저작자표시-비영리-변경금지 2.0 대한민국

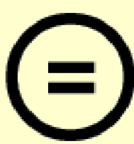
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

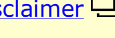


변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer


공학박사학위논문

협업 로봇을 위한 서비스 기반과 모델 기반의 소프트웨어 개발 방법론

Service-oriented and Model-based Software Development Methodology for Cooperating Robots

2020년 2월

서울대학교 대학원
전기.컴퓨터공학부
홍 혜 선

Abstract

Service-oriented and Model-based Software Development Methodology for Cooperating Robots

Hyesun Hong
Department of Electrical Engineering and Computer Science
College of Engineering
The Graduate School
Seoul National University

In the near future, it will be common that a variety of robots are cooperating to perform a mission in various fields. There are two software challenges when deploying collaborative robots: how to specify a cooperative mission and how to program each robot to accomplish its mission. In this paper, we propose a novel software development framework that separates mission specification and robot behavior programming, which is called service-oriented and model-based (SeMo) framework[1]. Also, it can support distributed robot systems, swarm robots, and their hybrid.

For mission specification, a novel scripting language is proposed with the expression capability. It involves team composition and service-oriented behavior specification of each team, allowing dynamic mode change of operation and multi-tasking. Robots are grouped into teams, and the behavior of each team is defined

[1]This thesis includes our recent work [1] https://ieeexplore.ieee.org/document/8412595 and [2] https://dl.acm.org/doi/10.1145/3061639.3062260. It is slightly amended version of [1] and [2].

with a composite service. The internal behavior of a composite service is defined by a sequence of services that the robots will perform. The notion of plan is applied to express multi-tasking. And the robot may have various operating modes, so mode change is triggered by events generated in a composite service. Moreover, to improve the robustness, scalability, and flexibility of robot collaboration, the high-level mission scripting language is extended with new features such as team hierarchy, group service, one-to-many communication. We assume that any robot fails during the execution of scenarios, and the grouping of robots can be made at run-time dynamically. Therefore, the extended mission specification enables a casual user to specify various types of cooperative missions easily.

For robot behavior programming, an extended dataflow model is used for task-level behavior specification that does not depend on the robot hardware platform. To specify the dynamic behavior of the robot, we apply an extended task model that supports a hybrid specification of dataflow and finite state machine models. Furthermore, we propose a novel extension to allow the explicit specification of loop structures. This extension helps the compute-intensive application, which contains a lot of loop structures, to specify explicitly and analyze at compile time. Two types of information sharing, global information sharing and local knowledge sharing, are supported for robot collaboration in the dataflow graph. For global information, we use the library task, which supports shared resource management and server-client interaction. On the other hand, to share information locally with near robots, we add another type of port for multicasting and use the knowledge sharing technique. The actual robot code per robot is automatically generated from the associated task graph, which minimizes the human efforts in low-level robot programming and improves the software design productivity significantly.

By abstracting the tasks or algorithms as services and adding the strategy

description layer in the design flow, the mission specification is refined into task-graph specification automatically. The viability of the proposed methodology is verified with preliminary experiments with three cooperative mission scenarios with heterogeneous robot platforms and robot simulator.

Keywords : High-level Specification, Dataflow, SDF Graph, Iterative Behavior Specification, Shared Information Management, Automatic Code Generation, Software Development Framework, Cooperating Robots
Student Number : 2013-20912

Contents

Abstract . . . . **i**

Contents . . . . **iv**

List of Figures . . . . **vii**

List of Tables . . . . **x**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 7

1.3 Dissertation Organization 9

Chapter 2 Background and Existing Research **11**

2.1 Terminologies 11

2.1.1 Robot 11

2.1.2 Multiple Robots 15

2.1.3 Robot Software 21

2.2 Robot Software Development Frameworks 25

2.2.1 Manufacturer-specific Software Framework 25

2.2.2 General-purpose Software Framework 25

2.3 Parallel Embedded Software Development Framework 31

2.3.1 HOPES Approach 35

Chapter 3 Overview of the SeMo Framework **41**
3.1 Motivational Examples 45
3.1.1 Example 1: Robots following specific object 45
3.1.2 Example 2: Scouting Mission With Multiple Heterogeneous Robots (Simple Version) 45
3.1.3 Example 3: Scouting Mission With Multiple Heterogeneous Robots (Complicated Version) 45

Chapter 4 Robot Behavior Programming **47**
4.1 Related works 48
4.1.1 Synchronous Dataflow (SDF) Model 48
4.1.2 Extensions of SDF model to Dynamic Behavior 50
4.1.3 Extensions of SDF model to Iterative Behavior 52
4.1.4 Extensions of SDF model to Shared Resource Management 53
4.2 Model-based Task Graph Specification for Individual Robots . . . 56
4.2.1 Applying Dynamic Behavior Specification 57
4.2.2 Extending Iterative Behavior Specification 58
4.3 Model-based Task Graph Specification for Cooperating Robots . . 70
4.3.1 Globally Shared Information Specification 70
4.3.2 Locally Shared Information Specification 72
4.4 Automatic Code Generation 74
4.5 Experiments 78
4.5.1 Memory Usage Comparison with ROS 79

Chapter 5 High-level Mission Specification **81**
5.1 Service-oriented Mission Specification 82
5.1.1 Service-oriented Programming 84
5.1.2 Multitasking and Dynamic Mode Change 87

5.1.3 Extensions for Multiple Robots 88
5.2 Strategy Description 93
5.3 Automatic Task Graph Generation 96
5.4 Related works 99
5.5 Experiments 104
5.5.1 Swarm Robotics Example 104
5.5.2 Scouting Mission with Heterogeneous Robots 104

Chapter 6 Conclusion 114
6.1 Future Research 116

Bibliography 118

Appendices 133
A The template for mission specification 133
B Illustration of the SeMo SW development flow for the experiment of Section 3.1.2 136
B.1 Mission Specification 136
B.2 Strategy Description 143
B.3 Task Graph Specification 147
C Mission Specification for the experiment of Section 3.1.3 149

요 약 158

List of Figures

Figure 2.1 Robot classification by use, image from [3], [4], [5], [6], [7], [8], [9] 12
Figure 2.2 Robot classification by movement method, image from Big Dog [10], DynaRoACH [11], crawling robot [12], e-puck [13], HUBO [14], Robobee [15], iBird [16], BOLT [17], Bitdrone[18], DJI Phantom 3 [19], Perambulator [20], Robotic lobster [21], Boxybot [22], Aqua AUV [23], water strider [24] 14
Figure 2.3 Examples of swarm robot 17
Figure 2.4 Cooperating robot 18
Figure 2.5 Examples of swarm robot behavior 20
Figure 2.6 Software architecture of robot 22
Figure 2.7 Comparison of the software structure: ROS and the proposed framework 26
Figure 2.8 Robot operating system 26
Figure 2.9 Design methods of parallel embedded software 32
Figure 2.10 Y-chart approach for designing MPSoC 34
Figure 2.11 HOPES Design flow 36
Figure 2.12 Software structure based on the universal execution model (UEM) 37
Figure 2.13 Hierarchical software architecture of the universal execution model (UEM) 38

Figure 2.14 A task graph and the code template of a task 39
Figure 3.1 Overview of our proposed SeMo framework 42
Figure 3.2 A captured screen of the SeMo script editor 42
Figure 4.1 An example of SDF graph 49
Figure 4.2 A control task and its example code 51
Figure 4.3 SADF specification of the example of Fig. 4.8 where E is the detector node. Symbols in the figure are the same as [25]. . 52
Figure 4.4 A library task and its example code 54
Figure 4.5 The code templates of the caller task in the figure 4.4 . . . 55
Figure 4.6 Task graph specification of individual robot platform 57
Figure 4.7 (a) An SDF graph that has a loop structure by sample rate changes, (b) SDF/L graph for the SDF graph of (a), (c) a possible mapping/scheduling of graph (a) 59
Figure 4.8 (a) Pseudo-code of an example with a loop structure, (b) SDF/L graph for the code (a) 60
Figure 4.9 (a) A pseudo-code example, (b) SDF/L graph representation of (a), and (c) a homogeneous SDF graph expanded from the SDF/L graph of (b) 62
Figure 4.10 SDF/L graph of k-means clustering 64
Figure 4.11 Architecture of deep neural network used for digits recognition 65
Figure 4.12 SDF/L graph of deep neural network with two hidden layers 67
Figure 4.13 Task graph specification for the internal behavior of a robot, supporting two types of information sharing 71
Figure 4.14 The code templates of a library task and the caller task . . . 71
Figure 4.15 Multicast port and the example code 72
Figure 4.16 Code generation flow in the SeMo framework 74
Figure 4.17 Scheduler code 75

Figure 4.18 Target specific code for thread creation 76
Figure 4.19 Heterogeneous robots work together to find colored paper like motivational example 2 79
Figure 5.1 Specification of behaviors by robot team "Team1" 83
Figure 5.2 Mission scripting language example associated with Figure 5.1 85
Figure 5.3 Mission scripting language example associated with Section 3.1.3 89
Figure 5.4 Dynamic group allocation 91
Figure 5.5 A strategy description example of moving service 94
Figure 5.6 Automatic generation of control task from mission specification and strategy description 96
Figure 5.7 Automatic generation of task graph from mission specification and strategy description 98
Figure 5.8 Quadruped robots share information to avoid falling 105
Figure 5.9 Change of the mission script 106
Figure 5.10 Task graph specification for two different modes of operation 107
Figure 5.11 Robot to follow a specific robot 108
Figure 5.12 Heterogeneous robots work together to find colored paper . . 110
Figure 5.13 Control task code of remote control mission 113
Figure B.1 Specification of behaviors by robot team "MasterTeam" . . . 136
Figure B.2 Task graph specification of the example 148
Figure B.3 Control task specification for the example 149

List of Tables

Table 1.1 The memory requirement comparison between ROS and the SeMo framework for a simple autonomous moving scenario of a robot (Unit: MB) 3
Table 4.1 Target platforms 78
Table 4.2 The memory usage (VSZ/RSS) comparison between ROS and the SeMo framework (Unit: MB) 79
Table 5.1 Framework for programming multiple robots 102
Table 5.2 Comparison with other languages 103
Table 5.3 Lines of the mission scripting, strategy, and task code 108
Table 5.4 Lines of the generated code 109
Table 5.5 Lines of the code 112

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a growing interest in multiple robots performing a single mission collaboratively in various types. On the one hand, in a distributed robot system, robots are usually assigned different tasks to accomplish a common goal collaboratively. For instance, when the military wants to know the enemy's situation, different robots can be used to obtain more accurate information. Alternatively, when trying to find a specific person in a building, different types of robots, such as an unmanned ground vehicle (UGV) and unmanned aerial vehicle (UAV), can work together. On the other hand, in swarm robotics, robot behavior is defined collectively without specification of the role of individual robots. For example, Amazon deploys up to 800 robots simultaneously to pick up and bring a parcel in its warehouse. A mixture of those two different collaboration styles is also possible. In the deployment of cooperating robots, there are two software challenges: how to specify the cooperative mission at the user level and how to program each robot to accomplish the mission. For general robot programming, we need to take into account a wide range of robot platforms, from insect-sized miniature mobile robots [13], [26], [27], that are small in size and have limited energy and computational performance, to humanoid robots [14], [28].

A traditional method to program a robot is to use the robot-specific programming environment provided by the robot manufacturer[29][30][31][32]. The programming environment typically provides device drivers and software libraries to control the robot hardware in a conventional style of programming. Since the robot software depends on the robot hardware platform, it is necessary to re-design the software again whenever the robot hardware platform is changed even for the same behavior specification. It is an inefficient and laborious practice to be overcome. Therefore, this method is not adequate for the behavior specification of cooperating robots that may be heterogeneous.

To increase the reusability of the software over diverse robot hardware platforms, several robotic software platforms have been developed recently for systematic software development. The most popular robot software platform is robot operating system (ROS) [33] that was first introduced ten years ago and has gained wide popularity. It is an open-source framework based on the component-based software design methodology. It provides a set of APIs (application programming interfaces) abstracting the hardware platform and libraries and tools to enable robot programming agnostic of the hardware platform. However, since it assumes a Unix-based operating system such as Linux, it has a disadvantage of heavy resource requirements for miniature robots. Our preliminary experiment with an autonomous moving scenario on a V-Rep simulator [34] environment reveals that the memory requirement of ROS-based software is 5.81 times higher than the software designed by the proposed framework as shown in Table 1.1. Also, ROS programming is not easy for a casual user that has little knowledge of computer programming [35]. Moreover, how to specify a collaborative mission of robots is not addressed in the ROS platform [36], [37].

Two approaches have been proposed for the software development of multiple robots: 1) bottom-up approach and 2) top-down approach. The bottom-up

Table 1.1: The memory requirement comparison between ROS and the SeMo framework for a simple autonomous moving scenario of a robot (Unit: MB)

	ROS	**SeMo**	**Ratio**
Application	73.48	45.37	1.62
ROS framework	190.33	-	-
Total	263.82	45.37	5.81

approach is to program the behavior of individual robots and their interaction with a predefined set of APIs for communication and synchronization between robots. This approach is widely used by extending the existent robot programming environment for distributed robot systems. Since it gives the developer complete control over the design, the developer is exposed to the overwhelming burden of design details such as synchronization and robust programming [38] [39] [40]. The top-down approach, on the other hand, is to abstract multiple robots into groups and specify their behaviors as a single robotic motion. Since it lacks expressive power to fine-tune specific robot behaviors, it is difficult to specify a cooperative mission in which heterogeneous robots behave differently [40]. Thus, it is applicable for swarm robotics.

We identify four requirements for desirable software design methodology of cooperating robots as follows:

(R1) The software design should be independent of the robot hardware platform, similar to ROS.

(R2) The resource requirement should be minimized for miniature mobile robots that have a limited budget of resources, which is not considered in ROS.

(R3) A user that lacks knowledge of the robot hardware and programming languages should be able to specify the mission of robots easily.

(R4) In the mission specification, a collaboration of robots and dynamic behavior

could be expressed intuitively.

(R5) Mission specification supports both distributed robot systems and swarm robotics.

The first two requirements are related to robot behavior programming, and the next three are about robot application programming or mission programming.

In this paper, we propose a novel software development methodology that integrates the top-down and the bottom-up approaches synergistically and satisfies these four requirements, by separating mission specification and robot behavior programming. For mission specification, a novel scripting language is introduced with the expression capability of dynamic mode change and multi-tasking. A mission is specified by a sequence of services that the robots can provide to the user. Since a service is independent of the robot hardware and software, the proposed mission specification method satisfies R3 and R4 requirements. To improve the robustness, scalability, and flexibility of robot collaboration, we include some key features of swarm robotics in the scripting language. We add a team hierarchy, which allows the developer to form a group of robots dynamically in a team. A team of robots may have several groups that perform different services at the same time. Also, a new notion of a service, called *group service*, is introduced, which corresponds to the cooperative mission specification in the top-down approach. Moreover, intra-team communication via broadcasting and local information sharing, which are essential for swarm robotics, are supported in the framework. Thus, the proposed framework enables a casual user to specify various types of cooperative missions for distributed robot systems, swarm robots, and their hybrid, which satisfies R5 requirement.

For robot behavior programming, on the other hand, we use an extended dataflow model for task-level behavior specification of each robot. Dataflow models attract attention for the design and implementation of a parallel application

on a multicore system since they explicitly specify the task-level parallelism of an application. A class of dataflow models, called decidable dataflow models, has restricted execution semantics so that we can analyze the application behavior at compile time to detect critical design errors such as deadlock and buffer overflow, which saves the huge overhead of testing and debugging of a parallel application. Nonetheless, they are not widely used in general, except for a limited set of signal processing and streaming applications, because the expression capability of decidable dataflow models is severely limited. Thus, extensive researches have been performed to enhance the expression capability by expressing the dynamic behavior of an application [25][41], allowing the use of shared resources [42], and so on. The operation of each robot is specified by a task graph that consists of tasks following the pre-defined formal semantics on inter-task communication and scheduling. For each robot, it is assumed that all tasks are programmed and prepared in the task library, which corresponds to the bottom-up approach. The actual robot software is automatically generated from the dataflow model after mapping and scheduling of tasks onto the robot hardware platform is determined. Thanks to its formal semantics, we can estimate the resource requirement and performance at compile-time, which enables us to make the best task mapping and scheduling decisions based on the estimation result. And thanks to automatic generated individual robot program, it relieves the programmer of the aforementioned burden of the bottom-up approach and reduces the risk of manual programming error drastically. Thus, R1 and R2 requirements can be satisfied with the proposed method of robot behavior programming.

Note that there is a significant abstraction gap between the mission specification and task-level behavior specification. To fill the gap, we add another specification layer, called a strategy description. A task or a task subgraph in the task-level specification is abstracted as a service at this layer. Since there may

exist multiple tasks providing the same service, we manage a database that relates a service with the tasks or task subgraphs that provide the service. In the strategy description, additional information is specified to decide which task or task subgraph is selected for a given service request from the mission specification.

The viability of the proposed methodology is validated with experiments with a robot simulator and distributed robot systems. The former demonstrates the added features for swarm robotics, and the other performs a common mission as case studies. In particular, we focus on how to specify the mission and the resultant dataflow specification translated through the strategy layer. To verify the productivity of software development, we compare the number of lines of the generated code.

1.2 Contribution

The contributions of this dissertation can be summarized as follows:

- We propose a novel software development framework for cooperating robots combining the top-down and bottom-up approaches. It separates the service-oriented mission specification and model-based robot behavior programming, so-called *SeMo* framework. It satisfies five requirements in the software design of cooperating robots: software reusability, resource minimization, the easy specification of the mission, the specification of robot collaboration and dynamic behavior, and the support of swarm robotics as well as distributed robot systems.
- A novel scripting language is proposed to specify a mission as a sequence of services so that the user who is ignorant of robot hardware and software programming can use it easily. The scripting language supports dynamic mode change and multi-tasking that are not supported in most existent scripting languages. Also, to support swarm robotics as well as distributed robot systems, we add features for dynamic group formulation, the definition of group services, local information sharing and so on. It improves the scalability, flexibility, and robustness of multiple robots.
- We use an extended dataflow model for the task-level specification of each robot's behavior. For individual robots, we adopt an extended SDF model with a finite state machine (FSM) to express dynamic behavior and propose a novel extension to the SDF model, called SDF/L graph to express and control the loop structure explicitly. The SDF/L graph can express the C-type loop easily that the SDF model cannot specify. For the cooperative operation of multiple robots, we adopt a library task to access global information and support server-client interaction and extend one-to-many communication

adding multicasting port and adopting the knowledge-sharing technique.

- We develop an automatic code generation technique from the model to the actual software code to run on each robot platform. By generating the run-time system and an adaptive resource management module automatically, we improve the productivity of software development and save resource requirements adaptively.

- By abstracting a task or a task subgraph as a service and introducing an additional strategy specification, we generate the task graph specification that corresponds to the mission specification automatically.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows:

- Chapter 2 presents a brief explanation of terminologies and a quick literature review on several topics related to the robot software development frameworks and parallel embedded software development framework.

- Chapter 3 describes the overview of the proposed SeMo framework. It has the refinement process among four levels of abstraction in software development. Also, we introduce several examples to explain our flow.

- Chapter 4 explains model-based task graph specification for individual robots and cooperating robots, respectively. Extended task graph models for dynamic and iterative behavior specification are used for individual robots. Especially as the demand for compute-intensive applications such as vision and machine learning grows, the SDF/L graph is formally defined in Section 4.2.2. For cooperating robots, two types of information sharing, global information shared and local knowledge sharing, are expressed explicitly. Automatic code generation techniques are explained in Section 4.4.

- Chapter 5 focuses on high-level mission specification, explicitly expressing team formation, dynamic mode change of operation, and multitasking in Section 5.1. To improve the robustness, scalability, and flexibility of robot collaboration, we append new features such as team hierarchy, group service, and one-to-many communication in Section 5.1.3. Also, it is refined into extended dataflow graphs, one for each robot, with the help of a strategy description file. Section 5.3 shows how to convert the mission description to the task model.

- Finally, we summary the proposed methodology and discuss possible areas

for future works in Chapter 6.

Chapter 2

Background and Existing Research

This chapter provides basic terminologies and a comprehensive overview of software development methodologies and frameworks for robot and embedded systems. Among many embedded system software development methodologies, we will focus on the special way, HOPES approach, used in this study. The goal of this chapter is not to provide a comprehensive and thorough literature review of the topic represented, but to provide an overview of the topics referenced throughout this dissertation.

2.1 Terminologies

2.1.1 Robot

A robot is a machine that can automatically carry out a complex series of movements. In particular, it can be programmed by a computer [43]. It has begun to be developed to perform dangerous or difficult tasks on behalf of humans, which is resulting in increased productivity. As technology advances, however, its scope of application is gradually expanding. For example, robots can collaborate with workers, improve quality of life, and even interact with people.

According to the International Federation of Robotics (IFR), robots can be classified into three categories [44], [45]: 1) industrial robots, 2) professional service

robots, and 3) personal service robots. Industrial robots are responsible for the automation process in the manufacturing industry. For instance, there may be integrated circuit handlers, dedicated assembly robots, and automated guided vehicles. Service robots perform useful tasks for unspecified individuals except for industrial automation applications. Service robots for personal and private use are used for non-commercial work, usually used by laypeople. Examples are home servant robots, sweeping robots, educational robots, and entertainment robots. And service robots for professional use are used in the commercial or professional task by properly trained operators. Examples are fire-fighting robots, medical operating robots, and guided robots in the military.

Robots can be divided into ground robots, flying robots, and marine robots according to the actual operating space. The most common ground robots can be categorized as crawling, walking, rolling, jumping, and climbing, depending on how they move. Their sizes vary widely, ranging from insect-sized small mobile robots [26], [27] to humanoid robots. Aerial robots are mainly made by imitating birds or flying insects. They are widely used with unmanned control, especially

Industrial Robot	Service Robot	
	Personal Service Robot	Professional Service Robot

Figure 2.1: Robot classification by use, image from [3], [4], [5], [6], [7], [8], [9]

for performing militarily dangerous tasks such as scouting and dropping bombs on behalf of humans. Marine robots are primarily developed for underwater exploration or military reconnaissance [46]. Furthermore, hybrid robots that can be operated on the ground and underwater or on the ground and in the air have been developed. Figure 2.2 shows the classification, where each robot has specialized form and function adapted to the characteristics of the activity area.

In order to operate a robot as a system, it is necessary to integrate components such as structural, driving, control, sensing, and power parts. In addition, it requires several algorithms to perform a mission [47], [48]. For example, path planning, localization, and mapping algorithms are required so as to move autonomously. It is imperative to control actuator or manipulator, which is the most distinguishing feature of the computer. And stabilization algorithm is important as well to control attitude and pose, because a robot with four legs, like a bug, is easy to flip while moving [49]. Recently, intelligent robots, which operate autonomously, are in the spotlight. They are equipped with various kinds of sensors, such as optical, acoustic, vibration, temperature, pressure, and chemical sensor, which are similar to the human five senses, such as visual, auditory, touch, smell, and taste [50]. They can recognize the external environment (perception), determine their status (cognition), and operate autonomously (manipulation). With the recent explosive development of deep learning, recent researches are based on this technology. Especially, it is useful for object recognition, which is to find out the information such as type, size, the orientation of the object based on previously learned knowledge. Besides, human-robot interface (HRI) technology has been developed to understand human emotions and intentions [51], [52]. Such intelligent robots are applied not only to the industrial area but also to civilian service fields such as medical care management and household utilities.

Boston Dynamics reported that BigDog robots shown in Figure 2.2 was de-

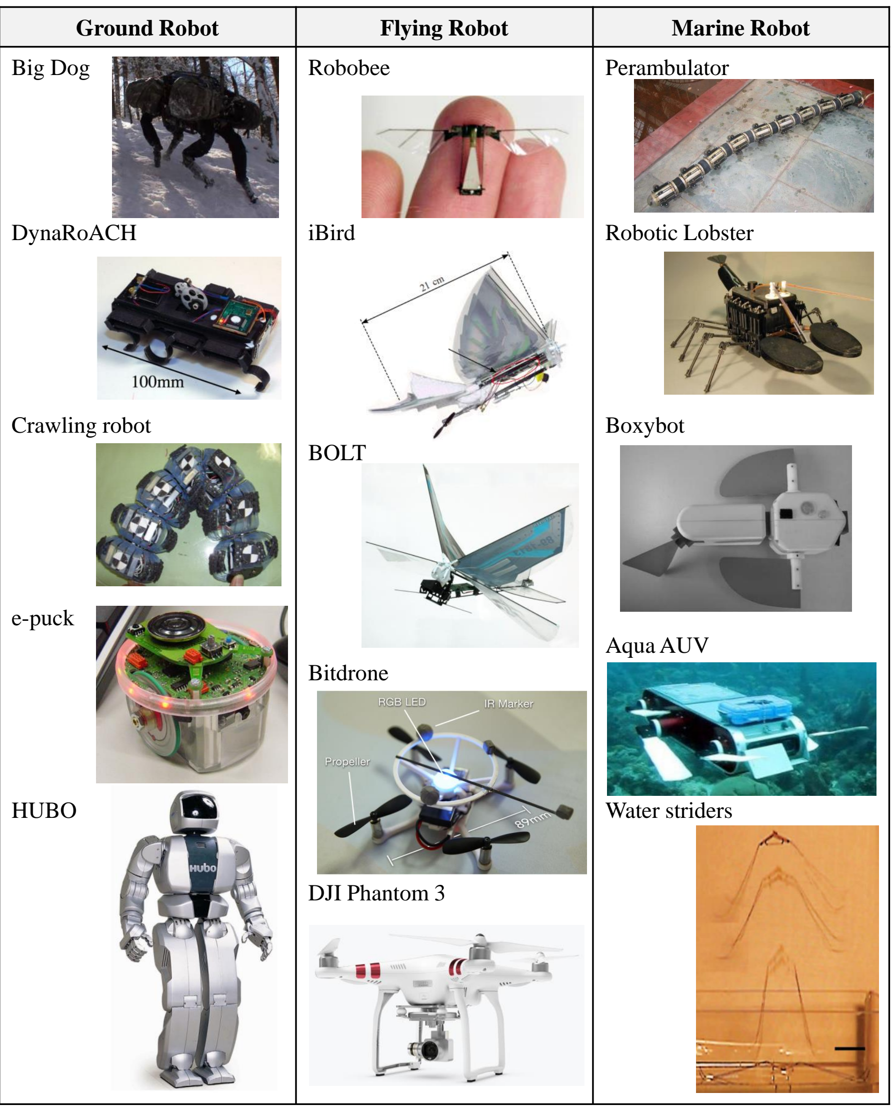


Figure 2.2: Robot classification by movement method, image from Big Dog [10], DynaRoACH [11], crawling robot [12], e-puck [13], HUBO [14], Robobee [15], iBird [16], BOLT [17], Bitdrone[18], DJI Phantom 3 [19], Perambulator [20], Robotic lobster [21], Boxybot [22], Aqua AUV [23], water strider [24]

veloped to go almost anywhere. Because outdoor terrain is too rough for existing wheeled and tracked vehicles to access, BigDog has a variety of locomotion behaviors using about 50 sensors, four legs, and a control system [10]. Seoul National University developed intelligent mobile cognitive robots based on machine learning [53]. Inspired by the human brain process, they combined deep learning models with memory models to overcome restricted paradigms of classical artificial intelligence (AI). Their robot has various perception modules, such as object detection and recognition, human pose estimation, scene description, and speech recognition modules, and action modules such as navigation and following modules. As a result, their robot provides essential social services to the customers in dynamic environments such as a house, hotels, restaurants, and even airports.

2.1.2 Multiple Robots

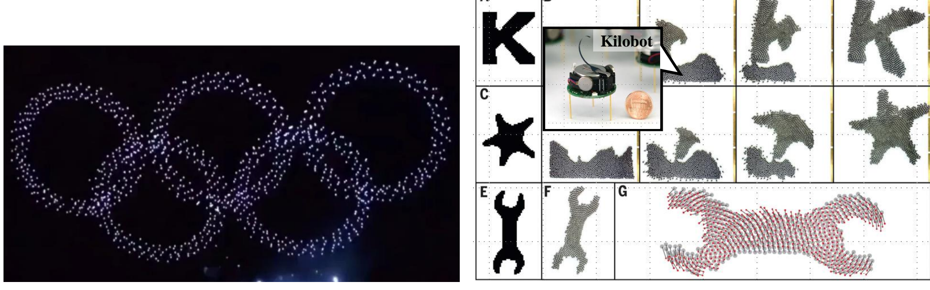
The concept of multiple robots appeared in the early 2000s, but the study has been actively researched and developed in the recent 4-5 years [54]. This is because multiple robots can cooperate effectively. As an example, when looking for survivors in the wreckage of an earthquake, it can be much better to deploy thousands of insect robots. Multiple robots are a set of robots that work in a shared environment to accomplish a given task. Deploying multiple robots efficiently offers several advantages over a single robotic solution, such as distributed control, fault-tolerance, redundancy, support of team members when needed, performing different tasks in parallel, and achieving faster missions [55]. For example, a small mobile robot has a limited battery, which allows it to perform its mission in a limited area within a limited time. Multiple robots can perform the same mission in more areas than a single robot. In general, multiple robots are likely to increase the robustness and reliability of the robotic solution. However, it is difficult to coordinate multiple robots so as to execute complex group tasks efficiently, while

maximizing group performance and optimizing the available resources.

Multiple robots can be composed of homogeneous robots or heterogeneous robots. Also, they can perform a similar task to save time and complement themselves by doing different tasks. For instance, robots can take care of the house by cleaning each room. On the other hand, robots can play soccer in teams [56]. Each robot plays the role of a defender, midfielder, and attacker, playing games to get as many goals as possible.

Inspired by ants, bees, birds, fish, and such social animals, swarm robots are large numbers of homogeneous robots [57], [38]. They allow each robot to move simply and communicate with each other. So one would think each robot's behavior is too trivial to do meaningful. However, their simple function would result in a sophisticated algorithm when a bunch of robots is doing it. In addition, even if a few are lost, the number is so large that it does not affect the entire mission. Therefore, swarm robots are robust, scalable, and flexible. To define three terms here [58], [38], robustness is the ability to cope with the loss of the individual. It improves when the robots are redundant, and there is no leader. And with local sensing and communication, swarm robots perform well in different group sized robots. Therefore, the introduction or removal of the individual robot does not result in a drastic change in the performance of the swarm. Lastly, flexibility is the ability to cope with different environments and tasks, which is related to task allocation.

It is common to see examples of swarm robots. At the opening ceremony of the 2018 Winter Olympics in Pyeongchang, many people admired the swarm of drones covered with LED lights to make patterns of the Olympic rings and its mascot. It consists of 1218 drones, each with built-in LED lights, and weighs just 280 g [59]. One operator can control hundreds of drones at once to create images in the sky. This is because drones are not controlled by hand one by one, but by



(a) A group of drones formed the Olympic rings at the 2018 Winter Olympics opening ceremony, Images from [62]

(b) Self-assembly using up to 1024 kilobots, Images from [63]

Figure 2.3: Examples of swarm robot

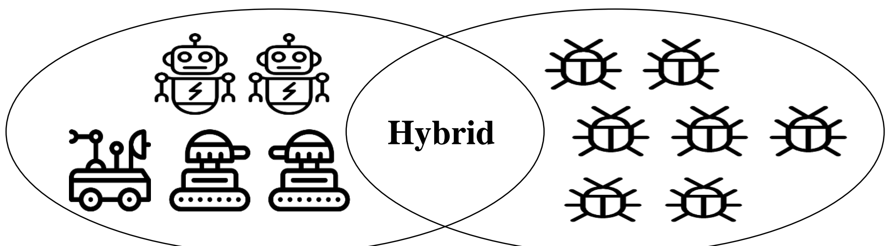
carefully moving in choreographed lines. Also, robots know where they are, so the operator may give commands again if they move abnormally.

Numerous biobots [60] resembling cockroaches were deployed to buildings destroyed by earthquakes or unknown environments, creating a map with wall followings [61]. It is especially beneficial when the global positioning system (GPS) is not available. The robots with sensors were randomly sprayed, and after a while, they kept moving until they found the wall, sending signals to the operator whenever they get closer to each other. When the map of the zone is drawn, the biobots moved to the next adjacent area under the control of the drone corresponding to the controller and continued to draw the map.

The most popular swarm robot is the Kilobot [63], whose size is 1024 units. The coin-sized robot has three rigid legs, two vibration motors for moving straight or turning on a flat surface, and an infrared transmitter and receiver. Robots do not have direct information about their global position, but stationary seed robots know their location and orientation. Other robots can move along the edge and receive messages to calculate their location and gradient values. Using a self-assembly algorithm, they can form a cluster and change shape for different purposes. Thus, we can anticipate kilobots to transport large objects or coordi-

nate the construction of bridges or specific structures like ant colonies [64]. Like this, swarm robots following simple rules and communicating with each other can achieve a specific desired result.

In this paper, we divide the cooperating robots into distributed robots and swarm robots, shown in Figure 2.4. To begin with, we define the distributed robots as multiple robots that move independently of each other. So each robot can do different things taking into account hardware and software constraints. In this way, individual robots can complement one another to perform one task. For instance, Amazon deploys up to 800 robots simultaneously to pick up and bring a parcel in its warehouse [65]. To minimize congestion, the system coordinates the route of every robot. On the contrary, a robot in the swarm robots is autonomous because it has its local sensing and communication capabilities. Thus, swarm robots can work together to tackle a given task like biobots and kilobots.



Distributed Robots

- Each robot can do different things
- Individual robot can perform one task complementarily, taking into account HW constraints
- Robots enable to consider the different environments each robot faces

Swarm Robots

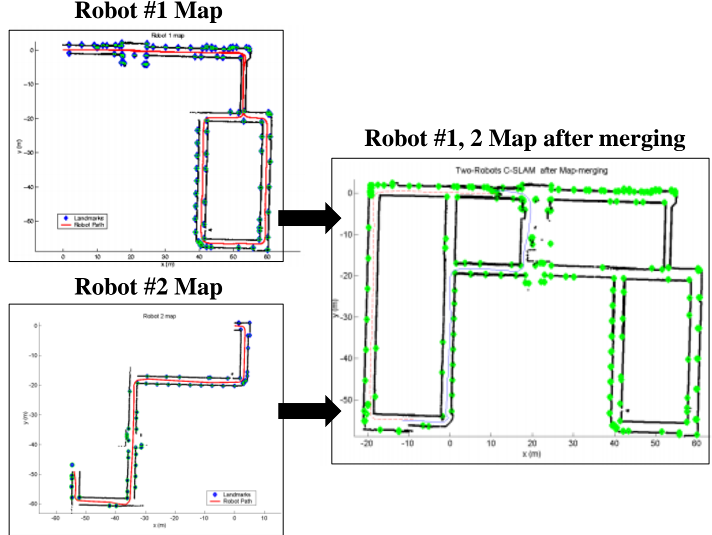
- Robots are autonomous
- Robot's sensing and communication capabilities are local
- Robots may not have access to centralized control and to global knowledge
- Robots cooperate to tackle a given task

Figure 2.4: Cooperating robot

To operate cooperating robots, four skills are required: behavioral control, situational awareness, network, and system integration. First, the control technology can be divided into centralized and distributed control methods. The centralized approach regards the entire robots as a system, controlling all robots from a single control tower to ensure optimality and completeness. However, as the number of robots increases, it requires the amount of computation and the complexity of communication. The distributed approach, meanwhile, is to control each robot as an independent system. Although there are advantages in that it can compensate for the shortcomings of the centralized approach, it is difficult to guarantee the optimality and completeness of the mission. Therefore, it is necessary to identify the characteristics of individual robots that constitute cooperating robots, and to determine what kind of control methods to use for the operation of multiple cooperative robots. In addition, the robot can allocate the role dynamically rather than statically to enhance the robustness. If one robot encounters a problem while performing a mission, other robots should be able to handle it on their behalf so that it does not become a problem. If you want to use a leader-follower structure within robots, you need additional technologies on how to check the leader periodically and determine the leader robot [39], [66].

Communication is indispensable to operate a mission with multiple robots. As though a single robot communicates with the operator, multiple robots may communicate only with the operator, not with each other [67]. However, communication between them is essential to make them more efficient. Communication between robots can be divided into one-to-one and one-to-many or many-to-many communication. One-to-one communication is useful in a stable communication environment, enabling highly reliable. On the other hand, one-to-many or many-to-many communication is useful when robots share information. Like control technology, one robot has information centrally, and the others can query infor-

(a) Robot formation switch from diamond to line, Images from [68]



(b) Multi simultaneous localization and mapping (multi-SLAM), Images from [69]

Figure 2.5: Examples of swarm robot behavior

mation to get it. Additionally, each robot has its information and update the information through communication with nearby robots. Since the method of sharing information may vary depending on the mission scenario, it is necessary to determine the characteristics of individual robots and mission scenarios.

Compared to a single robot, using multiple robots provides more available services. For example, they can move in formation [70], [68], [71], [72], [73] and transport objects larger than robots. A study [74] researches a probabilistic collision avoidance algorithm for navigation among other robots and moving obstacles. In experiments, two quadrotors share the space with two humans and sixteen quadrotors in the simulation verify the coordination approaches. In addition, the

trajectory planning algorithm is developed for swarm navigation [75]. With up to 200 quadrotors, it generates spare roadmaps with possible collisions and creates smooth trajectories in a few minutes.

Moreover, there are various studies about the observation of the environment with multiple sensors. Each micro aerial vehicles (MAVs) estimates its motion and centralized ground station generates its individual map for each MAV and merges maps to create collaborative localization and mapping in unknown indoor and outdoor environments [76], [77].

In this way, multiple robots can be used much more efficiently as hundreds of drones can save time and money than a single helicopter on the disaster scene. Currently, robots are mainly used in industrial areas or in particular service areas, but they are expected to be used much more as the prices go down in the era of the Internet of Things (IoT).

2.1.3 Robot Software

Robot software has a hierarchical structure similar to a general computer, but it is connected to various I/O devices and performs mainly a given task, as shown in Figure 2.6. It consists of four layers: 1) hardware abstraction layer, 2) operating system layer, 3)middleware layer, and 4) application layer.

The bottommost layer, the hardware abstraction layer (HAL), lies between the robotic system hardware and the operating system. It includes the boot loaders, device drivers, and other components. Similar to the basic inputoutput system (BIOS), the boot loader is a program that initializes hardware and prepares the hardware and software environment before the operating system kernel runs. Device drivers provide an integrated software interface for hardware components and peripherals.

The operating system layered on top of the HAL manages hardware and soft-

ware resources and provides standard services such as scheduling and networking. The operating system for robots may not only use general-purpose operating systems such as Linux and Windows, but may also use a specific operating systems such as sensor nodes or real-time operating systems depending on the purpose.

For example, Micro COS III [78] is a real-time operating system with preemptive multitasking. It has the advantage of being able to manage an unlimited number of tasks and be ROMed along with the application code. Its high portability to 8, 16, 32, and 64 bit systems makes it easy to port on many robots. Texas Instrument created uC-OS-based robots and provided the software development environment including libraries, called StellarisWare. Most robot software is preprogrammed in ROM, allowing programmers to focus on application development rather than setup. In addition, VXWorks [79] made by WindRiver is the world's most widely deployed real-time operating system and is widely used in industrial robots.

TinyOS [80] and Contiki [81] are operating systems for sensor networks that is

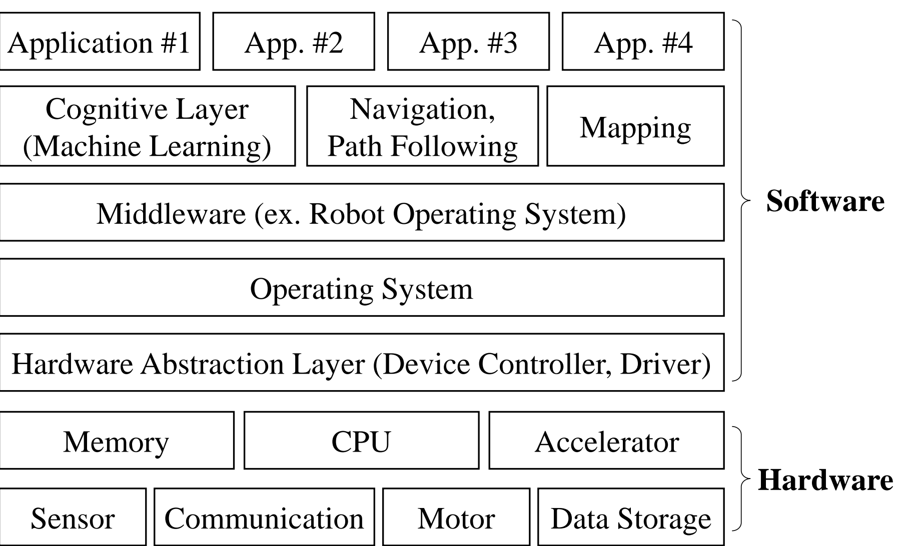


Figure 2.6: Software architecture of robot

lighter than Linux and use an event-based programming model that is convenient for specifying the behavior of robots. Event simulator TOSSIM for TinyOS applications gives the programmer a complete view of the network and makes it easier to develop. However, it does not provide all the components of the actual robot named CotsBots [82]. Contiki focuses on low-power wireless Internet-enabled devices. It also uses protothreads, which are very lightweight and stackless threads that can significantly reduce memory overhead [83].

Besides, the robot software often requires the entire code, including the scheduler code, to be compiled and loaded into the robot without an operating system. In this case, the robot manufacturers typically provide code for controlling the hardware of the robot in the form of a device driver or library or directly support program specification and code generation through its development environment.

Therefore, the operating system of the robot can be different, so this must be taken into account when making the software for the robot. In this thesis, we propose a methodology for developing software for various operating systems.

Robot middleware helps robot builders to simplify the development process, support communications and interoperability, and provide integration with other systems [84]. There are various robot middleware projects such as Player Project [85], Urbi [86], and robot operating system (ROS) [33]. The following section explains this in more detail.

The top-level software is the application layer that implements system functionality. Robot applications sometimes require specific services such as navigation, path following, and cognition. Then it executes additional program such as PyTorch [87] and TensorFlow [88]. Also, the application provides a graphical user interface (GUI) to interact with the user easily. When programmers develop robotic software applications, they use the robot simulators such as Webots [89], MORSE [90], Gazebo [91], and V-Rep [34] to verify their developed technology

before applying it to the actual robot environments. They provide realistic simulation environments that take into account robot-to-robot communication and two- or three-dimensional world modeling. They also support physic engines that consider disturbances to simulate the real world.

Many software systems and frameworks have been proposed to make programming robots easier. I'll explain more in the next section.

2.2 Robot Software Development Frameworks

The existent software development frameworks for robots can be divided into two categories: manufacturer-specific software framework and general-purpose software framework.

2.2.1 Manufacturer-specific Software Framework

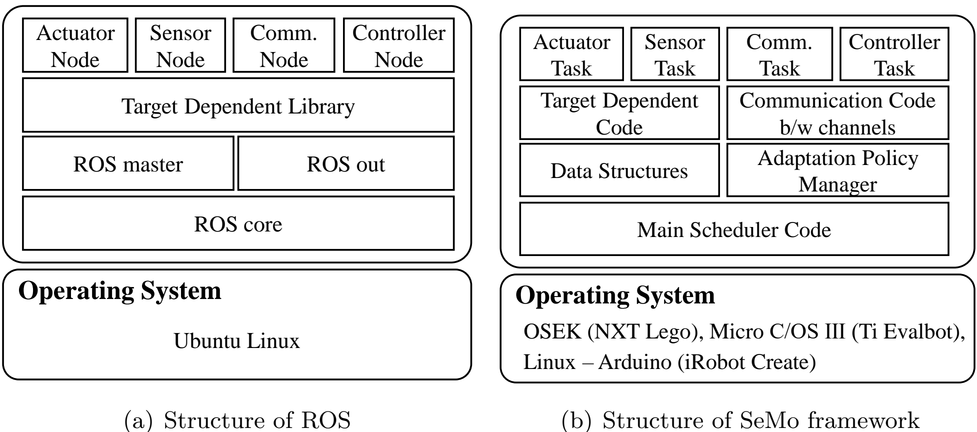
The traditional method to program a robot is to use the robot-specific programming environment that is provided by the robot manufacturer. The programming environment typically provides device drivers and software libraries to control the robot hardware in a conventional style of programming. For instance, the POB Robotics Suite supports the Software Suite, which is designed to allow an easy and quick start [30]. For beginners, the icon programming software is intuitive. A complete development environment allows advanced users to use common languages such as C and Java.

Since the robot software depends on the robot hardware platform, it is necessary to redesign the software again whenever the robot hardware platform is changed even for the same behavior specification. It is an inefficient and laborious practice to be overcome.

2.2.2 General-purpose Software Framework

To increase the reusability of the software over diverse robot hardware platforms, several robotic software platforms have been developed recently for systematic software development.

The ROS (Robot Operating System) [33] is the most popular robot software platform based on the component-based approach, which emphasizes reusability and scalability. It is a virtualization layer between applications and distributed computing resources, providing hardware abstraction, low-level device control,



(a) Structure of ROS (b) Structure of SeMo framework

Figure 2.7: Comparison of the software structure: ROS and the proposed framework

and message-passing between nodes. Nodes are processes that perform computation, corresponding to tasks in the SeMo framework. In ROS, a master node acts as a name server for node-to-node connection and message communication. Each node is initialized by registering itself and the topic to the master node. Then nodes can communicate with each other via publishing/subscribing topics, requesting/responding a service, or sending goals. While ROS is focusing on individual robots, several studies use two or more robots for a cooperative mission[92]. Even though ROS supports various languages such as Python, Java, and C++,

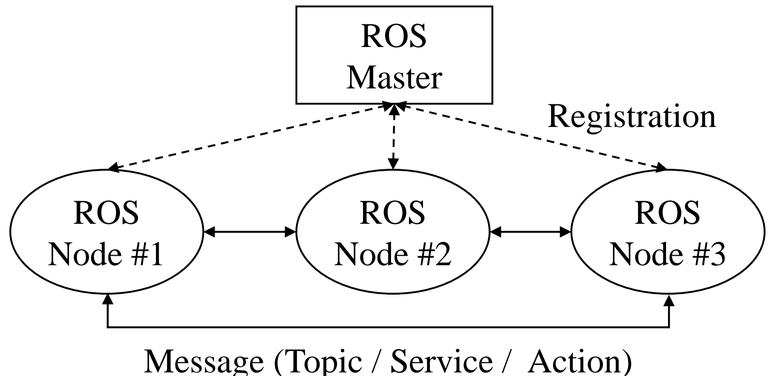


Figure 2.8: Robot operating system

there are no specific APIs defined for a cooperative mission. Hence it is difficult to specify a collaborative mission for robots[36]. Since ROS is based on a central node called *rosmaster* that provides naming and registration services for the rest of the nodes to discover one another, it is known that a robot may get disconnected in multi-robot systems due to unreliable network[37]. Thus, using one master inside each reliable network, which is typically one master per robot, is taken as a solution[93]. There are three significant differences between ROS and our SeMo framework. First, while ROS supports Unix-based operating systems only as illustrated in Figure 2.7 (a), SeMo framework does not assume any specific operating system. As shown in Figure 2.7 (b), we can synthesize kernel codes such as scheduling and adaptive policy manager if necessary. Second, since the operating system performs node scheduling in ROS, it is not possible to estimate the performance or to verify the feasibility of the specified scenario beforehand. On the other hand, in the SeMo framework, we can determine the task schedule considering the performance requirement and resource limitations, thanks to the static analyzability of the formal dataflow model. Thus, the proposed software framework can support miniature mobile robots, which usually have tightly constrained resources. Last but not least, ROS is mainly used by those who have background knowledge of robots and programming so that it is not easy for a casual user to use [35]. Moreover, it is laborious for an expert to specify a mission of cooperating robots in the ROS framework [36]. In contrast, while the proposed framework aims to make both application programming and behavior programming easy.

OPRoS (Open Platform for Robotics Service) [94] and Orocos (Open Robot Control Software) [95] are also component-based frameworks similar to ROS. Like ROS, they do not concern resource limitations, supporting mainly high-level operating systems such as Windows and Linux. While ROS focuses on the specification of algorithms performed on a single robot, OPRoS supports server-client robot

control and management, allowing more flexible specification and implementation of cooperating missions. In the case of Orocos, a scripting language is proposed to help users to write programs and state machines controlling the system, which corresponds to the mission script language in the SeMo framework. It provides a tool to generate C++ code from the script language, assuming that the code is run on a single processor sequentially. On the contrary, the mission specification is translated into a task graph model that can be mapped to a multicore platform in the SeMo framework.

MissionLab [96] is a robot software framework that consists of different levels of abstraction, which is similar to the proposed framework. It is designed to support cooperating robots. For each robot, it allows the user to specify the characteristics of the robot and the robot behavior with the state machine in the GUI (graphical user interface) environment. This high-level specification is refined to CDL (configuration description language), CNL (configuration network language), CMDL (command description language) and finally to C/C++ target code that is run on each robot, adding more information at each refinement step. Their high-level specification is not a scripting language, and its expressing capability is too restricted to specify complicated missions. For instance, CMDL does not support iterative and conditional constructs but supports only sequential control constructs. Moreover, it is not designed to support miniature mobile robots, targeting power robots that run the Linux operating system.

Some approaches use a popular scripting language to program the robot application since a scripting language is relatively easy to learn. One example is pyro [97] that uses the Python language. TMkit [98] belongs to this category since the domain-specific mapping from scenes to task state and task operators to motion planning problems can be written in Python or Common Lisp. Since the scripting language is independent of the hardware platform, it can be categorized

as a general-purpose framework as long as the Python interpreter is supported for the hardware platform. In these approaches, however, there is no distinction between the mission specification and the behavior specification unlike ours. We believe that a Python language is still hard for a casual user to use for the mission specification of cooperating robots.

Aseba [99] is an event-based software architecture that provides two specification methods for beginners and intermediate users. For beginners, the behavior of a robot can be specified by event-action pairs in a GUI environment. On the other hand, intermediate users can directly program the robot codes by using Aseba language that is an imperative language similar to PASCAL. Even though it considers beginners who lack knowledge of programming, this approach is different from ours that does not sacrifice the expression capability of beginners with a novel notion of abstraction hierarchy. DRONA [100] uses a new programming language, called P [101] for asynchronous event-driven systems. It describes the distributed mobile robots with state-machines and generates the robot code automatically from the high-level specification. Recent work in [102] is similar to our framework in that they generate robot code from a high-level specification that a casual user may learn easily. They propose the natural language interface of the LTL (linear temporal logic), called Structured English[103]. The LTL specification is translated to FSM, and finally to a ROS node. While it is similar to control task synthesis from mission specification in our framework, it does not consider computation tasks and no cooperating missions.

Karma[104] is a framework for programming and managing swarms of micro-aerial vehicles (MAVs) based on a centralized hive-drone model. A drone is an individual MAV that performs the specified commands without in-field communication. And the hive, as the central coordinator, orchestrates the drone and executes a given mission. The user specifies the cooperative mission by considering

many drones as a swarm rather than individuals. The centralized hive maps the drones by location, and each drone works in that area and returns to the hive to share information, which leads to long information latency. Although this method has the advantage of easy decision making thanks to the centralized hive and simplified programming of the MAV, it is not applicable for general distributed robot systems.

Another category of the software development framework usually aims to support robots that do not use high-level operating systems such as Linux and Windows, for instance, a light-weight operating system for a sensor node or a real-time operating system. For example, Texas Instruments provides StellarisWare as a software development environment for the robot based on Micro C/OS Ⅲ. In the case of Mindstorms NXT Lego robot based on OSEK, RobotC [105] is used as the programming language. Since it does not fully support C standard, care should be taken to consider its characteristics and limitations in programming.

In summary, there is no general-purpose robot software development framework that supports diverse operating systems and hardware platforms and allows a casual user to make applications of the cooperating robots, which motivates the invention of the new design methodology.

2.3 Parallel Embedded Software Development Framework

An embedded system is an electrical system that performs a dedicated function, often with real-time computing constraints. In other words, it is designed for specific tasks rather than a general-purpose computer for multiple tasks. And it is embedded as part of a complete device, including hardware and mechanical parts, and acts as the brain for a system that requires control [106]. Examples are smartphones, cameras, refrigerators, industrial robots, cars and so on.

As more processors are integrated into a single system and the application range of embedded systems continues to grow and become smarter, developing parallel embedded software has become more difficult. There are five requirements for embedded software development. As the complexity of embedded systems increase, more processing elements are involved. Therefore, multiprocessor systems-on-chip (MPSoC) requires parallel processing. And building real-time embedded software is emphasized because both value and time affect the physical outputs of embedded systems for interfacing with the real world. Throughput requirements are critical when the application runs iteratively with the input data stream, whereas latency requirements are important to control the application. In addition, embedded systems may have strict restrictions on memory size and power consumption. Moreover, custom hardware and software interface designs are often required. Finally, correctness and performance estimation are fundamental [107], because debugging a program at runtime is not easy.

For parallel embedded programming, there are several approaches to develop parallel embedded software for multi-core systems [108]. With a simple example shown in Figure 2.9, we compare the four approaches: 1) compiler-based approaches, 2) language-extension approaches with annotations, 3) language-

extension approaches with application programming interfaces (APIs), and 4) model-based approaches.

The compiler-based approach looks like a dream methodology. This is because traditional sequential languages like C are used for initial specification without modification, as shown in Figure 2.9 (a). The compiler then automatically parallelizes the application. The key technique is to find parallel regions of the code and check their data dependencies. After finding the data-parallel regions, the partitionermapper converts each parallel region into a set of concurrent tasks and

```
A()
B()
for ( i = 0; i < 99; i ++)
      C()
D()
```

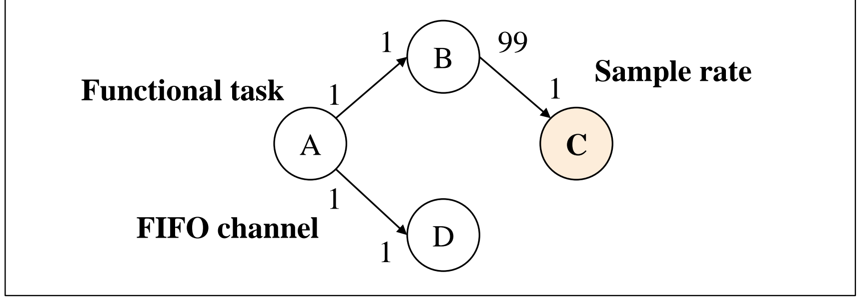
(a) Compiler-based approach

```
A()
B()
#pragma omp parallel for
for ( i = 0; i < 99; i ++)
      C()
D()
```

(b) Language-based approach (annotation)

```
A()
B()
kernel = clCreateKernel(progC, "C");
work_size[0] = 99;
range = clCreateNDRangeContainer(..., work_size, ...);
clExecuteKernel (..., kernel, ..., range, ...)
D()
```

(c) Language-based approach (API)



(d) Model-based approach

Figure 2.9: Design methods of parallel embedded software

maps them to multiple processors. This approach eliminates the burden of parallel programming for application programmers. However, the compilers support a limited set of applications because functional level parallelism cannot be easily detected.

Since it is not easy to extract parallelism from sequential code, programmers provide parallelism information such as where and how to parallelize the code. Although the programmers can program the code manually using threaded libraries, additional annotations and application programming interfaces (APIs) are widely used in the industry and academia [109]. In the Figure 2.9 (b), sequential code with high-level compiler directives specifies potential parallel regions. This allows the compiler to concentrate on exploiting the specified parallelism. The language extensions with annotations have the merit of reducing the amount of annotation overhead and the burden of the compiler. Nevertheless, it is not easy for this approach to control low-levels for optimization only with the annotations.

For low-level control, programmers use APIs to develop parallel programs. Compared with the annotation-based language extension, the APIs need to discover the parallel regions, distribute the code and data to the processors, and restructure the code using the APIs. This means that the programmer has a lot of work to do, as depicted in Figure 2.9 (c). Nonetheless, it is the most widely used approach, thanks to the low level of control.

An abstracted model simplifies the application behavior, decoupling computation and communication. In model-based design, a designer decides on an apposite model that explicitly represents the features of the application behavior. Thanks to the model, analysis and verification can be applied before implementation. It also increases productivity with automatic code synthesis based on models. In the example of the Figure 2.9 (d), nodes represent functions that can be executed when input data are available and arcs or edges represent first-in-first-out

(FIFO) queue that can expose dependencies between nodes. The number in each arc stands for the number of samples read or written in the arc per node execution. Thus, it explains that the node B and D are independent of each other and the node C should run 99 times for each execution of node B. The models of computation naturally express the available parallelism even if most programmers are not familiar with the computational model. With the use of a model of computation, a Y-chart approach is common. It is to explicitly separate application and architecture specifications [110]. A separate mapping specification describes how to execute the application spatially (binding) and temporally (scheduling) on the architecture. Design space exploration (DSE) is then performed by iteratively analyzing and optimizing the application, the structure of the underlying (hardware) architecture, and candidate mappings, as shown in Figure 2.10.

Formal models, including semantic restrictions, describe the behavior of the system at a high-level abstraction [111]. They have several advantages in software design. First, formal specifications can help to discover ambiguities, omissions,

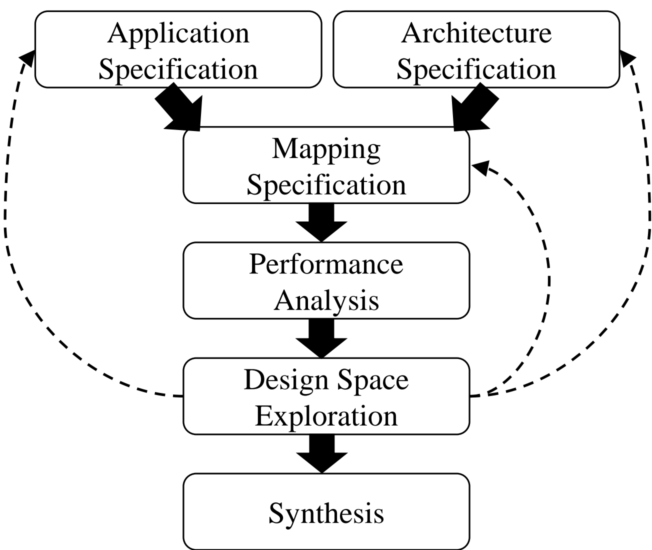


Figure 2.10: Y-chart approach for designing MPSoC

and contradictions early. Second, formal models can carry out formal verification, which determines decidability and complexity bounds as well as estimates the resource requirement and real-time performance at compile time. Simulation cannot guarantee the correctness of the design, so in recent years, researches focus on formal verification and model refinement for ”correct-design-by-construction”. Last but not least, we can synthesize the error-free target code. Kahn process network (KPN) model [112] and synchronous dataflow (SDF) model [113] are examples of formal models. In the KPN model, a node represents autonomous thread, and arc represents communication. A node communicates asynchronously with each other through unlimited FIFO channels, allowing the model to express task parallelism and pipelining naturally. The blocking read makes it deterministic, and non-blocking write enables parallel execution. Since it is determinate [110], the results do not depend on the execution order of processes, which is good for debugging. However, it cannot express asynchronous inputs. Like the KPN model, the existent formal models have severe restrictions in expression capability. There are several extensions to overcome these limitations. This paper will describe the SDF model and its extensions in detail in Section 4.1.

2.3.1 HOPES Approach

The proposed methodology is based on the HOPES (Hope of Parallel Embedded Software) approach [114] that supports parallel embedded software design, from the behavior specification to code synthesis. Figure 2.11 shows the overall design flow of HOPES. Although not included in the figure, there are steps for static performance estimation and design space exploration (DSE). While other environments focus on the design of hardware and software modules and static analysis, the HOPES framework separates the design and implementation of embedded software and places more emphasis on implementation. The formal-model

based design represents available parallelism and independence of hardware. By keeping the semantics of formal models, the implemented software is free from errors that programmers make through automatic code generation.

The HOPES approach introduces a novel concept of software architecture for heterogeneous multiprocessor embedded systems, called universal execution model (UEM) [115]. Figure 2.12 illustrates the vertical software structure based on the UEM. The UEM sits on top of the operating system layer to hide the low-level details of the architecture from the application programmers. The UEM layer consists of 1) execution engine, 2) UEM tasks, and 3) a set of application programming interfaces (APIs). The engine acts as middleware. It runs the UEM tasks on the target architecture. In the middle, a set of UEM tasks is generated from the application. Thanks to the APIs, the application programmer can design an embedded software on top of UEM without knowing the actual hardware platform.

Among many formal models, the synchronous dataflow (SDF) model is chosen as a base model in the HOPES framework. In the SDF model [113], an application is specified as a dataflow graph, where a node represents a functional task, and an edge is a FIFO channel between two adjacent tasks. The number of samples

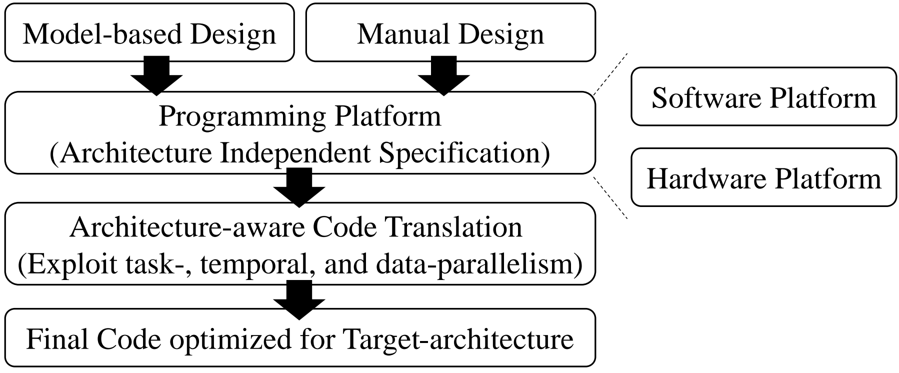


Figure 2.11: HOPES Design flow

consumed or produced at the input or output port is known a priori and is fixed at run-time per task execution. This number is called the sample rate of the relevant port. A task can only be run if the number of samples accumulated on each input channel is no less than the specified sample rate of the input port. These characteristics of the SDF model enable us to determine when and where to execute the tasks at compile time. Nevertheless, the SDF model has limited expressive power, so there are several extensions to existent models.

The UEM model adopts a hierarchical composition of different models of computation to represent the system behavior, as depicted in Figure 2.13. At the top level, the UEM model uses process networks to expresses the system behavior. If an application can be specified by the extended SDF graph, the application is encapsulated as a super node with an extended SDF graph at the bottom level. Note that the top-level process network and the extended SDF model themselves can be specified in a hierarchical fashion. The control process defined in the dynamic behavior of each application is specified formally by the finite state machine (FSM).

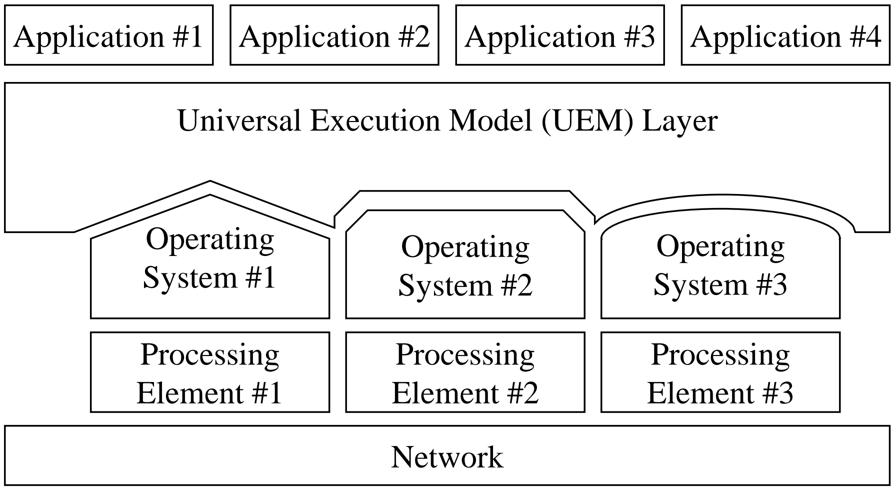


Figure 2.12: Software structure based on the universal execution model (UEM)

The task graph consists of UEM tasks and channels, as shown in Figure 2.14. Following the dataflow semantics, a task communicates with other tasks through connected channels through ports. There are two types of UEM tasks, time-driven and data-driven, depending on the triggering condition of the task. The source task without any input channel is specified as time-driven for the periodic invocation of the task graph. A pre-defined period is provided as a parameter. A data-driven task, on the other hand, is triggered when data samples arrive at the input ports. It is assumed that the input channels of data-driven tasks are the FIFO queues. The task graph can naturally specify data dependencies between tasks.

The task code template is shown in Figure 2.14. The task code consists of three segments defined by three keywords, TASK_INIT, TASK_GO,

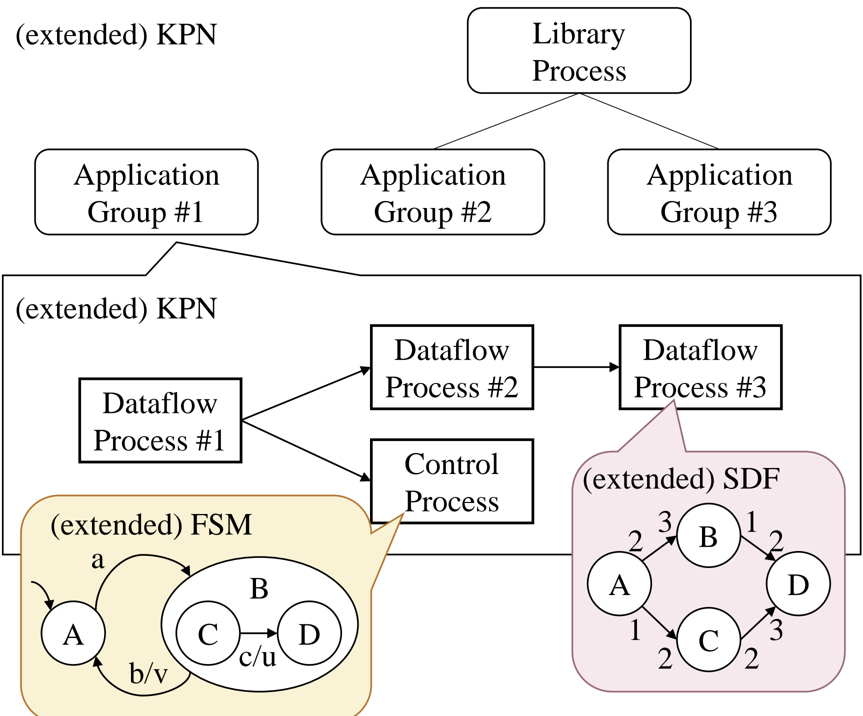


Figure 2.13: Hierarchical software architecture of the universal execution model (UEM)

and TASK_WRAPUP, depending on when the code segment is executed. The TASK_INIT section is executed when the task is initialized, and the TASK_WRAPUP section is executed just before termination. The TASK_GO function is the main body that repeats until the task is ended. A task can read and write to connected channels using generic APIs: *UFPort_ReadFromQueue* and *UFPort_WriteToQueue*. Note that the code template does not contain any platform-specific code. The *UFPort_Initialize* API in the TASK_INIT function initializes the ports used to read or write data in the TASK_GO function. The *UFPort_ReadFromQueue* API performs blocking read operation on the associated input port, while the *UFPort_WriteToQueue* API performs a non-blocking write operation on the associated output port. The *UFPort_GetNumOfAvailableData* API checks for the presence of the data on the input FIFO channel. That is, if there is no data, it blocks the *UFPort_ReadFromQueue* API but does not block *UFPort_GetNumOfAvailableData* API.

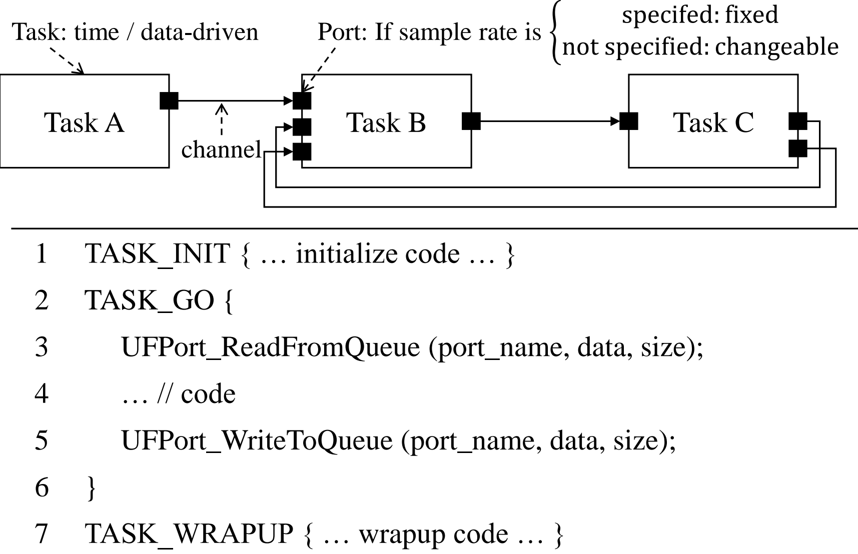


Figure 2.14: A task graph and the code template of a task

The UEM assumes that there is a hidden supervisor who manages the tasks. It defines a set of services that a task can request to the supervisor using a special API. For example, a task can ask for the supervisor to set the timer and to check the timer alarmed; int timer_id = *UFTimer_Set* (10, ”S”), int timeout = *UFTimer_GetAlarmed* (timer_id). The more services such as running or suspending a specific task will be explained in Section 4.1.

Chapter 3

Overview of the SeMo Framework

SeMo is a software development framework that separates high-level mission specification and robot behavior programming. The overall flow of the proposed software development methodology is shown in Figure 3.1, which can be understood as the refinement process among four levels of abstraction in software development.

The first step is *mission specification*, which corresponds to the highest level of abstraction. A user can express the mission of cooperating robots with a script program that is written in a new scripting language proposed in this work. In a script program, robots are grouped into teams that perform the same sequence of services. Since a team may consist of heterogeneous robots, the scripting language needs to be independent of hardware platforms. A service is an abstracted function that each robot can perform. The framework includes a script editor to aid the user for mission specification, as shown in Figure 3.2. The values and services of available robots are listed on the left side, and the program template is provided in the central window. The bottom window is the console window, and the help message is displayed on the right side.

The second step is *strategy description* that defines the second level of abstraction that tells more information on how to perform services. Suppose that

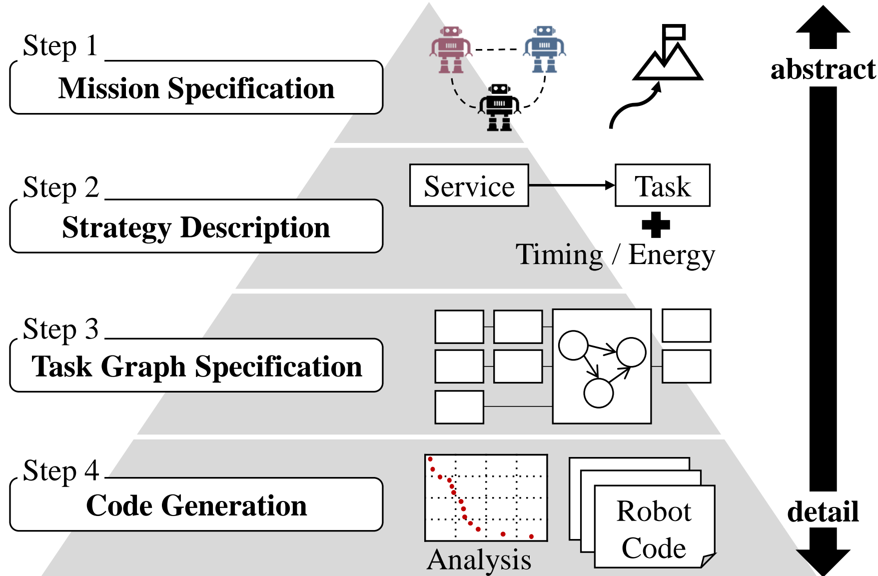


Figure 3.1: Overview of our proposed SeMo framework

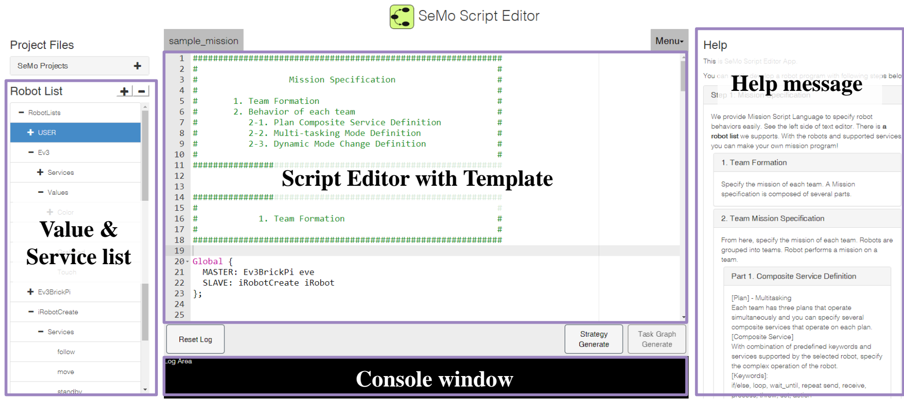


Figure 3.2: A captured screen of the SeMo script editor

we want to let a robot move from one position to another autonomously. In the mission specification, ”move” is specified as a service that the robot should perform. There may exist several algorithms of autonomous moving depending on various conditions and requirements such as speed and energy consumption. For service refinement, we describe conditions and requirements to select the proper algorithm for each service. To this end, the strategy description layer is introduced

in the proposed methodology, and it is written in an XML file. In addition, non-functional requirements, such as execution time and power budget, and special execution policies can be added in step, which will be explained in Section 5.2.

The next level of abstraction is *task graph specification* that depicts the internal behavior of each robot to perform the services. A task graph consists of tasks that will be scheduled and executed by the operating system of a robot. Unlike mission specification, we assume that a task is coded by an expert programmer and abstracted as a service function that a robot can perform. Recently the request for compute-intensive services such as vision and machine learning grows for intelligent robots. Then, a compute-intensive service can be specified by a task subgraph that can be mapped to multiple processors for parallel processing. In this case, the task subgraph as a whole is encapsulated as a service and registered in the database. A sequence of services depicted in the mission script at the highest level of abstraction is automatically translated into a task graph based on the information given in the strategy description step.

For the static analysis and performance estimation of robot behavior, we adopt a formal task graph model, called synchronous dataflow (SDF) model [113] for task graph specification. In the SDF model, inter-task communication is made by the FIFO communication channel between two tasks. And the number of samples consumed from an input channel or produced to an output channel is fixed at runtime for each execution of a task. A task becomes executable only when the number of samples accumulated on each input channel is no less than a specified number, called the *sample rate* of the input port. This restriction of fixed sample rates enables us to make the task scheduling decision at compile-time and estimate the performance and resource requirements based on the mapping and scheduling decision. However, it suffers from limited expression capability, especially for dynamic behavior specification and shared resource management. Thus,

the proposed methodology uses an extended SDF model with a finite state machine (FSM) to express dynamic behavior [116] and with shared resources for the cooperative operation of multiple robots, which will be explained in Chapter 4.

The bottom level of abstraction is the software code running on the robot hardware platform. We generate the target code automatically from the task graph specification after mapping and scheduling decision of tasks is made. With the same task graph, we generate different software codes depending on the robot hardware platform and operating system. The burden of making a code generator for a new robot platform is analogous to that of making a compiler for a new processor. Once a code generator is built, we can reuse the software that is designed for other robot platforms. As will be explained later, we can add software modules without the intervention of the user. For instance, we can add a self-adaptive resource manager to adjust the task scheduling and execution policy depending on the mode of operation in order to minimize the energy consumption for miniature mobile robots.

3.1 Motivational Examples

To explain the proposed methodology, we devise several cooperation scenarios for multiple robots.

3.1.1 Example 1: Robots following specific object

A team of robots has a mission to follow a specific object. The robots run in the remote control mode, where their movement is controlled by an operator manually. If the distance between a robot and the object is close enough, they follow the object automatically.

3.1.2 Example 2: Scouting Mission With Multiple Heterogeneous Robots (Simple Version)

Multiple heterogeneous robots move to a specific destination and then cooperate to find all colored papers in the bounded region like a treasure-hunting game. To find all colored papers in the bounded area, each robot moves straight and changes its direction periodically. If it detects a black line, it stops and turns back not to get out of the bounded zone. If it detects a colored paper that has not been found yet, it notifies the color to the master robot. Otherwise, it ignores the colored paper and keeps going. When all colored papers are found, each robot spins clock-wise and stops there. The global information is the set of colored papers that have been found so far. One of the robots plays the role of the master of this mission and has global information.

3.1.3 Example 3: Scouting Mission With Multiple Heterogeneous Robots (Complicated Version)

This example is a slightly more complicated situation based on the previous example. Similar to the previous example, the cooperative mission is for heteroge-

neous robots to scout a specific area. After they all move to a specific destination, a group of robots works together to find colored papers within a particular area, and the other robots watch around the area to detect any danger. When any danger is detected, they signal to the searching robots to hide. When they are considered safe, they continue searching for papers again. The robots share the information of the colored papers they have found with each other, and that information is periodically reported to the user.

Chapter 4

Robot Behavior Programming

In the SeMo framework, the internal behavior of each robot is specified by a set of task graphs following the dataflow models of computation. To adopt the formal model for software design, the software code will be correct by construction.

What features should be reflected in the formal model to describe the robot behavior? I think the three specifications are required. Robots change their behavior in response to events so that we can specify their dynamic behaviors. Moreover, the computation of the robotic application continues to increase. In particular, deep learning algorithms are commonly used for perception and cognition. Since deep learning algorithms contains a lot of loop structures that usually spend most of their execution time, loop structures have been the main object of parallelization for accelerating the application. Even though dataflow models express the task parallelism explicitly, they do not express the loop structures explicitly. So we need to express loop structures efficiently. Lastly, we can express shared information for multiple robots. For these reasons, we introduce robot behavior for individual robots and cooperating robots in order. Notably, some works are adopted in the extended models, and some works are designed for robotic operations. Automatic code generation techniques are explained, too.

4.1 Related works

Dataflow models attract attention for the design and implementation of a parallel application on a multicore system since they explicitly specify the task-level parallelism of an application. A class of dataflow models, called decidable dataflow models, has restricted execution semantics. So we can analyze the application behavior at compile time to detect critical design errors such as deadlock and buffer overflow, which saves the considerable overhead of testing and debugging of a parallel application.

We first review the synchronous dataflow (SDF) model and describe the extensions to the SDF model.

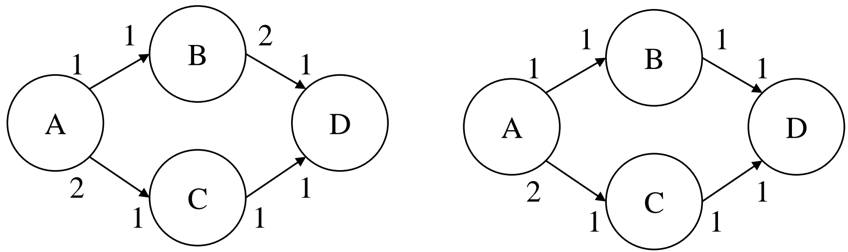
4.1.1 Synchronous Dataflow (SDF) Model

The synchronous dataflow (SDF) model [113] is a pioneering decidable dataflow model widely used for digital signal processing and computation-oriented algorithm specification. In the SDF model, an application is specified with an SDF graph where a node represents a functional task, called a task in this paper, and an edge represents a FIFO channel between two adjacent tasks. Figure 4.1 (a) shows an example of SDF graph. The number of samples consumed or produced from an input or to an output port is known a priori and fixed at run-time per task execution. This number is called the sample rate of the associated port, which is annotated on edge. The sample rate is greater than 1. When it is 1, no separate notation is used. If all rates are 1, it is called homogeneous dataflow. In the SDF model, a task becomes executable only when no input port has fewer data samples than the specified sample rate of the input port. We can compare the input and output sample rates to determine the relative execution rates. For example, the execution rate of task D should be twice that of task B in Figure 4.1 (a). This constraint can be formulated as the equation 4.1, called balance equation.

$$\gamma(A) \times produce(A) = \gamma(B) \times consume(B) \tag{4.1}$$

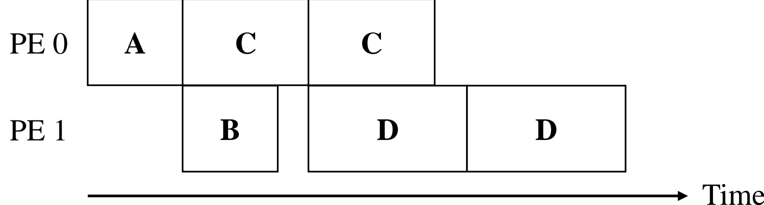
where $\gamma(A)$ means repetition counts of task A. In this example, task B produces (produce(B) = 2) sample and task D consumes (consume(D) = 1) sample per execution. Then $\gamma(B)$ and $\gamma(D)$ are 1 and 2, respectively. If there is a positive integer solution to the balance equations, an SDF graph is said to be consistent. Otherwise, the graph is called the sample rate inconsistent. Figure 4.1 (b) is an example of an SDF graph that has a buffer overflow error.

These characteristics of the SDF model enable us to analyze the application behavior at compile time. We can make a *scheduling* decision at compile time determining where and when to execute the tasks. Figure 4.1 (c) is one of valid periodic schedules of SDF graph (Figure 4.1 (a)). By mapping tasks onto two processing elements, we can get this schedule. Using a static schedule, we can detect



(a) An example of SDF graph

(b) An example of SDF graph that has a buffer overflow error



(c) One possible mapping and scheduling result of the SDF graph onto two processing elements

Figure 4.1: An example of SDF graph

critical errors such as buffer overflow and deadlock and estimate the performance and resource requirement.

Nonetheless, SDF model is not widely used in general, except for a limited set of signal processing and streaming applications, because the expression capability is severely limited. Thus, extensive researches have been performed to enhance the expression capability by expressing the dynamic behavior of an application [25][41], allowing the use of shared resources [42], expressing the loop structure which usually takes most of its execution time of an application [117], and so on. The following sections explain the extensions of the SDF model.

4.1.2 Extensions of SDF model to Dynamic Behavior

There are two types of dynamism: 1) inter-application dynamism, and 2) intra-application dynamism. To specify the dynamic behavior of a system, the similar researches, [116] and [25], are devised.

First, a set of the application running concurrently may change. In [116], dynamic behavior is specified as a control task that manages the execution of applications. Inside the control task, the finite state machine (FSM) is used to express the change of execution. Figure 4.2 shows the relationship between the computational task and control task. The control task can run, call, stop, suspend, or resume a task with UFControl_RunTask, UFControl_CallTask, UFControl_StopTask, UFControl_SuspendTask, UFControl_ResumeTask directives in HOPES framework.

An application may have multiple modes of operation. Or an application may take different execution times of tasks. To express intra-application dynamism, the dynamic behavior of an application can be expressed by defining each scenario of execution with a different SDF graph in the scenario-aware dataflow (SADF) model [25]. In [116] similar to the SADF model, an application consists of a finite number of operation modes, each of which is specified by an SDF graph. The finite

state machine model specifies the mode transition.

In this paper, our model is based on [116]. At the top level, there are a control task and computational tasks. A computational task may contain a subgraph inside, which makes a hierarchical graph. And it may have a finite number of

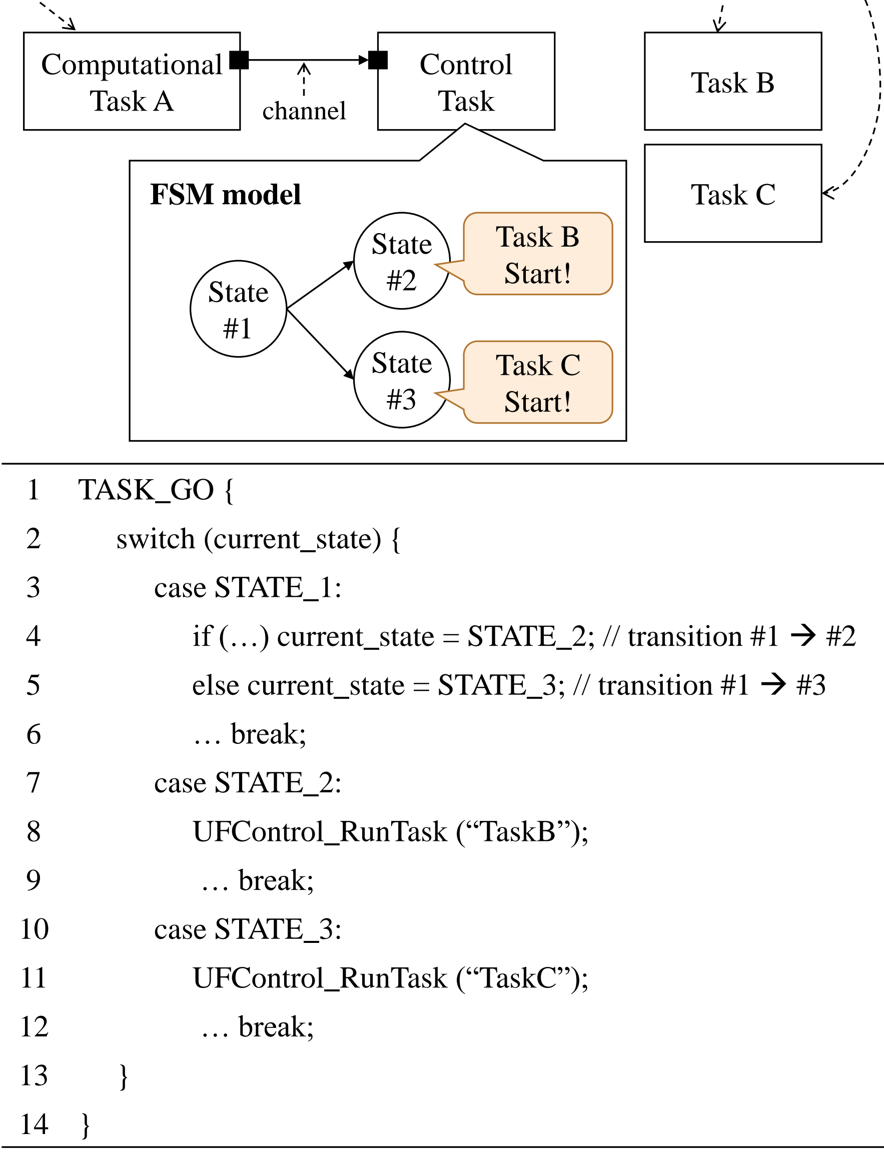


Figure 4.2: A control task and its example code

operation modes, and each mode is specified by an SDF graph.

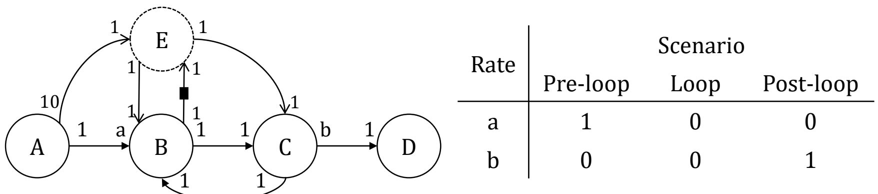
4.1.3 Extensions of SDF model to Iterative Behavior

In this section, we review the related work on the SDF extensions, which enhance the expression capability to support iterative applications.

Cyclo-static dataflow (CSDF) [117] and fractional rate dataflow(FRDF) [118] are decidable dataflow models to specify the periodic sample rate change. However, they cannot specify that the sample rate change is not periodic. Several SDF extensions have been proposed to express varying sample rates.

In the scenario-aware dataflow (SADF) model [25], the dynamic behavior of an application can be expressed by defining each scenario of execution with a different SDF graph. Figure 4.3 shows a SADF specification of the case when task B and C are repeated until a specific condition is satisfied. It has a detector node, E, that varies the sample rates of tasks B and C by sending control tokens via control channels. In the $Pre-loop$ scenario, task B receives a sample from task A and executes one iteration of the loop (B-C). In the *Loop* scenario, loop (B-C) can be executed without requiring any input from task A. In the $Post-loop$ scenario, task C sends the output to task D. In [116], behavior change is expressed by a separate FSM graph that controls the SDF graph.

Parameterized synchronous dataflow (PSDF) [41] specifies the dynamic behavior by defining a subgraph that can be configured dynamically by a set of



Rate	Scenario		
	Pre-loop	Loop	Post-loop
a	1	0	0
b	0	0	1

Figure 4.3: SADF specification of the example of Fig. 4.8 where E is the detector node. Symbols in the figure are the same as [25].

parameters. It requires two additional subgraphs, *init* and *subinit* graphs, to control the behavior of the *body* graph. Parameterized and interfaced dataflow meta-model (PiMM)[119] enhances the expression capability further by defining both hierarchy interfaces and parameterization. Parameterization makes it possible to reconfigure the production and consumption token rates at runtime. The PiMM model can specify a loop structure with varying iteration using a special actor, called configuration actor, and a special data FIFO, called a round buffer.

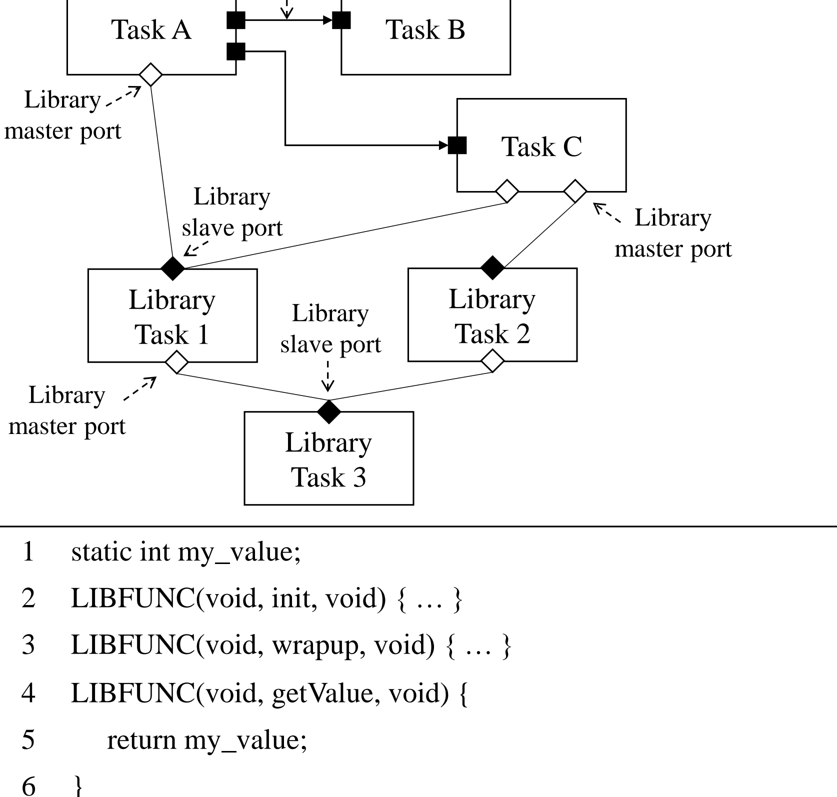
Hierarchical modeling of a loop structure has been proposed in [120] where the template of a dynamic structure is predefined as a subgraph. A loop with varying iteration can be expressed with a $do-while$ loop structure in their work. It is not an extended SDF graph, however, but a restricted model of dynamic data flow (DDF) that allows the use of non-SDF type nodes. Their approach uses the graph hierarchy to distinguish two different models of computations, SDF and DDF. A similar approach can be found in synchronous piggybacked dataflow (SPDF)[121] where the template of a dynamic structure is also defined in their model.

In summary, the SDF model and its extensions focus on expressing task-level parallelism explicitly.

4.1.4 Extensions of SDF model to Shared Resource Management

Dataflow models are good at explicitly exposing the potential parallelism, and various analysis methods and design tools have been developed. However, due to strict semantics, shared variables cannot be used among tasks. To this end, the model is extended with a special type of the task, library task, adopting the library task that was first introduced in [42].

A library task is a shareable and mappable object that defines a set of function




```
1  static int my_value;
2  LIBFUNC(void, init, void) { … }
3  LIBFUNC(void, wrapup, void) { … }
4  LIBFUNC(void, getValue, void) {
5      return my_value;
6  }
7  LIBFUNC(void, getValue, void) {
8      return my_value;
9  }
```

Library Task Code (Library Task 1)

Figure 4.4: A library task and its example code

interfaces inside. Figure 4.4 describes an SDF graph that consists of three normal computational tasks and three library tasks. The connection between a library task and a regular task is made through a pair of two particular ports: library master port and library slave port. They are represented by white diamonds and black diamonds, respectively. The edge between the library master port and library slave port represents a client-server relationship, not data forwarding. The library task acts as a server and requires a single slave port to connect to multiple master ports.

```
1   TASK_GO {
2       … // code
3        value = LIBCALL(master_port, getValue);
4       … // code
5       value = update();
6       LIBCALL(master_port, setValue, value);
7       … // code
8   }
```

Caller Task Code (Task A)

Figure 4.5: The code templates of the caller task in the figure 4.4

The code template for the library task is in Figure 4.4. Unlike a computational task, a library task is not invoked by input data but by a function call inside from other tasks.

Figure 4.4 and Figure 4.5 show the code templates how a normal task can communicate with a library task. A programmer defines a service as a function with LIBFUNC() directive in a library task. On the other hand, a client task can request a service with LIBCALL() directive.

4.2 Model-based Task Graph Specification for Individual Robots

In the SeMo framework, the internal behavior of each robot is specified by a set of task graphs following the dataflow models of computation.

A task graph consists of tasks and channels, as shown in Figure 2.14, as explained above. A task is a software component that performs the service specified in the mission script. Following the dataflow semantics, a task communicates with other tasks through connected channels via ports. The source task without any input channel is specified as time-driven for the periodic invocation of the task graph. Or the task is specified as control-driven for controlled invocation of the control task. The task graph can naturally specify data dependencies between tasks. Complex services can be expressed as a task subgraph, as explained earlier. For example, an object tracking service is expressed as a set of tasks in Figure 5.10 (b).

Note that the granularity of a task is made large enough to become the scheduling unit as a thread or a process in the generated code. Thanks to the formal semantics of the SDF graph, we can determine the mapping of tasks onto processing elements in the heterogeneous target system manually or automatically. Through profiling by each task, we can predict memory usage, energy consumption, and performance at compile time. Also, we can get a schedule that contains mapping and ordering of tasks on a multiprocessor system to minimize the resource requirement while satisfying the throughput or latency constraint.

To get benefits of static analyzability, we keep the restricted semantics of the SDF graph [113] as much as possible in the SeMo framework. Because the SDF model is not able to specify dynamic behavior, however, its model is not suitable for control-oriented applications that change their behavior dynamically

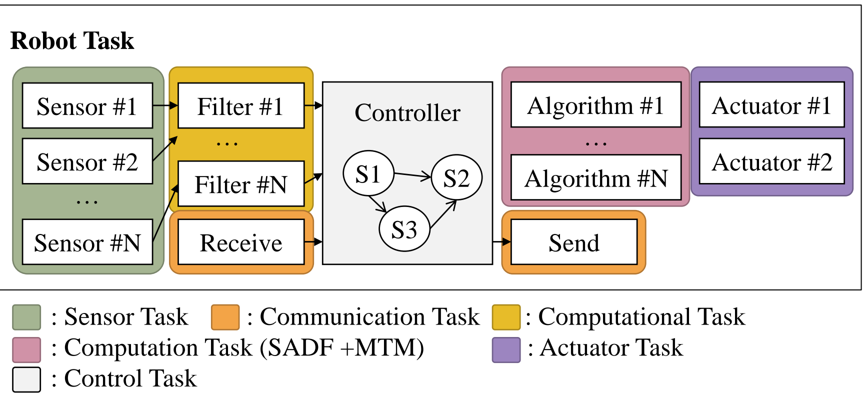


Figure 4.6: Task graph specification of individual robot platform

depending on internal/external events or interrupts. Therefore, we adopt an extended model for dynamic behavior specification. Besides, it is difficult to exploit the parallelism of loop structures since they are not explicitly specified in existent dataflow models. Thus, we propose a novel extension to the SDF model, specifying the loop structures explicitly in a hierarchical fashion.

4.2.1 Applying Dynamic Behavior Specification

To overcome limited expression capability, we use an extended task model that supports a hybrid specification of dataflow and FSM models, distinguishing two types of dynamism: OS-level or application-level [116].

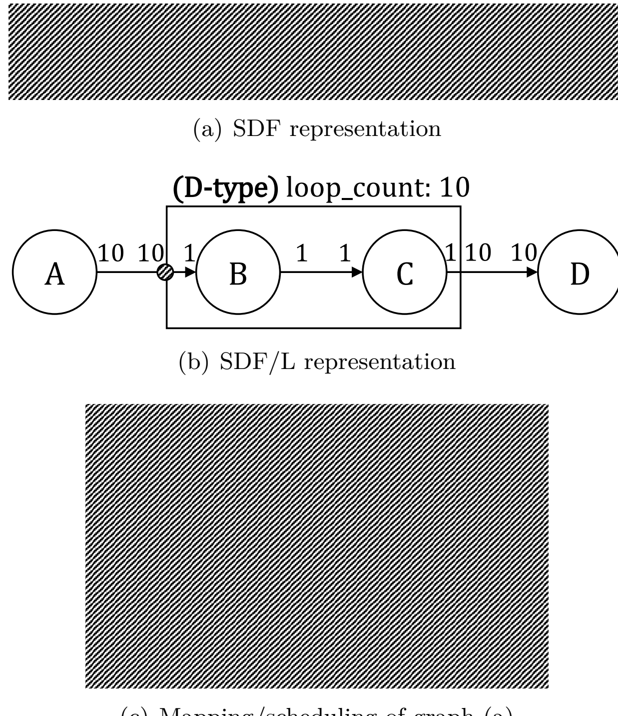
As shown in Figure 4.6, the behavior of each robot is specified by a task graph. Each sensor input task is triggered periodically, and the sensor value is transferred to the control task after being filtered. The control task is a special type of task to control the execution of other tasks and manage the overall system status. It plays the role of supervisor of the internal operation of a robot, similarly to the statechart in the old STATEMATE [122] environment. The control task has a finite state machine (FSM) template inside; a user can specify the state transition condition and the behavior at each state. Note that a task may have a

task graph inside to make the task graph hierarchical. In the example of Figure 4.6, the "Algorithm#1" task may have a task graph inside. Refer to [116] for more details in the hybrid specification of dataflow and FSM.

4.2.2 Extending Iterative Behavior Specification

Since an application usually spends most of its execution time in loop structures, loop structures have been the main object of parallelization for accelerating the application. Even though dataflow models express the task parallelism explicitly, they do not express the loop structures explicitly. In the SDF model, a loop structure is implicitly expressed by sample rate changes. Consider a simple SDF graph that consists of four nodes, as shown in Figure 4.7. Since tasks B and C are executed ten times after the execution of task A, a looped schedule A10(BC)D can be constructed among many possible schedules. In case 10 executions of (BC) can be parallelized with ten output samples from A, a user may want to construct a parallel schedule, as illustrated in Figure 4.7(c). However, identifying such a loop structure and parallelizing it are not easy because parallel scheduling techniques usually aim to exploit task-level parallelism only. Moreover, there is a type of loop structure that cannot be specified with an SDF graph when the number of task executions may vary at run time. In the example of Figure 4.8(a), the number of loop iterations varies depending on the exit condition (line 6).

To overcome these drawbacks, we propose a novel extension to allow the explicit specification of loop structures in an SDF graph. The extended SDF graph with loop structures is called as an SDF/L graph. In an SDF/L graph, a loop structure is encapsulated as a single SDF node at the upper layer, and it contains another SDF graph inside to make the graph hierarchical. Figure 4.7(b) and Figure 4.8(b) are examples of SDF/L graph where a black square represents an initial sample stored in the FIFO channel. A white circle denotes a broadcasting



(a) SDF representation

(b) SDF/L representation

(c) Mapping/scheduling of graph (a)

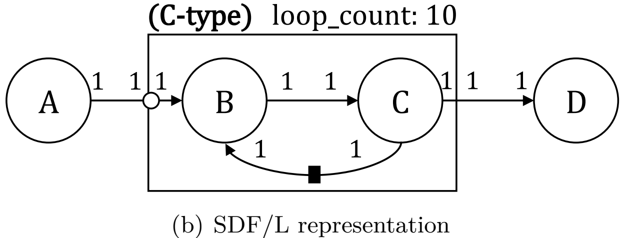
Figure 4.7: (a) An SDF graph that has a loop structure by sample rate changes, (b) SDF/L graph for the SDF graph of (a), (c) a possible mapping/scheduling of graph (a)

port and a shaded circle denotes a distributing port.

For the expression of loop structures in the SDF/L graph, two types of loop structures are distinguished: data loop (D-type) and convergent loop (C-type). In a data loop, the loop structure consumes new input data for each iteration so that the number of iterations is determined by the number of samples on the input FIFO channels. The loop structure in Figure 4.7(b) is a D-type loop. In a convergent loop, on the other hand, the loop structure consumes an input data from the outside and then iterates the internal SDF graph without consuming no more input data. After the iteration ends, it produces an output data. The number of iterations can be fixed statically or vary dynamically at run-time. The loop structure in Figure 4.8 (b) is a C-type loop. Note that a convergent loop

```
1   a = execute task A
2   c1 = initial value
3   for(i := 1~10){
4       b = execute task B with a, c1
5       (c1, c2) = execute task C with b
6       if ( c2 > threshold) break
7   }
8   execute task D with c2
```

(a) Code example of loop structure



(b) SDF/L representation

Figure 4.8: (a) Pseudo-code of an example with a loop structure, (b) SDF/L graph for the code (a)

cannot be expressed by the original SDF model because the sample rate at the boundary of loop structure varies.

We define the proposed SDF/L graph formally and explain the execution semantics of loop nodes.

4.2.2.1 SDF/L graph definition

Definition 4.2.1 (SDF/L Graph)**.** SDF/L graph G is defined as a tuple (N, L, E) where N denotes a finite set of nodes representing the computation tasks, L a finite set of loop nodes, and E a finite set of edges describing the data dependency between adjacent nodes or loop nodes.

Loop nodes are distinguished from normal SDF nodes since they enclose an SDF subgraph inside to construct a hierarchical graph.

Definition 4.2.2 (Loop Node)**.** A loop node L is defined as a tuple (*In, Out,*

loop_count, type, inG, exit_flag, T) where In and Out represent the set of input and output ports, respectively, such that $In \cap Out = \emptyset$, *loop_count* $\in \mathbb{N} \cup \{\infty\}$ the maximum number of the loop iteration, *type* the loop type which is either D-type(D) or C-type(C), *inG* an internal SDF/L graph in the loop node, $exit_flag \in \{true, false\}$ a shared flag that tasks in *inG* can access to check if the exit condition of the loop is satisfied or not, and T a designated task in *inG* that can set the $exit_flag$. Note that $exit_flag$ and T are omitted if the number of iterations is statically determined by *loop_count*.

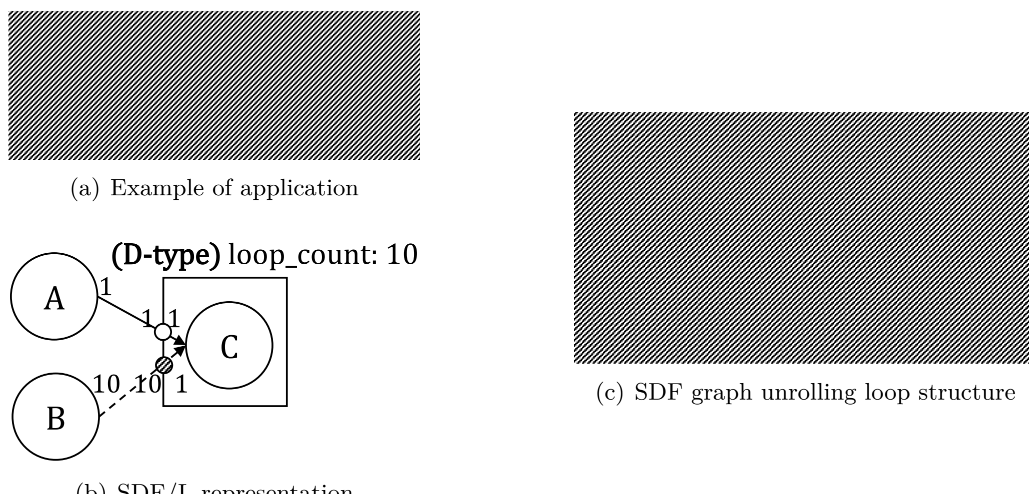
The exit_flag is used for C-type loop structures. In the SDF/L graph of Figure 4.8(b), the exit_flag becomes true when the exit condition ($c2 > threshold$) is satisfied or the loop iteration becomes 10. The exit_flag is initially false before the loop node is executed and a designated task T in *inG* may set it to be true during the execution.

Definition 4.2.3 (Input Port Types)**.** There are two types of input ports for a loop node: *distributing* and *broadcasting*. Input data samples from a distributing port are distributed to the iterations of the loop. Input samples from a broadcasting port are broadcast to or reused in all iterations of the loop.

For instance, the input port of the D-type loop node in Figure 4.7(b) is a distributing port: every iteration the loop node consumes one sample among 10 input samples. On the other hand, the input port of the C-type loop node in Figure 4.8(b) is a broadcasting port.

Definition 4.2.4 (Sample Rate Consistency Condition at Loop Boundary)**.** A port of a loop node is associated with two sample rates, one for the outside connection and the other for the connection to the inner subgraph. To satisfy the sample rate consistency, the following conditions should be held.

- For a *distributing* input port and an output port of a D-type loop, the sample



(a) Example of application

(b) SDF/L representation

(c) SDF graph unrolling loop structure

Figure 4.9: (a) A pseudo-code example, (b) SDF/L graph representation of (a), and (c) a homogeneous SDF graph expanded from the SDF/L graph of (b)

rate of the outside connection is equal to the product of the inside sample rate and the *loop_count* of the loop.

- For a *broadcasting* input port and an output port of a C-type loop, the sample rate of the outside connection is equal to the inside sample rate. It implies that the inside channel at the loop boundary is not a FIFO queue but a buffer.

From the sample rate consistency condition requirement, the following restriction on the port types is induced depending on the loop type.

Definition 4.2.5 (Port Type Restriction)**.** A D-type loop node may have both types of input ports, but a C-type node may not have a *distributing* input port.

Figure 4.9(b) shows an SDF/L graph that is associated with a pseudo code in Figure 4.9(a). 10 iterations of the loop can be instantiated as separate task instances in the homogeneous SDF graph expanded from the SDF/L graph as shown in Figure 4.9(c). A sample from task A is broadcast to all iterations of the loop node while 10 samples from task B are distributed to 10 instances of the loop node.

4.2.2.2 Execution semantics of a loop node

A loop node behaves like a normal SDF node at the upper layer: A loop node becomes executable when all input channels have the specified number of input samples and it consumes and produces the specified number of samples per execution. The internal interface behavior, however, depends on its loop type. If it is D-type, it consumes as many input samples as specified by the associated sample rate from each distributing input port and produces the specified number of samples at each execution. On the other hand, if it is C-type, it consumes the specified number of samples from each input port at the first loop iteration only and produces the specified number of samples at the last loop iteration only. In addition, the iteration number of an internal graph in a loop node is dependent on the loop type. For a D-type loop, the number of iterations is statically determined by the loop_count attribute of the loop node. For a C-type loop, however, the number of iterations may vary at run-time. If the exit_flag is set true, then the loop terminates before it reaches the maximum number of iterations. When the inner subgraph is scheduled at run time, care should be taken to satisfy the causality condition that the next iteration does not start before the designated task T that may set the exit_flag finishes its execution.

4.2.2.3 SDF/L specification of two machine learning applications

We choose two machine learning applications as benchmark applications to confirm the enhanced expression capability of the proposed SDF/L graph.

- K-means Clustering

K-means clustering [123] aims to partition n input data into k clusters in which each data belongs to the cluster with the nearest mean, serving as a proto-

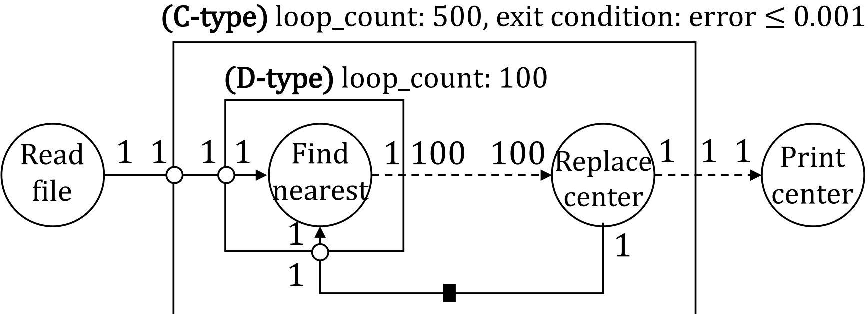


Figure 4.10: SDF/L graph of k-means clustering

type of the cluster. Figure 4.10 represents the SDF/L specification of a k-means clustering algorithm at a very coarse grain.

Based on the Rodinia benchmark implementation [124], the k-means algorithm proceeds by iteratively processing two steps, assignment step and update step, until the assignments no longer change. In our implementation, the threshold of error is set to 0.001 and the maximum iteration count to 500. The C-type loop node contains two tasks: a D-type loop node and *Replace center* that sets the exit_flag of the loop to terminate the C-type loop. The D-type loop node calculates the new means to be the centroids within each sub-cluster, which expresses the data parallelism explicitly.

- Deep Neural Network

Artificial neural network (ANN) [125], which is a machine learning technique inspired by biological neural networks, is composed of an input layer, hidden layers, and an output layer. A deep neural network (DNN) is a deep, fully-connected graph (artificial neural network) with multiple hidden layers between the input layer and output layer. Figure 4.11 shows a simple DNN structure for MNIST [126] dataset recognition. Each node denotes a neuron which reads an input, processes it, and generates an output. Each line represents a connection between two neurons and indicates the pathway for the flow of information. Each connection

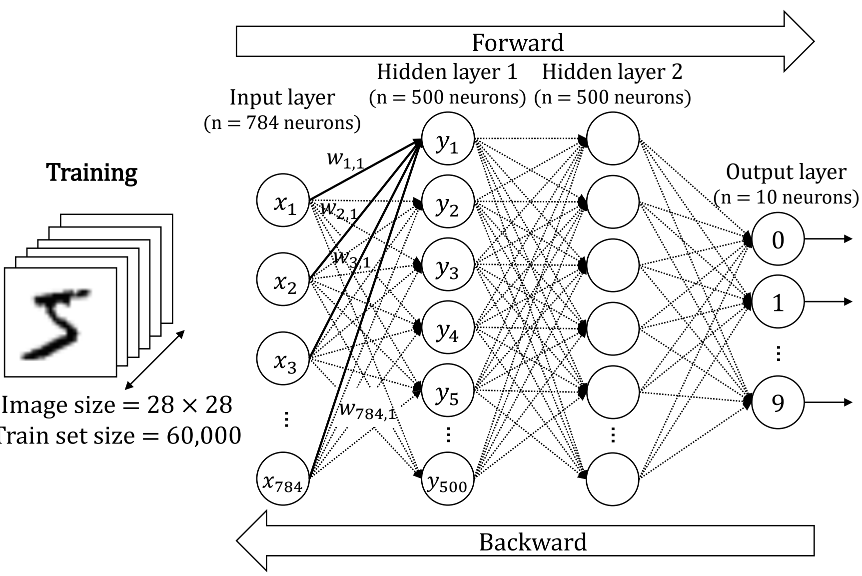


Figure 4.11: Architecture of deep neural network used for digits recognition

has a weight that indicates the amplification factor of the signal between two neurons.

In the Figure 4.11, the leftmost layer is the input layer accepting the input data from the MNIST dataset. Since each training data consists of a 28×28 grayscale pixel image of a handwritten digit, the input layer contains $784(= 28 \times 28)$ neurons. Each input value lies between 0 and 1, indicating the grayscale level of the pixel with 0.0 representing white and 1.0 black. The second and third layers of the network are hidden layers that consist of 500 neurons in this example. The output layer, which is the rightmost layer in the network, represents the features to recognize. In the MNIST dataset recognition, the output layer consists of 10 neurons that correspond to 0 to 9 digits respectively. In summary, Figure 4.11 shows a 784 - 500 - 500 - 10 network, which is similar to the one used in [126].

Note that Figure 4.11 shows the feed-forward inference network only. In the training or learning phase, however, there is an invisible feed-back path, called

back-propagation [127], in the network. The supervised learning is performed by repeating the following two steps until the convergence condition is satisfied. Initially, small random values are assigned to all edge weights. The first step is a feed-forward step that computes the inference outputs using the current edge weights following the forward direction in Figure 4.11. A neuron computes the output with the following equation where σ means an activation function that models a non-linear response of a neuron, w_{ji} means the weight value between the jth neuron in the former layer and the ith neuron in the current layer, and x_j means the input value. In this example, a rectified linear unit (ReLU) function, $f(x) = \max(x, 0)$, is used between input/hidden and hidden/hidden layers.

$$y_i = \sigma(\sum_j w_{ji} \cdot x_j) \tag{4.2}$$

The second step is the propagation of errors in the backward direction to adjust the edge weights to reduce the error, E, between the correct answer and the inference result. A gradient descent algorithm is used to adjust the weight values as follows where η indicates the learning rate.

$$w(t+1) = w(t) - \eta \frac{\partial E}{\partial w} \tag{4.3}$$

The neural network is trained by repeating these two steps.

Note that Figure 4.11 is not a dataflow model since it does not show all computations involved in the training phase, particularly the feedback path for back-propagation. It is very challenging to represent loop structures in the DNN application. The application requires a loop structure to present 60,000 iterative training step from the MNIST database for instance. In addition, this training step is repeated until the training error becomes smaller than a given bound or the maximum number of repetitions is reached. To the best of our knowledge, no

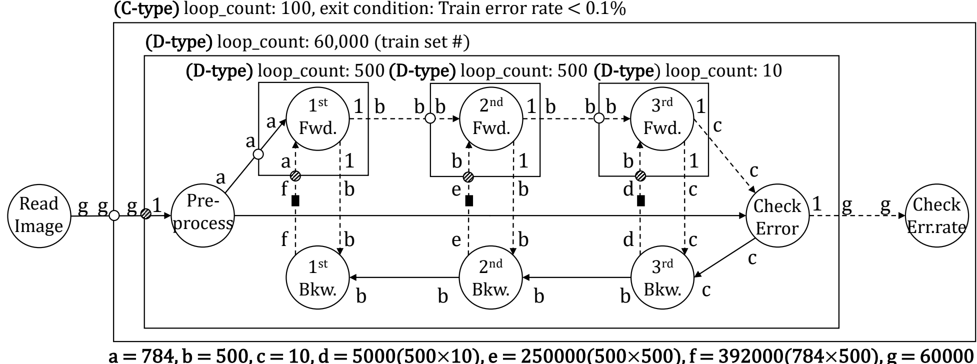


Figure 4.12: SDF/L graph of deep neural network with two hidden layers

dataflow modeling of a DNN has been tried due to limited expression capability of decidable dataflow models.

Figure 4.12 represents the DNN application by an SDF/L graph that has three levels of nested loop structures to build a hierarchical graph. Symbols are used to specify common sample rates where their values are shown in the figure.

1. D-type Loop Node for Each Layer

At the bottom level, three loop nodes specify the computation of two hidden layers and one output layer, respectively. Since each neuron performs the computation of the equation (4.2) independently, each layer is specified by a D-type loop node. Variable *loop_count* in the loop node denotes the number of neurons in the corresponding layer. Note that the input port from the predecessor in the feed-forward direction is a broadcasting port so that all input values are copied to all iterations of the loop node. On the other hand, the input port from the back-propagation path is a distributing port that receives the updated weight factors.

2. D-type Loop Node for One Epoch of Learning

At the second level, a D-type node specifies one epoch of learning. In this example, 60,000 train images are used to adjust the weight values by iteratively

performing the feed-forward and back-propagation algorithms. Since one image is processed in each iteration, *loop_count* is set to 60,000 and the input port is marked as a distributing port.

The inside SDF/L graph consists of eight tasks. Task *Preprocess* receives a single image and sends 784 pixel values to the first hidden layer. After processing the forward computation, task *Check Error* computes the error between the correct answer and the inference output, and triggers the back-propagation algorithm. Note that each layer in the DNN consists of two tasks: one (k^{th} Fwd) in the feed-forward path and the other (k^{th} Bkw) in the back-propagation path. A forward task provides the current weighted sum values to the associated backward task while the backward task sends the updated weight values to the forward task. A black square represents the initial weight values that are necessary for the first execution of the forward task.

3. C-type Loop Node at Top Level

Epochs of learning phase are repeated until the training error becomes smaller than a given error bound. In our example, the error bound is 0.1% and the maximum iteration count is 100. Since the 60,000 test images are reused in every epoch, the input port of the C-type loop is a broadcasting port. The outermost C-type loop node contains two tasks: a D-type loop node and *Check Err.rate*. Task *Check Err.rate* is designated as a special task that can set the exit_flag of the loop.

4.2.2.4 Summary of SDF/L specification

We introduce a novel extended SDF graph, called SDF/L, to express a loop structure as a super node explicitly to build a hierarchical graph. By distinguishing two types of loop structures, D-type and C-type, and two types of ports, broadcasting and distributing, the expression capability of the SDF/L graph is

extended significantly enough to express the full computation workload of a deep neural network (DNN) application in an SDF/L graph.

Since the aim of this study is to enhance the expression capability of decidable dataflow models to support iterative applications, not claiming that dataflow modeling is better than other specification methods for iterative applications, we can compare the related work on the SDF extensions in brief. Some models such as Cyclo-static dataflow (CSDF) [117] and fractional rate dataflow(FRDF) [118] model can not specify loop structures with varying iterations. Although the scenario-aware dataflow (SADF) model [25], [116] model, Parameterized synchronous dataflow (PSDF) [41] model can specify all of loop structures, they require additional specifications, which make them complicated. We believe that SDF/L graph is more natural and simpler to express the loop structures. Furthermore, in contrast to previous work on dataflow modeling, the proposed SDF/L graph allows users to control the execution of the loop structures explicitly and to perform task scheduling in a hierarchical fashion.

4.3 Model-based Task Graph Specification for Cooperating Robots

A significant difference between a single robot and cooperating robots lies in information sharing. We distinguish between two types of shared information management: centralized management and decentralized management. In a centralized approach, globally shared information is managed by a specific robot, ensuring global information consistency. Whenever new information is entered into the robot that manages global information, the information is updated, and the information is sent whenever a request occurs. In the decentralized approach, on the other hand, the whole robot has its own information, and it sends and receives information. Although global consistency is not guaranteed, it is more robust than a centralized method because it is not affected by the specific robot that manages global information. Both approaches have their pros and cons, so the programmer needs to decide how to share information based on the characteristics of the information and the robot. This section introduces how to specify shared information in two approaches.

4.3.1 Globally Shared Information Specification

For the cooperation of heterogeneous robot platforms, we need to support shared resource management and server-client interaction. To this end, global information is maintained by a special type of the task, called library task [42]. Figure 4.13 shows how multiple robots share information. As mentioned earlier, a library task can handle access conflicts for shared variables. It defines a set of function interfaces inside to share data or algorithms. And other tasks may request to get or set data or run algorithms for library tasks.

Figure 4.14 is the code example that shows how a normal task can commu-

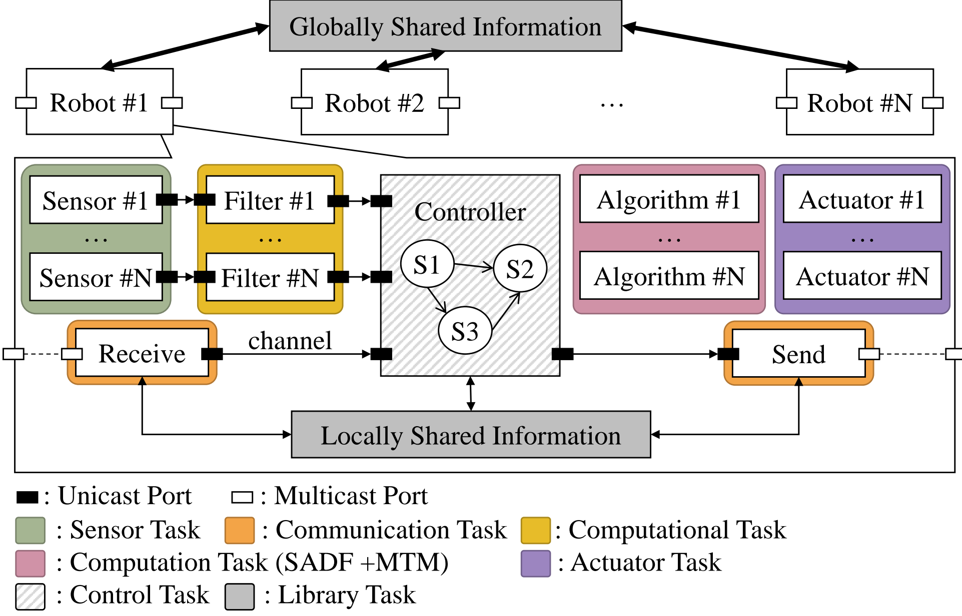


Figure 4.13: Task graph specification for the internal behavior of a robot, supporting two types of information sharing

nicate with a library task. For motivational example 2, a list of colors already founded by robots is shared. And each robot requests a service with LIBCALL() directive to update the list.

Since a library task is a shareable and mappable object, a robot that maps a library task has a separate thread that sequentially processes library requests from other tasks or other robots, and allows only one request to access shared information at a time. Therefore, all robots can access the same global information

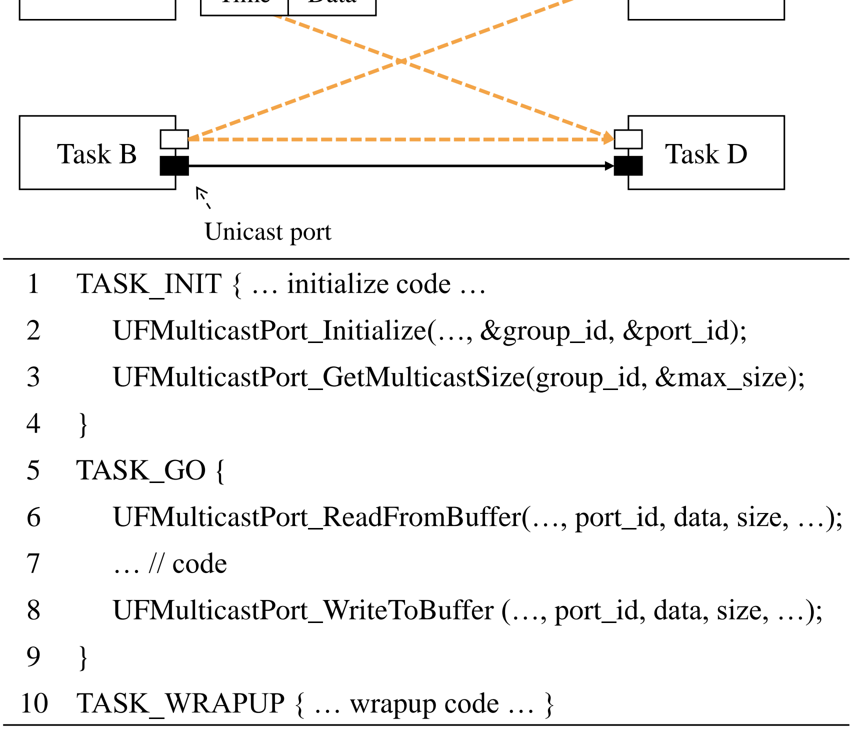
1 TASK_GO { … // code	static int currentColorList[N];
2 LIBCALL(…, updateColor, RED);	LIBFUNC(void, updateColor, int colorVal){
3 … // code	currentColorList[colorVal] = 1;
4 }	}
< Caller Task Code >	**< Library Task Code >**

Figure 4.14: The code templates of a library task and the caller task

by sending a query to the robot. However, if many tasks call the library at the same time, it can be a bottleneck. In addition, the robot is a single point of failure, so it is essential to keep the robot live at all times.

4.3.2 Locally Shared Information Specification

Robots may want to share information locally with near robots, without guaranteeing the global consistency of the knowledge. This local knowledge sharing is useful for performing a group service together. Adopting the knowledge sharing technique proposed in [128], we disseminate the local knowledge through broadcasting. Since the broadcasting message includes the creation time of knowledge, the robots can maintain up-to-date knowledge. By delivering the received up-to-




```
1   TASK_INIT { … initialize code …
2       UFMulticastPort_Initialize(…, &group_id, &port_id);
3       UFMulticastPort_GetMulticastSize(group_id, &max_size);
4   }
5   TASK_GO {
6       UFMulticastPort_ReadFromBuffer(…, port_id, data, size, …);
7       … // code
8       UFMulticastPort_WriteToBuffer (…, port_id, data, size, …);
9   }
10  TASK_WRAPUP { … wrapup code … }
```

Figure 4.15: Multicast port and the example code

date knowledge to neighbors, even distant robots will receive the local knowledge eventually. While the task graph model in the SeMo framework only supports one-to-one communication using channel through ports, we add another type of port for multicasting, as shown in Figure 4.13 and Figure 4.15.

Multicast is a group communication in which data transmission is addressed to a group of destinations simultaneously [129]. Therefore, the same multicast group can communicate using the multicast port without connecting tasks one by one. In Figure 4.15, task A and B communicate with task C and D using the multicast port. If the programmer uses only unicast ports that need to be connected to the channel, the communication looks complicated. Also, if task E is added while task A and B are running, it is difficult to represent new communication among task A, B and E. Multicast communication, on the other hand, allows task E to receive information from task A and B, if task E is the same multicast group with task A and B. Thanks to the characteristics of multicast, adding or removing robots dynamically does not affect the entire system while the robots work together.

However, multicast communication is implemented using buffers rather than FIFO queues, which does not guarantee the transfer of information between tasks. Therefore, multicast is not suitable when data must be delivered securely.

4.4 Automatic Code Generation

In the proposed technique, the actual code that will run on each processor is generated automatically from the task-graph specification based on the compile-time decision on mapping and scheduling. This feature will increase the design productivity of software by minimizing the possibility of human error in manual programming. Refer to [116] for more details. The overall code generation procedure in the SeMo framework is shown in Figure 4.16.

The code generation flow consists of two main steps: platform-independent code generation and platform-dependent code generation. In the first step, we construct data structures for tasks, channels, and libraries and the skeleton of the main scheduler code that creates the task threads. Inter-task communication method depends on the mapping result. For instance, if a client task calling a

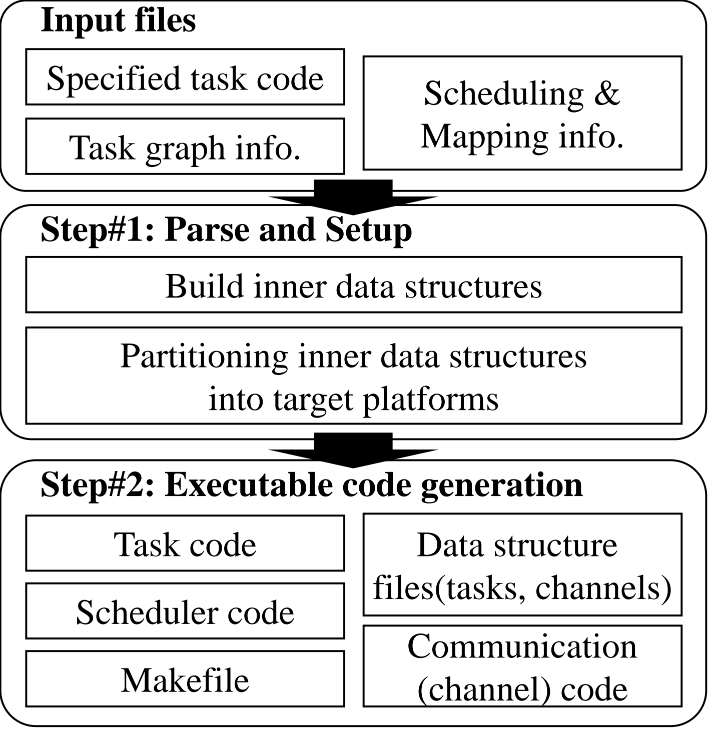


Figure 4.16: Code generation flow in the SeMo framework

```
1   void thread_routine(){
2       TASK_INIT;
3       while(true):
4           TASK_GO;
5           if (termination condition) break;
6           if (time_driven_task) wait remaining time;
7       TASK_WRAPUP; }
8   void main{
9       target_dependent_init(); init_channel();
10      for all tasks in a specified task graph
11          THREAD_CREATE(…, thread_routine);
12      wrapup_channel(); target_dependent_wrapup(); }
```

(a) Scheduler code

```
1   int Core_1_go(){
2       int i_a = 0;
3       for(i_a =0; i_a < 3; i_a++)
4           TaskA_GO();
5       TaskB_GO();  …
```

(b) TASK_GO code for a virtual task

Figure 4.17: Scheduler code

library function and the library task are mapped to the same processor, the code is simply generated in the form of a function call. Otherwise, the additional code should be generated to access the library function that is mapped to a different processor.

The skeleton of the scheduler code is shown in Figure 4.17 (a). In main() function, it first calls *target_dependent_init*() function for platform-specific initialization. Next, a function is called to initialize channels and allocate buffers for each channel. The scheduler invokes threads for all tasks. In the thread routine, it calls TASK_INIT/GO/WRAPUP functions of its task. Note that the internal

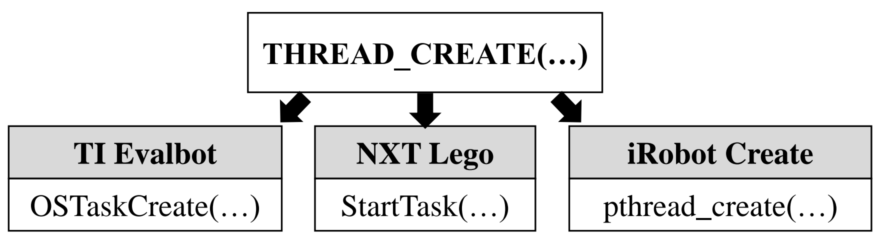


Figure 4.18: Target specific code for thread creation

code of a task is already prepared in the task library, and no platform-specific API is used in the code except some I/O tasks that access hardware components directly. Thus, we distinguish two kinds of task libraries: platform-independent task library and platform-dependent task library.

The second step of platform-dependent code generation depends on the operating system that the robot platform uses. For the tasks mapped on a processor, we may use the scheduler of the OS. However, we may want to follow the scheduling decision made at compile time. Then we make a virtual task that contains the subgraph and generate an appropriate data structure for the virtual task. After constructing a static schedule for the task graph, the TASK_GO function of the virtual task is generated that invokes the TASK_GO functions of tasks in the statically determined schedule order of tasks. Figure 4.17 (b) illustrates an example of TASK_GO function of a simple virtual task. In this step, target-independent APIs are redefined with target-specific APIs, and target-specific code is added for initialization and wrapup actions. Figure 4.18 shows an example of target-specific code for thread creation for each target platform used in our experiments (Table 4.1).

In addition to the application code, an adaptive resource management module can be synthesized in case the resource management policy is specified in the strategy description file to increase the lifetime of the robot under a given energy budget. The adaptive resource manager monitors the remaining battery power to

change the execution policy if necessary. It may change the invocation period of tasks or send signals to the control task so that specific algorithms and actuators can be turned on or off, or parameters such as motor speed in the actuator task can be changed.

4.5 Experiments

To prove the viability of the proposed methodology, preliminary experiments are conducted with a scenario of the cooperative mission. It verifies the automatic code generation process from task graphs to the actual robot codes. This experiment confirms the benefits of the modular structure of the SeMo framework; we may change the hardware platform easily without laborious manual coding. Finally, we compare the memory usage of ROS and our framework. The information on heterogeneous robots used in the experiments is summarized in Table 4.1.

The scenario is to perform a cooperative mission, which explained in Section 3.1.2. It verifies the reusability of software thanks to a platform-independent design with model-based task graph specification. Figure 4.19 shows the snapshot of this scenario using three robots. Automatic code generation also improves the productivity of software development since the error-prone job of interface code design and thread scheduling is all automated. The detailed experimental results and process from mission specification to task graph model specification can be found in the Appendix B.

As an indirect measure of effectiveness and productivity improvement of the

Table 4.1: Target platforms

	TI Evalbot	**NXT Lego**	**iRobot Create**
OS	uC-OS3	NXT-OSEK	Linux (Ubuntu)
Processor	LM3S9B92 (80MHz)	Atmel 32-bit ARM (48MHz)	ARM Cortex-A7 (quad, 900MHz)
Memory	96KB	64KB	1GB
Extension	Arduino UNO (for sensors)	-	Raspberry Pi 2 (for controller)
Code size(lines)	2721	3301	2960
Manual code ratio(%)	30.43	23.60	31.42

Figure 4.19: Heterogeneous robots work together to find colored paper like motivational example 2

Table 4.2: The memory usage (VSZ/RSS) comparison between ROS and the SeMo framework (Unit: MB)

Scenario	**ROS**	**SeMo**	**Ratio**
Remote control	282.8 / 64.02	25.55 / 0.44	11.07 / 146.32
Tracking object	568.92 / 103.73	206.82 / 45.18	2.75 / 2.30

proposed framework, in the last two rows of Table 4.1, we show the total lines of the generated code and how much is the portion of user-specified code. As shown in Table 4.1, about 70% of the total code is automatically generated on average. Considering that the average development period is proportional to the line counts [130], the result indicates that the productivity is improved about 3.3 times.

4.5.1 Memory Usage Comparison with ROS

The resource requirement is an essential metric for miniature mobile robots. To compare the memory footprint between ROS and our framework, we implemented two mission scenarios in ROS with an "iRobot Create" robot: one is a

simple remote control scenario, and the other is tracking a specific object. For other robots, TI Evalbot and NXT Lego, a comparison cannot be made since ROS does not support their operating systems. Table 4.2 displays the virtual set size (VSZ) and resident set size (RSS) value of the process, measured by using *ps* command. As shown in the table, ROS requires significantly larger memory than our framework, particularly when the computation task size is small. This is because additional processes such as *rosmaster* and *roscore* are required in ROS besides the computation tasks to manage nodes and messages when ROS is running.

Chapter 5

High-level Mission Specification

In this chapter, a new scripting language is devised to make it easy for casual users to learn and use for the mission specification of cooperating robots. In addition, to increase the robustness, scalability, and flexibility of robot collaboration, we add some critical features of swarm robotics in the scripting language.

The mission scripting language has the following features that are not fully supported by existent scripting language: hierarchical team composition, service-oriented programming, multitasking specification, dynamic mode change, dynamic group allocation. Also, to fill a large gap between two abstractions, mission specification and model-based task graph specification, an intermediate level of abstraction, called strategy description, is introduced. In this chapter, we explain these two levels of abstraction with two motivational mission scenarios explained in Section 3.1.2 and Section 3.1.3 in which robots move to a specific destination and collaborate to find some treasures. In addition, the task graph specification described in the previous chapter can be automatically generated from the mission specification and strategy description. Therefore, the mission specification is refined to the robot codes through strategy description and task graph generation in the proposed framework.

5.1 Service-oriented Mission Specification

For the mission specification of cooperating robots, a new scripting language is devised to make it easy for casual users to learn and use. The mission scripting language has the following four features that are not fully supported by existent scripting languages: team formation, service-oriented programming, multitasking specification, dynamic mode change. In this section, we explain these features with a motivational mission example described in the previous Section 3.1.2.

The robots are grouped into teams. In our example, there are two teams of robots, *Team1* and textitTeam2. A robot in *Team2* plays the role of the master that controls the actions of robots in *Team1*.

A robot may have multiple modes of operation depending on the environment and user requests. In our example, robots have three modes of operation: remote control (”RC_MODE”), autonomous move (”AUTO_MODE”), and treasure-hunt mode (”SEARCH_MODE””). An event triggers the mode change. For instance, in the remote control mode, the user sends commands to control the robot’s action remotely. To make a change to the autonomous move mode, the user can send a specific command that becomes a triggering event of mode change.

A robot usually performs several tasks at the same time, equipped with various sensors, actuators, and computation components. Thus, it is necessary to specify multitasking naturally. Multitasking is not easy to express in a popular scripting language such as Python and Lua. For multitasking specification, we adopt the notion of *plan* from an earlier work [131]. A plan is a sequence of executions like threads. To simulate multitasking, Python and Lua use the concept of a coroutine, which is different from multitasking because only one coroutine is executed at a time to emulate multitasking by time-sharing. In our example, we have three plans running concurrently: ”Listen”, ”Report”, and ”Action” plan. The ”Listen” plan describes that robots receive information from the operator

Mode \ Plan	Listen Plan	Report Plan	Action Plan
Remote Control	(a) Receive commands from operator	(b) Broadcast commands	Move by command
Autonomous Move	Receive warnings from other teams	Send its location	Move avoiding obstacles
Treasure Hunt		Send its view	Search target

(c)

Figure 5.1: Specification of behaviors by robot team "Team1"

or other robots periodically. The "Report" plan contains information on when it reports its status or action result to the operator or other robots. It may include the current position of the robot, hardware information such as battery status, and information about the object being searched. The "Action" plan depicts the behavior that the robot will perform, for instance, moving to a specific region and searching for a target object.

Figure 5.1 illustrates what tasks to be performed in each combination of mode and plan for Team1 of our example. As explained earlier, we abstract the task functions as ***services*** and register them in the database. For example, a robot may have an autonomous driving service that moves to a specific point, a video service that captures an image, and a detecting service that perceives a specific object. A user can define a ***composite service*** as a sequence of basic services. For instance, a "scouting" service can be defined as a sequence of driving, video, and detecting services. A composite service may include composite services inside to make it hierarchical.

The syntax of the scripting language is formally defined by the Backus-Nauer form (BNF). The suffixes "*", "+", and "?" mean "repeated zero or more times," "repeated one or more times" and "zero or one time," respectively.

5.1.1 Service-oriented Programming

The mission scenario is a collection of team scenarios that consists of two parts, team composition and team behavior. The team behavior comprises service definition, multitasking information, and dynamic mode change information.

```
<MissionScenario> ::= <TeamScenario>+
<TeamScenario> ::= <TeamComposition> <TeamBehavior>
<TeamBehavior> ::= <Function>* <Service>+
                   <Multitasking>+
                   <DynamicModeChange>
```

Figure 5.2 illustrates a part of mission specification for *Team1* written in our proposed scripting language.

As mentioned above, robots are grouped into teams. The syntax for team formation in BNF is following:

```
<TeamComposition> ::= <TeamName> : <RobotList>+
<RobotList> ::= <RobotType> <RobotName>
```

It specifies the composition of teams with a list of robots. Each robot is described by its type and name. In our example, there is a team whose name is *Team1* as illustrated in line 1 of Figure 5.2. *Team1* includes an *r1* which type is *lego_robot*.

A behavior of a robot is defined by a sequence of services in the scripting language. ”GroupStmt” will be explained in Section 5.1.3. The syntax of service is shown as follows:

```
<Service> ::= <TeamName>.<PlanName>.<CompServiceName>
              { <Stmt>+ } <RepeatStmt>?
<Stmt> ::= <GeneralStmt> | <GroupStmt>
```

```
1   Team1: lego_robot r1…                              # team formation
2   Team1.Listen.Comm_operator{          # composite service definition
3     receive (Team2, Team2.warning) …. } repeat (10 sec)
4   Team1.Report.Broadcast {
5     send (user, Team1.location) …
6   } repeat (mission_ongoing)
7   Team1.Action.ApproachDestination {
8     move (Torino Congress Center)
9     if (Team1.arrived) throw find_treasure
10  } repeat (Team1.not_arrived)
11  Team1.AUTO_MODE {                          # multitasking per mode
12    set(Listen, Comm_operator)
13    set(Report, Broadcast)
14    set(Action, ApproachDesination) }  ….
15  Team1.main {                     # dynamic mode change definition
16    case (AUTO_MODE):
17      catch(find_treasure): mode = SEARCH_MODE
18      catch(remote_control): mode = RC_MODE
19    case (SEARCH_MODE): ….
20    default: mode = AUTO_MODE }
```

Figure 5.2: Mission scripting language example associated with Figure 5.1

```
<RepeatStmt> ::= repeat (<LoopCondition>*)
<Function> ::= def <FunctionName>
              { <GeneralStmt>+ }
```

Since a service depends on the plan it belongs to, the plan name is explicitly specified in the syntax of a service. Iterative execution of a composite service is specified by *repeat* phrase with an argument that determines whether it continues or not. Three composite services, one for each plan, are defined in lines 2-10 in Figure 5.2.

Our scripting language supports conditional and loop constructs in a composite service definition. Conditional executions can be described by using *if* and *else* phases. A *loop* statement is used to express an iterative execution of services. Since communication statements are frequently used, *send* and *receive* statements are reserved for pre-defined services. A *throw* statement is used to generate an event in a composite service. This output event can be used to change the mode dynamically or to perform conditional execution of behavior. In line 10 of Figure 5.2, if *Team1* arrives at the destination, it throws an event to change the mode to ”SEARCH_MODE”. Moreover, there are various kinds of built-in-services such as ”move”, ”video capture”, and ”detect an object”, which mean the services registered in the database. Statements used frequently can be registered as functions, but they are not used to represent simple motivational examples. The syntax of general statements in BNF is following:

```
<GeneralStmt> ::= <ConditionalStmt>
              | <IterationalStmt>
              | <ExpressionStmt>
<ConditionalStmt> ::= if(<Condition>+) {<GeneralStmt>+}
                      <ElseStmt>?
<ElseStmt> ::= else {<GeneralStmt>+}
<IterationalStmt> ::= loop(<LoopCondition>+){<GeneralStmt>+}
<LoopCondition> ::= <PeriodTime>
                | <Condition>
<ExpressionStmt> ::= send(<TeamName>, <Attribute>+)
                 | receive(<TeamName>, <Attribute>+)
                 | <Built-in-Service, <TimeoutStmt>?>
                 | <FunctionName>
                 | throw <EventName>
```

```
<TimeoutStmt> ::= timeout = <PeriodTime>
```

5.1.2 Multitasking and Dynamic Mode Change

As explained earlier, multitasking is supported by having multiple plans running concurrently. Depending on the mode, the action of each plan may vary as shown in Figure 5.1 (b). Thus, we map a composite service to each plan in each mode by using the following syntax:

```
<Multitasking> ::= <TeamName>.<ModeName> {<SetStmt>+}
<SetStmt> ::= set(<PlanName>, <CompServiceName>, <TimeoutStmt>?)
          | set(<PlanName>, OFF)
```

To terminate a plan, *set* (<PlanName>, **OFF**) can be used. Lines 11-14 in Figure 5.2 show how to define the action of each plan in the "AUTO_MODE".

When a robot starts its execution, the *main* loop is executed forever, initially setting the mode to the *default* mode. A generated event triggers a mode transition. Remind that a composite service may generate an output event. The *main* loop decides the next mode of operation based on the event caught by *catch* phrase during the execution of the current mode. In Figure 5.2, mode conversion can be found in lines 15-19, where the initial mode is set to "AUTO_MODE" in line 20. The summary of the dynamic mode change syntax is following:

```
<DynamicModeChange> ::= <TeamName>.main{
                        <ModeChange>* <InitialMode>}
<ModeChange> ::= case(<ModeName>):<EventListener>*
<EventListener> ::= catch(<EventName>):<ModeAssign>
<ModeAssign> ::= mode = <ModeName>
<InitialMode> ::= default:<ModeAssign>
```

5.1.3 Extensions for Multiple Robots

A cooperative mission is specified as a sequence of services that the robots perform at a high level by a scripting language: grouping robots into teams and specifying the behavior of each team assuming that all robots in a team perform the same specified service. The robots that perform different tasks should be assigned to different teams. While it supports communication between teams, it does not allow robots in a team to communicate with each other. Moreover, it has no consideration of robot failure, which is a common assumption in swarm robotics. Since the behavior of the robots is statically determined, if one robot fails to perform its role, the entire mission is affected.

In order to improve the robustness, scalability, and flexibility of robot collaboration, we consider adding some key features of swarm robotics in the scripting language. Unlike the fixed team formation in a priori research, we add a team hierarchy, which allows the developer to form a group of robots dynamically in a team. A team of robots may have several groups that perform different services at the same time. Also, a new notion of a service, called ***group service***, is introduced, which corresponds to the cooperative mission specification in the top-down approach. Moreover, intra-team communication via broadcasting and local information sharing, which are essential for swarm robotics, are supported in the extended framework. Thus, the proposed framework enables a casual user to specify various types of cooperative missions for distributed robot systems, swarm robots, and their hybrid. In this section, we explain these features with a motivational mission example, which explained in Section 3.1.3.

Figure 5.3 shows a snippet of mission specification associated with the example mentioned above. Robots are grouped into teams (line 1 and 2) and the behavior of each team is defined with a ***composite service***; for instance, *ApproachDestination* (lines 3-5) and *Search* (lines 6-16) are two composite services

```
1   MasterTeam: iRobotCreate irobot                         # team formation
2   SlaveTeam: TurtlebotBurger burger[2], Ev3Robot ev3[2]
3   SlaveTeam.Action.ApproachDestination {
4     move (Paris palace of congress )   # composite service definition
5     throw find_color_paper } repeat (SlaveTeam.not_arrived)
6   SlaveTeam.Action.Search {
7     [[
8     group (instance of TurtlebotBurger) {
9       loop(2 SEC)
10        if  (SlaveTeam.lightness < 200 )
11          publish(SlaveTeam, SlaveTeam.Message = suggestHiding)}
12    others {
13      searchPaper();
14      publish(SlaveTeam, SlaveTeam.Message = suggestReturning) }
15    ]]
16  } ….
17  SlaveTeam.AUTODRIVE_MODE{          # multitasking per mode
18    set( Listen, CommOperator )
19    set( Action, ApproachDestination) } ….
20  SlaveTeam.main {               # dynamic mode change definition
21    case (AUTODRIVE_MODE):
22       catch(find_color_paper): mode = SEARCH_MODE
23       catch(remote_control): mode = RC_MODE
24    case (SEARCH_MODE): ….
25    default: mode = AUTODRIVE_MODE }
```

Figure 5.3: Mission scripting language example associated with Section 3.1.3

for SlaveTeam. The internal behavior of a ***composite service*** is defined by a sequence of services that the robots will perform. There are two plans, *Listen* and *Action*, defined for *SlaveTeam* as can be found on lines 17-19 in the example of Figure5.3. Besides, how to express mode transition can be found on lines 20-25.

In a priori research, the cooperative mission is described by a set of team services that may be changed dynamically depending on the mode of operation. The first extension is made to the composite service definition of a team, as displayed below in the BNF form.

```
<GroupStmt> ::= [[ <LeaderDef> <OtherStmt>? ]]
              | [[ <GroupDef>+ <OtherStmt>? ]]
<LeaderDef> :: = leader(<GroupCondition>) {<GeneralStmt>+}
<GroupDef> ::= group(<GroupCondition>) {<GeneralStmt>+}
<OtherStmt> ::= others {<GeneralStmt>+}
<GroupCondition> ::= instance of <RobotType>+
                   | capable of <RobotCapability>+
```

We introduce the team hierarchy in which the robots in a team can be grouped dynamically. While team formation is statically defined at the beginning of the mission script, groups are defined inside the composite service to support dynamic grouping. We support two types of group structure, leader-follower structure [39] and peer-peer structure [68], [71] as defined in "GroupStmt" in the above BNF description. As illustrated in Figure 5.4, the leader-follower structure is a vertical structure in which one robot is designated as the leader to control the other robots. The peer-peer structure, on the other hand, has a horizontal relationship between robots.

We can group robots based on the type or the capability of the robot ("Group-Condition" in the BNF description). For example, robots with a camera sensor can perform video shooting service. In the composite service of a team, we may divide the robots into multiple groups that are assigned different services. A pair of symbols, [[and]], are used to specify concurrent execution of services by multiple groups of the robots. Note that a robot can belong to one group at most; if it satisfies more than one grouping condition in a composite service, it has to

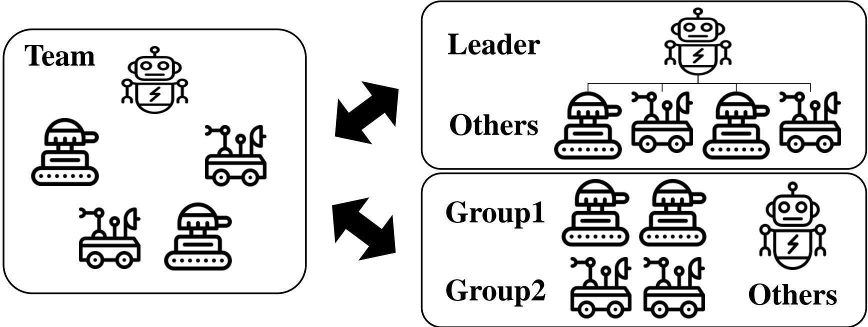


Figure 5.4: Dynamic group allocation

choose one group arbitrarily. In the example of Figure 5.3, *SlaveTeam* is divided into two groups in the *Search* service, as shown in lines 7-15. One group that is of type *TurtlebotBurger* detects a dangerous condition, while the other group that consists of the remaining robots search for the color papers.

The second extension is the introduction of group service that can be performed by two or more robots together without the specification of the role of individual robots. It is an example of a top-down specification for swarm robotics. How to perform a group service is elaborated in the strategy description in the proposed methodology. The service *SearchPaper* described in Figure 5.3 can be defined as a group service, for instance. Since we assume that a robot in a group may fail during operation, the group service is performed collaboratively by available robots.

As the third extension, one-to-many communication is added. The SeMo framework supports one-to-one communication only with two APIs, *send* and *receive*. Since multicasting can be realized by multiple one-to-one communications, it can specify the cooperative mission of a distributed robotics system as long as all robots are live during operation. Unfortunately, any robot may fail in swarm robotics, so it is recommended to use broadcasting. To this end, a pair of new APIs, *publish* and *subscribe*, is introduced. In line 11 and 14 of Figure 5.3, broadcasting

communication is used for each group to send a message.

Last not least, a new semantics for robot synchronization is defined. Since the performance of robots has a large variation, the robots in a team may sit in a different execution state. For group services, however, we need to synchronize the robots. Thus, in the proposed semantics, all robots involved in a group service are synchronized at the start of the group service. In the case of the leader-follower structure of grouping, we may need to select a new leader at the group formation step if the previous leader fails during operation. The leader selection scheme is not described in the mission specification phase but in the strategy description phase.

In summary, the proposed extension offers greater flexibility and robustness than the previous work, assuming that any robot may fail during the execution of group services, and grouping of robots can be made at run-time dynamically. Figure 5.3 shows how a group service is used in the service-oriented mission specification as a hybrid system example of distributed robotics and swarm robotics. The full script codes are covered in detail in Appendix C.

5.2 Strategy Description

There is a large gap between two abstractions, mission specification and model-based task graph specification from which the actual target code is generated automatically. To fill this gap, an intermediate level of abstraction, called a strategy description, is introduced in the SeMo framework.

In this layer, a high-level service specified in the mission script is refined into a set of lower-level services that can be mapped to the tasks in the task graph model. Suppose that ”move to X” service is used in the mission script. Since there may exist multiple algorithms that require different hardware resources and computation power for autonomous moving, it is necessary to specify which algorithm to use for actual code generation. For strategy description, we use the XML markup language instead of using a scripting language or programming language, as shown in Figure 5.5. The XML format is suitable for the description of complex requirements with the relatively simple schema. This XML description can be replaced with a graphical user interface (GUI) for user-friendliness. For example, when detecting a color value, some robots can use the color sensor directly, while others need to use the color filter after capturing an image from the camera. Figure 5.5 shows a part of the XML description that refines the ”move” service into a set of fine-grain services to accomplish the following action: ”check the current position and obstacles, and then move on wheels.” This refinement information is supposed to be prepared by a robot manufacturer or a knowledgeable user in the robot operation.

Since several extensions are made in the mission specification, the corresponding extension should be made in the strategy description. In particular, we need to clarify how to synchronize robots for the leader selection and execution of group service. It is also necessary to specify which algorithm to use for selecting leaders or groups. Besides, how to share the information and who has the information

should be expressed in this layer.

Also, non-functional requirements that are not expressed in the mission can be specified in the strategy description file. For instance, it is necessary to monitor the remaining battery energy of miniature mobile robots before performing a service.

```
1  <Strategy name="SlaveTeam" xmlns="…">
2    <ValueInfo>
3      <Value name="distance">
4        <TargetRobot name="TurtlebotBurger" .. />
5        <CheckSensor name="LaserDistanceSensor">
6            <Parameter name="distance" …/> </CheckSensor>
7      </Value> …. <!– other values > </ValueInfo>
8    <ServiceInfo>
9      <Service plan_name="Action" service_name="move">
10       <TargetRobot name="Ev3Robot" .. />
11       <ActionsInfo>
12         <Action name = "CheckGoal" actionId ="0" >
13           <Parameter name="current_location" .. />
14           <Parameter name="goal" .. />
15           <Transition srcId="0" dstId="7" >
16               <Condition cond="current_location == goal" />
17           </Transition> … <!– other transition>
18         <Action name = "CheckObstacle" actionId ="1"> …
19     </Service> … <!– other services >
20   <GroupingInfo>
21       <LeaderSelectionInfo>
22           <plan_name="Resolve" cs_name="decide" … /> …
23   <BatteryRequirement>
24     <Condition cond="battery < 50" >
25       <Sensor name="Distance Sensor" period="10" / >
26  … <!– non-functional requirement>
```

Figure 5.5: A strategy description example of moving service

Then, a user may change the algorithms depending on the remaining battery level and the characteristics of the work performed by the robot. In our example, we check the battery level and adjust the execution period of a particular service (sensing or actuating) or the algorithm for adaptive resource management at run-time.

5.3 Automatic Task Graph Generation

Based on the strategy description file, the task graph specification can be automatically generated from a given mission specification. For each robot, services that the robot should perform are identified. Since we assume that the task graph associated with each service is registered in the database, task graphs are instantiated as illustrated in Figure 4.13. We can instantiate all the required tasks from the mission specification in a straightforward fashion without any interconnection among tasks. With the instantiated tasks, we have to add dependency arcs between tasks by analyzing the dependency between services in the mission

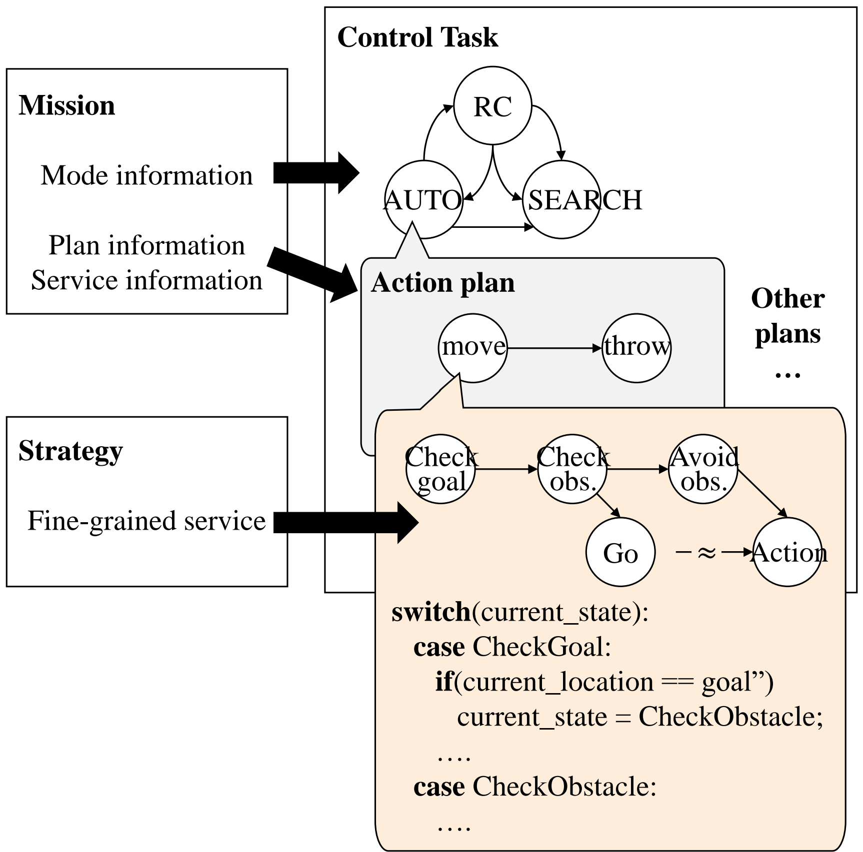


Figure 5.6: Automatic generation of control task from mission specification and strategy description

specification. In case the mission requires knowledge sharing, we identify the requested types of knowledge sharing and insert the additional tasks for knowledge sharing and associated communication ports and channels.

The most challenging is to synthesize the control task that realizes the dynamic behavior of each robot. In the control task, a hierarchical finite state machine (FSM) is constructed according to mission specification and strategy description. In the mission specification, a robot may change the operation mode, and grouping is changed dynamically. Such dynamic behavior is expressed by a hierarchical FSM inside a control task, *Controller* in Figure 4.6 or Figure 4.13. Figure 5.6 sketches how the mode and plan information from the mission is translated into the state transition diagram in the top-level FSM. In each mode, the sequence of services is translated into a hierarchical FSM. In bottom-level FSM, we may change the execution state of the task, such as suspending and resuming or change the parameter of the task. As illustrated in Figure 5.5, we can specify a condition variable used as a state transition condition in the synthesized bottom-level finite state machine.

The inside of the control task is shown in Figure 5.7 for the mission specification of Figure 5.3. Note that we generate different FSMs for two robots that belong to the same team since they belong to different groups in the *Search* composite service. Since the group conditions can be inferred from the candidate robots of the group in advance, the robots only contain a set of services that need to be performed. For example, a sensing task that checks brightness should be included only inside the TurtlebotBurger robot task, and only the parts that are necessary to search for color papers are created for Ev3Robot. If a leader exists in the team, the leader candidate robotic task should include a leader selection algorithm and periodically check the leader. Figure 5.7 shows that there are additional states that check the leader and separates what the leader or the rest needs to do in the

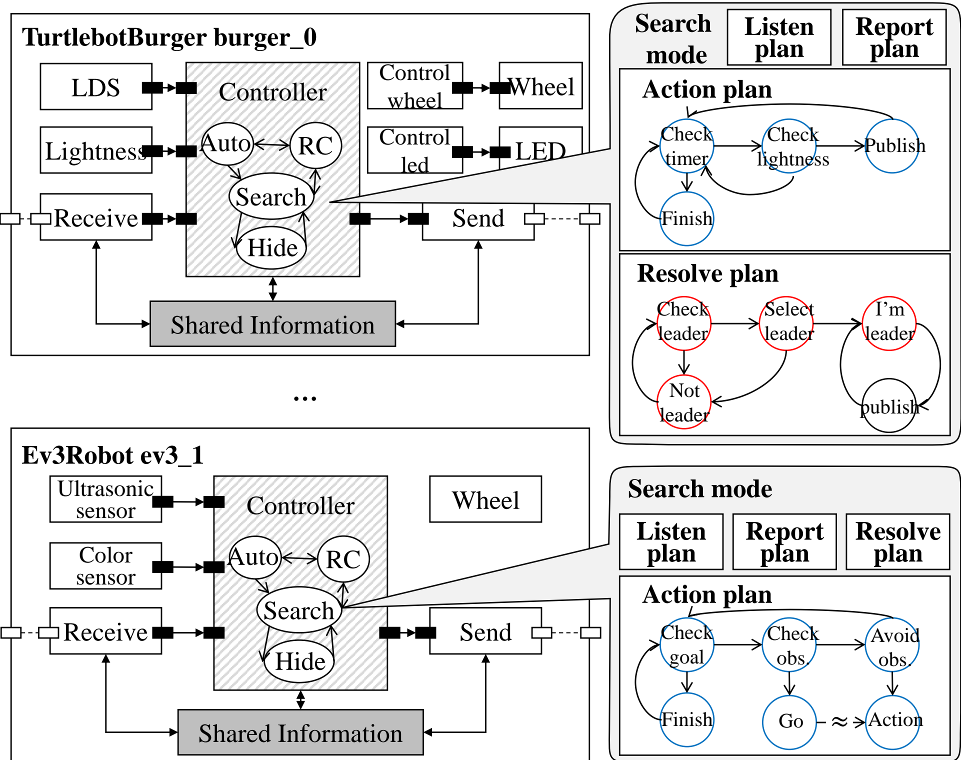


Figure 5.7: Automatic generation of task graph from mission specification and strategy description

"Resolve" plan.

Communication between robots can also be extracted from the communication statements in the mission specification. For information sharing, getter and setter functions can be automatically generated in a library task. The function that handles shared information is manually defined in the current implementation.

5.4 Related works

Several graphical languages have been proposed for easy specification of robot application for casual users [132], [133]. They typically use block diagram style of programming where programming is done by assembling blocks of pre-defined functions. Even if they are easy to learn and use, its expression capability is quite limited; it cannot specify a cooperative mission nor dynamic mode change.

Another popular approach is to extend existing scripting languages. For example, JavaScript has been extended to a variety of languages, such as RosLib.js [134], Cylon.js [135], and Johny-five [136], because it is easy to specify an application for event-based systems and to work with the web. Since event-based programming is a general programming paradigm, these languages can be used for robot application programming. Like ROS supports various languages such as Python, Java, and C++, RosLib.js [134] provides a library of JavaScript interfaces for ROS. Cylon.js [135] adds extra commands such as 'every' and 'after' to specify the periodic behavior. It helps the user can specify the behavior of the Internet of Things (IoT) device or robot more intuitively.

Dolphin [137] is a programming language that extends Groovy for autonomous vehicle networks. It assumes a centralized program, allowing a human to orchestrate an entire network of vehicles based on a global specification. Since the high-level abstraction helps the user to write a program without in-depth technical knowledge, Dolphin is included in Groovy as a domain-specific language. This work is similar to the proposed methodology in that it is easy for users to use with scripting languages, but it does not support swarm robotics.

The language called Buzz [40], which is a Lua extension, takes into consideration the characteristics of robots that repeat the behavior of 'sensing → communication → judgment/control → communication → actuation.' And it supports heterogeneous robots. Similar to our research, Buzz has taken both a top-down

approach and a bottom-up approach. Buzz includes several constructs designed explicitly for top-down swarm-level development, such as primitives for group formation and management, local communication, and global consensus. It can group many robots into one team and specify the action of each team, similarly to our proposed scripting language. Buzz is also designed to work with small systems. Since Buzz allows seamless mixing of bottom-up and top-down constructs in one language, it is difficult for novice programmers to use.

A general-purpose scripting language such as JavaScript and Lua is not easy to use for casual users who have little knowledge of computer programming. Moreover, multitasking is not supported in conventional scripting languages in general since a scripting language is interpreted at run-time. On the other hand, our scripting language specifies multi-tasking, dynamic mode change, and robot collaboration directly with corresponding keywords and syntax, which makes it different from general scripting languages that may express those behaviors with more programming efforts.

There exist some studies that have developed new languages for robots. As an example, MALLET [138] uses a team-oriented programming method including team intelligence, similar to human teamwork. It provides sequential, iterative, and conditional statements as well as parallel statements regarding control aspects. And the MALLET parser transforms into PrT nets (specialized Petri-Nets), which is similar to translating from mission to task graph model in our SeMo Framework. Also, there are four languages SDL, DDL, TDL, and MDL [139] to specify the robot behavior in detail. However, it is not easy to learn since it requires a user to learn four different languages to express the robot behavior up to code level.

Proto [140] is a functional programming language for homogeneous robots. It suggests a primitives library for group behaviors such as flock, scatter, disperse,

and cluster-by [141]. It also provides five constructs, including behavior assignment to groups and activation of the group action [141]. The program specifies a swarm-like behavior and neighborhood-based computation. It is compiled to an abstract bytecode that is deployed using a viral propagation mechanism over the network and then executed by each robot in a distributed manner. Bytecode execution uses a stack-based virtual machine for programs that may run on very lightweight microprocessor chips. There are several studies that extend Proto. While Proto is cumbersome because it is a functional language and has LISP-like syntax, Protelis [142] is a more recent Proto-based language built into Java. And Protoswarm [143] extends to program the swarm of robots. They focus on the swarm robotics mission that is performed by homogeneous robots, which is different from the proposed methodology.

There are several studies that use LTL (linear temporal logic) or its extension for the motion planning of autonomous robots [144], [145], and [146]. With LTL specification, they could verify the correctness of a motion plan formally and synthesize the task codes automatically. We believe that this approach is complementary to our work since the generated task code can be understood as a service definition that can be provided in a task or a task subgraph in the SeMo framework.

As a summary of related works, we compare the proposed one with some selected frameworks in terms of the following characteristics in Table 5.1: 1) development approach, 2) the orchestration type, 3) supporting type of cooperation, and 4) support of dynamic task allocation.

The development approach can be a top-down approach, a bottom-up approach, or a mixture approach. While most studies are using either the top-down approach or the bottom-up approach, Buzz and our proposed work take two approaches. Among them, several studies use two or more robots for a cooperative

Table 5.1: Framework for programming multiple robots

Framework	**Approach**	**Orchestration**	**Cooperation type**	**Dynamic allocation**
ROS	Bottom-up	Centralized	Individual/ Distributed	No
Karma	Top-down	Centralized	Swarm	Yes
Dolphin	Top-down	Centralized	Distributed	No
Proto	Top-down	Decentralized	Swarm	Yes
Buzz	Mixture	Centralized & Decentralized	Swarm	Yes
Ours	Mixture	Centralized & Decentralized	Both	Yes

mission, but ROS is focusing on individual robots [92]. It is challenging to specify a collaborative mission for robots [36], because there are no specific APIs defined for a cooperative mission. Since ROS is based on a central node called *rosmaster* that provides naming and registration services for the rest of the nodes to discover one another, it is known that a robot may get disconnected in multi-robot systems due to unreliable network [37]. Thus, using one master inside each reliable network, which is typically one master per robot, is taken as a solution [93].

The orchestration indicates whether multiple robots are centrally managed or not. Many studies, even the latest research [137], manage robots centrally, assuming that human intervention and control are necessary for robots because swarm robots contain a large number of robots. ROS is classified as centralized in terms of multiple robots because communication between nodes (robots) is made through the name server called ros master. However, Buzz and SeMo consider both orchestrations.

In the case of cooperation type, most of the researches only consider swarm robots but do not take into account distributed robots. Lastly, the dynamic task

Table 5.2: Comparison with other languages

-	**Python**	**Protoswarm**	**Dolphin**	**Buzz**	**Ours**
Number of Keywords	31	28	61	64	38
Example 1	-	11	22	43	16
Example 2	-	19	24	43	11
Example 3	-	20	22	47	21
Example 4 (Service)	-	-	-	30	3

allocation represents robots allocate groups dynamically depending on the environments or situations, which is closely related to robustness.

With regard to the lines of code in Table 5.2, we compare our mission script with other languages. The number of keywords is shown at the top of the table. Buzz and Dolphin are extensions of existing languages, so they have a relatively large number of keywords. On the other hand, new language Protoswarm and our mission script have a few keywords. Moreover, we compare the number of lines when representing the same example in each language. Example 1, 2, and 3 are simple scenarios taken from [141]. In Example 1, there are two teams, a red team and a blue team. The red team proceeds to the blue team, and then the blue team runs away when they find the incoming red team. The second example is about deployment, and the third example is more complex than other scenarios. One team patrols to find another team. When they detect, they are scattered. Since Buzz is based on the bottom-up approach, the lines of code are long. Example 4 is only expressing the service in the Buzz example. Our approach uses abstracted services so that the user can do the same with terse lines of code.

5.5 Experiments

To examine the viability of the proposed methodology, preliminary experiments are conducted with a V-Rep robot simulator [34] and real robots. A simple swarm robotics example is demonstrated in the V-Rep robot simulator in Section 5.5.1. And in Section 5.5.2, real robots are used to demonstrate some cooperative examples including the example explained in Section 3.1.3.

5.5.1 Swarm Robotics Example

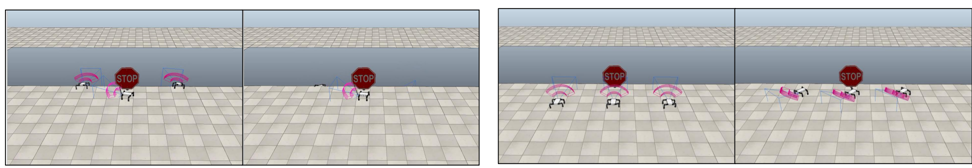
In this example, several 4-leg biomimetic robots that are equipped with a proximity sensor and a camera are marching in line, which is defined as a group service in the mission specification. Figure 5.8 (a) is a snippet of mission specification. When a robot detects an obstacle, it shares this knowledge with neighboring robots to stop marching and turn back. Without knowledge sharing, the other robots would crash the obstacle shown in Figure 5.8 (b). The proposed framework, however, generates the code with knowledge sharing correctly to avoid crashing. And we change the number of robots from 3 to 10 to test the scalability in swarm robotics. Since the detailed robot movement takes long in the V-Rep robot simulator, we used a small number of robots in this experiment. Some snapshots are shown in Figure 5.8 (c) and Figure 5.8 (d).

5.5.2 Scouting Mission with Heterogeneous Robots

To prove the viability of the proposed methodology, preliminary experiments are conducted with a scenario of the cooperative mission. The scenario verifies the refinement process from the mission script to task graphs. This experiment confirms the benefits of the modular structure of the SeMo framework; we may change mission, strategy easily without laborious manual coding.

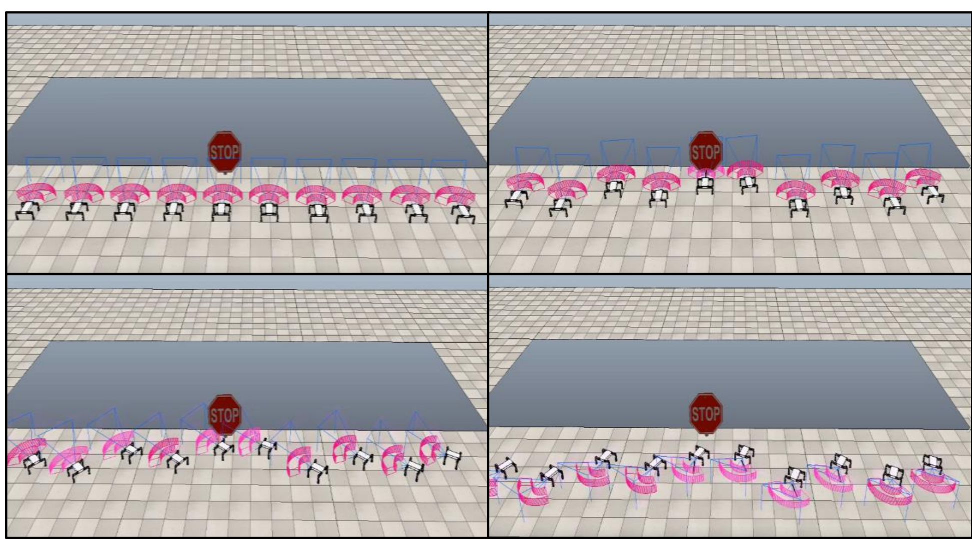
```
1  { Team : VRepQuadruptor quad[3] }
2  Team.Action.Move{
3    march()
4  } repeat()
5  Team.Report.LookAround {
6    if (obstacle == true)
7      publish(Team, Team.Message = suggestAvoiding)
8  } repeat ()
```

(a) A part of mission script example



(b) Snapshot of 3 robots without knowledge sharing

(c) Snapshot of 3 robots with knowledge sharing



(d) Snapshot of 10 robots

Figure 5.8: Quadruped robots share information to avoid falling

5.5.2.1 Scenario 1

In this scenario, a team of robots has a mission to follow a specific object. Initially, the robots run in the remote control mode where their movement is

```
1  Team1.Listen. Comm_operator{
2    receive(user, user.cmd) } repeat (1 sec)
3  Team1.Action.RemoteControl{
4    process (user.cmd) } repeat (1 sec)
5  Team1.RemoteControlMode{
6    set(Listen, Comm_operator)
7    set(Report, Broadcast)
8    set(Action, RemoteControl) }  ….
```

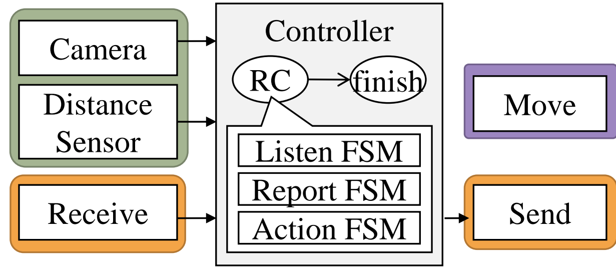
(a) Only remote control mode

```
1   Team1.Action.RemoteControl{
2     process (user.cmd)
3     if (distance < threshold) throw auto                // add condition
4   } repeat (…)
5   Team1.Action.AutonomousMovingMode {// new composite service
6     follow (object) } repeat (…)
7   Team1.RemoteControlMode{
8     set(Listen, Comm_operator)
9     set(Report, Broadcast)
10    set(Action, RemoteControl) }
11  Team1.AUTO_MODE {                                     // additional mode
12    set(Listen, OFF); set(Report, OFF)
13    set(Action, AutoFollow) }  ….
```

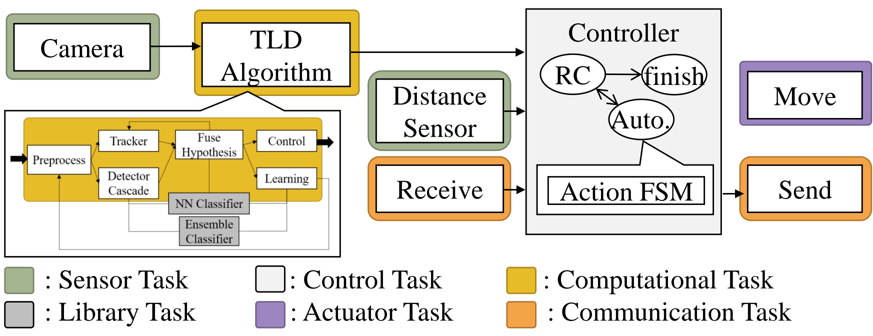
(b) Add autonomous moving mode

Figure 5.9: Change of the mission script

controlled by an operator manually, and a part of the associated mission script is displayed in Figure 5.9 (a). In the SeMo framework, the mission script is translated into a task graph that represents the internal behavior of a robot, as shown in Figure 5.10 (a) where the controller task is triggered by the events received by the communication module (”Receive”). Suppose the user wants to modify the mission scenario in which if the distance between a robot and the object is close enough,



(a) Remote control mode behavior specification



(b) Autonomous moving mode behavior specification based on the TLD algorithm

Figure 5.10: Task graph specification for two different modes of operation

they follow the object automatically. The changed scenario is the same as the motivational example in Section 3.1.1. It can be done by adding an *autonomous moving mode* in the mission script to follow the object as displayed in Figure 5.9 (b). In addition, we describe which algorithm is to use for autonomous moving in the strategy description. In this experiment, the TLD algorithm [147] is used for object tracking. Then the framework synthesizes the new task graph, as shown in Figure 5.10 (b), where a task subgraph that specifies the TLD algorithm task is added and the controller task is modified adding an *autonomous moving mode*. Remind that the task graph specification of the TLD algorithm is assumed to be given and registered as a service a priori by the robot manufacturer or a knowledgeable user. Figure 5.11 is some snapshots.

Table 5.3 compares the line of codes user has written with the generated code. If the robot are using only remote control mode, only 39 lines of mission

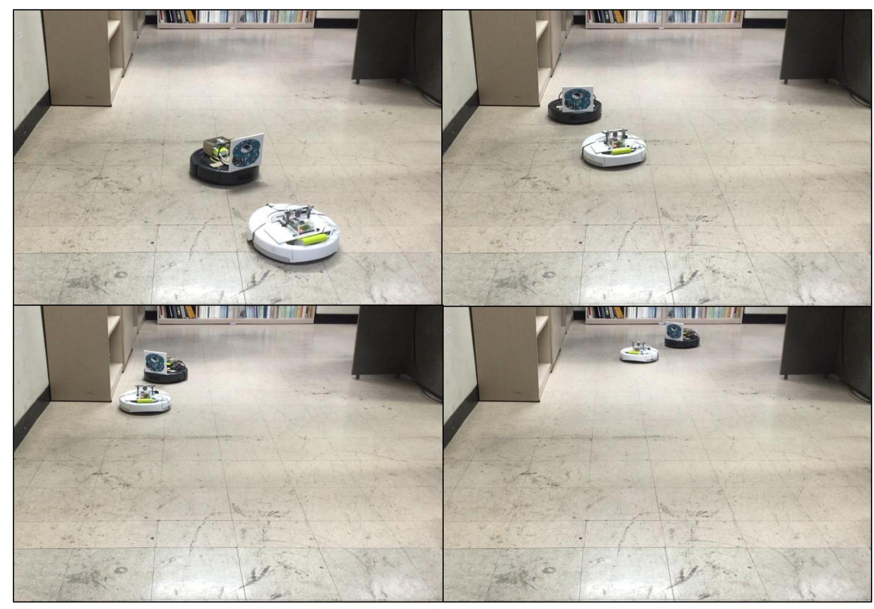


Figure 5.11: Robot to follow a specific robot

Table 5.3: Lines of the mission scripting, strategy, and task code

		Only RC mode	**Adding auto mode**
Mission		39	60
Strategy		123	172
Task graph code	Task code	375	2,591
	Control code	198	271
	Communication code	52	52
	Total code	625	2,914
Mission / Total task code (%)		6.24	2.06
Mission + Strategy / Total task code (%)		25.92	7.96

script are represented. After modifying the scenario, the user needs to write 20 more lines to add autonomous driving mode, which result in generating more than 2,900 lines of task code. Therefore, the modularity allows user to easily perform various scenarios with the help of our methodology, even with minor changes.

5.5.2.2 Scenario 2

For the scouting mission of Section 3.1.3, three different types of robots are used: one iRobotCreate [148], two TurtleBot3 Burgers [149], and two Ev3Robots [150]. The iRobotCreate is controlled by a Raspberry pi board with Ubuntu. TurtleBot3 Burger is equipped with a laser distance sensor, a light sensor, an LED, two wheels, and a Raspberry Pi 3 Model B+ (ARM Cortex-A53, quad-core, 1.4GHz) as a single-board computer and OpenCR 1.0 board [151](ARM Cortex-m7, 216MHz) as a micro-controller. Ev3Robot has a color sensor, a distance sensor, two motors, and a brick(ARM926EJ-S, 300MHz). All robots communicate with each other using Wi-Fi.

To verify the robustness of the group service, we test a scenario where the leader robot in the SlaveTeam fails during the operation. Since the leader robot is in charge of communication with the MasterTeam, its failure will fail the mission.

Table 5.4: Lines of the generated code

		iRobot Create	**Burgers**		**Ev3Robot**
			Pi	OpenCR	
Mission		211			
Strategy		203	654		
Robot Code	Task Graph Code: Task code	278	213	128	240
	Task Graph Code: Control task	207	475	-	455
	Task Graph Code: Comm. task	155	270	-	196
	Scheduler	3,926	3,926	826	3,926
	Data structure	1,168	2,141	375	1,163
	API, wrapper, etc	15,182	16,309	5,022	15,869
	Total code	20,916	23,334	6,351	21,849
Manual code ratio (%) (Mission / Total code)		1.01	0.90	3.32	0.97
Manual code ratio (%) ((Mission + Strategy) / Total code)		1.98	3.70	13.62	3.96

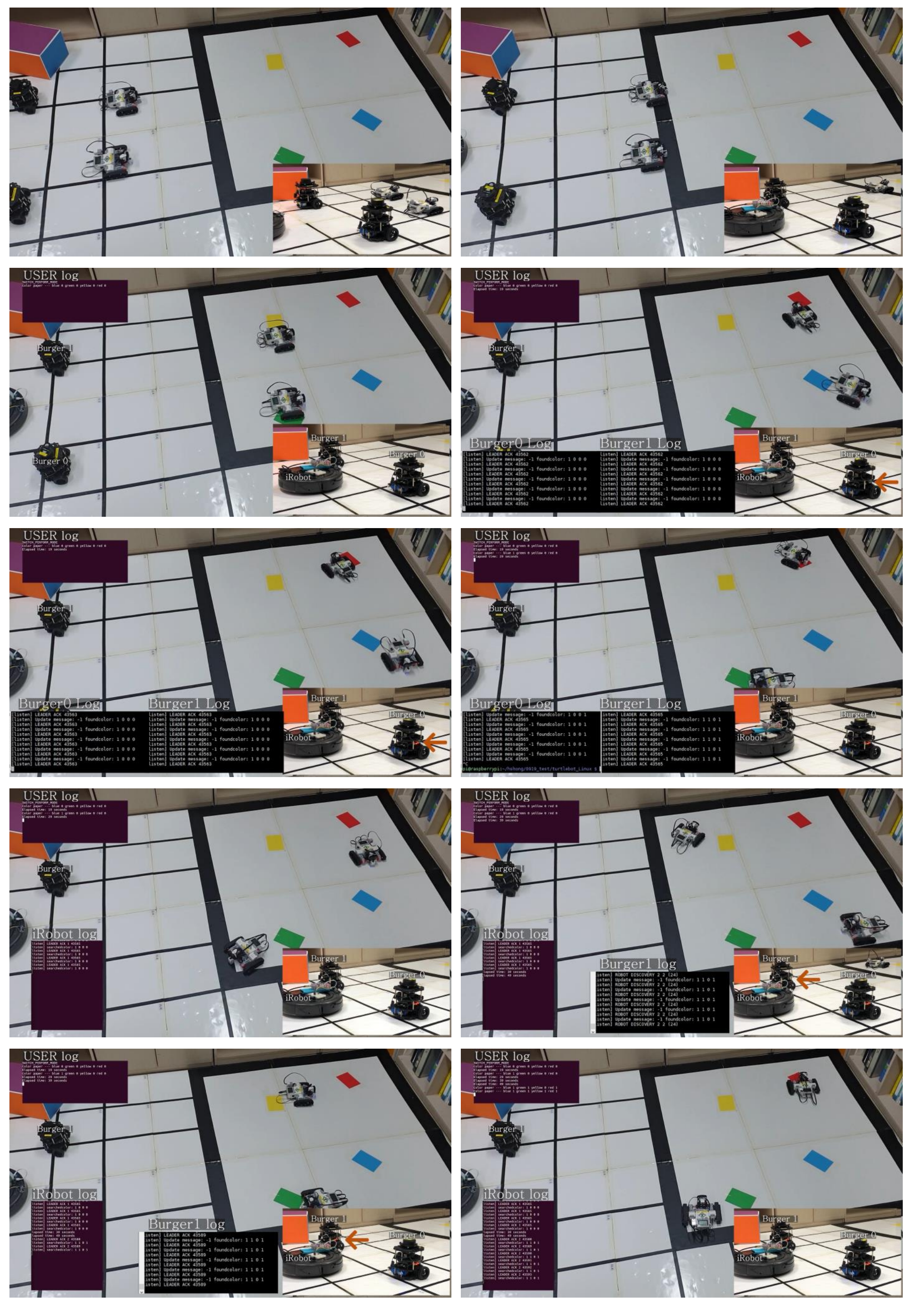


Figure 5.12: Heterogeneous robots work together to find colored paper

However, the proposed framework selects a new leader after detecting the failure of the leader by periodic monitoring and completes the mission correctly. This experiment also confirms that we generate the target codes of different robots automatically from the translated task graph. Table 5.4 shows the number of lines in the generated code. The scenario represented by only 211 lines in the mission script language is converted into control task code, communication task code, and task code associated with the hardware component of the robot. For TurtleBot3 Burger, two wheels, LED, and brightness sensor are connected to the OpenCR board, so there are no additional control task and communication tasks. Only serial communication code is added to transfer the value of each task to the control task in the Raspberry Pi board. The generated actual robot code contains data structures for tasks, channels, and libraries, and target specific code for initialization and wrapup actions. In addition, target-independent APIs used in the task graph are redefined as target-specific APIs.

5.5.2.3 Scenario 3

The low manual code ratio does not mean that the code generator is effective because unnecessary lines of code reduce the manual code rate. To verify the code quality, we compare the code lines between ROS and our methodology. We implemented a simple remote control mission scenario using an ”iRobot Create”. Table 5.5 displays the lines of code at each step. We wrote about 430 lines of code to develop all ROS node code themselves. On the other hand, the proposed methodology requires less programming but generates about 530 lines of code automatically. This is because the control task consists of a hierarchical finite state machine. Even if the robot performs simple action, the nested structures and duplicated structures make the line of code increase like Figure 5.13.

Compared to the overall software code, the ROS code line and SeMo code line

Table 5.5: Lines of the code

			ROS	Our proposed framework
Mission			-	38
Strategy			-	123
Robot Code	Task Graph Code	Task code	436	215
		Control task	-	198
		Comm. task	-	52
		Total task code	436	530
		Manual rate (%)	**100**	**30.38**
	Scheduler		3,194	3,926
	Data structure		883	854
	API, wrapper, etc		22,720	15,182
	Total code		27,233	20,492
Manual rate (%) (Manual/Robot code)			**1.60**	**0.78**

are similar. To execute the ROS node, the relevant code for topics and services is automatically generated and run by the ros master.

As a result, ROS requires more programming than our methodology.

```
1   TASK_INIT {… }
2   int service_go_forward() {
3     switch( current_state_go_forward){
4       case ID_STATE_GO_FORWARD_GOFORWARD: {
5         UFTask_SetIntegerParameter ("wheel", "cmd", CMD_FWD);
6         UFControl_CallTask("wheel");
7         break;
8       }
9     } …
10  }
11  int service_go_backward() {
12    switch( current_state_go_backward){
13      case ID_STATE_GO_BACKWARD_GOBACKWARD: {
14        UFTask_SetIntegerParameter ("wheel", "cmd", CMD_BWD);
15        UFControl_CallTask("wheel");
16        break;
17      }
18    } …
19  } …
20  void execute_Action_Move(){
21     switch(current_state_Action_Move){
22       case ID_STATE_Action_Move_if1: … break;
23       case ID_STATE_Action_Move_action2:
24         int result = service_go_forward(); … break; …
25  }
26  TASK_GO {
27    switch(current_mode){
28      case ID_STATE_RC_MODE:
29          execute_Action_Move();
30  …}
31  TASK_WRAP { … }
```

Figure 5.13: Control task code of remote control mission

Chapter 6

Conclusion

In this dissertation, we proposed a novel service-oriented and model-based software development framework to support distributed robot systems, swarm robots, and their hybrid. The proposed SeMo framework separates the high-level mission specification with an easy-to-use scripting language and the model-based task graph specification for algorithm-level behavior specification of each robot. The overall flow of the proposed software development methodology can be understood as the refinement process among four levels of abstraction in software development.

The first step is *mission specification* at the highest level of abstraction with a scripting language. It can express team configuration, service-oriented programming, dynamic mode change of operation, and multitasking. To our best knowledge, no existent script language has such expression capability [138], [133], [152] with easy-to-use syntax. The function each robot can perform is abstracted with a *service* in our framework, and a user requests a service by its name. We also allow a user to define a *composite service* that executes the primitive services sequentially. Multitasking is not easy to express in popular script languages like python or lua. To support multitasking, we adopt the notion of *plan* from [131]; multitasking of a robot is represented by mapping one composite service per plan.

A robot may have multiple operating modes depending on the environments and user requests. The generated event in the composite service can trigger a mode transition. In addition, to improve the robustness, scalability, and flexibility of robot collaboration, we extend the high-level mission specification by adding new features such as team hierarchy, group service, and one-to-many communication.

The second step is *strategy description* that defines the second level of abstraction. It provides more information on how to perform services. A service written in the mission can have different algorithms depending on various conditions and requirements. For service refinement, we describe the conditions and requirements for selecting the appropriate algorithm for each service. Non-functional requirements can be added at this step as well.

The next step is *task graph specification* that depicts the internal behavior of each robot to perform the mission. Unlike mission specification, we assume that the internal definition of a task is developed by a professional programmer. It is abstracted as a service function that a robot can perform. Recently compute-intensive services such as vision and machine learning are getting popular in robots. A compute-intensive service can be specified by a task subgraph that can be mapped to multiple processors for parallel processing.

To analyze the system behavior at compile-time, we apply a formal task graph model, extended from synchronous dataflow (SDF) [113] model for task graph specification. The SDF model that defines formal semantics for inter-task communication and task execution conditions allows us to make the task scheduling decision at compile-time and estimate the performance and resource requirements. Our extended model uses a finite state machine (FSM) to represent dynamic behavior [116]. In addition, we support shared information exchange between robot platforms. Two types of information sharing, global information shared and local knowledge sharing, are supported for robot collaboration in the dataflow graph.

For global information, we use a special type of task, called library task, to manage shared resources [42] among multiple robots. On the other hand, we extend the multicast port and adopt a knowledge sharing technique for local knowledge sharing.

The actual robot code per robot is automatically generated from the associated task graph, which minimizes the human efforts in low-level robot programming and improves the software design productivity significantly.

The viability of the proposed methodology is verified with preliminary experiments with three cooperative mission scenarios with three different robot platforms.

6.1 Future Research

Some extensions based on the proposed methodology can be identified to require further research. Some of these are:

- Support of automatic design method. We have been studying the mission specification that allows the user to describe the robot tasks in detail. In recently automated design methodologies, robots can learn on their own using deep learning technology when the user tells them only the desired goals.

- Verbal mission specification and graphical strategy description. Since the mission scripting language is simple, voice recognition and automatic language translation technique can be adopted to relieve the burden of learning the mission scripting language. Besides, developing a graphical interface for an easy strategy description will be pursued. It may be a good idea to get feedback from people who are not familiar with programming.

- Security-aware mission specification. The information shared by robots as

they perform their missions often requires security. Depending on the level of security, only limited robots should be able to access the information or to perform the specific services.

- Librarization to improve scalability. The written mission scripts can be registered with the library and can be called and used later by the user. In addition, even if robots are already on duty, it should be possible to add robots as needed to work with existing robots to fulfill their mission.

- Management of robots' work to improve robustness. In the current implementation, since the progress of each robot is not individually backed up and managed, it is not possible to take over the work so far if there is a problem with the robot.

- Efficient code generation considering non-functional requirements. We can consider non-functional elements as very rudimentary in the current implementation. If we add the adaptation policy manager in the generated code, you will be able to extend the robot's mission time more dynamically.

- Optimized code generation. We focus on auto-generated software code that runs functionally well. However, when the generated code is optimized as if written by hand, it would be a much more attractive methodology.

- Comparing the performance to ROS and other frameworks. Even though there are no benchmarks to compare performance, we did not compare any overhead, such as the communication time between tasks.

- Applying various kinds of robots and scenarios. Even if there are no benchmarks, we conducted experiments using case studies.

Bibliography

[1] Hyesun Hong, Hanwoong Jung, Kangkyu Park, and Soonhoi Ha. Semo: Service-oriented and model-based software framework for cooperating robots. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2952–2963, 2018.

[2] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. Hierarchical dataflow modeling of iterative applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 39. ACM, 2017.

[3] RenderHub. Industrial robot 3d model by weeray, 2017.

[4] Danbach Robot Jiangxi Inc. Dbh six axis industrial robot by danbach robot jiangxi inc.

[5] Robotics & Automation News. Pepper robot from soft bank robotics group corp., 2018.

[6] iRobot Corporation. irobot roomba 900 series, 2019.

[7] Day web chronicle. Business analytics of entertainment robots market to see excellent growth by 2025, 2019.

[8] SDR Tactical Robots. Lt2/f bulldog with a multi-axis robotic arm, 2017.

[9] Tim Moynihan. Davinci robots from google and johnson & johnson, 2015.

[10] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter. Bigdog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, 41(2):10822–10825, 2008.

[11] Aaron M Hoover, Samuel Burden, Xiao-Yu Fu, S Shankar Sastry, and Ronald S Fearing. Bio-inspired design and dynamic maneuverability of a minimally actuated six-legged robot. In *2010 3rd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 869–876. IEEE, 2010.

[12] Hayato Omori, Takeshi Hayakawa, and Taro Nakamura. Locomotion and turning patterns of a peristaltic crawling earthworm robot composed of flexible units. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1630–1635. IEEE, 2008.

[13] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions*, volume 1, pages 59–65. IPCB: Instituto Politécnico de Castelo Branco, 2009.

[14] Ill-Woo Park, Jung-Yup Kim, Jungho Lee, and Jun-Ho Oh. Mechanical design of humanoid robot platform khr-3 (kaist humanoid robot 3: Hubo). In *5th IEEE-RAS International Conference on Humanoid Robots, 2005.*, pages 321–326. IEEE, 2005.

[15] Robert J Wood. The first takeoff of a biologically inspired at-scale robotic insect. *IEEE transactions on robotics*, 24(2):341–347, 2008.

[16] Stanley S Baek, Fernando L Garcia Bermudez, and Ronald S Fearing. Flight control for target seeking by 13 gram ornithopter. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2674–2681. IEEE, 2011.

[17] Kevin Peterson and Ronald S Fearing. Experimental dynamics of wing assisted running for a bipedal ornithopter. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5080–5086. IEEE, 2011.

[18] Antonio Gomes, Calvin Rubens, Sean Braley, and Roel Vertegaal. Bitdrones: Towards using 3d nanocopter displays as interactive self-levitating programmable matter. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 770–780. ACM, 2016.

[19] DJI. Phantom 3 standard drone, 2015.

[20] Shumei Yu, Shugen Ma, Bin Li, and Yuechao Wang. An amphibious snake-like robot with terrestrial and aquatic gaits. In *2011 IEEE International Conference on Robotics and Automation*, pages 2960–2961. IEEE, 2011.

[21] Linda Dailey Paulson. Biomimetic robots. *Computer*, 37(9):48–53, 2004.

[22] Alessandro Crespi, Daisy Lachat, Ariane Pasquier, and Auke Jan Ijspeert. Controlling swimming and crawling in a fish robot using a central pattern generator. *Autonomous Robots*, 25(1-2):3–13, 2008.

[23] Florian Shkurti, Anqi Xu, Malika Meghjani, Juan Camilo Gamboa Higuera, Yogesh Girdhar, Philippe Giguere, Bir Bikram Dey, Jimmy Li, Arnold Kalmbach, Chris Prahacs, et al. Multi-domain monitoring of marine environments using a heterogeneous robot team. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1747–1753. IEEE, 2012.

[24] Je-Sung Koh, Eunjin Yang, Gwang-Pil Jung, Sun-Pill Jung, Jae Hak Son, Sang-Im Lee, Piotr G Jablonski, Robert J Wood, Ho-Young Kim, and Kyu-Jin Cho. Jumping on water: Surface tension–dominated jumping of water striders and robotic insects. *Science*, 349(6247):517–521, 2015.

[25] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 404–411. IEEE, 2011.

[26] Paul Birkmeyer, Kevin Peterson, and Ronald S Fearing. Dash: A dynamic 16g hexapedal robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, pages 2683–2689. IEEE, 2009.

[27] Yufeng Chen, Hongqiang Wang, E Farrell Helbling, Noah T Jafferis, Raphael Zufferey, Aaron Ong, Kevin Ma, Nicholas Gravish, Pakpong Chirarattananon, Mirko Kovac, et al. A biologically inspired, flapping-wing, hybrid aerial-aquatic microrobot. *Science Robotics*, 2(11):eaao5619, 2017.

[28] Kazuo Hirai, Masato Hirose, Yuji Haikawa, and Toru Takenaka. The development of honda humanoid robot. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, volume 2, pages 1321–1326. IEEE, 1998.

[29] Pololu Corporation. Pololu: https://www.pololu.com/docs/0j62, 2007.

[30] HumanRobotics. Pob robotics-pobtools, ri2bee: http://pobtools.software.informer.com, 2017.

[31] Parallax Inc. Parallax-propbasic: http://developer.parallax.com/propelleride, 2016.

[32] Parallax Inc. Parallax-spin: http://learn.parallax.com/tutorials/projects/christmas-scribbler/how-mixing-gui-and-spin-code-works, 2018.

[33] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[34] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

[35] Ricardo Tellez, Alberto Ezquerro, and Miguel Angel Rodriguez. Ros navigation in 5 days: Entirely practical robot operating system training. 2017.

[36] Lee Garber. Robot os: A new day for robot design. *Computer*, 46(12):16–20, 2013.

[37] Zhi Yan, Luc Fabresse, Jannik Laval, and Noury Bouraqadi. Building a ros-based testbed for realistic multi-robot simulation: taking the exploration as an example. *Robotics*, 6(3):21, 2017.

[38] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[39] Phillip Walker, Saman Amirpour Amraii, Michael Lewis, Nilanjan Chakraborty, and Katia Sycara. Control of swarms with multiple leader agents. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3567–3572. IEEE, 2014.

[40] Carlo Pinciroli and Giovanni Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3794–3800. IEEE, 2016.

[41] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE TSP*, 49(10):2408–2421, 2001.

[42] Hae-woo Park, Hanwoong Jung, Hyunok Oh, and Soonhoi Ha. Library support in an actor-based parallel programming platform. *IEEE Transactions on Industrial Informatics*, 7(2):340–353, 2011.

[43] Oxford University Press, editor. *Oxford English Dictionary*, chapter robot. 2015.

[44] International Federation of Robotics. *World Robotics Industrial Robots*. 2018.

[45] International Federation of Robotics. *World Robotics Service Robots*. 2018.

[46] H.W. Chun. Defense, it convergence: Focus on military robots. *2013 Electronics and Telecommunications Trends*, pages 107–117, 2013.

[47] Steven H Collins and Andy Ruina. A bipedal walking robot with efficient and human-like gait. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 1983–1988. IEEE, 2005.

[48] Scott A Bortoff. Path planning for uavs. In *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No. 00CH36334)*, volume 1, pages 364–368. IEEE, 2000.

[49] Jisu Kim Hyungbo Shim Dong-gil Lee, Hyemin Lim. Experiment of improving dash robot direct motion by eliminating the input disturbance via feedback control. In *한국군사과학기술학회 종합학술대회*, 2015.

[50] Joo-Hee Lee Gi-Hyuk Choi Eun-Sup Sim We-Sub Eom, Youn-Kyu Kim. Development trend of intelligent robots. *항공우주산업 기술 동향*, 11-1:150–160, 2013.

[51] Stefan Waldherr, Roseli Romero, and Sebastian Thrun. A gesture based interface for human-robot interaction. *Autonomous Robots*, 9(2):151–173, 2000.

[52] Markus Kohler. Vision based hand gesture recognition systems. *University of Dortmund, Website, Retrieved July*, 2005.

[53] Beom-Jin Lee, Jinyoung Choi, Chung-Yeon Lee, Kyung-Wha Park, Sungjun Choi, Cheolho Han, Dong-Sig Han, Christina Baek, Patrick Mokodir Emaase, and Byoung-Tak Zhang. Perception-action-learning system for

mobile social-service robots using deep learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] 조인혜. 군집로봇이 뜨는 이유. *ScienceTimes*, 2017.

[55] David Portugal, Luca Iocchi, and Alessandro Farinelli. A ros-based framework for simulation and benchmarking of multi-robot patrolling algorithms. In *Robot Operating System (ROS)*, pages 3–28. Springer, 2019.

[56] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[57] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[58] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Eric Bonabeau, and Guy Theraula. *Self-organization in biological systems*. Princeton university press, 2003.

[59] Jennifer Scott in BBC News. Could drones replace fireworks in the uk?, 2018.

[60] Jonathan C Erickson, María Herrera, Mauricio Bustamante, Aristide Shingiro, and Thomas Bowen. Effective stimulus parameters for directed locomotion in madagascar hissing cockroach biobot. *PloS one*, 10(8):e0134348, 2015.

[61] Alireza Dirafzoon and Edgar Lobaton. Topological mapping of unknown environments using an unlocalized robotic swarm. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5545–5551. IEEE, 2013.

[62] Tom Herbert in METRO. Games opened with 1,218 drones forming incredible olympic rings, 2018.

[63] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[64] Justin Werfel, Kirstin Petersen, and Radhika Nagpal. Designing collective behavior in a termite-inspired robot construction team. *Science*, 343(6172):754–758, 2014.

[65] Matt Simon in WIRED. Inside the amazon warehouse where humans and machines become one, 2019.

[66] L Breton, P Hughes, S Barker, M Pilling, L Fuente, and NT Crook. The impact of leader-follower robot collaboration strategies on perceived safety and intelligence. 2016.

[67] Andreas Kolling, Phillip Walker, Nilanjan Chakraborty, Katia Sycara, and Michael Lewis. Human interaction with robot swarms: A survey. *IEEE Transactions on Human-Machine Systems*, 46(1):9–26, 2015.

[68] Jakob Fredslund and Maja J Mataric. A general algorithm for robot formations using local sensing and minimal communication. *IEEE transactions on robotics and automation*, 18(5):837–846, 2002.

[69] Xun S Zhou and Stergios I Roumeliotis. Multi-robot slam with unknown initial correspondence: The robot rendezvous case. In *2006 IEEE/RSJ international conference on intelligent robots and systems*, pages 1785–1792. IEEE, 2006.

[70] Ross Mead, Jerry B Weinberg, and Jeffrey R Croxell. A demonstration of a robot formation control algorithm and platform. In *22nd National Conference on Artificial Intelligence, Vancouver, BC*, 2007.

[71] Kwang-Kyo Oh, Myoung-Chul Park, and Hyo-Sung Ahn. A survey of multi-agent formation control. *Automatica*, 53:424–440, 2015.

[72] Eduardo Montijano, Eric Cristofalo, Dingjiang Zhou, Mac Schwager, and Carlos Saguees. Vision-based distributed formation control without an external positioning system. *IEEE Transactions on Robotics*, 32(2):339–351, 2016.

[73] Javier Alonso-Mora, Stuart Baker, and Daniela Rus. Multi-robot navigation in formation via sequential convex programming. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4634–4641. IEEE, 2015.

[74] Hai Zhu and Javier Alonso-Mora. Chance-constrained collision avoidance for mavs in dynamic environments. *IEEE Robotics and Automation Letters*, 4(2):776–783, 2019.

[75] Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34(4):856–869, 2018.

[76] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative monocular slam with multiple micro aerial vehicles. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3962–3970. IEEE, 2013.

[77] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3863–3870. IEEE, 2017.

[78] Jean J. Labrosse. *uC/OS-3 User's Manual*. Micrium Press, 2016.

[79] Inc Wind River Systems. Vxworks 7 datasheet, mar 2019.

[80] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[81] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*, pages 455–462. IEEE, 2004.

[82] Kris Pister Sarah Bergbreiter, Jameson Lee. Cotsbots software, oct 2006.

[83] Contiki. Contiki 2.6 documents, jul 2012.

[84] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *RAM*, pages 736–742, 2008.

[85] Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.

[86] J-C Baillie. Urbi: Towards a universal robotic low-level programming language. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 820–825. IEEE, 2005.

[87] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[88] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[89] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[90] Gilberto Echeverria, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. Simulating complex robotic scenarios with morse. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 197–208. Springer, 2012.

[91] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[92] SangYoung Park and GuiHyung Lee. Mapping and localization of cooperative robots by ros and slam in unknown working area. In *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 858–861. IEEE, 2017.

[93] Sergi Hernandez Juan and Fernando Herrero Cotarelo. Multi-master ros systems. *Institut de Robotics and Industrial Informatics*, pages 1–18, 2015.

[94] Choulsoo Jang, Seung-Ik Lee, Seung-Woog Jung, Byoungyoul Song, Rockwon Kim, Sunghoon Kim, and Cheol-Hoon Lee. Opros: A new component-based robot software platform. *ETRI journal*, 32(5):646–656, 2010.

[95] Herman Bruyninckx. Open robot control software: the orocos project. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2523–2528. IEEE, 2001.

[96] Patrick Ulam, Yoichiro Endo, Alan Wagner, and Ronald Arkin. Integrated mission specification and task allocation for robot teams-design and implementation. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4428–4435. IEEE, 2007.

[97] Douglas Blank, Lisa Meeden, and Deepak Kumar. Python robotics: An environment for exploring robotics beyond legos. In *ACM SIGCSE Bulletin*, volume 35, pages 317–321. ACM, 2003.

[98] N T. Dantam et al. A task-motion planning framework: http://tmkit.kavrakilab.org/, 2016.

[99] Stéphane Magnenat, Philippe Rétornaz, Michael Bonani, Valentin Longchamp, and Francesco Mondada. Aseba: A modular architecture for event-based control of complex robots. *IEEE/ASME Transactions on Mechatronics*, 16(2):321–329, 2011.

[100] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. Drona: A framework for safe distributed mobile robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 239–248. ACM, 2017.

[101] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[102] Kai Weng Wong and Hadas Kress-Gazit. Robot operating system (ros) introspective implementation of high-level task controllers. pages 65–77, 2017.

[103] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[104] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 121–134. ACM, 2011.

[105] Robomatter Inc. Robotc: http://www.robotc.net, 2014.

[106] Wikipedia Contributors. Robot, 2019.

[107] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[108] Hae-Woo Park, Hyunok Oh, and Soonhoi Ha. Multiprocessor soc design methods and tools. *IEEE Signal Processing Magazine*, 26(6), 2009.

[109] Hanwoong Jung. *Dynamic behavior specification and design space exploration for real-time embedded systems*. PhD thesis, Seoul National University, 2016.

[110] Shuvra S Bhattacharyya, Ed F Deprettere, Rainer Leupers, and Jarmo Takala. *Handbook of signal processing systems*. Springer, 2018.

[111] Stephen Edwards, Luciano Lavagno, Edward A Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[112] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[113] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[114] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for mpsoc. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):39, 2008.

[115] Soonhoi Ha and EunJin Jeong. Embedded software design methodology based on formal models of computation. In *Principles of Modeling*, pages 306–325. Springer, 2018.

[116] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):135, 2014.

[117] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A Peperstraete. Cyclo-static data flow. In *ICASSP*, volume 5, pages 3255–3258. IEEE, 1995.

[118] Hyunok Oh and Soonhoi Ha. Fractional rate dataflow model for efficient code synthesis. *IOSR-JVSP*, 37(1):41–51, 2004.

[119] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *SAMOS XIII*, pages 41–48. IEEE, 2013.

[120] Soonhoi Ha and Edward Lee. Compile-time scheduling of dynamic contructs in dataflow program graphs. *IEEE TC*, 46:768–778, 1997.

[121] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended synchronous dataflow for efficient dsp system prototyping. *DAES*, 6(3):295–322, 2002.

[122] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[123] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[124] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE IISWC*, pages 44–54. IEEE, 2009.

[125] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[126] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[127] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *IJCNN*, pages 593–605. IEEE, 1989.

[128] Jinwoo Kim, Minyoung Kim, Mark-Oliver Stehr, Hyunok Oh, and Soonhoi Ha. A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing. *Journal of Parallel and Distributed Computing*, 72(4):564–578, 2012.

[129] Wikipedia Contributors. Multicast, 10 2019.

[130] Ronald Eugene Kmetovicz. *New product development: design and analysis.* John Wiley & Sons, 1992.

[131] Enrique Fernández Perdomo, Jorge Cabrera Gámez, Antonio Carlos Domínguez Brito, and Daniel Hernández Sosa. Mission specification in underwater robotics. 2010.

[132] Mario Ferrari and Guilio Ferrari. *Building robots with LEGO Mindstorms NXT*. Syngress, 2011.

[133] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[134] Brandon Alexander, Kaijen Hsiao, Chad Jenkins, Bener Suay, and Russell Toris. Robot web tools [ros topics]. *IEEE Robotics & Automation Magazine*, 19(4):20–23, 2012.

[135] Andrew Stewart. Cylon.js: https://cylonjs.com, 2015.

[136] Rick Waldron. Johnny-five: http://johnny-five.io, 2017.

[137] Keila Lima, Eduardo RB Marques, José Pinto, and João B Sousa. Dolphin: a task orchestration language for autonomous vehicle networks. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 603–610. IEEE, 2018.

[138] Xiaocong Fan, John Yen, Michael Miller, Thomas R Ioerger, and Richard Volz. Mallet-a multi-agent logic language for encoding teamwork. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):123–138, 2006.

[139] Daniel Augusto Gama de Castro Silva et al. Cooperative multi-robot missions: development of a platform and a specification language. 2013.

[140] Jonathan Bachrach, Jacob Beal, and James McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, 2010.

[141] Jacob Beal, Kyle Usbeck, and Brian Krisler. Lightweight simulation scripting with proto. *2012) colocated with AAMAS (W21)*, page 1, 2012.

[142] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1846–1853. ACM, 2015.

[143] Jonathan Bachrach, James McLurkin, and Anthony Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1175–1178. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[144] Susmit Jha and Vasumathi Raman. Automated synthesis of safe autonomous vehicle control under perception uncertainty. In *NASA Formal Methods Symposium*, pages 117–132. Springer, 2016.

[145] Savvas G Loizou and Kostas J Kyriakopoulos. Automatic synthesis of multi-agent motion tasks based on ltl specifications. In *IEEE Conference on Decision and Control (CDC)*, volume 1, pages 153–158. IEEE, 2004.

[146] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 239–248. ACM, 2015.

[147] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1409–1422, 2012.

[148] iRobot. Create 2 open interface spec: https://www.irobot.com/about-irobot/stem/create-2/projects, 2019.

[149] ROBOTIS. Robotis turtlebot3 e-manual: http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/, 2019.

[150] ev3dev. Ev3 programming: https://www.ev3dev.org/, 2019.

[151] ROBOTIS. Robotis opencr 1.0 e-manual: http://emanual.robotis.com/docs/en/parts/controller/opencr10/, 2019.

[152] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. *arXiv preprint arXiv:1507.05946*, 2015.

Appendices

A The template for mission specification

```
1   ##################################################################
2   #               1. Team Formation                                #
3   #   Syntax) TeamName: RobotClassName robot_name                  #
4   ##################################################################
5   {     };
6   ##################################################################
7   #             2. Behavior of each team                           #
8   ##################################################################
9   #————————————————————— MASTER Team ————————————————————#
10  ##################################################################
11  #   Part 1. Composite Service Definition                         #
12  #   Syntax)                                                      #
13  #   TeamName.PlanName.CompositeServiceName{                      #
14  #       Service1();                                              #
15  #       if(condition){                                           #
16  #           Service3();                                          #
17  #           throw mode_change_event;                             #
18  #       }                                                        #
19  #       else{                                                    #
20  #           throw another_mode_change_event;                     #
21  #       }                                                        #
22  #   }                                                            #
23  ##################################################################
```

```
24      TeamName.PlanName.CompositeServiceName{
25          # Service1();
26          # if(value == "value_candidate")
27          #    throw mode_change_event;
28      } repeat(2 SEC) #-> if repeat needed
29      ##############################################################
30      #    Part 2. Multi-tasking Mode Definition                   #
31      #    Syntax)                                                 #
32      #    TeamName.ModeName{                                      #
33      #        set(PlanName, CompositeServiceName);                #
34      #    }                                                       #
35      ##############################################################
36      TeamName.ModeName{
37          # set(Plan, CompositeService or OFF);
38      }
39      ##############################################################
40      #    Part 3. Dynamic Mode Change Definition                  #
41      #    Syntax)                                                 #
42      #    TeamName.main{                                          #
43      #        case(ModeName):                                     #
44      #            catch(ModeChangeEvent1): mode = ModeName1       #
45      #            catch(ModeChangeEvent2): mode = ModeName2       #
46      #        default:                                            #
47      #            mode = ModeName                                 #
48      #    }                                                       #
49      ##############################################################
50      TeamName.main{
51          # case(mode_name):
52          #    catch(event): mode = mode_name
53          # default: mode = default_mode_name
54      }
```

```
55    # If there are other teams, repeat the above steps...
```

B Illustration of the SeMo SW development flow for the experiment of Section 3.1.2

In this experiment of cooperation scenario, heterogeneous multiple robots in Table 4.1 cooperate to find all colored papers in a given grid area. There are two teams: MasterTeam and SlaveTeam. The most powerful robot ("iRobot Create") is designated as a master robot, and the other robots are grouped into a slave team. All robots first move from the start point (0,0) to the target point (5,5). After reaching the target point, the slave team performs the same action: searching the colored papers. If it detects a colored paper that has not been found yet, it notifies the color to the master robot. Otherwise, it ignores the colored paper and keeps going. Until the slave team finds all colored papers, the master team waits. When the slave team finds all of the colored papers, the master and slave team return to the starting point.

B.1 Mission Specification

<table>
<tr><th>Plan
Mode</th><th>Listen Plan</th><th>Report Plan</th><th>Action Plan</th></tr>
<tr><td>Remote Control</td><td rowspan="3">Receive commands from operator</td><td>Broadcast commands</td><td>Move by command</td></tr>
<tr><td>Autonomous Move</td><td rowspan="2">Send its location, sensor values</td><td>Move to specific point avoiding obstacles</td></tr>
<tr><td>Autonomous Return</td><td>Move to starting point avoiding obstacles</td></tr>
<tr><td>Wait</td><td>Receive status of SlaveTeam</td><td>Forward status of SlaveTeam to operator</td><td>Stand by</td></tr>
</table>

Figure B.1: Specification of behaviors by robot team "MasterTeam"

Figure B.1 illustrates what tasks to be performed in each combination of mode and plan for the $MasterTeam$. It has four different modes of operation and three plans are running concurrently. In the SeMo framework, we first write a

mission script for behavior specification as follows.

```
1  {
2    MasterTeam: iRobotCreate irobot
3    SlaveTeam: TIEvalbot ti, NXTLego nxt
4  }
5
6  MasterTeam.Listen.ReceiveCmd{
7    receive(USER, USER.RC_CMD)
8    if(USER.RC_CMD == "RC_MODE")
9      throw CHANGE_RC_MODE
10   else if(USER.RC_CMD == "AUTO_MODE")
11     throw CHANGE_MOVE_MODE
12   else if(USER.RC_MD == "RETURN_MODE")
13     throw CHANGE_RETURN_MODE
14 } repeat(2 SEC)
15
16 MasterTeam.Listen.ReportStatus{
17   receive(USER, USER.RC_CMD)
18   receive(SlaveTeam, SlaveTeam.COLOR)
19   receive(SlaveTeam, SlaveTeam.LOCATION)
20   if(USER.RC_CMD == "RC_MODE")
21     throw CHANGE_RC_MODE
22   else if(USER.RC_MD == "RETURN_MODE")
23     throw CHANGE_RETURN_MODE
24 } repeat(2 SEC)
25
26 MasterTeam.Report.SendDefault{
27   send(USER, MasterTeam.DISTANCE)
28   send(USER, MasterTeam.CAMERA)
29   send(USER, MasterTeam.LOCATION)
30 } repeat(2 SEC)
```

```
31
32 MasterTeam.Report.Forward{
33    send(SlaveTeam, USER.RC_CMD)
34 }
35
36 MasterTeam.Report.ForwardStatus{
37    send(USER, SlaveTeam.COLOR)
38    send(USER, SlaveTeam.LOCATION)
39 } repeat(2 SEC)
40
41 MasterTeam.Action.AutonomousReturn{
42    move("0,0")
43    if(LOCATION == "0,0"){
44       throw CHANGE_FINISH
45    }
46 } repeat(LOCATION != "0,0")
47
48 MasterTeam.Action.AutonomousMove{
49    move("5,5")
50    if(LOCATION == "5,5")
51       throw CHANGE_WAIT_MODE
52 } repeat(LOCATION != "5,5")
53
54 MasterTeam.Action.RemoteControl{
55    process(USER.RC_CMD)
56 } repeat()
57
58 MasterTeam.Action.Wait{
59    standby()
60 } repeat()
61
```

```
62 #MODE Definition
63 MasterTeam.RC_MODE{
64    set(Listen, ReceiveCmd)
65    set(Report, Forward)
66    set(Action, RemoteControl)
67 }
68 MasterTeam.AUTO_MODE{
69    set(Listen, ReceiveCmd)
70    set(Report, SendDefault)
71    set(Action, AutonomousMove)
72 }
73 MasterTeam.WAIT_MODE{
74    set(Listen, ReportStatus)
75    set(Report, ForwardStatus)
76    set(Action, Wait)
77 }
78 MasterTeam.RETURN_MODE{
79    set(Listen, ReceiveCmd)
80    set(Report, SendDefault)
81    set(Action, AutonomousReturn)
82 }
83 MasterTeam.FINISH{
84    set(Listen, OFF)
85    set(Report, OFF)
86    set(Action, OFF)
87 }
88
89 #main
90 MasterTeam.main{
91    case(AUTO_MODE):
92       catch(CHANGE_RC_MODE): mode = RC_MODE
```

```
93        catch(CHANGE_WAIT_MODE): mode = WAIT_MODE
94        catch(CHANGE_RETURN_MODE): mode = RETURN_MODE
95     case(RC_MODE):
96        catch(CHANGE_MOVE_MODE): mode = AUTO_MODE
97        catch(CHANGE_WAIT_MODE): mode = WAIT_MODE
98        catch(CHANGE_RETURN_MODE): mode = RETURN_MODE
99     case(WAIT_MODE):
100       catch(CHANGE_RC_MODE): mode = RC_MODE
101       catch(CHANGE_RETURN_MODE): mode = RETURN_MODE
102    case(RETURN_MODE):
103       catch(CHANGE_MOVE_MODE): mode = AUTO_MODE
104       catch(CHANGE_RC_MODE): mode = RC_MODE
105       catch(CHANGE_WAIT_MODE): mode = WAIT_MODE
106       catch(CHANGE_FINISH): mode = FINISH
107    default: mode = AUTO_MODE
108 }
109
110 #------------------------------------ SlaveTeam ------------------------------------
111 SlaveTeam.Listen.ReceiveCmd{
112    receive(MasterTeam, USER.RC_CMD)
113    if(USER.RC_CMD == "RC_MODE")
114       throw CHANGE_RC_MODE
115    else if(USER.RC_CMD == "AUTO_MODE")
116       throw CHANGE_MOVE_MODE
117 } repeat()
118
119 SlaveTeam.Listen.CheckStatus{
120    receive(SlaveTeam, SlaveTeam.COLOR)
121 } repeat()
122
123 SlaveTeam.Report.ShareStatus{
```

```
124     send(MasterTeam, SlaveTeam.COLOR)
125     send(MasterTeam, SlaveTeam.LOCATION)
126     if(COLOR == "RGB")
127       throw CHANGE_RETURN_MODE
128   } repeat(COLOR != "RGB")
129
130   SlaveTeam.Action.AutonomousMove{
131     move("5,5")
132     if(LOCATION == "5,5")
133       throw CHANGE_SEARCH_MODE
134   } repeat(LOCATION != "5,5")
135
136   SlaveTeam.Action.Search{
137     search(colored_paper)
138   } repeat(COLOR != "RGB")
139
140   SlaveTeam.Action.RemoteControl{
141     process(USER.RC_CMD)
142   } repeat()
143
144   SlaveTeam.Action.AutonomousReturn{
145     move("0,0")
146     if(LOCATION == "0,0"){
147       throw CHANGE_FINISH
148     }
149   } repeat(LOCATION != "0,0")
150
151   SlaveTeam.AUTO_MODE{
152     set(Listen, ReceiveCmd)
153     set(Action, AutonomousMove)
154   }
```

```
155 SlaveTeam.SEARCH_MODE{
156   set(Listen, CheckStatus)
157   set(Report, ShareStatus)
158   set(Action, Search)
159 }
160 SlaveTeam.RC_MODE{
161   set(Listen, ReceiveCmd)
162   set(Report, OFF)
163   set(Action, RemoteControl)
164 }
165 SlaveTeam.RETURN_MODE{
166   set(Listen, ReceiveCmd)
167   set(Report, OFF)
168   set(Action, AutonomousReturn)
169 }
170 SlaveTeam.FINISH{
171   set(Listen, OFF)
172   set(Report, OFF)
173   set(Action, OFF)
174 }
175
176 SlaveTeam.main{
177   case(AUTO_MODE):
178     catch(CHANGE_SEARCH_MODE): mode = SEARCH_MODE
179     catch(CHANGE_RC_MODE): mode = RC_MODE
180   case(RC_MODE):
181     catch(CHANGE_MOVE_MODE): mode = AUTO_MODE
182   case(SEARCH_MODE):
183     catch(CHANGE_RETURN_MODE): mode = RETURN_MODE
184   case(RETURN_MODE):
185     catch(CHANGE_RC_MODE): mode = RC_MODE
```

```
186         catch(CHANGE_FINISH): mode = FINISH
187     default: mode = AUTO_MODE
188 }
```

In lines 1-5, we specify how two teams are composed; *MasterTeam* has a single "iRobotCreate" robot and *SlaveTeam* consists of a "TIEvalbot" and a "NXTLego". Lines 8-109 describe the behavior of *MasterTeam* and lines 110-188 show the behavior of *SlaveTeam*. We'll focus on the *MasterTeam*'s behavior.

In Figure B.1, each cell represents a verbal description of a composite service that is described in the script language in lines 6-60. A service such as move or standby used in the composite service is a basic service and is assumed to be given in the database. The values such as distance, camera, and location are also predefined values that the robot can use. The script editor helps the user to know which values and services are available in each robot. Lines 63-87 present which actions will be taken for each mode of operation, which corresponds to each row of Figure B.1. In each mode, three plans are running concurrently with the specified composite service. By default, the *FINISH* mode is defined to finish all plans, which is written in lines 83-87.

The main loop of lines 90-108 depicts the mode transition based on the event caught by *catch* phrase during the execution of the current mode. The behavior of *SlaveTeam* is written similarly to that of *MasterTeam*.

B.2 Strategy Description

To fill the large abstraction gap between mission specification and model-based task graph specification, the strategy description is needed. The following shows a part of the strategy description file for *MasterTeam*.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <Strategy name="MasterTeam" xmlns="...">
```

```
3           <!-- ellipsis -->
4       <ServiceInfo>
5           <Service plan_name="Action" name="move">
6               <TargetRobot name="iRobotCreate" nickName="
                    irobot" ea="1"/>
7               <ActionInfo>
8                   <Action name="CheckGoal" initial="Yes"
                        actionId="0">
9                       <CheckSensor name="Vicon">
10                          <Parameter name="currentLocation"
                                type="viconInfo"
                                dataCharacteristic="channel"
                                size="1" srcTask="vicon" dstTask
                                ="ControlTask"/>
11                      </CheckSensor>
12                      <Parameter name="goalLocation" type="
                            viconInfo" dataCharacteristic="
                            variable" size="1"/>
13                      <TransitionInfo>
14                          <Transition srcId="0" dstId="1">
15                              <Condition cond="
                                    currentLocation==
                                    goalLocation"/>
16                          </Transition>
17                          <Transition srcId="0" dstId="4">
18                              <Condition cond="
                                    currentLocation!=
                                    goalLocation"/>
19                          </Transition>
20                      </TransitionInfo>
21                  </Action>
```

```
22                  <Action name="CheckObstacle" actionId="1">
23                      <CheckSensor name="Ultrasonic">
24                          <Parameter type="integer" name="
                                distance" dataCharacteristic="
                                channel" size="1" srcTask="
                                ultrasonic" dstTask="ControlTask
                                "/>
25                      </CheckSensor>
26                      <TransitionInfo>
27                          <Transition srcId="1" dstId="2">
28                              <Condition cond="distance&gt;20
                                    "/>
29                          </Transition>
30                          <Transition srcId="1" dstId="3">
31                              <Condition cond="distance&lt;20
                                    "/>
32                          </Transition>
33                      </TransitionInfo>
34                  </Action>
35                  <Action name="MoveRandom" actionId="2">
36                      <ExecuteActuator name="Move">
37                          <Parameter type="integer" name="cmd
                                " value="RandomNumber"
                                dataCharacteristic="sysRequest"
                                size="1" srcTask="ControlTask"
                                dstTask="wheel"/>
38                      </ExecuteActuator>
39                      <TransitionInfo>
40                          <Transition srcId="2" dstId="0" />
41                      </TransitionInfo>
42                  </Action>
```

```
43                  <Action name="AvoidObstacle" actionId="3">
44                      <ExecuteActuator name="Move">
45                          <Parameter type="integer" name="cmd
                                " value="Turn"
                                dataCharacteristic="sysRequest"
                                size="1" srcTask="ControlTask"
                                dstTask="wheel"/>
46                      </ExecuteActuator>
47                      <TransitionInfo>
48                          <Transition srcId="3" dstId="0" />
49                      </TransitionInfo>
50                  </Action>
51                  <Action name="Finish" final="Yes" actionId=
                        "4">
52                      <Parameter value="1" type="integer"
                            name="result" dataCharacteristic="
                            variable"/>
53                      <TransitionInfo>
54                          <Transition srcId="5" dstId="0"/>
55                      </TransitionInfo>
56                  </Action>
57              </ActionInfo>
58          </Service>    <!-- ellipsis -->
59      </ServiceInfo>
60      <NonFunctionalInfo>
61          <BatteryRequirement>
62              <Condition cond="Battery&lt;30">
63                  <Sensor name="Camera" periodLevel="2"/>
64                  <Sensor name="Ultrasonic" periodLevel="2"/>
65                  <Actuator name="Move" speedLevel="2"/>
66              </Condition>
```

```
67              </BatteryRequirement>
68          </NonFunctionalInfo>
69  </Strategy>
```

Lines 4-66 describe the *move* service defined in the *Action* plan. First, we specify which robot the service description is applied as shown in line 5. The *move* service is described by a textual specification of a finite state machine (FSM) that consists of six fine-grain services, called *actions*. This FSM will be included in the control task when a task graph is automatically synthesized. Each action is given an identifier (*id*) along with its name. Each action can execute an algorithm, execute an actuator, or receive an output value from a sensor or an algorithm. For instance, on lines 7-10, the *currentLocation* value is taken from an algorithm called *Localization*. And a transition is triggered by comparing it with the internal variable called *goalLocation*.

Also, non-functional requirements can be specified in the strategy description file. In this example, we add an adaptive resource management policy to save the battery energy. In lines 70-76, if the remaining battery energy is lower than 30, the period of *Camera* and *Ultrasonic* sensor is reduced to level 2, and the speed of *Move* actuator is reduced to level 2.

B.3 Task Graph Specification

Based on the mission script and strategy description file, the task graphs are automatically generated. The behavior of each robot is specified by a task graph as shown in FigureB.2. From the team formation information specified by the mission script, a task graph for each robot is created. For the communication between robots, a library task is added for the management of shared information. The sensor tasks used by a robot are instantiated by referring to the data to be transmitted and the parameter defined in the strategy stage. In the case of

algorithm and actuator tasks, they are added according to the service specified in the mission script and strategy description file. For example, the mission script says that $MasterTeam$ transmits the values of distance, camera, and location to the user. Thus, ultrasonic sensor task, camera task, and localization algorithm task are synthesized in the task graph. Note that a task may be expanded as a task sub-graph in case the sub-graph is registered in the database as a composite service that the task performs.

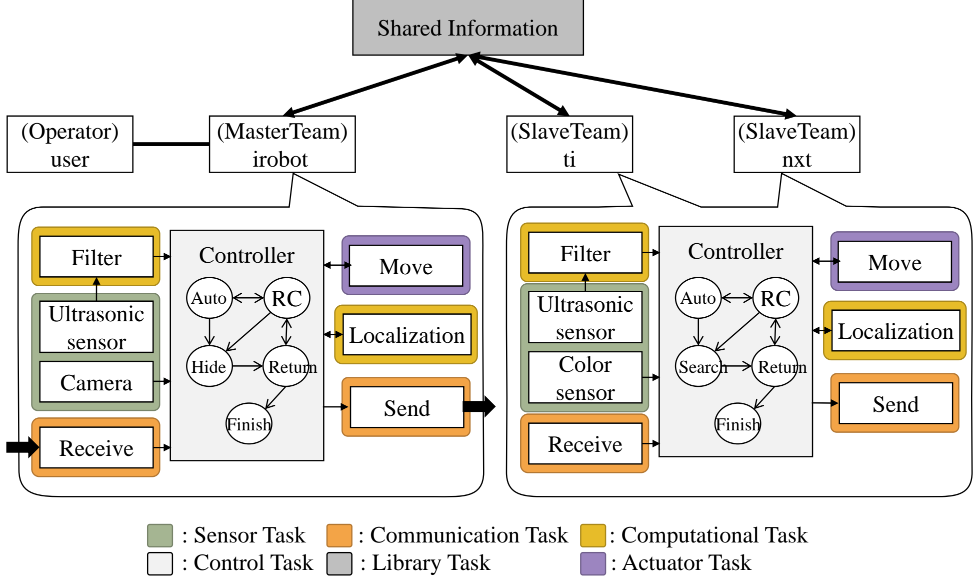


Figure B.2: Task graph specification of the example

The most challenging is the synthesis of the control task that is generated as a hierarchical FSM based on the mission specification and strategy description. Figure B.3 illustrates how the mode and plan information in the mission script is translated into a state transition diagram in the top-level FSM. Fine-grained services defined in the strategy description file are translated into a bottom-level FSM. In this example, the *move* service is refined into six actions, which are represented by six states in the FSM. In each state, values are received from the sensor and algorithm task, and an algorithm or actuator task is executed.

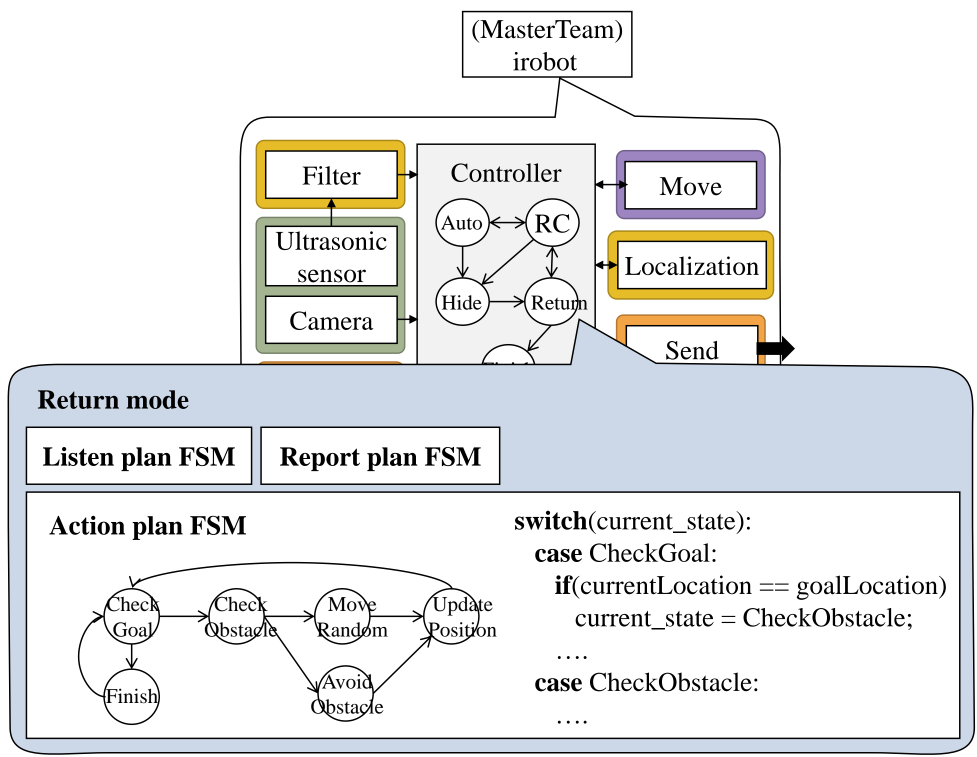


Figure B.3: Control task specification for the example

C Mission Specification for the experiment of Section 3.1.3

```
1  {
2            MasterTeam: iRobotCreate irobot
3            SlaveTeam: Turtlebot3_Burger burger[2], Ev3_Robot ev3[2]
4  }
5
6  MasterTeam.Listen.RemoteListen{
7        receive(USER, USER.RC_CMD)
8        if(USER.RC_CMD == SWITCH_PERFORM_MODE){
9            throw find_color_paper
10       }
```

```
11 } repeat()
12
13 MasterTeam.Report.DeliverToSlave{
14     publish(SlaveTeam, USER.RC_CMD)
15 } repeat()
16
17 MasterTeam.Action.RCMove{
18     if(USER.RC_CMD == CMD_FORWARD)
19         go_forward()
20     else if(USER.RC_CMD == CMD_BACKWARD)
21         go_backward()
22     else if(USER.RC_CMD == CMD_TURN_LEFT)
23         turn_left()
24     else if(USER.RC_CMD == CMD_TURN_RIGHT)
25         turn_right()
26     else if(USER.RC_CMD == CMD_STOP)
27         stop()
28 } repeat()
29
30 MasterTeam.Listen.AutonomousListen{
31     receive(USER, USER.RC_CMD)
32     if(USER.RC_CMD == SWITCH_RC_MODE){
33         throw switch_rc
34     }
35 } repeat()
36
37 MasterTeam.Action.Move{
38     move()
39     throw find_color_paper
40 }
41
```

```
42 MasterTeam.Listen.ListenSlaveTeam{
43     receive(USER, USER.RC_CMD)
44     if(USER.RC_CMD == SWITCH_FINISH){
45         publish(SlaveTeam, USER.RC_CMD)
46         throw finish
47     }
48     subscribe(SlaveTeam, SlaveTeam.SEARCHEDCOLOR)
49 } repeat()
50
51 MasterTeam.Report.DeliverToUser{
52     send(USER, SlaveTeam.SEARCHEDCOLOR)
53 } repeat()
54
55 MasterTeam.Action.Standby{
56     stand_by()
57 }
58
59 MasterTeam.RC_MODE{
60     set(Listen, RemoteListen)
61     set(Report, DeliverToSlave)
62     set(Action, RCMove)
63 }
64
65 MasterTeam.MOVE_MODE{
66     set(Listen, AutonomousListen)
67     set(Report, DeliverToSlave)
68     set(Action, Move)
69 }
70
71 MasterTeam.STANDBY_MODE{
72     set(Listen, ListenSlaveTeam)
```

```
73      set(Report, DeliverToUser)
74      set(Action, Standby)
75  }
76
77  MasterTeam.FINISH_MODE{
78      set(Listen, OFF)
79      set(Report, OFF)
80      set(Action, OFF)
81  }
82
83  MasterTeam.main{
84          case(RC_MODE):
85          catch(find_color_paper): mode = STANDBY_MODE
86      case(MOVE_MODE):
87          catch(switch_rc): mode = RC_MODE
88          catch(find_color_paper): mode = STANDBY_MODE
89      case(STANDBY_MODE):
90          catch(finish): mode = FINISH_MODE
91      default: mode = RC_MODE
92  }
93
94  SlaveTeam.RC_MODE{
95      set(Listen, ListenMaster)
96          set(Action, FollowMaster)
97  }
98
99  SlaveTeam.MOVE_MODE{
100     set(Listen, ListenMasterinMove)
101         set(Action, Move)
102 }
103
```

```
104 SlaveTeam.Listen.ListenMaster{
105     subscribe(MasterTeam, USER.RC_CMD)
106     if(USER.RC_CMD == SWITCH_PERFORM_MODE){
107         throw find_color_paper
108     }
109 } repeat()
110
111 SlaveTeam.Listen.ListenMasterinMove{
112     subscribe(MasterTeam, USER.RC_CMD)
113     if(USER.RC_CMD == SWITCH_RC_MODE){
114         throw swtich_rc
115     }
116 } repeat()
117
118 SlaveTeam.Action.FollowMaster{
119     if(USER.RC_CMD == CMD_FORWARD)
120         go_forward()
121     else if(USER.RC_CMD == CMD_BACKWARD)
122         go_backward()
123     else if(USER.RC_CMD == CMD_TURN_LEFT)
124         turn_left()
125     else if(USER.RC_CMD == CMD_TURN_RIGHT)
126         turn_right()
127     else if(USER.RC_CMD == CMD_STOP)
128         stop()
129 } repeat()
130
131 SlaveTeam.Action.Move{
132     move()
133     throw find_color_paper
134 }
```

```
135
136 SlaveTeam.SEARCH_MODE{
137         set(Listen, ListenAlarm)
138         set(Report, ReportMasterTeam)
139         set(Action, Search)
140         set(Resolve, Decide)
141 }
142
143 SlaveTeam.Listen.ListenAlarm{
144     subscribe(MasterTeam, USER.RC_CMD)
145     if(USER.RC_CMD == SWITCH_FINISH){
146         throw finish
147     }
148     subscribe(SlaveTeam, SlaveTeam.SEARCHEDCOLOR)
149     subscribe(SlaveTeam, SlaveTeam.Message)
150     if(SlaveTeam.Message == CMD_HIDE){
151         throw change_hide
152     } else if (SlaveTeam.Message == CMD_FINISH) {
153         throw finish
154     }
155 } repeat()
156
157 SlaveTeam.Report.ReportMasterTeam{
158     [[
159     leader(instance of Turtlebot3_Burger){
160         light_on()
161         publish(MasterTeam, SlaveTeam.SEARCHEDCOLOR)
162     }
163     ]]
164 } repeat()
165
```

```
166  SlaveTeam.Action.Search{
167      [[
168      group(instance of Turtlebot3_Burger){
169          loop (2 SEC) {
170              if( SlaveTeam.LIGHTNESS < 200){
171                  publish(SlaveTeam, SlaveTeam.Message = SUGGEST_HIDE)
172              }
173          }
174      }
175      others{
176          loop(SlaveTeam.SEARCHEDCOLOR == FULL) {
177              search()
178              publish(SlaveTeam, SlaveTeam.SEARCHEDCOLOR)
179          }
180          publish(SlaveTeam, SlaveTeam.Message = SUGGEST_FINISH)
181      }
182      ]]
183  }
184
185  SlaveTeam.Resolve.Decide{
186      [[
187      leader(instance of Turtlebot3_Burger){
188          if(SlaveTeam.Message == SUGGEST_HIDE)
189              publish(SlaveTeam, SlaveTeam.Message = CMD_HIDE)
190          else if(SlaveTeam.Message == SUGGEST_FINISH)
191              publish(SlaveTeam, SlaveTeam.Message = CMD_FINISH)
192      }
193      ]]
194  }
195
196  SlaveTeam.HIDE_MODE{
```

```
197             set(Listen, HideListen)
198             set(Action, Hide)
199 }
200
201 SlaveTeam.Listen.HideListen{
202         subscribe(SlaveTeam, SlaveTeam.Message)
203         if(SlaveTeam.Message == SUGGEST_SEARCH)
204             throw find_color_paper
205 }
206
207 SlaveTeam.Action.Hide{
208         [[
209             group(instance of Turtlebot3_Burger){
210             if(SlaveTeam.LIGHTNESS > 500){
211                 publish(SlaveTeam, SlaveTeam.Message = SUGGEST_SEARCH)
212             }
213         }
214         others{
215             scatter()
216         }
217         ]]
218 } repeat()
219
220 SlaveTeam.FINISH_MODE{
221             set(Listen, OFF)
222             set(Report, OFF)
223             set(Action, OFF)
224         set(Resolve, OFF)
225 }
226
227 SlaveTeam.main{
```

```
228         case(RC_MODE):
229                 catch(find_color_paper): mode = SEARCH_MODE
230         case(MOVE_MODE):
231                 catch(swtich_rc): mode = RC_MODE
232                 catch(find_color_paper): mode = SEARCH_MODE
233         case(SEARCH_MODE):
234                 catch(finish): mode = FINISH_MODE
235                 catch(change_hide): mode = HIDE_MODE
236         case(HIDE_MODE):
237                 catch(find_color_paper): mode = SEARCH_MODE
238         default: mode = RC_MODE
239 }
```

In lines 1-4, we specify how two teams are composed; $MasterTeam$ has a single "iRobotCreate" robot and $SlaveTeam$ consists of two "Turtlebot3_Burger" robots and two "Ev3_Robot" robots. Lines 6-92 describe the behavior of $MasterTeam$, and lines 94-239 show the behavior of $SlaveTeam$. We will focus on the $SlaveTeam$'s behavior.

There are four modes in $SlaveTeam$. In *SEARCH_MODE*, there are four plans: "Listen", "Report", "Action", "Resolve" (line 136-194). However, all robots in $SlaveTeam$ do not act the same. Only a leader robot which candidate is "Turtlebot3_Burger" can report $MasterTeam$ what they found colored papers (line 157-164).

요 약

가까운 미래에는 다양한 로봇이 다양한 분야에서 하나의 임무를 협력하여 수행하는 모습은 흔히 볼 수 있게 될 것이다. 그러나 실제로 이러한 모습이 실현되기에는 두 가지의 어려움이 있다. 먼저 로봇을 운용하기 위한 소프트웨어를 명세하는 기존 연구들은 대부분 개발자가 로봇의 하드웨어와 소프트웨어에 대한 지식을 알고 있는 것을 가정하고 있다. 그래서 로봇이나 컴퓨터에 대한 지식이 없는 사용자들이 여러 대의 로봇이 협력하는 시나리오를 작성하기는 쉽지 않다. 또한, 로봇의 소프트웨어를 개발할 때 로봇의 하드웨어의 특성과 관련이 깊어서, 다양한 로봇의 소프트웨어를 개발하는 것도 간단하지 않다. 본 논문에서는 상위 수준의 미션 명세와 로봇의 행위 프로그래밍으로 나누어 새로운 소프트웨어 개발 프레임워크를 제안한다. 또한, 본 프레임워크는 크기가 작은 로봇부터 계산 능력이 충분한 로봇들이 서로 군집을 이루어 미션을 수행할 수 있도록 지원한다.

본 연구에서는 로봇의 하드웨어나 소프트웨어에 대한 지식이 부족한 사용자도 로봇의 동작을 상위 수준에서 명세할 수 있는 스크립트 언어를 제안한다. 제안하는 언어는 기존의 스크립트 언어에서는 지원하지 않는 네 가지의 기능인 팀의 구성, 각 팀의 서비스 기반 프로그래밍, 동적으로 모드 변경, 다중 작업(멀티 태스킹)을 지원한다. 우선 로봇은 팀으로 그룹 지을 수 있고, 로봇이 수행할 수 있는 기능을 서비스 단위로 추상화하여 새로운 복합 서비스를 명세할 수 있다. 또한 로봇의 멀티 태스킹을 위해 '플랜' 이라는 개념을 도입하였고, 복합 서비스 내에서 이벤트를 발생시켜서 동적으로 모드가 변환할 수 있도록 하였다. 나아가 여러 로봇의 협력이 더욱 견고하고, 유연하고, 확장성을 높이기 위해, 군집 로봇을 운용할 때 로봇이 임무를 수행하는 도중에 문제가 생길 수 있으며, 상황에 따라 로봇을 동적으로 다른 행위를 수행할 수 있다고 가정한다. 이를 위해 동적으로도 팀을 구성할 수 있고, 여러 대의 로봇이 하나의 서비스를 수행하는 그룹 서비스를 지원하고, 일대 다 통신과 같은 새로운 기능을 스크립트 언어에 반영하였다. 따라서 확장된 상위 수준의 스크립트 언어는 비전문가도 다양한 유형의 협력 임무를 쉽게 명세할 수 있다.

로봇의 행위를 프로그래밍하기 위해 다양한 소프트웨어 개발 프레임워크가 연

구되고 있다. 특히 재사용성과 확장성을 중점으로 둔 연구들이 최근 많이 사용되고 있지만, 대부분의 이들 연구는 리눅스 운영체제와 같이 많은 하드웨어 자원을 필요로 하는 운영체제를 가정하고 있다. 또한, 프로그램의 분석 및 성능 예측 등을 고려하지 않기 때문에, 자원 제약이 심한 크기가 작은 로봇의 소프트웨어를 개발하기에는 어렵다. 그래서 본 연구에서는 임베디드 소프트웨어를 설계할 때 쓰이는 정형적인 모델을 이용한다. 이 모델은 정적 분석과 성능 예측이 가능하지만, 로봇의 행위를 표현하기에는 제약이 있다. 그래서 본 논문에서 외부의 이벤트에 의해 수행 중간에 행위를 변경하는 로봇을 위해 유한 상태 머신 모델과 데이터 플로우 모델이 결합하여 동적 행위를 명세할 수 있는 확장된 모델을 적용한다. 그리고 딥러닝과 같이 계산량을 많이 필요로 하는 응용을 분석하기 위해, 루프 구조를 명시적으로 표현할 수 있는 모델을 제안한다. 마지막으로 여러 로봇의 협업 운용을 위해 로봇 사이에 공유되는 정보를 나타내기 위해 두 가지 모델을 사용한다. 먼저 중앙에서 공유 정보를 관리하기 위해 라이브러리 태스크라는 특별한 태스크를 통해 공유 정보를 나타낸다. 또한, 로봇이 자신의 정보를 가까운 로봇들과 공유하기 위해 멀티캐스팅을 위한 새로운 포트를 추가한다. 이렇게 확장된 정형적인 모델은 실제 로봇 코드로 자동 생성되어, 소프트웨어 설계 생산성 및 개발 효율성에 이점을 가진다.

비전문가가 명세한 스크립트 언어는 정형적인 태스크 모델로 변환하기 위해 중간 단계인 전략 단계를 추가하였다. 제안하는 방법론의 타당성을 검증하기 위해, 시뮬레이션과 여러 대의 실제 로봇을 이용한 협업하는 시나리오에 대해 실험을 진행하였다.

주요어 : 상위 수준 명세, 데이터플로우, SDF 그래프, 반복 행위 명세, 공유 정보 관리, 코드 자동 생성, 소프트웨어 개발 프레임워크, 협업 로봇

학번 : 2013-20912