Concurrent and Real-Time Task Management for Self-Reconfigurable Robots

Harris Chi Ho Chiu and Wei-Min Shen
Information Science Institute, University of Southern California
4676 Admiralty Way, Marina Del Rey, CA 90292
{chiho, shen} @isi.edu

Abstract

We present a concurrent and real-time task management method for distributed control of modules in a self-reconfigurable robot. This method is essential for modules to simultaneously control multiple behaviors in real-time and supporting a concurrent programming style that will greatly ease the development of control software for large-scale reconfigurable robots. Although real-time operating systems and concurrent programming have been around for many years, it is only recently that the technology is miniaturized enough for embedded systems and self-reconfigurable robots. We have successfully implemented the method on our new SuperBot modules and have demonstrated its utility through the programming of locomotion gaits with the real modules.

Keywords: reconfigurable robots, real-time task management, concurrent programming

1 Introduction

A self-reconfigurable robot is a new type of modular robot that can change its shape, size and function to accomplish difficult tasks in dynamic and unforeseen environments. Such robots are made of many autonomous, intelligent, and self-reconfigurable modules that provide the basic units for the transformation. These modules are extreme examples of "design for reuse." For space applications, for example, one such robot can reconfigure its modules into an extended arm for extravehicular inspection and maintenance, or unloading payloads after landing. On the surface, the robot may morph into a set of mini-rovers to explore the environment, or a set of climbers to go down and up a crater, or become a single, large mobile platform to perform applications such as drilling, building, or sample collection. In many cases, such a robot may self-detect unexpected failures and repair malfunctions by reconfiguration of its modules.

Due to the fact that modules may reconfigure dynamically and have heterogeneous controllers, communication devices, sensors, actuators and connectors, the control of modules is a difficult problem. It must be flexible to satisfy both spatial and temporal constraints with limited computational resources. For the spatial constraints, the control must be *totally distributed* among modules. This is because centralized control is too vulnerable to module failures and partially distributed control must rely on certain non-local knowledge to function. An example for a partially distributed controller is a controller that requires unique global identifiers for modules. This would unnecessarily complicate the reconfiguration process because a trivial local modification in the system, such as a replacement of a single module, may cause many non-local changes that are expensive to implement. In this case, all non-local modules that use the replaced identifier must be informed to modify their programs.

For the temporal constraints, the control of modules must be *concurrent and real-time*. This is inevitable for a self-reconfigurable robot because a single module is often required to simultaneously perform and manage multiple behaviors that are all time critical. Such behaviors include sensor reading, motor controlling, communication, synchronization, and collaboration with other modules. The sequential programming style is inadequate for this purpose for several reasons. First, it is generally difficult to interleave multiple time-critical tasks correctly into a sequential program. For example, consider two behaviors that must be performed periodically at every x and y units of time, respectively. To sequence them properly, a programmer must calculate every time point that is critical for interleaving them in a proper order. This can be very hard if x and y do not have common denominators. The resultant program would be very long and extremely difficult to debug. Second, when there are many concurrent behaviors, say $\{b_1, b_2, \ldots, b_n\}$, to be managed, a programmer must consider all possible subset combination of these behaviors. This is very hard to manage when n is large. Third, interleaving time-critical behaviors is not always possible. This is especially true when the starting times or the duration of certain behaviors are not known in advance and must be dynamically triggered by external events that are beyond the control of the module. Both spatial and temporal constraints impose difficulties on programming global gait for all the modules. The spatial constraints require totally distributed control among modules, which limits a module to only communicate to its neighboring modules. The temporal constraints add

complexities in synchronizing modules to perform global gait. The programming of global gait has to consider both the protocol for synchronization and inter-module communication in addition to the timing control of various devices in individual module. Due to these difficulties, we believe it is absolutely necessary to have concurrent and real-time task management for programming self-reconfigurable robots with many modules.

Up to date, there is yet any controller for self-reconfigurable robot that is totally distributed (i.e., ID free), and concurrent in real-time. A concurrent programming would bring many advantages for programming self-reconfigurable modules that are not possible in sequential programming. For example, it will greatly ease the addition or subtraction of time critical behaviors in a module's management portfolio. It will increase the robustness of the control software because behaviors can be organized into intuitive and modularized software units. It will enhance the adaptability of the control program to different hardware because behaviors can encapsulate underlying hardware through a well-defined software interface. Furthermore, it will provide a natural interface for human operators to control the robot without being distracted by the detailed timing issues in the modules.

Inspired by the above motivation, this paper introduces a concurrent and real-time task management framework for behaviors/device control that is free from global unique identifiers. This method has been implemented and tested on a set of new self-reconfigurable modules called *SuperBot* that are developed for space applications. The utilities of this real-time programming framework are demonstrated through SuperBot locomotion where modules concurrently monitor and execute multiple behaviors simultaneously in real-time.

The rest of the paper is organized as follows. Section 2 discusses the related work for concurrent and real-time task management. Section 3 describes the complexity of task management for self-reconfigurable modules. Section 4 through 6 describes the detailed notion of tasks and their real-time management and communication, as well as an implementation that divides tasks into the system level and the application level. Section 7 illustrates the utility of this approach by applying it to a new implementation of distributed control without assuming any global identifiers for the modules. Finally, Section 8 concludes the paper with future works.

2 Related Work

Although real-time operating systems [1] and concurrent programming [2] have been around for many years, it is only recently that technology is miniaturized enough for embedded systems and self-reconfigurable robots. Good examples of these small real-time operating systems include: QNX [3], LynxOS/BlueCat [4], TinyOS [5], XMK [6], NutOS [7], FreeRTOS, [8], and AvrX [9].

However, the current research in self-reconfigurable robot is mostly focused on hardware design and algorithm development. How to program the modules effectively and efficiently is a relatively new topic. The current state of the art is to have a sequential, special-purpose program for every different thing a modular robot needs to do, and this simple approach has served its purpose well so far because the complexity of behaviors of these modules are mostly single threaded.

CANbus was initially developed by Robert Bosch for in-vehicle data transfer and was defined in 1984. Silicon became available in 1987 and CAN was first used in cars in 1992.The draft international standard was introduced in 1991 and this became a full standard (ISO 11898) in 1994. Several self-reconfigurable robots have used this technology, but the drawback is that it has only small addressing space so the number of modules in the system must be limited.

Zhang et al. [10] have proposed a software architecture and implemented it on a real-time operating system for modular reconfigurable robots. However, inter-module communication requires a bus and therefore global identifiers are required for addressing. If a module is added or replaced, the control program must be rewritten or the new module must be reprogrammed to retain the same global address. Shen et al. [11] provides a hormone-inspired, distributed control mechanism without global addressing by having point-to-point communication between neighboring modules. Program in every module is identical and can be replicated. Therefore, modules can replace one another without any non-local changes. This ensures robustness and scalability. However, the implementation of this method has been single-threaded and it is inadequate for more complex behaviors, where devices must be controlled simultaneously in real time.

3 Tasks for Reconfigurable Modules

To illustrate the complexity for task management of self-reconfigurable modules, we consider the SuperBot [12,13] as an example. Shown in Figure 1, SuperBot is a new self-reconfigurable robot developed for NASA space applications. Figure 1 shows two individual SuperBot modules, and six connected modules in a human-like configuration. Each SuperBot module has six dock faces that can be connected to other modules. A single SuperBot module is designed to be totally autonomous, and it has power suppliers, controllers, motors, sensors, communication devices, and docking guidance mechanisms. A single SuperBot module can move and turn autonomously to any direction it desires to. It can travel on batteries up to 500 meters on carpet in

an office environment. Multiple-module locomotion has also been demonstrated for a variety of configurations and gaits, such as a serpentine gait, a caterpillar gait, a butterfly stroke, rolling tracks, and gaits for legged configurations. Movies of these behaviors can be seen at the website http://www.isi.edu/robots/superbot/.

Inside a SuperBot module, there are batteries, two Atmega128 micro-controllers, nine motors, seven communication devices, six docking guidance mechanisms, and many types of sensors. The internal hardware architecture is shown in Figure 2. As we can see, there are many devices that must be managed by a single SuperBot module.

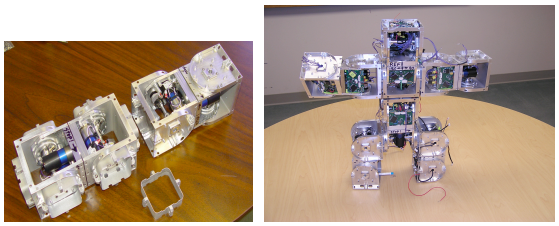

Figure 1: SuperBot modules and configurations.

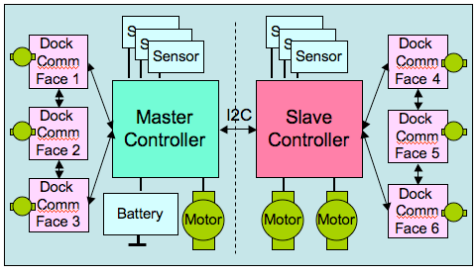

Figure 2: Necessary tasks for a SuperBot module.

To program SuperBot module properly, software tasks must be created and performed. Each device has different timing requirements and therefore each device requires one task to manage. There are seven communication tasks, nine actuator tasks, nineteen sensor tasks and one or more behavior tasks. Since many of these tasks are time critical, it is a nontrivial problem to manage them properly. Although it is possible to combine similar tasks into a single task and organize the tasks into a hierarchy, the complexity of managing them in a timely fashion is indeed a challenge. Furthermore, the onboard computational resource is highly limited due to the small energy budget and that increases the challenge even further. This analysis is not only true for SuperBot, but for any deployable self-reconfigurable modules in general. Thus, a concurrent and real-time management is essential.

4 Concurrent Tasks and Real-Time Management

In this paper, a concurrent task is a fundamental software unit for real-time management. Similar to a thread in an operating system, a concurrent task has its own stack and memory space for static variables and it appears to have a complete CPU control independent of other tasks. In our implementation, we have adopted the notion of task from AvrX.

AvrX [9] is an extremely small but powerful real-time kernel for Atmel AVR series of micro controllers, developed by L. Barello. AvrX contains approximately 40 API functions in six categories: Tasking, Semaphores, Timer Management, Message Queues, Single Step Debugging support, and Byte FIFO support with synchronization. The Kernel is written in assembly and is extremely small footprint – about 1500 bytes in code size and about 30 bytes SRAM per task. AvrX is a real-time operating system and maintains state information for the each task. So a programmer can write each task in as sequential code as if the task has the CPU for its own. It is easier to develop independent modules that can be wired together later on.

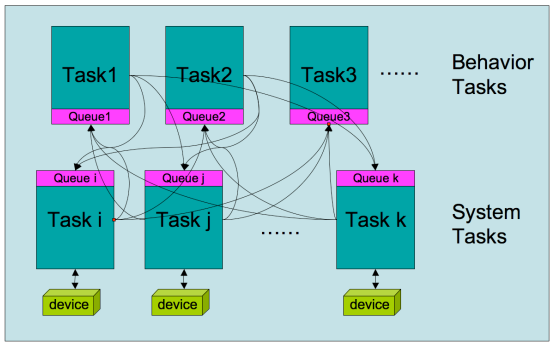

Figure 3: Tasks and their communication

To manage the computational resources well in real-time, we use AvrX to allocate CPU resources among tasks. At the same time, AvrX also provides us a very useful tool for software modularity. It provides a common interface for communicating with different hardware devices and a common mechanism for handling timing constraints. We use AvrX's mechanism for tasking, semaphores, timer management and message queues. The speed is also sufficient, it takes less than 20ns for context/task switching when running on Atmega128 at 16 MHz with 128kbytes flash memory and 4kbytes SRAM. All tasks are written in C language. Associated with each task is a message queue. Tasks can communicate with each other by placing messages into each other's queue. Tasks can be set up to run periodically or to be run "on demand." A task receives a message from other tasks by checking its own queue as shown in Figure 3.

To perform locomotion and reconfiguration successfully, both control software and hardware control have to meet real-time constraints. For example, a motor needs to be periodically monitored within certain period to hold its position while control software has to send out messages regularly to keep control information updated and synchronized.

Task-oriented real-time programming also helps to maximize resource utilization. The scheduler runs

tasks whenever the CPU is not busy. Timing can also be set independently in software modules. Execution conflicts among device software can be resolved systematically by using system utilities, like semaphores, timers and mutual exclusion (mutex). Therefore, real-time scheduler makes the change of timing constraints in tasks easier for the programmer, and devices can be controlled concurrently.

In our implementation, we divide the tasks in a module into two classes: the system tasks, and the behavior tasks. A high-level distinction between these two classes is that a system task encapsulates and manages a hardware device and can be shared by many behaviors tasks. A behavior task is to use the system tasks to control the module for locomotion, navigation, reconfiguration, and other functions. Behavior tasks can reuse the system tasks without any reprogramming.

5 System Tasks

System tasks are low-level software modules that hide the details of low-level control of the hardware from the high-level behavior software programmer. Figure 4 shows a simplified diagram of the tasks running on a SuperBot module, where two AvrX kernels, Master and Slave, are running on the two micro-controllers in each module.

The Master and Slave controllers use I2C serial communication to send messages to each other. The communication with other modules via the docks is handled by the IR tasks. Every dock has its own IR tasks, but for simplicity, the IR related tasks for only one dock on the Master and Slave are shown. Although the large number of tasks seems to add significant complexity, it actually minimizes the time that the CPU is blocked waiting on a task or resource.

The handling of incoming data through IR and I2C is interrupt-driven. The sending and receiving of data is therefore wrapped into single task: in this case the task switching cost is expected to be higher than the cost of not being able to send and receive simultaneously. The motor task implements a PID controller, which is being executed every one millisecond.

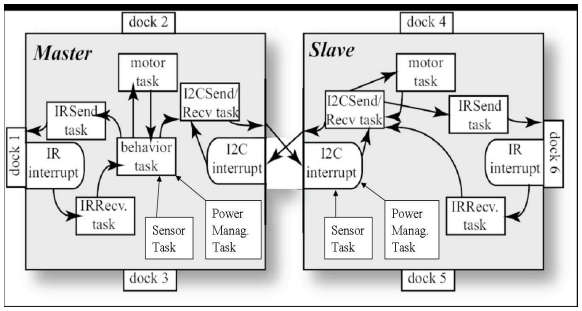

Figure 4: Tasks running on a SuperBot module.

The IR communication is much slower and tends to be noisier. For IR communication we have implemented the stop-and-wait ARQ (Automatic Repeat reQuest) protocol [14]. A task on a neighboring module cannot directly send a message to a low-level task on a module, but only to a behavior task. So a message received on any of the docks is routed to a behavior task. If the destination task specified in the header of the message does not run on a receiving module, then the message is simply ignored.

For each communication link, there are two tasks for sending and receiving messages. The sending task checks its own message queue and only sends when there is message in the queue. For interrupt driven communication, received data is collected in the interrupt. A message using a dynamic memory buffer is formed and put into the message queue of the receiving task. The receiving task picks up messages from the queue and delivers it to destination task.

The sensor tasks are executed by the behavior tasks. Once activated, a sensor task reports the status of the on-board sensors to the behavior task directly or through I2C channel. The power management task is responsible for checking each connector current, the status of the battery, charging the battery, and set/resetting the power switches in each docking face.

The tasks for controlling the docking motors are not shown in this figure, and they are being developed along with the mechanics and electronics development for the connectors.

6 Behavior Tasks

Behavior tasks are responsible to run high-level control software using the functions provided by systems tasks. They have no direct access to the low-level devices but they call the interfaces provided by the system tasks. This provides an abstraction for programmers so that when they are writing the behavior tasks, they do not need to worry about the details of timing the hardware devices. Thus, behavior tasks are less time critical in general, and they are given a lower execution priority than the system tasks.

Behavior tasks are typically designed for high-level jobs such as power management, locomotion, manipulation, navigation, and self-reconfiguration. Since modules may have heterogeneous hardware, it is important for behavior tasks to be abstracted from the hardware details. For this purpose, behaviors tasks are generally written through a set of API functions that are implemented by the underlying system tasks. This way, behaviors tasks can be portable to different modules even though the implementation of these API may be different from module to module.

7 Distributed Control without Global-IDs

Due to the nature of self-reconfiguration, it is highly desirable to have distributed controllers that are free

from any non-local knowledge (e.g., free from global unique identifiers for modules). This is essential for quick adaptation to dynamic and uncertain changes in the topology of reconfigurable robot. The real-time task management described in the paper can actually support this id-free approach for controlling a self-reconfigurable robot. The key idea is to provide messaging without the need of global addressing thus enable the development of high-level control to be independent from any specific arrangement of the modules.

To demonstrate the utility of the proposed concurrent and real-time task management framework, we describe one detailed example of a controller for the locomotion of a large centipede configuration that contains 100 modules without any global identifiers for the modules. This example will be done using the hormone-inspired distributed controller and the Adaptive Communication (AC) protocol described in [11]. To implement this control program using the task management framework, we introduce three behavior tasks: AC_SEND, AC_RECV and HORMONE, shown in Figure 5. All modules in the robot will run the same three behaviors tasks.

The AC_SEND Task:
Repeatedly probe neighbors to update the local topology.

The AC_RECV Task:
Receive probes and update the local topology of the module.

The HORMONE Task:
For each received hormone message:
Select and execute the proper local actions based on
(a) the local topology,
(b) the local sensor inputs,
(c) the local state/timer information,
(d) the received hormone message;
Propagate the hormone message to other connectors.

Figure 5: Pseudo-code for the hormone-inspired controller.

The AC_SEND task is to continuously update the connection information with neighboring modules by periodically sending probe messages to all six dock faces through the corresponding system-level IR communication tasks. If a neighboring module is present, then the AC_RECV task will receive a reply from a system-level communication task and thereby determine the local topology of the current module. The AC_RECV is continuously updating the local topology of the module and sends the result to the HORMONE task. Recall that IR communication is carried out by a system task that checks its queue and uses the IR device to send to its neighbor module. On the neighboring module, the receiving task of corresponding communication interface will pick up the message and deliver it to the AC_RECV task. The AC_RECV task gets update from its queue and sends corresponding topological information to the HORMONE task. The HORMONE task also receives messages from sensor tasks and timer tasks at the system level (for items (b) and (c) in Figure 5).

The HORMONE task checks its message queue for hormone messages from other modules and topology information from the AC_RECV task on the same module. Upon receiving a hormone message, the HORMONE task can determine what actions to perform and what hormone to propagate to neighboring modules using a set of "receptor" rules. Actions can be executed by putting messages to the queues of corresponding actuator tasks and the propagation of hormone messages can be achieved by putting message into the message queues of the sending task of the desired communication interface. Since the AC_RECV task constantly updates the local topology based on the results of probing neighbors, and a module acts differently according to the four factors (items a-d in Figure 5), the robot can run without reprogramming even when modules are reshuffled in the body.

Table 1: The Action Rules for A Centipede

Module Type	Local Timer	Received Hormone Data	Perform Action	Send Hormone
Head	0		Straight	[CP, A, {l,r,b}]
Head	0.5*Period		Straight	[CP, B, {l,r,b}]
Spine		A	Straight	[CP, B, {b}]
Spine		B	Straight	[CP, B, {b}]
Branch		A	Straight	[CP, B, {l,r,b}]
Branch		B	Straight	[CP, A, {l,r,b}]
Left Leg		A	Swing	
Right Leg		A	Holding	
Left Leg		B	Holding	
Right Leg		B	Swing	

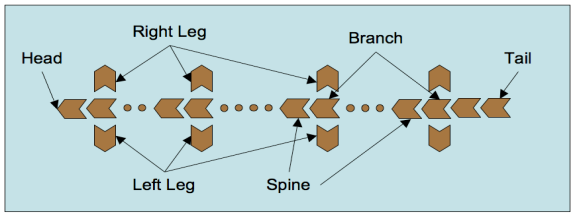

Figure 6: A centipede configuration of 100 modules.

Figure 6 shows an example of a centipede configuration with 100 modules. We assume each leg is one module long and the number of legs and the distribution of legs on the body are arbitrary. Each leg can either swing forward or hold in place to sustain the body. Notice that in this configuration, there are only six types of modules with respect to their local topology: head, tail, right leg, left leg, link torso, and branch torso. A *head* is a module that has only one connection and its back dock is connected to the front dock of another module. A *tail* is a module that has only one connection and its front dock is connected to the back of another module. A *right leg* (*left leg*) is a module has only one connection and its back dock is connected to the right (left) dock of another module. A *spine* is a module that has two connections, its front

is connected to the back of another module and its back is connected to the front of a different module. A *branch* is a module that has four connections, its front, left, right docks are connected to the back of other modules respectively, and its back dock is connected to the front of another module. Each module can dynamically discover its own type, among the six local topologies described, by using the AC_SEND and AC_SEND tasks discussed in Figure 5.

To control the movement of this centipede, the HORMONE task will use the rules listed in Table 1. The hormone message is named as CP. We use set notation such as {l,b,r} as a shorthand for sending a message to the left, back, and right dock, respectively. The action *Straight* means to hold all motors at 0 degree. The action *Swing* means to lift a leg module, swing the module forward, and then put the module down on the ground. The action *Holding* means to hold a leg module on the ground while rotating the hip to compensate the swing actions of other legs.

The first two rules indicate that the head module is to generate two new CP hormones with alternative data (A and B) for every cycle of period. This hormone propagates through the spine modules as it is, but alternates its data field (A↔B) at every shoulder module, and reaches the leg modules, which will determine their actions based on their types (left or right). The result is that every other left and right legs are swinging while the rest legs are holding, and if a left leg is swinging, then its corresponding right leg is holding, and vice versa.

This control mechanism is robust to changes in configurations. For example, one can dynamically add or delete legs from this robot, and the control will be intact. The speed of this gait can be controlled by the duration of period, which determines the frequency of hormone generation from the head module. Notice also that this set of rules works for configurations that have arbitrary pairs of legs and is scalable to large configurations.

8 Conclusion

This paper presents a concurrent, real-time task management framework for programming modules in self-reconfigurable robots. Compared to sequential programming, this method eliminates the difficulties of interleaving many time critical tasks that are necessary for the modules, modularizes the software in modules, and eases the integration and modification of software and hardware components. Furthermore, it supports an ID-free distributed controller. The framework has been implemented on the SuperBot self-reconfigurable modules, and demonstrated its utilities in several control programs for locomotion tasks. Our future work includes extension of the framework for reconfiguration, navigation, and other functions of self-reconfigurable robots.

9 Acknowledgements

This research is supported in part by NASA Cooperative Agreement NNA05CS38A, and in part by US Army Research Office under the grants W911NF-04-1-0317 and W911NF-05-1-0134. We are also grateful for other members in our Polymorphic Robotics Laboratory, especially Dr. Mark Moll and Dr. Behnam Salemi, for their useful comments on the drafts of the paper.

10 References

[1] Rob Williams, *Real-Time Systems Development*, Butterworth-Heinemann Publisher, 320 pages, December 2005.

[2] David Bustard, *Concurrent program structures*, Prentice Hall international series in computer science, December 1987.

[3] QNX http://www.qnx.com.

[4] LynxOS/BlueCat http://www.lynuxworks.com.

[5] TinyOS, http://www.tinyos.net/.

[6] XMK, extremely minimal kernel. http://www.shift-right.com/xmk/

[7] NutOS, http://www.ethernut.de/en/software.html

[8] FreeRTOS, http://www.freertos.org/.

[9] L. Barello, *AvrX Real Time Kernel*. http://www.barello.net/avrx.

[10] Zhang, Y., K. Roufas and M. Yim. Software architecture for modular self-reconfigurable robots. Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, October 2001.

[11] Shen, W.-M., B. Salemi and P. Will. Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots. *IEEE Trans. on Robotics and Automation*, 18(5):700–712, October 2002.

[12] Shen, W.-M., M. Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh. Multimode Locomotion for Reconfigurable Robots. *Autonomous Robots*, 20(2):165–177, 2006.

[13] Salemi, B., M. Moll and W.-M. Shen. SUPERBOT: A deployable, multifunctional and modular self-reconfigurable robotic system. In *Proc. 2006 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Beijing, China, October 2006.

[14] ARQ: Stop and Wait ARQ communication protocol, http://www.erg.abdn.ac.uk/users/gorry/eg3567/arq-pages/saw.html