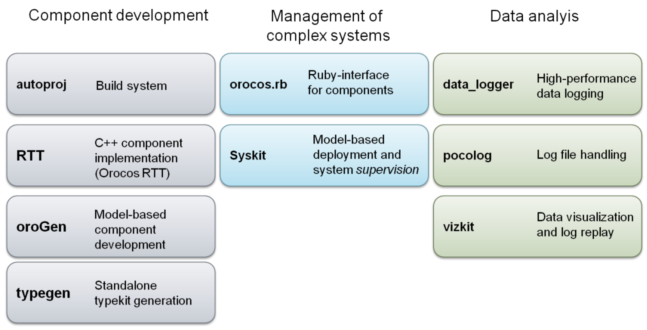
Component development
Management of complex systems
Data analyis
autoproj
Build system
orocos.rb
Ruby-interface for components
data_logger
High-performance data logging
RTT
C++ component implementation (Orocos RTT)
Syskit
Model-based deployment and system *supervision*
pocolog
Log file handling
oroGen
Model-based component development
vizkit
Data visualization and log replay
typegen
Standalone typekit generation

**Figure 1. Key parts of the Rock framework**

### 2.7 Model-Driven Development

Regardless of the framework of choice, everyone understands nowadays that building the software of a complete robot is no small task, and leads to a quite complex system with a lot of "moving" parts. The leading path to tame this complexity has been to integrate Model-Driven Development (MDD) within the robotic software development workflow. However, there is really no mainstream model-driven toolchain in the robotic domain. GenoM was first to use code generation, but the source model used by the code generator has not been used anywhere else. The version 3 of the GenoM tool used the same approach to offer cross-framework component development (at this time, it supports LAAS' own middleware pocolibs, ROS and BIP [2]). In the RTT world, the BRIDE tool [4] got developed within the BRICS project [3], with the same goal of being cross-framework. On the one hand, while the tool itself is cross-framework, the component developer targets a given framework. In other words, unlike in GenoM3 where a component developed for pocolibs can be deployed in ROS, a component developed in BRIDE for ROS cannot be reused within RTT. On the other hand, BRIDE also targets deployment, i.e. the creation of component networks, in addition to pure component development (something that GenoM completely leaves to the target software frameworks).

## 3 The Robot Construction Kit

The design of Rock was based extensively on our critical analysis of the state of the art in robotic frameworks. This section will present the various drivers for Rock's design and development, and will compare Rock to the state of the art. Not all parts of Rock will be discussed in the following section, but Figure 1 tries to provide a general categorization and overview on the parts that form the overall framework.

The main design drivers in Rock were:

**Application Agnostic** the ability to reuse a component across applications, adapting it at deployment time to the requirements of the application.

**Single-Purpose Components** Components should be designed to do one thing and do it well.

**Composability** the ability to build subsystems using components, as well as build bigger (sub)systems using these already defined subsystems. In addition, the ability to reuse subsystems across platforms and applications.

**Adaptation and Robustness** the ability to reconfigure a running component network at will, in order to always use "the right tool for the job". The driver here is to be able to safely change algorithms to fit the situation the system is in, instead of trying to develop an universal component that can do a certain thing in any situation.

### 3.1 Application Agnostic

At the component level, the main philosophy behind Rock's design are the separation of concerns between the development aspects of the "4Cs": Computation, Configuration, Connection and Communication [16, 18]. A Rock component is the pure representation of a Computation, i.e. the encapsulation of an algorithm that processes a set of input data and produces output data, using some parameters (Configuration interface) to define what it should do. Where the data comes from and goes to (Connection), as well as how (Communication) are left out of the component writer. This choice is made at *deployment time*, i.e. are made considering the actual application.

Since RTT already separates the 4Cs while offering all the capabilities a state of the art robotic framework offers, it has been chosen as Rock's underlying component implementation. To improve usability and standardization of components Rock offers the oroGen code generation tool. A component writer that uses oroGen needs to know about 12 *method calls* and no C++ RTT-specific declarations (which – like ROS – is very template-heavy). In addition, because written components have very little RTT-specific code, the code generation tool provided the ability to migrate across one major revision of RTT (version 1 to 2), as well as updates to the Rock tooling, without requiring any change from the component's developer (a gain any well-designed code generation tool such as GenoM provides).

The second use of the code generator is to provide *sane defaults*. While the 4Cs is a key design driver within the Rock toolchain, its "pure" use makes it actually harder to *discover* new components. Indeed, one needs to not only know what the component does, but also should determine how it should be deployed. oroGen components provide sane defaults for the configuration and communication aspects, allowing new users to only care about the connections (make the data flow across components). These defaults can then be overriden to better fit the application's requirements.

Additionally, Rock leverages RTT's design as a pure encapsulation of the Computation aspect. oroGen as well as Rock's tooling only have to target the RTT API since RTT already provides the means to *simultaneously* interface with multiple middlewares (separation of the Communication concern). One can for instance use CORBA, ROS and POSIX Message Queues *simultaneously* to transfer data within the same system. This offers true interoperability: Rock components can run alongside unmodified ROS nodes, both types of components being controlled by Rock tooling. This is in stark contrast to other code generation approaches, such as GenoM3, where the framework is truly the code generator plus the underlying runtime platform, e.g. GenoM3+ROS or GenoM3+pocolibs. GenoM3 by itself is not a *a software framework*.

## 3.2 Single-Purpose Components

As mentioned, the task-state pattern's goal was to ensure that a component-based layer (dataflow-driven) provides an interface that matches what is expected by traditional task-based coordination mechanisms (plan-based systems, state machines, …). However, its introduction has in practice complexified the task of the component developer tremendously. Not only components are meant to process data in one particular way, but must also provide all the modes that all applications that could use this component need at the task level. This complexity was required by the fact that the dataflow was assumed to be static (no coordination mechanism really knows how to deal with the component's connection network). Ways to change the component network's behaviour therefore have to be provided through the task-state pattern. It is the experience of the authors that the components' code end up being complex, having seldom-used tasks that were needed by some applications in the past, and that understanding the state of a running network is near impossible: it depends on both the component's configuration, but also which tasks are currently running and sometimes to which tasks have run in the past.

Interestingly, RTT components do not have the ability to provide multiple tasks within the components. Instead, in a Rock system, one component *is* one task. The system's behaviour is therefore determined by (1) which components are currently running, (2) the components configuration (the values of their properties) and (3) how they are connected. In practice, it allows to think of components as having to do one single thing (they are *single-purpose*), simplifying the component's code and the user's understanding of its role.

## 3.3 Composability

Another major advantage of using a code generation tool such as oroGen as well as a development paradigm that gets rid of the task-state pattern is that components are fully standardized (the only important bits are the runtime state machine, which is identical to all components, as well as its input and configuration interfaces). A component that can be run by simple deployment tools *can* be run by Syskit [7], Rock's advanced model-based system deployment tool, without modification. Systems based on the task-state pattern would probably have to add new tasks to it to fit the needs of their new application(s).

Syskit is a model-driven tool that allows to define compositions of components *in isolation*, and then a correct-by-construction way to combine them together. A key aspect of Syskit is that the composition is done *with sharing*, thus truly allowing to define compositions in isolation. A trajectory following composition that includes an IMU device does not need to know that another composition also uses the IMU. The tool will detect this and make sure that the IMU device (and its postprocessing algorithms) are present only once in the final network. This is a major advantage during the system's engineering as subsystems can be designed separately but still run together. Finally, the usage of a point-to-point connection model ensures that changing one component's interface cannot affect the rest of the system without the designer's intervention.

This is in contrast with the way nodes are composed in ROS. Compositions in ROS (often called "stacks") is based on a set of conventions on the naming of topics (ROS' nodes input/outputs) and services. The stack is then "made" by putting the stack's nodes in a separate namespace, to isolate their topics from the topics of the rest of the system. When a single stack is present, this is obviously an elegant solution. As soon as stacks are mixed, and parts get reused *across* stacks this elegance breaks. Moreover, it does not handle change in the nodes very well (if at all), since the creation of a new topic in a node can in principle collide with an already existing topic name and cause significant interference. Finally, nothing in a publish/subscribe system like ROS can warn at startup time that a publisher / subscriber got mislabelled (because, for instance, of a typo): (i) having a publisher-without-subscriber or subscriber-without-publisher is a normal situation when the system is being setup and (ii) getting the wrong provider for a subscriber simply cannot be detected if the types match.

## 3.4 Adaptation and Robustness

An autonomous robot needs to adapt to its situation. This is the very thing that will make it autonomous. Moreover, given the complexity of the task at hand, it is an illusion to think that we can build a single perfect versatile algorithms that will solve one of robotic's problems (let's say "localization") and cover every possible situations. The key to having *robust* autonomous systems is therefore, in the authors' point of view, to allow for seamless switching – at runtime – between different component networks.

Here, again, the design of Rock components as well as the use of oroGen to provide models for the components allow to adapt the component network without requiring any application-specific change to the components. Using Syskit, networks of components are designed statically, the tool being able to adapt from any state of a component network to a desired component configuration. Given that the start state can be arbitrary, it is possible to transition from non-nominal states, i.e. recover from component faults. Syskit allows to combine these composed network definitions within higher-level coordination models such as a state machine, through the use of the Roby [8] plan manager.

## 4 The Artemis Rover

The SpaceBot Cup [9] challenged 10 teams to perform a mission scenario with constraints similar to a space mission. The autonomous mobile manipulation systems had to be developed within 8 months. In the competition the robotic systems (ground-based or flying vehicles) were given a task to find, retrieve and transport objects in an unknown, mars like, rough terrain. A low-resolution elevation map was provided before the competition. Three objects were distributed in a 21 m × 21.5 m area at unknown locations. Two of them had to be collected and assembled with the third object. After assembly the system had to return to the starting point. Communication with the deployed robotic system was limited to three communication windows of 5 minutes each. Only during these control points direct control was possible and in general communication with the remote system suffered a delay of two seconds in either direction. The full mission had to be completed within 1 h.

### 4.1 Software Architecture

The developers of the software architecture in Artemis had to start with the most important choice first, i.e. deciding for the framework to use. An equal part of functionality existed in either framework ROS and Rock. The final decision was made to use Rock for the core development due to previous experiences with Rock and the following observations: (i) flexible and open meta-build system with support for a variety of Version Control System (VCS) and package build systems, (ii) inherent model-based development workflow, (iii) a clear

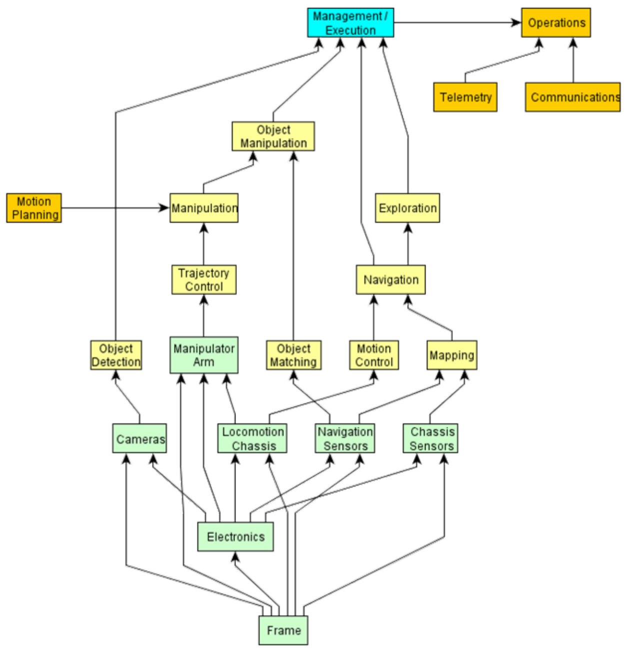

**Figure 2. Artemis' component architecture**

component-model with easily manageable specifications using a domain specific language (DSL), and (iv) close integration with Syskit allowing for runtime management of components.

Since Rock also supports a mix-in of ROS nodes, the final architecture also includes ROS nodes – mainly for manipulation – which have also been managed by the supervision along with Rock components. Using ROS nodes in the supervision is only possible using a belated introduction of specifications for already existing nodes – a significant different workflow compared to the model-based construction of Rock components.

Artemis' set of components covered the following capabilities: localization, mapping, locomotion, navigation, manipulation and object detection. Figure 2 illustrates the general relationships between the components.

Artemis' locomotion and navigational capabilities have been identified as the most critical part for the competition. Locomotion capabilities significantly benefit from a robust hardware design – here a Rocker-Bogie platform with a set of flexible wheels. Each wheel is controlled by an identical set of driver components which feed its status to a component called 'joint dispatcher' which – based on its configuration – allows multiplexing of motion commands and statuses. A motion controller component for Ackermann vehicles serves as the first level of abstraction for controlling the rover. Localization and mapping are based on the application of a Velodyne laser scanner which provides a 360° scan at 10 Hz. The scans are combined with odometry information in a SLAM backend for global map consistency. Navigation of Artemis relies on the localization and mapping results for global planning – applying the D* Lite [10] algorithm – and use input from an additional laser scanner mounted at the front of Artemis to allow for local planning – using VFH* [24].

Further details on the individual hard- and software components can be found in [19] and [14].

### 4.2 Control Architecture

A robotic system has to perform a variety of tasks, yet, has to operate resourcefully. Resources should only be used when needed for the current tasks or if essential for monitoring, e.g., the manipulation capability of Artemis is of no use for navigation. For Artemis this observation led to the design of different mutual exclusive operation modes: exploration, collection of objects, and assembly. Continuously running tasks were simultaneous localization and mapping, and long-range object detection. Navigation had been designed as fundamental capability and was available upon request in either operation mode.

Each operational mode corresponds to a network of components and a corresponding data flow. The supervision evaluates all existing constraints existing for components' configuration and connection, validates the compatibility of a running and to-be-run component network, and eventually runs the minimal sized component network that fulfills the requirements.

The mutual exclusion of operation modes is not only a result of safety and resource optimization, but modes come with contradicting or incompatible constraints. For instance, to collect an object a high resolution point cloud is required to get the exact object location. Thus, the front laser scanner is operated with a very low sweeping speed to maximise the resolution of the point cloud. Meanwhile, local navigation can deal with lower resolution point clouds so that the front laser scanner is operated with much higher speed. This is a good example for the reconfiguration of a component at run-time as explained in Sec. 3.4.

Furthermore, the model-based approach allows to perform offline validation of component networks (and thus of the control architecture) which have been designed in the supervision. Except for manipulation, all components are the result of the model-based workflow and the supervision uses the components specification for this offline validation. In order to validate the data-flow between Rock and ROS' components, a specification has been extracted from existing nodes, mapping ROS topic publishers and subscribers to equivalent Rock input and output ports.

## 5 Results

Artemis successfully participated at the SpaceBot Cup and managed to perform a number of autonomous navigation tasks. No ranking was given for the competition, since none of the participants managed to complete the full mission. The functionality of the system was however validated on various other occasions. The competition illustrated the possibility of applying Rock to design, implement and run a complex robotic system, even under severe time constraints.

Designing high-level control for a mission as given in the SpaceBot Cup is a serious challenge given the time-frame of less than a year for developing hardware and software. Only using an already established framework such as Rock made it possible to tackle all given challenges of the competition.

The component-based approach allowed a reuse of already existing software driver components, e.g., for motor controller, laser scanner and camera. Meanwhile, a number of components had to be continuously improved, e.g., the ones for navigation. The overall Rock workflow made it possible to reliably perform updates with minimal impact, i.e. continuous improvement of components and high-level integration of these single-purpose components could be easily performed. In parallel, complex tasks could be implemented in the supervision considering nominal operation as well as error handling strategies. Rock's workflow provided a strong guidance and support for developers, so that a high level of efficiency could be achieved. The possibility of including ROS nodes into the run-time system also proved to be helpful in balancing the solution in terms of realisability in the given time-frame over model conformance.

Components provide encapsulation on one level as do task networks or rather composition on another. This modularity allows to start development following a top-down approach. In practice and for Artemis, the development turned into a mixture of bottom-up and top-down design in order to maximise the reuse of existing components. The support for distributed development, however, showed some drawbacks. The robot's special capabilities such as object recognition and navigation, and high-level integration could be developed by separate teams from the people building the high-level task in the supervision – designing and orchestrating complex interactions in the supervision requires a special expertise which component developers do not necessarily have. However, this separation comes with a communication gap which needs to be seriously managed to provide a seamless integration path from single-minded components to component networks for performing complex tasks. This experience also showed that Rock can further improve by providing a seamless path from component development to high-level integration.

## 6 Conclusions & Future Work

The demanding requirements of space missions generate a strong requirement for autonomy and complex software in future robotics systems. Software frameworks will likely play an increasingly important role in this context. We have shown that the Rock framework provides the fundamental principles for being able to manage this complexity. It is available as open source[2], and ready for deployment in complex systems with large development teams, as was demonstrated on the Artemis rover. Rock has already been integrated to a number of ground systems, and has a strong support for autonomous underwater systems. While the software is already very usable, a lot of research is still required to improve the integration between the different levels of the system.

## Acknowledgements

The authors would like to thank all team members for their contribution to the development of the Artemis rover. The work presented here has been part of the Project "SpaceBot", which has been funded by the Federal Ministry for Economics and Technology (BMWI) through the German Space Agency (DLR) grant number 50RA1318.

## References

[1] J. M. Badger, S. W. Hart, and J. D. Yamokoski. Towards Autonomous Operation of Robonaut 2. In *AIAA Infotech@Aerospace*, volume 2, 2012.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *IEEE Software Engineering and Formal Methods*, 2006.

[3] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis. Brics - best practice in robotics. In *ISR/ROBOTIK*, 2010.

[4] BRICS. {BRIDE - BRICS Integrated Development Environment}. Retrieved April 24 2014, from http://www.best-of-robotics.org/bride.

[5] I. Crnkovic and M. Larsson, editors. {*Building Reliable Component-Based Software Systems*}. Artech House, Inc., 1rst edition, 2002.

[6] S. Fleury, M. Herrb, and R. Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IEEE/RSJ IROS*, pages 842–848, 1997.

[7] S. Joyeux and J. Albiez. Robot development : from components to systems. In *Control Architecture of Robots*, 2011.

[8] S. Joyeux, F. Kirchner, and S. Lacroix. Managing plans: Integrating deliberation and reactive execution schemes. *Robotics and Autonomous Systems*, 2010.

[9] T. Kaupisch and D. Noelke. DLR SpaceBot Cup 2013 - A Space Robotics Competition. *Künstliche Intelligenz*, 2014.

[10] S. Koenig and M. Likhachev. D*lite. In R. Dechter and R. S. Sutton, editors, *AAAI/IAAI*, pages 476–483. AAAI Press / The MIT Press, 2002.

[11] K.-K. Lau, L. Ling, and P. Velasco Elizondo. Towards composing software components in both design and deployment phases. In H. Schmidt *et al.*, editor, *Proc. 10th Int. Symp. on Component-based Software Engineering, LNCS 4608*, pages 274–282. Springer, 2007.

[12] I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, and S. Wachsmuth. Generic middleware support for coordinating robot software components: The Task-State-Pattern. *Journal of Software Engineering in Robotics*, 2(September):20–39, 2011.

[13] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. *2010 IEEE International Conference on Robotics and Automation*, pages 4627–4632, 2010.

[14] M. Manz, R. Sonsalla, J. Hilljegerdes, C. Oekermann, J. Schwendner, S. Bartsch, and S. Ptacek. Design of a rover for mobile manipulation in uneven terrain in the context of the spacebot cup. In *i-SAIRAS*, 2014.

[15] M. D. McIlroy. Mass-produced software components. In P. Naur and B. Randell, editors, *Proceedings of {NATO} Software Engineering Conference*, pages 138–155. NATO Science Committee, 1968.

[16] E. Prassler, H. Bruyninckx, and K. Nilsson. The Use of Reuse for Designing and Manufacturing Robots, 2009. White Paper.

[17] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Workshop on Open Source Software (Collocated with ICRA 2009)*, 2009.

[18] M. Radestock and S. Eisenbach. Coordination in Evolving Systems. In *Lecture Notes in Computer Science*, volume 1161, pages 162–176. 1996.

[19] J. Schwendner, T. M. Roehr, S. Haase, M. Wirkus, M. Manz, S. Arnold, and J. Machowinski. The Artemis Rover as an Example for Model Based Engineering in Space Robotics. In *ICRA Workshop on Modelling, Estimation, Perception and Control of All Terrain Mobile Robots*. IEEE, 2014.

[20] R. D. Snyder, D. Douglas, and C. Mackenzie. Robustness infrastructure for multi-agent systems. In *OCC*, 2004.

[21] P. Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. Phd, Katholieke Universiteit Leuven, 2006.

[22] P. Soetens and H. Bruyninckx. Realtime hybrid task-based control for robots and machine tools. In *IEEE ICRA*, 2005.

[23] C. Szyperski. *Component software: beyond object-oriented programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[24] I. Ulrich and J. Borenstein. Vfh*: Local obstacle avoidance with look-ahead verification. In *ICRA*, pages 2505–2511. IEEE, 2000.

[2]http://rock-robotics.org