Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

Embedded Component-based Framework for Robot Technology Middleware

Ryo Hasegawa[1,a)] Naofumi Yawata[2] Noriaki Ando[3] Nobuhiko Nishio[2] Takuya Azumi[1]

Received: November 18, 2016, Accepted: May 16, 2017

Abstract: This paper presents a component-based framework for robot technology middleware (RTM) to address real-time issues with RTM. To handle real-time applications, the proposed framework achieves collaboration between RTM and the TOPPERS embedded component system (TECS). TECS is employed to enhance real-time processing in the proposed framework. To implement the collaboration of RTM and TECS, we have adopted remote procedure call. In addition, extending a generator enables the generation of robot technology components from TECS components with source code generated by a model-based development tool such as MATLAB/Simulink. We have evaluated the processor cycle counts of the proposed framework in comparison with those of a conventional method. Moreover, we evaluated the execution time of serial communication and a motor application using the proposed framework. The evaluation results show that the proposed framework is functionally employed in a hard real-time system. Furthermore, we evaluated the amount of code generated by the proposed framework. The evaluation results reveal that the code generated by the proposed framework is reusable and can enhance productivity.

Keywords: robot middleware, component-based development, real-time system, model-based development

1. Introduction

Robotic technology is recently becoming widespread for handling complex situations and environments. Robotic technology ranging from rescue robots used in disasters to industrial robots in factors is being utilized on an ever larger scale. In addition, intelligent spaces are being developed to support human activity, and intelligent robots and network-embedded devices are being utilized in such intelligent spaces. However, robotic functions can rarely be reused because each conventional robot is developed with a different architecture. Component-based development has attracted significant attention due to its high reusability.

Robot technologies, such as robot technology middleware (RTM) [1] and robot operating system (ROS) [2], have been proposed to enhance productivity. RTM establishes basic technologies to integrate new functionality in robot systems using an RT-component (RTC) modularized software component. ROS is a collection of a software framework for robot software development that provides operating system-like functionality on a heterogeneous computer cluster.

To improve the software productivity, component-based development and model-based development (MBD) have been used. Component-based development such as TOPPERS embedded component system (TECS) [3], [4], AUTOSAR [5] is becoming predominant. Component-based development constructs a system with reusable components. By applying component-based development, developers can grasp the structure of software program easily. Software components are reusable. MBD, such as MATLAB/Simulink [6], is a concept of software development in which models are developed as work products at every stage in the development life cycle to save development cost and to improve development efficiency.

Robotic technologies enhance productivity; however, they cannot handle hard real-time and embedded systems, such as brushless DC motor control systems, because RTM uses common object request broker architecture (CORBA). CORBA manages packets in a FIFO manager. Therefore, RTM is not suitable for real-time systems. In addition, CORBA requires large amounts of resources and so is difficult to use in environments with limited resources such as embedded systems. Previous works [7], [8], [9], [10], [11], [12] have attempted to address these problems; however neither the real-time nor resource problems have been solved, and this work did not focus on software-reusability.

This paper proposes a component-based framework for RTM to support the development of intelligent robots that work with embedded systems. The proposed framework realizes a collaboration of RTM and TECS to handle real-time applications. There are four reasons why we employed TECS. First, TECS has generator functionality. Thus, by developing an RPC generator for RTM, a TECS generator can be extended as a plug-in. Software components required by the developers are easily generated by this TECS generator functionality. Second, TECS is employed by ASP3 kernel, which is the latest TOPPERS ITRON kernel. As such, TECS will be used in many scenes [13]. Third, as TECS is optimized for RTOS, it is relatively straightforward for TECS

1 Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560–8531, Japan
2 Graduate School of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
3 National Institute of Industrial Science and Technology (AIST), Chiyoda, Tokyo 100–8921, Japan
a) hasegawa@hopf.sys.es.osaka-u.ac.jp

to support real-time systems. TECS enables the use of several real-time operating systems (RTOSs) [4]. In TECS, periodic tasks are easily created [14]. Fourth, TECS already supports some device driver libraries. Further, developers can reuse these TECS libraries.

Furthermore, we employ a brushless DC motor as an example of real-time application to evaluate the proposed framework. For brushless DC motor control, we focus on MBD, vector engine (VE), and RTOSs. A portion of the brushless DC motor control is implemented by an MBD tool. We employed VE to support real-time processing. VE [15] is hardware that calculates the optimal pulse width modulation (PWM) duty in place of a processor. Using VE and RTOSs allows a low-power processor to process other tasks associated with motor control, such as communication with RTM in a sufficient amount of time. Software developers who have minimal or no knowledge of real-time processing will benefit from employing the proposed network. If there are existing applications such as those involving motor control, even developers who are unaware of the specific details of motor control can use this application from RTM.

To implement RTM and TECS collaboration framework, we employ a remote procedure call (RPC). An RPC is a client/server system that allows a subroutine to execute in another address space. There are two reasons why we employed RPC. First, RPC handles command data. Thus, using RPC, developers can reuse various control applications. Second, TECS supports basic parts of the RPC mechanism, such as marshaller and unmarshaller functions, which make it relatively straightforward to implement a collaboration of RTM and TECS. We extended OpaqueRPC (an RPC mechanism in TECS [16]) for the proposed framework. A stub for the RPC is generated automatically utilizing a plug-in in the TECS generator. Thus, when RTM and TECS work collaboratively, we can expect lower hardware costs because a low-order control system, such as real-time control, can realize a system that is not dependent on an advanced real-time Linux-based system (ART-Linux) [17].

We chose RTM because TECS matches RTM services using the RPC mechanism instead of ROS topics transport using a publish/subscribe model. RTM applications are mainly implemented on the basis of RTM services. Although ROS supports ROS services [*1], ROS applications are basically implemented on the basis of topics. Because TECS supports the RPC mechanism, it easily handles RTM services. Further, the proposed framework can be adapted for ROS services [*2].

Experiments using the proposed framework enabled us to estimate the WCET in environments using RTM, and we have proven that RTM can be used in hard real-time and embedded systems.

Contributions:

- The proposed framework enables the use of hard real-time systems using RTM applications. We adopt TECS because it is suitable for RTOSs and handles real-time applications. We conducted evaluations of real machines to confirm that real-time applications are available. The results show that the proposed framework achieves smaller variations compared with those of an original RTM application. Moreover, we evaluated brushless DC motor control using VE and the proposed framework as an example of a hard real-time application.
- The proposed framework enables the reuse of legacy code (TECS code) including device drivers (e.g., sensors and actuators). Moreover, the proposed framework enhances the range of RTM applications because TECS supports a variety of domains such as home electronics, sensor networks, automobiles, and aerospace applications.
- The proposed framework attempts to reduce RTC developer workload by generating RTC code and configuration files for robot technology, and a developer can obtain the RTC code by simply adding a single line of code.
- The proposed framework enables component-based development and model-based development. Wrapping with TECS, C code generated by model-based development can be used in the proposed framework.

The remainder of this paper is organized as follows. Section 2 explains basic technologies, i.e., RTM, TECS, and VE. Section 3 describes the design and implementation of the proposed framework. Section 4 shows experimental results obtained with the proposed framework. Section 5 discusses related work, and Section 6 concludes the paper with a summary.

2. System Model

Figure 1 shows a system model wherein a motor is controlled by voice. In Fig. 1, TECS manages motor control. Note that the motor control is a real-time system. An RTC is a robotic element of RTM. The Microphone Control and Voice Processing RTCs recognize a human voice, and then, the Control RTC executes the Motor Control component in TECS. The actual motor control is performed by the TECS Motor Control component including code generated by an MBD tool (MATLAB/Simulink) [19]. In addition, we employed VE because it can reduce the calculation load. In particular, VE supports the ability to satisfy 100 μs deadlines within the brushless DC motor control period. The control RTC calls up a TECS motor function using the proposed framework.

In the proposed framework, the target system is divided into non-real-time and real-time components, as shown in Fig. 1. This framework does not support real-time requirements for the non-real-time components. Further, we must handle real-time applications for the real-time component. As mentioned in the Section 1, TECS supports real-time applications, thus we assume that all real-time applications are executed on TECS using the proposed framework.

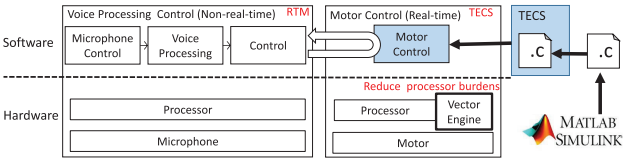


Fig. 1 A system model.

[*1] A ROS service is a synchronous communication mechanism of the client-server type of network. It is just like calling a function.

[*2] By using an RTM-ROS bridge [18], the proposed framework can be adapted for ROS services.

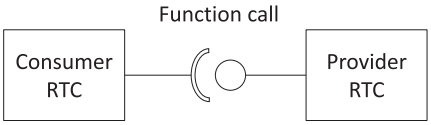


Fig. 2 Service port architecture of RTM.

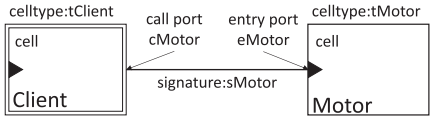


Fig. 3 Component diagram of TECS.

```
1 signature sMotor {
2   ER start([in]int32_t speed);
3   ER stop(void);
4   ER setSpeed([in]int32_t speed);
5 };
```

Fig. 4 Signature description of TECS.

```
1 celltype tClient {
2   call sMotor cMotor;
3 };
4 celltype tMotor {
5   entry sMotor eMotor;
6 };
```

Fig. 5 Celltype description of TECS.

For example, consider a mobile robot control system. A programmer creates non real-time components such as path planning in RTM and real-time components such as motor control in TECS. Motor control is applied via the motor control components while satisfying the 100 μs control cycle according to the path planning.

This section describes RTM [1], which is a distributed component middleware for robot technology, TECS [3], [14], which is a component system that functions as a runtime environment in the proposed framework, and VE, which is hardware that calculates the optimal PWM duty in place of a processor to reduce processor load.

2.1 RTM

RTM attempts to establish a common platform based on distributed object technology. RTM supports the configuration of a variety of networked robotic systems by integrating a variety of networks that enable RTCs. OpenRTM-aist [20] is an implementation based on the RTM framework.

2.2 RTC

An RTC is a component that can modularize a sensor, a device, or an algorithm. The RTC interface specification has been standardized as component specification by the Object Management Group [21], a software standardization organization. The RTC can be modularized at a variety of granularities and provides a distributed component framework that can be executed using various programming languages and OSs. It consists of the following elements.

Developers can define a service port to have any service interfaces, any interface, and any type of interface. Services are defined using the interface definition language (IDL). The service port has a service provider interface to provide services and a service consumer interface to use the services. **Figure 2** shows the service port architecture.

The configuration can have various sets that include a list of pairs of parameter names and values and can switch sets dynamically during operation. Note that component parameters in the configuration can be used as a reference and altered freely.

Two tools are provided for RTM development, i.e., the RTC Builder [22] and RT System Editor [23]. The RTC Builder is a template generator tool for RTCs that generates custom templates based on user-configured parameters. The RT System Editor is an editor that can connect, disconnect, or configure RTCs.

2.3 TECS

Here, we describe the features of TECS.

2.3.1 Component Model

A *cell* is an instance of a component in TECS. *Cell*s are properly connected to develop an appropriate application. A *cell* has *entry port* and *call port* interfaces. The *entry port* is an interface that provides services (functions) to other *cell*s. The service of the *entry port* is called the *entry function*. The *call port* is an interface that uses the services of other *cell*s. A *cell* communicates in this environment through these interfaces. The *entry port* and *call port* have *signatures* (i.e., sets of services) that define the interfaces in a *cell*. The *celltype* defines a *cell* such as the *Class* of an object-oriented language, and a *cell* is an entity of *celltype*.

Figure 3 shows an example of a component diagram. Each rectangle represents a *cell*. The dual rectangle represents an active *cell* that is the entry point of a program such as a task and interrupt handler. The left *cell* is a Client *cell*, and the right *cell* is a Motor *cell*. Here, tClient and tMotor are *celltype* names. The triangle in the Motor *cell* represents an *entry port*. The connection of the *entry port* in the *cell*s describes a *call port*.

2.3.2 Component Description of Task Cell

A component in TECS is described using the component description language (CDL). The CDL can be divided into three categories, i.e., *signature*, *celltype*, and *build* descriptions. The *signature* and *celltype* descriptions are described by component developers, and the *build* description is written by application developers.

The *signature* description is used to define a set of function heads. TECS provides the *in*, *out*, and *inout* keywords to determine whether a parameter is an input and/or an output. **Figure 4** shows an example *signature* description that defines the interfaces of a motor. A *signature* name, such as sMotor, follows a *signature* keyword to define the *signature*. Here the s in sMotor denotes *signature*. A set of function heads is enumerated in the body of this keyword.

The *celltype* description is used to define the *entry port*s, *call port*s, *attribute*s, and *variable*s of a *celltype*. **Figure 5** shows an example of a *celltype* description that defines components for an RTOS task.

A *celltype* name, such as tClient, follows a *celltype* keyword to define the *celltype*. Note that a *call* keyword is used to declare a *call port*. Two words follow the *call* keyword, i.e., the *signa-*

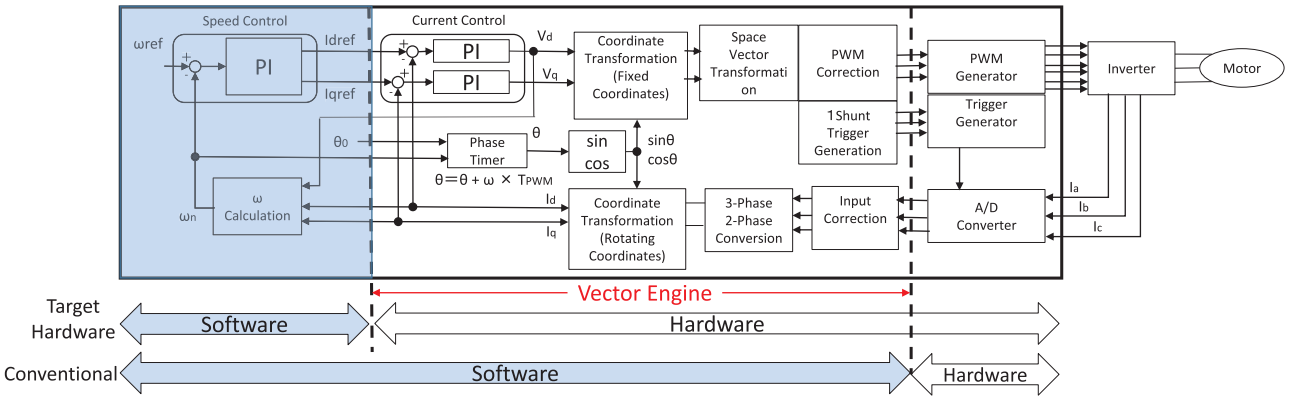


Fig. 7 Flow of a VE process.

```
1 // instantiation of Motor cell
2 cell tMotor Motor {
3 };
4 // instantiation of Client cell
5 cell tClient Client {
6   // connection of cell
7   cMotor = Motor.eMotor;
8 };
```

Fig. 6 Build description of TECS.

ture name, such as sMotor, and *call port* name, such as cMotor. Similarly, an *entry* keyword is used to declare an *entry port*.

The *build* description is used to declare *cells* and to connect between *cells* for creating an application. **Figure 6** shows an example of build description to connect the *cells*. To declare *cell*, a *cell* keyword is used. Two words follow the *cell* keyword: a *celltype* name, such as tMotor, and a *cell* name, such as Motor. In Fig. 6, eMotor (*entry port* name) of Motor (*cell* name) is connected to cMotor (*call port* name) of Client (*cell* name). The *signatures* of the *call port* and *entry port* must be the same to connect the *cell*s.

2.3.3 Development Flow

Three types of TECS developers exist, i.e., component, application developer, and plug-in developers.

The component developer defines *signature*s and *celltype*s. The *TECS generator* generates different interface codes (.h or .c) in the C language and several template code of the function from the *signature* and *celltype*. The component developer describes a program in the generated template code, called a *celltype* code. The application developer describes the build description to develop an application. The interface and header code, i.e., the glue code of the *cell*s, are generated from the build description. Then, the other *celltype* code that corresponds to a plug-in of the TECS generator is generated using interface code. The plug-in developer describes plug-in code, such as an RPC and log trace. Finally, the header, interface, and *celltype* code are compiled and linked. This process generates an application module.

2.4 VE

VE is hardware that calculates an optimal PWM duty in place of a processor. This calculation referred to as vector control. Using VE, the processor can execute a motor control task and other tasks simultaneously. In this paper, we adopt a brushless DC motor as a hard real-time application to evaluate the proposed framework. In brushless DC motor control, the calculation of optimal PWM duty is more complex than that of a brush motor, and a control period is typically less than 100 μs typically. Therefore, brushless DC motor control must meet hard real-time requirements. In addition, a low-power processor in an embedded system can only process a motor control task.

The VE process flow is shown in **Fig. 7**. VE obtains three driving currents of a brushless DC motor from an AD converter. First, VE transforms the three-phase signal into a two-phase signal to avoid the simultaneous handling of the three signals. Next, the two-phase signal is located on a fixed coordinate system and rotated with the rotor of the motor; however, if this signal was located on a rotating coordinate system, subsequent calculations would be easier than with the fixed coordinate system. Thus, VE performs a coordinate transformation from a fixed to a rotating coordinate system. Further, VE outputs the resulting two-phase current signal to the software on the processor which in turn calculates the ideal current values and rotor angle for controlling the motor speed. The software on the processor sends the ideal current values for the brushless DC motor to VE, which then calculates the PI control output as the voltage value. This voltage value is located on the rotating coordinate system. VE returns the voltage value to the fixed coordinate system and executes a space vector conversion from the two-phase voltage signal to a three-phase voltage signal to obtain the three-phase input signal for the motor driver. Using this process, VE performs vector control.

3. Design and Implementation

This paper proposes a component-based framework for RTM to handle real-time applications and use RTM in environment with limited resources. To implement the proposed framework for RTM and TECS, an RPC is utilized for an internal component of a robot that handles real-time applications. In the proposed framework, we assume that developers use existing real-time applications and a corresponding environment. In many cases, fixed-priority preemptive and fixed-priority non-preemptive scheduling algorithms are used for real-time scheduling [24].

3.1 TECS RPC

TECS supports two types of RPC [16], i.e., OpaqueRPC and TransparentRPC. OpaqueRPC, as shown in **Fig. 8** (1), is used for

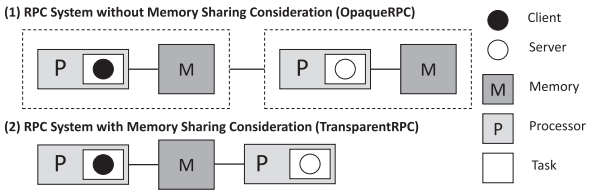


Fig. 8 Two types of TECS RPC.

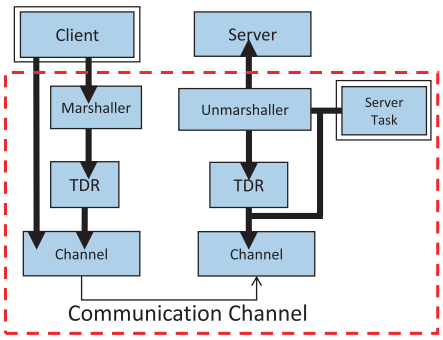


Fig. 9 Structure of OpaqueRPC.

systems without memory sharing functionality such as a network. TransparentRPC, as shown in Fig. 8 (2), is used for systems with memory sharing, such as multi-processor environments. We employ OpaqueRPC because we assumed that the systems used in this research would not require memory sharing. The structure and processing flow of OpaqueRPC are explained as follows.

3.1.1 OpaqueRPC

The components inside the dotted line in **Fig. 9** represent the OpaqueRPC structure. Each component is explained in the following.

The *Marshaller* component converts data to a suitable RPC message format. It can be serialized in the *Channel* component to communicate with data from the *Client* component. The *Marshaller* de-serializes data from the *Channel* component for analysis by the *Client* component.

The *Unmarshaller* component de-serializes data from the *Channel* component for analysis by the *Server* component. *Unmarshaller* calls the functions that correspond to the function IDs of the *Server* functions. The *Unmarshaller* component serializes the execution results and sends the results to the *Channel* component.

TECS Data Representation (*TDR*) layer performs RPC byte order by message, judgment, and type conversion.

The *Channel* component processes the data transmitted between the *Client* and *Server* components. The *Channel* component opens and closes the communication channel and can send and receive data. At present, two types of communication mechanisms are supported, i.e., TCP/IP and serial communication.

A *Server task* invokes the *Unmarshaller* component to wait for a client request.

3.1.2 OpaqueRPC Procedure

The OpaqueRPC procedure is described as follows. 1) A *Marshaller* sends a Start Of Packet (SOP) to begin transferring RPC messages. 2) The *Marshaller* sends the function ID executing in a *server*. 3) The *Marshaller* sends the arguments indicated *in* or *inout*. 4) The *Marshaller* sends End Of Packet (EOP) to stop

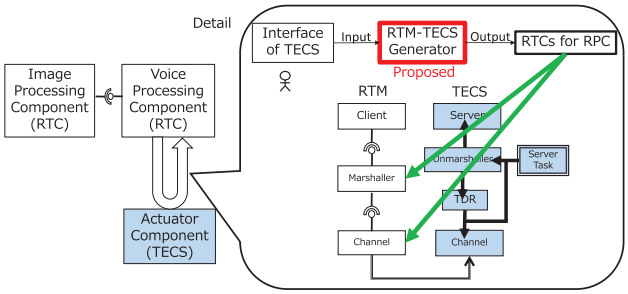


Fig. 10 RTM-TECS framework.

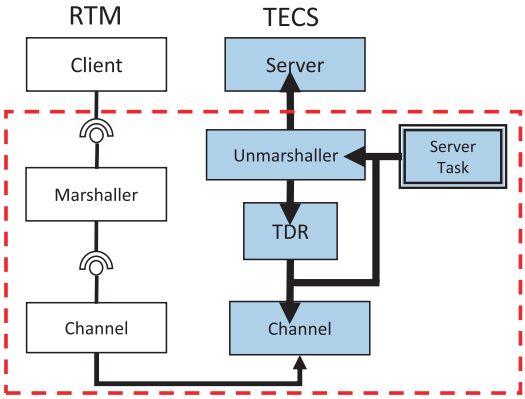


Fig. 11 OpaqueRPC (RTM:Client, TECS:Server).

sending RPC messages. 5) The *server* executes the function. 6) The *Unmarshaller* sends an SOP. 7) The *Unmarshaller* sends the arguments indicated *out* or *inout*. 8) The *Unmarshaller* sends the return value if one exists. 9) The *Unmarshaller* sends an EOP.

3.2 RTM-TECS (Proposed Framework)

An overview of the proposed framework is shown in **Fig. 10**. In the proposed framework, TECS handles real-time control, such as motor control, and RTM processes non real-time control, such as sound control and image processing. We employ TECS, because it utilizes several RTOSs such as OSEK [25] and ITRON [26], and it is appropriate for embedded systems. The proposed framework achieves communication between RTM and TECS for the RTC service ports. For the RTC service port, we propose a TECS RPC mechanism that adapts to RTM (Fig. 10, right). We employ an RPC because developers can use TECS functions.

3.2.1 Service Port

Using an RTC service port as a TECS component (Section 3.1.1) has been realized as an extension of OpaqueRPC. As shown in **Fig. 11**, the Opaque RPC client has been replaced by an RTC client. The RTC *Marshaller* component includes TDR functionalities. The service port is used to communicate among RTCs, e.g., between *Client* and *Marshaller* components on the client side. CMake is used to configure and compile the RTC code in an RTM development environment (Windows or Linux).

In Fig. 11, the RTCs generated by the proposed framework are inside the dotted rectangle. Note that there are two types of generated code and a file.

The *Marshaller* must send and receive RPC messages in a unique order for the RTM client side. The *Marshaller* behaves in the same manner as the OpaqueRPC *Marshaller* and TDR.

Channel is an RTC that communicates with TECS compo-

nents. Changing the *Channel* configuration can alter communication methods such as TCP/IP and serial communication, and values (e.g., port number).

RTM uses a CORBA data type which differs from a TECS data type; thus, the RTM data type must correspond to the TECS data type.

There is an RTM-TECS RPC plug-in for the TECS generator to generate RTM stub code for the RTM client. The RPC plug-in generates code or files, i.e., *Marshaller* or *Unmarshaller* code, *Channel* code, IDL files, and an XML file. The RPC plug-in matches the data size of TECS and RTM. The code is generated in folders, which allows CMake to build RTCs. The details are explained below.

An example of the ER start([in]int32_t speed) function is shown in **Fig. 12**. Note that the function takes *[in]* or *[out]* arguments. Since function name, function ID, arguments, and a return value depend on the function, RPC plug-in generates suitable *Marshaller* code. The *Marshaller* code follows the order of OpaqueRPC (Section 3.1.2).

The TECS *signature* and RTM IDL are different in function type and argument format. However, both represent interfaces between components. Therefore, mutual conversion is possible. **Figure 13** shows an example of converting the interface shown in Fig. 4.

RTC Builder reads an XML template file to generate RTC template code to communicate with a server or a client (OpaqueRPC). Therefore, developers can use RTC Builder without prior knowledge of OpaqueRPC structure.

3.2.2 Model-based Development for Component-based Framework

In the proposed framework, we employ an MBD tool (MATLAB/Simulink) to generate C code for embedded systems. Using MBD, we can enhance productivity. However, in MBD, it is difficult to integrate generated code with hardware dependent parts such as device drivers. In the proposed framework, TECS components handle device drivers, because TECS have some device driver libraries. Developers can reuse TECS libraries. On the other hand, MATLAB/Simulink codes handle control algorithm such as motor control algorithm. In addition, we wrap generated MATLAB/Simulink C code in TECS components [19] to adapt TECS RPC. The proposed framework achieves coexisting MBD and CBD.

```
1 CORBA::Long ServerSVC_impl::
        start(CORBA::Long speed)
2 {
3  //send a SOP
4  m_pOwnerRTC->sendSOP();
5  //send a function ID
6  m_pOwnerRTC->putInt16(START);
7  //send an argument
8  m_pOwnerRTC->putInt16(speed);
9  m_pOwnerRTC->sendEOP();
10 if(m_pOwnerRTC->receiveSOP()!=0)return -1;
11 //receive a return value
12 CORBA::Long ret = m_pOwnerRTC->getInt32();
13 if(m_pOwnerRTC->receiveEOP()!=0)return -1;
14 return ret;
15 }
```

Fig. 12 Example of Marshaller function.

```
1 interface Motor{
2    long start(in long speed);
3    long stop(void);
4    long setSpeed(in long speed);
5 }
```

Fig. 13 Example of converting the interface definition.

4. Evaluation

We performed evaluations to demonstrate the effectiveness of the proposed framework. We compared the service port cycle counts.

In addition, we compared the execution time of a hard real-time system (a brushless DC motor) for the proposed framework with legacy code, and the amount of generated code compared with code written by developers. In these evaluations, we performed measurements on an actual implemented system.

4.1 Service Port Evaluation

The processor cycle counts of the proposed framework (RTM-TECS) were compared with the cycle counts of the conventional method (RTM-RTM). Measurement range from the server receives the RPC message to send the RPC message. The evaluation environment is shown in **Table 1** and the evaluated configuration is shown in **Fig. 14**. Figures 14 (a) and (b) illustrate communication between RTM and TECS and between RTM and RTM, respectively.

In this evaluation, TECS executes on a low-frequency H8 microcontroller, and RTM runs on a high-frequency ARM processor. In this evaluation, the H8 micro-controller and the ARM processor are connected via Ethernet using TCP/IP. Owing to the different frequencies, we converted execution time to cycle counts. Therefore, we evaluated the proposed framework without considering differences between the H8 and the ARM processors. The average cycle count was based on 100 function calls.

The comparison of cycle counts in **Fig. 15** indicates that the proposed framework was able to control a hard real-time sys-

Table 1 Evaluation environment of service port.

-	OS	CPU	Memory
TECS	TOPPERS/ASP [27]	H8/3069	512 KB
	(an RTOS)	(25 MHz)	+16 K
RTM	Raspbian (Linux)	ARM1176JZF-S (700 MHz)	512 MB

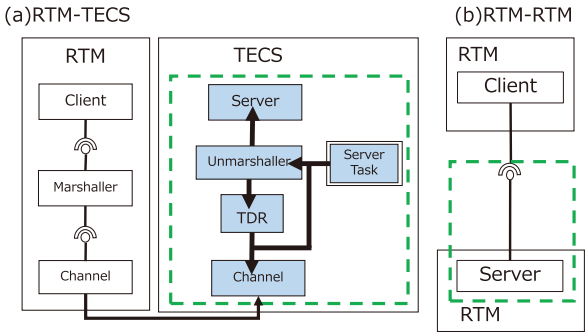


Fig. 14 Structure of evaluation for service port.

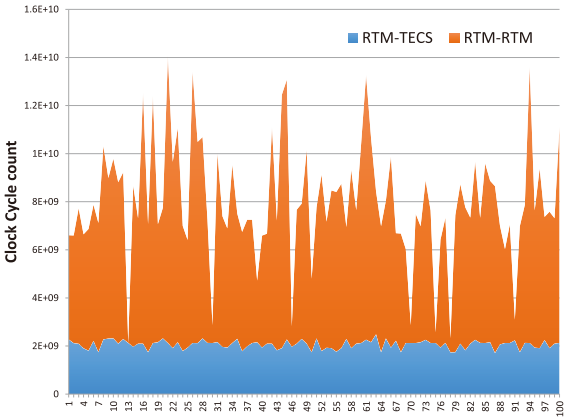


Fig. 15 Comparison of cycle counts.

Table 2 Evaluation environment of hard real-time system.

OS	CPU	Memory
TOPPERS/ASP [27]	ARM Cortex-M3	256 KB
(an RTOS)	(80 MHz)	+10 KB

tem using RTM (Section 1), which was one of the target contributions. In Fig. 15, the y-axis is cycle count, and the x-axis is the number of trials. The average cycle counts of the proposed framework were less than those of the conventional method. The average TECS and RTM cycle counts were 2,062,614,000 and 5,969,742,330, respectively. TECS has less overhead than RTM and is not a black box. As shown in Fig. 15, the cycle counts of TECS have less variation than those of RTM. In the proposed framework, the variance of the cycle counts is 3.10392×10^{16}. In contrast, the variance of the cycle counts of the existing method is 5.65211×10^{18}. In real-time processing, for WCET estimation, cycle count variation is more important than the average. It is possible to estimate the WCET of TECS applications, as discussed in Refs. [4], [14], [26]. Therefore, TECS supports meeting the difficult real-time constraints.

4.2 Evaluation of Hard Real-time System (Brushless DC motor)

We evaluated the proposed framework in a hard real-time system with a resource-limited device i.e., brushless DC motor control. The evaluation environment is shown in **Table 2**. The brushless DC motor used in this study is widely used in industry and other fields. In this evaluation, motor control, log, and RTM cooperation tasks run simultaneously. Moreover, the priority of real-time tasks is higher than that of an RTM cooperation task.

The target speed of the motor was 3,000 rpm and the state of the motor was stable. The measurement range was the period covering a motor control task. Further, the motor control command was called up by such as RTM, etc.

This evaluation results indicate that the proposed framework can reuse existing code to control embedded devices, which was one of the target contributions (Section 1). **Figure 16** shows the execution time of a brushless DC motor application using the proposed framework with an RTOS (TOPPERS/ASP [27]) compared with using legacy code. The legacy code is an infinite loop program that does not use an OS. In the brushless DC motor control

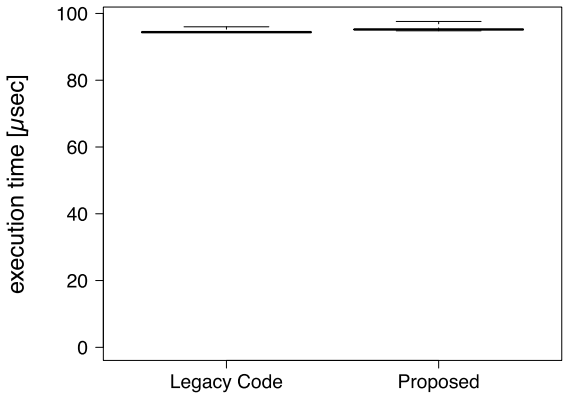


Fig. 16 Execution time in hard real-time systems.

Table 3 Execution time in a vector control process.

-	Software	Hardware (VE)	Differences
Vector Control Process	46.393 μs	17.904 μs	28.489 μs

Table 4 Comparison of the amount of lines in description.

-	Proposed	Conventional
Marshaller code	0	191+α
IDL file	0	32+β
CDL file	1	0
SUM	1	943+α+β

α and β depend on the number of functions and arguments

system, the 100 μs limit maintains the motor in a stable state. In the proposed framework, the variance of the cycle counts is as small as that of the existing method. The effect of the proposed framework on jitter is small as shown in Fig. 16. However, existing code also has an acceptable amount of jitter. Figure 16 shows that the WCET is within 100 μs, thus, the proposed framework facilitates WCET estimation and satisfies the 100 μs limit of the motor control system.

Table 3 shows the execution time of a vector control process using VE compared with the software method. As can be seen in Table 3, owing to VE, the processor has a margin of approximately 28 μs to perform its tasks, and the processor can process motor control and communication with RTM simultaneously.

In the proposed framework, we employed an RTOS to implement multi-tasking. Despite this RTOS, overhead was sufficiently low, thereby allowing control of the brushless DC motor. We confirmed that the hard real-time system (brushless DC motor) ran normally in the resource-limited device using the proposed framework. The evaluation results show that real-time performance is not reduced in the proposed framework.

4.3 Amount of Code

In this evaluation, the amount of code was compared. The evaluation results indicate that the proposed framework can reduce developer load, which was one of the target contributions (Section 1). The amount of handwritten code required by an RTC to realize the proposed framework is shown in **Table 4** ("Proposed" with the plug-in; "Conventional" without plug-in). If a developer uses the plug-in, the RTC code is generated from the TECS code

using the TECS generator.

It is evident that the proposed framework reduces the developer burden significantly. Creating an RTC to communicate with TECS requires more than 943 lines of code. In contrast, generating components using the TECS generator requires only a single line of code in the CDL file. Note that, in Table 4, comments and blank lines are not included, and α and β variables are proportional to the number of functions and arguments, respectively.

5. Related Work

This section discusses the work related to robot middleware for real-time processing. According to a report about the architecture for Real-time Control and Autonomous Distributed Execution (ARCADE) framework [28], the demand for real-time processing has increased. The ARCADE framework supports real-time execution from low to high-level and therefore enables real-time data transmission. However, developers cannot use embedded device legacy code, because the ARCADE framework does not provide a function call system.

Component-Integrated ACE ORB (CIAO) [7] is an implementation of the lightweight CORBA component model and Real-time CORBA. CIAO provides a component paradigm for DRE systems by abstracting DRE-critical systemic aspects, e.g., real-time QoS policies, as installable/configurable units supported by the component framework. CIAO implements the Component Implementation Definition Language compiler which extends the IDL compiler.

To reduce power consumption and develop distributed RTCs on microprocessors, Light-Weight RTC (LwRTC) [8] has been proposed to implement RTCs on embedded microprocessors. LwRTC supports CAN communication. RTC-Lite [9] realizes communication between the RTCs and the sensor nodes.

RTC-Lite provides a bridge component to communicate with small devices such as sensor nodes. However, RTC-Lite requires a proxy component for each device to communicate with the other RTC-Lites. Thus, the number of proxy components and RTC-Lites have been increasing in proportion to those of small devices. Consequently, it is difficult to use many sensor nodes with RTC-Lites.

An extended RTC framework [10] that considers timing constraints has been proposed. An extended RTC interface provides priority management and manages multiple periodic tasks and modified GIOP packets to notify the attributes of tasks to other RTCs. Since the framework is based on ART-Linux [17] which is an advanced real-time Linux for a real-time kernel developed for robotics, it is difficult to use the extended RTC framework [10] for embedded systems with limited resources.

Moreover, RTM runs on VxWorks [11] which is an RTOS. Since CORBA does not work with embedded systems, lightweight CORBA and libraries running on VxWorks have been proposed. As a result, RTM can run on embedded devices with VxWorks; however, real-time requirements were not considered [11]. In addition, this system does not work in a distributed environment, because it uses global variables.

The hybrid real-time ROS architecture on a multi-core processor (RT-ROS) consists of a non-real-time general operating system (GPOS) and an RTOS [12]. In the RT-ROS, each OS has its own processor, memory, interrupts, and peripheral devices. The RT-ROS provides real-time and non-real-time nodes that run on the RTOS and GPOS, respectively. The real-time performance of the RT-ROS has been tested [29]. The RT-ROS demonstrates efficient real-time performance, however it is difficult to use legacy code without modification in the RT-ROS, because the gap between the code and ROS nodes must be filled.

Table 5 RTM-TECS vs prior work.

-	RTM -TECS	CIAO	RTC -Lite	Extended RTC	RTM on VxWorks	RT-ROS
reusability	x					
real-time application	x	x		x		x
embedded system	x	x	x		x	
distributed system	x		x			x
MBD	x					

Table 5 shows a comparison of RTM-TECS and the above-mentioned previous work. Note that RTM-TECS demonstrates all of the features shown in Table 5.

6. Conclusion

This paper has proposed a real-time framework for RTM. The proposed framework has solved the problems of RTM, such as not being usable for real-time applications and not being suitable for embedded systems. Moreover, the proposed framework has enhanced the real-time processing of RTM using TECS. RPC mechanisms have been employed to use TECS components from RTCs. A new plug-in for a TECS generator has been developed to generate RTC code. Moreover, the proposed framework has supported code generated by the MBD tool. The evaluation results of the service port demonstrate that we have achieved an available hard real-time and embedded system using RTM. We have confirmed that the average and variance of cycle counts of an evaluation application were less than those of existing methods, thereby demonstrating that the proposed framework is suitable for real-time systems. Our examination of code generation indicates that reusing existing code can reduce the developer burden, e.g., developers can reuse existing device driver code by simply adding a single line to TECS code. The proposed framework currently provides its service to a single-client system, such as a motor control system. In the future work, the framework should support multi-client systems.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number 15H05305.

References

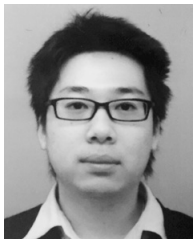
[1] Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T. and Yoon, W.-K.: RT-Middleware: Distributed Component Middleware for RT (Robot Technology), *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp.3933–3938 (2005).

[2] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y.: ROS: An open-source Robot Operating System, *Proc. ICRA Workshop on Open Source Software*, pp.1–6 (2009).

[3] Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H. and Takada, H.: A New Specification of Software Components for Embedded Systems, *Proc. 10th IEEE International Symposium on*

Object/Component/Service-Oriented Real-Time Distributed Computing, pp.46–50 (2007).
[4] Ohno, A., Azumi, T. and Nishio, N.: TECS Components Providing Functionalities of OSEK Specification for ITRON OS, *Journal of Information Processing*, Vol.22, No.4, pp.584–594 (2014).
[5] AUTOSAR: AUTOSAR, available from ⟨http://www.autosar.org/⟩.
[6] MathWorks: Simulink, available from ⟨http://www.mathworks.com/products/simulink/⟩.
[7] Balasubramanian, K., Wang, N. and Schmidt, D.C.: Towards Composable Distributed Real-time and Embedded Software., *Proc. 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, pp.226–233 (2003).
[8] Tsuchiya, Y., Mizukawa, M., Suehiro, T., Ando, N., Nakamoto, H. and Ikezoe, A.: Development of Light-Weight RT-Component (LwRTC) on Embedded Processor-Application to Crawler Control Subsystem in the Physical Agent System-, *Proc. SICE-ICASE International Joint Conference*, pp.2618–2622 (2006).
[9] Ohara, K., Suzuki, T., Ando, N., Kim, B.K., Ohba, K. and Tanie, K.: Distributed Control of Robot Functions using RT Middleware, *Proc. SICE-ICASE International Joint Conference*, pp.2629–2632 (2006).
[10] Chishiro, H., Fujita, Y., Takeda, A., Kojima, Y., Funaoka, K., Kato, S. and Yamasaki, N.: Extended RT-Component Framework for RT-Middleware, *Proc. 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp.161–168 (2009).
[11] Ikezoe, A., Nakamoto, H. and Nagase, M.: OpenRT Platform / RT-Middleware for VxWorks, *ROBOMECH2010*, pp.2A1–F19 (2010).
[12] Wei, H., Shao, Z., Huang, Z., Chen, R., Guan, Y., Tan, J. and Shao, Z.: RT-ROS: A real-time ROS architecture on multi-core processors, *Future Generation Computer Systems*, Vol.56, pp.171–178 (2016).
[13] Kawada, T., Azumi, T., Oyama, H. and Takada, H.: Componentizing an Operating System Feature Using a TECS Plugin, *Proc. 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, Nagoya, Japan (2016).
[14] Azumi, T., Ukai, T., Oyama, H. and Takada, H.: Wheeled Inverted Pendulum with Embedded Component System: A Case Study, *Proc. 13th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pp.151–155 (2010).
[15] TOSHIBA: VE, available from ⟨http://toshiba.semicon-storage.com/ap-en/product/microcomputer/lineup/arm-micon/tx03-series/function/vector-engine.html⟩.
[16] Azumi, T., Oyama, H. and Takada, H.: Memory Allocator for Efficient Task Communications by Using RPC Channels in an Embedded Component System, *Proc. 12th IASTED International Conference on Software Engineering and Applications*, pp.204–209 (2008).
[17] AIST: ART-Linux, available from ⟨http://www.dh.aist.go.jp/en/research/assist/ART-Linux/⟩.
[18] Open Source Robotics Foundation: RTM-ROS bridge, available from ⟨http://wiki.ros.org/rtmros_common⟩.
[19] Ohno, A., Hikawa, T., Nishio, N. and Azumi, T.: Integration Framework for Legacy and Generated Code in MBD, *Proc. WiP 26th Euromicro Conference on Real-Time Systems*, pp.9–12 (2014).
[20] AIST: OpenRTM-aist, available from ⟨http://openrtm.org/⟩.
[21] OMG: Object Management Group, available from ⟨http://www.omg.org/⟩.
[22] AIST: RTC Builder, available from ⟨http://openrtm.org/openrtm/en/node/1432⟩.
[23] AIST: RT System Editor, available from ⟨http://openrtm.org/openrtm/en/node/1322⟩.
[24] Ohno, A., Azumi, T. and Nishio, N.: TECS Components Providing Functionalities of OSEK Specification for ITRON OS, *Proc. 9th IEEE International Conference on Embedded Software and Systems*, pp.1434–1441 (2012).
[25] OSEK/VDX: Operating System, Version 2.3.3 (2005).
[26] Ishikawa, T., Azumi, T., Oyama, H. and Takada, H.: HR-TECS: Component Technology for Embedded Systems with Memory Protection, *Proc. 16th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, Paderborn, Germany, pp.92–99 (2013).
[27] TOPPERS: TOPPERS/ASP kernel, available from ⟨http://www.toppers.jp/en/asp-kernel.html⟩.
[28] Althoff, D., Kourakos, O., Lawitzky, M., Mörtl, A., Rambow, M., Rohrmüller, F., Brščić, D., Wollherr, D., Hirche, S. and Buss, M.: An architecture for real-time control in multi-robot systems, *3rd International Workshop on Human-Centered Robotic Systems*, pp.43–52, Springer Verlag (2009).
[29] Huang, M., Guo, S., Liang, X. and Song, X.: Research on Real-time Performance Testing Methods for Robot Operating System, *Journal of Software*, Vol.9, No.10, pp.2685–2692 (2014).

Ryo Hasegawa received his B.E. degree from Osaka University in 2015. His research interests include real-time and embedded systems.



Naofumi Yawata was born in 1989. He received his M.S. degree from the Graduate School of Information Science and Engineering, Ritsumeikan University in 2014. His research interest is embedded systems and robot systems.

Noriaki Ando received his Ph.D. degree in robotics from the University of Tokyo in 2002. Since 2003, he has been a Research Scientist of the National Institute of Advance Industrial Science and Technology (AIST) His research interests include robot software architecture. He is a member of SICE, RSJ, JSME and IEEE.

Nobuhiko Nishio received his B.E. and M.E. degrees in Department of Mathematical Engineering and Information Physics from the University of Tokyo, Tokyo, Japan, in 1986 and 1988, respectively, and the Ph.D. degree in from the Keio University in 2000. From 1993 till 2003, he worked at Keio University SFC. From 2007 till 2008, he worked as a Visiting Scientist at the Google Inc. Currently he is a professor of the College of Information Science and Engineering, Ritsumeikan University, Shiga, Japan. His current research interests are ubiquitous computing, location based systems, long term human activity recognition, real-time and embedded computing.

Takuya Azumi is an Assistant Professor at the Graduate School of Engineering Science, Osaka University. He received his Ph.D. degree from the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists for Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor at the College of Information Science and Engineering, Ritsumeikan University. His research interests include real-time operating systems and component based development. He is a member of IEEE, IEICE, ACM, and JSSST.