Development of Middleware for Rapid Prototyping of Robots Based on Parallel Behavior Architecture

MATSUNAGA Daisuke, YAMAMOTO Tomoya, FUJII Haruka and KOJIMA Takeshi

To realize robots that can move flexibly in human environments and automate nonstandard tasks, it is essential to parallel process functions like recognition, planning, and control to enhance responsiveness. The parallel behavior architecture addresses this requirement by executing high-level information processing in upper layers and motion capabilities in lower layers simultaneously. This structure ensures that upper processing can influence lower processing as needed, maintaining the responsiveness of the robot application. The authors developed middleware for robotic technology to enable rapid prototyping of robots with such architectures. This middleware provides functionalities that require expertise in implementing real-time processing and parallel processing, thereby streamlining the development process. Additionally, using this middleware, multiple robots with a parallel behavior architecture were developed in a short period of time. This paper presents the specifications and advantages of the middleware, comparing it with existing solutions. It also presents performance verification results and considerations regarding development efficiency using case studies of the middleware and its development environment.

1. Introduction

Against the background of the recent demographic problems, such as youth population shrinkage and labor shortages, needs are mounting for robot-driven productivity improvements. In response, OMRON has been committed to automating routine work, such as production line work, mainly in the FA field. To support a broader range of fields/work, OMRON aims to deliver various robots that flexibly move in the same space as humans and automate nonroutine tasks[1]. Developing such robots involves frequent changes in conditions or tasks. Such changes are challenging to foresee through the requirement definition process at the beginning of development. Therefore, a practical development approach should repeat improvements using rapid prototyping in response to user feedback.

Robots must reliably perform tasks while flexibly responding to unknown environments or dynamic environmental changes to work in the same space as humans. Achieving this goal requires coordinating the robot's internal processes at high cycles, such as recognition, planning, and control.

Robots' internal processes fall into high-order processes that handle high-order information, such as recognition/planning, and low-order processes that realize robot motion functions, such as motor control. Inputs to the low-order processes are the outcomes of high-order processes. Hence, the former processes depend on the latter. In addition, dependency exists among high-order processes, such as the planning and recognition processes. In conventional sequential execution methods directly modeled on these relationships, high-order processes cannot rate-control low-order processes to realize periodicity thereto, resulting in the problem of reduced responsiveness of robot applications. Brooks' Subsumption Architecture[2] and Yamamoto-Sugihara's Stacked Modulation Architecture[3] presented a direction for tackling this problem (Fig. 1). These parallel behavior architectures independently and parallelly execute high- and low-order processes so that high-order processes act on low-order processes as necessary. Therefore, these architectures can ensure the system's responsiveness without preventing low-order processes from executing robot motions. Achieving these architectures requires software that supports flexible multiple-controller implementation and multithread scheduling.

Contact : *MATSUNAGA Daisuke daisuke.matsunaga@omron.com*

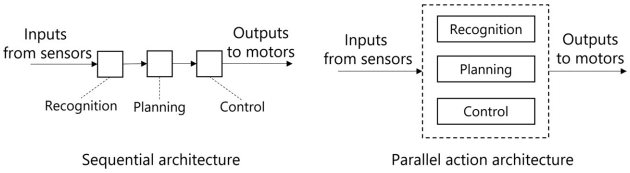

Fig. 1 Execution methods per the conventional and parallel behavior architectures

On the other hand, a look at the aspect of robot software development reveals that implementing functions requiring expert knowledge can often pose bottlenecks in deployment pipelines. For instance, implementing motor control and other cyclic processes requires considering real-time performance. A parallel behavior architecture requires thread-safe implementations involving no data conflicts and/or deadlocks, analyses of parallel processing-induced malfunctions, and implementation modifications. These tasks also require expert knowledge. Moreover, rework may occur because of the changing the execution environment, including the hardware or operation system (OS), while improvements are repeated for the robot system.

Software componentization and openization have been underway, aiming to improve robot development efficiency. However, no middleware has been made available with a primary focus on parallel process scheduling, whereby robots with a parallel behavior architecture are challenging to develop efficiently. Generally, enhancing software development efficiency requires improving software quality characteristics, such as maintainability and portability.

Based on the above circumstances, we developed robotics technology development middleware that reconciles achieving the aforementioned architecture and enhancing robot development efficiency. Our middleware makes it easier to solve the challenges in developing software for robots with a parallel behavior architecture, such as achieving real-time performance, preventing data conflicts, and changing the execution environment.

Achieving real-time performance requires various settings, including execution cycles, priorities, and CPU core assignments. This task demands expertise specific to OS/execution environment-dependent embedded developments. Moreover, these settings should be easily adjustable for rapid prototyping-based development. Hence, our middleware provides a multithread scheduling function to perform these settings easily. This function turns the concept of real-time processing into a model used to make various settings via the middleware's interface. Consequently, it has become possible to implement or reconfigure real-time processing without OS- or execution environment-related knowledge, enabling flexible adaptability to specification changes during development.

Data conflict prevention has been achieved by our middleware's inter-thread instruction transmission/reception function and pub/sub model-based data-sharing function. Integrating countermeasures against data conflicts during inter-thread access into these functions has enabled developers to achieve inter-thread access without considering data conflicts.

To solve the challenge of execution environment switching, we enabled cross-development in OS environments independent of development targets. This approach allows implementations in environments different than the target environment or bug analysis in debugging-friendly hardware/OS environments. We obtained this achievement by providing our middleware with an OS abstraction layer.

This paper compares our middleware with conventional methods and explains its advantages and usefulness. It also discusses the performance verification results and development efficiency based on a case of applying our middleware and its development environment.

2. Existing robot middleware

This section presents the ROS and RT-middleware, both representative of widely used middleware of the same category, and discusses the problems in each middleware.

2.1 ROS

The Robot Operating System (ROS)[4] is open-source middleware developed by Willow Garage for robot system development and the eco-system for development environment tool sets and deliverables that use the middleware. Now, it is maintained and administered by Open Robotics, a nonprofit organization. It serves as a basis for implementing various functions as units called nodes, which communicate with each other via P2P, thereby achieving robot systems. ROS is currently most broadly used in research and development venues for the following reasons: it enables building such loosely coupled systems, it abounds in development environment tool sets, its eco-system serves as an arrangement for publishing and sharing modules, it uses a BSD license, which does not require source code disclosure as long as the license conditions are met.

Developers are currently working on a next-generation version called ROS2[5]. This version is being newly developed based on the results of a requirement review as the requirement definitions established in the early days of ROS development are growing obsolete. The main differences from the previous version include supporting multiple robot control, embedded systems, and real-time control.

2.2 RT-middleware

RT-middleware (RTM)[6] is a software platform for building systems that use robots or robotics technologies (RT) developed mainly by the National Institute of Advanced Industrial Science and Technology (AIST).

RTM allows the construction of robot systems by combining software modules that have been componentized. These software modules are called RT components (RTC). The interface specifications are standardized by the Object Management Group (OMG) as official standards[7].

Moreover, RTM has been adopted as a development infrastructure for the Next-Generation Robot Intelligentization Technology Development Project under the Incorporated Administrative Agency New Energy and Industrial Technology Development Organization (NEDO).

OpenRTM-aist[8] is a software platform developed and distributed by AIST that includes the implementation of RT-middleware. It consists mainly of RTC execution management middleware, an RTC implementation framework, and a software development environment. With RTC frameworks and management functions made available, users can concentrate on core logic development.

2.3 Problems in existing middleware

ROS features a high degree of independence in processing units called nodes. Hence, it follows that ROS suits parallel process execution for achieving a parallel behavior architecture. However, its node processing and inter-node communication do not support real-time processing[9]. Hence, ROS does not suit our project herein, which intends to control highly responsive robots. ROS2 claims to support real-time control. However, real-time performance improvement has extended only to communication-related aspects but not beyond[10]. Accordingly, ROS cannot guarantee real-time performance and suffice for our objective of achieving a highly responsive robot system. The probable cause is that ROS is a framework intended to develop robot systems that include robots as components rather than control robots per se.

The OpenRTM-aist, an implementation of RT-middleware, uses a unit of processing called the RTC. The RTCs have a high degree of independence, similarly to the nodes in ROS. The OpenRTM-aist supports the real-time processing of node handling/communication. Hence, it follows that the OpenRTM-aist is designed suitably for our intended purpose of achieving highly responsive robots. However, achieving a parallel behavior architecture or similar configurations requires the following tasks for all the RTCs to be executed in real time with a patch applied to the kernel to make the standard OS (Ubuntu) real-time:

- Implement conversions to/from real-time and non-real-time processing before and after cycle execution portions,
- Set the execution cycle and the assignment to the context for each priority and
- Set the data transmission/reception method to suit its own internal process cycle or the sender/receiver-side cycle to prevent delays in cycle execution.

Such implemented systems differ from those that RTM inherently assumes, which are configured as networks of independent modules. Therefore, these configuration tasks are not supported by the standard framework and development environment and must be addressed and shouldered by the middleware users themselves[11].

3. Our proposed method

3.1 Requirements for our middleware

This subsection presents the primary requirements for solving the problems described in the previous chapter. It also summarizes the requirements for the middleware as a whole, which are necessary to enable rapid prototyping for each quality characteristic.

3.1.1 Primary requirements

1) Multithread scheduling

The middleware shall provide a multithread scheduling function that enables explicit high-priority assignments and system resource allocations to threads requiring high real-time performance, thereby reducing the influences thereon from other threads.

2) Data conflict prevention

The middleware shall provide an inter-thread instruction transmission/reception function and a pub/sub model-based data-sharing function. Countermeasures against data conflicts during inter-thread access shall be consolidated into these functions to provide thread-safe and non-blocking data-sharing arrangements.

3) Multiplatform support

The middleware shall have an OS abstraction layer to allow cross-development in development target-independent OS environments. This approach enables bug analysis on implementation/debugging-friendly OS/hardware environments different than the target environment.

When determining the specifications for the thread cycle

execution processing in Primary Requirement 1 and the data sharing in Primary Requirement 2, we used the task and variable synchronization functions' behavior[12] specified for our machine automation controller, the NJ/NX CPU unit, for reference.

3.1.2 Requirements for the middleware as a whole

This sub-subsection classifies the requirements for our middleware in accordance with the ISO/IEC 25010 software quality characteristics model[13]. Our middleware enables rapid prototyping and places weight on improving technology development productivity. As such, it does not consider the characteristics required of products, such as functional suitability, reliability, and security. Table 1 lists the required quality characteristics along with the requirements necessary for improved development efficiency, in other words, those for development environments and development support functions.

Table 1 Requirements for our middleware

Quality characteristics	Requirements
Performance efficiency	Allow multithread scheduling with a mixed inclusion of real-time and non-real-time processes.
Maintainability	Adaptable to robots with various kinematics through minimum modifications.
Compatibility	Connectable to external systems.
Portability	Operational on multiple platforms (multiple OS environments).
Usability	Enable development using high-level programming languages.
	Support building all such processes as recognition, planning, and control into a single system.
Others	Support CI (Continuous Integration)/CD (Continuous Delivery) in development environments.

3.2 Descriptions of functions

This subsection explains our middleware's primary functions, which meet the requirements presented in the previous subsection, and its auxiliary functions, which are integrations of the software requirements necessary to achieve rapid prototyping. Primary Functions 1 and 2 meet performance efficiency-related quality requirements, while Primary Function 3 meets portability-related quality requirements.

3.2.1 Primary functions

1) Multithread scheduling function

This function schedules process execution sequences based on the execution cycle/priority and scheduling method of each process. Parameters, such as cycle, priority, and scheduling method, can be set as desired by the developer to suit the desired system. This functional specification places weight on flexible execution scheduling, unlike conventional methods.

This function enables building a system that parallelly executes processes (such as the motor control part) that require real-time performance and other processes. The "real-time performance" herein means that a process takes place without delay at a constant cycle at expected times, constituting a necessary requirement for important processes for robot control. For example, a delay in the motor control part delays the tracking of motor command values, reducing robot position accuracy and path-tracking performance.

Fig. 2 schematically outlines the multithread scheduling function. The periodic and nonperiodic threads shown in Fig. 2 are generated based on the specified scheduling method. The execution processes executed in parallel ("proc#N" in Fig. 2) are assigned to threads as specified by the developer.

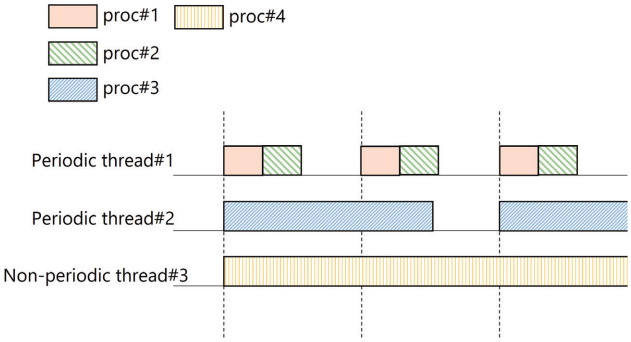

Fig. 2 Outline of multithread scheduling

2) Data-conflict prevention functions

The following two functions prevent inter-thread data conflicts:

① Inter-thread instruction transmission/reception function

General-purpose interface-based access (command function) is achieved to allow access to the middleware's internal modules from threads or other processes. The middleware allows a command receiver thread to stay resident to start each module's command execution thread upon command reception, guaranteeing the command execution sequence and ensuring the threads' execution. Moreover, two command transmission methods, one synchronous and the other asynchronous, are used to reconcile commands requiring such consecutive execution as with robot motions and those requiring no such sequentiality as with robots' internal status indication.

- Synchronous: After command transmission, the caller process is blocked until the callee accepts the command and receives the results. To be used when required to synchronize the caller process with command execution. Unsuitable for long execution time commands. In no periodic thread shall synchronous command transmission be

attempted. Failure to comply results in a blocked process.

- Asynchronous: After command transmission, the caller process becomes nonblocking when the callee accepts the command. The results are called back to a function registered at a specified command transmission time. Intended to be used for long execution time commands or in periodic threads.

The middleware includes a gadget that uses thread-safe queues for primary acceptance of commands to ensure that the middleware's internal modules receive commands on periodic threads, thereby minimizing the critical section of each periodic thread to reduce negative impacts on real-time performance.

② Data-sharing function

This function guarantees data change timing and provides access protection from other threads while maintaining inter-thread data consistency. The pub/sub model, also used in ROS and RT-middleware, is used to ensure each thread's data concurrency (Fig. 3). More specifically, this function defines the data used on individual threads as those to be published to or subscribed to from other threads, thereby ensuring publication after thread execution and subscription before thread start. Consequently, these data are updated before each thread's execution, ensuring the identity of the subscribed data during each thread's execution.

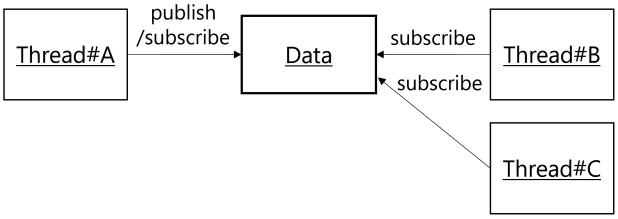

Fig. 3 Conceptual diagram of data sharing

3) Multiplatform support

The dependency between the operating system (OS) or hardware and the middleware should be reduced. Generally, the OS/hardware environment is often not fixed at the start of robotics technology development. Moreover, changes are often made to these environments during development. Therefore, the implementation process should use a development target-independent, implementation-friendly OS/hardware environment to pursue developments. The real-machine verification process should be flexibly adaptable to the target OS/hardware environment. Accordingly, an operating system abstraction layer (OSAL) is provided to abstract OS-specific processes to enable flexible switching between different operating systems (OS). A similar abstraction layer for abstracting hardware is provided to enable verification without hardware.

3.2.2 Auxiliary functions

This subsection describes the auxiliary functions one by one for each quality characteristic required of our middleware.

1) Maintainability

Fig. 4 shows the software architecture schematic. The software architecture consists of an application part corresponding to a user interface or debugger, a robot framework part (our proposed middleware), an OS part, and a hardware part. The robot framework includes function modules implemented with robot-specific features; a runtime for coordinating basic functions, such as scheduling, communication, and resource management; and an OSAL. Recognition, planning, control, and other robot-specific features are individually modularized/librarized in the function modules and structured to be plugged in. The resulting structure allows operation with a minimum module configuration on a desired system, achieving improved maintainability and reduced required computational resources.

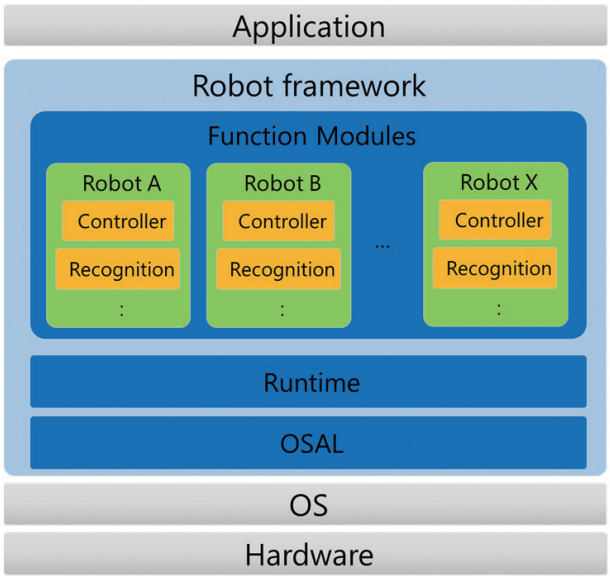

Fig. 4 Software architecture schematic

2) Compatibility

For functions outside the scope of technology development, external systems are used to reduce implementation man-hours and promote rapid prototyping. Therefore, a communication function with external systems is provided to exploit existing functions available from external systems, such as third-party software. Our middleware provides a communication function with ROS, allowing access to various systems that support ROS.

3) Usability/others

Our middleware supports cross-development and CI/CD environments. Achieving rapid prototyping in robotics technology developments requires turning the cycle of algorithm development and actual machine verification at high cycles. Accordingly, our middleware supports high-level language-friendly, Windows-based development environments or open-source, research-friendly Ubuntu-based development environments for algorithm development processes. Developers use these environments to perform implementations and simulations. Development deliverables undergo automated testing through a CI environment. On the other hand, the real-machine verification process auto-generates binaries for target environments through a CD environment. These resulting advantages reduce developers' workloads and human errors in simulation- or prototype-based preliminary verifications and actual robot verifications, achieving improved development efficiency.

4. Verification results

Based on a case of applying our middleware and its development environment, this section discusses our proposed robot middleware's performance verification results and development efficiency.

4.1 Verification environment

4.1.1 Robot system

This subsection presents an example of a mobile manipulator system built with our proposed robot middleware. Fig. 5 shows its outward appearance. A robot arm with eight degrees of freedom was installed on a wheeled mobile platform with two degrees of freedom. The wheeled platform and the arm end were fitted with an environment and object recognition camera.

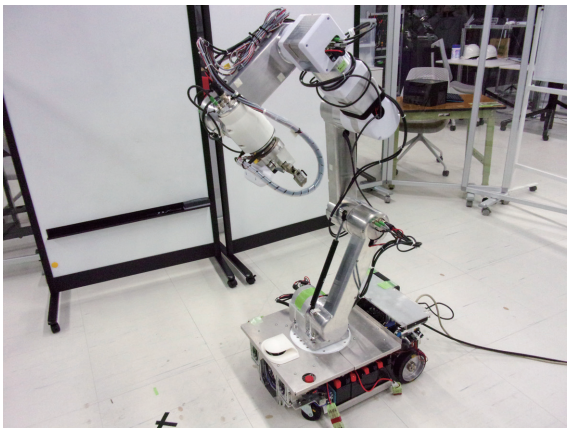

Fig. 5 Development case: External appearance of the mobile manipulator

Fig. 6 shows the internal configuration of our robot system. It consisted of cameras (vision sensors), an external-world recognition PC (environment recognition PC), a control PC and motors for wheel and arm operation (control PC and motors), and external sensors (sensors). This system executed self-position estimation (self-location estimation), path generation (path planning), path tracking (navigation), control (control), I/O control (I/O control), and motor command communication processing (motor control) in parallel, using simultaneous localization and mapping (SLAM)[14] to achieve the desired moving motions.

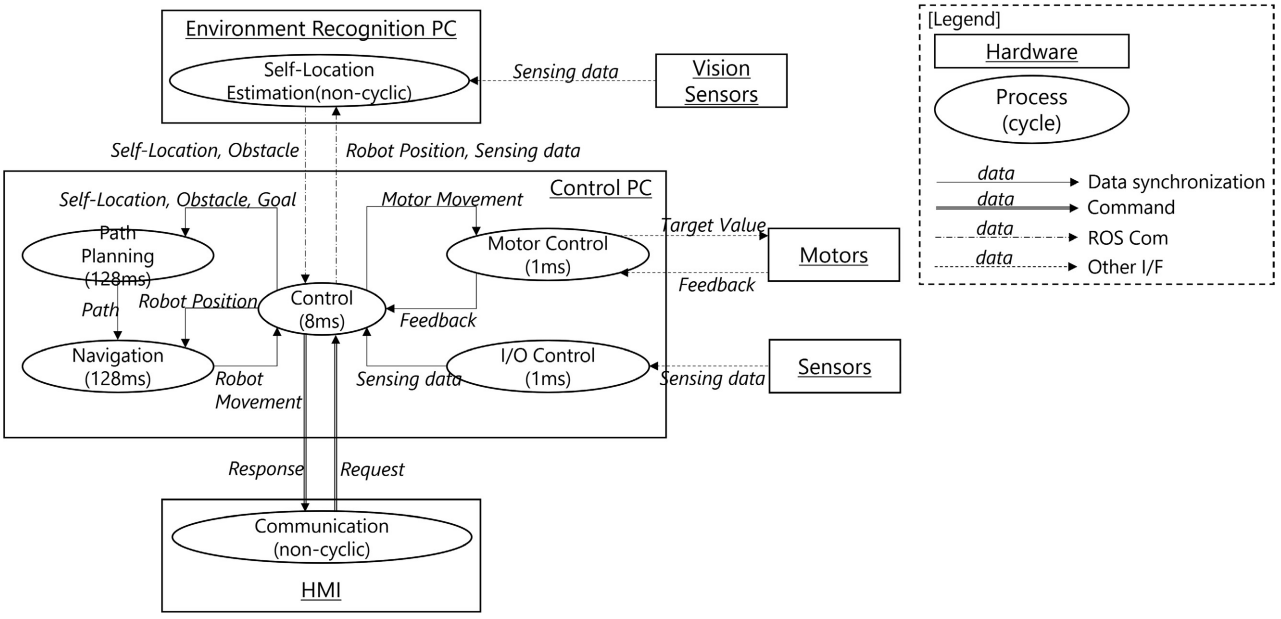

Fig. 6 Development case: Mobile manipulator system

Other than self-position estimation, processes were executed as periodic processes on the real-time OS control PC. Of these processes, the path planning and navigation processes, involving mathematical optimization and, hence, significant execution process time variations, were defined as a 128 [ms/cycle] thread with a long execution cycle. On the other hand, the control, I/O control, and motor control processes, requiring high real-time performance, were defined as a 1 [ms/cycle] and an 8 [ms/cycle] thread, respectively, with a short execution cycle. These periodic threads exchange data via the middleware's data-sharing function. For scheduling, the settings file was used to specify the cycle, priority, and execution thread.

Self-position estimation was executed as a nonperiodic process on the non-real-time OS environment recognition PC for reasons, such as a significantly longer required execution process time than the execution cycle or interfacing with sensors. This process was implemented in ROS to perform asynchronous communication with the control PC through the middleware's ROS communication function. Our robot system operated upon receipt of instructions from human-machine interfaces (HMI). The middleware's command function was used for motion instructions from HMIs.

4.1.2 Development environment

We built a development environment that executes such CI/CD jobs as in Fig. 7 to achieve improved development efficiency through multiplatform support. The algorithms and functions developed were controlled as function modules on GitLab to perform automated building/testing of pushed source code. The design quality was ensured by performing automated testing on execution binaries tailored to execution environments supported by our middleware.

The CI-CD build/test environment was built of a combination of a virtual machine on Amazon Web Services (AWS)[15] and an actual machine for deploying development deliverables. The virtual machine ran a docker to build and start CI environment containers.

4.2 Verification results and discussions

This subsection presents the results and discussions for each quality characteristic required of our middleware.

4.2.1 Performance efficiency

This sub-subsection discusses the performance efficiency of our built system when operating in an actual environment. The measurement environment used was a control PC installed with a VxWorks OS, an Intel Core i7−1165G7 2.8 GHz CPU, and 64 GB of memory.

1) Multithread scheduling

Fig. 8 shows a thread execution process for our built system during mobile manipulator travel. We obtained this information using the Wind River Workbench, a VxWorks integrated development environment (IDE).

Fig. 8 represents each thread's execution state in thick green lines and stand-by state in wavy lines. Each thread's name and priority are stated on the left side in Fig. 8, indicating that threads with smaller priority values were executed with higher priority. From top to down, 1 ms interrupts for periodic thread generation, (Interrupt 200), motor control, I/O control, control, and noncyclic threads 1, 2, ..., 5 are shown. Noncyclic threads were generated every time the inter-thread instruction transmission/reception function or the data-sharing function described in 3.2.1−2) and , respectively, was executed. Hence, several tens of threads were sometimes simultaneously generated. Fig. 8 demonstrates that multiple noncyclic threads (noncyclic threads 1 through 5), which significantly varied in execution process time, and the motor control, I/O control, and control threads, which strongly demand real-time performance, were stably and cyclically executed in an execution sequence

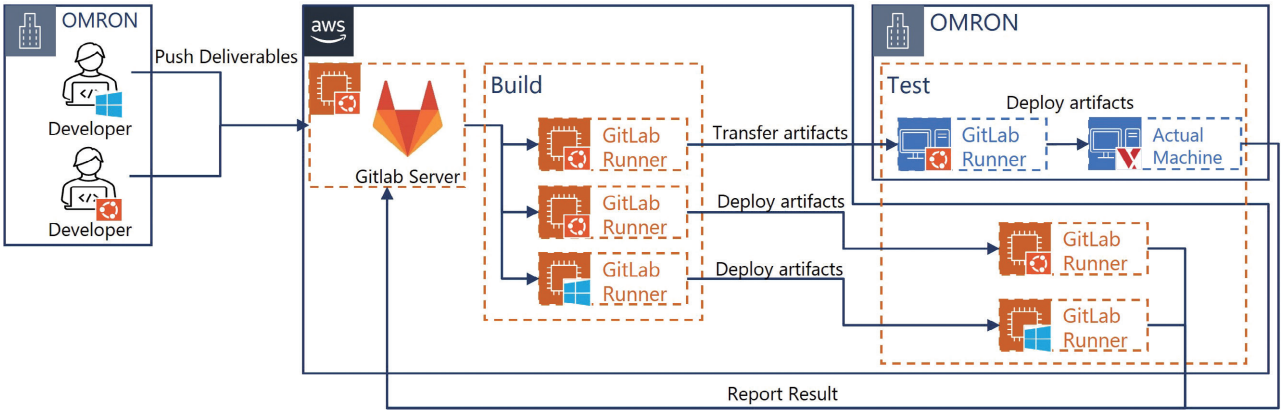

Fig. 7 Development environment

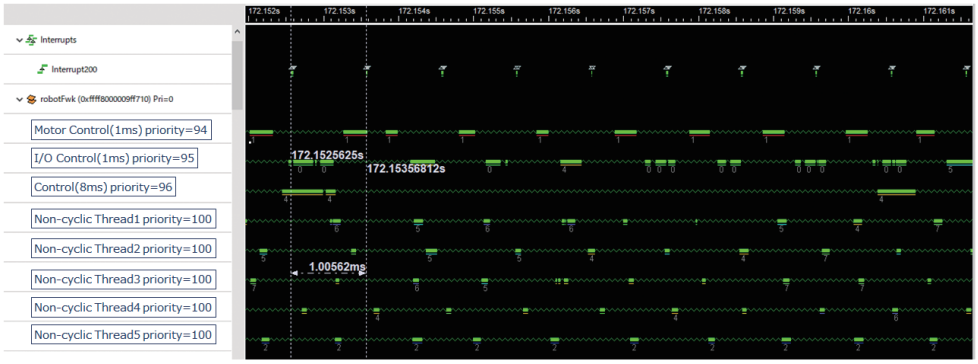

Fig. 8 Parallel thread scheduling

with an as-expected execution priority. Fig. 8 does not show the planning and navigation threads (128 ms cycle) for notational reasons.

2) Real-time performance

Table 2 shows the execution time statistics information (minimum, maximum, mean, and standard deviation) for the motor control, I/O control, and control threads, which strongly demand real-time performance:

Table 2 Execution time statistics information

Thread name (cycle [ms]/priority)	Minimum [ms]	Maximum [ms]	Mean [ms]	Standard deviation [ms]
Motor control (1/94)	0.066	0.117	0.074	0.006
I/O control (1/95)	0.163	0.229	0.172	0.011
Control (8/96)	0.211	0.369	0.276	0.031

Our robot system achieved periodic processing by executing threads at intervals specified based on 1 ms interrupts. The interrupt handling statistics data show that stable interrupts were achieved with a minimum value of 0.991 [ms], a maximum value of 1.009 [ms], a mean of 1.0 [ms], and a standard deviation of 0.001 [ms]. As shown in Table 2, each thread had a maximum value of less than one cycle, causing no spike that exceeded the cycle execution time. The standard deviation column shows that the motor control, I/O control, and control threads had a standard deviation of 0.006, 0.011, and 0.031 [ms], respectively, showing less variability in descending order of priority. The above confirms that threads requiring real-time performance were cyclically executed with stability.

4.2.2 Maintainability

In our robot system, robot-specific features were implemented and librarized in the function modules, as shown in Fig. 4. Meanwhile, shared features were achieved by loading necessary libraries for our middleware. Consequently, the robot designer was able to focus on developing the function module for the robots under their charge. As a result, two-armed robots were successfully developed concurrently with the theme under which our mobile manipulator was developed, confirming that our middleware can serve as a common framework for various robots.

4.2.3 Compatibility

The self-position estimation process was executed on a PC different from the control PC as shown in Fig. 6. We successfully developed the controls for this process and external sensors in a short period, using the ROS abundant with image processing libraries and sensor drivers. Using the communication function with the ROS, we easily imported this function into the robot system, confirming that we achieved improved development efficiency.

4.2.4 Portability

The command function allows access to module-specific functions in the runtime and function modules, whereby we improved development efficiency using verification tools suited for each development phase. For example, we were able to do the following, among other things: checking data published/subscribed via the data-sharing function during the runtime development phase, monitoring motor input/output values during the robot control function's development phase, and checking robot behaviors simulated by the robot simulator during the planning function development phase.

Besides, the command function, designed to be multiclient compatible, allows the simultaneous use of multiple tools as needed. Consequently, verification tools developed by individual function developers for their own work became readily available for reuse by others, enabling efficient analysis of problems encountered after the system became complicated as the development progressed.

Of these verification tools, those developed by individual developers were shared via CI environments to be improved by

other developers in a virtuous cycle. As a result, robot simulator tools have become so functionally sophisticated that they can visualize/verify random robots in virtual spaces. Visualized models of robots can be described in the Unified Robot Description Format (URDF)[16], a general-purpose robot description format. As such, they can be diverted to other future development themes and will be deployed together with our middleware.

4.2.5 Usability/others

1) GitLab-based CI (automatic build/test)

With this CI environment, the automated testing and execution binary generation for function modules developed by designers on the OS of their choice became performable on the actual machine in the target environment. Consequently, we achieved a seamless sequence of steps from verification by simulation to actual operation on the real machine without lead times. As a result, each designer's verification has extended to include operation on a real machine with real-time performance considerations, accelerating the development of algorithms and functions usable for real robots.

2) Server resource allocation optimization using AWS

Our middleware's operability on virtual machines has made copying and initialization easier, reducing the conventional workloads of server/network load adjustment and management. As a result, it has become possible to complete build/test server addition/removal or failure recovery tasks within 30 minutes according to the development status.

5. Conclusions

A technical issue exists to the early achievement of robots that flexibly move in the same space as humans and automate nonroutine tasks. One example is the rapid prototyping-based, efficient development of robots that flexibly adapt and respond to dynamic changes.

For solving this issue, this paper proposed technology development middleware that enables rapid prototyping-based development of robots with a parallel behavior architecture. More specifically, this paper described our proposed middleware's characteristics and advantages over conventional middleware. Our middleware has enabled scheduling with guaranteed multithread real-time performance, which is not readily achievable with ROS/ROS2. Moreover, it saves the need to consider individual data-sharing methods or scheduling. Hence, it follows that this middleware provides higher development efficiency to users than OpenRTM-aist.

Moreover, this paper discussed the performance verification results and development efficiency of our proposed middleware based on a case of applying it and its software development environment.

Moving forward, we will further improve our middleware and continue delivering innovative robots to the world with a view towards the social implementation of robots equipped with a parallel behavior architecture.

References

1) OMRON. "Next-Generation Lab Automation" for Personnel Motivation, Pursued by Chugai Pharma, OMRON, and OMRON SINIC X." (in Japanese), OMRON EDGE&LINK. https://www.omron.com/jp/ja/edge-link/news/688.html (Accessed: Jun. 5, 2024).
2) R. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE J. Robot. and Automat*., vol. 2, no. 1, pp. 14–23, 1986.
3) T. Yamamoto and T. Sugihara, "SEAN System of a Biped Robot Based on Stacked Modulation Architecture," in *Proc. JSME Annu. Conf. Robot. Mechatronics (Robomec),* pp. 1A1-D05, 2021.
4) Quigley Morgan et al., "ROS: An Open-source Robot Operating System," *ICRA Workshop on Open Source Softw.*, vol. 3, no. 3.2, p. 5, 2009.
5) S. Macenski et al., "Robot Operating System 2: Design, Architecture, and Uses in the Wild," *Sci. Robot.*, vol. 7, no. 66, 2022.
6) N. Ando et al., "RT-middleware: Distributed Component Middleware for RT (Robot Technology)," in *IEEE/RSJ Int. Conf. Intell. Robots and Syst.*, 2005, pp. 3933–3938.
7) *Robotic Technology Component Specification*, formal/08–04–04, 2008.
8) N. Ando et al., "A Software Platform for Component-Based RT-System Development: OpenRTM-aist," in *Int. Conf. Simul., Model., and Program. Auton. Robots*, 2008, pp. 87–98.
9) N. Ando, "Robotic Middleware Comparison of Real-time Processing and Mutual Exclusion," (in Japanese), *J. Robot. Soc. Jpn.*, vol. 34, no. 6, pp. 366–369, 2016.
10) J. Kay. "Introduction to Real-time Systems." ROS 2 Design. https://design.ros2.org/articles/realtime_background.html (Accessed: Jun. 5, 2024).
11) M. Sugaya et al., "IXM: Rapid Inter-process Communication Middleware for Robotics Software," *J. Inform. Process. Soc. Jpn.*, vol. 58, no. 10, pp. 1578–1590, 2017.
12) OMRON Corporation, *NJ/NX-series CPU Unit Software User's Manual* (SBCA-467Z), (in Japanese), pp. 5–46–5–90.
13) *Systems and Software Engineering*, ISO/IEC 25010, 2011.
14) H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part I," *IEEE Robot. & Automat. Mag*., vol. 13, no. 2, pp. 99–110, 2006.
15) AWS. "OMRON Builds R&D HPC Foundation on AWS to Lead Innovative Technology Development Using Optimal Computing Resources." (in Japanese), AWS Implementation Case: OMRON Corporation | AWS. https://aws.amazon.com/jp/solutions/

case-studies/omron-case-study/ (Accessed: Jun. 19, 2024).
16) ROS.org. “urdf.” urdf - ROS Wiki. https://wiki.ros.org/urdf (Accessed: Jun. 19, 2024).

About the Authors

MATSUNAGA Daisuke
Voyager Project Dept. Robotics R&D Center
Technology and Intellectual Property HQ.
Speciality: Software Engineering

YAMAMOTO Tomoya
Voyager Project Dept. Robotics R&D Center
Technology and Intellectual Property HQ.
Speciality: Software Engineering

FUJII Haruka
Voyager Project Dept. Robotics R&D Center
Technology and Intellectual Property HQ.
Speciality: Software Engineering

KOJIMA Takeshi
Voyager Project Dept. Robotics R&D Center
Technology and Intellectual Property HQ.
Speciality: Software Engineering