OPRoS: A New Component-Based Robot Software Platform

Choulsoo Jang, Seung-Ik Lee, Seung-Woog Jung, Byoungyoul Song,
Rockwon Kim, Sunghoon Kim, and Cheol-Hoon Lee

A component is a reusable and replaceable software module accessed through its interface. Component-based development is expected to shorten the development period, reduce maintenance costs, and improve program reusability and the interoperability of components. This paper proposes a new robot software component platform in order to support the entire process of robot software development. It consists of specifications of a component model, component authoring tool, component composer, and component execution engine. To show its feasibility, this paper presents the analysis results of the component's communication overhead, a comparison with other robotic software platforms, and applications in commercial robots.

Keywords: Robot software, component, component platform, authoring tool, component composer, component execution engine, OPRoS.

Manuscript received Mar. 15, 2010; revised June 4, 2010; accepted June 22, 2010.

This work was supported by the Industrial Foundation Technology Development Program of MKE/KEIT, Rep. of Korea [KI001800, Development of Open Platform for Robotic Services (OPRoS) Technology].

Choulsoo Jang (phone: +82 42 860 6726, email: jangcs@etri.re.kr), Seung-Ik Lee (email: the_silee@etri.re.kr), Seung-Woog Jung (email: swjung@etri.re.kr), Byoungyoul Song (email: sby@etri.re.kr), Rockwon Kim (email: rwkim@etri.re.kr), and Sunghoon Kim (email: saint@etri.re.kr) are with the IT Convergence Technology Research Laboratory, ETRI, Daejeon, Rep. of Korea.

Cheol-Hoon Lee (email: clee@cnu.ac.kr) is with the Department of Computer Engineering, Chungnam National University, Daejeon, Rep. of Korea.

doi: 10.4218/etrij.10.1510.0138

I. Introduction

Not only do robots have many different types of sensors, actuators, and degrees of freedom, but also their services are becoming more and more sophisticated allowing them to run autonomously, while providing complex services in an unknown or partially known environment [1]. Unfortunately, their services are often not reusable even in slightly different application scenarios because they are tied to specific robotic hardware, processing platforms, and communication infrastructures. Also, the assumptions and constraints about tasks and operational environments are hidden and hard coded in the software implementation [2]. This increased complexity has lead to increasing demands for modularity, productivity, reusability, integration, and maintenance.

Component technology seems to be an attractive approach to meet the demands in the robotic software field [3]. A component is a reusable and replaceable software module that enables complex functions to be developed easily. The main focus of component-based development is concerned with the assembly of pre-existing software components into larger pieces. Nevertheless, software reuse and component-based development are not yet state-of-the-art practice software development approaches in robotics [2].

Widely used component technologies such as EJB [4], .NET [5], and the CORBA [6] component model, have paid much attention to business applications. They seem to be relatively heavyweight and complex. Also, they do not address issues such as real-time applications, fault management, or other functionalities that are important for robots.

Recently, some research has actively been conducted on component-based robot software platforms [7]-[19]. They can

be categorized into three groups: the middleware-based component platform, robot device interface platform, and robot software architecture platform.

MSRDS [8], MARIE [9], Miro [10], RT-Middleware [11], OROCOS [12], ROS [13], and PEIS Ecology [14] are middleware-based component platforms that manage components and their communication with their component execution engine for multipurpose robot control software. However, they focus on the middleware framework, so they do not sufficiently support related tools to component development and its simulation. The robot device interface platforms such as Player [15] aim at providing interfaces for accessing robot sensors and actuators over the network. ERSP [16], Urbi [17], MobileRobots [18], and iRobot Aware [19] are robot software architecture platforms that provide layered software architecture.

We argue that a good robot software platform needs to offer much more than pure middleware such that it supports the full development lifecycle for robot software. To meet the above mentioned requirements, this paper proposes a new component technology called open platform for robotic services (OPRoS) [20]. It supports the full development lifecycle for robot software by providing a robot software component model, component execution engine, various middleware services, development tools, and a simulation environment.

The rest of this paper is organized as follows. A detailed description of a good middleware-based component platform for robot software is given in section II. In section III, the OPRoS component model is introduced. Section IV explains the component execution engine, and section V shows its development tools for authoring and composing components. An analysis of the OPRoS component platform and its application for commercial robots follows in section VI. Section VII concludes this paper.

II. Requirements Analysis

In this section, we analyze the desirable and required properties of a component software platform for robots, which is intended as a basis for a new component platform.

First, a component software platform for robots should support diverse operating systems such as MS Windows, Linux, and real-time operating systems because robot systems usually run under various operating systems.

Second, the component software platform needs to support distributed communications. A robot system is frequently distributed in order to expand computational power or to interact with its external servers.

Third, it is necessary that a component software platform be architecture independent. It should not rely on any specific robot software architectures such as Sense-Plan-Action, Subsumption [21], Hybrid [22], and other architectures, so that it can be applied to as many robots as possible, with whatever architecture they take.

Fourth, a component software platform for robots needs to support various execution semantics. Robots sometimes execute their jobs periodically as well as non-periodically for a classical control loop, method invocation for higher level control, or event-based stimulus-response.

Fifth, it is desirable for the components to be as simple as possible, but at the same time they should be composable so a more complex component can be assembled with other components. It would be of great help if we could compose several components so that they cooperate to achieve a shared goal.

Sixth, it should be easy to use. Robot developers usually like using their familiar development methodologies rather than learning a new one. Therefore, it is necessary to provide a simple component model, its development tools, and reusable service building blocks for easy use. In addition, transparency in supporting communication middleware is required for easy use because robot software developers are usually unwilling to do middleware dependant programming.

Finally, a component software platform for robots should have fault detection, recovery, real-time, and QoS support. Robots are embedded systems, and they often run for a very long period (hours, months, or years). They often run in environments that require prompt responses, and sometimes they need to run autonomously without any human interference.

III. OPRoS Component Model

This section introduces the OPRoS component model satisfying the previously-mentioned requirements.

1. Network Distributed

OPRoS components are reusable and replaceable software modules that do not need recompilation. They are distributed on a network. They run loosely coupled and independently, often representing a robot's devices. A robotic service is composed of these distributed components in a similar fashion as a robot hardware system is assembled with devices. A communication infrastructure including connection management of the components is provided by the component execution engine that is a runtime environment of the OPRoS components. By the separation of network management from component logic, developers can focus on the logic that they intend to develop without additional concerns about network management.

The granularity of a distributed OPRoS component can be at any level. For example, it can be at device level, algorithm

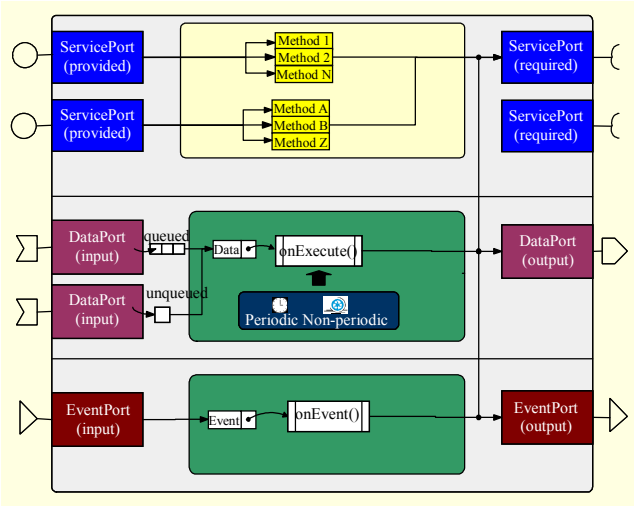


Fig. 1. OPRoS component model.

level, or coordination level, and so on, and it is up to component developers to decide which one is appropriate. With the distributed components of diverse granularities, a flat or hierarchical composition manner might be used for various robot software architectures.

2. Ports as Interface

In component-based robot software, components communicate with each other via connections. A connection is established from a port of a sending component to a port of a receiving component. We have observed that robot software developers usually use inter-component communication for sending or receiving three types of information: method invocation, data, and events.

To support these features, the OPRoS component model has three types of corresponding ports, that is, service, data, and event ports. A component has one or more ports of these types. Figure 1 depicts the OPRoS component model.

A service port allows other components to invoke its methods. It has an interface definition of a set of methods. A service port is either a provided or required type. A provided service port provides method services to other components. Methods of a provided service port are mapped to the user defined methods in the component. A required service port plays its role as a proxy to the user defined method of the connected component.

A data port is for exchanging data. It is either for input or output. An output data port sends data to input data ports of other components. Both the input and output should be of the same data type for a data exchange. A data port can have either a queue to store the received data or a single-sized buffer to store the most recently received datum. The received data are processed in the onExecute() method of a component in a periodic or non-periodic fashion.

An event port is for transmitting events. Although data ports and event ports are similar in that they transmit structured data, events are processed immediately by the network service thread with the onEvent() method, whereas the received data of a data port are buffered and then processed later by the component service thread.

Output ports for data/events do not block when transmitting data/events whereas service ports support blocking and non-blocking invocation according to the method types.

3. Execution Mode

The execution mode of a component is either periodic, non-periodic, or passive. In periodic mode, the onExecute() callback method of a component is called periodically to process data or execute its algorithm. It is useful for robot device components as they typically run periodically. Users can specify the execution period of a component in its component profile.

The onUpdate() method of the components of an equal period are invoked within the period right after all of their onExecute() methods are called. In contrast to the onExecute() method, which usually performs the component's primary logic and finishes as soon as possible, the onUpdate() method is intended for relatively expensive computational operations. This two-phase execution can minimizes latency and jitter, which is critical in real-time applications.

The non-periodic mode is used when the expected execution time of the onExecute() method is quite long or unpredictable. One thread is dedicated to each of the non-periodic components. A component in this mode iteratively continues its execution within the onExecute() method and doesn't release its dedicated thread until its destruction.

A component in passive mode has neither the onExecute() callback method nor its own thread. Instead, it is activated only when a stimulus such as an event or a method request arrives from other components.

4. Class Diagram

A user component should inherit from a base class called "component" of OPRoS as shown in Fig. 2. The component has one or more ports to interact with other components. It again inherits various interfaces such as lifecycle, port management, and property. These interfaces are used by the component container in order to manage the components. A user component inheriting the base class is realized by overriding the inherited callback functions and adding user defined methods. User defined methods are invoked by the network service thread upon receiving a request from client

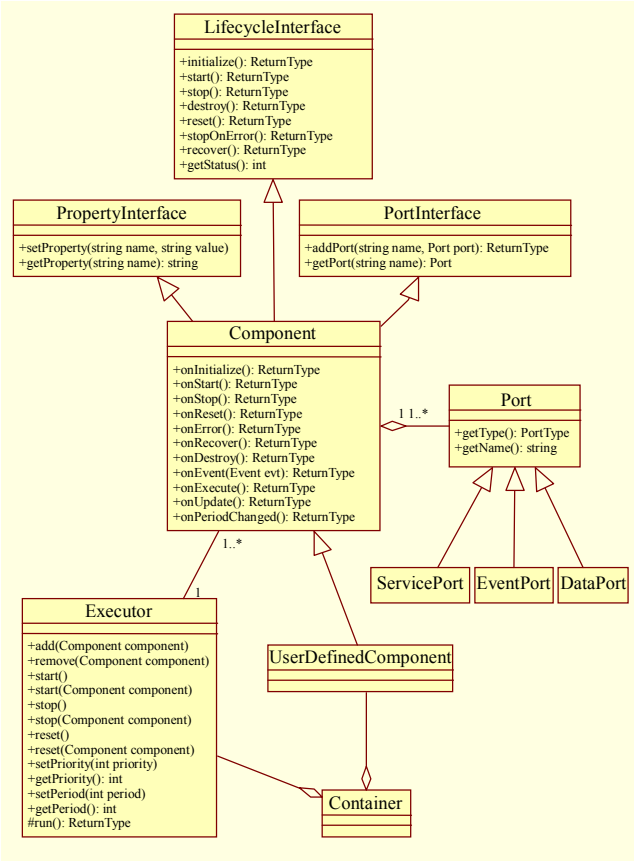


Fig. 2. UML class diagram of an OPRoS component.

whereas the methods of parent interfaces and callback methods of a component are invoked by the container.

The active execution of a component, either in periodic or non-periodic mode, is accomplished through the executor managed by the component container. The container registers components to the executor, and the executor runs the registered components, which have an equal period and the same priority, with an allocated thread from the container.

5. Lifecycle Management

A component runs through a sequence of states during its lifecycle as shown in Fig. 3. When a component's instance is created, it is in a Created state. Its state becomes a Ready state after onInitialize() is invoked by the container. The onStart() method leads the component into an Active state where it iterates the onExecute() and onUpdate() methods. When the onStop() method is called, it goes into an Inactive state and its execution is suspended until activated again by the onStart() method.

If an error occurs, the component transits into an Error state and its onError() callback is invoked by the container to deal with the error. When it recovers from the error, the component

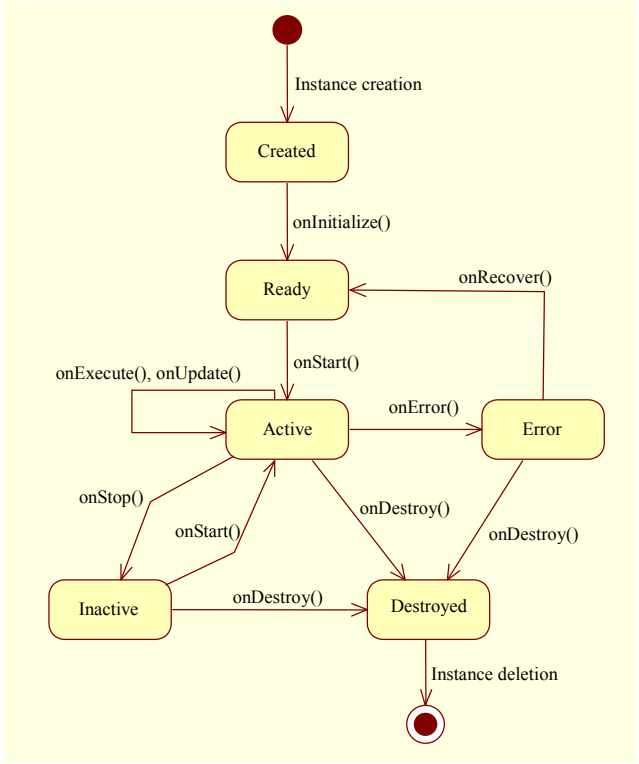


Fig. 3. State transition diagram of an OPRoS component.

goes into the Ready state right after the onRecover() method is invoked. A component instance is destroyed after its onDestroy() method is invoked.

6. Component Composition

Obviously, it is helpful if we can utilize existing components when making a new component in that this reduces the development time and errors that might occur when creating the component from scratch. This naturally leads to the types of components: atomic and composite.

An atomic component is made solely, and is mainly devised to abstract a low-level device or algorithm.

A composite component is composed of other components (either atomic or composite). A composite component accesses the ports of each contained component. When an interface of the composite component is called, it is delegated to a corresponding contained component. In this way, a composite component abstracts the interfaces of inner components so that users can access simplified interfaces.

7. XML Profiles

A component's port types, execution semantics, properties, and so on are described in an XML file called a component profile. The profile is interpreted by the component execution engine in order to operate the corresponding component.

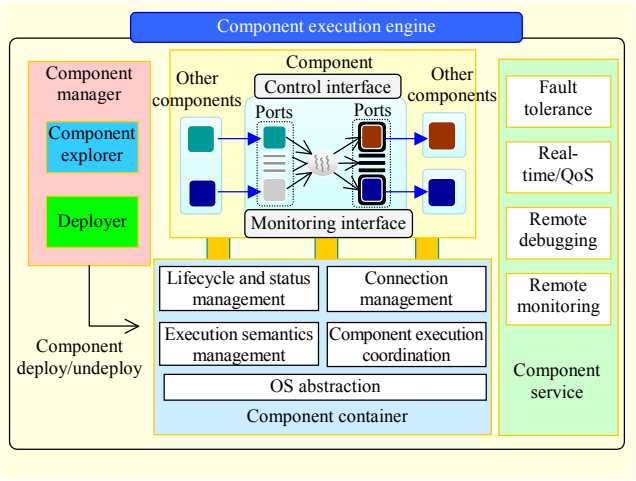


Fig. 4. OPRoS component execution engine.

The component's APIs are described in a separated service profile as a list of method signatures provided by a service port of a component. Also, the data profile describes data types or data structures used in method calls or data transfer between components. The two profiles are similar to CORBA IDL in the point that they describe interfaces and data types.

An application profile describing the network configuration of distributed nodes, references to participating components, and port connections between components is given to the engine for running a robot application.

IV. Component Execution Engine

The component execution engine manages and executes components in accordance with the application profile and each component's profile. It relieves robot developers from thread management, resource allocation, and state management so that they can concentrate on the application logic.

The engine has a component manager, component container, and component service as shown in Fig. 4. It explores and deploys components from a component composer, executes components harmonically, manages their lifecycle and states, connects components using the component container, and supports services such as monitoring and fault tolerance.

The component container interprets its application profile, loads participating components onto memory, establishes their connections, and activates the components in accordance with each component profile. When it comes to its turn, an activated component runs on an executor according to the component profile. The executor is allocated a thread by the scheduler module of the container. The scheduler allocates the same executor to the components of an equal period and priority to prevent threads from unnecessary context switching so that the performance is not weakened. The scheduler allocates one dedicated executor and a thread to each non-periodic component and executed it only once.

The component execution engine provides an appropriate abstraction of operating systems. The abstraction presents portable wrapper classes of the common functionalities such as thread functions, thread synchronization functions, and file I/O functions that are offered by any operating system. The wrapper classes encapsulate the system functions offered by the OS to which the code needs to be ported.

In addition to the OS abstraction classes, the engine provides connectors which are the abstraction class for the I/O communication to allow robot components to communicate across different various networks. The engine allocates a connector to establish the connection between two interacting ports. Connectors provide network connection management, marshalling functions, remote method calls, and data transfer functions. They can be bound to various network protocols or communication middleware. Currently, the engine provides three types of connectors: SocketConnector for TCP/IP, UPnPConnector for UPnP, and CorbaConnector for CORBA.

The execution engine should not fall into failure on faults or anomalies. The self-reconfigurable fault tolerance module detects faults or anomalies, and repairs them autonomously [23]. In particular, it focuses on the reliability of the threads encapsulated in the executor. Each thread processes user components periodically at the same cycle. As new components of the same cycle time are added to a executor, the executor might not be able to finish processing all the components within the cycle time, causing a violation of the timeliness of the components. To prevent this violation, each executor is monitored to detect violations. When a violation is detected at a component (referred to as a failed component), all the other components allocated to the same executor of the failed component migrate to a new executor and continue their execution. The failed component is still executed because it may finish its execution.

V. Development Tools

In general, making a component from scratch without any dedicated development tools is very time-consuming and error-prone. It seems necessary, therefore, to provide at least a tool for authoring atomic components and a tool for composing components. We provide two tools that run as plug-ins for the eclipse IDE [24], and therefore, can be installed and used on any OS platforms wherever eclipse is installed.

1. Component Authoring Tool

The user needs to specify the port interfaces, callback

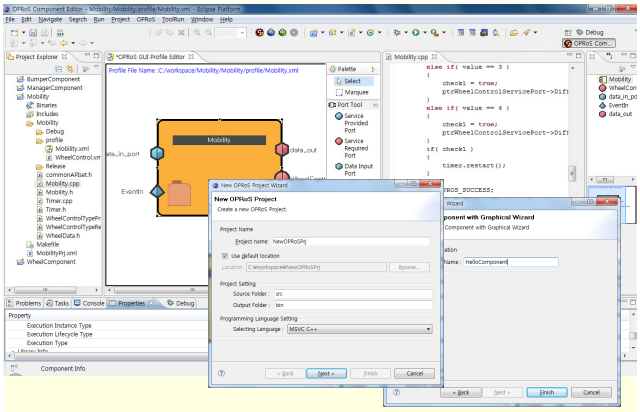


Fig. 5. Component authoring tool.

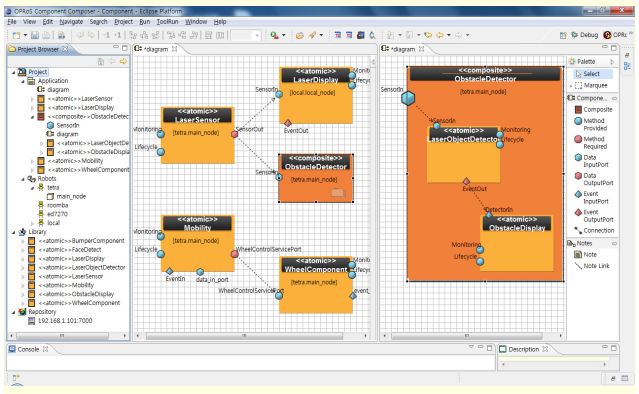


Fig. 6. Component composer.

functions, and a component profile when making an atomic component. The component authoring tool helps users to add implementations of callback functions and user-defined codes without any concern regarding various relationships between port interfaces and conformances defined in the component model, for example.

The component authoring tool runs as a plug-in into the eclipse C/C++ development tools (CDT). It supports the GCC and Microsoft Visual C++ compilers.

Figure 5 depicts diagram-based and graphic-based wizards of the tool. Via either a diagram-based or graphic-based wizard, the tool obtains required information about a component step by step from the component developer, and generates an XML component profile and C++ files for the component. The tool also produces a proxy code for the required ports and a skeleton code for the provided ports. These codes encapsulate the supporting communication middleware with connectors, so that developers can develop a component without a middleware dependent code. The proxy code transmits a request to the provided port connected to it. The skeleton code receives and delegates requests to its method. Many other codes for registering methods to the component, interface dispatching, and templates for user-defined methods are also automatically generated. Therefore, the user only needs to implement the callbacks and user-defined methods in the generated template source code.

Compiling of the code yields a component binary file as a shared library (either as the dynamic link library on MS Windows or shared object on Linux). Packaging this component binary and its XML component profile completes the authoring process. The package is ready for use by the component composer.

In addition, the tool supports debugging on an atomic component. It supports execution control (suspend, resume, and stop), stepping-in/over the code, and monitoring and evaluating variables.

2. Component Composer

The component composer is used for building robot applications by composing components. It has a local repository to store components and imports component packages from the component authoring tool. The application developer drags and drops components onto the main diagram and connects ports to build an application, as shown in Fig. 6.

It validates the data or service types of ports and lets them be connected only if they have the same type, which need to be shared between connected components.

A composite component can also be created, as shown in Fig. 6, by putting individual components into the composite component and connecting their ports to those of the composite component. The connection information is stored in the application profile generated automatically by the tool. In this way, external requests or data/events to the composite component can be delegated to the inner components, and vice versa.

The tool can remotely control and monitor multiple component execution engines simultaneously. The components on the main diagram are assigned to an execution engine by dragging the engine node onto them. Finally, the application profile and components are packaged and deployed to the component execution engine on a robot via a network.

VI. Analysis and Application

This section analyzes the performance of port communication and shows the results of a comparison with other robotic software platforms and application to commercial robots to show the feasibility of the OPRoS platform.

1. Port Communication Analysis

The robot developer needs to know the latency time of port-

to-port communication due to the fact that most robotic services have time constraints. A port communication type is either local or remote. In the case of local port communication, a sender component is connected to a receiver component, and both are in the same computing node. We optimized the communication by using memory copy instructions so that the communication overhead is minimized. In the case of remote port communication, the sender and receiver components need to use network communication, resulting in network connection overhead, data encapsulation overhead, and so on.

Evaluations were done with diverse variations on port types (data, event, and service), data sizes (an integer, a vector of million integers), and topologies. The topology configuration is represented as *x*:*y*, where *x* is the number of senders and *y* is the number of receivers. Depending on data types and sizes, a sender component sends an integer data or a vector of a million integers (via data or event port), or calls a function (via a service port) with the relevant argument. All tests were performed on a laptop computer with a 2.53 GHz Intel dual-core, 4 GB RAM, and Windows Vista OS. In the tests, *M* and *N* are 10. The remote tests were performed on a local loop back network connection in order to minimize other network interference. The results were averaged from evaluations.

Table 1 shows the results of the communication tests. The amount of time it takes to copy one vector of one million integers to another vector variable was evaluated as a benchmark, and its average value was 7.61 ms. Elapsed times for sending one integer locally for all topological configurations are at the microsecond level, meaning that local communication for small data is well optimized compared with remote communication. Elapsed times for sending a million integers locally in a "1:1" topology through ports are about twice that of the benchmark test.

Table 1. Elapsed times of port tests (ms).

Data	Topology	Port			Benchmark (a copy of one million integers)
		Data	Event	Service	
One integer	1:1 local	0.023	0.01	0.002	7.61
	1:1 remote	2.444	2.985	16.762	
	1:M local	0.034	0.036	NA	7.61
	1:M remote	3.345	1.336	NA	
	N:1 local	0.015	0.014	0.002	7.61
	N:1 remote	1.937	0.692	401.207	
	N:M local	0.053	0.032	NA	7.61
	N:M remote	2.799	5.332	NA	
One million integers	1:1 local	13.387	8.3	16.999	7.61
	1:1 remote	231.283	114.938	232.413	
	1:M local	47.194	89.849	NA	7.61
	1:M remote	752.07	651.35	NA	
	N:1 local	70.43	40.068	132.144	7.61
	N:1 remote	859.756	880.369	1163.663	
	N:M local	343.714	630.744	NA	7.61
	N:M remote	6955.456	6562.688	NA	

As expected, remote communications take more time than local tests because they need to encapsulate data and set up the socket. In particular, remote service port communication takes much more time mainly due to the fact that the service is inherently a remote procedure call over a network.

2. Comparison with Other Platforms

Today's service robotics market is often compared to the early PC market. Many companies have released robotic software platforms with the hope of building a standard robotic software platform. These platforms are competing and are incompatible with each other just like the early PC market [25]. They provide runtime environments, drag-n-drop graphical development tools, simulation environments, and operator control units.

Table 2 shows comparison results of the OPRoS component platform with other robot software platforms. Because the platforms target various robots ranging from toy-like robots to industrial or military robots, and the users vary from end-users to skilful robot engineers, it is quite difficult to say what the best approach is. Nevertheless, we argue that OPRoS has rich options for developers.

First of all, OPRoS is an open-source project and therefore free of charge for non-commercial use. It supports both Windows and Linux, which are regarded as the most widely used operating systems in service robots whereas some other platforms operate on a specific operating system. OPRoS will support more operating systems including real-time operating systems such as QNX for real-time processing.

Second, OPRoS is based on distributed component architecture having ports supporting various execution semantics including remote procedure calls and data/event flow control, which are partially supported in many other platforms.

Third, OPRoS is not geared toward any specific robot S/W architecture. Thus, the granularity of an OPRoS component can be at any level. By composing components of various levels, a higher level component is created. However, some platforms are dependent on a specific architecture and they cannot support functional expansion of components by composing them.

Table 2. Comparisons of robotic software platforms (updated from [25]).

	MSRDS	MARIE	MIRO	RT-Middleware	OROCOS	ROS	PEIS ecology	Player & stage	ERSP 3.1	Urbi	Mobile Robots	iRobot Aware	OPRoS
Open source	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Partial	No	No	Yes
Windows	Yes	No	No	Yes	No	No	Yes	Yes (simul. only)	Yes	Yes	Yes	Unknown	Yes
Linux	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Distributed services comm.	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes
Robot SW architecture independent	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	Yes	No	No	Yes
Data/event flow control	Yes	Yes	Event driven	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Remote procedure call control	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes
Composite component	No	No	No	Yes	No	No	No	No	Yes	No	No	No	Yes
Graphical drag-n-drop IDE	Yes	Yes	No	Yes	No	No	No	No	Yes	Yes	No	No	Yes
Simulation environment	Yes	Yes	No	No	No	Yes	Yes (play& stage)	Yes	No	Yes (Webots)	Yes	Yes	Yes
Middleware transparency	No	proprietary	No	No	No	proprietary	proprietary	proprietary	proprietary	proprietary	proprietary	proprietary	Yes
Fault-tolerance	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
Real-time	No	No	No	Yes (ARTLinux)	Yes	No	No	No	No	No	No	No	Planned

Fourth, OPRoS also provides development environments such as a GUI-based interactive component authoring tool, a drag-and-drop graphical component composer, a simulator, and about 80 reusable device and algorithm components including navigation, arm control, and face recognition. Robot software developers can exploit these development environments to develop components and robot services with ease compared with other platforms. In addition, middleware independent connector promotes easier developing.

Fifth, OPRoS supports a fault tolerance mechanism to prevent performance deterioration caused by the faults of a component and a callback mechanism to cope with the errors unlike some other platforms.

Finally, we have plans to enable OPRoS to support a real-time scheduling capability, which is supported by few platforms. Currently, OPRoS supports soft real-time only by using timer mechanism on MS-Windows with the time resolution of 5 ms or so depending on CPUs and the number of other processes. Furthermore, it will be designed to support a hard real-time scheduling capability by intercepting timer interrupts in the kernel layer of the operating system.

None of the robot software platforms fully satisfies all the requirements as described in Table 2. However, we argue that OPRoS has richer features for wide adoption and enhanced applicability to diverse fields than other software platforms.

3. Application to Commercial Robots

In order to verify the usefulness of the OPRoS platform, we applied it to commercial service robots. First, a robot application is deployed to several different types of robots to verify the reusability of OPRoS components. The second experiment is for showing OPRoS's support for distributed components by the cooperation of multiple robots.

A. Common Reusable Components

In this experiment, a robot application is composed of common OPRoS components, as shown in Fig. 7. The FaceDetector component processes images from the Camera component periodically. When it detects the face of a human, the robot approaches the person, detecting collisions with obstacles with a BumperSensor component. Both the FaceDetector and BumperSensor components are fused into the RobotMove component, which coordinates the robot via the WheelControl service port. The RobotProcessor component is a kind of proxy component connected to the hardware control board of the robot. It sends sensory data via its SensorOut data port and controls the robot's wheel when requested via the WheelControl service port from the RobotMove component. Every component has two additional provided service ports for monitoring and controlling by the component container. These ports are automatically attached by the authoring tool.

The application is deployed to several commercial robots: iRobot Create, iRobiQ of YujinRobot, and ED-7270 from ED Corporation. As iRobot Create has a mobile robot platform without any computational device, a laptop computer with a USB camera is attached to it for image processing and for executing the component execution engine. The other robots have an OPRoS component execution engine directly inside them.

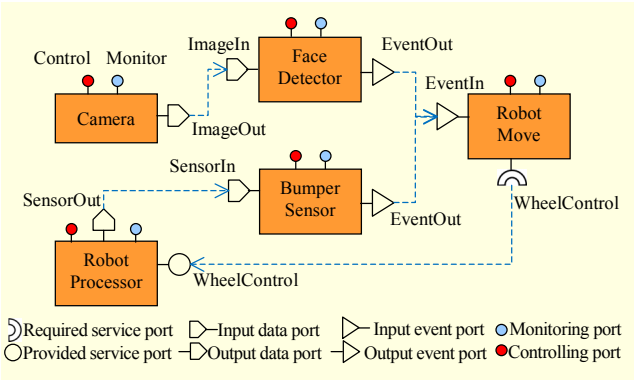


Fig. 7. Common application.

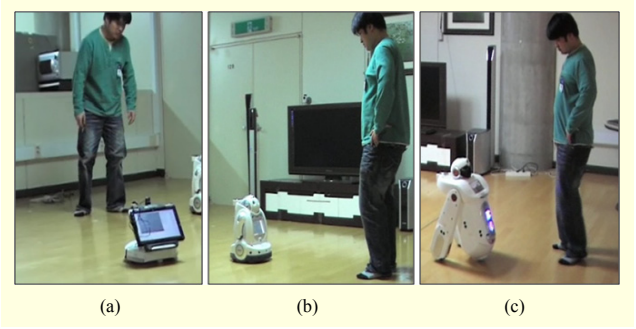


Fig. 8. Results of porting a common application to commercial robots: (a) iRobot Create, (b) Yujin Robot iRobiQ, and (c) ED 7270.

There is one essential problem that has to be solved in this experiment. As each robot has its own proprietary APIs for its hardware devices, these APIs cannot be directly used in reusability-enhanced components. To tackle this problem, proprietary APIs are wrapped with a common robot interface set (CRIS). Hence, the components accessing the robot's devices use CRIS to control hardware devices [26].

Figure 8 shows that the OPRoS components provide exactly the same services in different robots without any additional effort if they use the CRIS APIs.

B. Multi-Robot Cooperation

This experiment is to show that the OPRoS platform can be used in a distribution environment by applying it to multiple robots for a quiz game application. In the quiz game, Aldebaran's Nao robot plays as a coordinator that remotely

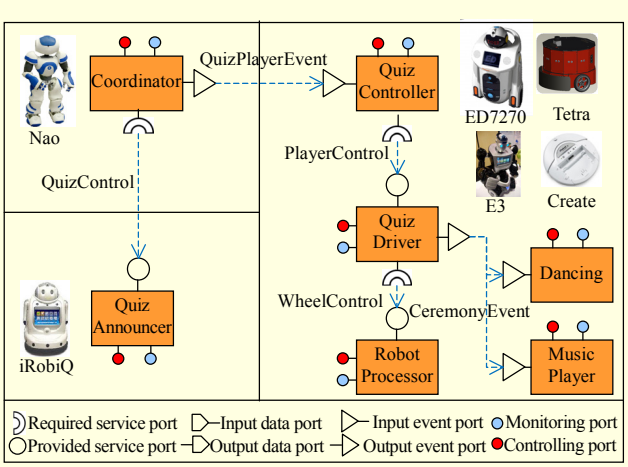


Fig. 9. Quiz applicaion.

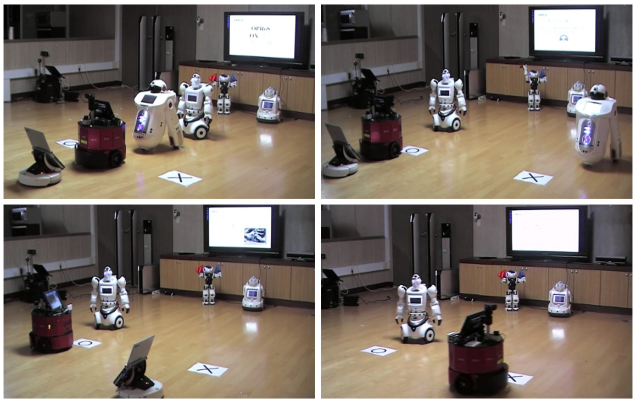


Fig. 10. Results of a quiz game using multiple robots: (in the upper left figure, from left to right) Create, Tetra, ED 7270, E3, Nao, and iRobiQ.

guides iRobiQ who is an announcer and poses questions. Other robots – iRobot's Create, Roboware's E3, ED 7270, and Dasa's Tetra compete to be the final winner.

As depicted in Fig. 9, the Coordinator component of Nao sends a request via a QuizControl service port to the remote QuizAnnouncer component of iRobiQ for announcing a question with text-to-speech (TTS) and a monitor connected to it. The QuizAnnouncer component returns the answer of the question to the Coordinator component. After announcing of a question, the Coordinator component sends an event to the QuizController component of the player robots via its QuizPlayerEvent port remotely. Then, each QuizDriver component of the player robots chooses an answer either 'O' or 'X' arbitrarily and moves to the chosen answer by using the RobotProcessor component described in the previous experiment. Next, the Coordinator component sends another event enveloping the correct answer of the question. Each player robot performs a ceremony using Dancing and MusicPlayer components if its answer is correct, otherwise it is out of the game and moves outside by using the RobotProcessor component.

Figure 10 shows snapshots of the robots cooperating for the game. This experiment verifies that OPRoS can be easily used in distributed environments.

VII. Conclusion

This paper introduced a robot component platform called OPRoS. The platform consists of specifications of a component model, a component authoring tool, a component composer, and a component execution engine. It supports various design patterns and execution semantics essential in robot software development, empowering robot developers to build a distributed component and to compose applications with the help of the authoring tool and component composer. It also provides various infra-services for components so that developers can concentrate their effort on the application logic itself, resulting in an efficient development of robot services.

A communication performance analysis and comparison with other robotic software platforms are expected to give developers information on choosing appropriate platforms for their robots.

Further research is necessary on the following topics. Currently, the OPRoS component platform does not support real-time scheduling. Periodic execution of components suffers from time variations mainly due to the non real-time scheduler of general purpose operating systems. However, it is necessary to support hard real-time scheduling capability in some cases. The component authoring tool needs to support remote debugging, and the component composer needs to support browsing and downloading components from a global component repository.

As an open source project, the OPRoS component platform is open to the public through a web site (http://www.opros.or.kr). Through this site, the platform is expected to be enhanced by many contributors.

References

[1] S.I. Lee et al., "Issues and Implementation of a URC Home Service Robot," *16th IEEE Int. Conf. Robot Human Interactive Commun.*, 2007, pp. 570-575.

[2] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering," *IEEE Robot. Autom. Mag.*, vol. 16, no. 4, 2009, pp. 84-96.

[3] I. Crnkovic, *Component-Based Approach for Embedded Systems*, New York: IEEE Press, 1994.

[4] EJB. Available: http://java.sun.com

[5] .NET. Available: http://www.microsoft.com/net/

[6] OMG, "Common Object Request Broker Architecture (CORBA/IIOP)," formal/2008-01-08, 2008.

[7] OMG, "Robotic Technology Component Specification," formal/08-04-04, 2008.

[8] J. Jackson, "Microsoft Robotics Studio: A Technical Introduction," *IEEE Robot. Autom. Mag.*, vol. 14, no. 4, 2007, pp. 82-87.

[9] C. Côté et al., "Robotic Software Integration Using MARIE," *Int. J. Advanced Robot. Syst.,* vol. 3, no. 1, 2006, pp. 55-60.

[10] H. Utz et al., "Miro-Middleware for Mobile Robot Application," *IEEE Trans. Robot. Autom.*, vol. 18, no. 4, 2002, pp. 493-497.

[11] N. Ando et al., "RTMiddleware: Distributed Component Middleware for RT (Robot Technology)," *IEEE/RSJ Int. Conf. Robots and Intelligent Systems*, 2005, pp. 3555-3560.

[12] H. Bruyninckx, "Open Robot Control Software: The OROCOS Project," *Proc. IEEE Int. Conf. Robot. Autom.*, 2001, pp. 21-26.

[13] ROS, Available: http://www.ros.org/wiki/

[14] A. Saffiotti and M. Broxvall, "PEIS Ecologies: Ambient Intelligence Meets Autonomous Robotics," *Int. Conf. Smart Objects and Ambient Intelligence*, 2005, pp. 275-280.

[15] B.P. Gerkey, R.T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," *Proc. Int. Conf. Advanced Robotics*, 2003, pp. 317-323.

[16] M.E. Munich, J. Ostrowski, and P. Pirjanian, "ERSP: A Software Platform and Architecture for the Service Robotics Industry," *IEEE/RSJ Int. Conf. Intelligent Robots Systems*, 2005, pp. 460-467.

[17] J.C. Baillie, "URBI: Towards a Universal Robotic Body Interface," *The 4th IEEE/RAS Int. Conf. Humanoid Robots*, vol. 1, 2004, pp. 33-51.

[18] K. Konolige "Saphira Robot Control Architecture," *SRI Int.*, 2002.

[19] Developers-Aware 2.0 Robot Intelligence Software. Available: http://www.irobot.com/gi/developers/Aware/

[20] B.Y. Song et al., "An Introduction to Robot Component Model for OPRoS," *Int. Conf. Simulation, Modeling Programming for Autonomous Robots Workshop*, 2008, pp. 592-603.
[21] R. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE J. Robot. Autom.*, vol. 2, no. 1, 1986, pp.14-23.
[22] J. Connell, "SSS: A Hybrid Architecture Applied to Robot Navigation," *IEEE Conf. Robotics Automation*, 1992, pp. 2719-2724.
[23] M.E. Shin and J.H. Ahn, "Self-Reconfiguration in Self-Healing Systems," *Third IEEE Int. Workshop Eng. Autonomic Autonomous Syst.*, 2006, pp. 89-98.
[24] Eclipse. Available: http://www.eclipse.org
[25] M. Somby, "Updated Review of Robotics Software Platform," Available: http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Updated-review-of-robotics-software-platforms
[26] C.S. Jang et al., "A Development of Software Component Framework for Robotic Services," *4th Int. Conf. Computer Sciences Convergence Inf. Technol.*, 2009, pp. 1-6.

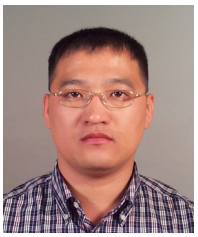
Choulsoo Jang received the BS in computer engineering from Inha University, Incheon, Korea, in 1995, and the MS in information and communication engineering from Gwangju Institute of Science and Technology (GIST), Gwangju, Korea, in 1997, and is currently working toward the PhD degree in computer engineering at Chungnam National University, Daejeon, Korea. Since 1997, he has been with ETRI. He has researched in the field of intelligent robot control systems. His recent interests include robot software platforms, embedded systems, and real-time systems.

Seung-Ik Lee received his MS and PhD in computer science from Yonsei University, Seoul, Korea, in 1997 and 2001, respectively. He is currently working for ETRI, Korea. He has published several papers on fuzzy logic control, evolutionary learning, and robot software and control. His research interests include evolutionary computation, fuzzy logic, and intelligent robot control and software.

Seung-Woog Jung has been a senior member of the engineering staff at ETRI, Korea, since 1998. He received his BS in computer science from Chonnam University, Korea, in 1996, and the MS degree in information and communications from Gwangju Institute of Science and Technology (GIST), Gwangju, Korea, in 1998. His research interests include robot S/W architecture, component-based robot application development, and robot middleware.

Byoungyoul Song received his BS and MS in electronics from Chonbuk National University, Jeonju, Korea, in 1995 and 1997, respectively. He is currently a senior member of the engineering staff of the Intelligent Robot Control Research Team at ETRI, Korea. His research interests are robotics and embedded systems and software engineering.

Rockwon Kim received his BS and MS in computer science from Chungbuk National University, Cheongju, Korea, in 1998 and 2000, respectively. He is currently pursuing the PhD in industrial and system engineering at KAIST, Daejeon, Korea. He is currently a senior member of the engineering staff of the Intelligent Robot Control Research Team at ETRI, Korea. His general research interests include robot task control, logic systems, and embedded systems and software engineering.



Sunghoon Kim received his BS and MS in electronics from Kwangwoon University, Seoul, Korea, in 1995 and 1997, respectively, and is also working toward the PhD at the College of Information and Communications, Hanyang University, Seoul, Korea. He is currently a team manager of the Intelligent Robot Control Research Team at ETRI, Korea. His research interests include robot intelligence, robot software engineering, especially for military robots.

Cheol-Hoon Lee received the BS in electronics engineering from Seoul National University, Seoul, Korea, in 1983, and the MS and PhD in computer engineering from KAIST, Daejeon, Korea, in 1988 and 1992, respectively. From 1983 to 1994, he worked for Samsung Electronics Company in Seoul, Korea, as a researcher. From 1994 to 1995 and 2004 to 2005, he was with the University of Michigan, Ann Arbor, as a research scientist at the Real-Time Computing Laboratory. Since 1995, he has been a professor in the Department of Computer Engineering, Chungnam National University, Daejeon, Korea. His research interests include parallel processing, operating systems, real-time systems, and fault tolerant computing.