Special Issue

Max Reichardt*, Tobias Föhst, and Karsten Berns

An overview on framework design for autonomous robots

Abstract: Robotic software frameworks have major 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.

Keywords: Autonomous mobile robots, framework design, software quality.

ACM CCS: Software and its engineering → Software creation and management → Designing software → Software design engineering

DOI 10.1515/itit-2014-1065
Received June 16, 2014; revised December 17, 2014; accepted January 6, 2015

1 Motivation

Software frameworks have major impact on development effort and quality attributes of robot control systems – especially when systems grow beyond a certain size. Hence, how to design such frameworks is an important question of research in order to make progress in robotics. Researchers and engineers 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 approaches and implementations to cope with this complex challenge. Numerous solutions following different design philosophies have been developed and it is time-consuming to get an accurate overview on the state of the art. Many notable approaches get only limited attention in the community and are not easy to find.

From extensive literature research as well as from our own experience with the development of complex robot control systems and frameworks, we attempt to provide a condensed overview in this paper – with a focus on the following questions:
– What are important and central areas of design?
– Which practices and principles are proposed?
– What is their impact on software quality?

Derived from this overview, the design choices we have taken in the FINROC [23] framework[1] are briefly presented and discussed. They were systematically evaluated with respect to their impact on relevant quality attributes of robot control systems and of the framework itself.

2 Design aspects and principles

Figure 1 lists quality attributes that we consider especially relevant across a wide range of control systems for service robots [23]. 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. Important relations are illustrated. In order to keep the figure clear, this is a very limited selection per item. In this chapter, many of these areas, principles, and relations are discussed.

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” [6]. Robot controls commonly consist of soft-

***Corresponding author: Max Reichardt,** University of Kaiserslautern, e-mail: reichardt@cs.uni-kl.de
Tobias Föhst, Karsten Berns: University of Kaiserslautern

1 http://www.finroc.org

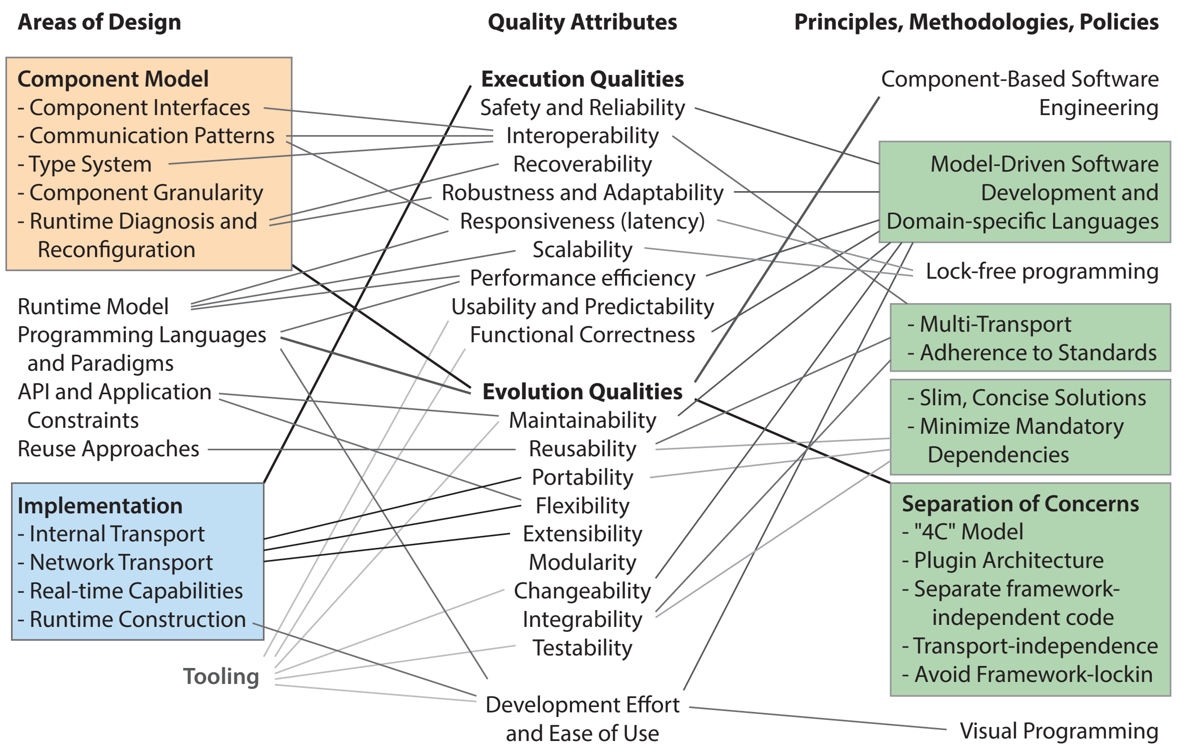


Figure 1: Overview on the complex topic of robotic framework design.

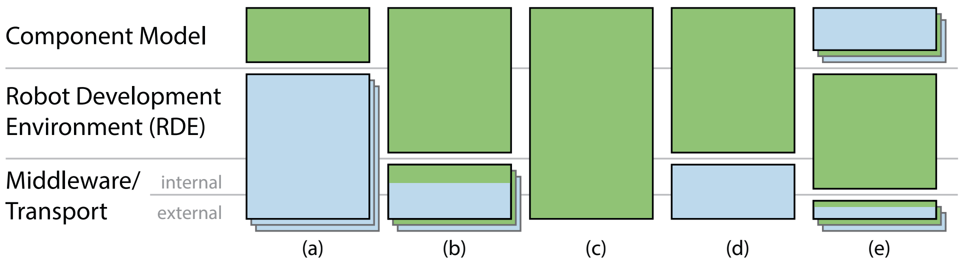


Figure 2: Overview on areas that frameworks cover (green). Blue indicates third-party artifacts.

ware entities that encapsulate e. g. algorithms or hardware drivers. Depending on the framework, these application building blocks are called “components”, “nodes” or “modules”. For simplicity, the term “component” is used in 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 [5, 24]. There are well-defined, formal component models that are independent from the underlying implementation [1, 24] (Figure 2a). A notable example are “Robotic Technology Components” (RTC) which are an OMG standard [19]. OpenRTM-Aist is an open-source implementation by the original authors [1], while e. g. Gostai RTC[2] is a commercial implementation by the developers of Urbi [3]. 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-* or *middleware-independence*, see Figure 2b) – such as Orocos [27, 28] or GenoM3 [15]. Other solutions do not have an explicit component model at all. As Wienke et al. [34] 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.

2 http://www.gostai.com/products/rtc

Figure 2 illustrates further cases. Examples for (c) are MCA2 [25] or Microsoft Robotics Developer Studio[3]. Orca 2 [5] and ArmarX [30] are examples for (d). Based on a professional third-party middleware[4], using this middleware is sufficient 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 – including third-party component models e. g. from mca2. There is experimental support for RT Components. Apart from that, Finroc is transport-independent regarding inter-process communication.

2.1.1 Component interfaces

Component interfaces typically consist of a set of communication endpoints called *ports*. There are two major types of such ports: *data flow* and *service* ports. Most frameworks support both. Data flow ports publish and consume data – either directly connected forming data flow graphs (*point-to-point*), or indirectly communicating via topics. ROS [21] is well-known for the latter. Joyeux et al. discuss advantages of the former [12]. Service ports provide and use interfaces with remote procedure calls or synchronous transactions – as known from web services or CORBA.

Some authors propose to explicitly separate data flow from events [31]. OPRoS [11], for instance, has additional port types for events. Other frameworks including mca2 and FAWKES [17] provide native support for blackboards – network-transparent shared memory.

Data flow graphs are simple and a natural fit especially for lower-level control loops. Designing interfaces based on data ports is typically straightforward. However, for complex interaction patterns, data flow is 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 [32] and the necessity of introducing standard interfaces. The numerous data types used in ROS 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. [24] 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) [24]. Instead of providing separate port types for services, these patterns are used.

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. Data ports support switching between push (newest) and pull strategy at runtime. The latter corresponds to "query" in SmartSoft.

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. ROS, currently the most wide-spread solution in research, provides a simple custom IDL. In most other IDL-based solutions, a third-party IDL is used – e. g. the ICE IDL in Orca 2 or the OMG IDL in OpenRTM-aist. [33] contains a brief overview on 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 Point Cloud Library 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. 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

3 http://www.microsoft.com/robotics

4 both frameworks use ZeroC ICE [10]

power of e. g. C++ for data type definition, which can be more flexible and efficient. With this design choice, 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. [33] 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 ports. Notably, types 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. [1], "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. For relatively small components to be feasible, development and runtime overhead need to be low.

Furthermore, some frameworks support creating *composite components* containing and encapsulating a set of components – e. g. OpenRTM-aist, OPRoS, or MCA2.

Generally, we believe that developers can themselves decide best on a suitable granularity for their reusable software artifacts. A framework should not impose limits in this respect. In our research on behavior-based networks, for instance, systems with more than thousand components have been created. Executing them in separate threads or even processes would not be feasible. Finroc was designed to be suitable for high numbers of components. Composite components ("groups") are supported in order to keep applications structured.

2.1.4 Runtime model

A framework's runtime model (see [16]) 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[5] or Orca 2. Notably, executing components in different processes can increase systems' robustness by preventing memory corruption from other erroneous components. This feature can be preserved if processes use shared memory for data exchange, as Hammer et al. [9] show. On the other hand, they discuss 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" [1]. 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 including factors such as 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 [. . .]" [32]. Wienke et al. [34] present an entirely event-based solution. Nesnas [16] states that "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.

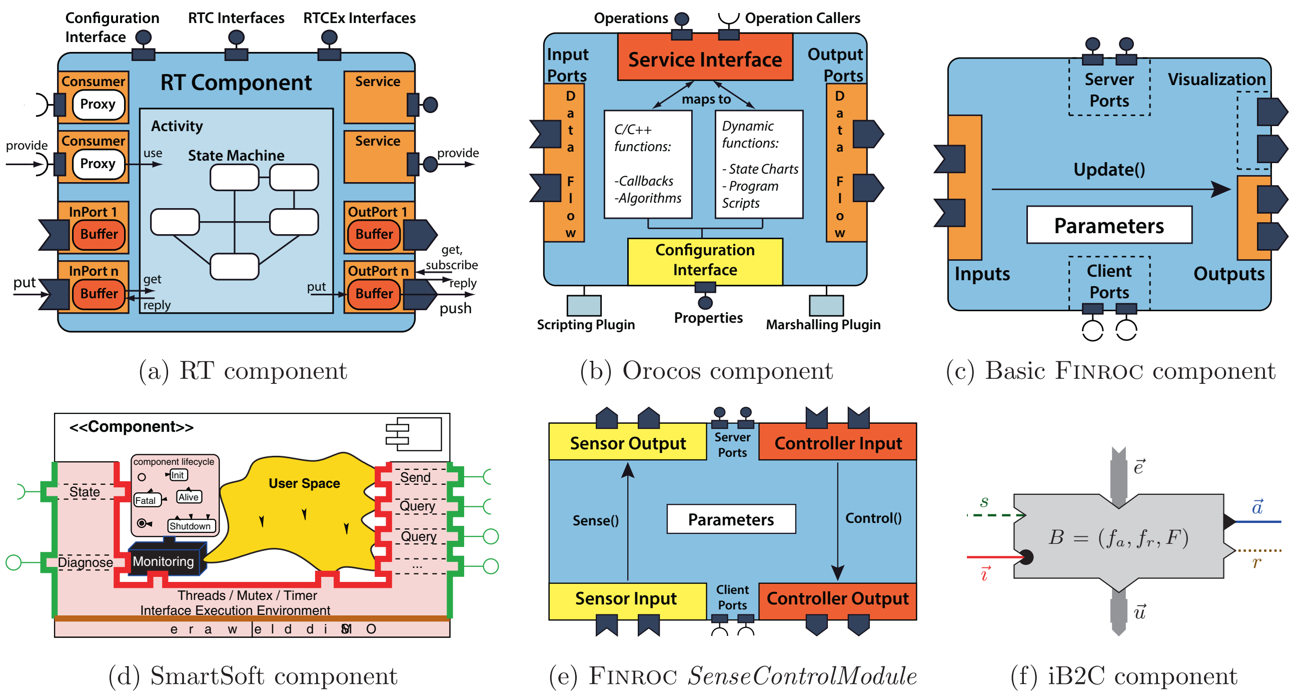
In Finroc, multiple components can be assigned to a thread. Both periodic and event-driven execution are supported.

2.1.5 Component models

Figure 3 illustrates component models used in different frameworks[6]. Notably, they have similarities. Data flow ports are central elements. In most cases, there a separate port types for realizing services. In SmartSoft, the available port communication patterns are used for this pur-

5 It is possible to use "nodelets" in ROS. This, however, requires adaptation of the components.

6 (a) and (b) have been adapted to a similar style as (c). The original authors' illustration of (a) can be found in [1], (b) on http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-components-manual.html. (d) was kindly provided by C. Schlegel and A. Lotz.



(a) RT component (b) Orocos component (c) Basic FINROC component

(d) SmartSoft component (e) FINROC *SenseControlModule* (f) iB2C component

Figure 3: Components in different frameworks.

pose. Some kind of configuration interface is a common element as well. Bruyninckx et al. [7] captured these commonalities in a meta model.

FINROC supports multiple component types via plugins. They are derived from a common base class, but have different kinds of interfaces, execution semantics, and e. g. connection constraints. The “SenseControlModule” was adopted from MCA2. The behavior components use only data ports. Their semantics are explained in [2].

2.2 Model-driven software development

Model-Driven Software Development (MDSD) and *Domain-Specific Languages (DSLs)* are topics which gained increased research interest in recent years (e. g. [7, 12, 18]). Schlegel et al. [24] 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 implementations. Currently, there are two such implementations based on CORBA and ACE. Similarly, there is a platform-independent model for RT components (see Section 2.1) for use with any implementation of the OMG standard. In practice, however, many existing RT components depend on the headers of OpenRTM-aist implementation. GenoM3

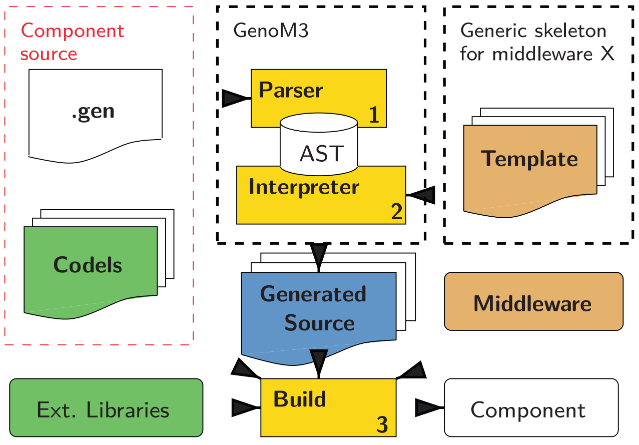


Figure 4: Overview of the GenoM3 workflow (from [15])[7].

(“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 in Figure 4.

Ortiz et al. [20] created the model-based toolchain “C-Forge”. Instead of generating code, a model loader interprets models and instantiates components accordingly.

7 The figure was kindly provided by Anthony Mallet.

Model-based approaches are also suitable for creating hard real-time applications, as several authors write [8, 20, 24]. Apart from generating code, temporal models can be used to perform real-time schedulability analysis. Cheddar[8] is a popular tool for this purpose.

Steck et al. [29] show that design-time models can be exploited for reasoning at runtime – an aspect arguably of increasing importance towards "cognitive" robotics [31].

Domain-Specific Languages (DSLs) are a related topic. They are special-purpose languages targeting specific problem domains, enabling the concise expression of relevant artifacts. Nordmann et al. [18] provide a survey on the many approaches. According to Bäuml [4], 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 independent from a specific robotic framework. URBIScript [3] is a domain-specific scripting language for robotics included in the URBI framework. It supports finite state machines, parallelism and concurrency in a sophisticated way. aRDx [4] is a new framework particularly suitable for integration of DSLs. Implemented in the scripting language Racket[9], 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 [13]. A central 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 [31]. Even mature code transformation and DSLs can complicate debugging. Notably, many solutions depend on the Eclipse platform [18].

The Finroc runtime environment can load and store application structure models (components, connections, and configuration) in XML files that may be generated, interpreted, and changed by external tools. In the context of behavior-based networks, model transformation and model checking approaches were realized [2]. Apart from that, adoption of existing approaches is intended. An experimental plugin adds support for the URBIScript language.

2.3 Separation of concerns

As system complexity and maintainability are challenges in robotics, *separation of concerns* is an important design principle. In this context, the Orocos and BRICS authors propose the "5C Model" as a best practice for robotic software development [7, 31] – keeping the 5 concerns *Communication*, *Computation*, *Coordination*, *Configuration*, and *Composition* fully separated in design and implementation. It is a variation of the "4C Model" originally introduced by Radestock et al. [22] – as Schlegel et al. explain in an introduction to these topics [24].

Having encountered maintainability issues, Makarenko et al. [14] 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 [21, 23]. Not being tied to any framework, libraries such as OpenCV or the Point Cloud Library 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. [14] 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* [9] or a *microkernel design* [21].

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.

8 http://beru.univ-brest.fr/~singhoff/cheddar

9 http://racket-lang.org

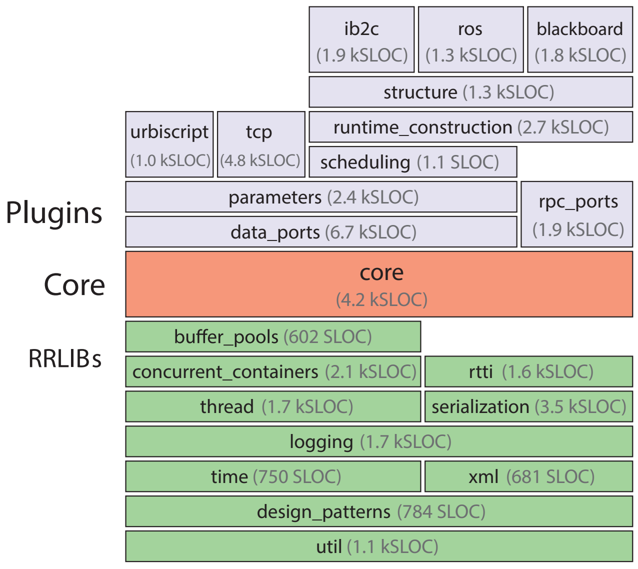


Figure 5: FINROC's modular core with a selection of plugins[10].

In consequence, FINROC was implemented in a slim and highly modular way [23]. As illustrated in Figure 5, it consists of many small 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. Notably, slim configurations can run without an operating system [26]. 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 to consider. Most popular robotic frameworks have implementations in 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 Section 2.2). According to Bäuml [4], “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. [13] 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 Section 2.1.2).

With respect to FINROC, we decided to create native implementations in C++11 and Java. Robot control systems are typically implemented in C++, while Java is used for tool support. 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. As Nesnas [16] points out, “an application framework must pay particular attention to avoiding unnecessary copying of data […]”. Components can be located in the same process (*intra-process*), on the same computing node (*inter-process*), or on different computing nodes (*inter-host*).

Efficiency also influences latency and scalability – imposing limits on the maximum number of components that are feasible (see Section 2.1.3). Locking can be an even bigger issue with respect to latency and scalability. Lock-

10 *Physical Source Lines of Code* (SLOC) were determined using David A. Wheeler's 'SLOCCount'

ing 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. [9] present an efficient implementation based on ring-buffers that is zero-copying even for inter-process communication. In [23], 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 induces computational overhead compared to a perfectly engineered monolithic solution. However, modern frameworks 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 Section 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 or DDS[11]. ICE is a popular middleware product not based on these standards. OpenRTM-aist, Orocos, OPRoS and SmartSoft are frameworks with a CORBA implementation, for instance. Several frameworks are interoperable with ROS – including FINROC. Notably, some transport-independent frameworks such as Orocos and GenoM3 can use the ROS transport for all components.

11 http://portals.omg.org/dds

3 Conclusion

There is a lot of important 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 [23].

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

References

1. N. Ando, T. Suehiro, and T. Kotoku. *A software platform for component based rt-system development: OpenRTM-Aist*. In S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pp. 87–98. Springer Berlin/Heidelberg, 2008.
2. C. Armbrust, L. Kiekbusch, T. Ropertz, and K. Berns. *Tool-assisted verification of behaviour networks*. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany, May 2013.
3. J.-C. Baillie. *Design principles for a universal robotic software platform and application to URBI*. 2nd National Workshop on Control Architectures of Robots (CAR'07), pp. 150–155, Paris, France, May–June 2007.
4. B. Bäuml. *One for (almost) all: Using a modern programmable programming language in robotics*. Proceedings of the eighth 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.
5. A. Brooks, T. Kaupp, A. Makarenko, S. B. Williams, and A. Orebäck. *Orca: A component model and repository*. In Brugali [6].
6. D. Brugali, editor. *Software Engineering for Experimental Robotics*, volume 30 of Springer Tracts in Advanced Robotics. Springer, Berlin/Heidelberg, April 2007.
7. H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. *The BRICS component*

model: A model-based development paradigm for complex robotics software systems. 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, March 2013.

8. S. Fleury, M. Herrb, and R. Chatila. *GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture.* International Conference on Intelligent Robots and Systems, pp. 842–848, Grenoble, France, September 1997.
9. T. Hammer and B. Bäuml. *The highly performant and realtime deterministic communication layer of the aRDx software framework.* Proceedings of the 16th International Conference on Advanced Robotics (ICAR), Montevideo, Uruguay, November 2013.
10. M. Henning. *A new approach to object-oriented middleware.* IEEE Internet Computing, 8(1):66–75, 2004.
11. C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee. *OPRoS: A new component-based robot software platform.* ETRI Journal, 32:646–656, 2010.
12. S. Joyeux, J. Schwendner, and T. M. Roehr. *Modular software for an autonomous space rover.* International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2014), Saint-Hubert, Canada, June 2014.
13. M. Klotzbücher, P. Soetens, and H. Bruyninckx. *OROCOS RTT-Lua: an execution environment for building real-time robotic domain specific languages.* 2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2010), pp. 284–289, Darmstadt, Germany, November 2010.
14. A. Makarenko, A. Brooks, and T. Kaupp. *On the benefits of making robotic software frameworks thin.* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007), San Diego, California, USA, October-November 2007.
15. A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. F. Ingrand. *GenoM3: Building middleware-independent robotic components.* Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA 2010), pp. 4627–4632, Anchorage, USA, May 2010.
16. I. A. Nesnas. *The CLARAty project: Coping with hardware and software heterogeneity.* In Brugali [6].
17. T. Niemueller, A. Ferrein, D. Beck, and G. Lakemeyer. *Design principles of the component-based robot software framework Fawkes.* 2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2010), Darmstadt, Germany, November 2010.
18. A. Nordmann, N. Hochgeschwender, and S. Wrede. *A Survey on Domain-Specific Languages in Robotics.* 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014), Bergamo, Italy, October 2014.
19. Object Management Group, Inc., Framingham, Massachusetts, USA. *Robotic Technology Component (RTC) – Version 1.1*, September 2012.
20. F. J. Ortiz, D. Alonso, F. Rosique, F. Sánchez-Ledesma, and J. A. Pastor. *A component-based meta-model and framework in the model driven toolchain C-Forge.* 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014), Bergamo, Italy, October 2014.
21. M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. *ROS: an open-source robot operating system.* Workshop on Open Source Software in Robotics, in conjunction with the IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan, May 2009.
22. M. Radestock and S. Eisenbach. *Coordination in evolving systems.* International Workshop on Trends in Distributed Systems (TreDS '96): CORBA and Beyond, pp. 162–176, Aachen, Germany, October 1996.
23. M. Reichardt, T. Föhst, and K. Berns. *On software quality-motivated design of a real-time framework for complex robot control systems.* Electronic Communications of the EASST, *Software Quality and Maintainability(60)*, August 2013.
24. C. Schlegel, A. Steck, and A. Lotz. *Robotic software systems: From code-driven to model-driven software development.*, Robotic Systems – Applications, Control and Programming, Dr. Ashish Dutta (Ed.), ISBN: 978-953-307-941-7, InTech, DOI: 10.5772/25896.
25. K.-U. Scholl, J. Albiez, and B. Gassmann. *MCA – An expandable modular controller architecture.* 3rd Real-Time Linux Workshop, Milano, Italy, 2001.
26. S. Schütz, M .Reichardt, M. Arndt, and K .Berns. *Seamless extension of a robot control framework to bare metal embedded nodes.* Proceedings of the Informatik 2014, pp. 1307–1318, Stuttgart, Germany, September 2014.
27. R. Smits and H. Bruyninckx. *Composition of complex robot applications via data flow integration.* 2011 IEEE International Conference on Robotics and Automation (ICRA), pp. 5576–5580, May 2011.
28. P. Soetens. A software framework for real-time and distributed robot and machine control. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006.
29. A. Steck and C. Schlegel. *Managing execution variants in task coordination by exploiting design-time models at run-time.* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011), pp. 2064–2069, September 2011.
30. N. Vahrenkamp, M. Wächter, M. Kröhnert, P. Kaiser, K. Welke, and T. Asfour. *High-level robot control with ArmarX.* Proceedings of the Informatik 2014, pp. 1283–1294, Stuttgart, Germany, September 2014.
31. D. Vanthienen, M. Klotzbücher, and H. Bruyninckx. *The 5C-based architectural composition pattern: Lessons learned from re-developing the iTaSC framework for constraint-based robot programming.* Journal of Software Engineering for Robotics (JOSER), Vol. 5(1), May 2014.
32. R. Vaughan, B. Gerkey, and A. Howard. *On device abstractions for portable, reusable robot code.* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), pp. 2121–2427, Las Vegas, USA, October 2003.
33. J. Wienke, A. Nordmann, and S. Wrede. *A meta-model and toolchain for improved interoperability of robotic frameworks.* 3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2012), Tsukuba, Japan, November 2012.
34. J. Wienke and S. Wrede. *A middleware for collaborative research in experimental robotics.* IEEE/SICE International Symposium on System Integration (SII2011), pp. 1183–1190, Kyoto, Japan, December 2011.

Bionotes

Max Reichardt
University of Kaiserslautern, Department of Computer Science,
D-67663 Kaiserslautern, Germany
reichardt@cs.uni-kl.de

Max Reichardt received his Dipl.-Inf. degree in computer science from the University of Kaiserslautern in 2008. Since august 2008, he is PhD student at the Robotics Research Lab. Software Engineering in the context of robotics is his main area of research. Topics of particular interest include software frameworks, their quality attributes, and design principles.

Tobias Föhst
University of Kaiserslautern, Department of Computer Science,
D-67663 Kaiserslautern, Germany
foehst@cs.uni-kl.de

Tobias Föhst received his Dipl.-Inf. degree in computer science from the University of Kaiserslautern in 2008. Since then, he works as a PhD student at the Robotics Research Lab. The field of interest lies within autonomous mobile robotics with respect to pattern-recognition, mapping, and navigation.

Prof. Dr. Karsten Berns
University of Kaiserslautern, Department of Computer Science,
D-67663 Kaiserslautern, Germany
berns@cs.uni-kl.de

Karsten Berns received his PhD from the University of Karlsruhe in 1994. Since 2003, he is a full professor for robotic systems at the University of Kaiserslautern. Present research activities are in the area of autonomous mobile robots and humanoid robots with a strong focus on control system architectures and behavior-based control.