Journal of Software Engineering for Robotics 8(1), December 2017, 151-163
ISSN: 2035-3928

One Fits More – On Highly Modular Quality-Driven Design of Robotic Frameworks and Middleware

Max REICHARDT[1,2] Steffen SCHÜTZ[2] Karsten BERNS[2]

[1] Robot Makers GmbH, Merkurstraße 45, 67663 Kaiserslautern, Germany
[2] Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, 67663 Kaiserslautern, Germany

Abstract—Robotics software systems have a large and domain-specific range of quality requirements that make development of reusable software a particular challenge. Frameworks and middleware have a major impact in this respect – on both quality characteristics and development effort. As framework design involves many tradeoffs, they have different quality and feature profiles – with no existing solution clearly superior. Analyzing existing approaches, the principle of customizable quality tradeoffs is identified. The proposed design approach aims at maximizing the principles of concern separation and customizable quality tradeoffs in frameworks: basically decomposing them into one (optional) module per concern. This allows localizing quality requirements and flexibly tailoring frameworks to application requirements. In particular, all operating-system-independent concerns can be run "bare metal". The proposed concept was implemented in the FINROC framework. The benefits with respect to quality characteristics are evaluated in a case study: the framework is run "bare metal" on an FPGA soft core – notably a platform not originally targeted. In addition, it is shown how knowledge on framework design is modeled.

Index Terms—Middleware Infrastructures, Modularity, Robotics Middleware, Software Architectures, Software Quality.

1 MOTIVATION

Due to the large and domain-specific range of quality requirements encountered across diverse robotic applications, development of reusable software artifacts in robotics is particularly challenging. Among the numerous relevant quality characteristics are e.g. performance efficiency, timing determinism, runtime modifiability, simplicity and conceptual integrity, portability, concern separation, stability and longevity, standard-compliance, and testability. Their relevance is application-specific.

Most quality characteristics are strongly interrelated to others and many software design decisions are tradeoffs between them. Quality characteristics are typically arranged in quality models. They should be domain-specific, as e.g. [1] states. ISO 25010 contains a well-known quality model to be tailored to different domains and applications.

Considering all relevant quality characteristics when developing complex robotic applications from scratch is extremely challenging – if not unrealistic. Therefore, applications are typically based on robotic frameworks, middleware, and toolkits[1]. They significantly reduce the complexity of application development. Universal frameworks handle many concerns that are relevant across a broader range of applications – typically including e.g. interface definition, network communication, scheduling, and tooling. Frameworks have a major impact on software quality. They can support or even guarantee certain quality concerns such as timing determinism, interoperability, or portability – e.g. if and how easily robotics software can be ported to small embedded computing nodes. Furthermore, the framework is the basis of an application's architecture – and the architecture has a fundamental impact on software quality [2].

Hence, research on frameworks is a key topic for improving quality and productivity in robotics software development.

2 RELATED WORK

Numerous Robotic frameworks have been proposed and developed. Although ROS [3] has been adopted by a significant number of research institutions, we believe that key statements of Makarenko et al. [4] are still valid:

1) "none of the existing solutions is clearly superior"
2) the difficulty in making comparisons "leads to a comparison between 'apples and oranges"'

1. For simplicity, the term *framework* is used for all three in the following.

Regular paper – *Manuscript received July 15, 2017; revised November 20, 2017. Digital Object Identifier: 10.6092/JOSER_2017_08_01_p151*

3) "a one-size-fits-all solution to building robotic systems may be unachievable and undesirable"

Regarding the first statement, many framework design decisions are tradeoffs between different quality attributes. They cannot be considered as generally "better" or "worse". The decision of whether to use an IDL is an interesting example. As a result, frameworks have different feature and quality profiles that make them more, less, or not suitable for specific applications. Notably, feature bloat is a delicate pitfall in this context – detrimental to many quality characteristics such as maintainability, portability, and ease of use.

Regarding the second statement, it is challenging to measure quality characteristics. Furthermore, there are many potentially relevant characteristics and they typically have multiple dimensions. Thus, most comparisons primarily focus on features – e.g. [5], [6]. There are also quantitative comparisons on a limited set of quality characteristics – most commonly performance efficiency, and latency (see e.g. [7]).

Regarding the third statement, we believe that this is an inevitable consequence of many design decisions being (quality) tradeoffs. In this work, however, we propose an approach for "one fits more" solutions. Minimizing the tradeoffs between specific pairs of relevant quality attributes is a related important area of research and design.

As mentioned, ROS is currently the most well-known and wide-spread solution. Its authors, however, identified significant shortcomings that motivated the development of ROS 2.0[2]. These include unsuitability for real-time systems and small embedded computing nodes – possibly bare metal. Notably, these challenges are addressed in this work.

The AUTOSAR standard from automotive industry is in several ways similar to a (model-driven) robotics framework and is occasionally also used for robotic applications. Instead of an implementation, it is an extensive specification that allows for competing implementations. Notably, its design is also motivated by quality characteristics such as modularity, scalability, and reusability[3]. Until recently, significant constraints (e.g. no HEAP allocation) for small computing nodes and timing determinism, however, also limited quality characteristics (such as flexibility) required on higher levels of autonomous systems. Therefore, the *AUTOSAR Adaptive Platform* was added in 2017. It is a new additional standard based on POSIX operating systems and targets more powerful computing nodes. Combining two frameworks (or layers) with different quality characteristics for low- and high-level robot control is not uncommon (see e.g. NASA's CLARAty Framework [8]). Interoperability between those layers and a small gap in the development process is critical with respect to development effort and maintainability of resulting systems.

The Player Project [9] is a well-known discontinued framework featuring an exchangeable network transport layer. It allows using different network transports – with different quality attributes – depending on application requirements. Apart from the default custom TCP implementation, there is e.g. an implementation based on the JINI standard. This way, Player's networking supports a broader spectrum of potential quality requirements – and therewith applications. We call such variability the **principle of customizable quality tradeoffs**.

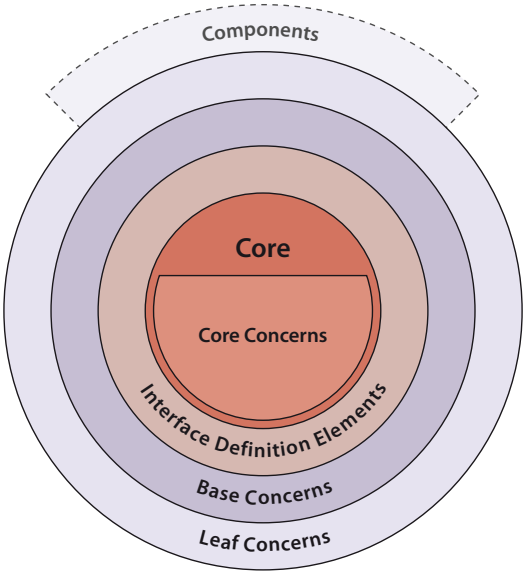
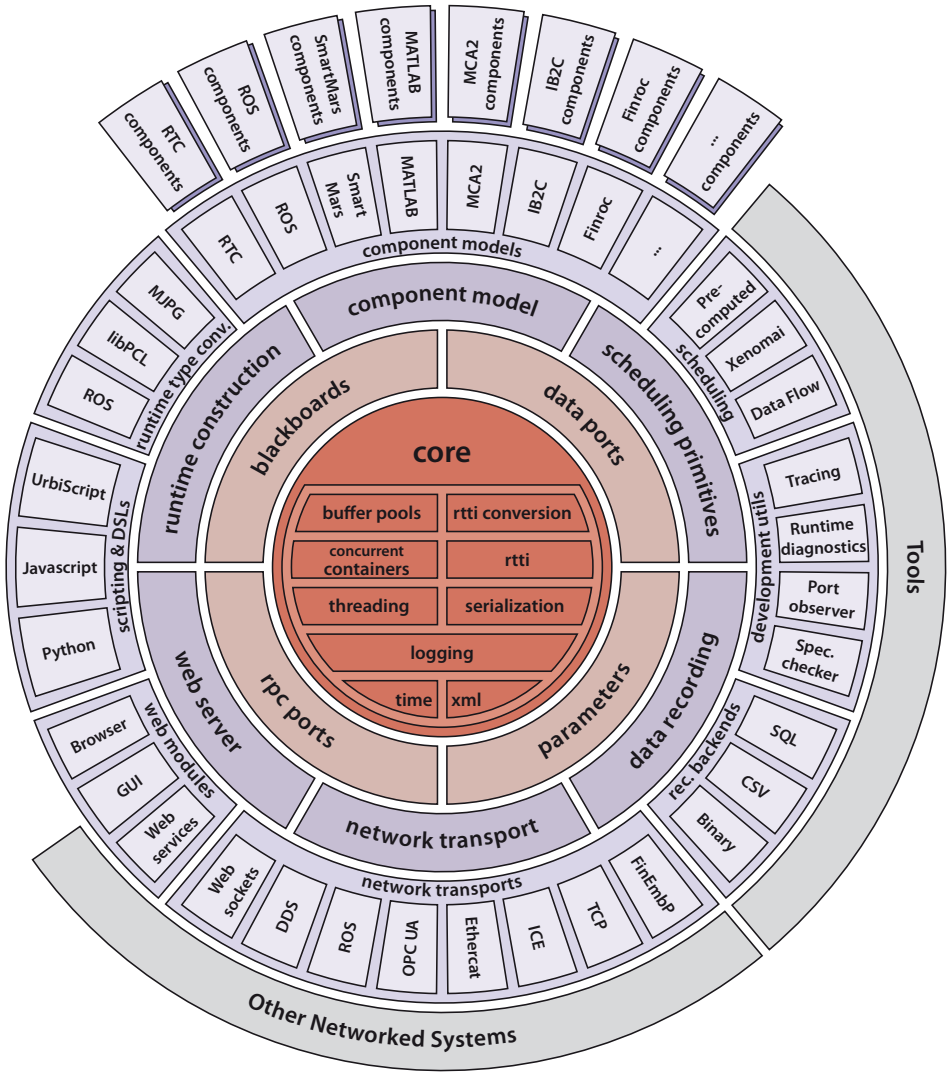


Fig. 1: Layered top-level decomposition of a robotic framework motivated by the *core/periphery* pattern.

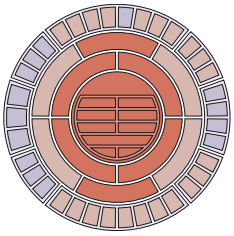
Somewhat similar, Orocos [10] allows using transports with different quality profiles also for intra-process communication (e.g. one for real-time support, one for high throughput of large buffers). It decouples these transports from interface definitions – an example for **separation of concerns**, a fundamental design principle that is beneficial with respect to many quality characteristics. Opros [6] and GenoM3 [11] also feature *transport-independence*, which is itself considered a relevant quality characteristic of robotics frameworks. Fitzpatrick et al. [12] discuss advantages of transport-indenpence in YARP with respect to stability and longevity in particular.

Furthermore, there are model-driven approaches such as the OMG *Robotic Technology Component* (RTC) standard[4] in robotics. They feature platform-independent software entities that are transformed to platform-specific artifacts. The target platforms can have very different quality characteristics – and may include small embedded nodes (see e.g. [13]). This also allows for customizable quality tradeoffs. Whether to use a model-driven approach is notably also a quality tradeoff. Therefore, research on both model-driven and non-model-driven approaches is considered important – and also

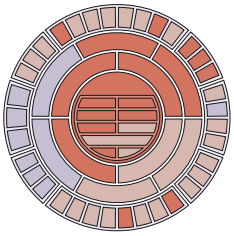
2. http://design.ros2.org/articles/why_ros2.html
3. http://www.autosar.org/about/
4. http://www.omg.org/spec/RTC/



(a) The proposed vision of a highly modular framework: The blocks are an example of how the different layers could be decomposed. Further elements can be added on all layers later on.



(b) Example "heat map" for many evolution qualities



(c) Example "heat map" for performance efficiency

Fig. 2: Highly modular framework design and localized quality requirements

complementary, as model-driven toolchains are developed that support non-model-driven solutions such as ROS (e.g. [11], [14]). A non-model-driven approach was chosen for the proof-of-concept presented in Section 5 – though highly modular designs should be applicable in Model-Driven Software Development (MDSD) also.

3 Highly Modular Framework Design

The presented research aims at applying the principles of concern separation and customizable quality tradeoffs conducted in earlier work more extensively – and to analyze the result with respect to its feasibility and its implications on quality characteristics. The modular approach is further motivated by common critical tradeoffs such as feature completeness vs. feature bloat – or stability and maturity vs. rapid development of new framework features (framework evolution). Additionally, it aims at localizing quality characteristics to preferably few software entities in order to cope with design complexity – as well localization of change in general, for improved modifiability and maintainability.

Motivated by the implied quality characteristics, major elements from the *microkernel pattern* [15] and the *core/periphery pattern* [2] are adopted. In this regard, Bass et al. emphasize that "the core must be small" [2] as well as "The core needs to be highly modular, and it provides the foundation for the achievement of quality attributes". Due to the large community, ROS can be considered an example of the *core/periphery pattern* in robotics.

On the highest level, we decompose frameworks into layers: a slim *Core* with various *Core Concerns*, *Interface Definition Elements*, *Base Concerns*, *Leaf Concerns*, and repositories of *Components* (see Fig. 1). Dependencies only exist from the outside to the inside and to other elements in the same layer – but not cyclic. The architecture is not *strictly layered*.

Typical *Base Concerns* of frameworks include component model(s) (application decomposition), scheduling and dispatching (runtime model), network communication, and runtime modifiability. *Leaf Concerns* can either specialize the abstract *Base Concerns* (e.g. different network transports) or provide additional functionality based on them.

Reasons for choosing this layered top-level decomposition include:

- The adopted patterns are already layered to some extent.
- A simple and clear decomposition structure is considered important – for both communicating it and to avoid architecture degeneration. With this target, the layers were derived from analyzing dependencies between framework concerns developed and identified in both related and less structured earlier work (e.g. [16]).
- Clear rules on unidirectional dependencies are beneficial for maintainability – a lesson also learnt from early work.

Notably, some solutions – as e.g. Orca 2 [4] or OpenRTM-aist [17] – rely on generic third-party middleware packages for interface definition and transport. This reduces development and maintenance effort. The quality characteristics of these middleware packages are, however, also adopted.

Fig. 2a illustrates a more fine-grained framework decomposition on this basis. This is the concept and vision of what is pursued in our work on frameworks: All framework concerns are separate software entities – possibly subdivided into sub-concerns. All blocks outside the core are optional. *Core Concerns* not portable to all desired target platforms must also be optional. Blocks may have multiple implementations, possibly platform-specific and with different quality profiles. By choosing a suitable fragment from the set of available blocks, the framework can be tailored to application requirements and even requirements in different phases of projects. Furthermore, the framework scope is flexible. Used without base and leaf concerns except of network transports, for instance, the framework is a plain communication middleware.

Every block has its own quality profile – and quality requirements can be localized to these blocks. For example, efficiency and real-time requirements are primarily important for transports and scheduling. This can be visualized as a heat map – as done in Fig. 2c. It displays the same blocks as Fig. 2a – without the outer ring containing components, *Tools*, and *Other Networked Systems*. Red indicates that a block is potentially of high relevance for the respective quality characteristic of the resulting system. Blue indicates low relevance. Timing determinism is an interesting example with respect to localization: if timing-critical functionality required from other blocks is time-bounded (e.g. lock-free), an application's timing requirements can be localized to possibly platform-specific scheduling blocks. These are instantiated and configured by the application (it may have a direct dependency to e.g. *Xenomai*). When required, the scheduling concern can be replaced without modifying the rest of the application. Several quality characteristics – such as stability, maturity, maintainability, portability, and longevity – share the simple heat map shown in Fig. 2b that is somewhat aligned to the layers.

The modular concept is also beneficial for portability – with operating-system-independence being a particularly interesting quality characteristic for single blocks: combinations of blocks which are all operating-system-independent can be used bare metal.

As common for modular designs, module interfaces play a key role with respect to overall system quality. Thus, the choice of interface definition technique is of major importance – and also a quality tradeoff. The proposed concept is not limited to specific interface definition techniques. In the proof-of-concept presented in Section 5, plain C++ headers are used – state of the art in open source C/C++ software development (e.g. libraries). All blocks are compiled to separate shared libraries. Apart from that, higher-level (outer) blocks may use elements from the *Interface Definition Layer* – e.g. for uniform configuration.

Supporting multiple component models to increase software reuse and integrability is another target. Component model implementations use common features from other layers – such as typically ports, connectors, and an abstract component class. Particularly these elements are found in many component models including the ones from the UML MARTE and AUTOSAR standards – or the RTC, and BRICS component models ([18], [14]) from robotics. Notably, concerns and tools operating on data structures from inner framework layers, can present and handle components from different models in a unified way.

The concept furthermore targets dynamic loading of blocks – allowing to add crosscutting concerns without recompilation and also at runtime. Use cases include concerns such as scripting, interoperability, data recording, or facilities for development and debugging. Notably, this enables dynamic reconfiguration also on a framework level – increasing the overall flexibility and adaptability. In other words, it enables runtime modification of a framework's quality profile and features.

4 Modeling Design Knowledge

Following a systematic approach when designing a robotic framework is a complex and challenging task. There are numerous relations between quality characteristics, design principles, framework concerns, requirements, and many other factors relevant for design.

Designers face many questions that are difficult to answer systematically – such as:

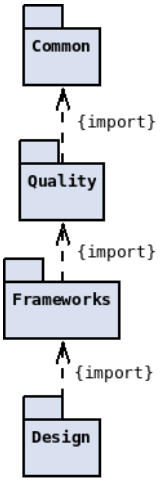


Fig. 3: Packages of the domain meta model

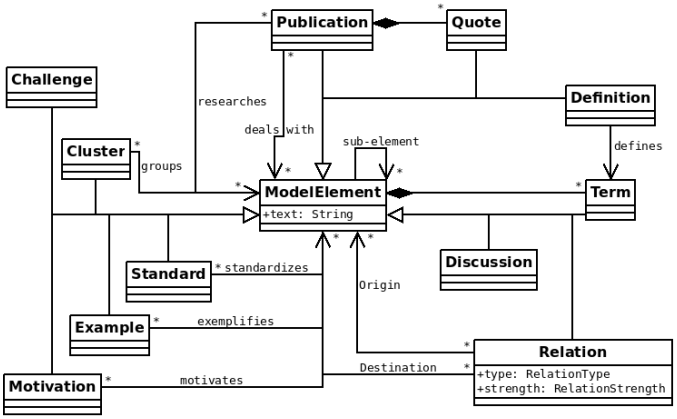


Fig. 4: The *Common* modeling package

- Quality Characteristics: What are relevant quality characteristics of robotics software? How are they related? How can they be measured?
- Design: Which design principles[5] have been proposed? Which quality characteristics do they influence? To which framework concerns have they been applied?
- Framework scope and concerns: Which concerns do frameworks usually cover? Which other concerns have been proposed and implemented?
- Application Requirements: What are typical application requirements for robotics frameworks? Which importance do quality characteristics have for different kinds of users and stakeholders?
- On specific framework concerns: What are design alternatives? Which quality characteristics do they influence? What are typical requirements? What are mandatory requirements if certain system properties are desired? Which quality characteristics are most relevant?

Qualified answers to these questions require a lot of knowledge in robotics software engineering. In many cases it would be difficult and elaborate to give *complete* answers, due to the numerous design factors and their relations. For unsystematically obtained answers, it can usually be questioned why specific elements have been mentioned – while others have been omitted. Nevertheless, such questions need to be answered in a systematic framework design approach.

As a good starting point, some authors collect and promote design principles (or "best practices") for robotics software design (e.g. [19]). Due to limitations in scope and space, publications can only give partial answers to many of the questions listed above. It is furthermore challenging to cover all relevant design factors and their relations in a (linear) text without extensive cross-referencing[6].

We believe that modeling the body of knowledge is required to some extent in order to cope with these challenges. It is furthermore a step towards more systematic reasoning on design alternatives. The result could serve as a kind of *Framework Design Catalog* or *Overview*.

We therefore developed a meta model for quality-driven framework design and used it to model a considerable amount of domain knowledge. The meta model is divided into four levels with different concerns – as shown in Fig. 3.

The *Common* package illustrated in Fig. 4 provides abstract common elements such as relations, terms, definitions, or publications. Being this generic, it could notably also serve as basis for modeling other domains. It was created with the following targets and considerations:

- It should be possible to express common relations and their properties formally in the model (e.g. "in conflict with", "sub-element of"). These common relations include the named relations in the class diagrams. As there are, however, many other kinds of relevant relations, simple associations between elements are also allowed. Such associations are processed if the model is e.g. queried for all elements that are related to *runtime modifiability* over two levels of relations. Thus, the model can be classified as a semantic network.
- It is possible to create relations to relations. For instance, two quality characteristics may typically be in conflict. Design Principles to minimize this conflict need to reference this relation.
- Authors may have different views on certain topics. It should be possible to model this. Therefore, publications (and optionally quotes) can be linked to model elements. This e.g. makes it possible to query who has written about

5. In the following, "design principle" is used as a generic term for "design approach", "design method", "design policy", "design pattern", "design philosophy", "design (best) practice", "architectural tactic" etc.

6. These experiences were actually made in initial attempts to answer some of the above questions in the scope of a PhD thesis. Towards increasing levels of detail and completeness, discussing these topics became increasingly difficult to structure, verbose, and hard to read.

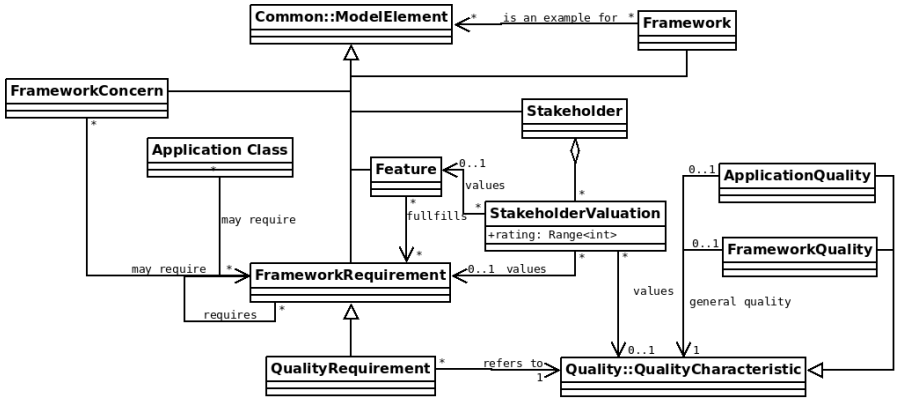


Fig. 5: The *Frameworks* modeling package

certain model elements.

- Instances may inherit from multiple meta model elements. For instance, the ISO-25010 quality model can be *QualityModel* and *Standard* at the same time. *Challenge*s are often also a *Motivation* for other elements.
- Clusters can be used to group model elements. They can be useful for defining relations that are valid for all of the elements.

Fig. 6 shows the *Quality* package to model quality characteristics and their relations – as well as quality metrics, and quality models. As different quality models arrange and name quality characteristics differently (e.g. "changeability" vs. "modifiability"), their hierarchy in each quality model is modeled separately using the *QualityModelNode* class.

The *Framework* package (see Fig. 5) deals with framework concerns and requirements. It distinguishes between quality characteristics of robotic frameworks and applications. For this purpose, there is an additional framework-specific and an additional application-specific instance for each general quality characteristic. When this distinction is relevant, relations reference these. For instance, application requirements already provided by the framework (e.g. interoperability) contribute to the application's simplicity – but not to the framework's simplicity. Also, specific quality characteristics of frameworks and applications are not necessarily aligned. While a portable application typically requires a portable framework, usability of an application is much less related to the usability of a framework.

Framework concerns are arranged hierarchically. When designing a framework, this hierarchy can be used as a guideline on its structural decomposition. Notably, such modeling has contributed to reaching the structural decomposition in Figs. 1 and 2a. Framework requirements, features, and their typical value for groups of stakeholders are also modeled. A *StakeholderValuation* rates an element with respect to one stakeholder group. The stakeholder groups can e.g. be researchers, end users, and product developers of small or large series.

Finally, the *Design* package displayed in Fig. 7 is used to model design principles and design alternatives. Design alternatives typically relate to specific framework requirements or concerns – and they may comprise applying the more generic design principles. The tradeoffs involved with choosing certain design principles or design alternatives are modeled with *DesignImplications*s. They relate to quality characteristics in particular.

The meta model as a whole conveys a coarse idea on the complexity of framework design: which classes of elements need to be considered and how they are related. Basically, all topics addressed in the introductory list of questions can be modeled. If required, the model can be extended for further classes of elements. Alternatively, they can be modeled as instances of the generic *ModelElement* and *Relation* classes.

Not surprisingly, creating useful models of design knowledge based on this meta model requires significant effort. The models we created so far contain e.g. more than 50 of each quality characteristics, design principles, framework requirements, and publications. Several approaches are applied with the aim reduce effort for model creation and maintenance:

- In order to split the model into manageable parts, one submodel is created per publication that models its contents. The final model is then created by merging these submodels. This way, all elements can automatically be related to publications.
- Subsequent, text mining on publications is used to autocreate model elements (including quotes and hyperlinks) based on a list of keywords. These can be used as a starting point. The modeler decides which auto-created elements are relevant – and adds relations and further elements manually.

Overall, the modeling approach is still in an early phase and further experience needs to be gained – including evaluation of how much value can be drawn from such models. On the positive side, a considerable amount of insights was obtained during modeling already – combined with a lot of reasoning on own designs. Regarding the modular design approach, many framework concerns from the decomposition in Fig. 2a

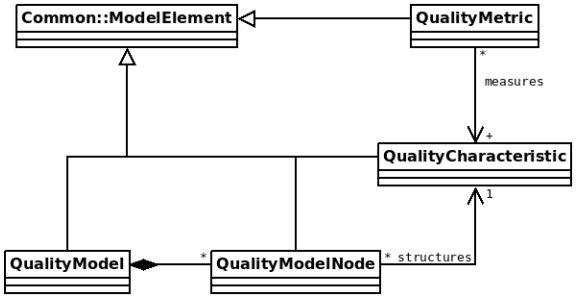


Fig. 6: The *Quality* modeling package

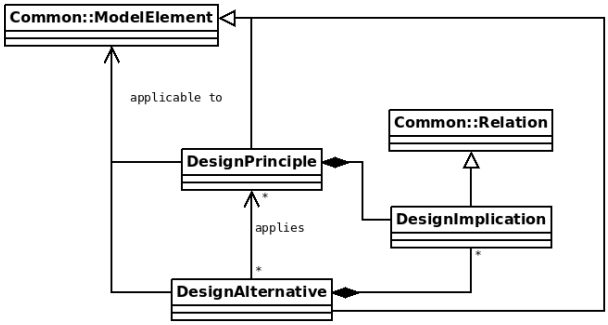


Fig. 7: The *Design* modeling package

are also resembled in model – together with a considerable set of relations to other model elements that are relevant when designing them. Awareness of relations – to quality attributes and common requirements in particular – contributes to reaching the target of "one fits more".

5 Implementation

The presented modular design concept was implemented in the FINROC framework[7] that serves as a proof-of-concept and for evaluation. Development was started in 2008 in the RRLab at the University of Kaiserslautern – with the first public release in 2013. It is still actively developed. FINROC preserves application style and valued quality characteristics of MCA2 [20] that was used before – but overcomes many of its limitations. Notably, FINROC is also used at Robot Makers GmbH for the development of commercial robot control systems (prototypes and smaller series). Professional support is offered as well.

There is a highly modular C++11 implementation used for robot control systems – and a native Java implementation used for tooling and Android-based user interfaces. Fig. 8 shows the C++11 FINROC software artifacts developed so far that fit into the concept presented in the last section. The biggest deviation are the scheduling concerns that currently lack separation. Each artifact is developed in its own code repository and compiled to a separate library. Any libraries not already linked to the application executable may be loaded dynamically at runtime (provided the *runtime construction* concern is present). Currently, the core, central plugins, and most tools are released as open source software[8]. Both the RRLab and Robot Makers use FINROC in all active robot control system development projects – with more than ten FINROC-based projects successfully completed at each site[9]. Both independently maintain and develop repositories with hundreds of components. This is already evidence that the highly modular design approach is feasible – also outside academia.

Furthermore, the small number of LOC per block is beneficial for maintainability – an aspect particularly stressed by [4].

Unlike e.g. Orocos, the implementation currently features only one intra-process transport. As presented in [16], it is, however, suitable for most relevant quality requirements – being both lock-free and efficient (zero-copy) with support for dynamic wiring – provided DCAS (double compare-and-swap) operations can be used. It is implemented in *data_ports* (see Fig. 8) – with significant dependencies to *buffer_pools* and *concurrent_containers* provided by the respective libraries.

thread and *time* deal with multi-threading and time representation. They are based on the respective functionality in the C++11 standard – which provides the basis for platform-independence. It is possible to compile *thread* in single-threaded mode. This replaces concurrent data structures with simpler ones – and also enables a single-threaded data port implementation (see [21]). The latter furthermore allows to map port values to static memory addresses – an opportune feature for shared memory communication. *xml* is an optional core library for dealing with XML. Furthermore, there are core libraries for *logging*, *serialization* and runtime type information (*rtti*). The latter allows to handle C++ types in a uniform way – providing operations such as instantiation, serialization, and comparison at application runtime. The new *rtti_conversion* allows for runtime type conversion – a feature often requested to seamlessly integrate components without the necessity for separate type-casting components. Any plugins, libraries, and applications can provide and register further type casting operations.

Concurrent, dynamic application structure – including dynamic (component) interfaces – are central features provided by the *core*. On the interface definition layer, there is furthermore optional support for *blackboards* used by some components – and *parameters* for uniform component configuration at runtime. Similar to other state-of-the-art frameworks, *RPC ports* allow to invoke functions (or "services") in other components – which is required for more complex component interaction patterns.

7. http://www.finroc.org

8. Source code, instructions, and tutorials are available on the FINROC homepage. The code can also be browsed via http://www.finroc.org/browser.

9. see http://robotics.unibg.it/tcsoft/simpar2014/22-Wednesday/WeP.6.pdf http://robotmakers.de/en/references/ or https://agrosy.cs.uni-kl.de/en/robots/

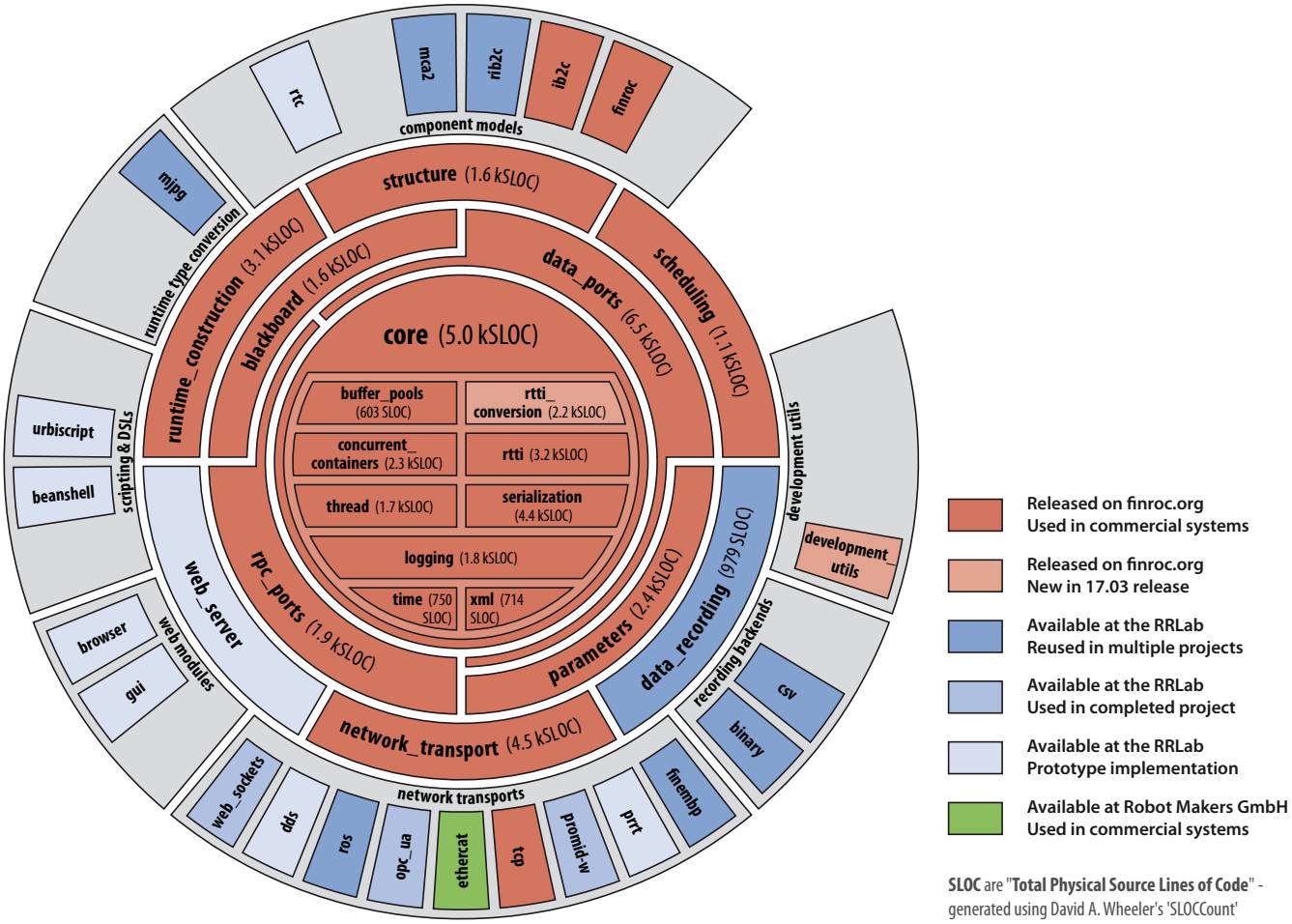


Fig. 8: Implementation of the proposed concept in the FINROC framework – status quo

structure and *network_transport* contain abstractions and common functionality for the respective leaf concerns. The former includes functionality for creating composite components. Since FINROC 17, the latter also contains a generic network transport suitable for full-featured peer-to-peer FINROC communication via virtually any communication channel with support for sending binary messages. It features simple quality of service (QoS) mechanisms – with runtime-modifiable parameters for desired update intervals, data priority, data queues, congestion control, and monitoring of roundtrip times. The *tcp* plugin instantiates this generic transport with TCP/IP sockets. It is currently the default network transport in FINROC. Combined with appropriate measures for security, it is notably suitable and used for accessing robots over the Internet – e.g. for remote maintenance and support.

runtime_construction provides functionality to instantiate, connect, and delete components at application runtime – including the possibility to store and restore networks of connected and configured components to and from XML files.

finembp is a custom, Ethernet-based transport for communication between bare metal nodes and a PC (see [21]) – including fixed-size messages with deterministic timing. It is used in the experiments presented in the next section.

For additional insights and to convey a better idea on the nature of developed commercial systems, statistics for six selected successful projects at Robot Makers were created. They are all based on the FINROC 14.08 release and were completed after August 2016. Table 1 contains totals on the number of components, ports, parameters, and connectors instantiated in the systems – values characterizing the complexity of the component integration task. Fig. 9 breaks the total number of components in these projects up into several categories. The solid bars represent the number of component types used in the project. If components are instantiated multiple times, the additional instances are visualized by hatched bars. The following component categories have been differentiated:

- *Type Converter* components: These components merely convert data types in order to connect data ports of different types. Since the FINROC 14.08 release, SI units are used in component interfaces where applicable.

Project	Components		Ports	Parameters	Connectors
	Types	Instances			
1	91	242	1878	705	1100
2	42	58	458	200	312
3	52	111	751	359	454
4	36	113	807	491	352
5	90	253	2641	1342	1388
6	104	572	3074	1705	1837

TABLE 1: Statistics on six commercial Robot Makers projects – from the domains of agricultural machines, construction machines, and special purpose vehicles.

Futhermore, there are multiple types for poses and twists – depending on dimension and uncertainty information. Notably, using strong types in interfaces has many benefits (e.g. regarding functional correctness), but also led to the situtation that reused components fit together less often. As a consequence, between 12 and 22 percent of component instances in the analyzed projects are type converters. This motivated supporting native type conversion capabilities in Finroc 17.03. This makes converter components obsolete – beneficial not only for maintainability and performance efficiency. A lesson learnt from this, is the importance of type conversion functionality when using strong types in component interfaces.

- *Trivial Components*: These component wrap simple mathematical or logical operations, such as "Multiplication" oder "Comparison". They are sometimes used to implement e.g. controllers out of simple blocks (somewhat similar to other graphical programming environments). Such structures have popularity due to FINROC's features regarding runtime modifiability: control structures can be changed online without recompiling and restarting the application. This is opportune for optimizations during field tests and indicates that this is a relevant quality characteristic.
- *Behaviour* components are based on the iB2C architecture's component model [22].
- *Standard* are components based on the standard FINROC component model that do not belong to any of the other categories.
- *Composite* components are components containing and encapsulating other components.

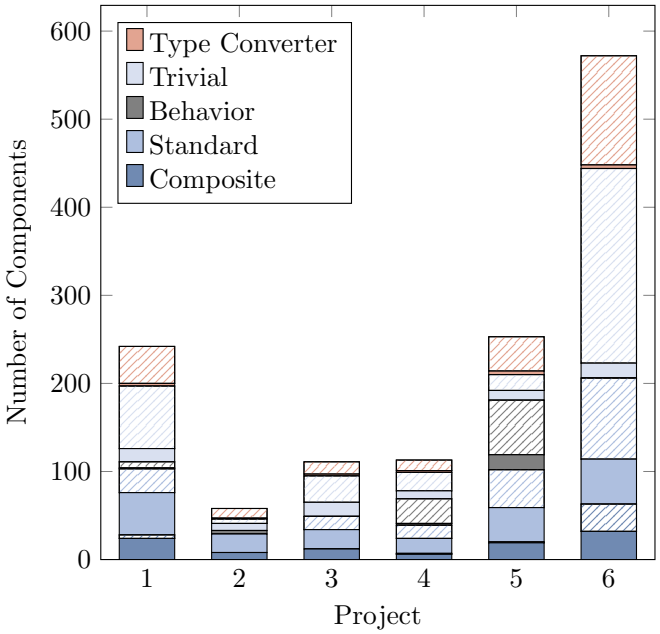


Fig. 9: Detailed breakdown of the number of components in six Robot Makers projects. The solid bars represent the number of component types per component category. The hatched bars above indicate the additional number of instances for each category (if components are instantiated more than once).

6 EXPERIMENTS AND EVALUATION

6.1 Running FINROC Applications Bare Metal

FINROC was originally developed for powerful computing nodes executing hundreds and in few cases even thousands of components. The underlying concept, however, makes its quality profile sufficiently flexible to be suitable for bare metal embedded applications also. Notably, components and applications are developed in the same way – the platform-independent ones can be used in both worlds. Also the same tooling can be used. This mitigates the gap in the development process usually found between PCs and embedded systems [23]. We are not aware of any other comparable robotic framework which is used bare metal. Notably, this a major design goal for ROS 2.0. The following section presents deployment of a FINROC application on a bare metal soft core running in an FPGA. This case study was selected as an evaluation of relevant quality attributes often in conflict with modular designs – and of the target of "one fits more".

Fig. 10 shows the FINROC configuration that is run bare metal in the experiments. It is cross-compiled and run in single-threaded mode with all libraries being statically linked. Notably, any further blocks that do not require an operating system can be added when needed. Only a small set of functions had to be added that were missing in the Altera C++11 implementation.

The targeted FPGA is at the core of an embedded system developed for the encapsulation of the RRLAB SEA – a Series Elastic Actuator (SEA) [24]. The SEAs, and hence the embedded nodes, are actuating the **C**ompli**a**nt **R**obotic **L**eg (CARL) (see Fig. 11). CARL is a bio-inspired leg that includes – inspired by the morphology of humans – mono- as well as biarticular actuation. As, in this scenario, small physical dimensions, a low energy consumption, high cycle

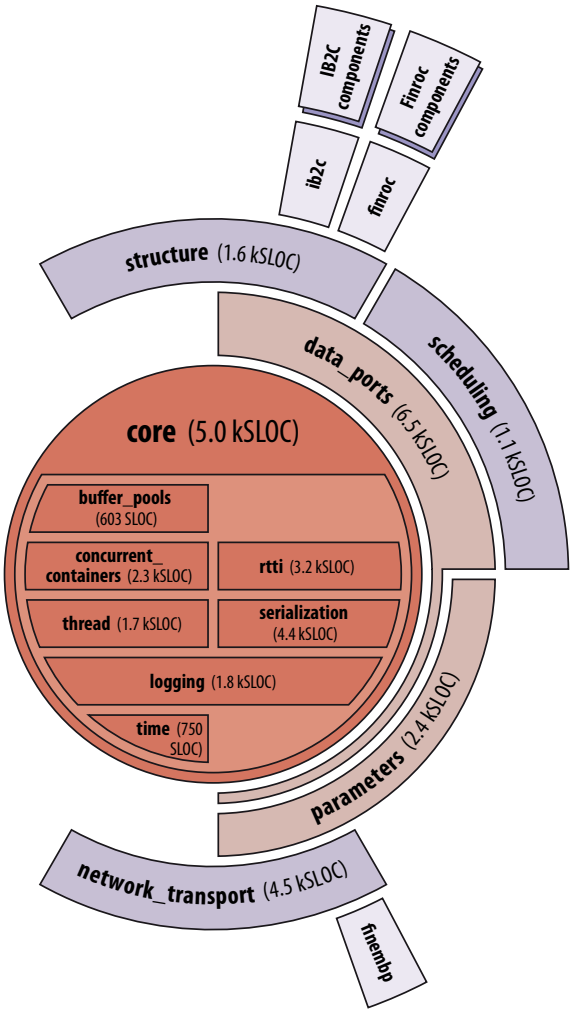
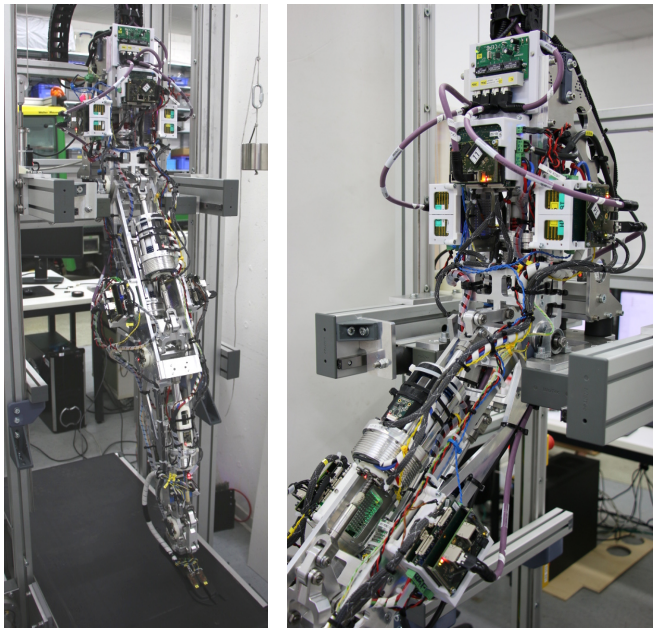


Fig. 10: FINROC configuration run bare metal



(a) (b)

Fig. 11: (a) The Bio-inspired **C**ompli**a**nt **R**obotic **L**eg (CARL) suspended by the enclosing test-rig. (b) Close view of CARL's torso and thigh. The five embedded systems as well as the Ethernet switch can be seen.

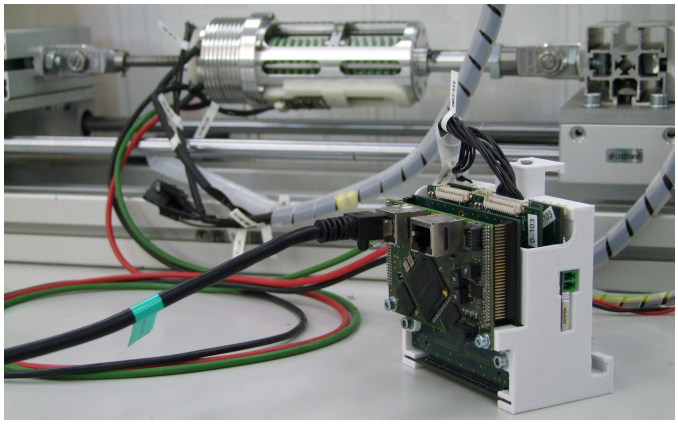


Fig. 12: The FPGA-based system with a RRLAB SEA in the background.

frequencies, and a deterministic timing are hard requirements for the embedded system, FPGAs are an opportune solution. They provide the flexibility to design a system that optimally supports the deployment of a full-featured framework to a bare metal embedded CPU.

The concept and implementation of the FPGA system for the deployment of FINROC is presented at an early stage in [21]. By following a HW/SW codesign approach, it is possible to achieve a good balance between computational performance on the one side and the application requirements on the other side. Conceptually, the Ethernet-based communication is decoupled from the application by distributing the two tasks to separate subsystems – communication and application system. At the core of each is a soft core – the NIOS II processor from Altera[10]. The two CPUs are coupled via a dual-ported RAM, all data transfers are executed by DMAs. Generally, the decoupling allows for a high communication bandwidth with a high cycle frequency as well as a low jitter on the application side. The system is accomplished by an ELMO Gold Twitter 25/100[11] which is handling the commutation and the current control of the BLDC motor driving the RRLAB SEA. The embedded system as well as the down-scaled version of the RRLAB SEA is shown in Fig. 12.

To abstract the RRLAB SEA as an impedance source while

10. https://www.altera.com/

11. http://www.elmomc.com/

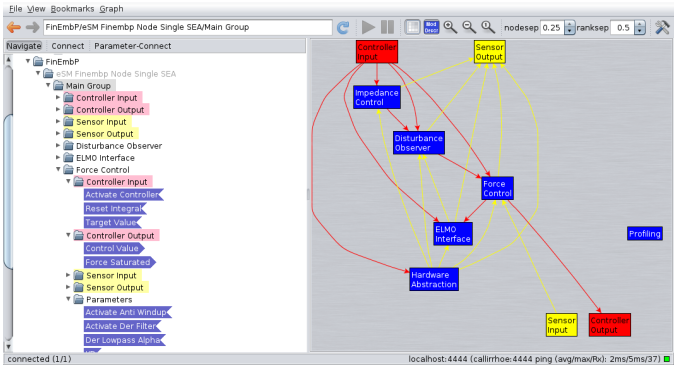


Fig. 13: FINROC system executed on the FPGA, illustrated by FINSTRUCT

Measuring Point	Avg. CPU cycles	Std. Deviation	Max. Jitter
Cycle Start	-1	14.3	64.5
HW Abstraction - Sense	711	16	72.7
ELMO Interface - Sense	4663	74.8	249.3
PID Control - Sense	5785	74.9	270.1
DOB - Sense	7182	117	364.9
Impedance Control - Sense	7536	138.4	414.2
HW Abstraction - Control	7914	138.9	414.2
Impedance Control - Control	8435	140.9	436.3
DOB - Control	10074	180.6	520.4
PID Control - Control	11105	201.4	569.4
ELMO Interface - Control	12074	201.5	569.4
Finroc Cycle End	13240	218.7	633.5
Data Handling End	22064	2201.3	5534.5
Cycle End	24868	13.7	49.1

TABLE 2: Timing during an execution cycle

maintaining high closed-loop frequencies, the application CPU is extended by dedicated IP-Cores. The structure of the cascaded control loops is discussed in [25]. Each element of the closed-loop control is implemented in software using a FINROC *Sense-Control Module* component. They are complemented by a component for hardware abstraction. Fig. 13 shows the resulting software system as visualized in FINSTRUCT. FINSTRUCT is FINROC's standard graphical tool for runtime visualization of component graphs, for accessing components' port and parameter values, as well as for runtime construction (adding, connecting, and deleting components; editing component interfaces).

In the following, the performance of the application software is analyzed with respect to the execution timing and the jitter of the different components. In order to capture the software timing without producing significant overhead, a dedicated IP-Core was implemented. The value of a counter driven by the 100MHz system clock is stored to embedded memory each time a flag is triggered from the software. It is ensured that the calls to the IP-Core are minimal – two consecutive snapshots take 2 clock cycles without any jitter.

The main functionality of FINROC *sense-control modules* is executed by its *Sense* and *Control* tasks. Tasks are attached to threads – in this application to a single thread. Based on the data flow, their execution order is determined by the default FINROC scheduler – and equals the listing order of the measuring points in Table 2. After a FINROC execution cycle, data exchange between the two CPUs is handled. Then, in order to keep the cycle time constant, the CPU idles until the next cycle start. The resulting timing within a cycle is plotted in Fig. 14, a quantitative interpretation of the data is given in Table 2. A timestamp is captured at the beginning of each control (red) and sense task (yellow). Additionally, timestamps are captured at the cycle start and end, after the FINROC execution, and after the data handling (grey).

It can be seen that the system can be run with a frequency of 4kHz (25000 CPU cycles per execution). This execution frequency compares well to the achieved performance in similar systems [23], [26], [27]. Within the FINROC loop, the jitter increases to a maximum of 569.4 cycles. Relative to the overall cycle length, the execution timing varies by 2.29%.

In this experiment, the NIOS II is configured with 32KB of instruction and data cache as well as dynamic branch prediction of 4096 entries. While improving the overall performance, the underlying heuristics of those features naturally introduce a jitter to the software execution. Hence, the numbers given above are heavily dependent on the functional interaction between the software implementation – e.g. number of modules and variables – and the configuration of the CPU. Nevertheless, they show that by following the proposed design approach, it is possible to deploy a full-featured robotic framework to a bare metal embedded system while matching or even exceeding the execution frequency of comparable systems. Additionally, the benefits coming with a full-featured robotic framework facilitate the development and the debugging of complex robotic systems.

6.2 Framework Extensibility and Modifiability

Achieving *extensibility*, *modifiability*, and – related – *maintainability* are central design goals of the proposed approach. Accordings to [2], "Tactics that split modules will reduce the cost of making a modification to the module that is being split" – so high modularity is desirable in this respect. Changes are typically more localized. With the possibilty to add, remove, or replacing blocks (framework concerns), highly modular frameworks are explicitly designed with a mechanism for modifiability. Section 3 and Fig. 2a sketch many scenarios for extending and modifying such a highly modular framework. As can be seen in Fig. 8, blocks in most of the mentioned categories have actually been implemented. Each can be seen as a case study and proof of concept of framework extensibility for the extension category they belong to – namely: supporting different network protocols, adding support for scripting and domain-specific languages, supporting further component models, or temporarily adding development extensions. More concern-specific are extensions

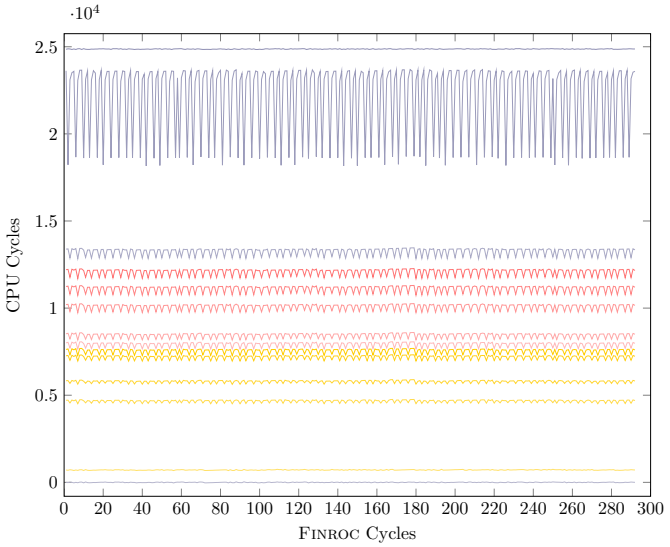


Fig. 14: Illustration of the execution timing within a cycle

that provide different data recording backends, web interface modules, or runtime type conversion operations.

7 Conclusion and Outlook

As discussed, the design of a robotics framework is in many ways a tradeoff between different quality characteristics and also other relevant design goals under given constraints. The proposed approach is no exception in this regard. With respect to bare metal embedded nodes, for instance, it is certainly possible to write software with better performance characteristics.

With the proposed modular approach, however, a framework can be suitable for a broader range of applications and target more platforms than without. As we show, the level of performance can actually be sufficient for applications on an FPGA soft core. In these cases, the approach brings significant added value with respect to other quality characteristics including e.g. runtime modifiability (creating, connecting and deleting components) or a consistent development process and tooling for all target platforms.

Overall, we show that proposed approach is generally feasible – and that is has a positive impact on many relevant quality characteristics of robotics software. In the limited scope of this paper, this cannot be analyzed for all the numerous relevant quality characteristics. Therefore, evaluation focused on quality characteristics (performance efficiency and timing determinism) that tend to be in conflict with very modular designs. Evaluation with respect to other characteristics will be part of future work.

As a further contribution, the decomposition in Fig. 2a contains an overview of many relevant framework concerns – and is the result of literature research, design experience, and domain modeling. For many of today's frameworks we miss an illustration of their architectural *decomposition structure* [2] – and encourage to create them for improved reasoning on designs.

In addition, we have presented how we currently model domain knowledge on framework design. This is still in an early state – and this topic certainly requires additional research. It would likely benefit from a collaborative approach.

Applying highly modular designs to model-driven development solutions would be another interesting direction of future research – as would be more formalized interface definitions of framework concerns. In general, as mentioned, how to support quality characteristics of robotics software by means implemented in frameworks is considered a key topic for further research.

References

[1] J. Dörr, *Elicitation of a Complete Set of Non-Functional Requirements*, ser. PhD Theses in Experimental Software Engineering. Fraunhofer Verlag, Stuttgart, 2011, vol. 34, ISBN: 978-3-8396-0261-4. 1

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012, ISBN: 978-0-3218-1573-6. 1, 3, 6.2, 7

[3] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and 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. 2

[4] A. Makarenko, A. Brooks, and 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. 2, 3, 5

[5] A. Y. Elkady and T. M. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012, doi:10.1155/2012/959013. 2

[6] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee, "Opros: A new component-based robot software platform," *ETRI Journal*, vol. 32, pp. 646–656, 2010. 2, 2

[7] T. Hammer and B. Bäuml, "The highly performant and realtime deterministic communication layer of the ardx software framework," in *16th International Conference on Advanced Robotics (ICAR)*, Montevideo, Uruguay, November 25-29 2013. 2

[8] I. A. Nesnas, "The claraty project: Coping with hardware and software heterogeneity," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30. 2

[9] B. P. Gerkey, R. T. Vaughan, and 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, pp. 317–323. 2

[10] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006. 2

[11] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. F. Ingrand, "Genom3: Building middleware-independent robotic components," in *IEEE International Conference on Robotics and Automation (ICRA 2010)*, 2010, pp. 4627–4632. 2

[12] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29–45, January 2008. 2

[13] C. Schlegel, T. Haßler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *14th International Conference on Advanced Robotics (ICAR)*, Munich, Germany, June 22-26 2009. 2

[14] D. Brugali and L. Gherardi, "Hyperflex: A model driven toolchain for designing and configuring software control systems for autonomous robots," *Studies in Computational Intelligence*, vol. 625, pp. 509–534, February 1 2016, doi: 10.1007/978-3-319-26054-9_20. 2, 3

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. 3

[16] M. Reichardt, T. Föhst, and K. Berns, "On software quality-motivated design of a real-time framework for complex robot control systems," *Electronic Communications of the EASST*, vol. 60: Software Quality and Maintainability 2013, August 2013, this publication is available at http://journal.ub.tu-berlin.de/eceasst/article/view/855. 3, 5

[17] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, Eds. Springer Berlin / Heidelberg, 2008, vol. 5325, pp. 87–98. 3

[18] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The brics component model: A model-based development paradigm for complex robotics software systems," in *28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. Coimbra, Portugal: ACM, March 18-22 2013, pp. 1758–1764, doi: 10.1145/2480362.2480693. 3

[19] M. Lutz, D. Stampfer, A. Lotz, and C. Schlegel, "Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns," in *Workshop Roboterkontrollarchitekturen, Informatik 2014*, ser. Lecture Notes in Informatics (LNI), Suttgart, Germany, 2014. 4

[20] K.-U. Scholl, J. Albiez, and B. Gassmann, "Mca - an expandable modular controller architecture," in *Proceedings of the 3rd Real-Time Linux Workshop*, Milano, Italy, November 26-29 2001. 5

[21] S. Schütz, M. Reichardt, M. Arndt, and K. Berns, "Seamless extension of a robot control framework to bare metal embedded nodes," in *Informatik 2014*, ser. Lecture Notes in Informatics (LNI), Stuttgart, Germany, 2014, pp. 1307–1318. 5, 6.1

[22] M. Proetzsch, T. Luksch, and K. Berns, "Development of complex robotic systems using the behavior-based control architecture iB2C," *Robotics and Autonomous Systems*, vol. 58, no. 1, pp. 46–67, January 2010, doi:10.1016/j.robot.2009.07.027. 5

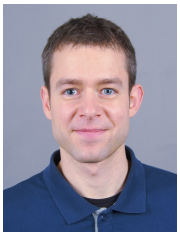
[23] N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater *et al.*, "Valkyrie: Nasa's first bipedal humanoid robot," *Journal of Field Robotics*, vol. 32, no. 3, pp. 397–419, 2015. 6.1, 6.1

[24] S. Schütz, K. Mianowski, C. Kötting, A. Nejadfard, M. Reichardt, and K. Berns, "RRLAB SEA – A highly integrated compliant actuator with minimised reflected inertia," in *IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*, 2016. 6.1

[25] S. Schütz, A. Nejadfard, and K. Berns, "Influence of loads and design parameters on the closed-loop performance of series elastic actuators," in *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2016. 6.1

[26] M. A. Hopkins, S. A. Ressler, D. F. Lahr, A. Leonessa, and D. W. Hong, "Embedded joint-space control of a series elastic humanoid," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 3358–3365. 6.1

[27] C. Ott, O. Eiberger, W. Friedl, B. Bäuml, U. Hillenbrand, C. Borst, A. Albu-Schäffer, B. Brunner, H. Hirschmüller, S. Kielhöfer, R. Konietschke, M. Suppa, T. Wimböck, F. Zacharias, and G. Hirzinger, "A humanoid two-arm system for dexterous manipulation," *Proceedings of the 2006 6th IEEE-RAS International Conference on Humanoid Robots, HUMANOIDS*, pp. 276–283, 2006. 6.1



Max Reichardt received his Dipl.-Inf. degree in computer science from the University of Kaiserslautern in 2008. Since August 2008, he is PhD student at the Robotics Research Lab. Software Engineering in the context of robotics is his main area of research. Topics of particular interest include software frameworks, their quality attributes, and design principles. He is a main author of the FINROC framework. In October 2015, Max joined Robot Makers GmbH where he works as robotics software engineer in research and development. In this position, he has contributed to more than a dozen of successful projects in the domain of mobile automation – for different customers and organizations.

Steffen Schütz received his Dipl.-Ing. degree in mechanical engineering specializing in mechatronics and microsystems from the Karlsruhe Institute of Technology in 2011. In late 2011, he joined the the Robotics Research Lab where he's currently workig towards a PhD in computer science.

His research interests lie in the area of biologically-inspired artificial bipedal walking. Within this area, the design of compliant bio-inspired walking machines is the main focus of his research activities. Besides the general mechanical structure and the actuation units, this includes the design of embedded control architectures. Driven by the functional, spatial and energetic requirements, they are composed of distributed, highly dedicated FPGA-based embedded systems designed following a HW/SW co-design approach.

Karsten Berns received his PhD from the University of Karlsruhe in 1994. As head of the IDS (Interactive Diagnosis and Service Systems) department of the FZI Research Center for Information Technology, Karlsruhe (until 2003) he examined adaptive control concepts for different types of service robots. Since 2003, he is a full professor for robotic systems at the University of Kaiserslautern. Present research activities are in the area of autonomous mobile robots and humanoid robots with a strong focus on control system architectures and behavior-based control. Karsten is a member of the IEEE, the Gesellschaft für Informatik (GI), and the CLAWAR Association. He is furthermore member of the executive committee of the German Robotics Association (DGR) and is currently leader of the technical committee for robotic systems of the GI.