Programming Large-Scale Multi-Robot System with Timing Constraints

Shan Jiang, Jiannong Cao, Yang Liu, Jinlin Chen, Xuefeng Liu
Department of Computing, The Hong Kong Polytechnic University, Hong Kong
{cssjiang, csjcao, csyangliu}@comp.polyu.edu.hk, jinlin.chen@connect.polyu.hk, csxfliu@gmail.com

***Abstract*—Recently years, research in multi-robot systems has attracted increasingly attentions. One important research topic is to design programming models that can facilitate the developers to programme large-scale multi-robot systems. However, existing works fail to manage the robots to perform tasks with real-time requirements. To address this issue, we propose a new programming model called RMR (Real-time Multi-Robot). RMR is a logic programming model with real-time support. On the basis of the logic programming paradigm, RMR allows the code for multi-robot system to be written from a global perspective, rather than managing a large collection of independent robots. Moreover, RMR allows developers to set timing constraints on the behaviors of an ensemble of robots, which is not implemented by state of the art. After designing RMR, we further develop a compiler and a runtime system for distributed execution of RMR programs. To evaluate the performance of RMR, we deploy it in a simulator and a test-bed, and then demonstrate RMR based on several applications. Our results indicate that RMR greatly facilitates implementing correct collaborative multi-robot applications.**

I. Introduction

The remarkable progress of robotics technology has made it feasible to deploy a large number of inexpensive robots with complicated tasks. The robots together form a multi-robot system (MRS), which has better reliability, flexibility, scalability and versatility than a single-robot system. There has been quantities of applications for MRS, such as multi-robot exploration, multi-robot surveillance and multi-robot manipulation. However, the management of the robots is a challenging issue. Among the difficulties towards managing MRS, one of the most important is the lack of dedicated tools. In particular, one problem that is significant but has received little attention is the programmability. To be specific, a scalable programming model is required to program MRS containing thousands or even millions of robots. Moreover, the programming model is supposed to support setting timing constraints on the behaviors of the robots, which is required in many real-time multi-robot applications.

Traditionally, robots are programmed using an imperative programming paradigm. Such programming tasks are expensive in terms of time consumption and code complexity. For example, the robots in our lab are programmed with embedded C, which is an imperative programming language. There are some other domain-specific imperative programming models such as TinyOS [1], Swarm [2], Paintable Computing [3], and CAs [4], all of which focus on the behavior of individual devices instead of the aggregate.

General multi-robot applications require coordinated movements with real-time decision making capabilities in an unstructured environment. Hence, with respect to programming a group of robots, the tasks of communication and coordination needs to be abstracted instead of being explicitly written for each of the robots. There have been several research efforts to develop such programming models. Unfortunately, all of them have focused on a dedicated programming model for a specific application.

Early success in the development of programming models that enable programmers to think on a macroscale arose from the field of overlay networks and sensor networks. P2 [5] and SNLog [6] showed that logic programming approach could be used to allow an ensemble to be programmed as a whole. Other such programming models for sensor networks include Hood [7], TinyDB [8] and Regiment [9]. However, these sensor network programming models are limited by their focus on sensing and data gathering without attending to actuation and control. Moreover, they presume a static network of immobile nodes which changes infrequently due to node failures. A notable exception is Pleiades [10], which could be used in situations with dynamic network topologies of sensor networks. But owing to adoption of a programming style similar to OpenMP [11], it eventually leads the programmers to focus on individual modules instead of the whole ensemble.

Inspired by the logic programming approach for sensor networks, researchers tried to extend their application for mobile robots. For example, Proto [12], which is effective for programming stationary sensor-actuator networks as a whole, has been extended to mobile robots as Protoswarm [13]. Protoswarm is a functional language that uses *Amorphous Medium Abstraction* [14] for programming MRS. LDP [15] is derived from a method for distributed debugging but is originally designed for modular robotics. While it works well in highly dynamic systems, it can lead to excessive messaging in more static environments.

Meld [16] [17] is another logic programming language for modular robots that enables the programmer to specify high-level logic of what is to be decided or achieved, and leaves the low-level details of data manipulation and communication to the implementation of the programming language. However, it is restricted to applications on modular robots which are independently executing modules robots where inter-robot communication is limited to immediate neighbors. Moreover, for many real-time multi-robot applications, besides being

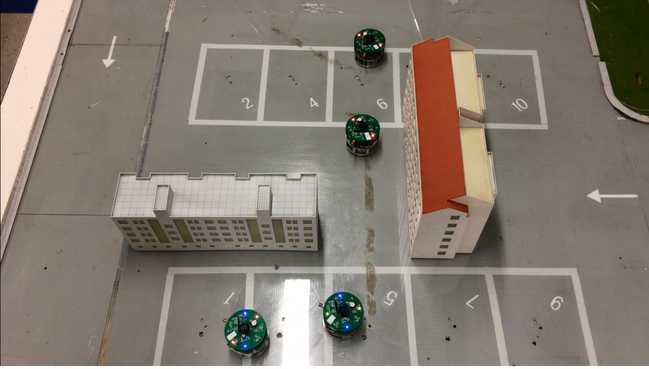


Fig. 1. Example application: multiple robots are passing through a narrow corridor

logically correct, satisfying certain temporal constraints is a hard demand to successfully achieve final targets with high precision. Meld doesn't provide any mechanism to support real-time scheduling of the robots.

In this work, we proposed and designed a new programming model for MRS, called RMR. It follows the logic programming paradigm, which enables it to achieve high scalability. Moreover, it allows developers to specify timing constraints on the behaviors of the robots, such as setting deadlines and identifying the time orders of actions. To support distributed execution of RMR programs, a compiler and a runtime system are developed for RMR. The compiler is able to convert the RMR programs into executable byte-codes, and then distribute the byte-codes to each robot. The runtime system is responsible for interpreting and executing the byte-codes. To evaluate the performance of RMR, we deployed RMR in a simulator and a realistic test-bed, and then developed several example applications. Fig. 1 shows one of the applications utilizing RMR, in which multiple robots cooperate with each other to pass through a narrow corridor. Our main contributions are:

- We designed and proposed a new programming model called RMR for MRS. RMR allows programmers to specify timing constraints for large-scale MRS in a easy-to-use fashion.
- We implemented and deployed RMR in both a simulator and a realistic test-bed. To support distributed execution of RMR programs, we developed a compiler and a runtime system. Furthermore, we did solid real-world experiments to evaluate the performance of RMR.

The reminder of the paper is structured as follows. Section II introduces the design philosophy and main features of RMR. In Section IV, we first describe the compiler of RMR , and then present the mechanisms in our runtime system to support the distributed execution of RMR programs. The deployment of RMR in both simulator and realistic test-bed and the evaluation in introduced in Section V. Section VI concludes the paper with some future improvement directions.

II. Design Principle

Logic programming has a long history and there exist a number of variants of logic programming languages. Commonly, a logic programming language encompasses a set of facts and rules as its basic elements. Facts can be used to specify system states, sensed physical events, system configuration, etc, while rules allow programmers to describe how the system evolves. In practical applications, it is important to design good abstractions to mask the complexity of programming MRS, and meanwhile provide real-time guarantee for the coordination of the MRS. To this end, the following principles are considered on designing RMR.

A. Ensemble-level abstraction

With respect to a group of robots assigned with a task, it is nature to think about what the robot ensemble as a whole should do. This leads us to consider the deign principle of ensemble-level abstraction. The entire MRS can be viewed as a single and monolithic unit while programmers write RMR program. RMR enables a developer to think about what the whole set of robots should do in a global and easy-to-understand perspective, and hide the detailed and complex implementation of how to do. This greatly simplifies the programming process, and benefits the developers to program MRS with a large number of robots. This abstraction technique is also used in programming modular robots [16], and is referred to as macro-programming in the area of wireless sensor networks [18].

B. Combination of forward and backward reasoning

In logic programming, forward reasoning and backward reasoning are two main methods of reasoning. Backward reasoning starts with a list of goals and works backwards from the consequent to the premises to see if the consequent is available. Backward reasoning is often used to specify the direction of reasoning, e.g. in Prolog [19]. One the contrary, forward reasoning starts with the available premises and uses rules to derive new facts until the goal is reached. Forward reasoning is often used to speed up the execution of programs, e.g., in Meld [20].

Both backward reasoning and forward reasoning have advantages and disadvantages. It is natural to expect that the general performance of the system could be improved by combining the two kinds of reasoning. RMR successfully combines forward and backward reasoning in the execution of RMR programs. In this way, RMR will derive facts with specific direction while guarantee the execution speed of the RMR programs.

C. Declarative specification of timing constraints

In MRS, it is likely that a series of reasoning steps are involved in order to perform a job. When a developer writing a program, he mainly concern about the global deadline of finishing the entire job rather than the local deadlines of individual reasoning steps. For example, in wireless sensor-actuator networks, the system need to respond in real-time to the physical world events captured by sensors. Actually, this is a complicated sensing-decision-actuation process involving many intermediate steps, such as data aggregation for complex

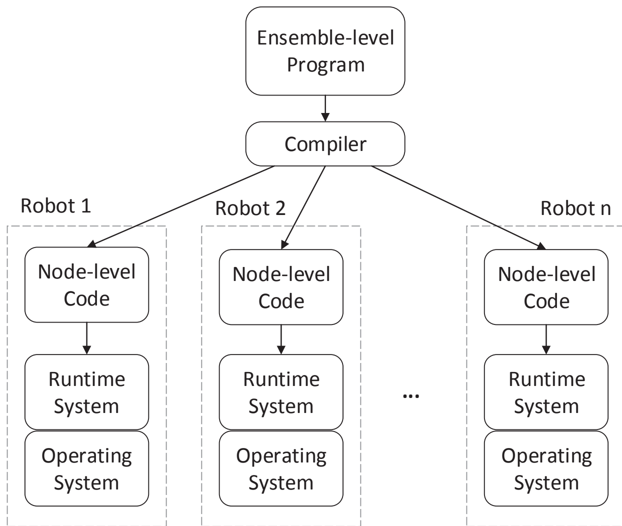


Fig. 2. Overview of RMR Compilation

event detection. It is highly desirable that the developers only need to describe the global deadline of the entire job and need not to worry about how the local deadlines can be meet at the intermediate steps. This style is referred to as a declarative specification of timing constraints, which has been incorporated in RMR.

With the above principles, RMR language will create a new way of writing programs for MRS. An overview of programming steps is shown in Fig. 2. The highest level is the RMR program, which is written by programmers and obeys a centralized, ensemble-level abstraction. A RMR compiler in the middle level is able to convert the ensemble-level program into node-level byte-codes ran on individual physical robots. To support the execution of such node-level code, a runtime system embedded in the operating system is needed. The runtime system plays the key role in ensuring the efficiency of distributed reasoning and the satisfaction of deadline requirements. In the following sections of this paper, we will introduce these components one by one.

III. Language Specification

The design principles in Section II enable us to design a language that can greatly simplify programmers' thinking processes and reduce their development efforts. In the following, we will describe the detailed specification of RMR. RMR comprises the following main elements.

A. Variable, Constant and Boolean Expression

RMR follows the conventions of logic programming model when defining variables and constants. Variable names begin with an uppercase letter, whereas constant names begin with a lowercase letter. Moreover, RMR supports boolean expressions that can be calculated into boolean values (true or false).

B. Fact

Facts are generally in the form of predicates, and they can return boolean values according to the results whether the facts are satisfied or not. Similar to constants, the identifier of a fact should begin with a lowercase letter. A fact generally has a predicate symbol (or name) and can take some arguments (variables) as input. The first argument of a fact must be a robot, which indicates the owner of the fact. Typically, a fact can be used to represent a system state, a detected event, or a relation between its arguments. In RMR, facts are divided into three categories: persistent fact, temporary fact and goal fact.

- Persistent facts refers to those that hold permanently, such as ID of a robot. They can not be consumed along the program lifetime, and must be declared with a bang mark '!', which means "of course".
- Temporary facts are those representing a temporary state and can be consumed. For example, `movealong(A,L)` is a temporary fact, which means an intermediate state that robot `A` is moving along the line `L`.
- Goal facts represent the goals of the program that must be satisfied if possible. They must be declared with a question mark '?', which means "why not" or "to be achieved".

Persistent fact and temporary fact are generally used in logic programming language while goal fact is proposed in our programming language and is one of the new elements. Goal facts enable programmers to specify intermediate states in their codes. This function is not supported in other logic programming language.

```
// initial fact that R is in s
location(R, s).

// goal fact that go to middle if possible
?location(R, middle).

// rule 1: if R is in s, then go to middle
location(R, L), L = s
  -o location(R, middle).

// rule 2: if R is in s, then go to t
location(R, L), L = s
  -o location(R, t).

// rule 3: if R is in middle, then go to t
location(R, L), L = middle
  -o location(R, t).
```

Fig. 3. RMR program to move a robot from area `s` to area `t` passing through area `middle` without *action*

For example, if we want a group of robots to go from area `s` to a specified area `t` while passing by area `middle` if possible. In traditional logic programming language for robotic system, such as Meld [16], we can write that there is an initial fact that the robots are in area `s`. Furthermore, there are three rules. The first one is that if the robots are in area `s`, then they can go to area `middle`. The second one is that if the robots are in area `s`, then they can go to area `t`. The last one is that if the robots are in area `middle`, they can go to area `t`. Since

there are two areas `middle` and `t` to which the robots can go from `s`, the robots will randomly choose one of them to go to. This is true because we can not specify the procedure how the robots go from `s` to `t` in traditional logic programming language.

However, we can achieve it by adding a new element called goal fact into logic programming language. The program is shown in Fig 3. In the program, we first write down the initial fact and the three rules. Furthermore, we add a goal fact `?location(R, m)`, which means that "Why not go to the area middle if possible". In this way, when the program runs, it will only choose the the first rule to apply and go to location `middle` if possible.

C. Action

As we have seen in Section III-B, there are nothing involved with actuation and motion in the robotics system. To support the motion control for realistic robots in language, we introduce the concept of *action*. An action also has a predicate symbol and can take some arguments as input. The first argument of an action also must be a robot, which is the executor of the action. Unlike fact, actions must be declared as an action type in previous, so that the actions will not stored and will be consumed immediately. The introduction of "action" connects the programming language and the realistic robot. A fact will only influence the execution of the program, while an action can have effect on the physical environment and the robot itself. To be specific, when an action is derived, there will be some underlying function call according to the action rather than being inserted in to the database of the facts. The programmers are able to define actions on their own.

```
type action moveto(robot, area).

location(R, s).
?location(R, middle).

// use action moveto instead
location(R, L), L = s -o moveto(R, middle).
location(R, L), L = s -o moveto(R, t).
location(R, L), L = middle -o moveto (R, t).
```

Fig. 4. RMR program to move a robot from area `s` to area `t` passing through area `middle` with *action*

```
function moveto(Area p) {
    // control the robot to go to Area p
    ...
    // insert the location fact back
    VM.insert_fact(type_location, p)
}
```

Fig. 5. The function *moveto* written in the operating system of a robot

We add actions in the example mentioned in Section III-B and the modified program is shown in Fig. 4. We will explain how the robotics system evolves in the following. At the head of the program, a predicate of `moveto` is declared as a new action type. Firstly, there is a initial fact `location(R, s)` meaning that the robot is in the area `s`. Then the runtime system will apply the first rule, delete the fact `location(R, s)`, and derive an action `moveto(R, middle)`. The action `moveto(R, middle)` will not be inserted into the database of the facts but will call the underlying function `moveto` in the operating system, which is demonstrated in Fig. 5. The function `moveto` will control the robot to go to the area `middle` and insert a new fact `location(R, middle)` back into the runtime system of the robot. In this time, the robot will have a new fact `location(R, middle)` and the robotics system continues to evolve.

D. Rule

Rules have the following structure:

$$p_1, p_2, \cdots, p_m \text{ -o } q_1, q_2, \cdots, q_n$$

where $q_i(i = 1, 2, \cdots, n)$ are facts or actions, and $p_i(i = 1, 2, \cdots, m)$ are facts or boolean expressions. The interpretation of the above expression is that all q_is can be derived if all p_is in the body of the rule are satisfied. Commas in the body are interpreted as logical conjunction. The rules make the derivation of new facts and new actions from existing facts possible.

For example, a rule

```
online(A,L) -o {B | edge(A,B) |
                ready(B,A)}
```

means that if robot `A` is on the line `L`, then a fact that `A` is `ready` will be derived in any robot `B` who has an `edge` with `A`. Here, the braces are the symbol of a comprehension, which enables the repeated application of a rule and is also used in other logic programming language. Also, the programs in Fig. 3 and Fig. 4 also contain some examples about the usage of rules.

E. Time assertion

We develop a new construct in RMR called time assertion to allow developers to specify timing constraints in declarative fashion. Specifically, the time assertion for a real-time job A should be described in the following form:

$$assert(s_A, f_A, d_A, v_A)$$

In the time assertion, s_A and f_A are facts (predicates) referring to the system states when the real-time job A starts and ends, respectively; d_A is the deadline that needs to finish the job A; v_A is an action that will be derived if a violation of the time assertion is detected. Let us denote the physical time when the system state enters s_A and f_A by t_A and t'_A, respectively. The above time assertion requires $t'_A - t_A \leq d_A$, or v_A will be derived.

For example, a time assertion

```
assert(offline(A,L), online(A,L), 10,
       broadcast(A))
```

means that if robot `A` does not move to the line `L` within `10` seconds since the last time it is not on line `L`, it will `broadcast` a failure message.

IV. Compiler and Runtime System

In this section, we will first introduce the function of our compiler, and then provide a suit of runtime system to support the distributed execution of RMR programs.

A. Compiler

The compiler is responsible for translating the code written in RMR to byte-codes. The byte-codes will be easier to be interpreted by the robots than the RMR code. Inside the byte-code file, there are hexadecimal data about the number of the facts, the number of the rules, the offset to the description of the first fact, the offset to the description of the fist rule and so on. Since the computational capability of each robot is limited, it will be much better to read such hexadecimal data than the strings in the original RMR code. During the compilation of RMR programs, the ensemble-level code will be transferred to node-level code. Then the node-level code will be distributed to each robot for interpretation and execution.

```
void Robot.receiveEvent(EveType type, ArgList list)
    if type is NEW_ACTION
        processAction(n, list)
    if type is NEW_FACT
        Wait Until VM is not busy
        VM.computePredicate()

Rule VM.selectOneRule()
    highPriority = list()
    lowPriority = list()
    for i in range(0, NUM_RULE)
        if rule(i) is satisfied
            if goal facts can be derived in rule(i)
                highPriority.push(rule(i))
            else
                lowPriority.push(rule(i))
    if highPriority is not empty
        return a random element in highPriority
    if lowPriority is not empty
        return a random element in lowPriority
    return null

void VM.computePredicate()
    busy = true
    rule = selectOneRule()
    if (rule is null)
        busy = false
        return
    Process The Rule rule
    if there is any Action ac derived:
        robot.receiveEvent(NEW_ACTION, ac)
    if there is any Fact derived:
        robot.receiveEvent(NEW_FACT)
    busy = false

void VM.startVM()
    Do Initialization With The Byte-codes
    VM.omputePredicate()
```

Fig. 6. The Workflow of the Runtime System

B. Workflow of the Runtime System

To run the byte-codes generated by the RMR compiler, we developed a runtime system. The workflow of the runtime system is shown in Fig. 6.

When the robot is started, the runtime system will also be started using the function `VM.startVM`. Inside the function, the runtime system will be initialized with the byte-codes at first. The initialization includes initializing the database of the facts, inserting basic facts into the database, interpreting and storing the rules, handling the actions and the time assertions, etc. Then, the runtime system will see if the state of the system can be updated using the function `VM.computePredicate`.

In the function `VM.computePredicate`, the runtime system will see if there is any satisfied rule. If so, a piece of selected rule will be applied, which means facts may be deleted or added and actions may be derived. Then, if there is any new actions or new facts derived due to the applied rule, there will be events passing from the runtime system to the operating system of the robots. For example, if new actions are derived, there will be an event called `NEW_ACTION` received in the operating system of the robot, and the robot will react accordingly using the function `Robot.receiveEvent`.

Now, we come back to the function `VM.selectOneRule`. This function is used to select a piece of satisfied rule if any. To be specific, if there are any satisfied rules, this function will classify the rules into two categories on the basis of whether goal facts can be derived. The rules in which goal facts can be derived will have higher priority to be applied. If there are multiple rules with same priority, the function will randomly choose one to return. The usage of different priorities of different rules makes the functionality of the goal facts possible.

C. Distributed Scheduling

In the previous part IV-B, we figured how the facts, rules and actions work in the runtime system. In this part, we will figure how the time assertions work. In Section III-E, we introduce the specification of time assertion. In a time assertion, if the starting event s_A happens, the time assertion will be triggered. The robot will be aiming at its finish event f_A, which can be seen as its target. When multiple robots are aiming at their individual targets, there may be high resource competition of time and space among the robots. For example, when two robots are going to pass through a narrow corridor at the same time, collision may happen if there is no cooperation and coordination between them. To address such issue, we proposed and implemented a distributed scheduling algorithm in the runtime system. Fig. 7 shows the flow chart of our algorithm. To be specific, our distributed scheduling algorithm will schedule multiple robots with different deadlines for a same event to make more robots satisfy their deadlines.

For a single robot `A` in which an event `E` is triggered, it will broadcast a message to see if there has already been a leader for the event `E` at first. If there is a piece of response message that robot `B` is the leader for event `E`, then robot `A` will transmit a piece of message to robot `B` for purpose of joining the group of event `E`. Otherwise, robot `A` will create a new group and serves as the initiator and leader of the event `E`. After that, robot `A` will wait n seconds for new participators

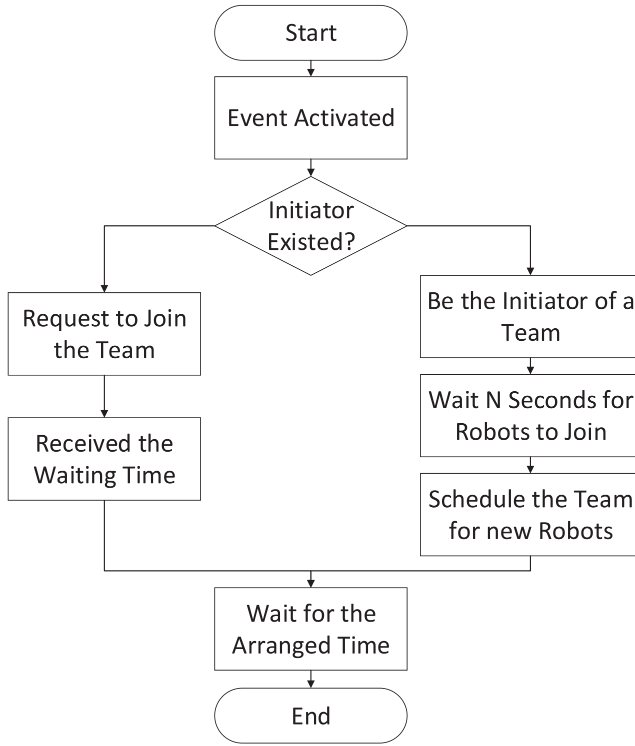


Fig. 7. Distributed Scheduling

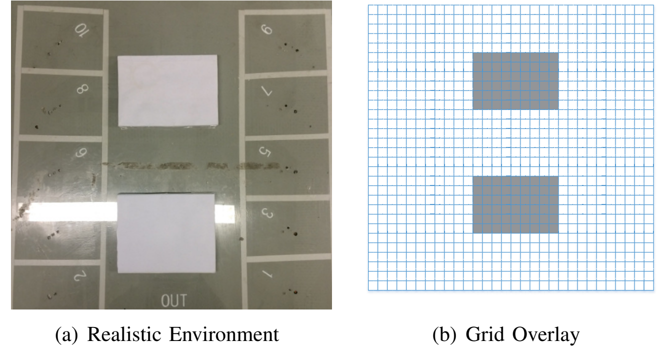
of event E. The waiting time n will be properly set according to the deadline of event E in robot A. If robot A waits for too long time, robot A will be more likely to miss the deadline. If the waiting time is too short, the size of each group may be too small, which leads to little coordination and worse performance. For a leader of an event E, it will update the scheduling list if there is a robot trying to join the group of event E. The scheduling list is maintained according to the deadline for the event.

D. Path Planning

When a robot is scheduled to start moving, it has to plan its path from its current position to its destination. In this phase, we will introduce the path planning problem for a group of robots. In our test-bed, we found that the physical distances between robots in the same group are not too large. In the path planning algorithm, we can utilize such property to save energy. Therefore, we will address the path planning problem separately for the first robot to move (leader) and other robots (followers).

1) Different Algorithms for the Leader and the Followers: With respect to the leader, we utilize grid-based A* search algorithm as the path planning algorithm, which is demonstrated in the following four steps:

- Overlay a grid graph on the working space
- Map the starting position S_1, the target position T_1 and all the obstacles O_1, O_2, $\cdots$, O_m into grid points S_1', T_1', $\{O_{11}, O_{12}, \cdots, O_{1n_1}\}$, $\{O_{21}, O_{22}, \cdots, O_{2n_2}\}$, $\cdots$, $\{O_{m1}, O_{m2}, \cdots, O_{mn_m}\}$.



(a) Realistic Environment (b) Grid Overlay

Fig. 8. Grid Overlay for the Environment

- Utilize A* search algorithm to find a shortest path P_1' from S_1' to T_1'. The result P_1' will be reduced based on the grid.
- Output the path planning result P_1 as $S_1 \mapsto S_1' \mapsto P_1' \mapsto T_1' \mapsto T_1$

For the followers, they will make full use of the leader's planning result to determine their paths. The planning result of robot k will be $P_k : S_k \mapsto S_1' \mapsto P_1' \mapsto T_1' \mapsto T_k$. In this way, we can save the energy, especially when the working space is considerably huge.

2) Grid Overlay: The size of our test-bed is approximately $1.5m \times 1.5m$, which acts as the working space. We overlay a $N \times N$ grid graph on the working space. The decision of N is of significance. If N is too large, the cost of path planning will be too huge. Otherwise if N is too small, the obstacles in the working space cannot be properly abstracted. In our work, we assign N to be 30, by which the test-bed is divided into nine hundred $5cm \times 5cm$ small grids. Fig. 8 shows the realistic environment of our test-bed and the corresponding grid graph in which N equals 30. In the abstract grid graph, the grey grids represents the obstacles while the white grids stands for the reachable places.

3) Grid-Based A Search Algorithm:* After overlaying a grid graph on the working place, we want to find a path from the starting position S to the target position T in the grid graph. First, we define the connectivity in the grid graph: two grids are connected if their differences in x-axis and y-axis are no more than 1. Moreover, we define the distance between two connected grids as the Euclidean distance. That's to say, the distance between two grids sharing an edge is 1 while the distance between two connected grids which do not share an edge is $\sqrt{2}$. Then, we use a best-first search strategy to find a least-cost path from S to T. As we traverse the grid graph, we build up a tree of partial paths. The leaves of this tree are stores in a priority queue that orders the leaf nodes according to a cost function, which combines a heuristic estimate of the cost to each T and the distance traveled from S. In detail, the cost function is $f(n) = g(n) + h(n)$. Here, $g(n)$ is the known cost of getting from S to node n, which is tracked by the algorithm, and $h(n)$ is a heuristic estimate of the cost to get from n to T. In our algorithm, $h(n)$ is set as the Euclidean

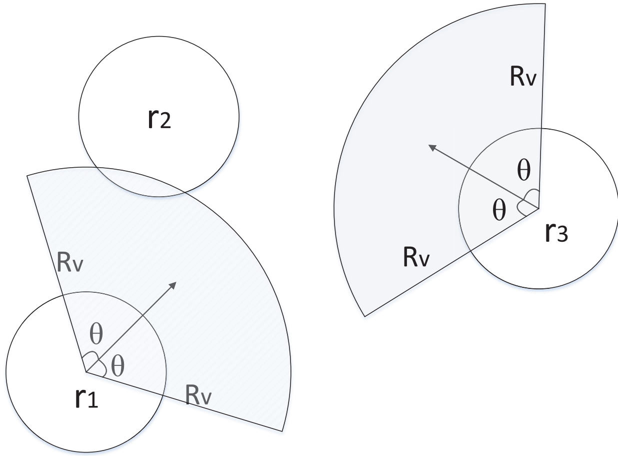


Fig. 9. the Field of View of the Robots

distance between node `n` and `T`. We can prove that $h(n)$ is admissible since $h(n)$ never overestimates the actual cost to go to `T`.

E. Collision Avoidance

When the path of each robot has been determined, the robots will start moving in the working space. At this point, collisions may happen between robots from different groups. Furthermore, in the same group, there can also be collisions on account of out-of-control. Therefore, efficient mechanism is supposed to be proposed to guarantee collision-free movements. To achieve collision-free movements, we divide the mechanism into two aspects: collision detection and collision avoidance.

With respect to collision detection, we can define the field of view by three parameters: orientation, angle of view and depth of view. Then problem of collision detection is solved with knowledge of computational geometry. To be specific, if there is a robot R_2 who enters the field of view of another robot R_1, we say collision is detected in robot R_1, which is shown in Fig. 9. Our strategy of collision avoidance is based on a principle that "stop if dangerous". If some others robots or obstacles appear in the field of view of a robot, it will stop its motion at once, and then turn to another direction to continue its movement.

V. Deployment

To demonstrate the usefulness and evaluate the performance of RMR, we deployed RMR in a simulator and a realistic test-bed, developed two example applications, and tested the execution time of the first example application.

A. Simulation

Our simulator is adapted from VisibleSim [21]. The simulator is even driven, which means that everything is modeled as an event and is scheduled for processing. The robot in the simulator maintains a list of events ordered by time to be processed. It always consumes the one at the top of the event list. Consuming an event means calling its associated callback function. For the application developers, there are only two steps to do to add a user-defined event. The first step is to create a new type of event inherited from the base event and the second step is to override the callback function of processing it.

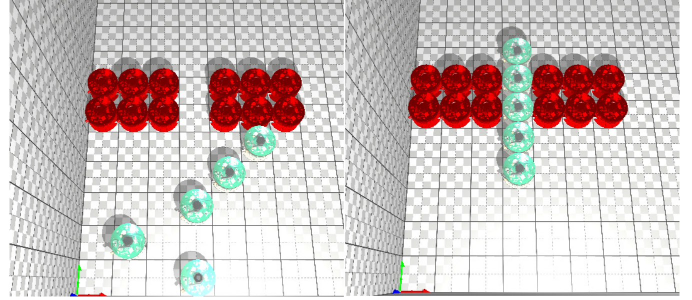


Fig. 10. Simulation: multiple robots pass through a corridor

In VisibleSim, only wire communication is supported. We have altered the communication approach from wire communication to wireless communication by implementing a message pool. The message that a robot want to send will be pushed into the message pool. After that, the message pool will deliver the message to the target robots. The robots who receives a piece of message will receive an event called `ReceiveMessage`. The robots are able to process the message by processing the event. Furthermore, features of wireless communication, such as package loss rate, bandwidth, communication range can also be simulated.

When our simulator starts to work, it first use RMR compiler to compile RMR programs into byte-codes. Then, it initializes and renders the working space according to a configuration file. Finally, its virtual machine interprets and runs the byte-codes. Fig. 10 shows the simulation that six robots pass through a narrow corridor. In the simulation, the six robots first form into a line facing the corridor and then pass through the corridor in order.

B. Real-world Experiments

Our realistic test-bed is composed of three components, which are a localization system, multiple (currently 9) intelligent robots and a programming environment.

1) Localization System: The location system consists of two ultrasonic sensor. Utilizing ultrasonic sensors which broadcast ten times per second, each intelligent robot is able to get its position on the domo platform.

2) Intelligent Robots: Our test-bed contains a set of robots, which use FreeRTOS [22] as their operating system.

The robots are home-made in our laboratory. The real picture of one robot is shown in Fig. 11. All the robots are in the same shape and with the same size, which is approximately a cylinder with 7cm radius and 18cm height. At the bottom of the robots are the wheels and motors. On the top of each robot, various sensors are equipped. In the middle part, there is a 8.4V lithium battery, which supply power for the whole robot.

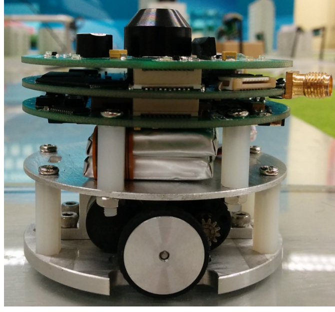


Fig. 11. Real Picture of a Robot

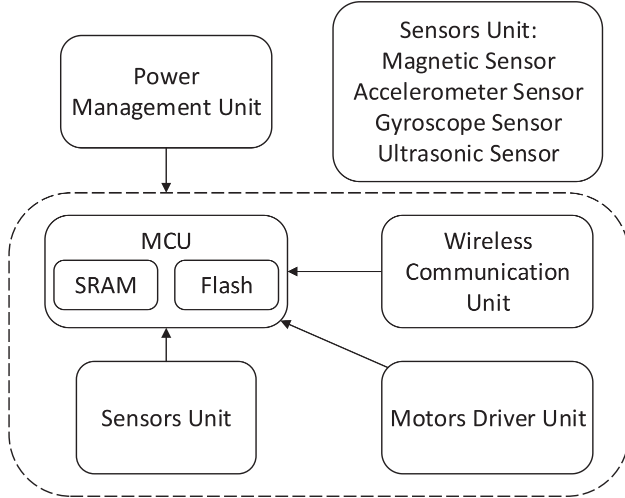
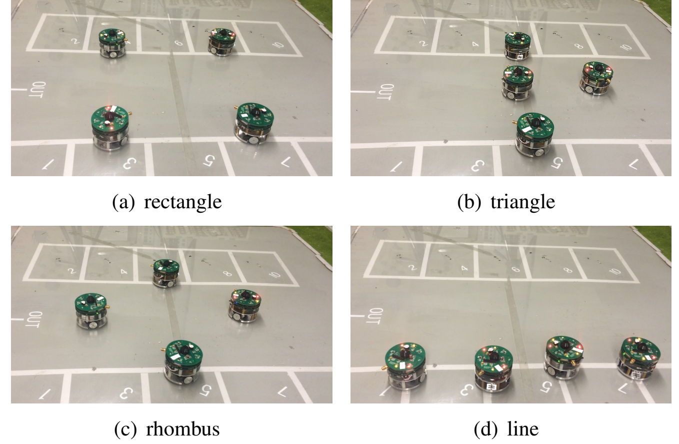


Fig. 12. Structure Diagram of a Robot

The robot's structure diagram is shown in Fig. 12. Each robot is powered by the power management unit. In each robot, the microcontroller unit (MCU) is responsible for data storage and processing. Inside the MCU, data can be stored in either 8Mbit static random access memory (SRAM) or 512Kbit flash memory. Each robot is driven by the motors driver unit, which can control the left motor and right motor separately. The wireless communication unit, which uses IEEE 802.15.4 as its protocol, enables communication between robots. Received signal strength indicator (RSSI) can be used to evaluate distances between it and other robots. The sensors unit is used to acquire information from external environment, which contains an accelerometer sensor, a gyroscope sensor, an ultrasonic sensor and a magnetic sensor.

Equipped with different kind of sensors, the robots have various functions. Also, more sensors can be equipped on the robots if necessary. For example, the accelerometer sensor can be used to calculate the moving distance. The magnetic sensor is used to determine the orientation of each robot. And the ultrasonic sensor, communicated with the beacons in the localization system, the robots can be aware of where they are on the platform.

3) Programming Environment: The programming environment is based on FreeRTOS [22], a popular real-time operating system kernel for embedded devices. In FreeRTOS, programmers can define a set of tasks with priority to be executed concurrently. For example, a task to control motors, a task to get information from embedded sensors, a task for localization and a task for purpose of user-defined application. Then, in every time unit (1/60 second in this test-bed), the tasks will be executed one by one according to the pre-defined priorities. If not all tasks can be finished during the time unit, tasks with lower priorities may be neglected.



(a) rectangle (b) triangle

(c) rhombus (d) line

Fig. 13. Example application: formation control of multiple robots

Based on FreeRTOS, we successfully deployed RMR in our robots with three steps. At first, we compile the RMR source program into node-level byte-codes using the compiler and write them into the SRAM of each robot. Then, we create a task as the runtime system for each robot to interpret and run the byte-codes. Finally, we write the callback functions in the robots' operating system triggered by the runtime system if any. To summary, RMR is deployed into the robots by creating a parallel task as the runtime system for RMR program. In addition, a tiny database is implemented to manage (insert, delete, and query) the facts and the time assertions are stored in a priority queue (sorted by time) and are triggered by hardware interrupts.

C. Example Applications

We developed two example applications to demonstrate the usefulness of RMR. Fig. 1 and Fig. 13 show demos implemented by RMR in our test-bed. In Fig. 1, multiple robots are attempting to pass through a narrow corridor. The robots coordinate with each other to form a line formation facing the corridor, and then move through the corridor in order. In Fig. 13, formation control is implemented to enable four robots to generate different shapes of formations. The robots first form into a rectangle in Fig. 13(a). Then they form into the shape of triangle in Fig. 13(b), rhombus in Fig. 13(c), and line in Fig. 13(d) respectively.

We further evaluate the efficiency of the runtime system in the first demo. In detail, we conduct experiments in our test bed to observe how execution time changes as the number of robots increase. Here, the execution time refers to the total

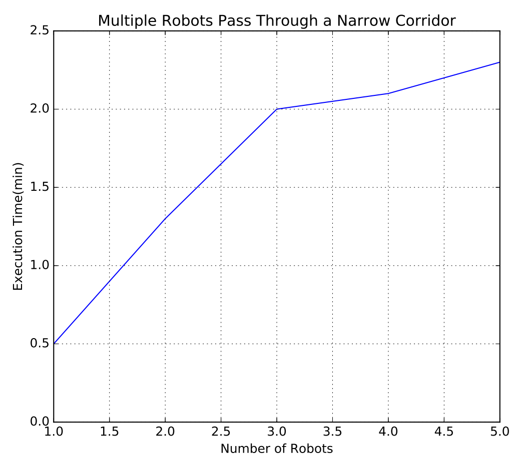


Fig. 14. Execution time of system with the increasing number of robots

time spent on starting up robots, moving forward based on the planned path and lining up at the destination. We run the first application with from one to five robots and count the execution time. As illustrated in Fig. 14, the result shows that the total execution time increase slowly as the number of robots increase, which demonstrates the efficiency of our runtime system.

VI. Conclusion

While programming MRS with real-time requirements is a difficult task at present, it can be greatly simplified by making use of appropriate programming models. In this paper, we presented RMR, a new programming model targeting at programming large-scale MRS with timing constraints. With the new elements "action" and "time assertion" proposed in RMR, RMR enables programmers to specify motions and real-time requirements in multi-robot tasks. After designing RMR, we developed a compile system and a runtime system to support the distributed execution of RMR programs. Furthermore, we have deployed RMR in a simulator and a test-bed to demonstrate the usefulness and evaluate the performance of RMR. By means of experiments in a real deployment, we claimed that RMR is easy-to-use programming model for multi-robot applications with timing constraints. In the future, we can improve the performance of the runtime system by proposing more efficient mechanisms of distributed scheduling and others. Also, we can improve the programming model to enhance the real-time support for MRS.

VII. Acknowledgements

The research is partially supported by the ANR/RGC Joint Research Scheme with No A-PolyU505/12, and NSFC with number 61332004.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ACM SIGOPS operating systems review*, vol. 34, no. 5. ACM, 2000, pp. 93–104.

[2] N. Minar, R. Burkhart, C. Langton, M. Askenazi *et al.*, "The swarm simulation system: A toolkit for building multi-agent simulations." Santa Fe Institute Santa Fe, 1996.

[3] W. J. Butera, "Programming a paintable computer," Ph.D. dissertation, Citeseer, 2002.

[4] N. Margolus, "Cam-8: a computer architecture based on cellular automata," *Pattern Formation and Lattice-Gas Automata*, pp. 167–187, 1996.

[5] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: language, execution and optimization," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 97–108.

[6] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein, "Entirely declarative sensor network systems," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 1203–1206.

[7] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM, 2004, pp. 99–110.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[9] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 2007, pp. 489–498.

[10] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 200–210.

[11] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[12] J. Beal and J. Bachrach, "Infrastructure for engineered emergence on sensor/actuator networks," *Intelligent Systems, IEEE*, vol. 21, no. 2, pp. 10–19, 2006.

[13] J. Bachrach, J. McLurkin, and A. 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*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 1175–1178.

[14] J. Bachrach and J. Beal, "Programming a sensor network as an amorphous medium," 2006.

[15] M. De Rosa, S. Goldstein, P. Lee, P. Pillai, and J. Campbell, "Programming modular robots with locally distributed predicates," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. IEEE, 2008, pp. 3156–3162.

[16] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, 2007, pp. 2794–2800.

[17] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A language for large ensembles of independently executing nodes," in *Logic Programming*. Springer, 2009, pp. 265–280.

[18] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Distributed Computing in Sensor Systems*. Springer, 2005, pp. 126–140.

[19] U. Nilsson and J. Małuszyński, *Logic, programming and Prolog*. Wiley Chichester, 1990.

[20] F. Cruz12, R. Rocha, and S. C. Goldstein, "A parallel virtual machine for executing forward-chaining linear logic programs," in *Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014*, p. 125.

[21] D. Dhoutaut, B. Piranda, and J. Bourgeois, "Efficient simulation of distributed sensing and control environments," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 452–459.

[22] R. Barry, *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.