Design Principles in Robot Control Frameworks

Max Reichardt, Tobias Föhst, Karsten Berns

Robotics Research Lab, Department of Computer Science
University of Kaiserslautern
Gottlieb-Daimler-Straße
67663 Kaiserslautern, Germany
{reichardt, foehst, berns}@cs.uni-kl.de

Abstract: Robotic software frameworks have critical impact on development effort and quality of robot control systems. This paper provides a condensed overview on the complex topic of robotic framework design. Important areas of design are discussed – together with design principles applied in state-of-the-art solutions. They are related to software quality attributes with a brief discussion on their impact. Based on this analysis, the approaches taken in the framework FINROC are briefly presented.

1 Motivation

Robotic software frameworks have critical impact on development effort and quality of robot control systems – especially when systems grow beyond a certain size. How to design such frameworks is an important question of research in order to make progress in robotics – as researchers and engineers in almost every research group spend a significant amount of time in software development and integration. Many authors have shared their views and insights on this topic, as well as presenting their approaches and implementations. These are fundamental contributions to understanding this complex challenge. It is, however, laborious to get a reasonably accurate overview over relevant activities in research groups around the world. Many notable approaches get only limited attention in the community and are not easy to find.

From literature research as well as from our own experience with the development of complex robot control systems and frameworks, we attempt to answer several questions in the scope of this paper:

- What are important areas when designing robotic frameworks?
- Which practices and principles are proposed for each of them?
- What is their impact on software quality?

Before implementing the FINROC [RFB13] framework[1], we investigated and evaluated

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

possible solutions for each of these areas in a systematic design process. With the experience gained from using FINROC in several projects and continuously improving it for the first public release, we decided to do an updated and more profound analysis again – considering also very recent work.

2 Design Principles

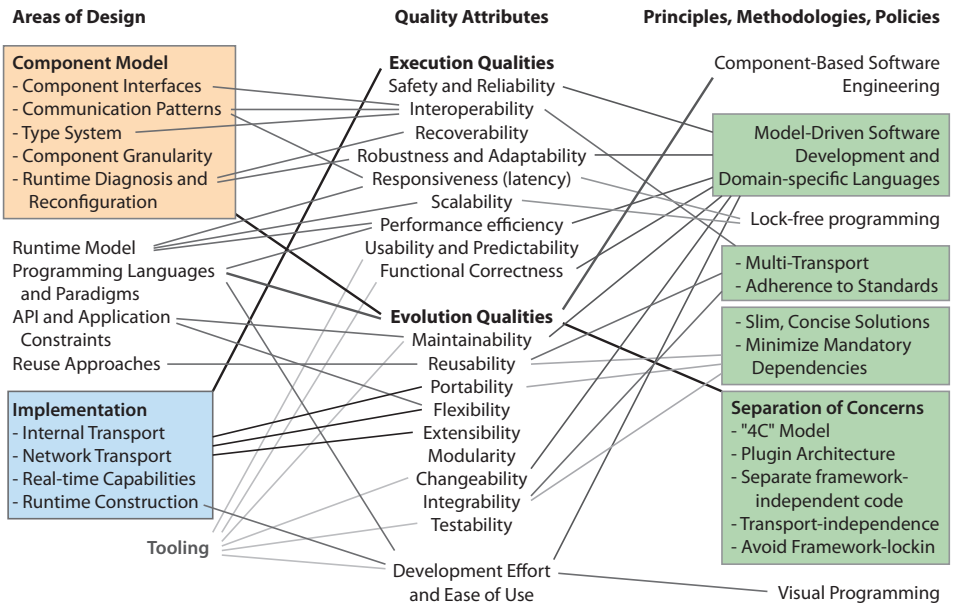


Figure 1: Overview on the complex topic of robotic framework design

Figure 1 lists quality attributes that we consider especially relevant across a wide range of control systems for service robots [RFB13]. Furthermore, a selection of important areas in robotic framework design is presented – as well as a range of design principles, methodologies, and policies proposed in literature. These design decisions and principles have an impact on many quality attributes of robot control systems. The figure illustrates major relations. In order to keep it clear, this is a very limited selection per item. The remainder of this chapter discusses many of these areas, principles, and relations.

2.1 System Decomposition

For system decomposition, all popular robotic frameworks follow a modular approach – aiming at reusable software artifacts that applications are constructed from. “It is both

desirable and necessary to develop robotic software in a modular fashion without sacrificing performance" [Bru07]. Robot controls commonly consist of software entities that encapsulate algorithms, hardware access etc. Depending on the framework, these application building blocks are called "components", "nodes" or "modules". The term "component" is used in the remainder of this paper. Typically, components can be connected in a network-transparent way to easily create distributed applications.

Component-Based Software Engineering (CBSE) is often named as primary approach and some authors propose targeting a component market for robotics [BKM+07, SSL12]. There are well-defined component models that are independent from the underlying implementation [ASK08, SSL12] (fig. 2a). A notable example are "Robotic Technology Components (RTC)" which are an OMG standard [Obj12]. OpenRTM-Aist is an open-source implementation by the original authors [ASK08], while e.g. Gostai RTC[2] is a commercial implementation by the developers of Urbi [Bai07]. Most other frameworks have component models that are more or less tied to a specific implementation. However, some are independent of the middleware that is used when instantiating the components (*transport-independence*, see fig. 2b) – such as Orocos [Soe06, SB11] or GenoM3 [MPH+10]. Other solutions do not have an explicit component model at all. As Wienke et al. [WW11] argue, this can have advantages as well. All these approaches can be found in state-of-the-art frameworks, and the choice has an impact on, especially, the evolution qualities of implemented systems.

Figure 2 illustrates further cases. Examples for *(c)* are MCA2 [SAG01] or Microsoft Robotics Developer Studio[3]. Orca 2 [BKM+07] is an example for *(d)*. Based on a professional third-party middleware[4], it is sufficient to use this middleware in order to communicate with a robot control using any supported programming language.

Unlike other solutions, FINROC *(e)* includes several component types that can be added via plugins as required. The support of third-party component models is planned. Furthermore, FINROC is network transport-independent.

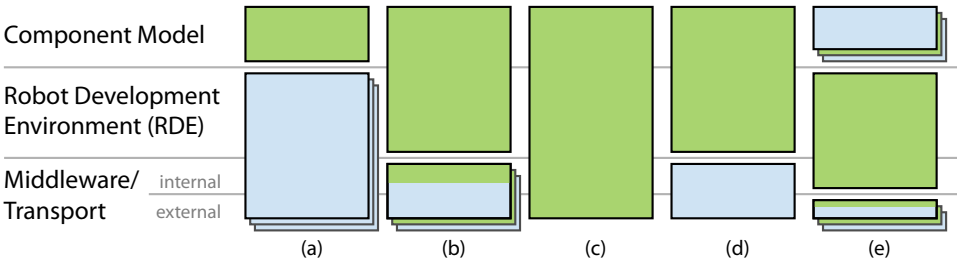


Figure 2: Overview on framework types. Blue indicates third-party artifacts.

[2] http://www.gostai.com/products/rtc
[3] http://www.microsoft.com/robotics
[4] ZeroC ICE in the case of Orca 2 [Hen04]

2.1.1 Component Interfaces

There are two major styles of interfaces used in robotic components: *data flow* and *services* (or *control flow*). Component interfaces typically consist of a set of communication endpoints called *ports* (sometimes also "publishers" and "subscribers"). Data ports publish and consume data – either directly connected forming data-flow graphs, or indirectly communicating via topics. Service ports provide and use interfaces with remote procedure calls or synchronous transactions as known from web services or CORBA. Most frameworks support both.

OPRoS [JLJ+10] has additional port types for events. Other frameworks including MCA2 and FAWKES [NFBL10] provide native support for blackboards – network-transparent shared memory.

Data flow graphs are simple and a natural fit especially for lower-level control loops. The order of execution is inherent. 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. However, for complex interaction patterns, data flow graphs are not appropriate. Supporting only services, on the other hand, can lead to many marginally different, incompatible interfaces, which hinders reuse. The developers of the Player Project discuss this difficulty [VGH03] and the necessity of introducing standard interfaces. On the other the hand, the numerous data types used in the ROS community show that this problem can also occur with data ports.

Depending on the framework, data ports differ in supported communication patterns. Common patterns are what Schlegel et al. [SSL12] call "push newest" and "push timed" in SmartSoft. The former corresponds to subscription types "New" and "Flush" in OpenRTM-aist, the latter to "Periodic". Many frameworks support (only) one of these patterns.

- "Push timed" pushes data at a fixed rate. This behavior is typically expected from e.g. ROS nodes.
- "Push newest" pushes new data to subscribers as soon as it is available. As different components might access ports at different rates, most frameworks provide optional FIFO buffers for this pattern.

SmartSoft furthermore supports "send" (one-way communication), "query" (two-way request) and "event" (asynchronous notification) [SSL12]. Instead of providing separate port types for services and events, these patterns are utilized.

Design decisions on component interfaces influence the flexibility of a framework and its ease of use. A lack of suitable interface patterns for certain use cases leads to workarounds that are detrimental to maintainability and possibly also to performance of systems.

In FINROC, port types are provided by optional plugins. Currently, there are plugins for data ports, service ports and blackboards. In a way, this makes the FINROC component model extensible. Data ports support switching between push (newest) and pull strategy at runtime. The latter corresponds to "query" in SmartSoft. Currently, this feature is used in tools in order to reduce bandwidth requirements.

2.1.2 Type System

Which kind of data types to allow in component interfaces is another central question. While some frameworks use an IDL (e.g. ROS, OpenRTM-aist), others allow native C++ types that meet certain criteria – e.g. Orocos requires data types to be copyable. ROS, which is currently the most wide-spread solution in research, provides a simple custom IDL. Most other IDL-based solutions use a third-party IDL – for instance, the Ice IDL in Orca 2 or the OMG IDL in OpenRTM-aist and a subset in GenoM3. [WNW12] contains a brief overview of IDLs relevant in robotics.

Frameworks that allow native data types in components need to know how to serialize them in order to create distributed systems. It is good practice to allow framework-specific serialization to be defined without modifying those types. C++ operator overloading or *traits* are suitable mechanisms for achieving this. This way, classes from framework-independent libraries such as, for instance, the PCL (Point Cloud Library)[5] can be used directly. Notably, this use of domain types reduces overhead for data conversion.

Being able to use specified data types in any supported programming language is a major advantage of IDLs. Furthermore, many IDLs are standardized and well-defined. However, an extra toolchain is required for code generation. Supporting native types, on the other hand, allows exploiting the full power of for instance C++ for data type definition, which can be more flexible and efficient. Data types generated by IDLs may be used as well.

Making applications based on two different frameworks interoperable is easiest if they use the same data types or at least the same IDL. Wienke et al. [WNW12] present a solution for interoperability with different IDLs and discuss the difficulties involved.

Again, the choice of type system has an impact on a framework's flexibility and ease of use. Furthermore it determines interoperability and possibly efficiency of systems – as well as reusability, portability and integrability of individual components.

In FINROC, native C++11 data types are used in port interfaces. Notably, they do not need to be copyable. Framework-specific serialization is defined via operator overloading.

2.1.3 Component Granularity

Another interesting question is, which granularity components should and may have. According to Ando et al. [ASK08], "various" component sizes need to be supported – with data flow ports mainly being used by more fine-grained components and service ports by the more coarse-grained ones. Small components can be easier to reuse. Furthermore, some frameworks support creating *composite components* containing and encapsulating a set of simple components - e.g. OpenRTM-aist, OPRoS or MCA2.

Generally, we believe that application developers can themselves decide best on a suitable granularity for their reusable artifacts. A framework should not impose limits in this respect. For relatively small components to be feasible, development and runtime overhead need to be low.

[5] `http://pointclouds.org`

In our research on behavior-based networks, components performing simple mathematical operations are sometimes necessary in order to join behavior components together. Executing them in a separate thread or even process would be a waste of resources. Due to this research, FINROC is suitable for a high number of components – possibly thousands. Composite components ("groups") are supported in order to keep applications structured.

2.1.4 Runtime Model

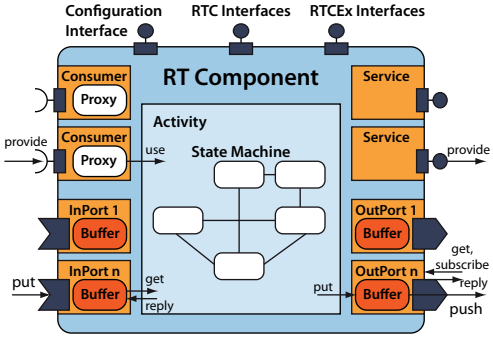
A framework's runtime model (see [Nes07]) comprises whether execution is synchronous or asynchronous and how it is triggered – periodically, or by events. Furthermore, it defines how threads are mapped to components. At the one extreme, each component has its own thread – or even process, as in ROS[6] or Orca 2. Executing components in different processes can increase a system's robustness by preventing memory corruption from other components. As Hammer et al. [HB13] show, this feature can be maintained with a highly efficient shared memory implementation. On the other hand, they mention the importance of avoiding *thread clutter* from too many running threads.

In other frameworks (e.g. OpenRTM-aist, MCA2), components can be assigned to threads. For instance, this allows the execution of "tightly coupled RTCs in a single (real-time) thread" [ASK08]. As mentioned, the option of executing multiple components by the same thread is necessary for small components to be feasible. How many processes to distribute components to is a trade-off between robustness and efficiency.

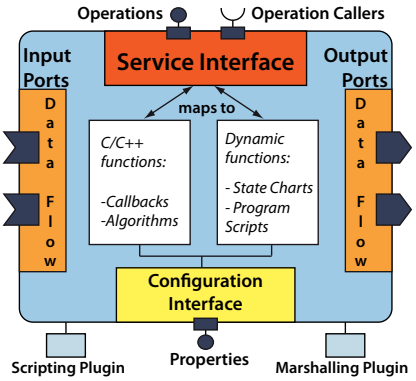
Synchronous implementations are typically simpler than asynchronous ones, but the latter often lead to lower latencies – especially when many threads are involved. MCA2 and the Player Project are examples for frameworks that trigger execution periodically only. This has the advantage that it is simple and predictable. However, "it imposes an average delay of a half cycle on all data [...]" [VGH03]. Wienke et al. [WW11] present an entirely event-based solution.

As Nesnas [Nes07] states, "robotic systems require both synchronous and asynchronous execution of different activities". Thus, a general-purpose framework should support both concepts.

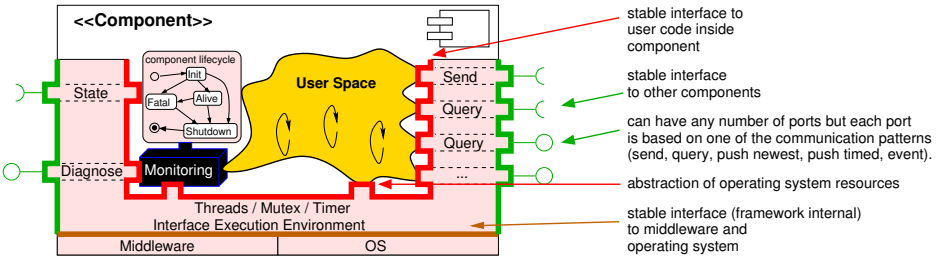
Design choices in the runtime model have an impact especially on responsiveness, scalability, robustness and efficiency of a system. In FINROC we adopted the approach from MCA2 to assign multiple components to a thread. There is support for periodic as well as event-triggered execution. The latter may be triggered by incoming data port values or service calls. Apart from that, FINROC's intra-process communication is highly optimized, favoring putting many components into one process.



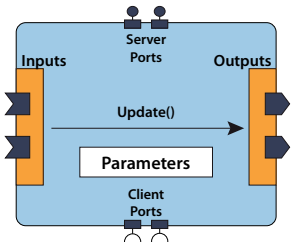
(a) RT component



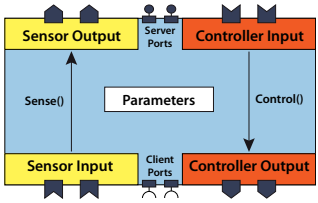
(b) Orocos component ("TaskContext")



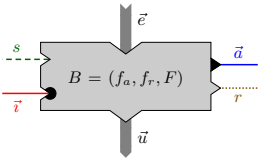
(c) SmartSoft component



(d) FINROC standard component



(e) FINROC "SenseControlModule"



(f) FINROC iB2C behavior

Figure 3: Components in different frameworks

2.1.5 Component Models

Figure 3 illustrates component models used in different frameworks[7]. Notably, they have similarities. Data flow ports are a central element in all of them. In SmartSoft, services

[6] It is possible to use "nodelets" in ROS. This, however, requires the adaption of the components.

[7] (a) and (b) have been adapted to a similar style as (d). The author's illustration of (a) can be found in [ASK08], (b) on `http://people.mech.kuleuven.be/~orocos/pub/documentation/rtt/v2.6.x/doc-xml/orocos-components-manual.html`. (c) was kindly provided by Christian Schlegel and Alex Lotz.

are realized using the same ports with different communication patterns. The other frameworks have separate port types for this purpose. Some kind of configuration interface is a common element as well[8]. FINROC plugins furthermore provide the "SenseControlModule" – a component type we learned to appreciate from MCA2– and a behavior component. The latter only uses data ports. Its semantics are explained in [AKRB13]. FINROC components are derived from the same base class, but have different kinds of interfaces, execution semantics, and e.g. connection constraints.

2.2 Model-Driven Software Development

Model-Driven Software Development (MDSD) and *Domain-Specific Languages (DSLs)* are topics which have gained increased research interest in recent years. Schlegel et al. [SSL12] elaborately propose adopting MDSD approaches in robotics – as well as "model-centric robotic systems". Consequently, SmartSoft is based on model-driven concepts: Components are implemented in a platform-independent way. The MDSD toolchain can then be used to generate instances for specific SmartSoft implementations. Currently, there are two such implementations based on CORBA and ACE. Similarly, RT components (see chapter 2.1) are implemented based on a platform-independent component model and may be used with any implementation of the OMG standard. GenoM3 ("Generator of Modules") is another notable approach that generates middleware-specific instances of middleware-independent component artifacts. It allows, for instance, generating components for use in ROS. The workflow is illustrated fig. 4 from [MPH+10][9].

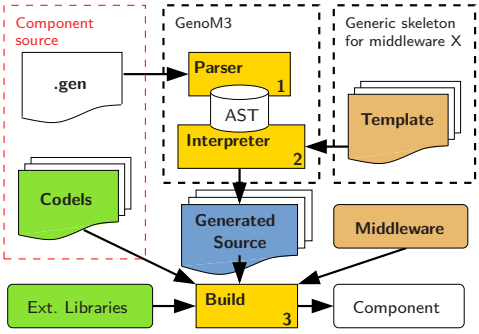


Figure 4: Overview of the GenoM3 workflow

Ortiz et al. [OSA+13] created the model-based toolchain "C-Forge". Instead of generating code, a model loader directly interprets the models. It instantiates and executes components accordingly.

Model-based approaches are also suitable for creating hard real-time applications, as sev-

[8]The *parameters* interface of the iB2C component is not illustrated.
[9]The figure was kindly provided by Anthony Mallet.

eral authors write [FHC97, SSL12, OSA+13]. Apart from generating code, temporal models can be used to perform real-time schedulability analysis. Cheddar[10] is a popular tool for this purpose.

Steck et al. [SS11] show that design-time models can also be exploited for reasoning at runtime.

Domain-Specific Languages (DSLs) are a related topic. They are special-purpose languages targeting specific problem domains, enabling the concise expression of relevant artifacts. According to Bäuml [Bäu13], there are many areas in robotics that "[...] would benefit from a specialized language which supports the respective abstractions also syntactically and so helps to avoid a lot of boiler plate code" – examples including the "kinematic/dynamic/geometrical description of a robot" or a "language for complex and concurrent state machines". Most DSLs are completely independent from a specific robotic framework. URBIScript [Bai07] is a domain-specific scripting language for robotics that is included in the URBI framework. It supports finite state machines, parallelism and concurrency in a sophisticated way. aRDx [Bäu13] is a new framework particularly suitable for integration of DSLs. Implemented in the scripting language Racket[11], it allows to "build whole towers of languages". Due to maintenance effort, Orocos recently switched from its own scripting language RTT to a Lua-based internal, real-time DSL [KSB10]. Its primary use case are hierarchical state machines.

Models can be used to verify certain properties of a system (model checking). This way, they can have an impact on various execution qualities of a system such as safety, responsiveness, robustness or functional correctness. Code generation provides chances to improve efficiency. Regarding evolution qualities, using models can contribute to e.g. maintainability and changeability, as well as overall development effort.

However, MDSD and DSLs can also have drawbacks. Especially the development of a new, non-trivial meta model or DSL together with sufficiently mature code transformation and debugging facilities, requires a huge amount of effort – possibly much more than it saves in the end. Immature solutions are detrimental with respect to maintainability and changeability of systems. Even mature code transformation and DSLs can complicate debugging. The former add an additional toolchain, often depending on a platform such as Eclipse. Efforts with a small user base might be discontinued.

In the mandatory parts of the FINROC core, we deliberately minimized the amount of generated code to optional string constants. A developer has the option to make the complete system behavior evident from plain, versioned C++11 code. Optionally, structure of applications and composite components can be stored in a simple XML file format that can be generated, interpreted and changed by external tools as well as the FINROC runtime environment. In the context of behavior-based networks, model transformation and model checking approaches were realized [AKRB13]. An experimental plugin adds support for the URBIScript language.

[10] `http://beru.univ-brest.fr/~singhoff/cheddar`
[11] `http://racket-lang.org`

2.3 Separation of Concerns

With system complexity and maintainability being major issues in robotics, *separation of concerns* is an important design principle. In this context, especially the Orocos authors propose the "4C Model" for robotic components (as e.g. mentioned in [SB11]) – a variation of the concept originally introduced by Radestock et al. [RE96], as Schlegel et al. explain in an introduction to these topics [SSL12]. It proposes "[...] dividing the specification and implementation of distributed systems into four parts – communication, computation, configuration and coordination" [RE96].

Having encountered maintainability issues, Makarenko et al. [MBK07] discuss this topic regarding the Orca 2 framework and propose a clear separation of concerns with respect to *(1) Driver and Algorithm Implementations*, *(2) Communication Middleware*, and the *(3) Robotic Software Framework*. This good practice of separating framework-independent code from framework-dependent code is increasingly promoted [Roc, QCG+09, RFB13]. Not being tied to any framework, libraries such as OpenCV[12] or the PCL are (re)used in research institutions around the world.

Furthermore, code complexity and maintainability are correlated. Simple, independent artifacts with compact source code require less maintenance effort and are less likely to contain programming errors. Makarenko et al. [MBK07] discuss the many benefits of frameworks having a slim and clearly structured code base – especially regarding development and maintainability of a framework itself. Some authors explicitly target *minimalism* [HB13].

A clear separation of concerns is beneficial with respect to virtually all evolution qualities of software systems – maintainability in particular. Reusability and portability of software artifacts are also increased significantly.

In consequence, we implemented FINROC in a slim and highly modular way [RFB13]. As illustrated in fig. 5[13], it consists of many small independent software entities. RRLIBs are framework-independent libraries. Functionality that is not needed in every application is generally implemented in optional plugins. This way, FINROC can be tailored to the requirements of an application. Plugins can contain almost anything, including communication port types (*data_ports*, *rpc_ports*, *blackboard*), component types (*structure*, *ib2c*), network transports (*tcp*, *ros*), or support for DSLs (*urbiscript*). With only communication plugins, FINROC could be configured as a plain middleware.

2.4 Programming Languages

The choice of programming language has a major impact on performance efficiency and on virtually all evolution qualities of a system. Suitability for real-time implementations, development effort, and the availability of reusable software artifacts are further factors

[12] `http://opencv.org/`

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

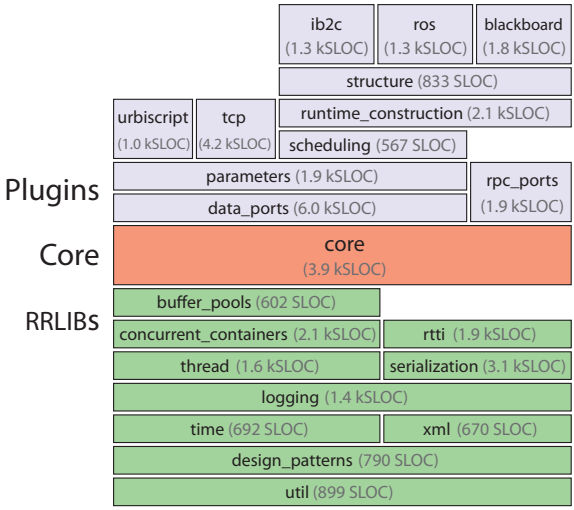


Figure 5: FINROC's modular core with a selection of plugins

to consider. Most popular robotic frameworks have implementations in C or C++ – a good choice with respect to many of these aspects. Increased development effort and its difficulty level are arguably drawbacks. ROS, for instance, is partly implemented in Python. Notably, aRDx has mainly been implemented in the scripting language Racket (see chapter 2.2). According to Bäuml [Bäu13], "Racket performs only about 5x slower than C/C++ and about 10x faster than Python". Thus, small parts of performance-critical functionality are implemented in C/C++. Apart from this, many advantages for the robotics domain are listed – including maintainability, productivity, and the embedding of DSLs.

Various frameworks support multiple programming languages for the development of robot control systems – e.g. ROS or openRTM-aist. Java and Python are often an option. Microsoft Robotics Developer Studio allows any .NET language to be used. CLARAty is explicitly separated into a functional and a decisional layer. The latter is programmed in LISP. As Klotzbücher et al. [KSB10] point out, using embedded scripting languages improves robustness of a system, as errors in script code do not affect unrelated components in the same process.

Multi-language approaches can be realized by implementing the complete framework in every supported language or by creating bindings in these languages to a single implementation – often based on C++. The former is more portable, but also leads to increased maintenance effort. Usually, frameworks that do not use an IDL to specify data types also do not support several general-purpose programming languages (see chapter 2.1.2).

With respect to FINROC, we decided to create full, native implementations in C++11 and Java – initially using an automated co-development process. We chose C++11 because its features support making implementations safer, shorter and more efficient compared to C++03. It, however, requires a modern C++ compiler – somewhat limiting portability. Data types required in both languages need to be implemented in both unless a type's string or XML representation is sufficient. The Java version is suitable for Android platforms.

2.5 Implementation

In the mobile robotics domain, software performance is a critical factor – as this determines required computing resources and battery power. Regarding frameworks, a key issue is sharing data among connected software components and threads. The components can be located in the same process (*intra-process*), on the same computing node (*inter-process*), or on different computing nodes (*inter-host*). As Nesnas [Nes07] points out, "an application framework must pay particular attention to avoiding unnecessary copying of data [...]".

Efficiency also influences latency and scalability – imposing limits on the maximum number of components that are feasible (see chapter 2.1.3). Locking can be an even bigger issue with respect to latency and scalability. Locking buffers exclusively from different components quickly causes significant, varying delays.

Support for meeting hard real-time requirements in component interaction is another important feature – especially for low-level control loops and safety-critical systems. Not being limited to a single component for real-time tasks increases reusability – e.g. with separate components for accessing sensors and actuators in control loops. Several frameworks support this, including OpenRTM-aist, Orocos, OPRoS, SmartSoft, GenoM3, aRDx and MCA2. For real-time implementations, unboundedly varying delays must be avoided. Lock-free implementations are advantageous in this respect.

Zero-copy transport mechanisms, typically use either ring buffers or pools of buffers with reference counters. The former is simpler to implement, while the latter is more flexible. Hammer et al. [HB13] present an efficient implementation based on ring-buffers that is zero-copying even for inter-process communication. In [RFB13], we explain FINROC's efficient, lock-free, zero-copy intra-process transport based on buffer pools.

Due to the modular application style, using a framework will always induce computational overhead compared to a perfectly engineered monolithic solution. However, frameworks such as Orocos show that computational overhead can be low, despite a relatively loose coupling. In practice, as soon as it comes to buffer management or multithreading, we often observe that framework-based solutions actually outperform custom standalone code – sometimes drastically. This is due the fact that efficient, lock-free buffer management is complex to implement.

High bandwidth, low latency, low computational overhead, robustness, and support for quality of service (QoS) are desirable attributes of a network transport used for distributed robotic systems. Interoperability and security can be further requirements. As discussed in chapter 2.1, several frameworks are independent from a specific network-transport. This is beneficial with respect to varying requirements of applications. 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 [Obj98] or DDS[14]. ICE [Hen04] is a popular middleware product not based on these standards. OpenRTM-aist, Orocos, OPRoS and SmartSoft are frameworks with a CORBA implementation, for

[14] `http://portals.omg.org/dds`

instance. Several frameworks are interoperable with ROS – including FINROC [ARHB13]. Some transport-independent frameworks such as Orocos and GenoM3 can use the ROS transport for all components.

3 Conclusion

There is a lot of interesting research in the context of robot control frameworks. The contribution of this paper is to give an overview on activities in this broad scope, present important design areas and principles, as well as relating them to software quality attributes. Numerous robotic frameworks have been developed. The projects referenced in this paper are only a small subset of relatively recent work. With FINROC we believe to have made an interesting contribution to this research. Regarding future activities, questions of suitable measures in a framework to support or even guarantee certain quality attributes of robot control systems are a relevant direction of research – as we discuss in [RFB13].

Acknowledgments Funding by the German Ministry of Education and Research (grant 01IC12S01W, Software-Cluster project SINNODIUM) is gratefully acknowledged.

References

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

[ARHB13] Michael Arndt, Max Reichardt, Jochen Hirth, and Karsten Berns. Requirements for Interoperability and Seamless Integration of Different Robotic Frameworks. In Davide Brugali, editor, *Proceedings of the eighth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII), in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*, pages 38–40, Karlsruhe, Germany, May 2013.

[ASK08] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In Stefano Carpin, Itsuki Noda, Enrico Pagello, Monica Reggiani, and Oskar von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin / Heidelberg, 2008.

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

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

[BKM+07] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan B. Williams, and Anders Orebäck. Orca: A Component Model and Repository. In Brugali [Bru07].

[Bru07] Davide Brugali, editor. *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*. Springer - Verlag, Berlin / Heidelberg, April 2007.

[FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *In International Conference on Intelligent Robots and Systems*, pages 842–848, Grenoble, France, September 7-11 1997.

[HB13] Tobias Hammer and Berthold Bäuml. Raw Performance of Robotic Software Middleware: A Comparison and aRDx's New Realtime Communication Layer. In Davide Brugali, editor, *Proceedings of the eighth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII), in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 2013.

[Hen04] Michi Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[JLJ+10] Choulsoo Jang, Seung-Ik Lee, Seung-Woog Jung, Byoungyoul Song, Rockwon Kim, Sunghoon Kim, and Cheol-Hoon Lee. OPRoS: A New Component-Based Robot Software Platform. *ETRI Journal*, 32:646–656, 2010.

[KSB10] Markus Klotzbücher, Peter Soetens, and Herman Bruyninckx. OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. In *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPA)*, pages 284–289, Darmstadt, Germany, November 15-16 2010.

[MBK07] Alexei Makarenko, Alex Brooks, and Tobias 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.

[MPH+10] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and François Felix Ingrand. GenoM3: Building middleware-independent robotic components. In *ICRA*, pages 4627–4632, 2010.

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

[NFBL10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design Principles of the Component-Based Robot Software Framework Fawkes. In *Proc. of Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Lecture Notes in Computer Science, Darmstadt, Germany, 2010. Springer.

[Obj98] Object Management Group, Inc., Framingham, Massachusetts, USA. *The Common Object Request Broker: Architecture and Specification – Version 2.2*, July 1998.

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

[OSA+13] Francisco J. Ortiz, Francisco Sánchez, Diego Alonso, Francisca Rosique, and Carlos C. Insaurralde. C-Forge: a Model-Driven Toolchain for Developing Component-Based

Robotics Software. In Davide Brugali, editor, *Proceedings of the eighth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII), in conjunction with the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 2013.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009.

[RE96] Matthias Radestock and Susan Eisenbach. Coordination in Evolving Systems. In *International Workshop on Trends in Distributed Systems (TreDS '96): CORBA and Beyond*, pages 162–176, Aachen, Germany, October 1-2 1996.

[RFB13] Max Reichardt, Tobias Föhst, and Karsten Berns. On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems. In *Proceedings of the 7th International Workshop on Software Quality and Maintainability (SQM), in conjunction with the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, Genoa, Italy, March 5 2013.

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

[SAG01] Kay-Ulrich Scholl, Jan Albiez, and Bernd Gassmann. MCA- An Expandable Modular Controller Architecture. In *3rd Real-Time Linux Workshop*, Milano, Italy, 2001.

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

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

[SS11] Andreas Steck and Christian Schlegel. Managing execution variants in task coordination by exploiting design-time models at run-time. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 2064–2069, 2011.

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

[VGH03] R. Vaughan, B. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, pages 2121–2427, Las Vegas, Nevada, USA, October 27-31 2003.

[WNW12] Johannes Wienke, Arne Nordmann, and Sebastian Wrede. A Meta-Model and Toolchain for Improved Interoperability of Robotic Frameworks. In *SIMPAR2012 - SIMULATION, MODELING, and PROGRAMMING for AUTONOMOUS ROBOTS*. Springer Heidelberg Berlin, 2012.

[WW11] Johannes Wienke and Sebastian Wrede. A middleware for collaborative research in experimental robotics. In *System Integration (SII), 2011 IEEE/SICE International Symposium on*, pages 1183–1190, 2011.