
Introduction for Orocos developers

Open RObot COntrol Software

Copyright © 2006,2007 Herman Bruyninckx, Peter Soetens

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at <http://www.fsf.org/copyleft/fdl.html>.

Revision History
Revision 0.22.0 10 March 2006 ps
Created from Orocos Overview document

Abstract

This document explains the goals, vision, design, implementation and packages of Orocos [<http://www.orocos.org>], the *Open RObot COntrol Software* project.

Table of Contents

1. Development Vision	1
1.1. Software engineering requirements	2
Bibliography	5

1. Development Vision

Developing code for the Orocos project is guided by

1. *Software engineering requirements.*

What quality and approach are expected from contributions to the project?

2. *Structure of the source code.*

How is the project structured into sub-projects?

The project maintainers will *strictly impose* adherence to the project's vision and roadmap when considering incorporation of contributions to the project. However, they always welcome a motivated discussion about these topics on the project's mailinglist. [<http://lists.mech.kuleuven.be/mailman/listinfo/orocos>]

1.1. Software engineering requirements

Orocos has a vision on *how* developers should design, implement and present new functionality, in order to guarantee the project's coherence and scalability. These are the software engineering guidelines that support the implementation of the project's vision:

1. *Object-oriented design.*

This seems an obvious requirement for modern-day software development. However, the experience within the project has learned that designing an appropriate class hierarchy is a *major* challenge, mainly because of the diversity of the field.

2. *Extreme decoupling and modularity.*

This is the fundamental criterium to keep in mind when designing and/or refactoring the project's class hierarchy. Object classes encapsulate data and activities, and the choice of classes should be such that:

- The implementation of one class should not rely on knowing something about the internals of another class. Relying on such knowledge is called *coupling* between both classes, and this must be avoided at all costs, because it prevents the independent evolution or re-implementation of the classes.
- Every class should have an interface that fits more or less on one single page. Elaborate interfaces are a sign that the implemented functionality may better be split into multiple objects.

The strong desire to *decouple* software parts wherever possible is a key feature of the project; it may seem a bit of an overkill for each individual application built with the Orocos code (and hence a burden for the individual contributor), but for the whole project it is expected to become its major competitive advantage.

3. *Small and shallow interfaces.*

It is useful to separate the *interface* of a certain functionality from its *implementation* (as a class or as a component). The interfaces are what Application Builders and End Users need to construct their programs; the class and component implementations are constructed by the Framework and Component Builders.

Separated interfaces give more flexibility in changing implementations, and in allowing “third-party” providers of components. They also allow to discuss functionality without being biased by possible implementations problems or opportunities.

Orocos prefers interfaces that are:

- *Small.* This improves the focus of the interface, and hence (hopefully) its quality. In this sense, it corresponds to the minimality requirement discussed below. Another motivation is that Orocos wants to avoid “exceptions” in implementations of the interfaces: when the interface is large, the chance in-

creases that a particular developer will not be able to provide a *complete* implementation. In that case, the implementation must return an exception “functionality not implemented”, and this complicates the execution logic of the component that uses the interface, and reduces the real-time performance.

- *Shallow.* In an object-oriented paradigm, much functionality is provided by class *hierarchies*. Every hierarchy may seem natural in the context on one particular application, but it will most certainly not be natural in other applications. For example, a robotics engineer may find it normal to have a “robot” object at the top of a large and deep hierarchy. But that same word “robot” will not be well received by machine tool builders that construct milling machines, or laser cutters. Although most of the functionality of motion control and task execution will be the same in both application areas. Therefore, class hierarchies should not be deep.

In addition to the technical reasons to use small and shallow interfaces, they are a key feature for free software projects that expect significant contributions from their community: the complexity of the whole system can only be tackled by restricting the scope of each individual piece of code. An *increased flexibility* is a derived property that comes for free.

4. *Distributable.*

Future machine control systems will most certainly use multiple processors, connected through a network with sensors and actuators that most probably will have their own embedded intelligence. Hence, the robot/machine control software will have to be scalable and distributable.

Having a design that is modular and maximally decoupled fulfills already more than half of the requirements for such distributed control systems. The other half will come from making software components with an internal design that allows their different parts to cooperate over a network.

Two software aspects are important in this respect: the ubiquitous use of *events* for synchronization, and of a *virtual (network) time* (instead of the time delivered by the local processor). Both events and virtual time are general *abstractions*, that are easily mapped on corresponding primitives of the particular operating system the component is running on. For example, a hardware interrupt is also an event, and for the logic of a component it almost never matters whether the event comes from the hardware or from a local “stub” of remote hardware. So, make sure that your components don't use system calls that tie them to a particular operating system or a particular device.

Orocos provides *abstraction layers* for both the operating system and the interfacing hardware. These abstractions follow the principles of minimality and decoupling. Especially for the operating system abstraction this is a big advantage, because typical (real-time) OSs tend to have way too large and too primitive APIs, that are easily abused.

5. *Minimalistic.*

Developers should only offer features that are *absolutely* necessary: practical ex-

perience has learned that the availability of superfluous APIs leads to implementations with similar functionality but with different implementations. And, worse, to implementations where the programmers do not know very well what parts of the available functionality to use. Both effects lead to *sub-optimal* results, and applications that are more difficult to re-use and maintain.

6. *Platform independent.*

Because of its Free Software nature, Linux is the normal environment for both host development and target runtimes. This can in practice lead very quickly to an unperceived bias towards Linux, that would compromise the portability to other operating systems. Developers must try to use only portable language constructs.

Of course, sooner or later, the project will be confronted with the trade-off between portability and the choice for a particular desired feature that can not be supported by all initially targeted platforms. These trade-offs must be discussed on the mailinglist.

7. *Thorough large-scale design.*

Developers must *always* consider that their designs will possibly be used in a very complex, distributed and hard real-time implementation. Therefore, the design of such large-scale robotics systems must be made clear before starting the implementation of classes and components. And the implementators must provide contributions that can safely and efficiently work in such large-scale systems.

One particular item is *real-time*: if an algorithm *is* inherently real-time (i.e., it has a deterministic execution time), then its implementation *must* be implemented with constructs that are real-time safe. For example, all variables and temporary objects must be allocated before execution of the functionality proper, and no exceptions or run-time type checking must be used. (So, Java and C++ programmers need to be careful.)

All these requirements illustrate the project's emphasis on *design*: most things in robotics have already been tried out in various ways, so the Orocos implementation should *at least* be as good as the best of these tried-out predecessors. In addition, Orocos aims at a number of extra features, such as “ultimate generality”. This generality is the reason why the project follows a development approach which is maybe a bit different from traditional free software projects: the trade-off during the current development is towards having a very well thought-out and very general design, and not in the first place towards adding as many functionality features as possible. This approach is motivated by the observation that all existing robot control software packages are very difficult to use in applications that are different from the first application they were built for. And that sharing source or binary code is very hard in practice, because of ill-defined interfaces, that are most often also strongly coupled to one particular application.

The development approach of the project follows from the above-mentioned vision:

the project works in a “bottom-up” fashion, on small interfaces at a time, *but* while keeping the long-term goal in mind. So, it is currently building the lowest levels of the framework, such as the hard real-time “Core”, motion control functionality, primitives for communication between Orocos components, robot kinematics, etc.

Bibliography

- [posa96] *Pattern-oriented software architecture: a system of patterns* . Frank Buschmann, Regine Meunier, and Hans Rohnert. 1996. Wiley Chicester.
- [gof94] *Design Patterns Elements of Reusable Object-Oriented Software* . Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. Addison Wesley.
- [Johnson97] “Frameworks = (components + patterns)”. R. E. Johnson. 39–42. 40. 10. 1997. *Communications of the ACM*.
- [Szyperski98] *Component Software: Beyond Object-Oriented Programming*. C, Szyperski. Addison Wesley. 1998.