OpenRDK: a modular framework for robotics software development

Daniele Calisi, Andrea Censi, Luca Iocchi, Daniele Nardi

***Abstract*— In this paper we conduct an analysis of existing frameworks for robot software development and we present OpenRDK, a modular framework focused on rapid development of distributed robotic systems. It has been designed following users' advice and has been in use within our group for several years. By now OpenRDK has been successfully applied in diverse applications with heterogeneous robots and as we believe it is fruitfully usable by others we are releasing it as open source.**

I. INTRODUCTION

More than sixty years after the creation of the first programmable computers, programming is still an art [8] rather than a science. The advancements of hardware capabilities, and the networking revolution of the last decade, exacerbate the problem because new applications of unprecedented complexity are possible and desirable.

The discipline of software engineering aims at providing paradigms, methods, and tools for designing and realizing complex software systems. The problem with such paradigms, methods, and tools is that they tend to come and go in a few years, a time scale that is incomparable with other engineering fields. Object oriented languages (C++, Java), UML, distributed services (Web 2.0), agile languages (Python, Ruby, etc.) are just a few examples of a very fast evolution of methodologies and techniques in less than twenty years.

The situation is even worse when we look at software for robotic applications. In this paper, we refer to "robotic applications" as complex systems based on mobile robotics technology and we consider robotic software to be in the middle ground of two fields: it is as complex as other standard ICT software, and it has the same hard real-world requirements as industrial automation. On the one hand, developing a full control stack for a team of heterogeneous robots is as hard as any other large distributed software system; however, standard tools of ICT software might not be applicable because of the real-world issues like network unreliability or real-time constraints. On the other hand, formal methods used in automation for developing distributed control systems, like the IEC 61499 standard [12], are not really applicable to robotics, because they were designed only for relatively simple systems.

The scientific community is undertaking an intense effort to find a common structure in robotic applications, both from a conceptual and from an implementation point of view, in order to achieve a wide deployment of standard design techniques, architecture styles, and reusable components. This resulted to be very difficult because of the wide range of domains where robots can be exploited, the variety of forms and functions that a robot can have, and, moreover, because of the diversity of people involved in robotics [2]. It is then natural that people working with robots felt the need to develop their own solutions.

Although the primary development of mobile robots will eventually be the industry, nowadays the coolest-acting robots (e.g., those seen in DARPA Grand Challenge, or the Mars rovers) still come from academia or other research environments. Consequently, much of the related work on robotic frameworks makes sense only for the needs of a relatively small academic research group.

Our work on OpenRDK is focused on the needs of a research group, whose aim is to create innovative algorithms for complex mobile robotic systems, without spending more time than necessary on the software infrastructure. The research group we are considering here is formed by many people (e.g., 10 undergraduate students) working for a short time (e.g., 6 months) on a complex project (e.g., multi-robot heterogeneous exploration in a disaster scenario), usually with no or only a few experience on robotic applications and on software engineering. Hence our main goals are modularity and code re-usability, but we need to take into account also other real-world related problems, like efficiency, noisy perception, unreliable networks, etc.

The contributions of this paper are twofold. On the one hand, we conducted a deep analysis of the issues that a robotic software framework should address, and how existing frameworks take their choices. On the other hand, we present our own framework, designed to meet the goals described above through its unique combination of design choices.

The software can be downloaded from `http://OpenRDK.sourceforge.net` and can be used under the terms of the GPL (GNU General Public License).

II. RELATED WORK

One tenet of engineering is reducing the complexity of a problem by dividing it into smaller problems (divide et impera!). In software, this leads to dividing an application in smaller "modules": mutually decoupled software units with precise interfaces. In the following we will use the term "module" to refer to this kind of entity, regardless of the use of other words in other frameworks (e.g., component, service, client, driver, etc.).

D. Calisi, L. Iocchi, D. Nardi are with the Dipartimento di Informatica e Sistemistica "A. Ruberti", "Sapienza" Università di Roma, via Ariosto 25, I-00185 Rome, Italy. {calisi,iocchi,nardi}@dis.uniroma1.it

A. Censi is with the Control & Dynamical Systems department, California Institute of Technology, 1200 E. California Blvd., 91125, Pasadena, CA. andrea@cds.caltech.edu

With a modular architecture, there are two main choices to be made: how to arbitrate the modules access to the computational resources, and how to make the modules communicate. These are the two main dimensions in which we frame the related work, a summary of which is shown in Table I. The table also shows which tools the frameworks provide for speeding up development: examples are simulators; loggers for sensor data; remote inspection tools to observe/modify the module state; and configuration utilities.

A. Concurrency model

We classified three possibilities, as follows.

- Modules are *processes* distributed over one or more machines. In this case, developers have the highest freedom. The major drawback is the need of some communication infrastructure that allows for inter-process communication.
- Modules are *threads* inside a single process. With multi-threading, sharing information is easily accomplished using shared memories, but this requires that the framework provides some mechanism for data access concurrency and thread synchronization.
- Modules are defined by *call-back functions* and there is a single process (i.e., a scheduler) that repeatedly calls them in response to some event or periodically. The call-back functions are preferable for frameworks whose focus is on low-level device interaction: in particular, call-backs allow for a better scheduling control, that can be used to enforce real-time constraints. On the other hand, writing this kind of call-back functions is difficult, because they need to return quickly to the scheduler and thus distribute the computation over more than one call.

The distributed process model is a very common choice: most existing frameworks use this architecture. Nevertheless, these choices are not mutually exclusive, and some frameworks use of a hybrid architecture.

B. Information sharing model

There are two metaphors that can be used to model the exchange of information among modules:

- modules have input and output "ports" from which they can receive or send messages to other modules that are connected to these ports;
- there is a central object where modules "publish" their data and where they can read other modules data, using some sort of addressing scheme.

In practice, there are two main mechanisms that may be used: modules within the same process may use shared memory, while modules distributed among different processes/hosts have to use some inter-process communication service.

If modules are implemented as threads or as call-back functions, shared memory is the most efficient communication method. In the case of threads, concurrency management primitives are necessary. We remark that the shared memory mechanism cannot allow for all semantics which is usually needed. An example is if two modules act as a producer/-consumer couple: in this case the framework must provide a mechanism to implement some kind of data "queue" in shared memory.

Some of the existing frameworks make use of third-party middleware to accomplish inter-module communication. The advantage of this choice is that they are broadly experimented and stable, but there are also some drawbacks: often their goals, like multi-platform/multi-language support and application independence, are not of primary importance in robotics. Writing a proprietary middleware can be a difficult and a long work, but it allows to fit the specific needs of the robotic application, without unnecessary added complexity.

TABLE I

SUMMARY OF EXISTING SOFTWARE ROBOTIC FRAMEWORK

Framework	Concurrency model	Information sharing	Tools	Focus
OROCOS[1] [3]	call-backs, threads	lock-free data ports (CORBA)	remote inspection, logging	low-level devices
Orca[2] [9]	processes	ICE[13]	remote inspection, logging	mobile robots
CARMEN[3]	processes	IPC	logging, visualization	mapping and navigation
OpenRTM-aist[4] [1]	threads	CORBA	configuration GUI	general robotics
Microsoft Robotics Studio[5]	processes	HTTP/DSSP via DSS	3D simulator	general robotics
Player[6] [5]	threads (server)	client/server, proprietary over TCP	2D and 3D simulators	low-level device drivers
MOOS[7]	processes	centralized, proprietary over TCP	logging, viewers	mobile robots
CLARAty[8] [10]	threads, processes	relies on ACE[14]	none	real-world systems
MARIE[9] [6]	processes	many (3rd party)	configuration GUI	connecting different frameworks
MOAST[10]	processes	NML/RCS[11]	logging, visualization	USARSim, mobile robots
MIRO[11] [11]	processes	CORBA	logging	mobile robots
SPQR-RDK [7]	call-backs, threads	proprietary over TCP	remote inspection	mobile robots
OpenRDK	threads	shared memory, proprietary TCP/UDP	remote inspection, logging	mobile robots

[a] http://www.orocos.org
[b] http://orca-robotics.sourceforge.net
[c] http://carmen.sf.net
[d] http://www.is.aist.go.jp/rt/OpenRTM-aist
[e] http://msdn2.microsoft.com/en-us/robotics/
[f] http://playerstage.sf.net
[g] http://www.robots.ox.ac.uk/∼pnewman/TheMOOS/
[h] http://claraty.jpl.nasa.gov/
[i] http://marie.sf.net
[j] http://moast.sf.net
[k] http://www.isd.mel.nist.gov/projects/rcslib
[l] http://smart.informatik.uni-ulm.de/MIRO/
[m] http://zeroc.com/ice.html
[n] http://www.cs.wustl.edu/ schmidt/ACE.html

III. OpenRDK

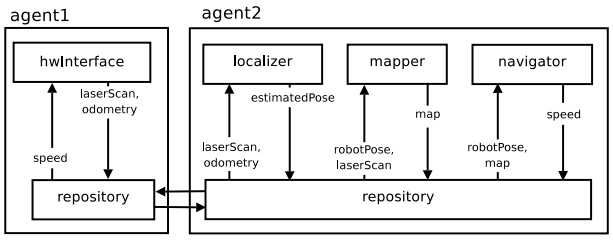
OpenRDK is written in in C++ and it runs on Unix-like operating systems (Linux, OS X).

The main entity is a software process called **agent**. A **module** is a single thread inside the agent process; modules can be loaded and started dynamically once the agent process is running.

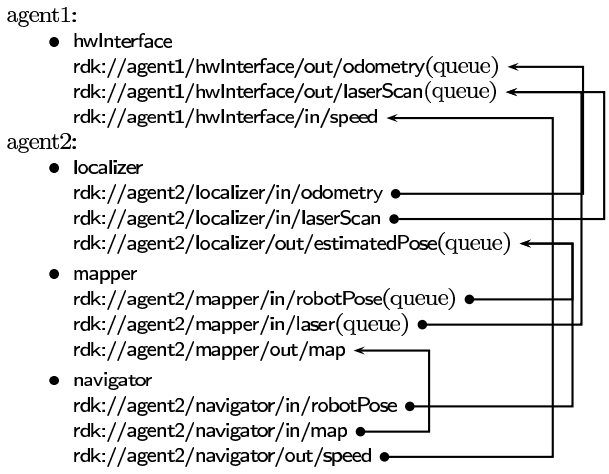
An agent **configuration** is the list of which modules are instantiated, together with the value of their parameters and their interconnection layout. It is initially specified in a configuration file.

Modules communicate using a blackboard-type object, called **repository** (see Figure 1(a)), in which they publish some of their internal variables (parameters, inputs and outputs), called **properties**. A module defines its properties during initialization; after that, it can access its own and other modules', within the same agent or remotely, through a global URL-like addressing scheme. Access to remote properties is transparent from the module perspective and it reduces to (regulated) shared memory for local properties. Special **queue** objects also reside in the repository and they share the same global URL-like addressing scheme of other properties.

In Figure 1(a) we see an example. Two agent are executed on two different machines and three modules run inside them: ***hwInterface*** retrieves data from a laser range finder and the odometry by a robotic base; given these two piece of information, ***localizer*** uses a scan-matching algorithm in order to estimate the robot positions over time; ***mapper*** uses the estimated robot positions, together with the laser scans, to build a map of the environment. The ***hwInterface*** module pushes laser scan and odometry objects into queues, that are remotely accessed by the ***localizer*** module, which, in turn, pushes the estimated poses in another queue, for the ***mapper*** to access to them. The map property is then updated by the ***mapper*** and being used by the ***navigator***.



(a) Logical module interconnection



(b) Properties and links realizing the interconnection

Fig. 1. Example configuration of two agents running on two different hosts

A. Concurrency model

OpenRDK uses multiple processes with multiple threads. We rejected the alternative of using one thread per process and implement modules as call-back functions because we saw, we saw, through the years, that our typical user does not have the required discipline to implement them correctly. The multi-threading solution is a good compromise, even though it required an infrastructure for concurrent data access and synchronization.

At run-time, each module (thread) is waiting for some event to occur. Typical events are fixed interval timeouts, new data on a queue or the change of a property value; modules can wait on more than one event.

B. Repository, properties and URLs

The repository is the place where all modules publish the data they want to share with others. Published properties are inputs, outputs and parameters. A path-planner module, for example, could publish the current and target robot poses as inputs, and the resulting path as output.

Each property is assigned an URL with the syntax: `rdk://<agent-name>/<module>/<property>`
Note that the agent name is different than the host name, as there can be more than one OpenRDK agents on the same host. Some examples of properties can be found in Figure 1(b), that shows the properties of the example of the agents in Figure 1(a). These globally unique URLs allow every module to transparently access a property on any other agent (on any other host).

C. Object serialization

Serialiazion is the process of transforming run-time objects in a form suitable for storage or transmission. In OpenRDK, each class implements a serialization API which has the same interface for writing both in XML (typically for storage purposes, for example to save configuration files that can be easily read by a human) and in a more efficient binary format (for transmission over the network). We chose not to provide any automatic mechanism to generate this kind of functions (for example, using an IDL-like language), because platform independence is not among our goals and, moreover, because in this way we have a great flexibility that allows for the implementation of ad-hoc serializations (e.g., for lossy/lossless compression of maps and images).

D. Configuration and object persistence

A configuration is a list of modules to be loaded and executed, their interconnections and the values of their

parameters. Agent configurations are saved as XML files and contain a serialized representation of each module and its properties. Since parameters are properties, they can easily be saved using the mechanism seen in the previous subsection. Moreover, since we save all module properties in the configuration file, this technique can also be used to save the state and load it afterwards. For example, we can have the robot building a map of an environment, save it once in the configuration file, and then use it for all subsequent runs.

E. Property links

The device of "property links", analogous to Unix symbolic links, introduces a level of indirection in the repository that allows to make the modules as decoupled as possible.

The main problem that links solve is that two modules needing to share an information must agree on some well-known place (in our case, a URL) where this information is to be accessed, and this creates an unnecessary coupling between them.

Links can point to remote properties as well, and this allows to distribute the computation in a way which is completely transparent to the module developer.

Links are specified in a configuration file; since the data flow is not hard-coded, modules can be easily re-used for different applications. In practice, this encourages the developers to create many small re-usable modules instead of big monolithic ones.

For example, consider Figure 1(b): the ***mapper*** module needs a robot pose in order to build the map. Thanks to the property links, the development of this module does not have to be delayed until the ***localizer*** is finished. In fact, the `mapper/in/robotPose` property can be linked to `hwInterface/out/odometry`. When the ***localizer*** is ready, there is no need to change any code, only the configuration. Moreover, there are, for example, applications and scenarios that do not need a ***localizer*** module and the odometry is sufficient.

F. Queues as object dispatchers

OpenRDK implements two models for sharing data between modules: publisher/reader and producer/consumer. Regular properties realize the former, and special "queue" properties implement the latter.

Queues are very smart FIFO containers:

- They support multiple readers; thread-safeness is ensured without object duplication;
- They own the objects that are pushed into them and take care of garbage collection, by destroying the objects when no reader is interested in them anymore;
- They allow subscribing modules to listen to particular objects entering in the queue, and to be awoken on that event.
- They are 'passive' objects: no additional thread is required.

Although the OpenRDK queues implementation is rather complex, it is kept "under the hood" and they actually make

```
// Module hwInterface, on agent1
RDK2::RODometry* odom = new RDK2::RODometry(/* ... */);
session->queuePush("odometry", odom);
```

```
// module localizer, on agent2: during initialization
session->subscribeQueue("odometry");
// "odometry" is linked to "rdk://agent1/hwInterface/odometry"

// module localizer, on agent2: during execution
while (session->wait(), !exiting) {
    vector<const RDK2::RODometry*> v =
        session->queueFreezeAs<RODometry>(ODOMETRY_URL);

    for (size_t i = 0; i < v.size(); i++) {
        const RODometry* odom = v[i];
        // process odometry data in the queue
    }
}
```

Fig. 2. Example of using a queue object in OpenRDK (see also Figure 1(a))

module writing much easier. See for example Figure 2, in which we show the communication between two modules, `hwInterface` and ***localizer***: the first module takes the odometry reading from the sensor device and pushes in a queue called "odometry", on another agent, the module ***localizer*** subscribes to that queue during initialization and then is able to retrieve the values in a very simple way.

G. Inter-agent information sharing

As we described above, information sharing among modules that are executed inside the same agent (process) is accomplished using the repository. Inter-agent (i.e., inter-process) communication is accomplished by two methods: through *property sharing* and *message sending*.

In the first case, one agent refers to properties of another agent by specifying the name of the remote agent in the URL of the property (this is usually done when specifying a property link in the configuration). The repository is in charge of requesting the remote property value and publish it in a local copy (proxy) for the requesting module to read.

Property sharing can be tuned by using a set of parameters, that can be specified in the configuration files and are explained in the following.

- The subscriber can request a property update every time it changes on the published repository (`ON_CHANGE`) and optionally set a minimum interval between two subsequent updates. As an alternative, it may request that the update have to be sent at fixed intervals (`PERIODIC` in OpenRDK terms).
- The subscriber can request to use one of two transport protocols: UDP or TCP.

OpenRDK also partially implements a data reconstruction layer, i.e., some object can be split in multiple packets and reconstructed in the destination repository. Moreover, for some objects can be requested a `LOSSLESS` (default) or `LOSSY` compression. For example, if an image have to be sent to an image processing module, chances are that it need the image as it was on the source side. On the other hand, if

the property is requested solely for visualization purpose, a lossy compression is more effective.

In addition to the property sharing mechanism, we also implemented a standard message-sending (mailbox) method, for when this feature is a more straightforward mapping with the semantic of the application. When a module wants to send a message to another module, being it on the same agent or on a remote one, it writes the address of the recipient on the message object and push it into a special queue called "outbox". Receiving a message requires to subscribe to the "inbox" queue and to be able to discern interesting messages.

H. Tools contained in the OpenRDK

RConsole: RConsole is a graphical tool for remote inspection and management of modules. We use it as both the main control interface of the robot and for debugging while we develop the software. RConsole was very easy to implement thanks to the property sharing mechanism: it is just an agent that happens to have some module that displays a GUI. Through the reflection used in the repository, graphical widgets visualize the internal module state and allow the user to change their parameters while running. Advanced viewers allows to interact with images and maps, moving robot poses, seeing visual debug information provided by modules, etc.

Modules for logging and replaying: OpenRDK provides a configurable module that, reading from a sensors queue, is able to write a log file containing the sensor data. This file can be processed off-line using third-party tools or used in conjunction with another module that provides the "playback" feature.

Connection with simulators: As we explained in the Section II, OpenRDK provides modules that allow to connect to both USARSim and, through Player, to Stage and Gazebo. The modules expose the same interface of the real ones, thus resulting in a transparent behavior for the modules that connects to them.

IV. Some applications of OpenRDK

The OpenRDK framework has been successfully used in a wide range of robotic applications. Our group has a long record in RoboCup competitions. OpenRDK has been extensively used in all competitions in which we have been involved: RoboCupRescue Real Robots, RoboCupRescue Virtual Robots, RoboCup@Home, and also RoboCup soccer Standard Platform League (with two-legged humanoid robots). In particular, the latter league makes use of Aldebaran's Nao humanoid robots. OpenRDK currently runs on the Nao's internal computation unit and our team has developed modules for a humanoid robotic soccer application.

A. Single rescue robotic system

Our group is involved in rescue robotics, whose goal is to develop robots to assist human rescuers during emergency operations. The main capabilities needed by such a robot are:

- to build a map on an unknown environment;
- to be able to move autonomously in a cluttered scenario;
- to report to the human rescuers the interesting features found during the exploration (for example, possible human victims that are entombed or trapped, or possible treats).

The system has been developed as an OpenRDK agent. The real robot was equipped by two personal computers and two agents run on each of them: in this way, we were able to divide the computation weight among two machines. In particular, the first was responsible for the robot mapping and navigation subsystems, as well as the mission manager module; the second machine contained the modules for vision processing. In this application, one example of property sharing is that the vision module published a queue of "possible human sightings" that was read remotely by the mission manager module on the other PC. See Figure 3(a) for details.

By simply substituting the real sensor and robot modules by modules that connected to a simulator, we have been able to test exactly the same software system in both real and simulated scenarios. The simulated rescue scenario allowed us to conduct experiments with a large number of robots in a large environments.

B. Assistive robots

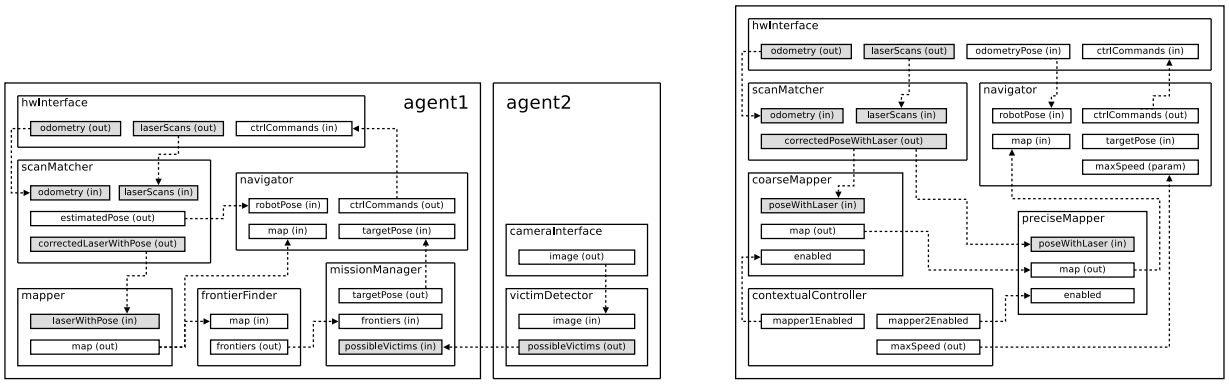
The RoboCare Project[1] aims at building a system for assistance of the elderly and the impaired person. Such non-invasive technology is a distributed and heterogeneous system and should be easily integrated in the environment and able to interact with the person and to monitor his behavior. Some of the main components is a multi-camera system that can follow the human in the environment and track his position, a wheeled robot that can move in the environment and interact with the human through a human-robot interface, and a PDA that the assisted person can use to interact.

In this project, two OpenRDK agents are involved and interconnected to a pre-existent system. One of them is responsible of managing the mobile robot. It includes modules for localization in a known environment as well as path-planning and dynamic obstacle avoidance. Another OpenRDK agent is running connected with the camera tracking system and is responsible for sending the image data to the PDA and to send the tracked human position to the robot agent.

C. Context-based online configuration

In a recent work of ours [4], we studied the possibilities of a system that is able to control the behavior of other modules by using "contextual" information. Using the OpenRDK framework, we were able to test this idea in a straightforward way. The only thing we needed was to implement the contextual controller and connect its outputs to the *parameters* of the other modules. In this way, the contextual controller was able, for example, to reduce the maximum speed of the path planner, when the situation required slower movement,

[1] http://robocare.istc.cnr.it



(a) a single rescue robotic system application with two agents

(b) a context-based online configuration architecture

Fig. 3. Two examples of configuration: gray properties are queues

or to switch between two mapper modules. In Figure 3(b) we can see a simplified diagram of this system.

V. Conclusions and on-going work

In this paper, after a deatiled analysis of the many existing robotic frameworks and how they address the most important issues, we presented our own OpenRDK framework. Our design choices reflect the need for fast development of complex robotics applications in a research environment.

With respect to the other full-featured frameworks, OpenRDK's most unique features are the multi-threaded multi-processes structure and the blackboard-type inter-module communication and data sharing. These allow to seamlessly distribute the computation among several hosts in a transparent way and encourage the users to develop many small decoupled modules with well-defined capabilities.

The most immediate future work is to extend the Quality of Service (QoS) settings provided by the OpenRDK, regarding network property sharing, to include additional features defined by the DDS specification. For example, at the moment OpenRDK communication happens either over TCP or UDP; in the case of noisy wireless networks neither behave well, as UDP messages are simply lost, and TCP keeps resending old data (thus aggravating the network overload). It would be useful to have a mechanism similar to DDS's "latency budget", which keeps resending data only for a fixed period of time. Another useful DDS notion is the set of rules for detecting whether a peer is not reachable anymore and acting consequently.

The QoS approach can be extended to other forms of computation on robotic platforms. There are many algorithms, whose output has a quality that can be tuned: for example, with particle filters one can have more precise estimates by using more particles, in RRT path-planning one can find shorter paths by expanding more nodes. Integrating the"computation QoS" in the framework, by providing some means for the modules to declare a QoS, seems particularly interesting because it addresses a need which is very particular to robotics.

Finally, we are working at tools for the analysis of configuration files, that will be very useful for detecting critical situations (such as possible deadlocks on resources) and verify properties of the applications (for example, constraints on the schedule and on the activation of modules).

In conclusion, we are not worried by the apparent proliferation of robotic frameworks. The fact that there are currently lots of frameworks projects available is not necessarily a bad thing, it just means that the field is alive and well and that there are many directions being explored.

References

[1] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. RT-middleware: distributed component middleware for RT (robot technology). In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, pages 3933–3938, August 2005.

[2] Davide Brugali. *Software Engineering for Experimental Robotics (Springer Tracts in Advanced Robotics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[3] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings of Int. Conf. of Robotics and Automation (ICRA'01)*, pages 2523–2528. IEEE, 2001.

[4] Daniele Calisi, Alessandro Farinelli, Giorgio Grisetti, Luca Iocchi, Daniele Nardi, S. Pellegrini, D. Tipaldi, and Vittorio A. Ziparo. Contextualization in mobile robots. ICRA'07 Workshop on Semantic Information in Robotics, 2007.

[5] T.H.J. Collet, B.A. MacDonald, and B.P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005)*, December 2005.

[6] Carle Cotè, Yannick Brosseau, Dominic Letourneau, Clément Raïevsky, and Francois Michaud. Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1):55–60, March 2006.

[7] A. Farinelli, G. Grisetti, and L. Iocchi. SPQR-RDK: a modular framework for programming mobile robots. In D. Nardi et al., editors, *Proc. of Int. RoboCup Symposium 2004*, pages 653–660, Heidelberg, 2005. Springer Verlag. ISBN: 3-540-25046-8.

[8] Donald E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley, 1973–.

[9] A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for robotics. In *Int. Conf. on Intelligent Robots and Systems (IROS'06), Workshop on Robotic Standardization*, December 2006.

[10] I.A. Nesnas. Claraty: A collaborative software for advancing robotic technologies. In *Proc. of NASA Science and Technology Conference*, June 2007.

[11] S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transaction on Robotics and Automation*, 18:493–497, August 2002.

[12] IEC TC65/WG6. *IEC 61499-1: Function Blocks Part 1: Architecture*. International Electrotechnical Commission, Geneva, Switzerland, 2005.