Electronic Communications of the EASST
Volume 60 (2013)

Proceedings of the Seventh International Workshop on Software Quality and Maintainability

-

Bridging the gap between end user expectations, vendors' business prospects, and software engineers' requirements on the ground (SQM 2013)

On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems

Max Reichardt, Tobias Föhst, Karsten Berns

20 pages

Guest Editors: Eric Bouwers, Yijun Yu
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems

Max Reichardt[1], **Tobias Föhst**[1], **Karsten Berns**[1]

[1] Robotics Research Lab
Department of Computer Science
University of Kaiserslautern, Germany
{reichardt, foehst, berns}@cs.uni-kl.de

Abstract: Frameworks have fundamental impact on software quality of robot control systems. We propose systematic framework design aiming at high levels of support for all quality attributes that are relevant in the robotics domain. Design decisions are taken accordingly. We argue that certain areas of design are especially critical, as changing decisions there would likely require rewriting significant parts of the implementation. For these areas, quality-motivated solutions and benefits for actual applications are discussed. We illustrate and evaluate their implementations in our framework FINROC – after briefly introducing it. This includes a highly modular framework core and a well-performing, lock-free, zero-copying communication mechanism. FINROC is being used in complex and also in commercial robotic projects – which evinces that the approaches are suitable for real-world applications.

Keywords: Software Quality, Robotics, Framework Design, Software Maintenance

1 Motivation

Reaching high levels of software quality is in many ways a decisive factor for success when developing robot control systems – especially when systems grow beyond a certain size. With the intent to commercially develop and sell increasingly complex autonomous service robots on emerging (mass) markets, importance of this topic will rise even further. Certification for safety is a central requirement in this context.

Complex robot control software is typically implemented based on a robotic framework, toolkit, or middleware. As these terms overlap, "framework" will be used in the remainder of this document. By defining a component model and dealing with many common problems, the selected framework has fundamental impact on the quality of robot control software: For example, if the framework is portable, efficient, or scalable, robot control software will more likely be too.

In fact, we observe the framework as a central adjustable factor determining software quality – with significant potential to introduce measures that actually guarantee or enforce certain quality requirements. From this point of view, frameworks are an important research topic for making progress in robotics software development. Due to its special nature, characteristics, and constraints (e.g. autonomy in complex dynamic environments), it is worthwhile to investigate this topic specifically for the service robotics domain, as possibilities and difficulties differ from other areas of (embedded) software development.

Table 1: Relevant quality attributes in robotics software

Execution Qualities	**Evolution Qualities**
Performance efficiency	Maintainability
Responsiveness (latency)	Reusability
Safety and Reliability	Portability
Robustness and Adaptability	Flexibility
Recoverability	Extensibility
Scalability	Modularity
Usability and Predictability	Changeability
Functional Correctness	Integrability
Interoperability	Testability

In order to implement quality measures and perform experiments, it is sometimes necessary to have full control over the framework core, architecture, tools, or code repositories. Therefore, using a framework developed and maintained by a third party can be a limiting factor. Thus, we implemented FINROC[1] (see section 4) considering factors altering software quality from the initial design phase in 2008. Nonetheless, as this is in fact a software quality attribute as well, we pay close attention that our efforts are interoperable with other projects in the open source robotics community. The fact that FINROC is being used in complex and also in commercial robotic projects (see section 6) evinces that the presented approaches are actually suitable for real-world applications.

With respect to effort spent on software quality assurance, professional product development and most research groups are two different worlds. In academics, spending time on this task is typically not rewarded – as long as systems run sufficiently robust and efficient. In robotics, however, systems often have considerable complexity. Keeping them maintainable across multiple generations of PhD students is a major challenge. In practice, many systems are abandoned when their developers leave. Due to time constraints in this context, measures to raise software quality are of particular interest if they cause only little extra effort in application development. Again, we see measures and policies implemented in the framework as a promising area to work on.

2 Software Quality in Robotics

Software quality is strongly related to non-functional requirements, as these "characterize software quality and enable software reuse" [BS09]. "Although the term 'non-functional requirement' has been in use for more than 20 years" and there is consensus that they are important, "there is still no consensus in the requirements engineering community what non-functional requirements are" [Gli07]. International standards such as ISO/IEC 9126 and ISO/IEC 25010 structure non-functional requirements in quality characteristics and subcharacteristics. Mari and Eila [ME03] distinguish between "execution qualities" and "evolution qualities". Following this,

[1] http://www.finroc.org

we believe the quality attributes collected in table 1 are especially relevant across a wide range of complex control systems for service robots.

Various publications on robotics software deal with the necessity and difficulties reaching these quality attributes, such as software reuse [VG07, BS09, BDWL11], robustness and reliability [Sma07], scalability [SHRK11], or interoperability and integration [CLR07, SB11].

Clearly, it is challenging for a software developer to consider all these attributes when designing a robotic application. Using a suitable framework can simplify this task significantly. We distinguish three levels of framework support:

- In the ideal case, quality attributes can be ensured "seamlessly". E.g. if the framework provides convenient facilities for efficient, scalable, and robust inter-component data exchange – possibly providing real-time guarantees – the developer does not need to worry and the resulting application will not have deficiencies in this respect. Interoperability is another example.

- If bad code or bad component behavior can be detected automatically, requirements can be enforced by notifying the developer. This way, for instance, memory allocation or locking can be prevented in real-time code. Introducing a total ordering on locks – as shown in [Wil12] – avoids dead-locks.

- There are many other measures to support certain software quality attributes that do not provide any guarantees though. Promoting good software development practices such as separating framework-independent from framework-dependent code will increase reusability and portability of software artifacts.

With FINROC as an object of study and validation, we investigate measures that can be implemented in a framework to support or even guarantee certain quality attributes.

3 Critical Areas of Design

Many important decisions must be taken when designing a framework – often involving trade-offs. Ideally, those decisions are well-founded. This requires a thorough understanding of the available options. Studying existing solutions, we identified alternatives, best practices, and lessons learnt with respect to different areas of design. Their implications on a robot control's quality attributes were evaluated carefully.

Notably, design decisions greatly differ in criticality. In fact, some decisions can easily be changed later should other options seem superior. Others, however, can only be undone with immense development effort, which might not be realistic as this would often require rewriting major parts of a framework. Adding real-time support to a framework that was not designed with this in mind is an example. It is important to be aware of this fact.

Fig. 1 is an attempt to rate features and areas of design with respect to their criticality. Note that these ratings assume that a highly modular framework core is used (see section 3.2): e.g. a *network transport* mechanism is only easily exchangeable if it is clearly separated from the rest of the framework. As long as the framework is implemented in a portable programming language and depends – if at all – on platform-independent libraries, *portability* is not a major

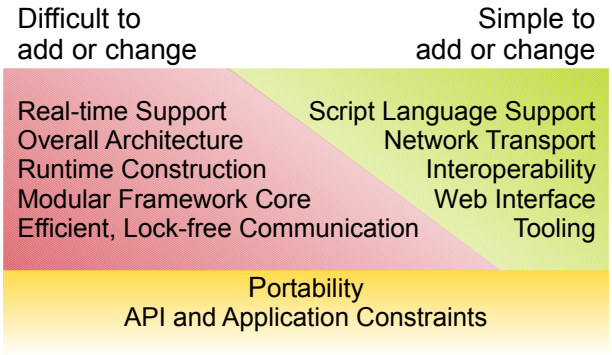


Figure 1: Subjective criticality ratings for areas of design

issue. Furthermore, any binary serialization mechanisms should take care of endianness. The *API* can always be extended with little effort. However, changing it can become laborious once a considerable amount of components and applications exist.

Having identified critical areas of design, we focused on getting those right in the initial FINROC release. As we believe that all the features on the left – *real-time support*, a *modular framework core*, *runtime construction*, and *efficient, lock-free communication* – are beneficial to software quality, they are all supported. As it turned out, it is somewhat challenging to implement those features in combination. FINROC is actually the only framework we are aware of that supports them all. As this is not always obvious, their relevance for the service robotics domain is briefly discussed in the following.

3.1 Efficient, Lock-free Real-time Implementation

Less computational overhead leads to lower latency and lower power consumption of a robot control. More tasks can be executed on a computing node – or smaller, cheaper nodes can be used. Nesnas [Nes07] shares the view that communication overhead is critical: "An application framework must pay particular attention to avoiding unnecessary copying of data when exchanging information among modules".

Due to the modular application style, using a framework will always induce computational overhead compared to a perfectly engineered monolithic solution. However, frameworks such as Orocos [Soe06] show that computational overhead can be low, despite a relatively loose coupling. In practice, as soon as it comes to buffer management or multithreading, we often observe that framework-based solutions actually outperform custom standalone code – sometimes drastically. This is due the fact that efficient, lock-free buffer management is complex to implement. As monolithic implementations are furthermore detrimental with respect to reusability, maintainability, and tool support, they are no alternative to using an efficient framework for realizing complex robot control software.

Efficiency also determines limits on the maximum number of components that are feasible and therewith component granularity. In our research on large behavior-based networks, for instance, we develop systems that consist of far more than thousand components.

Notably, locking can be an even bigger issue with respect to latency and scalability – as we

experienced exchanging data via blackboards in MCA2 [SAG01]. Although based on an efficient shared-memory implementation, locking them exclusively from different components quickly causes significant, varying delays. Such delays in the image processing components of our humanoid robot ROMAN, for instance, hindered natural interaction. Furthermore, overall cycle times are high. Being able to use shorter cycle times in FINROC immediately solved several problems we had with a lateral controller steering an agricultural vehicle. Lock-free implementations are certainly advantageous.

Finally, functionality with real-time requirements can only be realized without completely bypassing the framework (and taking care of multithreading and lock-free data sharing manually) if the framework provides real-time support. While real-time requirements can often be neglected in scientific experiments, this is a central topic for safety-critical parts of commercial robot control systems in order to guarantee that a robot will always react in time to certain events. For our bucket excavator THOR (see section 6) this is certainly critical.

Notably, it is possible to separate the component communication mechanism ("transport") from the rest of the framework to make it exchangeable later – an approach taken in Orocos. This might seem to contradict the criticality rating in fig. 1. However, it does not as the framework must be deliberately designed for this. Apart from that, the common API for all transports can limit efficiency: e.g. if empty buffers are not obtained from the API, unnecessary copying cannot be avoided in multithreaded applications.

3.2 Modular Framework Core

Makarenko *et al.* [MBK07] discuss the many benefits of frameworks having a slim and clearly structured code base – especially regarding development and maintainability of a framework itself. A modular framework core is furthermore beneficial regarding flexibility, extensibility, and changeability of a framework and the resulting robot control systems. Furthermore, portability is increased, as many advanced features such as script language support or a web interface can be made optional. This allows creating slim – possibly single-threaded – configurations of a framework with minimal library dependencies that are suitable for small computing nodes. Generally, this configurability can allow to tailor a framework to applications, computing hardware, and network topology. In the end, this makes the framework suitable for a broader range of systems.

The concept of "stacks" in ROS [QCG+09] goes into this direction. Several frameworks feature an exchangeable network transport layer. The Player Project is an early example [GVH03]. Few are completely transport-independent, such as Orocos [Soe06]. In contrast, as MCA2 was tightly coupled to a custom TCP-based protocol, we encountered limitations with respect to robustness and efficiency that could not easily be resolved – especially in the context of teleoperation over weak wireless connections and the internet. With respect to ratings in fig. 1, a modular framework implementation with a clear separation of concerns is essential in order to classify design areas as simple to change.

3.3 Runtime Construction

The term "runtime construction" refers to the possibility to instantiate, connect, and delete components at application runtime. Some frameworks support this on a process level: processes

containing components can be started and terminated. If performance is not to be sacrificed, however, runtime construction is also relevant for components located inside the same process. Whether or not a framework should support the latter is a controversial topic – due to limited advantages and significantly complicating the framework implementation. We see some use cases for dynamic application structure:

- When operating in smart environments, robots typically need to condense the available sensor information in homogeneous views of the environment. Suitable components to perform such tasks can be created as sensors are encountered.
- Smart [Sma07] proposes graceful degradation in order to increase robustness of systems. This includes dynamic rewiring of components in the case of sensor failure.
- If developers can modify an application while a robot is running – possibly using a graphical tool – effort for recompiling and restarting robot controls during experiments can be reduced significantly.
- It simplifies implementations – e.g. of network transport plugins with a dynamic set of I/Os. Furthermore, it provides the necessary facilities to handle application structure in e.g. XML files instead of source code. This has the advantage that structure changes do not require recompiling and tools for round-trip engineering are simpler to realize.

Thus, runtime construction contributes to adaptability and flexibility of a system.

4 FINROC Implementation and Evaluation

For the past decade, we have been using MCA2 [SAG01] for developing robot controls, and learned to appreciate many of its qualities – such as real-time support, its scalability, and its application style. By distinguishing between sensor and controller data, it is well-suited for application visualization. MCA2 was originally developed at the FZI[2] ("Forschungszentrum Informatik") in Karlsruhe. Over the years, we realized various modifications and enhancements in the MCA2-KL branch[3]. As its kernel is monolithic, difficulties improving several of the areas listed in fig. 1 were encountered. Unable to find an open source framework with the properties we rate critical (see section 3), we decided to implement FINROC. As Makarenko *et al.* [MBK07] argue, developing and maintaining a framework can be feasible.

System decomposition is similar to MCA2: applications are structured in "modules" (components), "groups" (composite components), and "parts" (OS processes). The interfaces of modules, groups, and parts are a set of ports. These can be connected if their types are compatible. A FINROC application consists of a set of interconnected modules and can be visualized as a data flow graph. "Groups" encapsulate sets of modules (or other groups) that fulfil a common task. They structure an application. Modules and groups can be placed in "thread containers" to assign them to operating system threads. Unlike in MCA2, it is also possible to trigger execution by asynchronous events – as Nesnas [Nes07] recommends to support.

[2] http://www.fzi.de/
[3] http://rrlib.cs.uni-kl.de/

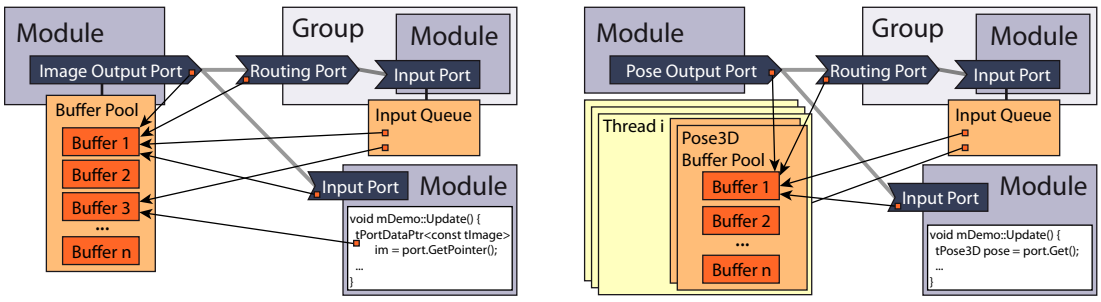
Furthermore, we separate framework-dependent from framework-independent code – as also [MBK07, Roc, QCG+09] encourage. In our experience, this is the best way to make software artifacts portable and reusable across research institutions. In fact, most of our code base are actually framework-independent libraries (called RRLIBs). Over the last decade, we developed a considerable collection of drivers and libraries for robot controls available as RRLIBs. Many of them have already been integrated in FINROC.

There is a C++11 and also a native Java implementation of FINROC. The C++11 version currently depends on the platform-independent boost libraries[4]. We will remove this dependency, as soon as compilers on our systems have sufficient C++11 support. C++11 being the first C++ standard with a multithreading-aware memory model [Wil12], it is preferable to C++03 for safe, lock-free implementations. Notably, lock-free code that is safe on one CPU architecture might not be on another, due to different behavior with respect to memory ordering.

FINROC works well on ARM-based platforms such as Gumstix[5] or the PandaBoard[6]. Furthermore, FINROC's Java implementation compiles on Android, so apps can utilize it to communicate with robots.

4.1 Lock-free, Zero-copy Intra-process Communication

Aiming to maximize (intra-process) communication efficiency, FINROC features a lock-free implementation that does not copy data. It is illustrated in fig. 2a. The implementation supports input queues and allows switching between pushing and pulling data at application runtime. Notably, port connections can be changed without blocking threads that currently publish data via the same ports. Ports allow *n:m* connections – although connecting multiple output ports to one input port is typically discouraged.



(a) Standard implementation

(b) Optimized implementation for cheaply copied types

Figure 2: FINROC's lock-free, zero-copy data exchange

[4] http://www.boost.org/
[5] http://www.gumstix.com/
[6] http://pandaboard.org/

4.1.1 Implementation

The lock-free implementation is based on buffer pools, typically managed by output ports. Apart from the actual data, buffers contain storage for a timestamp and management data such as a reference counter. In order to publish data via an output port, an unused buffer is obtained from the port and filled with the data to be published. If all buffers are in use, another buffer is allocated and added to the pool. For real-time code, pools need to contain sufficient buffers so that this does not occur.

Ports contain atomic pointers, pointing to the ports' current values (symbolized by small orange squares in fig. 2a). Publishing data replaces these pointers and updates reference counters. Obtaining the current buffer from a port is the tricky part: the pointer to the current buffer is read in one atomic operation and the reference counter is increased in another. In a naive implementation things can go wrong if the buffer is returned to the buffer pool (and possibly published again) in between these two operations. Therefore, we use tagged pointers and tagged reference counters. The reference counter may only be increased if it is not zero and if tags match. Otherwise, a new buffer has arrived and the procedure is repeated.

Once published, buffers are immutable (*get* operations on ports return `const` pointers). Almost any C++ type can be used in ports – including types without copy constructor and assignment operator, as well as `std::vectors` of them. Merely, stream operators for binary serialization need to be provided. If the type has no default constructor, a template needs to be specialized so that buffers of this type can be instantiated.

Lock-free buffer pool management and atomics-based reference counting causes some computational overhead. Therefore, FINROC uses an optimized port implementation for small data types of constant size (see fig. 2b). For such types, thread-local buffer pools are used and only the owner thread accesses the reference counter. Again, pointers are tagged in order to avoid the ABA problem.

4.1.2 Benchmarks

In order to evaluate our implementation's impact on computational overhead, a benchmark with uncompressed, high-resolution camera images was set up – as they are quite costly to copy. References are MCA2-KL and Rock/Orocos with a state-of-the-art, intra-process communication model – either locked or lock-free with copying.

A producer-consumer scenario was set up in each of these frameworks[7]: One producer sends HD RGB24 images (1920×1080) at 50 fps to several consumer tasks – filling the image buffers via `memcpy`[8]. Consumers are port-driven and calculate the arriving frames per second. CPU load and memory consumption were determined via `htop`. Only thread-safe communication mechanisms were used.

The results are shown in table 2. Using lock-free communication in Orocos, CPU load and memory consumption grow drastically with an increasing number of consumers. As the authors noted, the lock-free transport is optimized for safe real-time operation and less for high

[7] All benchmarks were performed on an Intel Core i7 @2.67 GHz PC running Ubuntu 12.04, 32-bit

[8] Notably, this is not always necessary in FINROC. Consumers directly receive the buffers obtained from the v4l2 driver, for instance.

Table 2: Results of image transport benchmark

Framework	Consumers	ØFPS	CPU	RAM
Orocos Locked	0	N/A	9 %	35 MiB
	1	50.00	40 %	52 MiB
	7	50.00	40 %	52 MiB
	15	50.00	41 %	52 MiB
Orocos Lock-free	0	N/A	2 %	29 MiB
	1	50.00	11 %	61 MiB
	7	50.00	49 %	202 MiB
	15	49.20	100 %	392 MiB
FINROC	0	N/A	6 %	32 MiB
	1	50.00	6 %	32 MiB
	7	50.00	6 %	33 MiB
	15	50.00	7 %	36 MiB
MCA2-KL	0	N/A	6 %	19 MiB
	1	50.00	6 %	19 MiB
	7	50.00	6 %	19 MiB
	15	50.00	7 %	19 MiB

throughput. In FINROC on the other hand, adding consumers has minimal impact on CPU load or memory usage. Possibly, an Orocos transport plugin based on the mechanism presented here would be feasible. MCA2-KL uses only a single buffer and therefore has the lowest memory footprint. It has, however, issues with blocking (see section 3.1).

To measure the theoretical limits imposed by computational overhead from intra-process communication, five simple modules were connected to a control loop – each module reading and publishing a 4×4-matrix in every cycle. This control cycle can be executed with more than 1 MHz by a single thread that never pauses.

4.2 Highly Modular Framework Core

Targeting a high level of modularity with a clear separation of concerns, we opted for a plugin architecture in FINROC. Furthermore, framework-independent functionality was realized as independent RRLIBs. Fig. 3 illustrates how this currently looks like for a selection of plugins[9].

Functionality that is not needed in every application is generally implemented in optional plugins. By combining plugins with the relevant functionality, FINROC can be tailored to the requirements of an application. Furthermore, functionality that supports developers can be added via plugins during the development process of a system. When software is to be deployed – possibly on small embedded nodes – this functionality can simply be removed.

As motivated by Makarenko *et al.* [MBK07], a central target is keeping the code base of FINROC's core components slim, as less code means lower maintenance effort and fewer errors. Quality assurance can be focused on important and relatively small core components. Experimental enhancements are ideally realized in plugins with no impact on quality of the core components that are used in important projects.

[9] Lines of code were counted using David A. Wheeler's 'SLOCCount'

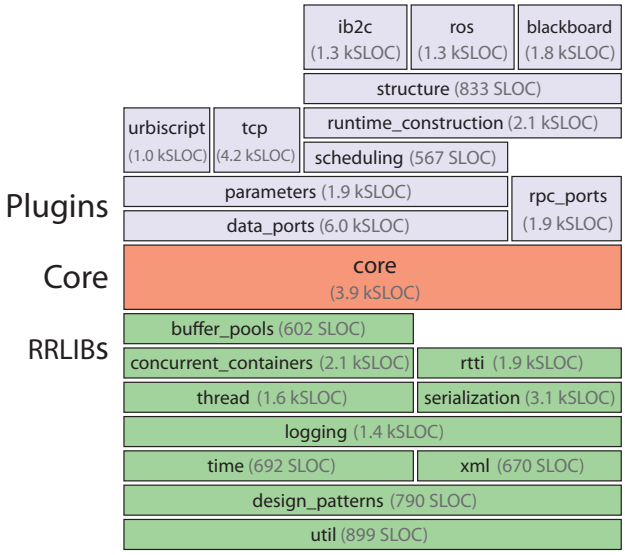


Figure 3: FINROC's modular core with a selection of plugins

As depicted in fig. 3, FINROC consists of many small independent software entities. These typically consist of only a few thousand lines of code. Hence, they can be reimplemented with reasonable effort. As long as interfaces stay the same, each of these entities can be replaced with alternative implementations – possibly optimized for certain hardware, application constraints, or in some way certified. Single-threaded implementations, for instance, would be simpler and more efficient.

The plugins *data_ports*, *rpc_ports*, and *blackboard* provide different mechanisms for component interaction. *data_ports* contains the implementation presented in section 4.1 and is the primary mechanism for data exchange. The RRLIBs *concurrent_containers* and *buffer_pools* comprise lock-free utility classes that are central for its implementation. *structure* is the default FINROC API and provides the base classes for modules, groups, and parts as introduced in section 4, whereas *ib2c* contains the component types for different kinds of behaviors in our iB2C architecture [PLB10]. *urbiscript* adds experimental support for the scripting language from the Urbi framework [Bai07]. *ros* enables interoperability with ROS [QCG+09]. *tcp* provides a slim TCP-based peer-to-peer network transport supporting the publisher/subscriber pattern and featuring simple quality of service. It is currently the default in FINROC. *runtime_construction* contains functionality presented in 4.3.

There are four fundamental classes in the FINROC core (see fig. 4). We tried to come up with a simple structure to which the elements in MCA2, the FINROC API, and possibly other frameworks can be mapped.

The central one is `tFrameworkElement`. It is the base class for all components, ports, and structural entities. Framework elements are arranged in a hierarchy. In our tooling, this hierarchy is typically shown in a tree view on the left (see fig. 5). Then there is `tAbstractPort`. Ports of compatible data types can be connected – the core allows *n*:*m*. Such connections are network-transparent. The root framework element is the `tRuntimeEnvironment` singleton.

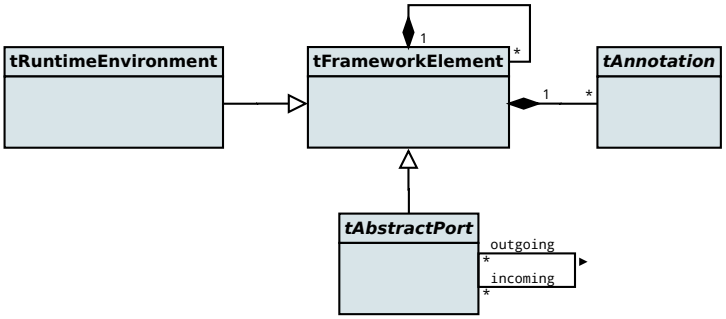


Figure 4: Central classes in core

Several plugins need to attach information to framework elements – such as parameter links to config files or tasks that need to be scheduled. Allowing the attachment of arbitrary annotations appears to be a fortunate design choice with respect to decoupling.

4.3 Runtime Construction

Decisions on support for concurrency significantly influence the effort required to implement runtime construction. In FINROC, two threads cannot make changes to the application structure (component hierarchy and port connections) at the same time. All operations required in control loops, however, execute concurrently so that real-time threads in particular are not blocked performing their tasks. This requires using lists that allow iteration concurrently to modifications in several core classes. Furthermore, it is critical to ensure that no thread accesses a port or component when deleting it.

With these preparations in the core, the *runtime_construction* plugin has three central tasks. First of all, it manages a global list of instantiable component types. By default, FINROC component classes register on application startup providing class name and callback for construction. Applications do not necessarily need to be linked against the shared library files containing the components that are used. They can also be loaded dynamically at runtime using this plugin. Based on these mechanisms, components can be instantiated, connected, and removed graphically at application runtime using the FINSTRUCT tool (see fig. 5). Finally, it allows to store the current application structure together with changes made to it in a simple XML format and to restore it on the next run. Notably, these XML files can easily be generated, interpreted, and changed by external tools as well.

Using FINSTRUCT is optional. Components can also be constructed and connected in source code. This can be more flexible if application structure is to be generated at runtime. As structure created via FINSTRUCT is marked and handled separately by the FINROC runtime environment, using both mechanisms in parallel works well. If the interfaces of components change, it might no longer be possible to instantiate all elements as stored in XML files. In this case, components or edges that have become invalid are discarded and possibly need to be recreated.

Elements are automatically arranged utilizing graphviz[10]. Apart from that, FINSTRUCT is the

[10] http://www.graphviz.org/

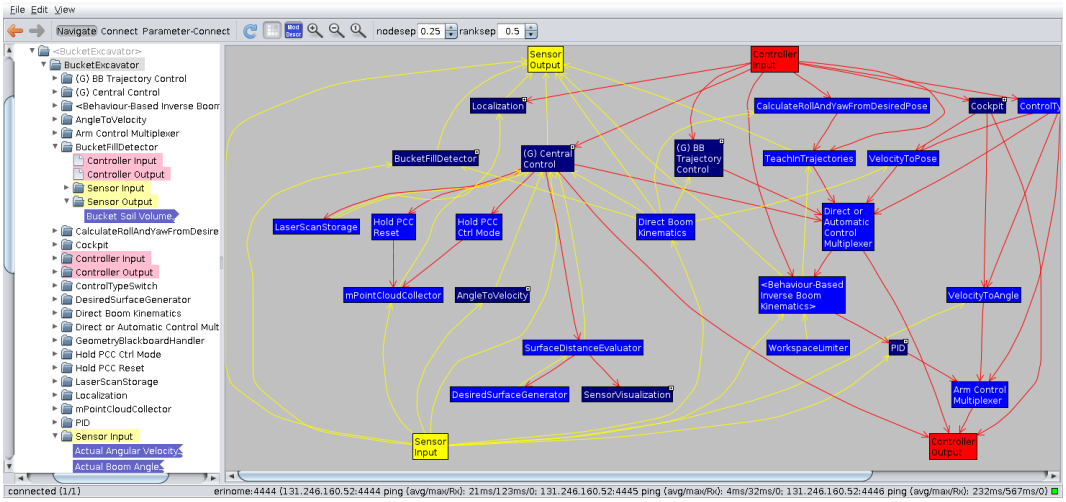


Figure 5: FINSTRUCT: application visualization, inspection, and construction

tool to visualize and inspect applications. It can display and change the current values of all ports and parameters with supported data types. This includes unknown data types which can be serialized to strings or to XML. FINSTRUCT is the counterpart to MCABrowser in MCA2.

5 Policies to Increase Maintainability

Frequent changes are a typical characteristic of software projects in the service robotics domain, and it is important to keep the resulting effort for software maintenance low. With projects growing beyond a certain size, this is certainly (also) critical in university groups with the main developers always leaving after a few years. Framework design and policies have significant influence on this issue. In the following, we briefly discuss some of our experience and decisions we have taken in FINROC in this respect.

5.1 Application Interface and Constraints

An interesting question is whether imposing contraints on application structure is good practice – and also what they should look like. Frameworks such as ROS [QCG+09], Player [GVH03], or Orca [BKM+07] strive to prescribe as little as possible or necessary, as unnecessary contraints can be a nuisance and lead to ugly workarounds. In MCA2, for instance, it is forbidden to connect modules in a way that lead to cycles in data dependencies. This led to inserting loop-back modules to realize such cycles in many applications.

On the other hand, strict guidelines help to avoid chaotic implementations. In our university group, many developers contribute to projects and libraries. Most existing code is from developers no longer working here. Experience shows that not all developers write clean code. Enforcing guidelines contributes significantly to keeping large software systems maintainable. Furthermore, controls of different robots have increased similarities, which facilitates reuse. Apart from

that, guidelines such as separating sensor and controller data in MCA2 allow visualizing applications in a clearer way – compared to using a typical layout algorithm on a "raw" data flow graph.

Since different kinds of APIs and programming styles are suitable for different levels of robot controls – e.g. CLARAty [Nes07] explicitly separates a functional from a decisional layer – we decided to add APIs (e.g. component types and communication mechanisms) via plugins. This way, they are clearly separated from the framework core. Relatively strict constraints and guidelines are enforced in those APIs only, while the core prescribes as little as possible.

Ideally, the relevant API classes can all be mapped on the basic primitives the core provides. This allows using the same tools to interact with application parts based on different APIs – an advantage compared to using unrelated subframeworks.

5.2 Size of Libraries and Components

Having refactored a considerable amount of code from members that are no longer in our group, we experienced that – as a rule of thumb – reusable libraries with up to 5000 SLOC are typically comfortable to maintain and can be understood relatively quickly. So we try to keep all our libraries – including the framework core and its plugins – below this boundary (see 4.2). When a library becomes larger, it is checked whether there is a good way of splitting it up – targeting a low coupling between the new software entities. Sometimes libraries contain relatively independent functionality. In other cases, core functionality and optional extensions can be identified. Dependencies among libraries may only be one-way. Cycles in the dependency graph are are not allowed. With these policies, we hope to support a clear separation of concerns and avoid heavyweight software artifacts as well as feature bloat. This also increases suitability for embedded systems.

5.3 Separating Framework-independent Code

In our experience, reuse of software artifacts across research institutions works best with code that is framework-independent. The OpenCV library [Bra00] is a good example. Developers of some frameworks explicitly encourage separating framework-independent code ([MBK07, Roc, QCG+09]) – and we fully agree.

Over the years, a considerable repository of reusable MCA2 libraries for all kinds of applications evolved. Before porting them to FINROC, we decided to separate the framework-independent code. As it turned out, most of the code actually is – leaving only thin modules that wrap this code for MCA2. Equivalent FINROC modules are even thinner. Notably, using it in an ADTF [Sch07] component for an industrial partner was not a problem either. So this appears to be good practice for migration and for avoiding framework lock-in: If most of the code is independent, migrating existing projects to other frameworks becomes much less of an issue.

5.4 Coping with Variability

A major challenge with respect to reusable software artifacts is handling variability across a broad range of projects (as discussed in [BDWL11]). We try to cope with this issue primarily

using C++11 templates. In our view, this is a very powerful and appropriate mechanism allowing amazing designs – without any additional tooling – and providing strict type-safety along with high flexibility by decomposing type-behavior into policies [Ale01]. On the downside, however, code can be hard to read for developers not familiar with these concepts.

5.5 Code Generation

Custom code generators are sometimes integrated into a framework's build toolchain with the intention to reduce development effort. In our experience, constraints, unforeseen side-effects and reduced transparency when tracking bugs can quickly outweigh any benefits – so adoption must be considered carefully. We deliberately minimized the amount of code generated by the framework to optional string constants for enums and port names. With respect to transparency, we want the complete system behavior to be evident from plain, versioned C++ code that an IDE such as Eclipse can index.

Developing separate versions of the framework in Java and C++11 causes development overhead – an issue with respect to maintenance. Notably, the first versions of FINROC were developed in a subset of Java – automatically generating respective C++ code via a custom Eclipse plugin. This approach actually worked and produced clean, readable code. It was, however, dropped due to limitations. For instance, policy-based design, the RAII idiom, or move-semantics could not really be exploited for the C++11 implementation, whereas the Java implementation was more complicated than necessary. Fixing C++-specific bugs required changing the Java implementation and regenerating code. Today, primary development is performed on the C++11 version. The Java version is typically somewhat behind. As the main tooling is implemented in Java, important parts are adapted immediately though. We attempt to cope with this issue by keeping the implementation slim and relatively stable.

5.6 Stability of Central Software Artifacts

A common issue in software development are changes to software artifacts that many other artifacts depend on – especially when they involve modifications to their interfaces. Projects follow different policies with respect to stability and backward-compatibility. With respect to robotic frameworks, diverse new requirements typically emerge when applications and plugins are developed. Significant refactorings can be the best way to maintain a clean design. However, these can require changes to a lot of components – possibly major effort and a potential source of errors. With most of the component code currently under our control, we perform such changes in FINROC if a positive impact on long-term quality of framework and applications is expected. Adapting existing code causes considerable effort though. With a growing number of projects and users, backward-compatibility will rise in importance – although stable FINROC releases are not affected by such changes.

6 Current Applications

The autonomous mobile bucket excavator THOR (see fig. 6 and 7) is the first major project we ported to FINROC. Apart from that, we are successfully using it in projects on agricultural ma-

Figure 6: The autonomous mobile bucket excavator THOR

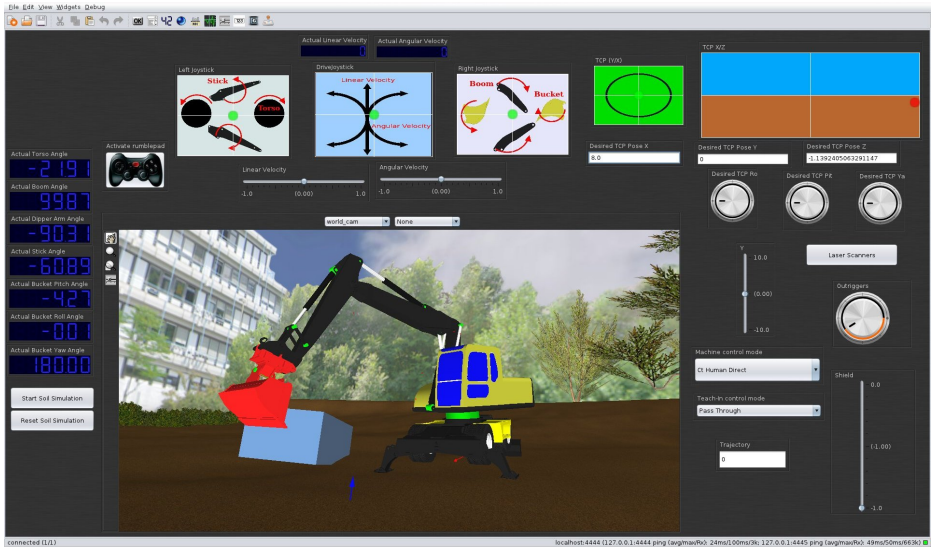


Figure 7: User interface created in FINGUI tool connected to simulated robot

chinery as well as several mobile indoor platforms. Further projects are currently being migrated. As mentioned in section 4.2, there is a FINROC-based implementation of our behavior-based architecture iB2C. For the first time, the rules and guidelines worked out in [PLB10] are properly and automatically checked and enforced. Furthermore, model transformation and model checking approaches were realized [AKRB13].

Robot Makers GmbH[11], use FINROC in their product lineup. This includes the mobile offroad robot VIONA (Vehicle for Intelligent Outdoor NAvigation) – a commercially available robot platform with double-ackermann-kinematics (see fig. 8). Furthermore, they have developed FINROC support for the EtherCAT[12] real-time bus. This facilitates integrating standard components from automation industry.

[11] http://www.robotmakers.de
[12] http://www.ethercat.org/

Figure 8: Robot VIONA developed by Robot Makers GmbH

7 Other Frameworks

Numerous frameworks for robotics exists. However, we are not aware of any solution providing all the features we identified as critical in section 3 in combination.

With a similar component style and support for efficient intra-process communication as well as real-time requirements, OpenRTM-aist [ASK08], OPRoS [JLJ+10], Orocos [Soe06], and the derived Robot Construction Kit (Rock) [Roc] probably come closest. None of them, however, features the same level of modularity in their implementation. Only OpenRTM-aist provides intra-process runtime construction of components. Notably, the RT component model used in OpenRTM-aist has been standardized by the OMG [Obj12].

The "Robot Operating System" (ROS) [QCG+09] is currently the best-known and most widespread solution in academic institutions. Several frameworks are interoperable with it. In this way, ROS has contributed to reusability and integrability of available robotics software. Several frameworks including ROS sacrifice performance for the benefits of an extremely loose coupling: components usually run in separate threads and exchange data exclusively via network sockets. Should the computational overhead be an issue in ROS, it is possible to use frameworks such as Orocos or FINROC inside a ROS node – making them somewhat complementary. Alternatively, ROS itself also has limited support for intra-process communication. This is, however, less sophisticated with respect to buffer management and also components need to be adapted.

Because of the limitations of MCA2, its original developer – the FZI in Karlsruhe – started working on MCA3 [UGO+11].

The many other robotic frameworks include Urbi [Bai07], Microsoft Robotics Developer Studio[13], CLARAty [Nes07], or YARP [FMN08]. Urbi features urbiscript – an integrated scripting language tailored to the requirements in robotics. Similar to ROS, it does not provide support for hard real-time applications. Both Urbi and YARP feature efficient intra-process communication. YARP focuses on middleware aspects and particularly targets to ease interoperability with software artifacts based on other frameworks and technology. Microsoft Robotics Developer Studio

[13] http://www.microsoft.com/robotics/

is a robotics development environment including a visual programming language and a powerful simulator. Limited to Windows platforms and featuring an architecture similar to web services, its popularity in the robotics community is, however, moderate. CLARAty is an efficient framework whose developing institutions include the NASA Jet Propulsion Laboratory. It is explicitly divided into a functional and a decisional layer. The latter is programmed in LISP. Judging from the website, the constricted public efforts have been discontinued – with latest entries and releases from 2008.

aRDx features a particularly efficient inter-process communication mechanism [HB13]. Notably, the implementation is based on ring buffers which is simpler and somewhat less flexible than reference counted buffers, as presented in section 4.1. Major parts of framework and applications are implemented in the scripting language Racket[14] which is especially suitable for integrating domain-specific languages [Bäu13]. aRDx is currently not available to the public.

ADTF [Sch07] is a solution with somewhat similar concepts used in the automotive industry. In this domain, AUTOSAR (AUTomotive Open System ARchitecture)[15] is an emerging architecture for the many control systems in vehicles sold on the mass market. It therefore needs to be slim and provide the high quality standards necessary for safety-critical applications. Its complexity, however, is an issue that currently hinders adoption.

Model-driven software development provides possibilities to increase various quality attributes of robot control systems. Among others, Schlegel et al. [SSL12] elaborately propose adopting respective approaches in robotics, as they have done in SmartSoft.

8 Conclusion and Outlook

In this paper, we discuss the impact of a framework on software quality of robot control systems and propose systematic framework design aiming at high levels of support for all relevant quality attributes. In the scope of this document, we limit discussions to areas we identified as especially critical for initial design. Further areas are investigated in [RFB13] – involving a detailed discussion on the state of the art. The sections on the approaches implemented in FINROC show how solutions for these areas can look like. Applications in research and industry indicate that the presented concepts work well in practice. However, it should be noted that these are not necessarily the best solutions. Over the years, considerable effort was spent on reengineering and modularizing the framework. We hope to convey valuable experience gained during this process.

In its current state, we believe that FINROC provides the necessary means to conveniently create efficient, complex robot control systems. Being interoperable with ROS, our behavior-based architecture, for instance, may also be used inside ROS nodes.

Future activities will include supporting further standards and evaluating third-party component models for possible intergration. In order to increase portability and maintainability, we target reducing implementation size even further while preserving the same set of features.

Regarding measures to support software quality, the ones implemented are only the very beginning. There is certainly a lot more potential. Identifying, implementing, and evaluating suitable such measures should be subject of future research.

[14] http://racket-lang.org
[15] http://www.autosar.org/

Notably, relevance is not limited to robotics. In a very different domain, CAST Research recently concluded that there is a significant correlation between Java EE frameworks and application quality [SSC13]. Furthermore, "frameworks do in fact help develop applications of predictable quality". Regarding the presented approaches, a modular framework has benefits in virtually any domain. With its small core, the Eclipse IDE was a prominent archetype when reasoning about FINROC's design. The efficient realization of data flow ports together with runtime construction, on the other hand, is more specific to concurrent, data-flow-oriented applications that require a certain level of flexibility.

Bibliography

[AKRB13] C. Armbrust, L. Kiekbusch, T. Ropertz, K. Berns. Tool-Assisted Verification of Behaviour Networks. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*. Karlsruhe, Germany, May 6-10 2013.

[Ale01] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[ASK08] N. Ando, T. Suehiro, T. Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In Carpin et al. (eds.), *Simulation, Modeling, and Programming for Autonomous Robots*. Lecture Notes in Computer Science 5325, pp. 87–98. Springer Berlin / Heidelberg, 2008.

[Bai07] J.-C. Baillie. Design Principles for a Universal Robotic Software Platform and Application to URBI. In *2nd National Workshop on Control Architectures of Robots (CAR'07)*. Pp. 150–155. Paris, France, May 31-June 1 2007.

[Bäu13] B. Bäuml. One for (Almost) All: Using a Modern Programmable Programming Language in Robotics. In Brugali (ed.), *Proceedings of the eighth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII), in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany, May 2013.

[BDWL11] C. R. Baker, J. M. Dolan, S. Wang, B. B. Litkouhi. Toward Adaptation and Reuse of Advanced Robotic Software. In *IEEE International Conference on Robotics and Automation (ICRA)*. Pp. 6071–6077. Shanghai, China, May 9-13 2011.

[BKM+07] A. Brooks, T. Kaupp, A. Makarenko, S. B. Williams, A. Orebäck. Orca: A Component Model and Repository. In [Bru07].

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobbs Journal of Software Tools* 25(11):120, 122–125, nov 2000.

[Bru07] D. Brugali (ed.). *Software Engineering for Experimental Robotics*. Springer Tracts in Advanced Robotics 30. Springer - Verlag, Berlin / Heidelberg, April 2007.

[BS09] D. Brugali, P. Scandurra. Component-based Robotic Engineering Part I: Reusable building blocks. *Robotics Automation Magazine, IEEE* 16(4):84–96, December 2009.

[CLR07] C. Cote, D. Letourneau, C. Ra. Using MARIE for Mobile Robot Component Development and Integration. In [Bru07].

[FMN08] P. Fitzpatrick, G. Metta, L. Natale. Towards long-lived robot genes. *Robotics and Autonomous Systems* 56(1):29–45, January 2008.

[Gli07] M. Glinz. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference. RE '07*. Pp. 21–26. New Delhi, India, October 15-19 2007.

[GVH03] B. P. Gerkey, R. T. Vaughan, A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *11th International Conference on Advanced Robotics (ICAR 2003)*. Coimbra, Portugal, June 30 - July 3 2003.

[HB13] T. Hammer, B. Bäuml. Raw Performance of Robotic Software Middleware: A Comparison and aRDx's New Realtime Communication Layer. In Brugali (ed.), *Proceedings of the eighth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII), in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany, May 2013.

[JLJ+10] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, C.-H. Lee. OPRoS: A New Component-Based Robot Software Platform. *ETRI Journal* 32:646–656, 2010.

[MBK07] A. Makarenko, A. Brooks, T. Kaupp. On the Benefits of Making Robotic Software Frameworks Thin. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007)*. San Diego, California, USA, October 29-November 2 2007.

[ME03] M. Mari, N. Eila. The Impact of Maintainability on Component-based Software Systems. In *Proceedings of the 29th Conference on EUROMICRO*. EUROMICRO '03, pp. 25–32. IEEE Computer Society, Washington, DC, USA, 2003.

[Nes07] I. A. Nesnas. The CLARAty Project: Coping with Hardware and Software Heterogeneity. In [Bru07].

[Obj12] Object Management Group, Inc. Robotic Technology Component (RTC) – Version 1.1. Framingham, Massachusetts, USA, September 2012.

[PLB10] M. Proetzsch, T. Luksch, K. Berns. Development of Complex Robotic Systems Using the Behavior-Based Control Architecture iB2C. *Robotics and Autonomous Systems* 58(1):46–67, January 2010. doi:10.1016/j.robot.2009.07.027.

[QCG+09] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng. ROS: an open-source Robot Operating System. In *Proceedings of the Workshop on Open Source Software in Robotics, in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*. Kobe, Japan, May 12-17 2009.

[RFB13] M. Reichardt, T. Föhst, K. Berns. Design Principles in Robot Control Frameworks. In *Informatik 2013*. Lecture Notes in Informatics (LNI). Springer, Koblenz, Germany, September 16-20 2013.

[Roc] The Robot Construction Kit. http://rock-robotics.org/.

[SAG01] K.-U. Scholl, J. Albiez, B. Gassmann. MCA- An Expandable Modular Controller Architecture. In *3rd Real-Time Linux Workshop*. Milano, Italy, 2001.

[SB11] R. Smits, H. Bruyninckx. Composition of complex robot applications via data flow integration. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. Pp. 5576–5580. May 2011.

[Sch07] R. Schabenberger. ADTF: Framework for Driver Assistance and Safety Systems. In *International Congress of Electronics in Motor Vehicles*. Baden-Baden, Germany, 2007.

[SHRK11] A. Shakhimardanov, N. Hochgeschwender, M. Reckhaus, G. K. Kraetzschmar. Analysis of software connectors in robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Pp. 1030–1035. San Francisco, CA, USA, September 25-30 2011.

[Sma07] W. D. Smart. Writing Code in the Field: Implications for Robot Software Development. In [Bru07].

[Soe06] P. Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006.

[SSC13] J. Sappidi, A. Szynkarski, B. Curtis. CRASH Special Report – Impact of Java EE Frameworks on the Structural Quality of Applications. CAST Research Labs, April 2013. http://www.castsoftware.com/research-labs/crash-reports.

[SSL12] C. Schlegel, A. Steck, A. Lotz. *Robotic Systems - Applications, Control and Programming*. Chapter 23. Robotic Software Systems: From Code-Driven to Model-Driven Software Development. InTech, http://www.intechopen.com/books/robotic-systems-applications-control-and-programming/robotic-software-systems-from-code-driven-to-model-driven-software-development, 2012. ISBN: 978-953-307-941-7, InTech, DOI: 10.5772/25896.

[UGO+11] K. Uhl, M. Göller, J. Oberländer, L. Pfotzer, A. Rönnau, R. Dillmann. Ein Software-Framework für modulare, rekonfigurierbare Satelliten. In *60. Deutscher Luft- und Raumfahrtkongress 2011*. Bremen, Germany, September 27-29 2011.

[VG07] R. Vaughan, B. Gerkey. Reusable Robot Software and the Player/Stage Project. In [Bru07].

[Wil12] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning Publications, Shelter Island, NY, USA, February 2012.