OROCOS: design and implementation of a robot control software framework

Herman Bruyninckx
K.U.Leuven, Mechanical Engineering
Leuven, Belgium
http://www.orocos.org

April, 2002

Abstract

This document discusses the shortcomings in the current state of robot control software, and explains how the Orocos project works towards more open software systems. This project builds a flexible and distributed framework for robot control, consisting of object libraries and components, and a standard middleware. The project targets multiple levels of "users " and large portability through availability of source code. The most important short-term results of the project will be: (i) a solid design basis for further development, and (ii) real-time motion control functionality for manipulators and mobile robots.

1 Introduction

The current commercial robot market is characterized by extreme incompatibilities between robot programming languages of different vendors. Moreover, the semantic richness of these languages is quite limited, and not much more advanced than what is offered by classical machine tools (i.e., point-to-point motions without much feedback opportunities from external sensors). This situation is reasonably viable for the current market of robot users (i.e., large-scale "motion control" applications without much flexibility needs), but is of little use for robotics researchers wanting to program advanced sensor-based robot tasks. Motion control of robotic devices is a quite mature field, ranging from off-the-shelf single-axis PID controllers, to customized systems with full dynamic, friction and elasticity compensations, multi-degrees of freedom trajectory generation, closed kinematic loops, etc. In practice, commercial robot controllers are invariably black-box products (hardware and software), in which the users have only access to some tuning parameters and the setpoint inputs. Moreover, while some open standards and open and modularly extensible APIs (Application Programming Interfaces) exist at the level of motion control primitives, e.g., [10, 11, 1], users lack *available implementations* of these APIs, that is, access to source code *and* the possibility to adapt that code. This results in:

- significant "*user lock-in.*" That is, once users purchase a controller from one vendor, it becomes hard to integrate it with products of other vendors, or to switch to a new vendor.
- a market that is *closed* to small innovative companies with specific niche market developments, because the set-up cost for robot control software development (and subsequent commercial distribution and support) is huge.
- small-scale users with *advanced or unique robot control needs* having difficulties to find solutions on the market.

Moreover, the fact that no open source solutions exist leads to:

- *de facto* centralized controllers. Because the control software is available in binary form only, vendors cannot provide their software for exactly the distributed computer network that the customer wants to use.
- *one-size-fits-all* solutions. Binary distribution of software also limits their configuration capabilities to be adapted to the exact functional and performance needs of the customer. This often results in sub-optimal use of software and equipment.

This situation is especially critical for research institutes, which most often don't get the needed support from the control vendors to extend existing equipment to prototype and validate their research on servo algorithms, motion interpolators, inter-process commu-

nication between distributed control components, "intelligent" sensor processing, advanced robotic devices, etc. The few cases in which they do get this support are cooperative developments, where the innovative developments are protected by non-disclosure agreements. Hence, progress in the field is slower than needed.

2 Long-term vision and short-term goals

The Orocos project is ambitious in its long-term goals:

- *Cover all robot control software needs.*
- *Be accepted by academia and industry.* Of course, industrial acceptance can only start with small and/or start-up companies, that see an advantage in using an existing open source code base to build a new (service-oriented) business. The goal is to be complementary to the ABBs and Kukas of this world, not to strive to compete with them in their own markets.
- *Fully distributable over LANs and WANs*, by compile time configuration. In this respect, the availability of the source code is a strong competitive advantage with respect to binary commercial packages.
- *Satisfying the needs of four categories of users*:
  1. *End users.* For example, a SME that uses one or two robots for a visual inspection task.
  2. *Application builders.* For example, the company that programs, services and maintains this visual inspection application.
  3. *Component builders.* For example, the researcher who codes an algorithm for visually guided motion control, and makes it available as a component.
  4. *Framework builders.* The people that develop the Orocos framework.
- *Ported to all Common-off-the-Shelf hardware, and (real-time) operating systems.*

The short-term goals must pave the road for this ambitious vision. Hence, they aim at building a working system, with a relatively limited set of features, but with the large potential for scalability provided by the design issues discussed in Section 5. The current, concrete activities of the project are: real-time motion control; kinematics and dynamics library; sensor processing library; task specification and execution; middleware for communication. More details are given in Section 6.

3 Flexibility

The Orocos project has investigated a lot of time to *design*. The envisaged flexibility is difficult to reach by the traditional open source approach to software development, i.e., starting from a working system developed by an individual programmer or a small group, opening the source code can bring a lot of contributions from interested people, but these contributions seldom lead to fundamental changes in the original design of the software. The success stories in open source all started from more or less mature examples, for which the basic design issues had been solved.

This situation is partially true for robot control software too: real-time motion control is a quite mature and stable field; and there are not many new things to be said about robot kinematics and dynamics. But many other aspect of robot control software have been much neglected by the commercial robot control manufacturers: distribution; tight integration between sensing and control; complex control tasks; multi-level task specification and execution monitoring; etc.

The presented framework design does not cover just one single system architecture, nor does it provide one single software component. The framework is rather an *application generating* code base, i.e., a *set of components* from which the skeleton of any robot control application can be constructed.

The major question in this framework design task is: How to design the framework components such that they provide *maximum flexibility* for building robot control system architectures? This paper uses "flexibility" as the common term for the following requirements:

1. *How to support distribution?* Developers may want to distribute software components over different networked processors ("decentralization"). Or they may want part of the implementation to run on very dedicated hardware, in order to get optimal performance. Therefore, the framework provides the flexibility to make the distribution of an application over multiple computing nodes as transparent as possible for the user.

2. *How to support modularity?* Different users require different sets of features, and want to rewrite only a minimum of existing code. Therefore, the framework components should have the "right" granularity.

3. *How to support configurability?* Different users want the available features to be configured differently, or even want to reconfigure a running system. Therefore, the design provides an appropriate granularity, and a far-reaching configuration abil-

ity. That is, the ability to adapt the software system in an anticipated way, without redesign or recoding, in several possible directions: *architecture* (how to connect components together), *distribution* (how to distribute components over a network), *communication* (choice of communication primitives between components), *functionality* (choice of control algorithms).

4. *How to support portability?* The presented design can be implemented and configured on a variety of hardware and software platforms. Care is taken to use only common operating system functionalities *and* to provide abstraction layers for hardware devices and operating system functionalities.

5. *How to support scaling?* A system scales well if its performance reduces not (much) more than proportional to the applied load. One important factor in the context of distributed robot control is the load increase generated by distributing components over a network. Different component communication architectures scale drastically different under distribution.

6. *How to support maintainability?* Every software system is in almost continuous evolution, and people that are not the original developers will be responsible for this maintenance. High-quality documentation of the system design and implementation is very important for maintainability, and hence, the presented framework builds its design as much as possible in terms of well-documented Software Patterns.

3.1 Component architectures

Some of the flexibility specified in the above-mentioned requirements is extremely difficult to achieve with traditional "binary-only" distribution of software. It can only be realised when application developers and users have *access to the source code*.

But also the *architecture* of the whole system determines the flexibility that can eventually be achieved: How is the functionality of the system divided over components? How flexible are the components themselves? What is the most appropriate granularity of "messages" exchanged between components? ...

"Component architecture" is the modern buzzword in the literature on flexible and complex systems. But much confusion still exists about what a component-based architecture really is, and what component-based architecture is most appropriate in a particular application domain. This Section gives a short answer to these questions for the robot control application domain.

Roughly speaking, a component is a piece of software that delivers a *service* that other programs can use. And delivering this service can be done with several degrees of flexibility. Internet services is the major area of application for component architectures, and in that area flexibility usually means "the ease with which one implementation of the service can be replaced by another implementation of the same service." This, however, is not the only meaning this text is interested in, as described in the previous Section.

3.2 Flexibility of component architectures

In order of increasing "replacement flexibility," one has the following three architectures:

1. *Client–server.* The "client" is any program that wants to make use of the service offered by a "server" component. In order to use the service, the client must *know* both the identity and the programming interface of the server.

2. *Object request broker (ORB).* The client doesn't have to know the identity of the server, but asks its local ORB program to connect it to the server. This connection can be achieved if the server has previously announced itself to its local ORB. The ORB "middleware" (i.e., the software that links all local ORBs together, e.g., using CORBA [12], Java-RMI/JavaBeans [], or DCOM [3, 9] implementations) takes care of realising the connection transparantly over a network or within one single computer. Once the connection is established, the client calls the methods of the server as if both were running on the same computer platform.

   The advantage of using an ORB architecture is that clients and servers can be replaced without recompilation or reconfiguration of all components they interact with (as long as the published interface doesn't change!). The disadvantages are that: (i) client and server still have to know each other's *exact interface* (changing the interface requires recompilation of all servers and clients!), and (ii) *all* method calls on a remote object must pass through the network between both components.

3. *Service request broker (SRB).* In a SRB architecture, the components don't do method calls on each other's interface, and they even don't have to know the exact API of the interface. Instead, the client launches a *request for a service* (often also called a *transaction*), the SRB middleware interprets this request, looks for a server that can deliver the service, translates the client request in a format that

the server can understand, connects to the server to start the execution, and returns the result to the client.

The request and the answer consist of one single data structure, so the granularity with which client and server talk to each other is much larger than in the ORB case. This reduces the overhead of method calling over the network, and it relieves the server from the burden of keeping *state information* about the client. Moreover, changing the interface of a server doesn't require recompilation of its clients, because only the middleware must be informed about the changes. This can be done by simple adaptation of description files, without recompilation.

The SRB concept exists for a very long time already (e.g., on mainframes, which are typically used for large-scale "batch" processing jobs), but re-gains popularity through the Jini and .NET initiatives.

A client–server architecture scales less than an ORB architecture, which scales less than a SRB architecture. "Scaling" is only one aspect of flexibility: the ability to (transparantly) add more server components under higher loads of service requests.

3.3 Flexibility in robot control systems

The previous Sections discussed flexibility issues of *general* component-based software systems. The flexibilities that are important in the context of robot control are:

- the possibility to replace components by alternatives with the same interface but with different implementations. This is particularly useful on the commercial market, to enhance competition between commercial component vendors.
- the fact that the larger part of the infrastructural design and software can be reused when extensions are needed for new or special-purpose applications.
- the possibility to re-use the same design but opt for different optimizations. For example: hard real-time; large-scale distribution and reduction in processing power available on each distributed processor; secure access; large number of synchronized axes; reduced network capacity; etc.

These flexibility requirements will be the major inspiration for the design choices made in the presented robot control software framework, Section 5. The rest of the paper will not discuss the *scalability* issue, because the robot control applications that are the subject of this paper typically don't need the kind of scaling flexibility provided by SRB middleware. At least not at the hard and soft real-time levels of robot control discussed in this paper: the goal of a robot control system is *not* to have a robot control task executed by *whatever* robot control server is available, because the system that must execute the robot task is exactly specified. However, part of the above-mentioned SRB scaling would be useful in robot control applications such as:

- *remote robot control.* For example, controlling the motions of a robot arm on Mars, or on an unmanned submarine. The significant time delays involved in these tasks call for autonomous "batch processing" at the remote system.
- *"intelligent" robot systems.* These systems have to do more than just execute a specified motion, because they have to observe the (largely unknown) environment in which they move, and take decisions on how to adapt the specified motion in order to avoid catastrophic interaction with the environment.

These advanced control systems are somewhat beyond the scope of this paper, which focuses on the hard and soft real-time components, and not on the *(autonomous) decision making* aspects. In any case, it remains useful to think about all of the above-mentioned scaling issues when designing a robot control API. An API that requires many method calls will perform worse in a distributed environment than one that has a more "transaction-oriented" API, i.e., one with a coarser granularity in its method interface.

4 Terminology

This paper uses terminology that may have different meanings in different contexts. This Section explains what meaning is used in the context of this paper.

4.1 Architecture

The architect of a system is the person who decides which components to put together, and in which way, in order to build a system that performs according to the specifications of the customer. Classical architects construct diverse "systems," such as houses, schools and theaters, from more or less standardized building blocks; similarly, software architects (should) build very diverse software products from standardized software components. (However, the software industry has not yet reached the level of mature standards in component-based system building.) In both cases, there is a need

for at least two complementary kinds of architectural design:

- *System architecture* ("architecture-in-the-large"): a specific choice of connecting functional building blocks ("components") together, in order to build a software system that performs according to specification.
- *Component architecture* ("architecture-in-the-small"): the internal design of one single component, in order to guarantee that the component performs according to its interface "contract."

Of course, the system designed by one architect can be used as one single component in the design of another architect.

This paper focuses on the software component architecture of a robot control system. The *system architecture* is the responsibility of the system builders, and is outside the scope of this paper. The paper also doesn't consider the problem of *software verification*, [2, 15, 14].

4.2 Components and Objects

"Components" and "objects" are terms that show up in every modern software engineering textbook, [16]. While the meaning of the term "object" is currently quite well understood and agreed on, there is still discussion about what exactly is the difference between an "object" and a "component."

This text uses the following interpretation for "component." A component is a piece of software that can execute its functionality ("deliver its service") independently of the context in which it is used, i.e., without having to know during design and implementation who is going to use its services, and how the're going to be used. This independence is related to the *functional* aspects of the component, i.e., the services it offers, but it is not absolute: there probably are implementation dependencies on the underlying operating system, and the service offered by the component probably requires some "boundary conditions" to be fullfilled, e.g., minimum processing power or memory resources. The component provides its functionality in the form of a well-defined method and event interface ("contract") that other components can use. The contract also states what the component expects from its context, e.g., quality of service required from the operating system.

Internally, the component implements its interface by one particular, configurable *architecture* of objects or other components (also called *sub-components*, or *atomic components*), but it does not depend on the knowledge of the internal workings of other components.

Components can have different acces points, each with its own interface, and each delivering a different service, catering to the needs of different clients. Often, the interfaces required in a particular system architecture are not exactly the interfaces offered by already available components, i.e., the interface "contracts" are more often than not imposed by the customer, not by the component builders, who have to comply to the required interface. So putting a matching "facade" around these components (in order to make them fit in the overall design) is a commonly encountered need. And availability of the components' source code is a major advantage in this case.

Components are for the user what objects are for the programmer: components are meant to interact with, while objects are the building blocks the internals of components are made from. Components are used through *service requests* (events and messages), not through *method calls*. They also don't have a persistent state (while objects do), i.e., you should not rely on knowing what the state of the component is when you make a service request. The state of the service itself is kept in the message objects given to, and received from, a component. As such, components are much better suited for the higher-level, coarse grain distribution of Service Request Brokers than the lower-level, fine-grained remote method invocation of Object Request Brokers and Client–Server systems. This does *certainly not* mean that the idea of a component is not useful at these "lower level"! It only means that objects are not useful as atomic building block at the "higher" level.

4.3 Coupling

The design of both components and objects should maximize *loose coupling* between components: one component or object should try not to use knowledge about the identity, the method or service interface, the location, or the internal state of other components or objects. For example: the IPC between components is decoupled if the "sender" doesn't have to know the identity of the "receiver"; component A can cause component B to execute a certain action also by sending an event (which doesn't require A to know B) to which B reacts, and not only by directly calling the corresponding method of B. Coupling comes in two forms, *syntactic* and *semantic*:

- *Syntactic coupling.* This is coupling at the level of the *programming language*: object A uses some syntactical means available in the programming language (a *method call*, for example) to identify object B.

  Syntactical coupling leads to less flexible software.

- *Semantic coupling.* This is coupling at the level of the *meaning* of the program: object A uses some

service of object B, whose respons depends on coupling between the "states" and/or "activities" of other objects, possibly including A.

Semantical coupling is a human design error, that gives rise to *erroneous* software: sooner or later, the objects' behaviour is not deterministic, because it depends on irreproducible, accidental couplings between objects.

One typical example of semantic coupling is a software system where component B bases its decision on the fact whether component A is in either state "1" or state "2," while component A is in no well-defined state at all because its "state" is that it has requested a service from component B and this service has not been completed yet!

4.4 Framework

A framework is *a design and an implementation* providing one possible solution in a specific problem domain. So, it is possible to compile the code provided by the framework and solve some real problems. Application-dependent implementation parts are localized in so-called "*hot spots*" [4, 7, 8]. The framework itself is portable, but the hot spots should in general be rewritten when porting to a new hardware platform. Typical hot spots in the context of robot control systems are:

- The *device interface layer* of the framework, i.e., device drivers for sensors and actuators.
- The *OS interface layer*, i.e., the API of the underlying (real-time) operating system.
- The *protocol interface layer*, i.e., the communication protocols (XML-RPC, SOAP, CORBA, ...), the motion specification (G-code, Step-NC, ...), and the user interfaces.

The flexibility of a framework is improved by: (i) a good identification and description of its hot spots (*portability* and *maintainability*), (ii) a component-based design (*modularity*, *scalability* and *distributability*).

4.5 Software Pattern

A Software Pattern, [6, 13], is a proven design solution (*not* an implementation!) to a general problem, resulting from years of real-world experience in balancing a number of opposing "forces." The terminology "forces" is used in the Software Pattern literature to describe the aspects of the application domain that drive the design into different directions. For example, timing constraints, the ability to execute services as an *atomic transaction*, resource constraints, etc.

A Software Pattern is not an implementation but a *design*; the same pattern can have multiple implementations, depending on the application context. The use of Software Patterns in the design of a new framework has the advantage that these patterns are very well documented and scrutinized in the literature, by developers with different backgrounds and applications. So, refering to this Software Patterns literature whenever possible helps developers decrease their documentation efforts while at the same time increasing the quality of their documentation, and hence the *portability* and *maintainability* of their implementation.

4.6 Module—Library

A module (or library) is a coherent set of implemented *functionality*, using an object-oriented data encapsulation as a useful (but not necessary) design paradigm. A module can be compiled separately, and is portable to different platforms given compatible compiler and operating system support. Of course, inherently hardware-dependent modules such as device drivers are not portable.

Examples of modules in the context of this paper are: trajectory generation algorithms, feedback control laws, device kinematics, interprocess communication support, motion estimation algorithms, etc.

4.7 Users and Builders

This paper presents a software framework for distributed robot control. "Distribution," however, means more than the ability to run the software on multiple networked processors: also the human activities of building a robot control application are distributed, over three categories of "users" with complementary needs and responsibilities:

End Users The End Users only focus on the functionality of their application, that means configuring the available motion generation and robot control components, within the context of the robot control system given to them by some Builders.

Application Builders The Application Builders use the *infrastructure* and the *components* offered by the framework to assemble a specific robot control application. Application Builders could be: a robot control service company; or a PhD student in need of a controller for his new humanoid robot; or a manufacturing company adding its in-house designed peripheral machinery to a legacy machining centre.

The assembly activity of the Builders consists of: (i) the configuration of components and IPC in the

framework; (ii) the implementation of the *hot spots*; (iii) the implementation of a specific architecture and default implementations for the targeted user base; (iv) the specification of the "language" with which Users interact with the application.

Component Builders These people develop functionality, make it available in a module, and wrap it in a component, for use by the Application Builders.

Framework Builders Finally, there is also this third party involved, who design and write the code with which Component Builders, Application Builders and End Users can work.

The Orocos project is mainly focused on the Component and Framework Builders.

5 Design

This Section discusses the *design considerations* underlying the presented robot control framework. These choices are inspired by the issues discussed in the previous Sections.

5.1 Separation of architecture and functionality

Only the Developer of the framework must worry about how to implement the most appropriate *component architecture*, making sure the best software engineering practice and programming language constructs are used. The application Builders using the framework are themselves also developers, but only of the *system architecture* (i.e., they decide which components of the framework to use, and how to connect them), and the *component functionality* (i.e., they code the algorithms in the sub-components).

5.2 Separation of mechanism and policy

All developers should make a clear distinction between *implementing* functionality (*what* can be done, i.e., the *mechanism* of the system) on the one hand, and deciding *how to use* this functionality on the other hand (i.e., the *policy*). Too much focus on one particular application leads to policy-based trade-offs early in a software project's lifetime, inevitably compromising its flexibility and maintainability.

5.3 Separation of data flow and execution control

Complex, distributed systems typically have several active components, that need to communicate with each other, and that rely on each other's services. Because several components can want to access the functionality of one specific component, each component should encapsulate the execution of its functionality from the communication with other components, [5]. In other words, the *data flow* (i.e., exchange of objects through messages) should be decoupled from the *execution control* (i.e., the sequencing and synchronization of activities within the internals of a component). Of course, this decoupling does *not* mean that the communication of an object (with an *event* being the simplest possible object exchange) could not *influence* the execution control: it only means that it is the component architecture of the server that decides how and when to service the event, and not the calling client component. In addition, with respect to the execution control, each component should guarantee that:

- well-defined parts of its execution can take place without interruption by communication (or, in other words, *atomically*).
- dynamic reconfiguration of the component's functionality (e.g., transition from one control law to another) can take place, in synchronization with other components.

5.4 Separation of Builder and User services

The framework targets decentralized, portable and dynamic systems, hence, on-line reconfiguration must be possible. This requires different interfaces for "supervisor" and "normal" user access to the same component. The supervisor user is, in fact, a Builder (Section 4.7) as far as (re)configuration of the framework is concerned.

5.5 Separation of Quality-of-Service and OS primitives

Most operating systems offer a powerful but primitive interface to the application programmer. That is, an application program with the appropriate privileges can access and control each and every resource in the system. In this way, it *could* perform its task in exactly the way it likes (given sufficient resources). But this is not really what an application program *should* do: it should focus on its core functionality, and the computing services it requests should be "outsourced" to the operating system. However, there currently are very

few *Quality of Service* operating systems available, let alone a standardized API to access such a QoS operating system. This implies that the presented framework will have to implement its own QoS interface *hot spot* to operating systems.

5.6 Architecture instead of management

This paper presents the design of robot control components; it does *not* present a full robot control *system architecture*, but concentrates on the *component architecture* of the robot control components. This component architecture should be such that the component performs its contract *without* the need for a "supervisor" or "manager" sub-component. The task of such a supervisor would be to make sure that all actions within the component are done in the correct order, that the correct events are signalled at the correct times, that the produced values are checked for correctness, etc. However, it is way better to have a component architecture that *produces correct behavior by design* rather than *by management*. Some real-world examples should make the fundamental difference between both approaches more tangible:

- World-wide express courier services: after one hands in a package for delivery, there is not a manager taking care of your package, but it is rather the organisation ("architecture") of the express courier that steers your package through countless manipulations and transfers.
- Traffic: one can put traffic officers on a large square to keep the traffic coming from all directions under control, or implement an infrastructure of lanes and traffic lines.

A manager is more flexible, but encourages unstable adaptations ("I'll fix it, and fix it quickly!") and doesn't scale well (because it needs centralized communication).

6 Progress

This Section explains the current state of the active parts in the Orocos project. The project is financially supported by a small grant from the European Union (IST 31064), with three partners: LAAS in Toulouse, France; KTH in Stockholm, Sweden; and K.U.Leuven in Leuven, Belgium. But much of the design and implementation work is performed by developers that are not paid by this project money. The project's progress can be followed through the webpages at `http://www.orocos.org`.

6.1 Real-time motion control

This task has reached the level of basic real-time motion and force control of a 6DOF robot arm. The basic features are:

- Running on top of RTAI and RTlinux, on PC hardware, with a control frequency of more than 1 kHZ.
- Pluggable component architecture with Trajectory interpolator, Servo, Event Handler, Time Server, and Task Execution.
- Data monitoring and reporting.

6.2 Task execution and specification

Each controller can accept a specification that requires strictly synchronized execution of sub-task. To this end, each part of a feedback control module (Trajectory interpolator, Servo, ...) is running through an event-driven state machine.

6.3 Kinematics and dynamics

The basic kinematic routines are implemented, but the design allows scaling to general dynamic systems beyond the mechanical domain. The design is also scalable in the sense that geometric visualisation can be dynamically added to a kinematic structure. The kinematics can handle serial, parallel, hybrid and mobile structures.

6.4 Communication and middleware

A distributed system has a large need for communication functionality and middleware that allows distribution transparancy. The evolution of the CORBA standard [12] is very promising in this respect: CORBA 3.0 has standardized most of the needs for distributed components in the Orocos project.

References

[1] Open system architecture for controls within automation systems (OSACA). `http://osaca.isbe.ch/osaca`.

[2] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[3] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C.-Y. Wang, and Y.-M. Wang. DCOM and CORBA side by side, step by step, and layer by layer. `http://www.cs.wustl.edu/~schmidt/submit/Paper.html`.

[4] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building application frameworks: object-oriented foundations of framework design*. Wiley, 1999.

[5] S. Fleury, M. Herrb, and R. Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Int. Conf. Intel. Robots and Systems*, pages 842–848, Grenoble, France, 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[7] R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[8] M. E. Markiewicz and C. J. P. de Lucena. Object oriented framework development. *ACM Crossroads*, 7(4), 2001. `http://www.acm.org/crossroads/xrds7-4/frameworks.html`.

[9] Microsoft. Distributed Component Object Model (DCOM). `http://www.microsoft.com/com/tech/DCOM.asp`.

[10] National Institute of Standards and Technology. Open modular architecture controls (OMAC). `http://www.isd.mel.nist.gov/projects/teamapi/`.

[11] OPC Foundation. OLE for process control (OPC). `http://www.opcfoundation.org/`.

[12] Open Management Group. CORBA: Common Object Request Broker Architecture. `http://www.corba.org/`.

[13] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture. Patterns for concurrent and networked objects*. Wiley, 2001.

[14] D. Simon, B. Espiau, K. Kapellos, and R. Pissard-Gibollet. The orccad architecture. *Int. J. Robotics Research*, 17:338–359, 1998.

[15] A. Sowmya and S. Ramesh. Extending statecharts with temporal logic. *IEEE Trans. Software Engineering*, 24(3):216–231, 1998.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.