IDF-Autoware: Integrated Development Framework for ROS-Based Self-Driving Systems Using MATLAB/Simulink

Shota Tokunaga
Graduate School of Engineering Science, Osaka University, Osaka, Japan

Yuki Horita
Hitachi, Ltd., Yokohama, Japan

Yasuhiro Oda
Hitachi Automotive Systems, Ltd., Hitachinaka, Japan

Takuya Azumi
Graduate School of Science and Engineering, Saitama University, Saitama, Japan

Abstract

This paper proposes an integrated development framework that enables co-simulation and operation of a Robot Operating System (ROS)-based self-driving system using MATLAB/Simulink (IDF-Autoware). The management of self-driving systems is becoming more complex as the development of self-driving technology progresses. One approach to the development of self-driving systems is the use of ROS; however, the system used in the automotive industry is typically designed using MATLAB/Simulink, which can simulate and evaluate the models used for self-driving. These models are incompatible with ROS-based systems. To allow the two to be used in tandem, it is necessary to rewrite the C++ code and incorporate them into the ROS-based system, which makes development inefficient. Therefore, the proposed framework allows models created using MATLAB/Simulink to be used in a ROS-based self-driving system, thereby improving development efficiency. Furthermore, our evaluations of the proposed framework demonstrated its practical potential.

2012 ACM Subject Classification Information systems → Open source software

Keywords and phrases self-driving systems, framework, robot operating system (ROS), MATLAB/Simulink

Digital Object Identifier 10.4230/OASIcs.ASD.2019.3

Funding This work was partially supported by Mr. Tohru Kikawada and JST PRESTO, Japan (grant No. JPMJPR1751).
Yasuhiro Oda: Presently with Hitachi Industry & Control Solutions, Ltd.

1 Introduction

Self-driving systems continuously increase in complexity along with the increasing number of required functionalities. One approach to the development of complicated systems is the use of Robot Operating System (ROS) [5] [12] [13]. ROS characteristics, such as abstracting hardware and improving code reusability, make the development of such systems more efficient. A ROS-based self-driving system is Autoware [1]. Autoware is open-source software for autonomous vehicles and can be used in embedded systems, such as NVIDIA DRIVE PX2 [10] and Kalray MPPA-256 [11].

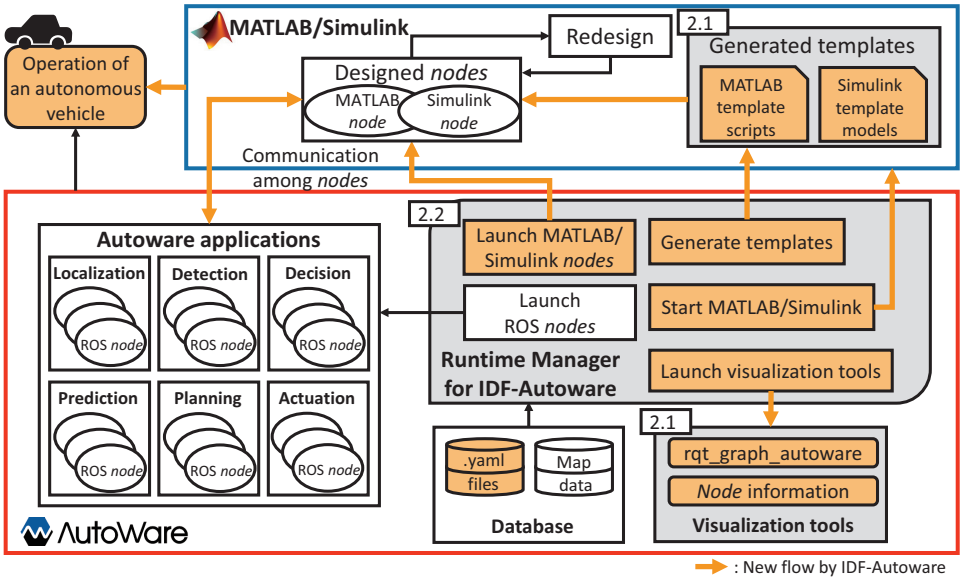

Figure 1 System model of IDF-Autoware.

However, in the automotive industry, the design of self-driving subsystems, such as detection, planning, and control have often used MATLAB®/Simulink® [3]. The models designed using MATLAB/Simulink can not be directly linked to Autoware in the currently adopted development framework. To integrate such models into Autoware, it is necessary to generate and incorporate the associated C++ code. Although MATLAB/Simulink has a C++ code generation functionality, code corresponding to Autoware (i.e., ROS) can not be generated, thereby deteriorating development efficiency. Moreover, it is possible that a model ported to Autoware will not perform as designed because the MATLAB/Simulink environment differs from that of Autoware. To address these limitations, we propose a framework called IDF-Autoware [2] (Figure. 1) that manages models designed using MATLAB/Simulink as *nodes* that represent individual processes in ROS. This enables data exchange between Autoware and MATLAB/Simulink, thereby allowing the models to be used without incorporation into Autoware.

To the best of our knowledge, this is the first work that co-simulation and operation of a real vehicle using MATLAB/Simulink for self-driving systems. The main contributions of this study are as follows:

- We confirmed the practicality of the method by comparing the data transfer time and processing capacity of ROS and MATLAB/Simulink (Section 3.1), as well as that the *nodes* designed using MATLAB/Simulink could be applied to the co-simulation and operation of an autonomous vehicle;
- We improved the design efficiency in MATLAB/Simulink based on IDF-Autoware generating MATLAB template scripts and Simulink template models (Section 3.2), which help a developer design *nodes* for Autoware using MATLAB/Simulink;
- We improved usability by extending Runtime Manager, which is a graphical user interface (GUI) tool for Autoware, to enable operations for MATLAB/Simulink (Section 3.3), as well as making available the other functionalities provided by IDF-Autoware (e.g., template generation).

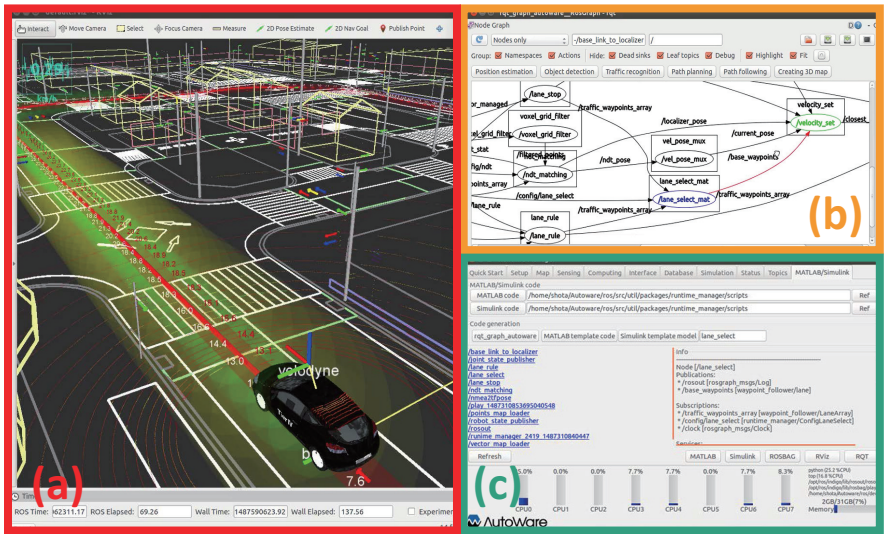

Figure 2 Screenshot of co-simulation using IDF-Autoware: (a)RViz displaying Autoware status, (b) the rqt_graph_autoware, and (c) the Runtime Manager for IDF-Autoware.

2 Design and Implementation

The functionalities provided by IDF-Autoware facilitate the integrated development of Autoware and MATLAB/Simulink. The key functionalities are as follows (Figure. 1):

- They generate MATLAB template scripts and Simulink template models, and provide visualization tools to aid template generation (Section 2.1);
- They enable MATLAB/Simulink to operate on Runtime Manager, to display *node* information, and to make use of the other provided functionalities (Section 2.2)

In this section, we discuss the design and implementation of each of these functionalities, and use cases of the proposed framework are shown.

2.1 Template Generation

When MATLAB/Simulink is used to design *nodes* for Autoware, the *nodes* must contain essential information, such as a *node* name, the *topics* to publish/subscribe, and the *message* type of each *topic*. This information can be obtained by analyzing the source code of Autoware and executing ROS commands. However, the need for such analyses places a burden on developers, especially on those who are unfamiliar with ROS. Therefore, we provided functionalities that allow the generation of MATLAB template scripts and Simulink template models that include this necessary information, as the templates help developers design *nodes* in MATLAB/Simulink. Additionally, we made two visualization tools to aid the template generation. One is the rqt_graph_autoware plugin (Figure. 2 (b)). In addition to the functionalities of rqt_graph [7], rqt_graph_autoware can render *node* dependency, such as sensing, perception, decision, and planning, for Autoware applications. The other tool displays a list of the running *nodes* and provides information on any *node* selected from the list.

As noted, before the template of a desired *node* is generated, it is necessary to obtain node information; therefore, a .yaml file containing information pertaining to all Autoware *nodes* was created. Based on this information, templates are created using functions provided by Robotics System Toolbox™ [4], which provides the interface between ROS and MATLAB/Simulink. Developers can create *nodes* for Autoware in MATLAB/Simulink using the generated template.

To implement the rqt_graph_autoware plugin, we created .dot files that render *node* dependency graphs for each Autoware's application. Moreover, to create the GUI for rqt_graph_autoware, we added buttons to rqt_graph using Qt designer, which is a Qt tool for designing a GUI. The buttons were configured to open each .dot file, and clicking on these buttons cause a graph to be drawn. This allows developers visualization of the *nodes* included in each Autoware's application.

To display *node* information, we used a rosnode command-line tool [6] that includes commands that fetch *node* information, including *rosnode list* and *rosnode info node_name*. The *rosnode list* command displays a list of running *nodes*, whereas *rosnode info node_name* displays information about the *topics* to be published/subscribed by the *node*. Displaying the results of these commands in Runtime Manager renders the *node* information easily comprehensible. Section 2.2 describes the method for displaying these results in Runtime Manager.

2.2 Runtime Manager for IDF-Autoware

Autoware and MATLAB/Simulink are operated with different GUI tools; thus, this is troublesome for users who want to use the two simultaneously. Therefore, we added GUIs to the Autoware's GUI tool (i.e., Runtime Manager) to allow use of MATLAB/Simulink and the functionalities provided in IDF-Autoware (Figure. 2 (c)). These GUIs enabled the following functionalities:

- Starting MATLAB, Simulink, and rqt_graph_autoware;
- Executing MATLAB scripts and Simulink models;
- Generating MATLAB template scripts and Simulink template models;
- Displaying *node* information.

This unification of operation method simplifies the MATLAB/Simulink operation and the utilization of the provided functionalities.

Runtime Manager was designed using the wxPython toolkit [9]. Therefore, we designed the GUIs for the added functionalities using wxGlade [8], and outputted its designs as wxPython. The GUIs involve buttons and panels that execute each functionality.

We next modified the Runtime Manager execution code to configure them for GUI functionalities. The execution code imports modules, including the code generated by wxGlade, and loads the .yaml files. In the execution code, loading .yaml files initiates functions that align simple operations to specified buttons. Therefore, by creating a yaml file for MATLAB/Simulink, we configured the initiation of MATLAB, Simulink, and rqt_graph_autoware to each button.

To allow the execution of MATLAB scripts and Simulink models from Runtime Manager, we created multiple GUIs with the following configurations:

- A button to open a dialog for file selection;
- A panel displaying the absolute path of the selected file; and
- A button to execute the file displayed on the panel.

This execution button was designed to run if the selected file was a MATLAB/Simulink file (i.e., a .m or .slx file).

Table 1 Evaluation environment.

CPU	**Model number**	Intel Core i7-6700K
	Cores	4
	Threads	8
	Frequency	4.00 GHz
Memory		32 GB
ROS		Indigo
MATLAB/Simulink		R2016b
OS		Ubuntu 14.04.5 LTS

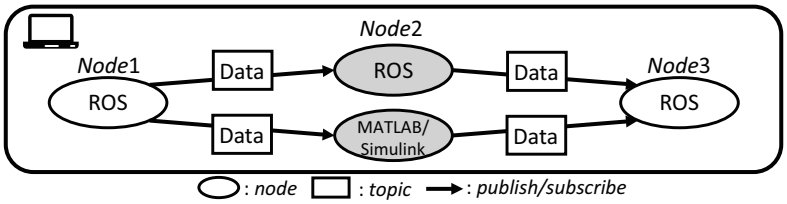

Figure 3 Measurement of transfer time.

To generate MATLAB template scripts and Simulink template models, we designed the following GUIs: a panel to input the *node* name and buttons to run the execution code that generates the template of the input *node*.

For the *node* information display, we designed two panels, with the first displaying the output of the executing *rosnode list*. When a *node* is selected from the list, the second panel displays the output of *rosnode info the_selected_node_name*, which eliminates the need to enter the rosnode command.

2.3 Use Case

IDF-Autoware allows co-simulation of Autoware and MATLAB/Simulink. The demonstration video can be viewed at the following hyperlink: `https://youtu.be/X4d9VbXnPeg` (Figure. 2). In this video, one of the *nodes* necessary for planning is executed by MATLAB/Simulink. This simulation facilitates an operational check of MATLAB/Simulink *nodes*. Moreover, it can also be used for experiments using an autonomous vehicle. The demonstration video showing operating of the autonomous vehicle using IDF-Autoware can be seen at the following hyperlink: `https://youtu.be/wusCU2VPGGQ`.

3 Evaluations

The main goal of this study was to improve development efficiency. To demonstrate this improvement, the practicality of IDF-Autoware, efficiency, and usability were evaluated. To evaluate the practicality, we compared the communication times among *nodes* within ROS and between ROS and MATLAB/Simulink. Additionally, we performed a co-simulation and operation of an autonomous vehicle to show the practicality of the proposed framework. We investigated the design efficiency by measuring the generated MATLAB/Simulink template. To evaluate the usability, we compared the development environments with Autoware, Robotics System Toolbox, and IDF-Autoware. These evaluations demonstrated that IDF-Autoware improved the development efficiency. Table 1 summarizes the software and hardware environments used in the experiments.

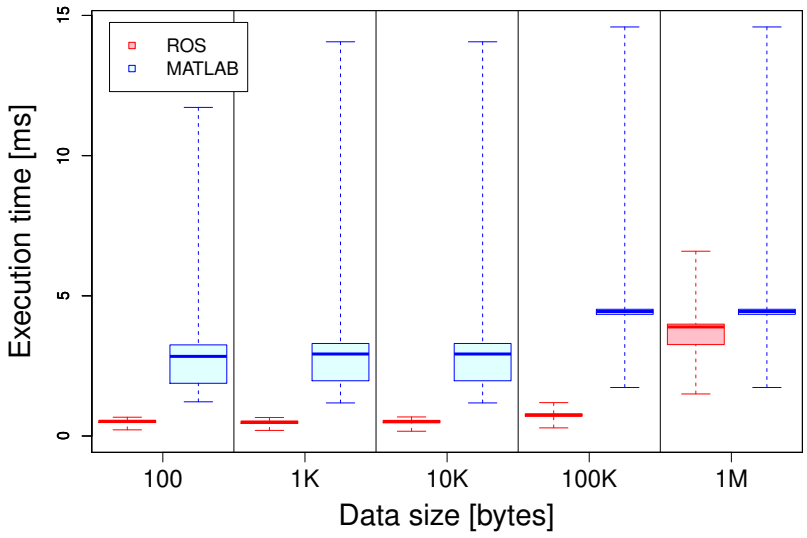

Figure 4 The average transfer time according to the size of the *message* data.

3.1 Practicality

IDF-Autoware enabled the communication of nodes designed using MATLAB/Simulink with Autoware *nodes* to improve the development efficiency. However, it was necessary to consider the effect of using Autoware with only ROS and together with MATLAB/Simulink together. Therefore, to evaluate practicality, ROS and MATLAB/Simulink were compared as follows:

1. According to the relationship between the transfer time and the data size when a *message* is sent via ROS and via MATLAB/Simulink, respectively; and

2. According to the processing capacity when the same type of method was used.

As shown in Figure. 3, the transfer time was defined as the elapsed time when *Node* 1 published the *message* to *Node* 3, which subscribed the *message* via *Node* 2. The processing capacity was compared with the processing time over 1,000 iterations and using the same machine (*Node* 1 published the *message* at 10 Hz).

We measured the ROS and MATLAB/Simulink transfer time when the *message* data size on each *topic* was set to 100, 1 K, 10 K, 100 K, and 1 M bytes. Figure. 4 shows the transfer times via ROS and MATLAB/Simulink plotted against each data size. Both the ROS and MATLAB/Simulink transfer times increased along with data size, although the data transfer by MATLAB/Simulink had an overhead exceeding that of ROS. However, the MATLAB/Simulink transfer time did not exceed the Autoware maximum of 32 Hz.

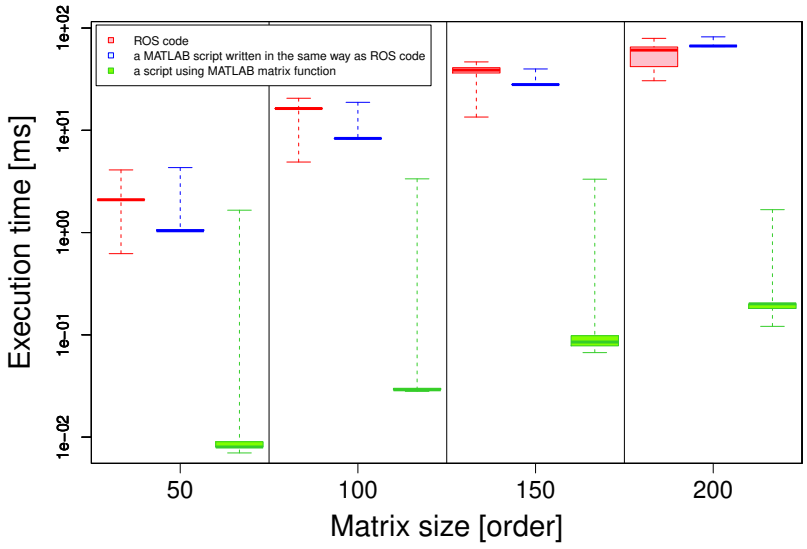

Figure 5 The average processing time according to each matrix size.

Table 2 Task reduction using MATLAB template scripts.

	MATLAB template scripts
Generated lines	(1) + α(2) + β((3) + 2(4))

(1): Defining *node*
(2): Defining *publisher*
(3): Defining *subscriber*
(4): Defining callback function
α: The number of *publishers*
β: The number of *subscribers*

To evaluate the processing capacity, we measured the processing times of ROS and MATLAB/Simulink when multiplying square matrices on the order of 50, 100, 150, and 200, which served as easy points of reference to enable comparison of ROS with MATLAB/Simulink rather than as a requirement for self-driving. The evaluation measured the time required to process the time complexity at each matrix size and assessed the performance of the functions provided by MATLAB/Simulink. Therefore, the MATLAB/Simulink processing time was measured using two MATLAB scripts: one written in the same way as the ROS code, and the other using MATLAB matrix functions. Figure. 5 shows the processing times at each matrix size. When using the MATLAB script written in the same way as the ROS code, the processing times of ROS and MATLAB/Simulink were approximately the same. By contrast, when the MATLAB script used matrix functions, its processing time was significantly shorter than that of the other two methods, because processing was executed on multiple cores with multiple threads, even when this was unspecified. Comparison of the processing times with the transfer times revealed that the script using matrix functions was again significantly faster, thereby confirming that application of the functions provided by MATLAB/Simulink code enabled the handling of processes with large time complexity (e.g., image processing), even when accounting for the transfer time. Therefore, as shown the videos in Section 2.3, the practicality of IDF-Autoware is demonstrated.

Table 3 Task reduction using Simulink template models.

	Simulink template models
Simulink blocks	$\alpha((1) + (2) + (3)) + \beta((4) + (5) + (6))$
Settings	(i) + $(\alpha + \beta)$((ii) + (iii) + (iv) + 2(v))

(1): Placing *Publisher*	(i): Defining model name
(2): Placing *Message*	(ii): setting *message* name
(3): Placing Bus Assignment	(iii): setting *topic* name
(4): Placing *Subscriber*	(iv): Configuring *topic* source
(5): Placing Bus Selector	(v): Connecting blocks
(6): Placing Terminal	α: The number of *publishers*
	β: The number of *subscribers*

Table 4 Functionalities available with Autoware, Robotics System Toolbox, and IDF-Autoware.

	Autoware [1]	**Robotics System Toolbox** [4]	**IDF-Autoware** [2]
Operating Autoware	✓		✓
Operating MATLAB/Simulink		✓	✓
Communicating between Autoware and MATLAB/Simulink		✓	✓
Drawing node dependency	✓		✓
Generating MATLAB/Simulink templates			✓
Displaying node information			✓

3.2 Efficiency

To improve the design efficiency, a functionality to generate both MATLAB template scripts and Simulink template models was provided. These templates help developers design *nodes* for Autoware in MATLAB/Simulink.

Table 2 shows the amount of the template generated by a MATLAB template script. The MATLAB template script defines the essential information, as mentioned in Section 2.1, and creates callback functions utilized when a *topic* is subscribed. For example, the lane_stop *node* required for planning has one *publisher* and five *subscribers*. One line is generated to define a *node*, a *subscriber*, and a *publisher*, and two lines are generated to define the callback function. Therefore, in total, 17 lines are generated for the MATLAB template script for the lane_stop *node*.

When creating a Simulink model, it is necessary to place and configure the Simulink blocks, to define the model name, and to connect the blocks. Table 3 summarizes the number of Simulink blocks placed and the settings created by a Simulink template model. The Simulink template model defines the model name and places the essential Simulink blocks, thereby creating a model for Autoware. Additionally, the Simulink blocks are configured and connected together. For example, when the Simulink template model of lane_stop *node* is generated, 18 Simulink blocks are placed and 31 settings are configured in total.

If the functionality allowing MATLAB/Simulink templates to be generated is not provided, the developer must examine the *node* information and define it in a MATLAB script or a Simulink model. By contrast, when the templates are used, this becomes unnecessary; therefore, this improves design efficiency.

3.3 Usability

IDF-Autoware enables the operation of MATLAB/Simulink in Autoware and provides functionalities to improve the usability. Here, we compared the available functionalities between Autoware, Robotics System Toolbox, and IDF-Autoware, as summarized in Table 4.

Autoware cannot operate MATLAB/Simulink, and Robotics System Toolbox cannot operate Autoware. IDF-Autoware provides functionalities required to operate MATLAB/Simulink in Runtime Manager for IDF-Autoware, such as starting MATLAB/Simulink or executing MATLAB scripts and Simulink models. Therefore, IDF-Autoware can operate both systems. Communication between Autoware and MATLAB/Simulink is possible in Robotics System Toolbox and IDF-Autoware. Moreover, IDF-Autoware provides a drawing to visualize *node* dependency using the rqt_graph_autoware plugin created by extending rqt_graph available in Autoware. In addition to these features, IDF-Autoware can generate MATLAB/Simulink templates and display *node* information. Because this increases the number of available functionalities, the usability is also enhanced, which in turn improves development efficiency.

4 Conclusion

In this paper, we described the development of an integrated development framework for Autoware with MATLAB/Simulink (IDF-Autoware) that enabled communication between Autoware and MATLAB/Simulink. We evaluated the data transfer time and processing capacity of MATLAB/Simulink, and confirmed the practicality of the method by using both co-simulations and experiments using an autonomous vehicle. IDF-Autoware facilitated the generation of MATLAB/Simulink templates that can help developers create models using MATLAB/Simulink for Autoware, thereby improving the design efficiency. Furthermore, the functionalities added to IDF-Autoware allow Runtime Manager to operate MATLAB/Simulink and various functionalities, further improving usability. Our findings confirmed that IDF-Autoware improved the development efficiency.

References

1 Autoware/github.com. URL: `http://github.com/CPFL/Autoware`.
2 IDF-Autoware/github.com. URL: `https://github.com/T-Shota/IDF-Autoware`.
3 MATLAB/Simulink. URL: `http:///www.mathworks.com`.
4 Robotics System Toolbox. URL: `https://mathworks.com/products/robotics.html`.
5 ROS.org. URL: `http://www.ros.org`.
6 ROS.org/rosnode. URL: `http://wiki.ros.org/rosnode`.
7 ROS.org/rqt_graph. URL: `http://wiki.ros.org/rqt_graph`.
8 wxglade.sourceforge.net. URL: `http://wxglade.sourceforge.net`.
9 wxpython.org. URL: `http://wxpython.org`.
10 S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *Proc. of ICCPS*, 2018.
11 Y. Maruyama, S. Kato, and T. Azumi. Exploring Scalable Data Allocation and Parallel Computing on NoC-Based Embedded Many Cores. In *Proc. of ICCD*, 2017.
12 M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Proc. of ICRA, Open-Source Software Workshop*, 2009.
13 Y. Saito, T. Azumi, S. Kato, and N. Nishio. Priority and Synchronization Support for ROS. In *Proc. of CPSNA*, 2016.