RSCA: Middleware Supporting Dynamic Reconfiguration of Embedded Software on the Distributed URC Robot Platform

Jaesoo Lee, Jiyong Park, Seunghyun Han, and Seongsoo Hong

School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{ jslee, parkjy, shhan, sshong}@redwood.snu.ac.kr

Abstract – *The URC robot project has recently been launched with the aims of popularizing robot systems and putting them to practical use by promoting technical innovations in home service robots. In this paper, we propose a standard software architecture, RSCA, for URC robot applications. RSCA provides a standard operating environment for robot applications together with a framework which expedites the development of such applications. The operating environment of RSCA is comprised of a real-time operating system, distribution middleware, and deployment middleware, which collectively form a hierarchical structure. First, the RTOS is a minimal abstraction layer able to flexibly execute robot applications on an assemblage of diverse hardware devices. The distribution middleware provides an abstraction layer to the applications running in a heterogeneous distributed environment by hiding that heterogeneity and the distribution characteristics from application developers. Finally, the deployment middleware supports reconfiguration of component-based robot applications including such functions as installation, creation, start, stop, tear-down, and un-installation.*

Keywords: URC robot, distributed real-time systems, middleware, software component model, reconfiguration

1 Introduction

In the latter half of the 20th century, the IT field has seen the gradual convergence of computers, home appliances, and information technology. No longer confined to the desktop, computing and information technology is increasingly integrated into a range of common household devices. As a result, the field of Information Appliances has emerged with a number of Korean companies at its forefront. This type of technology convergence has only intensified in recent years. A representative example of this has been the automotive industry where electronics, information, and software technologies are unified with automatic control and mechatronics technologies. Following this trend, robots are now also being thought of as a staging ground for the convergence of various disciplines and devices.

Robotics is a field which could benefit tremendously from the efficient integration of disparate technologies. While the usefulness of an intelligent robot has been evident for a long time, its emergence as a common household device has been painfully slow. This is due in part to the variety of technologies involved in creating an effective robot. A single modern robot is often actually a small, self-contained distributed system, typically composed of a number of embedded processors, hardware devices, communication buses and computers. The logistics behind integrating these devices are dauntingly complex, especially if the robot is to interface with other household devices. The ever-falling prices of high performance CPUs and the evolution of communication technology has made the realization of robots' potential closer than ever. What is left is to address the complexities of robotic technology convergence.

At the head of this effort is the URC Robot project. Under development since last year, the Ubiquitous Robotics Companion (URC) has been conceived as "a robot friend to help us anywhere, anytime." The project's aim is to improve robot technology and facilitate the spread of robot use by making them more cost-effective and practical. Specifically to that end, it has been proposed that a robot's most complex calculations be handled by a high-performance remote server which is connected via a Broadband Communication Network (BcN). For example, the vision or navigation systems which need a high performance MPU or DSP would be implemented on a remote server, and the robot itself would act as a sort of thin client, making it cheaper and more lightweight.

But this type of system demands a very sophisticated software platform to operate it. In order to make the logistics of such a system manageable, the software should provide (1) a framework in which programs can be executed in a distributed environment, (2) dynamic deployment by which a program can be dynamically loaded and reconfigured, (3) real time capabilities allowing the robot software to meet hard deadlines, (4) QoS which can support the robot's vision and voice processing, and (5) fault tolerance which makes software reliable even in adverse circumstances. In addition, the robot software should be flexible enough to use the very limited resources

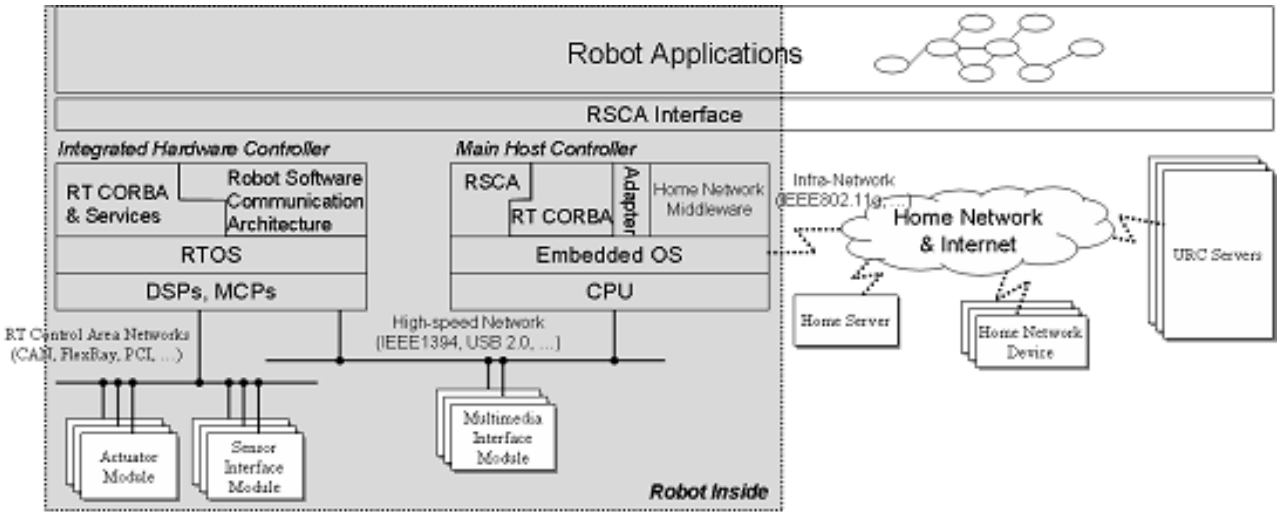

Figure 1 Structure of Hardware and Software on URC Robot

and heterogeneous hardware inherent to most modern robot systems.

Beyond simply creating such a platform, the desired goal is to create a standard which could serve the robotics community at large. Recently there has been a great deal of research activity in this area, and yet there is still no current standard which has garnered international approval. In this paper, we will describe the Robot Software Communications Architecture (RSCA). We will show how this architecture, developed as part of the URC Robot research, meets the needs of the URC Robot and robots in general.

2 Overview of the URC Robot and System Software

Among the most essential properties of the URC Robot are (1) that it can utilize a high capacity server, provided by a URC service vendor, and (2) that it can interface with various smart home-appliances and sensor networks, which are connected to a larger home network. That is, the URC Robot is not a system consisting of only robots but rather an overall distributed system including the URC server and various home network appliances.

The robot's application software has to be designed and implemented in accordance with the requirements of a distributed system which consists of various hardware nodes. Hence both reconfigurability and flexibility are needed, and the application software must be configured via a component-oriented software model which addresses those issues. Furthermore we need a system software structure which acts as a framework to support the distributed component software model. RSCA is the proposed standard system software for the URC Robot.

In this chapter, we first look into the hardware structure of the URC Robot. And then we describe the properties which the system software of the URC Robot must have in supporting that structure.

2.1 Hardware structure of URC Robot

Figure 1 depicts the hardware and software structure of the URC Robot. First we shall look at the hardware structure.

There is a MHC (Main Host Controller) and many IHCs (Integrated Hardware Controller) in the URC Robot. The MHC acts as the interface within each of the robot users. The IHC interfaces the DSP, FPGA and ASIC which receive input from sensors and control the motion of the actuators.

The IHC and MHC exchange relatively huge amounts of data such as video information, so they must be connected by a high bandwidth communication medium like Giga-bit Ethernet, USB 2.0 or IEEE1394. The IHC and sensors like DSP along with actuator interfaces are connected by a real-time control communication line such as CAN or FlexRay.

2.2 System software structure of URC Robot

The URC Robot must have a structure which can support the following application properties:

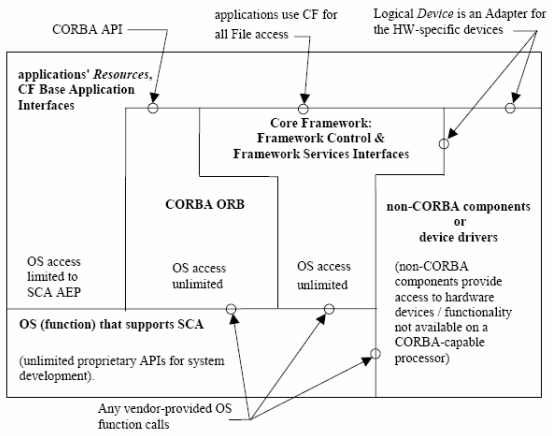

Figure 2 Overview of Operating Environment of RSCA

- heterogeneous distributed computing
- dynamic system reconfiguration
- QoS and real-time guarantees
- heterogeneous resource management and resource constraints

Figure 2 depicts the relationship between the application software and the operating environment of the robot. The operating environment, the system software of the robot, consists of a software-placement middleware called the core framework, the distribution middleware RT-CORBA[1] , and a real-time operating system(RTOS).

The application software which runs in an operating environment such as this must be programmed using the services provided by this operating environment. That is, application software accesses files through the file system service supported by the core framework, and makes use of the standard interfaces and services provided by RT-CORBA and RTOS. In providing a common platform to robot software, the standardized operating environment maximizes the reusability of robot applications.

2.3 Application software structure of URC Robot

Figure 3 depicts the most typical hybrid architecture [2][3] of robot software. Much of current robot software is based on a model which resembles the behavior of animals. The hybrid architecture in Figure 3 has been devised for adding the human-like abilities of planning and deliberation to that model.

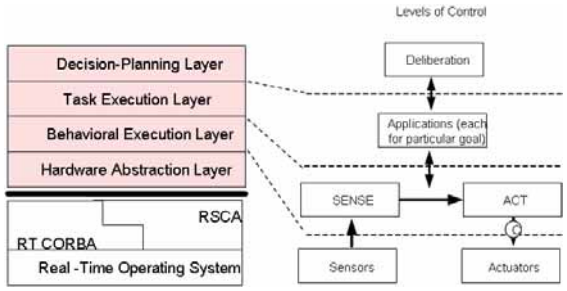

Figure 3 Application Structure of Hybrid Robot

The Decision-Planning Layer decides the next goal of the robot through consideration of the robot's current internal and environmental state, and through interactions with the users. The robot's goal is selected by a set of conditions and that selected goal triggers the behaviors which are appropriate to achieve that goal. Sometimes a goal is associated with the value achievable when the goal has been accomplished. These associated values also affect the decision process.

The Task Execution layer is called the Coordination layer or Sequencing layer. It consists of a set of behaviors to achieve a specific goal. For example, let's assume that a software module in the Decision-Planning layer has received instructions to clean the user's house. First of all, the robot must schedule the order in which rooms are cleaned. And then the robot must move into each room and clean that room in an efficient way according to the structure of the room. The Task Execution layer consists of parallel or sequential actions, conditions for switching between actions, and the instruction and setting values to be passed down to lower layer modules.

The Behavioral Execution layer consists of the basic behaviors of the robot. A behavior is treated as a software module that produces results after processing inputs. It is also a basic unit of the robot software. A behavioral module can receive input from and send output to the hardware abstraction layer, or it can also receive input from and send output to other Behavior Execution layer modules. As such the Behavior Execution layer represents a network of interconnected behavior modules.

The Hardware Abstraction layer implements software modules that control the sensors attached to the robot in addition to input/output devices including actuators. In some cases through one software module we can control many hardware devices, thus allowing high level control instruction instead of direct control on a lower level. For example instead of controlling a motor by specifying PWM, we can create an implementation which allows us to control it by something more intuitive and abstract such as RPM.

The Application software of a robot must be designed and implemented according to the special demands of a distributed robot system made up of many hardware nodes. Hence, reconfigurability and flexibility are paramount. To allow this, application software must be constructed according to a component-oriented software model. Basically, the URC Robot system software structure described in the previous section will be the framework supporting this distributed component software model.

RT-CORBA middleware is the essential feature that makes it possible to construct a distributed, component-based architecture. RT-CORBA plays the most important role in the communication of distributed objects in that it hides the fact that they are distributed and heterogeneous.

3 The Design of RTOS

The most fundamental part of the URC robot core system software is the RTOS (Real-Time Operating System). In this section we will show the concepts and functions of general RTOS, RTOS design for the URC robot, and IEEE POSIX.13[4], which is selected as the OS standard in RSCA.

3.1 Concepts and roles of RTOS

A RTOS is an OS (Operating System) for a Real-Time system, designed to allow the system to generate correct results in given time. The main functionalities of RTOS in a system are support for Real-Time tasks, abstraction of hardware properties, and efficient resource management. Additionally the RTOS performs various functions like providing a file system, memory allocation, and network protocol processing.

Table 1 IEEE POSIX.13 System Profile

	PSE51	PSE52	PSE53	PSES54
	Minimum Real-Time system profile	Real-Time control system profile	Dedicated Real-Time system profile	Multi-purpose Real-Time system profile
Target system	Controlling one or more specific I/O devices without user's intervention			Using together with Real-Time and Non Real-Time tasks, Interaction with users
Target hardware	Uniprocessor with memory. no MMU	Multiprocessor supporting. MMU possible		High-speed storage devices, display, network supporting devices. MMU is used.
Multitasking	UniProcess (Multi thread)	Multiprocess		
File system	No	File system interface support	FS without directory hierarchy	File system interface support

3.2 RTOS design for URC Robot

The first step of RTOS design is task modeling and the assignment of an appropriate scheduling method. URC robot software is a set of distributed, interacting tasks which are in charge of operations such as reading sensor values periodically or handling aperiodic events in real-time. So the URC robot system consists of periodic and aperiodic tasks and each task has hard dead line.

For these task models, a scheduling algorithm needs to be different from existing RTOSes. Usually a fixed priority based scheduling algorithm or a dynamic priority based scheduling algorithm is used in a Real-Time system. RM (Rate-Monotonic)[5] and EDF(Earliest Deadline First) [6] are typical examples of those. These algorithms primarily consider periodic tasks and have defects like a lack consideration for aperiodic tasks and accepting just soft real-time scheduling. So in order to satisfy the requirements of the hard real-time periodic and aperiodic tasks of the URC robot, we have to use an improved scheduling algorithm. The author of [7] suggests a scheduling algorithm which not only improves upon the EDF algorithm but also integrates support for aperiodic tasks requiring hard Real-Time deadline guarantees.

Furthermore a RTOS needs a protocol for resource sharing between tasks. But if this is permitted, priority inversions may occur. So we need an algorithm to resolve this problem. Authors of [8] suggest PCP (Priority Ceiling Protocol) as a solution to the problem. But because PCP needs too much run time overhead in reality, IIP (Immediate Inheritance Protocol) was devised. IIP resolves resource sharing issues with low overhead and reduces the number of context switches.

For the control of various hardware in the URC robot system, the RTOS must provide a hardware abstraction layer. And efficient hardware resource management is requisite. URC robot hardware consists of many processors such as antennas, RF modems, DSP, sensors, motor controllers and so on. If application programs control these devices directly without the aid of a RTOS, it is a rather unreasonable situation in terms of flexibility, scalability, maintenance, and reliability of software. And in embedded systems which have stringent restrictions on hardware resources, efficient resource management is essential.

Lastly a RTOS for the URC robot should support various network protocols. For a robot to provide services via external internet servers, it needs network protocols like HTTP and TCP/IP. And it needs UDP protocol for handling streaming data or sensor data with a low overhead. As MHC (Main Host Controller), IHC (Integrated Hardware Controller), and DSP in URC are connected through IEEE1394, CAN, TTP, and FlexRay, RTOS should support these protocols and satisfy Real-Time restrictions for communication between nodes.

3.3 RTOS standard for URC robot

In order to guarantee the portability and inter-operability of URC robot application programs, a standardized RTOS interface is needed. As a common interface for the RTOS, POSIX, as defined by IEEE, deserves much consideration. POSIX is an abbreviation for 'Portable Operating System Interface (into a UNIX like kernel)' and is the de-facto standard API for UNIX-like operating systems. An embedded system's RTOS does not need to support all of the POSIX API, but rather is better-suited supporting a subset of POSIX. So we need a standard for an appropriate subset of the POSIX API and POSIX.13 Real-Time system profile [4] is such a standard. POSIX.13 defines subsets of the POSIX API for each Real-Time system characteristic. It consists of 4 profiles (Table 1), PSE51 to PSE54 (table 1). The URC robot system needs an RTOS compliant with a standard higher than PSE52, which provides multithreaded capabilities and a simple file system, to fully support distribution middleware as explained later.

4 The Design of Middleware

The middleware part of the URC Robot system software is comprised of distribution and deployment middleware as we see in figure 2.

4.1 Distribution middleware (CORBA)

We will explore the concepts and the role of distribution middleware as a part of the URC robot system software and investigate CORBA as a RSCA proposed standard in the following section.

4.1.1 Primary roles of distribution middleware

■ **Resolving the problems of heterogeneity in a distributed system.**

It is very difficult to write a program which will be run on a distributed system such as the URC robot. The difficulty lies in the heterogeneity of the nodes which form the distributed system. The heterogeneity means a variety of hardware, operating systems, network protocols, and programming languages.

The most effective solution to this problem is distribution middleware. In other words, if a program is written based on middleware, it can run the same way regardless of which kind of processing node it is executed on. So when writing a program that will run on a distributed system which takes advantage of middleware, we need not consider the heterogeneity of following items.

- hardware (ex. big-endian and little-endian)
- OS (ex. Win32API and Linux POSIX API)
- network bus protocol (ex. IEEE 802.3 and CAN bus)
- network software protocol (ex. TCP/IP and UDP/IP)
- programming language (ex. C++ and JAVA)

■ **Offering various services needed in a distributed environment**

When we want to make a program run in a distributed environment, we should consider many aspects of the program itself. The most representative example is that the program should be able to know which node the entity to communicate with is in. This is because the node location of an entity (a program or an object) is not determined statically, but dynamically according to system states. And we should consider the case that a node is inactive. In a distributed system, it is often the case that it is impossible to communicate with another node due to various reasons such as network or hardware problems in the node. In this case, an ideal distribution middleware will be able to deal with the problem elegantly.

Middleware provides the following services to handle these issues.

- Naming Service : a service which gives names to objects on distributed nodes and makes it possible to get a reference to an object using only that name.
- Event Service : a service which makes it known when a specific event occurs on a remote node.
- Time Service : a service which synchronizes the clocks of nodes in a distributed system.

In addition, there are other various services such as the Security Service, Trading Service, Relationship Service, Property Service, etc.

4.1.2

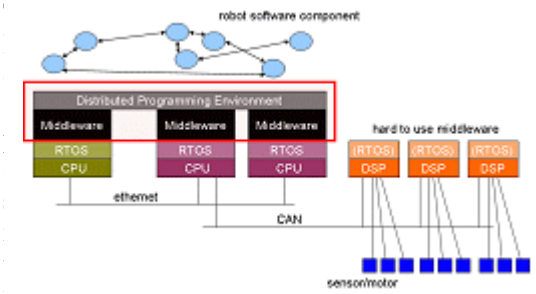

Figure 4 Middleware in URC Robot

of a distribution middleware in the URC Robot

The distribution middleware of the URC robot is executed only on MHC and IHC nodes. The distribution middleware can not be executed on DSP nodes. This is because a DSP node has insufficient processing power to smoothly execute complex software such as distribution middleware. And because the middleware was originally developed for general processors, it is difficult to port it to a special purpose processor such as DSP. So, middleware is not used on DSP nodes. The situation is shown in Figure 4.

4.1.3 URC distribution middleware

The Common Object Request Broker (CORBA)[9] is used as a distribution middleware for URC. CORBA is a standard for distribution middleware defined by the Object Management Group (OMG). Currently the latest version of CORBA is version 3, and this has RT-CORBA facilities for distributed real time environments. The core concepts of the ORB, IOP, and stub objects are described here.

(1) ORB and IOP

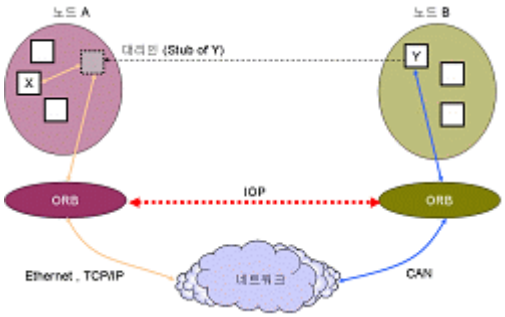

Figure 5 Basic Concept of CORBA

The ORB is a protocol translator. An ORB exists on each node and an entity should send a message to the ORB to communicate with other objects on other nodes. But the mechanisms behind using the ORB and the formats of message sent to the ORB are different from node to node. This is due to the fact that each node may have a different endian byte order, programming language, communication protocol, or operating system. But the ORB gets messages, translates them according to a standardized mechanism called IOP, and then sends them to the network. Then the ORB on the receiving node translates the messages

according that node's properties and then sends them to the appropriate object. The functions of the ORB and IOP are depicted in Figure 5.

(2) Stub object

CORBA also includes the concept of stub objects. As shown in figure 5, if object X would communicate with object Y, object X needs on its node Y's stub object. Stub objects act as a proxy for an object. In order to communicate, object X considers Y's stub as Y and sends messages accordingly. And then Y's stub object transfers those messages to the actual Y through the ORB.

4.2 RSCA core framework

In this section we investigate the concepts, roles, and structure of the RSCA core framework which acts as the deployment middleware for the system software of URC.

4.2.1 Concept and role of deployment middleware

URC robot application software consists of application software components which are connected and cooperate with each other as illustrated in Figure 6. Deployment middleware deploys URC robot software consisting of several application software components on each processing node of the URC robot.

Specifically, deployment entails a series of tasks that include determining which robot software module should be executed in which processing node, connecting the robot software modules enabling them to communicate with each other, and starting or stopping the whole URC robot software structure.

The deployment middleware presented in this paper supports the following deployment functionalities:

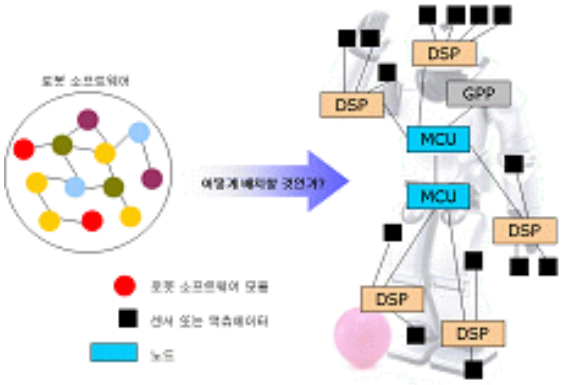

Figure 6 Role of Deployment Middleware

- Install and Uninstall of robot software
- Create and Release of robot software
- Start and Stop of robot software
- Configuration changes to robot software and software modules
- Common services (file system, logging, etc)

4.2.2 URC deployment middleware

Deployment middleware consists of several objects cooperating with each other. The DomainManager object implements installation and un-installation of robot software, the ApplicationFactory object implements creation of robot software, and the application object implements start, stop, and release of robot software. Basic services such as a file system interface and logging are implemented in FileSystem and LogService objects respectively.

Figure 7 illustrates the relationship of interfaces used in URC deployment middleware. Boxes with no lines constitute the core framework interface, which is provided by the deployment middleware. Through these interfaces, a user (or program) can manipulate the deployment middleware to deploy, create, release, or start robot software.

Boxes with lines represent interfaces which URC robot software should provide to the deployment middleware. URC robot software implements these interfaces. Many software modules in robot software are deployed, connected and executed automatically after the domain profile (to be described later) is added on properly.

This section describes the core framework interface and the domain profile.

(1) Core framework interface

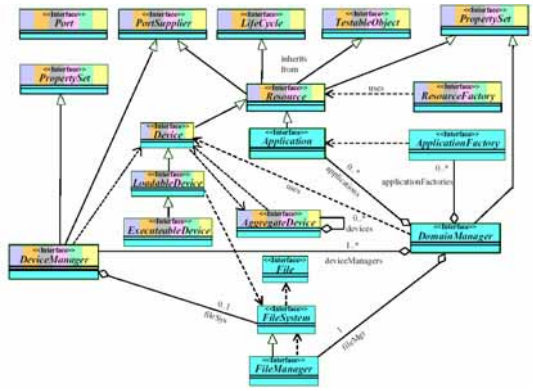

Figure 7 Structure of Deployment Middleware

The DomainManager is the key object in deployment middleware. A DomainManager component manages Applications, ApplicationFactories, hardware devices and DeviceManagers within the system. Hence the DomainManager provides interfaces to install/uninstall software and register/unregister devices or DeviceManagers.

An application is a kind of Resource and consists of one or many software Resources. An application is an object that is automatically created by the deployment middleware when a user installs new software, and provides interfaces by which a user may start or stop the installed software. Each resource in an application needs implementations of its start and stop operations. The LifeCycle interface provides these start/stop interfaces and

Resource and Application objects inherit from it. This means that each of the application components which are distributed throughout multiple nodes can be started or stopped collectively in the distributed environment of a robot.

Device objects represent devices that applications use, which may be simply a device driver for a hardware device, or a logical device of a processing node such as a FPGA, DSP, or GPP. A LoadableDevice is a type of Device which provides interfaces by which codes can be downloaded onto devices. An ExecutableDevice provides interfaces by which the downloaded codes can be executed.

A DeviceManager integrates the management of Devices. (e.g. registering other application components to the DomainManager so that the device can be used by them.) Usually one DeviceManager exists on one processing node and integrates the management of devices which are connected to that node. Instead of having its own, a node such as a DSP or a FPGA in which CORBA is not available due to lack of resources, can use the DomainManager of another node. In this case, a proxy device exists in the node which provides services through communication with servers inside the DSP or FPGA.

A FileManager object manages file system services provided by the deployment middleware, and like a distributed filesystem allows distributed files in the domain to be seen as if in a same address space. Usually a DeviceManager creates a new file system and registers it to the FileManager object, which mounts it in a directory which is assigned the same name as the DeviceManager.

(2) Domain profile

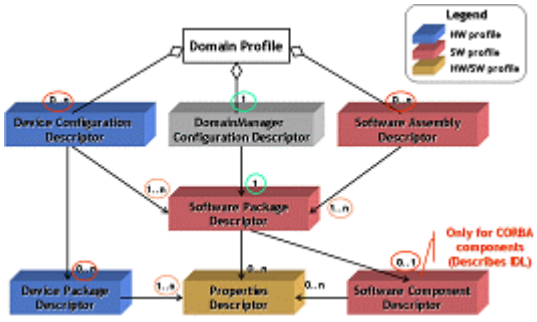

Figure 8 Structure of Domain Profile

The Domain Profile consists of XML descriptor files which describe properties of the hardware and software in the URC robot system, as depicted in Figure 8. Information about installed hardware / software components can be built from the domain profile. There also can be additional information that describes the combination of components when installing new software.

A Device Configuration Descriptor describes the configuration of hardware, and a Software Assembly Descriptor describes the configuration of software. These descriptors consist of one or more software package descriptors, which describe software components or individual hardware devices. Software package descriptors that describe DomainManager components are included in the DomainManager Configuration Descriptor, which also describes services used. A Properties Descriptor describes re-configurable properties, initial values, and executable parameters provided by individual components.

(3) Feature and functionality of URC deployment middleware

URC deployment middleware provides (1) heterogeneous distributed computing, (2) dynamic system reconfiguration, (3) QoS and real-time guarantees, and (4) heterogeneous resource management. This section describes how these features can be provided.

■ **Heterogeneous distributed computing**

With the support for a heterogeneous systems provided by CORBA, URC deployment middleware describes the system's features in the domain profile so that they can be applied at deployment time. A given component can have more than one implementation according to characteristics of the system, so its information is described in software package descriptors and applied at deployment time. For example the vision component of a robot can include an implementation executed in JVM, an implementation executed in Windows XP, and an implementation executed in Linux on a SPARC machine. The Deployment middleware selects one of them and deploys it at the appropriate processing node.

Moreover, URC deployment middleware hides distributed features from applications by having distributed nodes be seen as a single virtual system (domain.) Robot applications don't need to consider how many processing nodes the domain consists of, or how they are connected to each other.

■ **Dynamic system reconfiguration**

The area of system reconfiguration can be divided into individual component reconfiguration, application reconfiguration, and deployment time reconfiguration. Individual component reconfiguration is made possible by URC deployment middleware by providing reconfigurable parameters as stereotypes, and including various component implementations that satisfy wanted constraints (e.g., Hardware features, memory and performance requirements, QoS and real-time requirements, etc.) Among URC deployment middleware interfaces, PropertySet provides reconfigurable parameters as stereotypes.

Application reconfiguration is provided by the software assembly descriptor, which describes various available configurations of applications. Application developers can vary configurations and parameters of components to make software assemblies satisfy various requirements and system features. URC deployment middleware chooses one of these assemblies as is appropriate to current resource conditions.

Deployment time reconfiguration is provided by the ApplicationFactory, which searches for the most suitable

processing units (devices) that can satisfy the system dependence and resource requirements of a given component.

- **QoS and real-time guarantees**

The deployment middleware, distribution middleware and RTOS all support QoS and real-time guarantees. The distribution middleware and the RTOS also support individual component's resource requirements and real-time guarantees. On the other hand, the deployment middleware supports QoS and real-time guarantees for the whole system which is a combination of individual components.

The ApplicationFactory searches for the most suitable deployment based on resource requirements and system dependencies, described in a software component descriptor. It then reserves the required resources for each device to satisfy QoS and real-time guarantees described by the software assembly descriptor in the domain profile.

- **Heterogeneous resource management**

URC deployment middleware supports heterogeneous resource management via the Device interface. The Device interface provides interfaces that allocate and free a certain capacity, such as a memory requirement, CPU requirement, or bandwidth requirement. It also supports synchronization of resource usage by providing the resource usage status and management status. Of course, a user should choose and implement how the resources are allocated and synchronized, therefore determining the efficiency of resource usage.

5 Example of Developing Robot Software Using RSCA

To aid the reader in understanding RSCA, here we will describe the development process for a simple robot application as an example. The robot development process consists of two phases, one of which is the robot hardware development process, and the other is the robot software development process. Now we will assume the following robot hardware has already been developed.

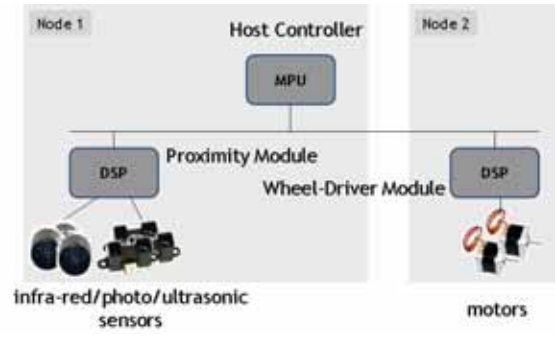

Figure 9 Composition of the Robot Hardware

As shown in Figure 9, the example robot has three processors. SensorModule consists of distance estimation sensors (ex. ultrasonic, infrared sensors) and a DSP which controls them. ActuatorModule also consists of a DSP and a motor which actuates the robot. HostController consists of I/O devices (ex. LCD, touch panel, touch sensor, speaker, mike, and etc.) for Human-Robot Interface (HRI) and an MPU which controls them. These SensorModule, ActuatorModule, and HostController processors communicate with each other through a bus.

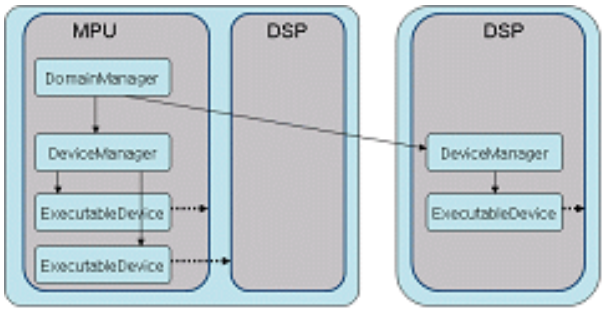

Figure 10 Initial Configuration State of RSCA

For the robot hardware described above, at first, RSCA should be implemented as a real time operating system, a distributed controlling middleware, etc. Figure 10 shows the initially configured state in which the DomainManager and DeviceManager are implemented. The DomainManager which takes care of the whole robot system is implemented in Node1 (a logical node which covers HostController and SensorModule, refer to Figure 9), and a DeviceManager which controls devices connected to a node is implemented in each node. In this system because the SensorModule's resources are too scarce to implement CORBA or deployment middleware, HostController controls the DSP-connected devices and the DSP itself in the SensorModule. So Node1 has 2 program execution devices MPU and DSP, and Node2 (a logical node for ActuatorModule) has only one DSP for program execution. The relationship between the file system and the file manager is omitted for convenience.

In this chapter we will describe the development process for a simple robot application which will make the aforementioned robot move in any direction. The robot application decides which direction the robot will move, and changes the direction periodically. The application does not move the robot just in any direction, but rather when it detects an obstacle, it should evade the obstacle.

Below, we will introduce the whole development process using RSCA from the view point of developers who might participate in the development of the robot described above.

5.1 Role assignment

In the RSCA robot software development process, three kinds of developers participate: hardware device developers, application component developers, and application developers. And they develop different parts of the robot software. Figure 11 shows the role assignment among the three kinds of developers.

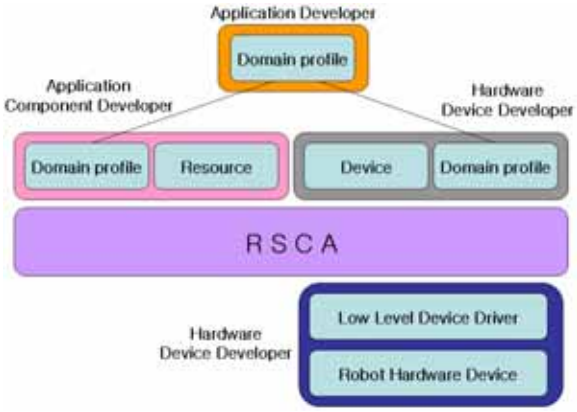

Figure 11 Roles of Robot Application Developers

At first, hardware device developers provide the robot hardware devices with the device drivers which control them. Hardware developers also provide the logical device software which gives RSCA device interfaces. And this kind of logical device software is described separately by the domain profile.

Application component developers develop each component which constitutes an application. Examples of these components are a vision processing component, a navigation component, a component which finds a way in a labyrinth, and so on. These components should provide RSCA with a resource interface. Application components also need a domain profile described separately. This domain profile describes the requirements (CPU, memory requirements, hardware, operating system version, etc.) which are needed to execute the components.

Application developers are the ones who develop the robot application in the final phase. They choose application components which are appropriate for a specific robot application, then compose and configure them. Application developers must describe these components, their composition, and their configuration in the domain profile.

5.2 Hardware device developer

Hardware device developers work with hardware devices attached to the robot. Motors, photo sensors, infrared sensors, and speakers are all examples of hardware devices. When one constructs a robot, one buys the hardware devices from a hardware device developer and assembles them.

Hardware device developers provide not only physical robot hardware but also the software behind it. Additional software allows a hardware device to be recognized as a logical device by an application developer. To make higher-level software, we first need a device driver to control robot hardware. Usually, device drivers are made as Operating System dependent modules.

Next, we need a wrapper which allows a device driver to be recognized as a logical device by robot application developers. Through that wrapper, RSCA can control different devices in a uniform way. RSCA gives a detailed definition of the interface that the wrapper must provide. For basic implementation of these interfaces, the Device interface is provided. If the target device can load or execute software just like a CPU or DSP, then the interface for LoadableDevice or ExecutableDevice is provided.

At the same time, characteristics of each Device interface have to be described by the domain profile. The domain profile that used in this case is a SPD (Software Package Descriptor). The SPD describes the environment (OS, CPU architecture, CPU demand, memory demands, programming language) in which the Device interface is executed and specifies additional options (the sensing period of a sensor for example) which the Device can set.

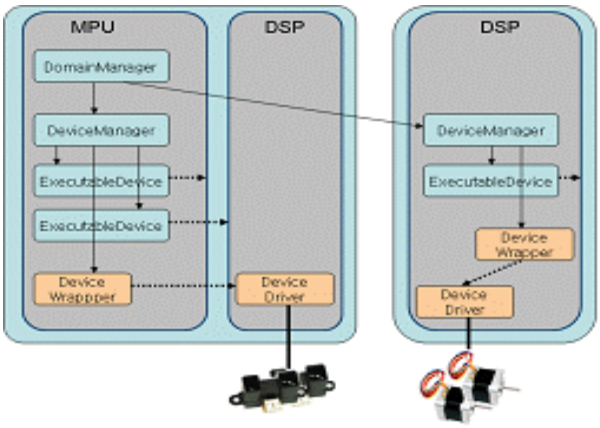

Figure 12 Components after Loading Hardware Devices

In the example of robot software development above, the hardware developer first provides a sensor and a motor to the robot. Various sensors, such as ultra sonic sensors and infrared sensors can be used as obstacle avoidance sensors. A variety of motors can be used as the motor. The hardware developer has to provide not only the physical device but also a low-level device driver, Device wrapper and Domain Profile. The low-level device driver is code which is executed on the DSP. This relationship is represented by Figure 12. And the Domain Profile provides the information outlined in Figure 13. A total of four software components are needed, each describing the name of the executable file, the executable CPU architecture, and the CPU power needs. In addition, optional parameters like a sensing period, sensing sensitivity and a motor control period are specified in Domain Profile. RSCA checks this information and loads four software components in the

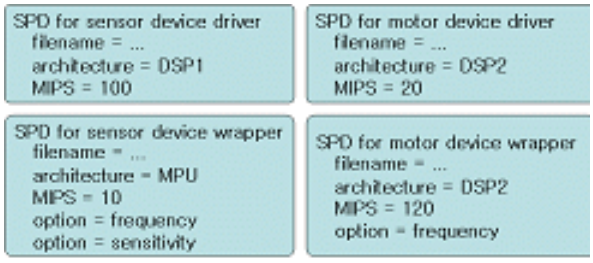

Figure 13 Domain Profiles of Software Components for Hardware Devices

appropriate nodes.

5.3 Application component developer

An application component developer creates the components which make up a robot application. A robot application is composed of a number of cooperative components. Each component can be made by an independent developer.

RSCA doesn't define the internal structure of a component. Even the connections or communication between components do not need to be composed by RSCA (through CORBA). In this case, components lose generality, but for the purpose of optimizing communication speed it may be necessary.

But components have to have a minimal external structure which is specified by RSCA. In other words, a component has to provide a minimal Resource interface. A Resource interface is needed by RSCA to handle each application component with uniformly. A Resource interface includes an interface for component lifecycle management, reconfiguration management and connection management.

An Application component must also be described by the Domain Profile. The Domain Profile used in this case is also a SPD. A SPD describes the environment (OS, CPU architecture, CPU demand) in which an application component is executed and specifies additional configurable options (sensing period of a sensor) which are provided by the component.

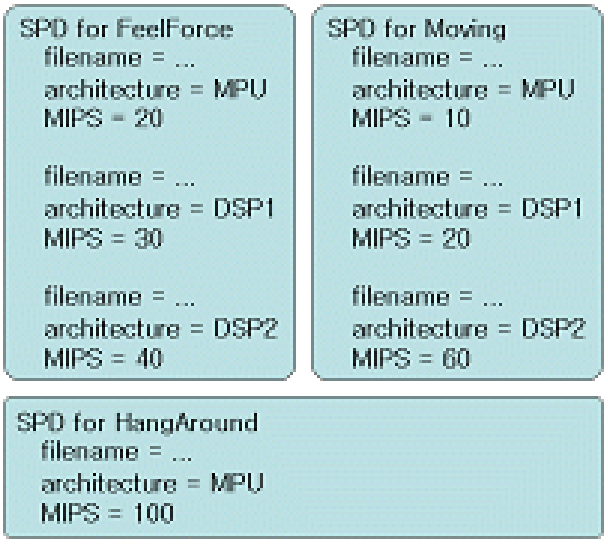

Figure 14 Domain Profiles for Robot Software Components

In the example above, the software component developer develops three important functions as components: FeelForce, Moving, and HangAround. FeelForce is a component which decides whether an obstacle exists or not by using data from the distance detecting sensor. Moving is a component to control the motor device by direction (forward, backward, left, right) and velocity. HangAround in turn uses these two components. HangAround decides the proper direction with information from the FeelForce component and moves the robot in that direction using the moving component.

These three components are described by the Domain Profile as in Figure 14. Moving and FeelForce components are compiled for MPU, DSP1, and DSP2 compatibility. But let's assume that the HangAround component is compiled only for MPU, because there is some restriction which prevents compilation on the other devices. Then, RSCA must decide where these components will execute from among MPU, DSP1, and DSP2, considering the constraints imposed by CPU power.

5.4 Application developer

An application developer organizes the hardware device wrappers and software components and develops an application which may do independent work. Software components can be reused, be purchased or sold from a person or company, or be specially developed in-house.

The connections between components are described by the SAD (Software Assembly Descriptor) of the Domain Profile. The SAD describes the type of components involved in an instance of an application, their initial values, and most importantly the connections between each component instance. In describing a component, the SDP file path must be noted. Also, when describing the connection relationships, the instance name and method to get the instance must be noted. For example, if one uses the CORBA naming service, then the name which is used to register must be described.

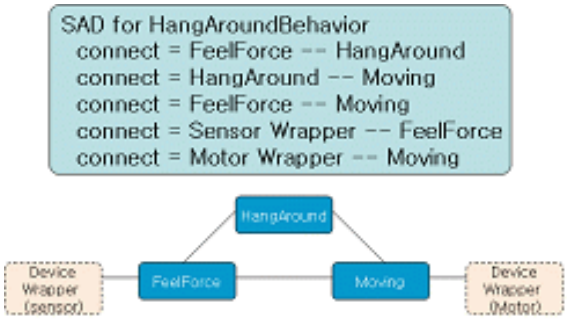

Figure 15 Domain Profile and Connections of Each Robot Software Components of HangAroundBehavior Application

Last, we look at the role of the application developer in this example of robot software development. The application developer wants to make a robot application which will allow the robot to avoid collision and wander using the aforementioned three components. Let's call the name of the application HangAroundBehavior. In this application, three components will be connected in the following way. The HangAround component connects with FeelForce and the Moving component. FeelForce is connected directly with the Moving component. This is so that when an obstacle appears in the neighborhood

suddenly, the robot can stop or change path quickly using the Moving component. The application developer must describe this connection using the SAD in the Domain Profile. This procedure is represented by Figure 15. The Domain Profile shown is truncated to show only necessary information.

Figure 16 shows the final deployment of the HangAroundBehavior application.

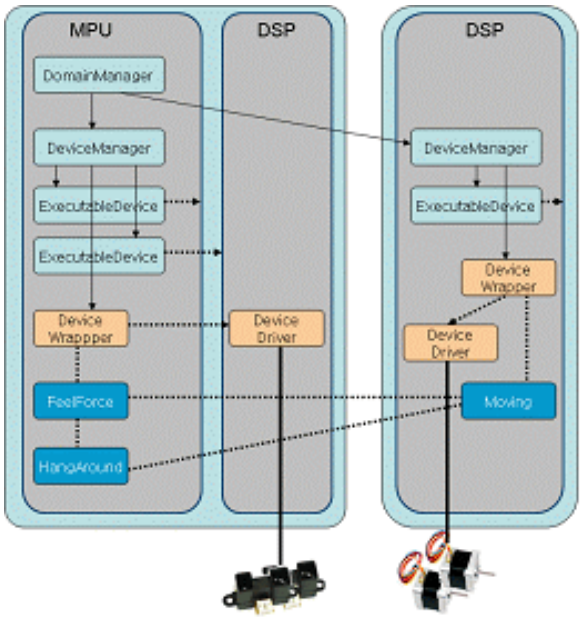

Figure 16 Final Deployment of HangAroundBehavior Application

6 Related Work

So far, the research on Robot software architecture has been mainly focused on the application software framework ([10][11][12][13]) with the goal of providing assistance in programming robot applications themselves. But because robots have become extremely distributed there has arisen a need to overcome the inherent complexity of software and shorten the time-to-market. So standardized system software architecture and a framework which supports component-based application development have become highly desirable, together with the above application software framework.

There is Commercial Off-The-Shelf (COTS) distribution middleware like CORBA, DCOM, Java RMI, and .NET which can support decentralization and openness. But it was the general view that this middleware has performance and memory overhead which outweighs the benefits they provide. Thus, there are examples of robots that use RPC-level middleware which have a only part of the functionality of those middleware ([14][15] [18]).

DROS [14] provides a naming service using NameServer to get a remote component handle and the abstraction layer of the communication protocol called GeneralComms. Service calls for the handle finally reach the remote component through GeneralComms, as transmission protocols TCP, UDP, shared memory, etc. can be used. In addition to the abstraction of multi-tasking, there are database and StateManager services provided by the framework.

Connexis[15] is a RoseRT[16] plug-in module made by the Rational company. Originally RoseRT is a tool that provides a development environment for UML-RT[17] modeling of embedded real time systems and also makes binary code which can be executed on a target system, but it does not support distributed systems. Connexis addresses that very issue. A system of communication using ports between components on the different nodes is achieved by the Transportation Integration framework which Connexis provides.

But there are problems in that these are not standardized and the facilities are very restricted and inflexible. Also, the evolution of hardware technology and continually falling prices make the performance and memory overhead of existing COTS distribution middleware negligible, so the research directed at using this COTS distribution middleware has grown recently.

MIRO[19] and OCP[20] are both examples of this. MIRO has three layers: a Device layer for OS-level abstraction of hardware, a Service layer for CORBA component-level abstraction, and the framework layer for application components. OCP is a software framework for control applications and utilizes RT-CORBA as a distribution middleware. The most important characteristics of OCP are that there is real time facility support for applications, and that OCP guarantees QoS for control applications by the mechanisms such as compromise, feedback control, etc. OCP also describes the parameters and connections between components and the QoS parameters by XML.

In the meantime, SCA[21] is a standard system software architecture for Software Defined Radio (SDR) applications. SCA specifies a standard for real time operating systems, distribution middleware and deployment middleware as an operating environment for applications. Specifically it specifies IEEE POSIX 1003.13 PSE52 or higher as a real time operating system, minimum CORBA or higher as a distribution middleware, and describes the standard specification of SCA core framework as a deployment middleware. The RSCA of this paper is based on SCA.

7 Conclusion

In this paper, we described the RSCA architecture as an integrated middleware which supports the dynamic deployment of embedded software on the URC distributed robot platform. Specifically we described the URC hardware and the application software architecture, and the necessity and functionalities of the RSCA standard operating environment. This RSCA standard operating environment is comprised of the RTOS, RT-CORBA distribution middleware, and the deployment middleware (RSCA core framework).

The RTOS provides the basic abstraction layer which makes robot applications both reliable and flexible on various hardware devices.

The distribution middleware provides an abstraction layer that hides the heterogeneity of the distributed nodes in the URC robot and hides the decentralization of the system, thereby making the distributed application components able to interact with each other flexibly. Also RT-CORBA distribution middleware supports the software component model for the distributed component-oriented computing for robot applications.

The core framework is the deployment middleware of RSCA, supporting reconfigurability for robot applications in addition to the deployment of distributed component-based applications. Thus in the reconfiguration process it is easy to download, install, and remove applications, as well as to build, start, stop and banish applications.

And lastly we described the methods and processes behind developing an application with RSCA. Specifically we explained the roles of the hardware developer, component developer, and application developer and how we may use RSCA.

We built a prototype to verify the usefulness of RSCA in this paper for an actual robot. We used Linux as an RTOS and TAO as an RT-CORBA implementation, both of which are open-source. And we implemented the deployment middleware with C++. Now we are implementing robot applications to confirm the usefulness of RSCA, and doing research on performance problems which may occur.

It is our opinion that RSCA is a core software which meets the requirements of the URC robot. We are now doing research on performance improvement, fault tolerance, security, and QoS for practical applications.

Acknowledgment

The work reported in this paper was supported in part by "Embedded Component Technology and Standardization for URC" project of Ministry of Information and Communication.

References

[1] Object Management Group. " Real-Time CORBA Specification Revision 1.1. " OMG document formal/02-08-02 (August 2002).

[2] Erann Gat. On Three-Layer Architectures. Artificial Intelligence and Mobile Robots, 1997. MIT/AAAI press.

[3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. "An Architecture for Autonomy," the International Journal of Robotics Research Special Issue on "Integrated Architectures for Robot Control and Programming", 1998.

[4] ISO/IEC ISP 15287-2, IEEE Std 1003.13, Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP), February 2000.

[5] J. Lehoczky, L. Sha, and Y. Ding, " The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," In Proceedings of IEEE Real-Time Systems Symposium, pp. 166-171, Dec. 1989.

[6] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," J. ACM, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[7] J. Park, M. Ryu, and S. Hong, " Deterministic and Statistical Admission Control for QoS-Aware Embedded Systems," Journal of Embedded Computing. 2004.

[8] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, 39(3), pp. 1175-1185, Sep. 1990.

[9] Object Management Group. " The Common Object Request Broker Architecture: Core Specification Revision 3.0," December 2002.

[10] Evolution Robotics, "ERSP 3.0 Users Guide," http://www.evolution.com, 2004.

[11] Peter Soetens, "The Complete OROCOS Software Guide," http://www.orocos.org.

[12] ORiN Forum, "Specification of ORiN(Ver. 0.5)," http://jara.jp/e/orin/En_ORiN.pdf.

[13] David J. Miller and R. Charleene Lennox, "An Object-Oriented Environment for Robot System Architectures," IEEE International Conference on Robotics and Automation, Cincinnati, Ohaio, Aug. 13-16, 1990.

[14] Dave's Operating System, http://www.dros.org.

[15] IMB Rational Software Corporation, "Rational Rose RealTime Connexis User Guide: Revision 2003.06.00," 2003.

[16] IBM Rational Software Corporation, "Rational Rose RealTime User Guide: Revision 2001.03.00," 2000.

[17] Unified Modeling Language (UML), http://www.uml.org.

[18] O. Kubitz, M. O. Berger, and R. Stenzel, Client-Servier-Based Mobile Robot Control, IEEE/ASME Transactions on Mechatronics, Vol. 3, No. 2, June 1998.

[19] H. Utz, and et. el., "Miro - middleware for mobile robot applications," IEEE Transactions on Robotics and Automation, Volume: 18, Issue: 4 , Aug. 2002, Pages:493 – 497.

[20] James L. Paunicha, Brian R. Mendel, and David E. Corman, "The OCP – An Open Middleware Solution for Embedded Systems," Proceedings of the American Control Conference, Arlington, VA June 25-27, 2001.

[21] Joint Tactical Radio Systems. "Software Communications Architecture Specification V2.2." November 2002.