Introducing **FINROC**: A Convenient Real-time Framework for Robotics based on a Systematic Design Approach

Max Reichardt, Tobias Föhst, Karsten Berns

**Abstract**—In this paper, we discuss critical aspects in the design of general-purpose software frameworks for the robotics domain, and present **FINROC** – our attempt to deal with the identified non-functional requirements in the best possible way. Numerous publications on this subject were taken into consideration. Furthermore, we stress the importance of consistently separating framework-independent from framework-dependent code. With maximum (intra-process) communication efficiency in focus, **FINROC** features a lock-free, zero-copying implementation including support for queues and various communication patterns. Decoupling being a paradigm for the framework internals, it has a minimal core extended via plugins. There is optional graphical tooling that supports to instantiate, connect, and remove components at application runtime. As we plan to continue research on huge behavior networks, **FINROC** is designed to cope with thousands of components and hundreds of thousands of ports.

I. **Motivation**

Developing controls for software-intensive autonomous robots is a challenging and time-consuming task – especially when systems grow beyond a certain size. Difficulties include reliably operating in complex environments with only limited information, restricted computing resources, and system requirements as well as hardware platforms that frequently change. As this is critical for progress in robotics, considerable research is performed in how to deal with these challenges. Many well-engineered frameworks, algorithms, and toolkits have been created and released that provide solutions for many common problems – a fundamental contribution to coping with this complexity.

A suitable framework is of great importance, as it has critical impact on software quality. It should provide all the necessary facilities so that a developer can concisely and conveniently address problems arising in the application domain.

We have been using **MCA2** [1] for developing robot controls in the past decade, and learned to appreciate many of its qualities. It features real-time support and scales well. By imposing certain constraints and guidelines on application structure, it is well-suited for visualizing applications. Controls of different robots look fairly similar and familiar. Reuse across different projects such as **RAVON**, **ROMAN**, or an autonomous bucket excavator (see fig. 1) works well, too1.

However, **MCA2** has some flaws. The network layer is not robust and there is no support for pushing data (publisher/subscriber) or quality of service. Complex types need to be transferred via blackboards, which causes issues related to locking. There is no support for asynchronous events. The strict structure led to some questionable workarounds. Connecting components must be done in source code and can be cumbersome. A former colleague stated that he spent half of his PhD time only doing this.

Discussing these issues, we agreed that we want a framework with the strengths of **MCA2** but without its weaknesses – and backward-compatible for migration of existing systems. Furthermore, it should remain lightweight, feature a lock-free design, a plugin architecture, and optional support for instantiating and connecting modules at runtime using a graphical tool. It must be suitable for implementing huge behavior-based networks for our research in behavior-based robot control [2]. Apart from that, non-functional requirements are of particular interest to us, since they “characterize software quality and enable software reuse” [3].

Numerous robotic frameworks have been proposed and developed in the past. However, we could not find one that comes close to satisfying our requirements and expectations (see III). Finally, we decided to develop a framework which we consider – with respect to our requirements – superior to existing solutions. As argued in [4], it can be perfectly feasible to do this.

---

II. On the Importance of Separating Framework-independent Code

Decoupling software artifacts is good software engineering practice – especially when aiming at software reuse. Developers of some frameworks explicitly encourage doing this ([4], [5], [6]) – and we fully agree. While this might seem banal, it does in fact prevent many problems when performed systematically. In our experience, reuse of software artifacts across research institutions works best with code that is framework-independent. OpenCV [7] is a good example.

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 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.

III. Popular Robotic Frameworks

Numerous general-purpose frameworks for robotics exist. Notably, few of them actually provide real-time support. [8] assess deficits with respect to non-functional requirements.

ROS [6] is currently the best-known and most widespread robotic “operating system”, prominently supported by Willow Garage. Designed with different objectives in mind, application building blocks are nodes which each run in a separate process. Below this level, ROS itself does not provide any structure or facilities for intra-process communication. This limits suitability for our requirement of instantiating several hundred behaviors. [9] discusses the computational overhead induced. Notably, ROS is relatively simple to integrate with other robotic frameworks.

Orocos [10] and the derived Robot Construction Kit [5] appear to come closest to our requirements. An interesting design choice is detaching the component API from the actual framework backend (“transport plugin”). This way, the same components can be used in different frameworks and contexts (ROS, for instance). Both FINROC and Orocos feature a lock-free, real-time intra-process communication mechanism. Unlike Orocos, however, FINROC does not copy data buffers – as encouraged by [11] – resulting in less CPU load and latency (see VI). Besides, FINROC provides tooling to instantiate and connect components at runtime. Similar to Orocos, any C++ type can be sent via FINROC ports. No assignment operator is required, however.

Because of the limitations of MCA2, its original developer – the FZI in Karlsruhe, Germany – started working on MCA3.

The many other robotic frameworks include Microsoft Robotics Developer Studio2, URBI [12], CLARAty [11], OpenRTM [13] or cisst [14]. ADTF [15] is a framework with somewhat similar concepts used in the automotive industry. [88x726]P

IV. Critical Areas of Design

This chapter discusses FINROC’s central design decisions and conventions in a broader context.

When designing a framework, many critical decisions must be taken – involving tradeoffs. It is almost impossible to change some of them later without reimplementing major parts of the framework. Real-time support is a good example. Therefore, we followed a systematic approach:

1) Try to identify all critical areas of design: Where do philosophies in existing solutions differ? Which are best practices, common problems and lessons learnt?
2) Take well-founded decisions, after carefully evaluating the available options and implications involved.

These are some of the results:

a) System Decomposition & Component Interfaces:

Virtually all popular robotic frameworks follow a modular approach. “It is both desirable and necessary to develop robotic software in a modular fashion without sacrificing performance” [16]. Robot controls commonly consist of software entities that encapsulate algorithms, hardware access etc. Depending on the framework, these application building blocks are called “components”, “services” or “modules”. There are two major styles of interfaces among such entities: 1) ports that simply provide and consume data – either directly connected forming data-flow graphs, or indirectly communicating via topics; and 2) interfaces with remote procedure calls or synchronous transactions as known from web services or CORBA. Data flow graphs are simple and a natural fit especially for lower-level control loops. MCA2 applications, for instance, are entirely based on this paradigm. The order of execution is inherent. However, for complex interaction patterns, data flow graphs are not appropriate.

Designing interfaces based on data ports is typically straightforward. There is a realistic chance that two independent developers implement components that can be connected and used together – e.g. in image processing. RPC-interfaces, in contrast, can lead to many marginally different, incompatible interfaces, which hinders reuse. The developers of the Player Project discuss this difficulty [17] and the necessity of introducing standard interfaces. Frameworks such as ROS and Orocos support multiple types of interfaces. We believe that this is the right approach. In FINROC, directly connected data ports are the primary paradigm. But should this be insufficient, RPC-interfaces can be used.

Another question is, which granularity reusable software entities should have. Do components make sense that perform simple mathematical operations? Because of the limited benefits and overhead involved, this is usually not recommendable. However, in our behavior-based networks, such mathematical operations are sometimes useful for joining components. Generally, we think that application developers can themselves decide on a suitable granularity for their reusable artifacts. Regarding the framework, it is important that it does not impose limits on reuse. Thus, relatively small components should be feasible – causing minimal development and runtime overhead. Performing simple mathematical

2http://www.microsoft.com/robotics/
operations in a separate thread or even process is certainly a waste of resources and induces latency.

b) API & Application 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, Player or Orca [18] strive to prescribe as little as possible or necessary. Unnecessary contraints can be a nuisance and possibly lead to ugly workarounds. In MCA2, for instance, cycles in the data flow graph are forbidden. This led to inserting loop-back modules to realize such cycles.

However, strict guidelines help to avoid chaotic implementations. In our university group, numerous developers contribute to projects and libraries. Most existing code is from developers no longer working at the university. 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 is explicitly separated into a functional and a decisional layer – we suggest clearly separating APIs from the framework core and enforcing significant guidelines in those APIs only. The core should prescribe 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.

c) Promoting software reuse: In the robotics community, systematic software reuse is widely regarded as a key issue for making progress (see [19]). [3] names three aspects that are essential for reuse: quality, technical reusability, and functional reusability.

It is much more difficult to create general solutions that can be reused across a broad range of systems with vastly differing requirements than to produce slim, specific ones tailored to a limited scope. It is a recurring and difficult question for software developers whether the benefits of reusing existing code outweigh the necessary adaptions and possible drawbacks such as the risk of discovering missing features, unfortunate design decisions or performance issues – possibly late in the development process. Licensing issues and limited possibilities with respect to bugfixes can complicate things further. Many publications including [20] and [3] deal with difficulties regarding reusable artifacts in robotics.

As discussed in chapter II, separating framework-independent code is a simple and very effective step in this direction – as is decoupling in general. However, inside FINROC, interfaces should just fit – without creating a wrapper. In IV-a we explained why we think that data port interfaces are advantageous in this respect.

Having refactored a considerable amount of code from members that are no longer in our group, we discovered that – as a rule of thumb – reusable libraries with up to 5000 lines of code are very 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. When a library becomes larger, a split-up should be considered. With this policy, we hope to support a clear separation of concerns and avoid heavy-weight software artifacts as well as feature bloat. This also increases suitability for embedded systems. The boost library project acts as an archetype here: There is one, handy, well-engineered library (“RRLIB” in our terms) for every topic of interest. To minimize dependencies and to ease integration, standard constructs from the C++11 and Java APIs should be used whenever possible.

A major challenge with respect to reusable software artifacts is handling variability across a broad range of projects (as discussed in [20]). 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 [21].

d) Programming Paradigms & Language Expressiveness: An area not often addressed in robotic frameworks is programming language expressiveness [22] and the support for different programming paradigms in general [11].

Some frameworks provide scripting languages that support handling finite state machines, parallelism and concurrency in a sophisticated way. An example is URBIScript [12]. Orocos recently switched from its own scripting language to LUA for maintenance reasons [23]. Another interesting approach is ROCIScript [24]. ROS includes major support for Python, and Python-based SMACH [25] is an interesting framework for higher-level robot controls.

We designed FINROC in a way that supports easily adding scripting languages via plugins – interfacing with only a small number of basic primitives is sufficient (see V). We do not intend to develop a scripting language ourselves and rather consider integrating URBIScript or LUA.

[22] encourages using dimensional analysis. As a first step, FINROC allows assigning SI units to numeric ports – to avoid errors when components use different units in their interfaces. Values are automatically converted.

e) Generality, Flexibility & Extendibility: As framework maintainers, we receive frequent requests for adding helpful features. In an extensible framework, such missing features can be added with reasonable effort. Feature bloat and degeneration of the framework’s architecture can be avoided, if the respective code is clearly separated from the framework core – and optional. Software systems with a plugin architecture are certainly very extensible. We therefore chose such an architecture for FINROC: Almost anything ranging from a scheduling mechanism to a web interface can

http://www.boost.org/
be implemented and added as a plugin. A growing repository of optional plugins should furthermore increase flexibility.

**f) Code Complexity, Maintainability & Quality:** Code complexity and maintainability correlate. Simple solutions with compact source code require less maintenance effort and are less likely to contain programming errors. [4] discusses this topic regarding the Orca 2 framework and proposes a clear separation of concerns and a minimal framework core. We very much agree and try to implement this in FINROC – by means discussed in IV-b, IV-c and IV-e. This, furthermore, allows focusing efforts on ensuring software quality on relatively small and important core repositories.

**g) Efficiency, Scalability & Coupling:** In the mobile robotics domain, software performance is a critical factor – as this determines required computing resources and battery power. Generally, efficiency becomes more critical the smaller a robot is. It is therefore highly desirable that frameworks for mobile robotics allow creating efficient applications – consuming only minor computing resources for management and communication tasks. A key issue is sharing data among connected software entities and threads. If those entities are located on the same system, there are several options with great differences regarding efficiency. The most efficient solution is a shared memory approach. However, concurrent access has to be handled with care. Costs for locking and synchronization easily outweigh the performance gain. Another approach is generally creating deep copies of data exchanged by modules. There are also frameworks that always encode data in XML for data exchange. This causes significant computational overhead.

Good scalability is another critical topic. Fortunately, “Modular design promotes a style of application architecture that scales well and is a natural fit for distributed computing” [24]. This is especially true if modules are connected with data flow semantics (see IV-a). Modules with no data dependencies can be simply executed in parallel. Modules with dependencies may be pipelined. Network transparency is clearly advantageous for scalability, allowing adding further computing nodes if processing power is insufficient. This is supported by almost any robotic framework.

In FINROC we strive for the maximum performance that is reasonably possible with a modular approach. We therefore opted for shared buffers in order to avoid the cost of copying data. This has to be done in a lock-free way without allocating memory in order to allow efficient, scalable, predictable, real-time execution. We accept an increased memory footprint, as memory is not an issue on our systems.

**h) Network Layer:** Robustness, quality of service, interoperability, security, supported communication patterns and CPU load are important aspects when choosing the mechanism to transfer data over the network. Their relevance varies with respect to the application scenario. In a server rack on a robot, high throughput and minimal CPU load are advantageous. For tele-operation via WLAN and Internet, robustness, data compression and security are desirable.

With respect to varying scenarios and interoperability, we believe that an exchangeable transport mechanism – as found in ROS or Orocos – is good practice.

Regarding the primary transport mechanism, some frameworks rely on custom TCP-based implementations tailored to their requirements. This includes ROS, Player and mCA2. Others rely on professional middleware packets based on standards such as CORBA [26] or DDS4. ICE [27] is a popular product not based on these standards.

This is a somewhat controversial topic: “By building a custom protocol and server instead of adhering to a generic communications standard, such as CORBA or Jini, we are free from the computational and programmatic overhead that is generally associated with the practical application of such a standard” [28]. [18] state that creating and maintaining a custom middleware for Orca 1 required significant effort – with a result that was inferior compared to professional middleware products.

In FINROC we opted for an exchangeable transport mechanism. The default is a light-weight boost_asio-based TCP implementation with less than 3500 lines of code that integrates tightly with FINROC’s buffer management. It is a peer-to-peer mechanism featuring basic quality of service (QoS) and automatic reconnecting. The publisher/subscriber pattern as well as synchronous and asynchronous calls are supported. It uses binary serialization.

We do not intend to implement a transport mechanism with sophisticated QoS ourselves. We rather consider a FINROC plugin based on the DDS standard.

**i) Ease of Use & Reduction of Development Effort:**

As discussed in [19], ease of use is critical regarding a framework’s acceptance. Ideally, its added value quickly exceeds the efforts for adopting it. Major collections of reusable components, convenient tooling, a simple installation, as well as a clear and intuitive API are certainly advantageous.

With FINROC, we decided to adhere to Debian/Ubuntu standards very closely, since our primary development platform is wide-spread and works well. All our reusable libraries and FINROC plugins can be installed via apt and used like any other open source library. Inspired by ROS, we do not wrap an application’s main function. There is, however, an optional default implementation. This way, FINROC ports may be integrated into virtually any program or library.

Furthermore, we ensured that wrapping framework-independent code in FINROC requires little effort and code. Exploiting C++11 smart pointers and move semantics, we strive to make it impossible for application developers to use the port API in any way that causes memory leaks or other undesired behavior.

Since connecting ports in any written way (regardless of the framework used) can become tedious, we developed optional graphical tooling for instantiating and connecting modules. Stored in an XML file, manually editing the structure is possible and does not require recompiling.

Code generation is sometimes proposed in order to reduce development effort. In practice, constraints, unforeseen side-

4http://portals.omg.org/dds/
effects and reduced transparency when tracking bugs can quickly outweigh any benefits – so adoption should be considered carefully. We deliberately minimized the amount of generated code 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.

j) Robustness, Reliability & Fault Tolerance: These are very desirable features for a robot control. With respect to frameworks, removing one component must not crash any others. For weak WLAN links, the network layer should support QoS and automatic reconnecting (see IV-h). Generally, graceful degradation is desirable in case of sensor failure. Concepts for falling back on another sensor may only be feasible if the framework allows instantiating and connecting components at runtime – so this is supported in FINROC.

k) Runtime Model: A framework’s runtime model (see [11]) comprises whether execution is synchronous or asynchronous and how it is triggered – periodically, or by events. Furthermore, it defines how threads are mapped to components. At the one extreme, each component has its own thread – or even process, as in ROS. The other extreme executes all components in a single thread.

Synchronous implementations are typically simpler than asynchronous ones, but the latter perform better – especially when many threads are involved. MCA2 is an example for a framework that triggers execution periodically only. This has the advantage that it is simple and predictable. However, “it imposes an average delay of a half cycle on all data [...]” [17]

The target of a general-purpose framework is a runtime model suitable for all scenarios for which a programmer may need it. FINROC supports periodic as well as event-triggered execution as suggested by [11]. Furthermore, we decided that developers should be able to assign modules to threads.

l) Implementation Language, Portability & Interoperability: Interoperability and portability are important aspects with respect to software reuse and system integration.

An exchangeable transport mechanism (see IV-h) and the possibility of integrating a framework into other applications (see IV-i) are appropriate measures to increase interoperability. Ensuring interoperability with ROS is a good idea in this respect, as other frameworks are interoperable with ROS, too.

The choice of programming language has major impact on portability, performance, suitability for real-time implementations, development effort and the availability of reusable software artifacts. Most popular robotic frameworks have implementations in C or C++ – a good choice with respect to many of these aspects. We decided to create full, native implementations of FINROC in C++11 and Java – initially using an automated co-development process (see V-C).

We chose C++11 because its features support making implementations safer, shorter and more efficient compared to C++. It, however, requires a modern C++ compiler – somewhat limiting portability. We use GCC versions 4.5 and newer. Our implementation depends on the platform-independent xml2 and boost libraries. The Java version runs on further platforms and reduces development effort for our tooling. It has no dependencies.

m) Framework Longevity: In the past, many robotic frameworks were created – but also abandoned. Especially at academic institutions, efforts are often discontinued when their original developers leave. But major companies also cease to support projects if they do not appear profitable. It is therefore advisable to implement systems in a way that allows changing to another framework later without too much hassle (see II).

Non-maintainable code and unfortunate fundamental design decisions – especially with respect to non-functional requirements – will inevitably lead to abandoning or rewriting a framework. Regular refactoring can inhibit degeneration of a framework architecture – contributing to longevity. In this respect, a decision is required as to whether the framework API should be stable. Stable APIs impose limitations on refactoring. Unstable APIs cause effort in adapting applications – possibly annoying users.

V. FINROC IMPLEMENTATION

This chapter intends to give a brief overview of FINROC’s implementation. It is an example of how a framework core can be split up into small self-contained entities (see fig. 2). RRLIBs are framework-independent libraries. In the following, central libraries and plugins are briefly introduced.

To achieve our aims with respect to efficient, lock-free, zero-copying design, the implementation makes extensive use of buffer pools and thread-local storage. concurrent_containers provides a selection of efficient, lock-free linked queues. Due to their intrusive nature, they do not need to allocate memory. On this basis,
buffer_pools provides different variants of pools with reusable objects. For the lock-free implementation, we use stamped pointers in some classes – to avoid the ABA problem. This requires DCAS operations. For 64-bit systems, platform_abstraction_atomics provides an atomic integer with 128 bits.

We have our own serialization library for efficient stream-based binary, string, and XML serialization. We pay close attention that our RRLIBs, not related to the core, are decoupled from this library. In C++, external operator overloading is a good technique to achieve this.

Since the core does not know the types that might be used in ports and parameters, rtti provides runtime information for handling arbitrary types in a uniform way – e.g. for instantiating or serializing them. It also stores many available type traits for optimizations.

To ensure that the implementation is free of dead-locks, we enforce a total ordering on locks.

util_patterns contains best practice design patterns – inspired by [21]. These are template classes used throughout our implementation. finroc_core_utils is a temporary repository for diverse utility classes which are not yet part of another RRLIB.

data_ports provides raw data ports to realize applications based on the data-flow paradigm, whereas rpc_ports provides ports for object-oriented interfaces (see IV-a).

We intend to make these plugins optional and independent. data_ports_api provides convenient typed wrappers around the “raw” port classes. We implemented major parts of data ports in non-template classes to avoid code bloat. Notably, the RPC mechanism is not based on an IDL. Instead, available methods are implemented as slim variadic templates. Inside a process, procedure calls diminish to calling C++ functions directly – with no marshalling or additional objects involved.

parameters handles static and runtime parameters that can be set via config files, tooling or command line parameters. runtime_construction provides facilities to construct, connect and remove components at runtime. Using this plugin, FINROC applications do not need to be linked to the components they comprise. They can be loaded dynamically just as well – as long as the lack of additional thread-local storage is not an issue. tcp contains the lightweight, tightly integrated, boost_asio-based transport mechanism introduced in IV-h. blackboards provides network-transparent areas of shared memory – modifyable by acquiring write locks or using asynchronous transactions.

The MCA2 compatibility layer maps the MCA2 API to FINROC primitives. It allows running MCA2 programs without modification in FINROC.

structure contains the default FINROC API: Analogue to MCA2, basic application building blocks of every FINROC application are modules with input and output ports. A FINROC application is basically a set of interconnected modules. To structure applications, modules can be arranged in groups that have separate interfaces to encapsulate their contents.

![Fig. 3: Central classes in core](image1)

![Fig. 4: Fingui GUI editor](image2)

A. FINROC Core

There are four fundamental classes in the FINROC core (see fig. 3). 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. 4). 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.

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.

These central classes are implemented with support for concurrency and robust memory management to allow modifying the application structure – while it is running.

B. Graphical Tooling

Currently, there are two major tools for FINROC – both implemented in Java. The Fingui (fig. 4) is a convenient GUI editor (see [29]). Widgets can be arranged on a canvas and connected to any port or parameter in an application that has a compatible type. Fingui is the counterpart to MCAGUI in MCA2.
Table I: Results of image transport benchmark

<table>
<thead>
<tr>
<th>Framework</th>
<th># Consumers</th>
<th>ØFPS</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orocos</td>
<td>0</td>
<td>N/A</td>
<td>4 %</td>
<td>50 MiB</td>
</tr>
<tr>
<td>Locked</td>
<td>1</td>
<td>50.00</td>
<td>19 %</td>
<td>72 MiB</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>50.00</td>
<td>19 %</td>
<td>72 MiB</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>50.00</td>
<td>19 %</td>
<td>72 MiB</td>
</tr>
<tr>
<td>Orocos</td>
<td>8</td>
<td>N/A</td>
<td>4 %</td>
<td>44 MiB</td>
</tr>
<tr>
<td>Lock-free</td>
<td>1</td>
<td>50.00</td>
<td>11 %</td>
<td>78 MiB</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>50.00</td>
<td>50 %</td>
<td>228 MiB</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>49.20</td>
<td>100 %</td>
<td>419 MiB</td>
</tr>
<tr>
<td>FINROC</td>
<td>0</td>
<td>N/A</td>
<td>5 %</td>
<td>28 MiB</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>50.00</td>
<td>5 %</td>
<td>28 MiB</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>50.00</td>
<td>5 %</td>
<td>28 MiB</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>50.00</td>
<td>5 %</td>
<td>28 MiB</td>
</tr>
</tbody>
</table>

Fig. 5 shows Finstruct, a tool to inspect running applications. Similar to the MCABrowser tool in MCA2, the current values of all ports and parameters can be viewed and edited. Furthermore, the application structure and behavior networks are visualized. Unlike MCA2, new modules can be instantiated and connected – at runtime. Arrangement of modules is performed automatically by graphviz, so developers do not need to spend time arranging objects spatially – something that [19] criticizes in existing solutions.

C. Co-Development via Code Generation

Developing FINROC, we followed an unconventional implementation approach – inspired by Google’s Web Toolkit6. The framework prototype was implemented in a subset of Java based on a minimal API. We implemented the same minimal API in C++. This allowed rapid prototyping in Java and automatic code transformation to C++ when the result was satisfactory. Our conversion algorithm – in this case implemented as Eclipse-Plugin – had to be able to infer where to use references, pointers, smart pointers etc. We provided hints via Java annotations where necessary. Furthermore, it was possible to insert hand-crafted C++ code. The converter was continuously optimized to produce better C++ code – e.g. following guidelines by [30].

When FINROC was sufficiently stable, we ceased doing this. The code was tidied and more sophisticated APIs were created. We had mixed experiences with this approach. It works quite well to keep implementations in different languages aligned.

VI. Current Applications & Benchmarks

At our lab, pilot projects on smaller indoor robots were successfully ported to FINROC – and we are currently in the process of migrating the software of our mobile autonomous bucket excavator (see fig. 1c). Our largest project, RAVON (see fig. 1a), was successfully run using the MCA2 compatibility layer. With five processes and almost 100000 ports, this is a good benchmark with respect to scalability.

In our spin-off, RobotMakers GmbH7, FINROC is used on most platforms in development.

As FINROC focuses on communication efficiency, we were interested in how far the zero-copying implementation has an impact on computational overhead – compared to other frameworks. Uncompressed, high-resolution camera images are quite costly to copy or even serialize, so we chose those. We used Rock/Orocos as reference, as it features a state-of-the-art, intra-process communication model within one deployment – either locked or lock-free with copying.

A producer-consumer scenario was set up8: One producer sends HD RGB images (1920 × 1080) at 50 fps to several consumer tasks – filling the image buffers via memcpy9. Consumers are port-driven and calculate the arriving frames per second. CPU load and memory consumption were determined via htop. The same scenario was set up in FINROC. Only thread-safe communication mechanisms were used.

The results are shown in Table I. Using lock-free communication in Orocos, CPU load and memory consumption grow drastically with an increasing number of consumers. In FINROC, on the other hand, adding consumers has minimal impact on CPU load or memory usage. Notably, the gap in performance could be temporary – as an Orocos plugin based on the mechanisms used in FINROC should be feasible.

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.

VII. Conclusion and Outlook

In this paper, we present a systematic approach to designing and implementing a framework for robotics. We strove to find an optimal solution with respect to all relevant non-functional requirements. FINROC offers a unique

6http://code.google.com/webtoolkit/

7http://www.robotmakers.de

8All benchmarks were performed on an Intel Core i5-650 PC running Ubuntu 11.10, 64-bit

9Notably, this is not always necessary in FINROC. Consumers directly receive the buffers obtained from the v4l2 driver, for instance.
combination of features – most prominently efficient, lock-free, zero-copying, real-time intra-process communication, as well as convenient tooling for runtime construction. It is backward-compatible to MCA2 and preserves some of the latter's positive properties, coupled with a similar application style. Development started in late 2007. With virtually no dead-line, there are no compromises in the design or implementation that might have been caused by time constraints.

FINROC is possibly the first major robotic framework implemented in C++11. While this allows exploiting the new language features to create a safe and efficient implementation, compiler support is an issue.

As long as code reuse and interoperability are not significantly affected, we believe that diversity and competition with respect to robotic frameworks is positive – just as, for instance, different operating systems are.

FINROC is currently in beta state, and we plan to release it in the autumn of 2012. There is a GPL version of FINROC, but it is also possible to obtain commercial licenses. These comprise a smaller set of plugins and libraries. This way, we want to ensure that users contribute to the project. Should there be significant interest in FINROC, we would aim at organizing it in a foundation.

FINROC’s design was significantly inspired by other researchers publishing their insights – especially [11] and [4]. We hope to convey some interesting ideas in this paper, and are looking forward to feedback.

REFERENCES