One Fits More – On the Relevance of Highly Modular Framework and Middleware Design for Quality Characteristics of Robotics Software

Max Reichardt, Steffen Schütz, and Karsten Berns
Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, 67663 Kaiserslautern, Germany
Email: {reichardt,schuetz,berns}@cs.uni-kl.de

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 on 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.

I. 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, 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 toolkits1. 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 software can be ported to 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.

II. 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’”
3) “a one-size-fits-all solution to building robotic systems may be unachievable and undesirable”

1For simplicity, the term framework is used for all three in the following.
Regarding the first statement, many framework design decisions are tradeoffs between different quality attributes and cannot be considered as generally “better” or “worse”. Whether to use an IDL is an interesting example. As a result, frameworks have feature and quality profiles that make them more, less, or not suitable for specific applications. In this context, feature bloat is a delicate pitfall – detrimental to 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\(^3\) standard from automotive industry is in several ways similar to a (model-driven) robotics framework and is occasionally used for robotic applications. Due to constraints (e.g. no HEAP allocation), it arguably lacks flexibility required on higher levels of robot control systems. Therefore, some authors use a two-layer approach and combine solutions with different quality characteristics. This two-layer approach has the drawback of a gap in the development process.

The Player Project [8] 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 supports a broader spectrum of quality requirements and therewith applications. We call such variability the principle of customizable quality tradeoffs.

Somewhat similar, Orocos [9] allows using transports with different quality profiles also for intra-process communication (e.g. one for real-time support, one for high throughput). It decouples transports from interface definitions – an example for separation of concerns, a fundamental principle beneficial with respect to many quality characteristics. Opros [6] and GenoM3 [10] also feature transport-independence.

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 – including embedded nodes (see e.g. [11]). 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 complementary, as model-driven toolchains are developed that support non-model-driven solutions such as ROS (e.g. [10], [12]). A non-model-driven approach was chosen for the proof-of-concept presented in Section IV – though highly modular designs should be applicable in Model-Driven Software Development (MDSD) also.

### III. 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. It is further motivated by avoiding feature bloat and localizing quality characteristics to preferably few software entities in order to cope with design complexity.

Motivated by the implied quality characteristics, major elements from the microkernel pattern [13] 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, interface definition elements and intra-process transport(s), base concerns, leaf concerns, and repositories of components (see Fig. 1). Dependencies may 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. Notably, not all solutions implement all of these concerns:

- Some – as e.g. Orca 2 [4] – rely on generic middleware packages for interface definition and network communication. This reduces development and maintenance effort – notably, the quality characteristics of these middleware packages are also adopted.
- Some model-driven approaches such as the RTC standard mainly define a component model.
- Conversely, other solutions including ROS do not really define a component model.

Leaf concerns can either specialize the abstract base concerns (e.g. different network transports) or provide additional functionality based on them.

Fig. 2a illustrates a more fine-grained framework decomposition on this basis. This is the vision of what is pursued in our work on frameworks: All framework concerns are separate.

---

\(^2\)http://design.ros2.org/articles/why_ros2.html

\(^3\)http://www.autosar.org/

\(^4\)http://www.omg.org/spec/RTC/
software entities – possibly subdivided into sub-concerns. All blocks outside the core are optional. They 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 the projects. Used without base and leaf concerns, for instance, it is a plain middleware.

Every block has its own quality profile – and quality requirements can be localized to these blocks. For instance, efficiency and real-time requirements are primarily important for transports and scheduling. This can be visualized as a heat map (see Fig. 2c). In particular, if timing-critical functionality required from other blocks is time-bounded (e.g. lock-free), a framework’s timing requirements are localized to possibly platform-specific scheduling blocks that are instantiated and configured by the final 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 share the simple heat map shown in Fig. 2b – e.g. stability, maturity, maintainability, portability, and longevity. The modular concept is also beneficial for portability – with operating-system-independence being a particularly interesting quality characteristic for single blocks: a combination 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 – e.g. the stability and size of interfaces with many dependencies greatly affects maintenance effort. 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 IV, 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 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 mutation of a framework’s quality profile and features.

IV. IMPLEMENTATION

The presented concept was implemented in the FINROC framework\(^5\) 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 [14] that was used before – but overcomes many of its limitations. FINROC is also used at Robot Makers GmbH for the development of commercial robot control systems (prototypes and small 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. 3 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

\(^5\)http://www.finroc.org
concern is present). Currently, the core and central plugins are released as open source software. 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. 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, there is currently only one intra-process transport. As presented in [15], 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 – based on 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 [16]). The latter furthermore allows to map port values to static memory addresses – an opportune feature for shared memory communication. xml is an optional 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 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 support for blackboards used by some components and parameters for uniform component configuration. 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.

network_transport and structure contain abstractions and common functionality for the respective leaf concerns. The latter includes functionality for creating composite components. 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 [16]). It is used in the experiments presented in the next section.


![FinROC Configuration Run Bare Metal](image)

Fig. 4: FinROC configuration run bare metal

V. Experiments

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. This is currently a unique feature among robotic frameworks and a major design goal for ROS 2.0. This 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. 4 shows the FinROC configuration that is run bare metal. 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) [17]. The SEAs are envisaged to be integrated in a compliant robotic 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 frequencies, and a deterministic timing are hard requirements for the embedded system, FPGAs are an opportune solution.

The concept and implementation of the FPGA system for the deployment of Finroc is presented at an early stage in [16]. 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. The embedded system as well as the down-scaled version of the RRLAB SEA is shown in Fig. 5.

To abstract the RRLAB SEA as an impedance source while maintaining high closed-loop frequencies, the application CPU is extended by dedicated IP-Cores. The structure of the cascaded control loops is discussed in [18]. 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. 6 shows the resulting software system as visualized in FINSTRUCT.

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 I. 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. 7, a quantitative interpretation of the data is given in Table I. 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).

<table>
<thead>
<tr>
<th>Measuring Point</th>
<th>Avg. CPU cycles</th>
<th>Std. Deviation</th>
<th>Max. Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Start</td>
<td>1</td>
<td>14.3</td>
<td>64.4</td>
</tr>
<tr>
<td>HW Abstraction - Sense</td>
<td>711</td>
<td>16</td>
<td>72.7</td>
</tr>
<tr>
<td>PID Control - Sense</td>
<td>5785</td>
<td>74.9</td>
<td>270.1</td>
</tr>
<tr>
<td>DOB - Sense</td>
<td>7182</td>
<td>117</td>
<td>364.9</td>
</tr>
<tr>
<td>Impedance Control - Sense</td>
<td>7536</td>
<td>138.4</td>
<td>414.2</td>
</tr>
<tr>
<td>HW Abstraction - Control</td>
<td>7914</td>
<td>138.9</td>
<td>414.2</td>
</tr>
<tr>
<td>Impedance Control - Control</td>
<td>8435</td>
<td>140.9</td>
<td>436.3</td>
</tr>
<tr>
<td>PID Control - Control</td>
<td>10074</td>
<td>180.6</td>
<td>520.4</td>
</tr>
<tr>
<td>PID Control - Control</td>
<td>11105</td>
<td>201.4</td>
<td>569.4</td>
</tr>
<tr>
<td>ELMO Interface - Control</td>
<td>12074</td>
<td>201.5</td>
<td>569.4</td>
</tr>
<tr>
<td>Finroc Cycle End</td>
<td>13240</td>
<td>218.7</td>
<td>633.5</td>
</tr>
<tr>
<td>Data Handling End</td>
<td>22064</td>
<td>2201.3</td>
<td>5534.5</td>
</tr>
<tr>
<td>Cycle End</td>
<td>24868</td>
<td>13.7</td>
<td>49.1</td>
</tr>
</tbody>
</table>

TABLE I: Timing during an execution cycle

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 [19], [20], [21]. 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, a full-featured robotic framework facilitates the development and the debugging of complex robotic systems.
VI. CONCLUSION AND OUTLOOK

As discussed, 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 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 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, we evaluated 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 and design experience. 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.

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

Fig. 7: Illustration of the execution timing within a cycle

REFERENCES


