SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus on Software Engineering for Internet Computing

August 2016, Vol. 59 080102:1–080102:17
doi: 10.1007/s11432-016-5593-x

CBBR: enabling distributed shared memory-based coordination among mobile robots

Xue JIANG & Yu HUANG*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023*, China*

Received April 25, 2016; accepted May 25, 2016; published online July 18, 2016

Abstract Coordinating mobile robots are widely used in commercial and industrial settings to fulfill various tasks. However, to program the coordination among mobile robots is challenging. A coordination framework is needed to shield the programmer from handling low-level details of robot control and communication, while supporting flexible and cost-effective coordination at the same time. The coordination framework should also be able to well coexist with the underlying robot control. To this end, we propose the Coordination-enabled Behavior-Based Robotics (CBBR) framework. CBBR employs Distributed Shared Memory (DSM) to support coordination. The shared memory illusion built by the DSM greatly simplifies the coordination logic. Moreover, the flexible access patterns of the DSM and the rich consistency semantics of the DSM reads and writes enable flexible and cost-effective coordination. With the coordination support from the DSM, CBBR naturally extends the classical Behavior-Based Robotics (BBR) for robot control. From the perspective of robot control using BBR, the shared variables in the DSM act as the logical sensors capturing the status of coordination. The coordination algorithms are encapsulated into coordination behaviors. Thus, the physical environment status and the coordination status may trigger the physical and the coordination behaviors. The scheduling of both types of behaviors integrates coordination into robot control. We conduct a case study to demonstrate the use of CBBR. The performance measurements show the cost-effectiveness of coordinating mobile robots based on CBBR, in terms of time, space, and energy consumption.

Keywords distributed shared memory, behavior-based robotics, coordination, behavior hierarchy, behavior scheduling

Citation Jiang X, Huang Y. CBBR: enabling distributed shared memory-based coordination among mobile robots. Sci China Inf Sci, 2016, 59(8): 080102, doi: 10.1007/s11432-016-5593-x

1 Introduction

Mobile robots are widely used in commercial and industrial settings [1–3]. For example, mobile robots are deployed to patrol in the chemical plants [4], monitoring hazardous material leak [5]. Warehouses have been using mobile robots to efficiently move materials from stocking shelves to order fulfillment zones [6].

A group of coordinating mobile robots are often deployed to fulfill a given task [7–9], since using a group of mobile robots can have several potential advantages over one single robot. Specifically, the operation environment usually has larger size than that of one robot. The environment has rich and

* Corresponding author (email: yuhuang@nju.edu.cn)

complex semantics, which is also beyond the sensing and controlling capacity of one robot. Moreover, to accomplish a given task the system of mobile robots can potentially be simpler to program and cheaper to build, than using one single powerful but complex and expensive robot. The deployment of a system of mobile robots also has the potential to increase overall versatility and reliability.

Behavior-Based robotics (BBR) [10] is widely used to program mobile robots. Using BBR, the complex task for the robot is decomposed into multiple behaviors. The behavior design encapsulates the closed-loop control of the mobile robot. The robot first gets sensor data of the physical environment and evaluates the triggering condition of the behavior. When the triggering condition holds, the robot conducts the action of the behavior, which results in the next loop of robot control. The scheduling of all the behaviors according to predefined behavior priorities guarantees the fulfilment of the task. BBR achieves the separation of concerns in mobile robot control by separating the design of robot behaviors from the design of the behavior scheduling strategy.

Though BBR simplifies the programming of one single mobile robot, it lacks the support for programming robot coordination. We need to naturally extend BBR to enable the programming of one coordinating robot. By "natural extension of BBR", we mean that the status of coordination needs to be captured in certain coordination data, and the actions concerning the coordination should be encapsuled into the "coordination behavior". Thus, both types of conditions (evaluating the physical sensor data and the coordination data) can trigger both types of robot behaviors (physical behaviors and coordination behaviors). The arbitrator schedules both types of behaviors to control the robot to fulfil the task, coordinating with its companions when necessary.

The characteristics of BBR-based robot coordination motivate us to consider data-driven coordination mechanisms. Existing data-driven coordination mechanisms are mainly based on shared tuple spaces [11]. Distributed entities coordinate by getting tuples from and putting tuples to the shared tuple space. The communication is generative since the tuple generated by certain entity has an independent existence in the tuple space. Though the shared tuple space and the generative communication make it easier to program coordination, it comes at the cost for finding the tuples and for storing the tuples. Using shared tuple spaces may be expensive. On the one hand, the communication may not need to be generative. In our scenario, the robots know all the participating entities (though certain robots may get out of reach due to mobility or failures). On the other hand, the semantics of the tuple space primitives are limited. We need rich semantics for more flexible and reliable coordination.

To address the challenges mentioned above, we propose the Coordination-enabled Behavior-Based Robotics (CBBR) framework to support the programming of coordination among mobile robots controlled by BBR. Specifically, CBBR has the following features:

- *Distributed Shared Memory-based Coordination*: The Distributed Shared Memory (DSM) [12] builds the shared memory illusion among mobile robots, i.e., the robots access globally synchronized shared data like accessing local data. The key construct in the DSM is the shared variables. The robot accesses shared variables with pre-defined access patterns (e.g., Single-Writer-Multiple-Reader or Multiple-Writer-Multiple-Reader). Different replicas of the shared variable are automatically synchronized according to pre-specified consistency semantics (e.g., atomicity [13] or causal consistency [14]). Leveraging the shared variables, versatile shared memory distributed algorithms can be devised to support robot coordination. The fine-grained tuning of access patterns and consistency semantics enables flexible and cost-effective coordination.
- *Coordination-enabled Behavior-based Robotics*: In CBBR, the closed-loop control of BBR is retained and naturally extended to support coordination. The robot persistently senses changes in the environment and takes certain action which changes the environment. The environment information not only includes the physical environment, but also includes the logical environment. In comparison with the physical environment data from sensors, the logical environment data denotes status of coordination and is stored in the shared variables. The robot can take actions according to the physical environment, the logical environment or a combination of both. Similarly, in comparison with controlling the physical environment through actuators, the robot can also control the logical environment through coordination behaviors. The coordination behaviors are implemented as the shared memory algorithms over the DSM.

Coordination behaviors interact with the coordination behaviors of other robots, thus making changes to the logical environment. The arbitrator assigns priorities to different behaviors and schedules both types of behaviors.

We first propose the formal semantics of CBBR. Then we propose the design and implementation of CBBR. A case study of two mobile robots mutual exclusively passing a gate is conducted to demonstrate the effectiveness of CBBR. The performance measurements show that CBBR induces limited overhead for enabling coordination, in terms of time, space, and energy consumption.

The rest of this work is organized as follows. Section 2 presents the formal semantics of CBBR. Section 3 presents behavior design and scheduling of CBBR. Section 4 presents the architecture of CBBR. Section 5 conducts the case study and Section 6 presents the performance measurements. Section 7 discusses related work. In Section 8, we conclude the work and discuss the future work.

2 Formal semantics for CBBR

In this section, we use the I/O automata [15] to formally describe the key constructs in CBBR. This enables us to clearly and unambiguously present the design of behaviors and behavior scheduling strategies. This also facilitates further employment of verifications techniques and tools. In the following, first we discuss the distributed shared memory. Then we give the description of physical behavior and coordination behavior. Finally, we discuss the scheduling of both behaviors.

2.1 Distributed shared memory

We organize the DSM into three layers: message passing, shared registers, and shared-memory coordination algorithms. The key construct in the DSM is the shared register. The shared registers help the robot obtain situation of the logical environment. Provided with registers, almost all existing shared memory distributed algorithms can be implemented and leveraged to enable robot coordination. The I/O automata description of the shared register is in Appendix A.1.

The shared register masks the low level details of network communication and provides uniform Read/Write interfaces for shared memory access. There are two essential issues to be addressed in the construction of a shared register: Read/Write permission and consistency semantics. Detailed implementations of the DSM are discussed in the design and implementation of CBBR in Section 4.

2.2 Physical behavior

In CBBR, the robot can not only obtain situations of the physical environment via the sensors, but also obtain situations of the logical environment via the shared variables.

Each physical behavior includes a triggering condition and an action. The condition is a predicate corresponding to certain situation of the environment. In response to the situation of its concern, the robot takes certain actions, which usually involves the operations of the actuators (e.g., the motor). Details about the formal specification in I/O automata can be found in Appendix A.2.

2.3 Coordination behavior

Coordination of the robots is achieved by carefully designed coordination algorithms. Provided with the DSM support, almost all existing shared memory distributed algorithms can be leveraged to enable robot coordination. Provided with the underlying BBR-based robot control, the coordination algorithms are encapsulated into coordination behaviors.

Similarly, the coordination behavior has a condition and an action. The condition is also a predicate of environment data. The action involves invoking a coordination algorithm, which includes read/write operations of shared registers. The details of formally description of coordination behavior is in Appendix A.3.

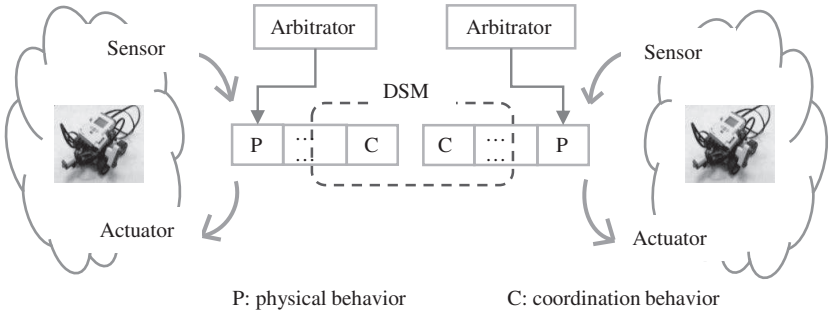

Figure 1 DSM-based coordination among mobile robots.

2.4 Scheduling

Basically, the behavior is scheduled when its triggering condition turns true. Since there may be multiple behaviors whose triggering conditions are all true, the behaviors are assigned totally ordered priorities. A dedicated arbitrator persistently scans the behaviors with decreasing priorities. It schedules the behavior with the triggering condition getting true and the highest priority to be executed. In order to decrease the cost of finding the behavior which can be scheduled, we give an exemplar implementation by using the array. The behaviors are placed in the array in the order of decreasing priority. More formally, the arbitrator works as shown in Appendix A.4.

3 Behavior design and scheduling in CBBR

CBBR incorporates coordination into the traditional BBR mechanism for robot control. In CBBR, each robot not only needs to sense the physical environment, but also needs to sense the logical environment, i.e., status of the coordination. The coordinating actions of the robot also need to be integrated into the close-loop robot control. So in CBBR, each robot has both physical behaviors and coordination behaviors. When multiple behaviors can possibly be conducted, they need to scheduled to fulfil the task. We divide the behaviors into four types, three types of physical behaviors, plus the coordination behavior. We assign different priorities for each type of behaviors according to the characteristics of the behavior.

The overall framework of the CBBR is shown in Figure 1. In the following, we discuss in detail how the complex task is decomposed into the hierarchy of robot behaviors. We also discuss the primary principles of deciding the behavior priorities and scheduling the behaviors.

3.1 Design of the behavior hierarchy

Usually the task is complex and needs to be fulfilled by combination of multiple behaviors. In design of the robot behavior, we decompose the task into less complex behaviors. The decomposition is recursively conducted until we reach the basic behaviors of the robot. For example, in the case study, the task of "pass the gate one by one" can be decomposed into the behavior hierarchy shown in Figure 2.

In implementation of robot behaviors, we construct the behaviors in a bottom-up manner according to the behavior hierarchy. Usually the robot has a limited number of basic behaviors. Complex behaviors are the combination of the basic behaviors. For example, in our case study, the basic (physical) behaviors of the robot include "going forward/backward" and "turning left/right". These basic behaviors makes up complex behaviors e.g., "passing the gate" and "recovering from the stuck". The hierarchical composition from simpler behaviors to more complex ones greatly facilitates the reuse of behavior implementation.

Since our task involves the coordination among multiple robots, the coordinating actions of the robot are programmed as the coordination behavior. In this way, we integrate the robot coordination into the BBR-based robot control. Core of the coordination behavior is usually a shared memory distributed algorithm, e.g., the Bakery algorithm [16] for mutual exclusion in our case study below.

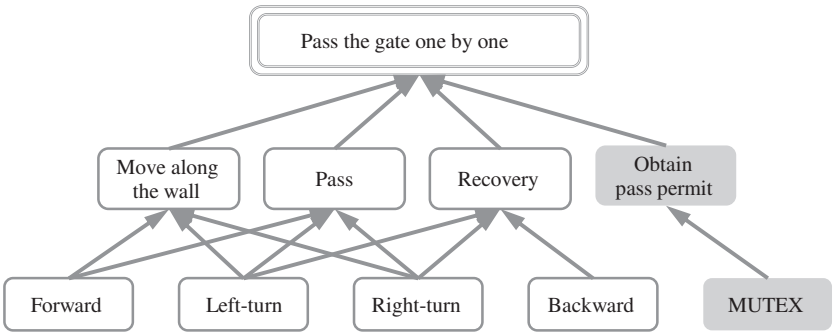

Figure 2 Behavior hierarchy.

3.2 Design of the behavior scheduling strategy

Given the design of the behavior hierarchy, we still need to decide the timing of each behavior. In BBR, a dedicated *arbitrator* is employed to decide the timing of each behavior, as shown in Figure 6. In this way, we achieve the separation of concerns by separating the design of the behaviors from the design of the scheduling strategy.

The behavior scheduling includes two primary issues. First, the arbitrator needs to evaluate the triggering condition of the behavior. Second, when there are multiple behaviors whose conditions are all true, the arbitrator needs to schedule the behaviors according to their predefined priorities. We discuss these two issues in detail below.

Whether to trigger a behavior is decided by the triggering condition. In the scenario of multiple coordinating robots, the condition can be about the status of the physical environment, e.g., "the robot is less than 0.1m to the wall". The condition can also be status of the logical environment, i.e., the status of the coordination. For example, the condition "having obtained the permit to pass the gate after coordination" can be the triggering condition of the physical behavior "passing the gate".

Condition of the physical environment is evaluated based on the physical environment data obtained from the sensors. Condition of the logical environment is evaluated based on the coordination status data, which is obtained from the DSM. These two types of conditions can be combined by logical connectors such as conjunction and disjunction.

Since there could be multiple behaviors whose conditions are simultaneously true, the arbitrator needs to decide in advance the priority of each behavior. In principle, there are mainly three types of physical behaviors. Besides, there is also the coordination behavior. We assign priorities to each type of behaviors as follows, and we discuss the behaviors in the order of increasing priority:

• *Routine physical behavior*: This is the default behavior the robot conducts, thus it has the lowest priority. For example, "moving forward" is often the routine behavior of the mobile robot. When the robot does not need to conduct other behaviors (e.g, it does not need to change the direction or circumvent an obstacle), it just goes forward. Other behaviors of higher priorities can be defined to ensure that the robot can handle different situations as it goes forward.

• *Objective physical behavior*: This type of behavior directly corresponds to the task of the robot. For example, if the task is to pass the gate, the behavior "passing the gate" (going forward and making left and right turns to go through the gate) is the objective behavior. This type of behavior has higher priority, and it is often designed to ensure that the robot proceeds in the right manner toward fulfilling the task.

• *Coordination behavior*: Since fulfillment of the task requires the coordination among multiple robots, the robot first needs to coordinate with its companions. Then it can decide what itself should do. To this end, we assign the coordination behavior higher priority than that of the objective behavior.

• *Fault-handling physical behavior*: This type of behavior is dedicated to enabling the robot to get recovered when something goes wrong. We assign these behaviors the highest priority to ensure that the robot can always get recovered from faults in the first place.

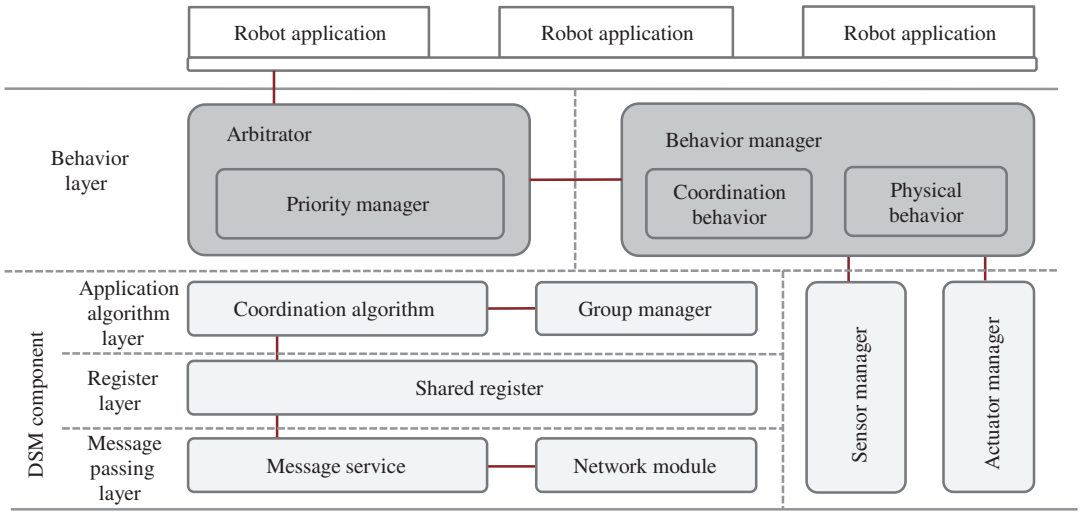

Figure 3 Architecture of CBBR.

Given the priority of each behavior, the arbitrator schedules the behavior with the triggering condition getting true and the highest priority to be executed.

4 System architecture of CBBR

In this section, we present the design and implementation of CBBR, which greatly simplify programming multiple robots to fulfill coordination tasks. The design process of the CBBR framework consists of several logically independent steps. Thus we adopt a layered architecture for CBBR. The layered architecture groups related functionalities into distinct layers and provides software engineering benefits such as separation of concerns, information hiding, extensibility, and reusability. The architecture of CBBR is listed in Figure 3. It contains two layers, the DSM layer and Behavior layer. In the following of this section, we present the design of these two layers in turn.

4.1 DSM layer

We organize the DSM into three layers: message passing, shared registers, and shared-memory coordination algorithms. The key construct in the DSM is the shared register. The shared register masks the low-level details of network communication and provides uniform *Read/Write* interfaces for shared memory access. There are two essential issues to be addressed in the construction of a shared register:

- *Read/Write permissions*: We can assign which processes can read/write (the replica of) the shared register. In this way, we enable flexible interactions among the processes via the shared register. Widely used read/write permissions include: Single Writer Multiple Reader (SWMR) and Multiple Writer Multiple Reader (MWMR). In the former case, each process has its "own" register. Only the owner process can write its register, while all processes can read the registers. In the latter case, all processes can read and write the register.
- *Consistency semantics*: The shared register provides the *Read* and *Write* interfaces. The concrete implementation of *Read* and *Write* provides some consistency semantics of the register.

For example, in the case study in Section 5, each robot has its own SWMR atomic register. We employ the ABD algorithm [12] which maintains atomicity/linearizability [13] to implement the *Read* and *Write* interfaces of the register. Exemplar codes of the construction and use of the register are shown in Figure 4 and Figure 5, respectively.

Implementation of the consistency semantics of the shared register depends on the underlying wireless communication among the mobile robots. Mobile robots can communicate with each other via a variety

```
public class AbstractRegister {
   public abstract VersionValue read(Key key);
   public abstract void write(Key key, String val);
}

public class SWMRAtomicRegister extends AbstractRegister {
   private boolean flag;

   public SWMRAtomicRegister(boolean f) {
      this.flag = f; //Flag indicates whether the processor is the writer
   }

   public VersionValue read(Key key) {
      <contact a quorum of server replicas for the latest VersionValue>
      <extract the latest VersionValue>
      <write back the VersionValue into a quorum of server replicas>
      <return the latest VersionValue>
   }
         Implementation of the read primitive based on the ABD algorithm

   public void write(Key key, String val) {
      <construct new VersionValue>
      <write the value into a quorum of server replicas>
   }
}
         Implementation of the write primitive based on the ABD algorithm
```

Figure 4 Exemplar code of the SWMR atomic register.

```
public class WheeledRobot {
   private AbstractRegister register;
   private ArrayList<Behavior> behavior_arrray;
   private Arbitrator arbitrator;
   private ArrayList<Thread> sensor_thread_array;

   public WheeledRobot () {
      if (WriterRegisterList.WRITER_OF_R1) {
         this.register = new SWMRAtomicRegister(true);
      } else {
         this.register = new SWMRAtomicRegister(false);
      }
                  Construct replica according to the initial configuration
                  of the read/write permission

      Behavior move_along_wall = new Behavior();
      Behavior pass_gate = new Behavior();
      Behavior obtain_pass_permit = new Behavior();
      Behavior recovery = new Behavior();
      barray.add(move_along_wall);
      this.barray.add(pass_gate);
      this.barray.add(obtain_pass_permit);
      this.barray.add(recovery);
                        Construct behaviors and initial behavior array

      this.arbitrator = new Arbitrator(this.barray);

      UltrasonicSensorThread t1 = new UltrasonicSensorThread();
      PressureSensorThread t2 = new PressureSensorThread();
      CameraThread t3 = new CameraThread();
      sensor_thread_array.add(t1);
      sensor_thread_array.add(t2);
      sensor_thread_array.add(t3);
                  Construct sensor threads and initial sensor thread array
   }
}
```

Figure 5 Exemplar code of the WheeledRobot.

of wireless networks. To mask the heterogeneity of the underlying network, we abstract the networking operation into two message passing primitives: *send* and *receive*. Based on the message passing primitives, we can implement concrete read and write algorithms for the shared register.

Based on the shared registers with required consistency semantics, shared memory algorithms can be leveraged to enable versatile coordination among mobile robots. Detailed implementations of the shared memory algorithms are discussed in the design of the coordination behavior for the BBR-based robot in Section 5.

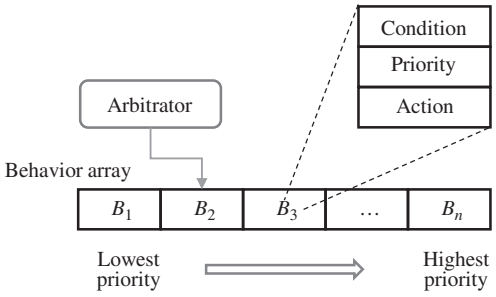

Figure 6 Behavior scheduling by the arbitrator.

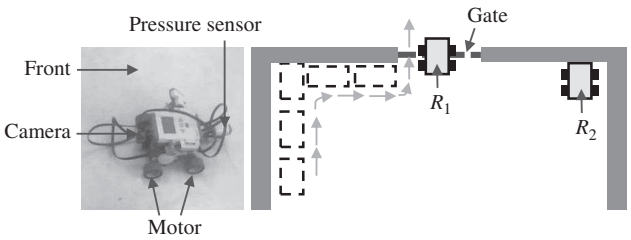

Figure 7 The mobile robot and the scenario.

4.2 Behavior layer

The Behavior layer is in charge of the management of behaviors and the scheduling of behaviors. When the environment data including physical environment data and coordination data is received, it should decide to schedule the behaviors according to their triggering conditions and priorities. We detail the management of behaviors and the scheduling in the below.

As mentioned in the before section, the coordination task is complex and we decompose the task into less complex behaviors. The recursively decomposed results construct a hierarchy. In implementation of the behavior hierarchy, each behavior is a object, the relation between complex behaviors and basic behaviors are mapped to the dependency between corresponding classes. We use the *BehaviorManager* to store the behaviors which can be scheduled. Note that only the complex behaviors take part in the scheduling.

Given the behavior hierarchy, we still need to decide the timing of each behavior. In the architecture, the *Arbitrator* is employed to decide the timing of each behavior. As shown in Figure 3, in this way, we achieve the separation of concerns by separating the design of the behaviors from the design of the scheduling strategy.

The behavior scheduling includes two primary issues. First, the arbitrator needs to evaluate the triggering condition of the behavior. Second, when there are multiple behaviors whose conditions are all true, the arbitrator needs to schedule the behaviors according to their priorities. We use the *Priority Manager* to take charge of managing the priorities of behaviors. In this way, when adding new behaviors, changing predefined priorities of behaviors will not involve each behavior class, so the behavior classes are loosely coupled.

When scheduling behaviors, the arbitrator always chooses the behavior with the triggering condition getting true and highest priority. An exemplar implementation is shown in Figure 6. The behaviors are placed in the behavior array in the order of decreasing priority, and the arbitrator iteratively scans the array from the highest priority behaviors to the lowest ones to find the scheduled behavior.

5 Case study: mutual exclusively passing the gate

5.1 Scenario description

We investigate how to use the CBBR to program two mobile robots to pass the gate during their patrol of certain space, as shown in Figure 7. Since the size of the gate only permits one robot to pass at the same time, the robots need to coordinate and pass the gate one by one. We use two LEGO NXT robots. The sensors and actuators used in the robot together with the explanations are listed in Table 1. The exemplar video of the robots mutual exclusively passing the gate can be found in http://cs.nju.edu.cn/yuhuang/robotcar.htm.

In the following, we first discuss how to design the coordination behavior and the physical behavior. Then we discuss how to design the arbitrator to schedule the behaviors to fulfill the task.

Table 1 Sensor and actuators of the robot

Sensor/Actuator	Description
Camera	Obtaining image of the ground
Pressure sensor	Checking collisions on the wall
Ultrasonic sensor	Sensing distance from the wall
Motor	Controlling movements of the robot

5.2 Coordination behavior

The gate is modeled as a critical section, where no two robots can be in it at the same time. Thus, when two robots Robot_1 and Robot_2 need to pass the gate at the same time, they need to coordinate. The triggering condition of the coordination is "when the robot arrives at the gate". This can be detected by the pressure sensor in the front of the robot.

The core of the robot coordination is the Bakery algorithm [16] for mutual exclusion over shared memories. The Bakery algorithm requires a pair of shared registers on each robot. On Robot_i $(i = 1, 2)$, we have shared registers $State[i]$ and $Number[i]$. Both registers are SWMR atomic registers, which can only be written by Robot_i and can be read by both robots. Two robots communicate via Bluetooth. The message passing over the wireless connection enables construction of the shared registers required for the coordination. Pseudo codes of the coordination algorithm are listed in Algorithm 1.

Algorithm 1 OBTAIN-PASS-PERMIT(i)

Initially
$Number[i] = 0$ and $State[i] = \text{FARFROMGATE}$, for $0 \leqslant i \leqslant n-1$;

⟨**Arrive at the gate**⟩
$State[i] \leftarrow \text{ATGATE}$;
$Number[i] \leftarrow 1 + Max(Number[0], \ldots, Number[n-1])$;
$State[i] \leftarrow \text{WAITTOPASSGATE}$;
if $i < 0$ **then**
  $j \leftarrow 0$;
end if
for j from 0 to $n-1$ **do**
  if $j! = i$ **then**
    wait until $State[j] = \text{WAITTOPASSGATE}$
    wait until $Number[j] = 0$ or $(Number[j], j) > (Number[i], i)$
    $State[i] \leftarrow \text{PASSGATE}$;
  end if
end for
⟨**Passing the gate**⟩
⟨**Finished**⟩
$Number[i] = 0$;
$State[i] \leftarrow \text{FARFROMGATE}$;

5.3 Physical behavior

In this scenario, the robot goes forward by default, hoping to arrive at the destination. In order to handle different situations during its move, we need to design more behaviors. The behavior hierarchy is shown in Figure 2. We have discussed the coordination behavior above, and now we discuss the physical behaviors in this case study:

• B_p^1*: going forward along the wall.* When the robot directly goes forward, we find that it has not the capability to ensure that it goes straightly in the vertical direction. After going forward for a certain amount of distance, it will always go bumping into the wall or go farther and farther away from the required route. To solve this problem, we make the robot constantly monitor the horizontal distance from the wall. If the robot goes nearer and nearer to the wall (or farther and farther from the wall) B_p^1

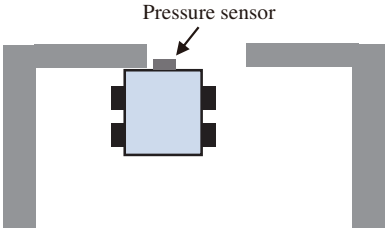

Figure 8 Stuck at the gate.

Table 2 Behavior array of the arbitrator

Behavior	B_p^1	B_p^2	B_c^1	B_p^3
Condition	Always true	Pass permit obtained	Arrive at the gate & no pass permit	Get stuck
Priority	1 (Lowest)	2	3	4 (Highest)
Action	Go forward	Pass the gate	Mutual exclusion	Recover from stuck

will control the robot to make a small turn and keep on going straightly forward. The combination of all these basic actions defines B_p^1. Here, B_p^1 is the routine physical behavior.

• B_p^2*: passing the gate*. According to the physical shape of the gate, the robot needs to "take a turn and go horizontally to the center of the gate" and "take a turn and go vertically to pass the gate". Here, B_p^2 is the objective physical behavior.

• B_p^3*: recovering from stuck*. During the operation, we find that sometimes the robot may get stuck. Specifically, the robot may be blocked by the gate, but it cannot detect this situation since the pressure sensor does not bump into the gate, as shown in Figure 8. In this case, the robot may keep on running the motor never knowing when to stop, but the robot actually always stays where it is. To handle this case, we use the camera to detect this situation. When the motor is running, images of the ground do not change, it will trigger the behavior "recovering from stuck" (B_p^3). Behavior B_p^3 requires the robot to "move backward a little" and "make a left/right turn". Here, B_p^3 is a fault-handling physical behavior.

5.4 Behavior scheduling by the arbitrator

The behavior array to be scheduled by the arbitrator is listed in Table 2. We exemplify the scheduling strategy by explaining how the arbitrator of Robot_1 schedules its behaviors to correctly pass the gate. During fulfilment of the task, Robot_1 will undergo the following states (listed in Table 3):

• *Far from the gate*: At the beginning, the robot is far from the gate. Only the triggering condition of B_p^1 is true. The arbitrator schedules B_p^1 and the robot goes forward along the wall until it arrives at the gate.

• *Arrive at the gate*: When the robot arrives at the gate (its pressure sensor in the front touches the wall), the triggering conditions of both B_c^1 and B_p^1 are true. Thus, B_c^1 is scheduled since it has higher priority. The robot coordinates with the other robot to decide which one can obtain the permit to pass the gate.

• *Get pass permit*: We assume that Robot_1 coordinates to obtain the permit to pass the gate first. Thus, the condition of B_p^2 becomes true, and the robot passes the gate.

• *Recovery from stuck*: During pass of the gate, if Robot_1 gets stuck, B_p^3 will be triggered to get it recovered from the stuck.

• *Leave the gate*: After having passed the gate, Robot_1 keeps going forward along the wall.

6 Performance measurements

In this section, we quantify the performance of CBBR in our case study above. In the case study, the robot records the states of each sensor and states of the coordination. It also records the memory consumption

Table 3 Conditions of the behaviors in Robot_1

State	B_p^1	B_p^2	B_c^1	B_p^3
Far from the gate	√	×	×	×
Arrive at the gate	√	×	√	×
Get pass permit	√	√	×	×
Get stuck	√	×	×	√
Leave the gate	√	×	×	×

Clock	Coordinate	State	Pressure sensor	Ultrasonic sensor	Coordination state	Voltage	Memory
12	(−0.002034402,66.20905)	NORMAL	FALSE	15	NORMAL	7.574	16
14	(−7.397828E−4,101.2753)	NORMAL	FALSE	15	NORMAL	7.616	18
16	(0.02071392,140.0199)	NORMAL	FALSE	15	NORMAL	7.574	20
18	(0.04605147,165.5227)	NORMAL	TRUE	15	MEET	7.546001	21
19	(0.06380627,188.0828)	NORMAL	TRUE	15	MEET	7.546001	23
21	(0.08599974,212.6047)	HITWALL	TRUE	15	AVOID	6.937	25

Figure 9 Exemplar log data.

Table 4 Time consumption

Time (ticks)	Exp1	Exp2	Exp3	Exp4	Exp5	Average
T_{coor}	3	2	1	2	2	2.0
T_{turn}	4	3	4	5	4	4.0
T_{pass}	241	229	228	224	235	231.4

and the voltage of the battery. A piece of exemplar log data is shown in Figure 9. In the experiments, we study the time, memory, and energy consumption. Detailed discussions on the experimental results are presented below.

6.1 Time consumption

In this experiment, we investigate the time consumption for robot coordination based on CBBR. We compare the time for coordination with that for different physical behaviors. Specifically, we denote the time for coordination by T_{coor}. Here T_{coor} stands for the time from the robot (which gets the pass permit first) starting the coordination to obtaining the permit to pass. We compare T_{coor} with the time for making one left/right turn (T_{turn}) and with the time for passing the gate (T_{pass}). Table 4 shows the time for each operation in 5 experiments.

Note that one tick of time lasts 100 ms. The coordination is finished within 1 to 3 ticks. In comparison, one basic physical behavior (making a turn) takes 3 to 5 ticks, while a more complex physical behavior (passing the gate) takes much more time (224 to 241 ticks). On average, T_{coor} is about half of T_{turn} and 0.86% of T_{pass}.

We find in this experiment that, during fulfilment of the task, the coordination imposes limited time consumption, compared to that for the physical behaviors. Thus, in order to reduce the total time consumption for fulfilling the task, better design and arrangement of the physical behaviors is the more critical issue. Even if we spend certain time for coordination, it can potentially be well compensated by the reduction in the time consumption for physical behaviors.

6.2 Memory consumption

We then evaluate the memory consumption. We conduct 5 experiments to compare the memory consumption between two robots passing the gate. We use Robot_1 to denote the robot which passes the gate first, and use Robot_2 to denote the other robot. The memory consumption of these two robots is

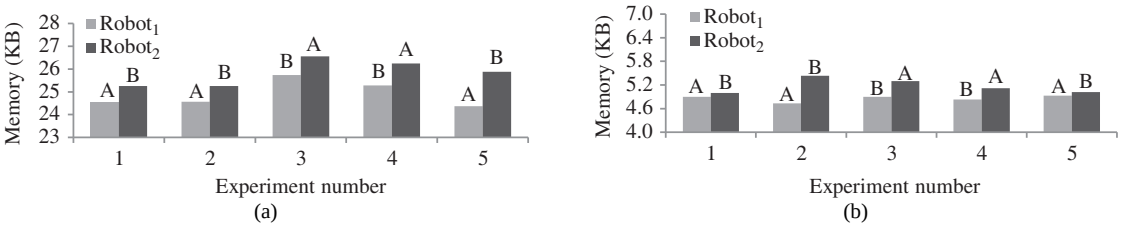

Figure 10 Memory consumption. (a) Average; (b) standard deviation.

denoted by M_1 and M_2. We also conduct a baseline experiment, where one robot directly passes the gate without any coordination. In this way, we investigate the extra memory consumption mainly imposed by the DSM-based coordination.

From Figure 10(a), we find that though the memory consumption for different experiments varies, M_2 is more than M_1 in all the 5 experiments. Similarly, the standard deviation of M_2 is also more than that of M_1 in all the 5 experiments. This is mainly because Robot_2 needs to persistently coordinate to apply for the permit to pass the gate (via the Bakery algorithm in the coordination behavior B_c^1), while Robot_1 gets the permit first and can quickly go on with the following physical behaviors. When we focus on the phase of mutual exclusion, the average memory consumption on Robot_2 is 14% higher than that on Robot_1. The higher memory consumption for mutual exclusion also results in the higher deviation.

In the baseline experiment where the robot passes the gate without any coordination, we denote the memory consumption by M_3. The average of M_3 is 20.37 KB, while the average of M_1 and M_2 is 24.90 KB and 25.84 KB respectively. M_1 and M_2 are more than M_3 by 18% and 21%, respectively, which means that the extra memory consumption for the coordination is quite limited.

6.3 Battery voltage statistics

We study the voltage statistics of two robots, and Figure 11 (a) and (b) show the voltage statistics of Robot_1 and Robot_2 respectively. We find that the voltage of Robot_1 remains stable for most of the time. More importantly, we find that there are a few sharp decreases. Further investigating the robot states at the time of the sharp decreases from the experiment log of Robot_1, we find that all the sharp decreases correspond to the physical behavior of "making a turn". Making a turn requires more forces than going forward smoothly. This requires stronger current in the motor, which significantly increases the energy consumption, and impacts the voltage.

In comparison, the coordination behaviors which mainly consist of computing and communication in the cyber space result in little impact on the voltage. They consume much less energy than physical behaviors, e.g., making turns or changing from going forward to going backward. This shows that during the operation of the mobile robot, the control actions concerning the physical spaces dominate the overall energy consumption. The DSM-based coordination contributes to a limited portion of the overall energy consumption. Thus the more important issue in reducing the energy consumption is to design more efficient robot control strategies, potentially based on better coordination among the robots.

We find similar pattern concerning the voltage statistics of Robot_2. The main difference is that the voltage decrease of Robot_2 is much less than that of Robot_1. This is mainly because Robot_1 keeps working without obvious stop, while Robot_2 "rests" for a while when it waits for Robot_1 to pass first. Compared to Robot_1, the same burden of coordinating as well as passing the gate (making two turns and going forward) is "amortized" to a longer period of time. The battery recovers during the rest when there is little current going out. This results in the less sharp changes in the battery voltage.

6.4 Discussion

Based on the performance measurements in our case study, we find that the DSM-based coordination among the mobile robots does not impose significant burden in terms of time, space, and energy consumption on the mobile robot. Most resources are consumed in actions concerning the physical space.

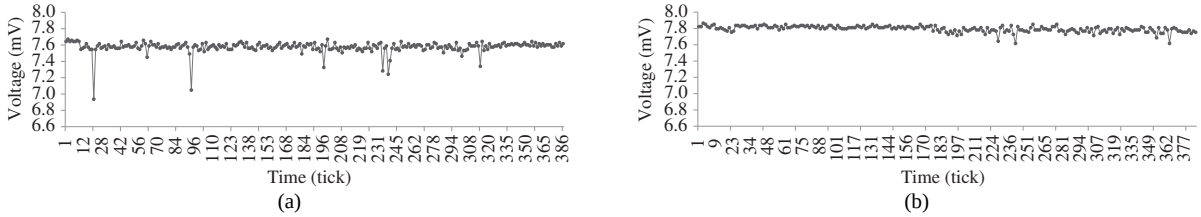

Figure 11 Battery voltage changes. (a) Robot$_1$; (b) Robot$_2$.

Thus, the key issue here is to better design and schedule the physical behaviors probably by effective coordination. Even if we pay a little more resources on the coordination, this may potentially be well compensated by the reduction in the cost for physical behaviors.

7 Related work

BBR is widely used to program mobile robots. Using BBR, the complex task for the robot is decomposed into multiple behaviors. BBR achieves the separation of concerns in mobile robot control by separating the design of robot behaviors from the design of the behavior scheduling strategy [17]. Though BBR simplifies the programming of one single mobile robot, it lacks the support for programming robot coordination.

So far, a number of solutions have been proposed to support the coordination among multiple robots. pub/sub is an interaction scheme where an event notification services decouples the subscribers and the publishers. Similar to the tuple space scheme, the deep decoupling of potentially a great number of publishers/subscribers comes at the cost of maintaining an event notification service (usually implemented in a middleware). In our scenario of coordinating mobile robots, we do not require the deep decoupling among the robots but are more concerned of the cost include by the coordination scheme. Mottola et al. [18] presented a team-level programming model for collaborative sensing drones. In team-level programming, users express sophisticated collaborative sensing tasks without resorting to individual addressing and without being exposed to the complexity of concurrent programming. The team runtime system automatically chooses the actions for each drone that will collaboratively accomplish the sensing tasks. It facilitates implementing correct and complete collaborative drone applications. However, team-level programming cannot address individual drones, it cannot express actions that involve direct interactions between drones. In our scenario, each robot is more independent and autonomous. Each robot needs to handle the complex situation it faces. Thus we need strong programming support to precisely control the robot behavior from the perspective of one independent and autonomous robot.

Besides, there are some other work to solve coordination indirectly based on low-level communication primitives. Matarić et al. [7] presented a strategy whereby the robots communicate with each other to take turns controlling the actions at each time step in order to coordinate the delivery of the box to the goal. Marcolino and Chaimowicz [19] used decentralized coordination to allow swarms of robots to navigate and synthesize patterns, overcoming local minima in environments containing unknown obstacles. Jäger and Nebel [20] described a decentralized scheme to avoid collisions and deadlocks among multiple mobile robots. Because in these works, the coordination is directly built based on low-level communication primitives, the coordination and the low-level robot control system are tightly coupled.

In order to decouple the coordination and the robot control system, some other work use the finite state machine to support the coordination among multiple robots. Kube and Bonabeau [21] described a case study of pushing a box with multiple robots that are controlled by finite state machine. Klotzbücher and Bruyninckx [22] described a hierarchical finite state machine model used for robot coordination. However, the increase in the complexity of coordination may result in significant growth in the number of states in the state machine. Also, the state machine-based coordination cannot be seamlessly integrated into the underling BBR-based robot control.

In order to naturally extend BBR to enable the programming of one coordinating robot, where the design of coordination and that of control of a robot are separated, shared tuple spaces [11] supports

generative communication among computing entities. The communication is generative since the tuple generated by certain entity has an independent existence in the tuple space. The computing entities are thus highly decoupled. Each entity participates in the coordination only by accessing data, without knowing or interacting with other entities. Though such data-driven coordination and generative communication greatly simplifies the programming of coordination, they impose communication cost on the underlying network system. Such cost may be prohibitive in mobile wireless networks. CBBR naturally extends BBR by using DSM. In comparison with the tuple space, DSM-based coordination relies on message-passing among entities. It provides the illusion of a shared memory on top of message-passing. The illusion built by the DSM greatly simplifies the coordination logic and programming coordination. Though the DSM-based coordination does not provide decoupling, it supports flexible coordination. The key construct in the DSM is the shared variables. The robot accesses shared variables with pre-defined access patterns (e.g., Single-Writer-Multiple-Reader or Multiple-Writer-Multiple-Reader). Different replicas of the shared variable are automatically synchronized according to pre-specified consistency semantics (e.g., atomicity or causal consistency). The fine-grained tuning of access patterns and consistency semantics enables flexible coordination.

CBBR uses DSM [12] to isolate the complexity of coordination from the complexity of robot control. Many coordination problems, e.g., mutual exclusion [16, 23], leader election [24–26], consensus [27, 28], and resource allocation [29, 30], have been studied in distributed computing. The distributed algorithms for these problems over the shared memory model can be leveraged over our CBBR framework to enable versatile robot coordination. Meanwhile, the robot control system only needs to integrate the distributed shared memory. It is shielded from implementing the complex of the upper layer coordination.

8 Conclusion

In this paper, we propose the CBBR scheme to program the coordination among mobile robots. CBBR uses distributed shared memory to leverage various distributed algorithms to enable coordination. The distributed shared memory can also be easily integrated into the BBR-based mobile robot control. We design and implement CBBR and conduct a case study where we program a group of two mobile robots to pass the gate in a mutual exclusive way using CBBR. The performance measurements demonstrate the cost-effectiveness of coordination based on CBBR.

In our future work, we will implement more coordination algorithms over the shared memory model and program the mobile robots to fulfill more complex tasks. We will also investigate how to apply the shared memory model to enable coordination on other platforms, e.g., the coordination among smart phones.

Acknowledgements This work was supported by National Basic Research Program of China (973) (Grant No. 2015CB352202) and National Natural Science Foundation of China (Grant Nos. 61272047, 91318301, 61321491). This work was also partially supported by Tencent, Inc.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Antonelli G, Leonessa A. Underwater robots: motion and force control of vehicle-manipulator systems. IEEE Control Syst Mag, 2008, 138–139
2 Borenstein J, Koren Y. Real-time obstacle avoidance for fast mobile robots. IEEE Trans Syst Man Cybern, 1989, 19: 1179–1187
3 Fujita M, Kageyama K. An open architecture for robot entertainment. In: Proceedings of the 1st International Conference on Autonomous Agents. New York: ACM, 1997. 435–442
4 Zhan G, Shi W. Lobot: Low-cost, self-contained localization of small-sized ground robotic vehicles. IEEE Trans Parall Distrib Syst, 2013, 24: 744–753
5 Russell R A, Thiel D, Deveza R, et al. A robotic system to locate hazardous chemical leaks. In: Proceedings of IEEE International Conference on Robotics and Automation, Nagoya, 1995. 1: 556–561

6 Wurman P R, D'Andrea R, Mountz M. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. AI Mag, 2008, 29: 9
7 Matarić M J, Nilsson M, Simsarin K T. Cooperative multi-robot box-pushing. In: Proceedings of 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems: Human Robot Interaction and Cooperative Robots, Pittsburgh, 1995. 3: 556–561
8 Burgard W, Moors M, Fox D, et al. Collaborative multi-robot exploration. In: Proceedings of IEEE International Conference on Robotics and Automation, San Francisco, 2000. 1: 476–481
9 Wawerla J, Vaughan R T. A fast and frugal method for team-task allocation in a multi-robot transportation system. In: Proceedings of IEEE International Conference on Robotics and Automation, Anchorage, 2010. 1432–1437
10 Brooks R A. A robust layered control system for a mobile robot. IEEE J Robotic Autom, 1986, 2: 14–23
11 Gelernter D. Generative communication in Linda. ACM Trans Program Lang Syst, 1985, 7: 80–112
12 Attiya H. Robust simulation of shared memory: 20 years after. Bull Eur Assoc Theor Comput Sci, 2010, 100: 99–113
13 Lamport L. On interprocess communication. Distrib Comput, 1986, 1: 86–101
14 Ahamad M, Neiger G, Burns J E, et al. Causal memory: definitions, implementation, and programming. Distrib Comput, 1995, 9: 37–49
15 Lynch N A. Distributed Algorithms. San Francisco: Morgan Kaufmann, 1996
16 Lamport L. A new solution of Dijkstra's concurrent programming problem. Commun ACM, 1974, 17: 453–455
17 Arkin R C. Behavior-Based Robotics. Cambridge: MIT Press, 1998
18 Mottola L, Moretta M, Whitehouse K, et al. Team-level programming of drone sensor networks. In: Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems. New York: ACM, 2014. 177–190
19 Marcolino L S, Chaimowicz L. No robot left behind: Coordination to overcome local minima in swarm navigation. In: Proceedings of IEEE International Conference on Robotics and Automation, Pasadena, 2008. 1904–1909
20 Jäger M, Nebel B. Decentralized collision avoidance, deadlock detection, and deadlock resolution for multiple mobile robots. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, 2001. 3: 1213–1219
21 Kube C R, Bonabeau E. Cooperative transport by ants and robots. Robot Auton Syst, 2000, 30: 85–101
22 Klotzbücher M, Bruyninckx H. A lightweight real-time executable finite state machine model for coordination in robotic systems. Technical Report, 2007
23 Dijkstra E W. Solution of a problem in concurrent programming control. Commun ACM, 1965, 8: 569
24 Peterson G L. An $o(n \log n)$ unidirectional algorithm for the circular extrema problem. ACM Trans Program Lang Syst, 1982, 4: 758–762
25 Chang E, Roberts R. An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun ACM, 1979, 22: 281–283
26 Hirschberg D S, Sinclair J B. Decentralized extrema-finding in circular configurations of processors. Commun ACM, 1980, 23: 627–628
27 Fischer M J, Lynch N A, Paterson M S. Impossibility of distributed consensus with one faulty process. J ACM, 1985, 32: 374–382
28 Dolev D, Lynch N A, Pinter S S, et al. Reaching approximate agreement in the presence of faults. J ACM, 1986, 33: 499–516
29 Dijkstra E W. Hierarchical ordering of sequential processes. Acta Inform, 1971, 1: 115–138
30 Lynch N A. Upper bounds for static resource allocation in a distributed system. J Comput Syst Sci, 1981, 23: 254–278

Appendix A I/O automata of CBBR

Appendix A.1 Shared register

The I/O automata of the shared register includes two transitions, *read* and *write*. Both *read* action and *write* action execute operations according to the read/write permission and the consistency semantics.

Shared Register

Actions
Input/Output:*read*, *write*

Transitions
read
Effect:
/ ∗ *operations according to read/write permission and consistency semantics* ∗ /

write
Effect:
/ ∗ *operations according to read/write permission and consistency semantics* ∗ /

Appendix A.2 Physical behavior

Formally, the environment data variables $v_1, \ldots, v_k$ are persistently updated by the sensors, and the coordination status data variables $r_1^{(i)}, \ldots, r_m^{(i)}$ of robot i are updated by the DSM. The robot has p actuators, where each actuator may have multiple APIs, e.g.: $acutator_1.action_1()$, $actuator_2.action_2(), \ldots$. The physical behavior can be defined below.

Physical Behavior b_i

Variables: $v_1, \ldots, v_k, r_1^{(i)}, \ldots, r_m^{(i)}$

Actions of i
Input:$check\text{-}flag_i$
Internal:$action_i$
Output:$finish_i$

States of i
$s \in \{check\text{-}flag, action, finish\}$**,** $initially\ finish$

Transitions of i
$check\text{-}flag_i$
  Effect:
    if $C(v_1, \ldots, v_k, r_1^{(i)}, \ldots, r_m^{(i)}) = true$ **then**
      $s := action$
      $b_i.flag := true$
    end if

$action_i$
  Precondition:
    $s = action$
  Effect:
    $\ldots$
    $actuator_1.a_1()$
    $\ldots$
    $actuator_1.a_2()$
    $\ldots$
    $s := finish$

$finish_i$
  Precondition:
    $s = finish$
  Effect:
    $b_i.flag := false$

Specifically, in the description below, the $flag$ indicates whether the condition of b_i is true. There are three actions, $check\text{-}flag_i$, $action_i$, $finish_i$. When the arbitrator scans the behavior b_i, action $check\text{-}flag_i$ is executed. If the condition is true, it executes $action_i$, which contains different actions of actuators. After the actuators finish executing actions, the $flag$ is set as false and it waits for the next round of scanning by the arbitrator.

Appendix A.3 Coordination behavior

The coordination behaviors are implemented as the shared memory algorithms over the DSM. The algorithms includes read/write operations of shared variables. Provided with read/write operations of shared variables, the I/O automata of coordination behavior can be described below. It has three states and three actions. When the arbitrator scans the behavior b_i, action $check\text{-}flag_i$ is executed. If the condition is true, it executes $action_i$, which calls the coordination algorithm. After finishing the algorithm, it sets the flag of b_i as false.

Appendix A.4 Arbitrator

As an exemplar implementation, the behaviors $b_1, \ldots, b_n$ are placed in an array in the order of decreasing priority. At first, the arbitrator is in the *terminate* state. When the robot starts running, the arbitrator changes the state from *terminate* to *schedule*. It scans the array from the highest priority behaviors to the lowest priority ones. When behavior b_i is scanned, it sets the state of b_i as $check\text{-}flag_i$ and b_i executes actions. If the arbitrator finds the flag of b_i is true, the behavior b_i is scheduled and it starts the next round of scanning.

Coordination Behavior b_i

Variables: $v_1, \ldots, v_k, r_1^{(i)}, \ldots, r_m^{(i)}$

Actions of i
Input$check\text{-}flag_i$
Internal$action_i$
Output$finish_i$

States of i
$s \in \{check\text{-}flag, action, finish\}$**,** *initially* $finish$

Transitions of i
$check\text{-}flag_i$
Effect:
if $C(v_1, \ldots, v_k, r_1^{(i)}, \ldots, r_m^{(i)}) = true$ **then**
$s := action$
$b_i.flag := true$
end if

$action_i$
Precondition:
$s = action$
Effect:
/***read/write operations of shared registers according to the coordination algorithm***/
$s := finish$

$finish_i$
Precondition:
$s = finish$
Effect:
$b_i.flag := false$

Arbitrator

Variables: $b_1, \ldots, b_n$, *in the order of decreasing priority*

Actions
Input:run
Output:$schedule$

States
$s \in \{terminate, schedule\}$, *initially* $terminate$

Transitions
run
Effect:
$s := schedule$

$schedule$
Precondition:
$s = schedule$
Effect:
for all $b_i, 1 \leqslant i \leqslant n$ **do**
$b_i.s := check\text{-}flag_i$
if $b_i.flag = true$ **then**
continue
end if
end for