An Ensemble-Level Programming Model with Real-Time Support for Multi-Robot Systems

Shan Jiang, Junbin Liang, Jiannong Cao, and Rui Liu
Department of Computing, The Hong Kong Polytechnic University, Hong Kong
Email: {cssjiang, csljunbin, csjcao, csrliu}@comp.polyu.edu.hk

***Abstract*—In this paper, we propose a novel programming model RMR (Real-time programming for Multiple Robots) targeting at programming multi-robot system (MRS) with timing constraints. RMR is a logic programming model with real-time support. In the light of the logic programming paradigm, RMR allows developers to write simple code to accomplish complex tasks and deploy the program in MRS in an efficient way. Moreover, RMR supports timing constraints on the behaviors of an ensemble of robots, which is not implemented by existing works. We deploy RMR in a simulator and a test-bed to test its performance in both cyber and physical world, and then demonstrate RMR based on several applications. This paper presents our current prototype.**

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 MRS, which has better reliability, flexibility, scalability and versatility than a single-robot system. However, the management of the robots is a challenging issue. Therefore, a scalable programming model is required to program MRS containing thousands or even millions of robots. Moreover, the programming model is supposed to support timing constraints on the behaviors of the robots.

Traditional programming models for robots, such as NesC [1] and ROS [2], can express almost all the robotic collaborative behaviors, but require developers to address every detail of the behaviors. In their programs, each robot should be given a specific sequence of actions while performing tasks. The actions include moving, sensing, communicating, etc. These approaches have high programming complexity and low scalability, so they are difficult to be used in the large-scale MRS. Some other related works, such as P2 [3] and Meld [4], consider each MRS as a whole and allow developers to specify high-level description of what the set of the robots should achieve. These languages implemented the low-level details so that the developers do not need to pay more attention to implementation. These languages have low programming complexity and high scalability, but they are not suitable for real-time tasks due to lack of real-time support.

In this work, we propose a novel 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 order of actions. To support distributed execution of RMR programs, a compiler and a runtime system are developed in 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.

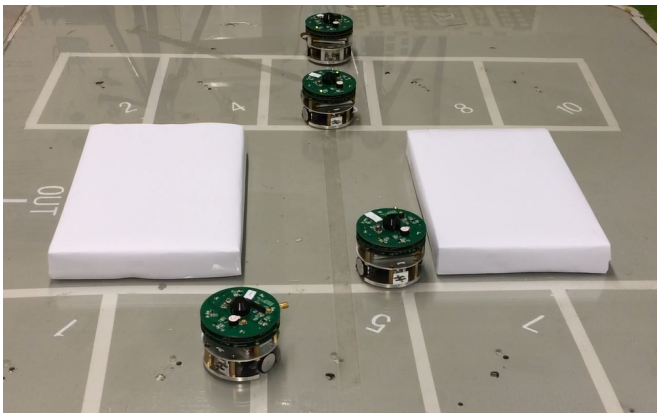


Fig. 1. Multiple robots are passing through a narrow corridor

To evaluate the performance of RMR, we implemented and 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.

II. Principles and Specifications of RMR

In this section, we first introduce the principles of designing RMR. Then, we present the key features of RMR, including syntax and semantics.

A. Design Principles of RMR

RMR is a logic programming model, which encompasses a set of facts and rules as 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.

1) Ensemble-level abstraction: The entire MRS can be viewed as a single and monolithic unit while developers write RMR program. RMR enables a developer to think about what

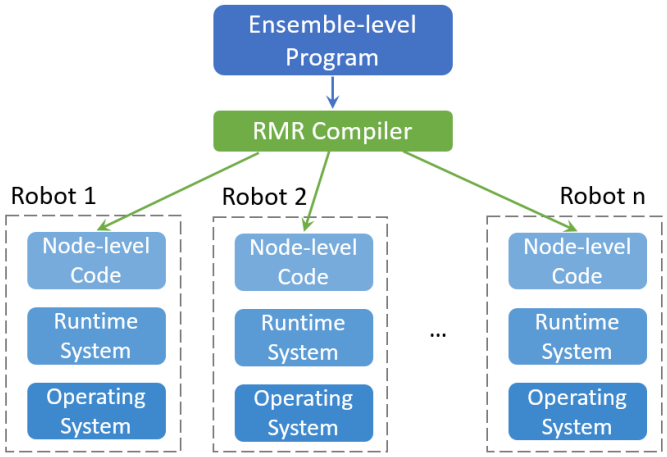


Fig. 2. Overview of RMR Compilation

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 [4], and is referred to as macro-programming in the area of wireless sensor networks [5].

2) Backward chaining logic: RMR adopts an inference method named backward chaining for the execution of its programs. Backward chaining starts with a list of goals and works backwards from the consequent to the premises to build a reasoning tree. The robots will use the reasoning tree to derive new facts. Backward chaining has been proved to be effective in the famous logic language Prolog [6].

3) Declarative specification of timing constraints: In MRS, any job should contain a series of reasoning steps. 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. Therefore, 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.

B. RMR Specification

In the following, we will describe the detailed specification of RMR.

1) Variable, Constant and Boolean Expression: RMR follows the conventions of logic programming model when defining variables and constants. Moreover, RMR supports boolean expressions that can be calculated into boolean values (true or false).

2) 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. A fact generally has a predicate symbol (or name) and can take some arguments (variables) as input. In RMR, facts are divided into three categories: persistent facts, temporary facts and goal facts.

- Persistent facts refer 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 a robot `A` is moving along the line `L`.
- Goal facts represent the goals of the program that must be satisfied at some time. They must be declared with a question mark '?', which means "why not" or "to be achieved".

3) Rule: Rules have the following structure:

$$p_1, p_2, \cdots, p_k \text{ -o } q$$

where q is a fact, and $p_i (i = 1, 2, \cdots, k)$ are facts or boolean expressions. The interpretation of the above expression is that q 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 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`.

4) 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.

With the principles mentioned in II-A, RMR creates a new way of programming MRS applications with timing constraints. After a program is written, it will be compiled into byte-codes that can be executed on the robots. An overview of the compilation is shown in Fig. 2. The highest level is the RMR program, which is written by a developer and obeys a

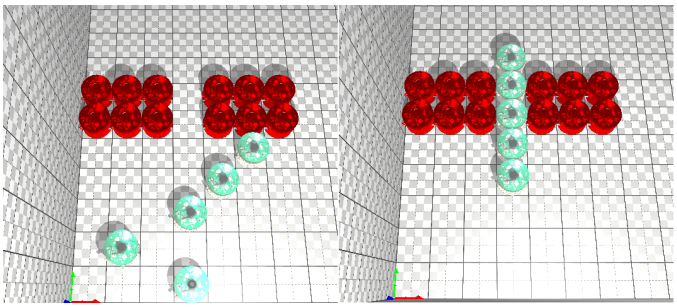


Fig. 3. Simulation: multiple robots pass through a corridor

centralized and ensemble-level abstraction. A RMR compiler in the middle level can convert the program into distributed byte-codes ran on individual physical robots.

III. Prototype Deployment

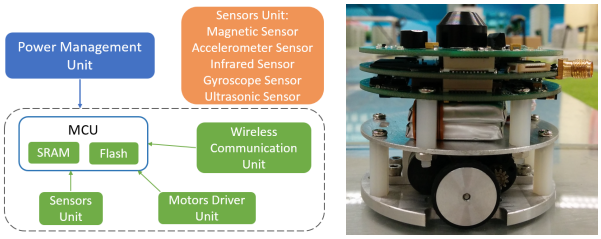
To evaluate performance of RMR, we have deployed RMR in a simulator and a realistic test-bed, and have developed several example applications. Moreover, we have implemented the same applications with a traditional language, embedded C. The comparison result indicates that RMR programs are substantially more concise (more than 10x shorter) than programs written in embedded C, while running nearly as efficiently.

Our simulator is adapted from the VisibleSim simulator [7]. When our simulator starts to work, it first use RMR compiler to compile RMR programs into byte-codes. Then, it initializes and renders the scenario according to a configuration file. Finally, its virtual machine interprets and runs the byte-codes. Fig. 3 shows the simulation that 6 robots pass through a narrow corridor.

Our test-bed contains a set of robots, which use FreeRTOS [8] as their real-time operating system. The robot's structure diagram is shown in Fig. 4(a), and the real picture of the robot is shown in Fig. 4(b). 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 infrared sensor, an ultrasonic sensor and a magnetic sensor. Moreover, each robot can move smoothly by using feedback controller.

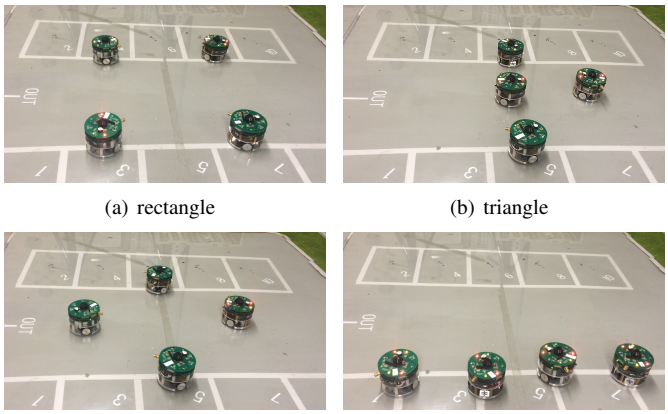
RMR is deployed into the robots by creating a parallel task to process the rules in RMR program. A tiny database is implemented to manage (include inserting, deleting, and querying) facts. Time assertions are stored in a priority queue (sorted by time) and invoked by hardware interrupts.

Fig. 1 and Fig. 5 show demos implemented by RMR in the test-bed. In Fig. 1, multiple robots are attempting to pass



(a) Structure Diagram (b) Real Picture

Fig. 4. Structure Diagram and Real Picture of a Robot



(a) rectangle (b) triangle

(c) rhombus (d) line

Fig. 5. Demo: formation control of multiple robots

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. 5, formation control is implemented to enable 4 robots to generate different shapes of formations (rectangle in (a), triangle in (b), rhombus in (c), and line in (d)).

References

[1] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Acm Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 1–11.

[2] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: A programming model for event-driven embedded systems," in *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, 2003, pp. 698–704.

[3] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 75–90.

[4] 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.

[5] 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.

[6] W. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[7] 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.

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