Implementing CRSM Rendezvous Communication for a Distributed Robotic System

Dissertation

Submitted in partial fulfillment of the requirements for the degree of Master of Technology

by

B. Election Reddy
Roll No: 07305054

under the guidance of Prof. Kavi Arya and Prof. Ramesh S

June, 2009

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Abstract

Our work explores the design of communication primitives for a distributed robotic system. This system constitutes a Globally Asynchronous Locally Synchronous (GALS) system. In order to do so we studied the principles of Communicating Reactive State Machines (CRSM) and Rendezvous Communication which we have adapted in our own communication API. We have demonstrated the use of these primitives in our design of Treasure Hunt case study. These experiments were conducted using the versatile Fire Bird robot developed by Embedded and Real Time Systems Lab, IIT-Bombay.
# Contents

1 Introduction  
2 Communicating Reactive State Machines  
  2.1 CRSM Constructs  
  2.1.1 CRSM Node  
  2.1.2 Hierarchical Composition  
  2.1.3 Synchronous Parallel Composition  
  2.1.4 Signal Hiding  
  2.1.5 Asynchronous Composition  
  2.2 Status of CRSM  
  2.2.1 CRSM Editor  
  2.2.2 CRSM Simulator  
  2.2.3 Verification  
  2.2.4 CRSM2HandelC  
3 Fire Bird  
  3.1 Hardware Specification  
  3.2 Programming  
  3.2.1 Reading and Writing the I/O ports  
  3.2.2 Motion control of Robot  
  3.3 Communication  
4 Rendezvous Implementation In CRP  
  4.1 Description  
5 CRP Rendezvous Implementation For Fire Bird Platform In uC/OS  
  5.1 Data Structures  
  5.2 Global Variables  
  5.3 Interface
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4 Tasks</td>
<td>21</td>
</tr>
<tr>
<td>5.5 Usage</td>
<td>24</td>
</tr>
<tr>
<td>5.6 Network Problems</td>
<td>24</td>
</tr>
<tr>
<td>5.7 Flowcharts</td>
<td>25</td>
</tr>
<tr>
<td>6 Case Study</td>
<td>28</td>
</tr>
<tr>
<td>6.1 Treasure Hunt</td>
<td>28</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>31</td>
</tr>
<tr>
<td>7.1 Summary</td>
<td>31</td>
</tr>
<tr>
<td>7.2 Problems Faced</td>
<td>31</td>
</tr>
<tr>
<td>7.3 Future Extensions</td>
<td>31</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Hierarchical Composition ................................................. 3  
2.2 Synchronous Parallel Composition ................................. 4  
2.3 Asynchronous Composition ............................................. 5  
2.4 CRSM Tools ........................................................................... 6  
2.5 CRSM Editor ........................................................................... 7  
2.6 CRSM Simulator ................................................................. 8  

3.1 Fire Bird IV ................................................................. 10  

4.1 State diagram ........................................................................... 17  

5.1 List Of Rendezvous Requests ................................................. 20  
5.2 CRP Rendezvous using uC/OS ............................................. 22  
5.3 CoordinatorTask Flowchart .................................................. 26  
5.4 PassiveTask Flowchart .......................................................... 27  

6.1 Abstract View Of Demo ...................................................... 29  
6.2 CRSM Representation Of Node1 .......................................... 29  
6.3 CRSM Representation Of Node2 .......................................... 30  
6.4 CRSM Representation Of Node3 .......................................... 30
Chapter 1

Introduction

Distributed Control Systems are gaining popularity due to their advantages like fault tolerance, concurrency and modularity. A distributed control system consists of number of autonomous subsystems controlling their own physical systems and at the same time having system-wide control laws. Real-time distributed systems are very complex as the individual nodes besides controlling their local environments, have to interact with other nodes to maintain system-wide properties. Communicating Reactive State Machines(CRSM) is a language for describing behaviors of such distributed controllers. It has a pictorial syntax, a precise formal semantics, can be easily verified and can be efficiently implemented.

Fire Bird is a lab robot developed by NEX Robotics[1] and ERTS Lab, IITB for the purpose of teaching. Although these Fire Bird robots are cheap, fairly complex systems can be build using them. Currently, programming environment is available for C that means they have to be programmed in C programming language. Initially we thought of implementing a tool which converts given CRSM descriptions to its equivalent C program that can be further compiled to run on Fire Bird. Implementing that tool will require the implementation of CRP Rendezvous Communication primitives for Fire Bird platform. Finally we have developed CRP Rendezvous Communication primitives for a distributed Fire Bird platform using uC/OS.

In this report, chapter 2 introduces CRSM, constructs provided by CRSM and current status of CRSM. Chapter 3 briefly explains hardware and software specifications of Fire Bird. Chapter 4 describes the work done by Prof. Ramesh on Rendezvous Implementation in Communicating Reactive Processes(CRP)[2]. Chapter 5 gives the implementation details of CRP Rendezvous communication using uC/OS[9] for Fire Bird. Chapter 6 is for concluding points and chapter 7 explains a case study Treasure Hunt.
Chapter 2

Communicating Reactive State Machines

A CRSM program consists of a set of processes, each process describing the behavior of an autonomous component of the distributed controller being described. In a distributed controller, a process besides controlling its local environment, communicates with other processes to maintain system-wide control laws. In CRSM, the local behavior of individual autonomous component is described using Argos (Maraninchi, 1992) and the interaction between nodes is described using CSP (Hoare, 1978) communication primitives for communication.

2.1 CRSM Constructs

CRSM is a network $N_1 || N_2 || \ldots || N_m$ of independent reactive programs or nodes. Each node $N_i$ has its own reactive interface with separate input/output signals and its own notion of instants. Intuitively, each node locally drives a part of a complex process that is handled globally by the network.[4]

2.1.1 CRSM Node

Each CRSM node is described using a structured Mealy machine in the style of Argos. In a simple form, a CRSM node is an edge labeled finite state machine with an initial state but no final state. The label on the transition is of the form $b/S$ where $b$ is a boolean expression involving input signals and $S$ is a set of output signals generated as a result of taking the transition. The input expression $b$ describes a combination of signal presence/absence status which when true, triggers the transition. The constructs
supported by CRSM for easy and efficient modeling of large descriptions are hierarchy, concurrency and signal hiding.

2.1.2 Hierarchical Composition

This construct is used to introduce hierarchical structure in a flat state machine. Let $A, B$ be two machines and $q$ be a state in machine $A$. Now the machine $B$ can be placed in the state $q$ of $A$. The resulting machine will behave as $A$ all the time except when it is in state $q$; in this state the behavior is decided by both $q$ and $B$. The state $q$ expresses an important operation preemption: in state $q$, computation proceeds as specified in $B$ so long as none of the transitions out of $q$ takes place, when an outgoing transition from $q$ takes place, the inside computation is preempted and the system transits to the target state of the transition and then on behaves like $A$.

![Figure 2.1: Hierarchical Composition](image)

The example machine shown in the figure 2.1 behaves as below

- Initially the machine will be in state S1
- On giving input $a$ it will take transition and goes to state S2. In this state the machine will behave as the inside automata so long as an input signal $b$ is not given
- Whenever input signal $b$ is given, machine will come out of state S2 and goes to state S1.
2.1.3 Synchronous Parallel Composition

Given two machines $A$ and $B$, a parallel machine that runs both $A$ and $B$ concurrently can be described using this construct. It describes a machine that simultaneously behaves both like $A$ and $B$. That means given a set of input signals representing the state of environment, appropriate transitions in both the automata are taken and result is the union of set of signals generated by both the automata. Further, the concurrent components can interact: signals generated by one component can trigger transitions in other components; such a triggering is instantaneous with no delay between the firing of the transitions in different components.

The example machine shown in figure 2.2 behaves as both automata simultaneously.

- Initially the machine will be in states A1 and B1

- On giving input $a$ it will take transition from A1 to A2. Then the system will be in states A2 and B1.

- In that state giving an input signal of $a$ will make the machine to transit from A2 to A1 by giving an output signal $b$. This output signal $b$ will trigger transition from B1 to B2. So the machine will be in states A1, B2.

![Figure 2.2: Synchronous Parallel Composition](image)
2.1.4 Signal Hiding

Signal hiding or local signal construct is defined to limit the scope of interaction between concurrent subcomponents. This construct is used to declare a signal as local to a subset of components so that generation of this signal will not trigger any transition in other components. This construct can also be viewed as an abstraction primitive that enable hiding of internal details. In the example shown in figure 2.2, signal $b$ is declared as local so that it will not trigger any transitions outside the corresponding automaton.

2.1.5 Asynchronous Composition

The above constructs are useful for describing centralized controllers only. For distributed controllers, CRSM nodes should have capability to communicate with each other. This is achieved by introducing a special kind of state called rendezvous state. A rendezvous state indicates a communication with another node via a channel. The channel name is specified as a label in the state. Entering rendezvous state initiates an attempt to communicate via specified channel. The communication is synchronous in the sense that both the nodes involved should be ready to communicate. Rendezvous state has (at least) one special exit transition which is taken when the communication succeeds; exactly which exit edge is taken is decided by the presence/absence of local signals specified on the edge. The machine stays in the rendezvous state till the communication succeeds or any normal preemptive transition takes place. Rendezvous states are flat with no hierarchy.

![Diagram](image)

Figure 2.3: Asynchronous Composition
The CRSM model shown in figure 2.3 describes a system where both the nodes will be following white line initially. Node2 will go to rendezvous state of type receive $CH$? and be in that state waiting for other node to communicate with it. On finding obstacle, node1 will stop moving and goes to rendezvous state of type send $CH!$. Now both the rendezvous are ready and so communication will be successful. After successful communication node2 will also stop moving.

2.2 Status of CRSM

The programming environment of CRSM consists of *editor, verification engine, simulator* and *CRSM2HandelC* tool. The Editor is used to describe systems graphically and can be saved these programs as textual CRSM(.tcrsm) and graphical format(.sim). The graphical format of CRSM model is used by simulator to run simulations over it. A translator is used to translate textual CRSM programs to *Promela* code which is the input language for verification tool *SPIN*. Models described in CRSM can be verified using the verification tool SPIN. The CRSM2HandelC tool also uses textual CRSM file as input and translates to *HandelC* which can further be compiled for FPGA.

![Figure 2.4: CRSM Tools](image)

2.2.1 CRSM Editor

CRSM Editor as shown in figure 2.5 is a user-friendly GUI to draw CRSM models. The CRSM constructs *viz* hierarchy, concurrency, signal hiding and asynchronous composition
can be easily specified using this editor. The editor gives a layer-wise view of the system by depicting the inner details of refined states at different levels. It provides a zoom-in, zoom-out and random exploration facility so that user can see the various levels of refinement.

![CRSM Editor](image.png)

**Figure 2.5: CRSM Editor**

CRSM models can be drawn using the buttons provided in the tool bar and this model can be saved in three different file formats:

- `.crsm` - graphical format to be re-opened by editor
- `.tcrsm` - textual format used by verification engine and CRSM2HandelC tool
- `.sim` - this file format is used by simulator

### 2.2.2 CRSM Simulator

The simulator user interface is as shown in figure 2.6. It is used to run the simulations over the model, which was described using the editor. It accepts input signals from the user, who acts on behalf of environment, and displays the state change to him. A simulation over a model depicts an execution sequence on that model. Simulator provides two windows *viz* Display window and Environment window.
Display Window

On starting, Display window shows an abstract view of the model as a network of nodes and channels. At the lower level it shows the consisting automata, the states and the transitions. It shows a hot spot on currently active state. By default the hot spot lies on the Init state. The Display window shows the movement of the hot spot and transition from the source state to the target state in response to the signals applied. If the target state has refinement, the display window will show the inner refined level of that state and another new window, named as Popped Frame, reflecting the parent status will be generated.

Environment Window

This window is used to control the simulation of the model by giving input signals to specific nodes. The set of input signals form the environment to which the model has to react. Initially the Environment window gives a provision to select one of the nodes from the available nodes in the model to which the user has to give input signals. After selecting the node, the Environment window shows the pure inputs,
pure outputs and local signals of the selected node. Now the user can select the input signals and start simulation. The result of the simulation for that reaction will be displayed on the Display window and the output signals generated are shown in the Environment window. The user can select other node by doing a context switch.

2.2.3 Verification

Verification of CRSM programs can be done by converting the textual CRSM program to PROMELA language, which is the input language for verification tool SPIN. Verification Engine has an inbuilt translator which takes .tcrsm file as input and generates functionally equivalent promela code, which can be fed to SPIN for verification. If the model is not correct then a valid counter example is generated. If a counter example exists then SPIN generates a .trail file, which contains information about simulation of counter example and number of steps after which system has entered BUG state.

2.2.4 CRSM2HandelC

CRSM2HandelC is FPGA implementation of CRSM. This will take .tcrsm file as input and generates equivalent Handel-C code. The Handel-C code thus generated by such a system can thereafter be used to configure FPGA based boards.[5]
Chapter 3

Fire Bird

Fire Bird is an educational robot built at NEX Robotics[1] and ERTS LAB, IITB for the purpose of teaching Embedded Systems and Sensor Networks courses at IIT Bombay. We have used Fire Bird IV robot running uC/OS for our implementation.

Figure 3.1: Fire Bird IV
3.1 Hardware Specification

- Microcontroller: ATMEL ATMEGA128

- Sensors:
  Three white line sensors
  One forward looking Sharp GP2D120C infrared range sensors (optional two additional sensors for left and right direction)
  Five bump sensors
  Two position encoders (shaft encoders)
  One directional light intensity sensor
  Battery voltage sensing
  Servo mounted sensor pod (optional)
  Wireless color camera (optional)
  Ultrasound scanner (optional)

- Indicators:
  2 x 16 Characters LCD
  Indicator LEDs
  Buzzer

- Control:
  Stand-alone
  PC as master and robot as slave
  Robot as master and other robot/robots as slave

- Communication:
  Wired RS232 (serial) communication
  Simplex infrared communication (From infrared remote to robot)
  Wireless 2.4 GHz XBee communication

3.2 Programming

3.2.1 Reading and Writing the I/O ports

The ports of Atmega128 microcontroller are bi-directional 8 bit I/O ports with optional internal pull-ups. Each individual pin of the I/O port can be separately configured as
input or output pin. Each parallel I/O port has three associated I/O registers:

1. Data direction register, \( DDRx \)
   The purpose of the data direction register is to determine which pins of the port are used for input and which pins are used for output. If logic one is written to \( DDRx \), then the port pin is configured as an output pin. If logic zero is written to \( DDRx \), then the port pin is configured as an input pin.

   \[
   DDRB = 0xF0;  
   \] //sets the 4 MSB bits of portB as output and
   \[
   //sets the 4 LSB bits of portB as input
   \]

2. Port driver register, \( PORTx \)
   If the port is configured as output port, then the value of \( PORTx \) register is given as output.

   \[
   DDRA = 0xFF;  
   \] //set all 8 bits of PORTA as output
   \[
   PORTA = 0xF0;  
   \] //output logic high on 4 MSB pins and
   \[
   //logic low on 4 LSB pins
   \]

   For pins configured as input, we can instruct the microcontroller to supply a pull up register by writing a logic 1 to the corresponding bit of the port driver register

   \[
   DDRA = 0x00;  
   \] //set all 8 bits of PORTA as input
   \[
   PORTA = 0xFF;  
   \] //pull up register is assigned to all the pins

3. Port pins register, \( PINx \)
   Reading from the input bits of port is done by reading port pin register

   \[
   x = PINA;  
   \] //read all 8 pins of port A

### 3.2.2 Motion control of Robot

1. Direction Control
   Two DC motors are connected to PortA of the microcontroller. IC L293D is used as a driver between the microcontroller and the DC motors. Direction of the motor is controlled by asserting one of the inputs to motor to be high(1) and the other to be low(0). To move the motor in opposite direction just interchange the logic applied to the two inputs of the motors. Asserting both inputs to logic high or logic low will stop the motor.
2. Controlling the velocity of the motor

Velocity of the motor can be controlled by applying Pulse Width Modulated (PWM) pulses to “Enable pins” of L293 driver. PWM is the scheme in which the duty cycle of a square wave output from the microcontroller is varied to provide a varying average DC output. By applying a PWM pulse, the motor is switched ON and OFF at a given frequency. In this way, the motor reacts to the time average of the power supply.

Timer3 of the microcontroller is configured to generate PWM with an 8 bit resolution. Timer3 counts from BOTTOM to MAX and then restarts from BOTTOM. The pulse width of the PWM for left and right motors is controlled by OCR3AL and OCR3BL registers of the timer. Timer3 generates a high pulse on PWM pin until the timer reaches the count mentioned in the OCR register and then switches to low until the timer reaches MAX count and then again the timer outputs a high pulse on the PWM pin and restarts counting from the BOTTOM.

The duty cycle of the PWM waveform is controlled by modifying the OCR3 register. The extreme values for the OCR3 register represent special cases when generating a PWM waveform output in the fast PWM mode. If OCR3 is set to BOTTOM, the output will be a narrow spike for each MAX+1 timer clock cycle. Setting OCR3 equal to MAX will result in a constantly high output i.e. the motor is rotating with max velocity.

Different ports are used to control different parts of the Fire Bird like LCD interface, Serial Communication etc. The complete functionality is explained in the Fire Bird manual[1].

3.3 Communication

Fire Bird IV comes with XBee wireless communication module attached. XBee module operates within the ZigBee protocol. It takes minimal power (1 mW) and provides reliable delivery of data between nodes. It operates at 2.4 GHz frequency band. Its range in indoor environment is 30 meters and in outdoor environment is 100 meters[14]. We have used this XBee modules for wireless communication between Fire Birds.

It has the Carrier Sense Multiple Access/ Collision Avoidance (CSMA/CA) multiple access protocol enabled to reduce the collisions when there is a contention for medium. In CSMA/CA, the sender checks the medium before sending the data to make sure that
no other node is transmitting at that time. If the medium is clear then the data is sent. Otherwise the node waits for a randomly chosen period of time, and then checks again to see if the medium is clear. This period of waiting time is called backoff factor and is counted down by a backoff counter. If the medium is clear when the backoff counter reaches zero then the node transmits the data. If the medium is not clear when the backoff counter reaches zero then the backoff factor is set again and the process is repeated.

CSMA/CA reduces the number of collisions but doesn’t make it zero. There can be collision if more than one node detects the medium at the same time when there is no node is transmitting. These collisions due to interference leads to the message corruption. A message is said to be corrupted if it doesn’t belong to the set of predefined messages. In our implementation, we have taken care of these corrupted messages as below.

- If a Task receives a corrupted message while protocol is running, it just discards that message, sends IGNORE message to the partner and restarts the rendezvous
- If a Task receives a IGNORE message it discards that message and restarts the rendezvous
- So, the state of the robot will be same before and after a corrupted message is received
Chapter 4

Rendezvous Implementation In CRP

In CRP, individual nodes are described using extended Esterel which has rendezvous commands. Compiling a CRP program involves compiling each node into a reactive process having its own environment. The environment of a CRP process along with I/O processes, will have a special process called Coordinator which handles the rendezvous statements. The interaction between the reactive process and coordinator is by means of logical signals.

This interaction involves three types of signals for each rendezvous.

1. **Start**($S_r$): request signal output by the node (reactive process) when it is ready to execute rendezvous $r$

2. **Return**($R_r$): input signal generated by the coordinator to indicate to the node that it can go ahead with the completion of the execution of the rendezvous $r$

3. **Kill**($K_r$): signal output by the node when a local reaction preempts the rendezvous $r$

The semantics of rendezvous requires that a node can complete a rendezvous provided its partner node also completes the rendezvous. Thus, in order to generate return signals, coordinators of different CRP nodes execute a complex protocol at the end of which they can generate the corresponding return signals to the respective CRP nodes.

Implementation of rendezvous is non-trivial as the following constraints are to be satisfied:

1. Mutual agreement over selection of a rendezvous required between the concerned processes
2. The implementation has to be distributed with no single process knowing instantaneously global state

This problem has been solved\cite{11}\cite{12}\cite{13} for Hoare’s CSP\cite{10}, but none of these solutions can be extended to CRP. The reason is that in CRP, an enabled rendezvous can be preempted by a local reaction. The paper\cite{3} proved that its impossible to have a fully distributed implementation of CRP that allows complete autonomy to reactive nodes.

The above problem is explained more clearly here. Rendezvous commitment problem is complicated due to the following assumptions A1, A2 and requirements P1, P2\cite{2}.

- **A1**: All communications by the coordinators are through asynchronous messages that take finite, nonzero but unbounded time to travel from source to destination

- **A2**: The reactive processes are invoked at arbitrary points of times; specifically not dependent on the states of the coordinators. The status of any input signal is independent of the state of the node

- **P1 Safety**: Coordinators, commit to only those rendezvous that are enabled. Further, they agree on the commitment of rendezvous in the sense that if one coordinator commits to a rendezvous then the corresponding partner coordinator also should commit to the same rendezvous

- **P2 Liveness**: It will not be the case that no rendezvous is committed when there is a rendezvous that is enabled for ever

But it has been shown in \cite{3} that it is impossible to have a solution that ensures both P1 and P2 with the assumptions A1 and A2.

The independent behavior of local reactive computations and the asynchronous nature of communication prevent any coordinator from arriving at the conclusion that there is a pair of CRP nodes ready to execute a rendezvous, within a finite amount of time using a finite number of messages \cite{2}.

We made the following assumptions about the underlying network that connect the CRP nodes.

- The nodes are connected to each other via point-to-point links that do not fail

- The links are bidirectional and the coordinators use these to communicate with each other
• The network is reliable i.e., messages are not lost and their order is preserved during transmission
• The nodes are numbered with distinct indices and each node knows its index

4.1 Description

The behavior of a coordinator can be explained by the following state diagram[2].

As shown in the fig. the coordinator(say $P_i$, coordinator of node $i$) can be in any one of the 5 states namely, E, W, Q1, Q2, F. Initially it is in E-state. In this state, it is waiting for the local reactive node to request for a rendezvous. Upon such a request, it jumps to Q1-state and executes a sequence of actions called the commitment sequence for committing to the rendezvous requested. If there are more than one requests from the reactive node then each of these requests are tried out in some order.

The commitment sequence for a rendezvous has two distinct phases: a consent phase followed by a confirmation phase. The consent phase is a sort of preliminary round in which the coordinator determines whether both the nodes involved in the rendezvous are ready to execute the rendezvous; the confirmation phase is the final round in which the actual commitment from the nodes are sought.
Suppose that $P_i$ selects the rendezvous $r$ for commitment which requires the participation of node $j$. Then $P_i$ enters the *consent phase* by transiting to Q2-state. In this state, it proceeds to get the consent (to execute $r$) from $P_j$ and itself. $P_i$ is said to capture a process (which may be itself), whenever the latter process agrees to execute the rendezvous $r$. This involves $P_i$ sending a request message to the other process followed by a positive or negative response from the latter. The *consent phase* succeeds when $P_i$ captures both the involved processes and fails otherwise.

If the *consent phase* is successful $P_i$ enters *confirmation phase*. In this phase $P_i$ first disables the local reaction, checks whether the rendezvous is not yet preempted and sends a *confirm* message to $P_j$ and transits to F state. $P_j$ upon receiving the *confirm* message, disables the next reaction of its local node and checks the state of the reactive node. If the requested rendezvous has not been preempted then it commits the rendezvous, enables the local reaction and sends a positive reply to $P_i$; in its subsequent reaction, the local reactive node at $j$ finds that the rendezvous is committed and takes appropriate action. $P_i$ upon positive reply to its *confirm* message, commits to the rendezvous, enables its local reactive computation, completes the *confirmation phase* as well as the *commitment sequence* for $r$ successfully and enters the E-state. If the rendezvous is found to be preempted by local reactions either at node $i$ or $j$, then $P_i$’s attempt to execute the *confirmation phase* fails; it then frees any captured process and goes back to Q1-state and executes the commitment sequence for another rendezvous, if any.

If the *commitment sequence* for all the requested rendezvous has failed, then $P_i$ transits to W-state. In this state, it responds to other coordinator processes’ requests. If any new rendezvous is requested by the local node, then $P_i$ enters the E-state and executes the commitment sequence for this rendezvous.
Chapter 5

CRP Rendezvous Implementation
For Fire Bird Platform In uC/OS

We implemented CRP (Communicating Reactive Processes) communication primitives [2] for Fire Bird using uC/OS. Here are the implementation details.

5.1 Data Structures

- **rendezvous**: It is a self-referential structure to maintain the information about rendezvous like channel id, source node, destination node, status, type.

- **message**: It is the structure used to represent a message. Its members include source node, destination node, channel id, value.

- **delayEntry**: It is the structure which represents an entry in the delayQueue. Its members are channel id and otherEnd(other end node).

- **capture**: This structure is used to maintain the list of nodes to be captured for rendezvous to complete successfully. Its members are head node, tail node and length. Capturing of nodes follows an order (like ascending) in order to overcome the problem of deadlock.

5.2 Global Variables

- **captured**: It is a boolean variable used to check whether a node is captured by some rendezvous.
• **rendez:** It is a pointer to the head of the list of rendezvous requests registered. Initially it points to NULL

• **rendezDone:** It is a boolean variable used to come out of processing of rendezvous request when the rendezvous is done by the Passive Part of the Coordinator

• **timeout:** It is a boolean variable used to come out of processing of rendezvous request when there is no response from other end for about 1 second

![List Of Rendezvous Requests](image)

**Figure 5.1: List Of Rendezvous Requests**

### 5.3 Interface

• **rendezvousSend:** It registers the send rendezvous i.e., adds an entry to the *rendez* list. It takes the parameters as source node, destination node and channel id. After registering, it will waits till the status of that rendezvous becomes COMMITTED. Semaphore is used while accessing the *rendez* list for consistency as it is shared by multiple tasks. Its prototype is as below

```c
void rendezvousSend(INT8U from, INT8U to, INT8U channel)
```

• **rendezvousReceive:** It registers the receive rendezvous i.e., adds an entry to the *rendez* list. It takes the parameters as source node, destination node and channel id. After registering, it will waits till the status of that rendezvous becomes COMMITTED. Semaphore is used while accessing the *rendez* list for consistency as it is shared by multiple tasks. Its prototype is as below

```c
INT8U rendezvousReceive(INT8U from, INT8U to, INT8U channel)
```
5.4 Tasks

- *InitializationTask*: This task is responsible for initializing all ports. This task will run only once. It deletes itself after initializations are done.

- *CoordinatorTask*: It is the main part of the CRP communication protocol. It periodically checks for the entries in the *rendez* list. If it is not NULL then selects one of the entry from the list and process that. It is the initiator of the protocol. The list of messages send by CoordinatorTask are REQUEST, CONFIRM, ABORT.

- *PassiveTask*: According to [2], it is part of Coordinator. Here in our implementation, we made it as a separate task so that it can be run in parallel with CoordinatorTask. This task will wait for some other coordinator to initiate the protocol. The list of messages send by PassiveTask are YES, NO, CONFIRM-YES and CONFIRM-NO.

- *NormalTask*: This is the user task which calls rendezvousSend or rendezvousReceive depending on the application requirements.

Tasks should be prioritized in the following order:

\[ \text{InitializationTask.PRIO} > \text{CoordinatorTask.PRIO} > \text{PassiveTask.PRIO} > \text{UserTask.PRIO} \]

i.e., CoordinatorTask should be given high priority than that of PassiveTask and so on.

The Active part of the Coordinator i.e., CoordinatorTask will perform the following actions.

- Prepares the *toCapture* structure with head as the lower node id and tail as the higher node id of the nodes involving in the communication. This will avoid the problem of deadlock by ordering the capturing of nodes. *toCapture.length* will be initialized to 2 as there are two participants point-to-point communication.

- Coordinator can be in any of the following states. It is initialized to *Q1*
  - Q1
    * sends the REQUEST message to the head of the *toCapture* structure
    * changes coordinatorState to *Q2*
  - Q2
    * receives reply from the head of the *toCapture* structure.
    * if the reply is YES, then
Figure 5.2: CRP Rendezvous using uC/OS
• sends the REQUEST message to the tail of the toCapture structure if it is not sent already
• changes coordinatorState to $Q^2$
* if both the nodes are captured, then
  • suspend all other tasks
  • sends the CONFIRM message to the other end node
  • changes coordinatorState to $F$
* if it fails to capture any one of them i.e., receives reply as NO, then
  • sends the ABORT message to the other end node.
  • changes coordinatorState to $Q^1$

– $F$
  * receives reply from the other end node
  * if the reply is CONFIRM_YES, then
    • resume all other tasks and the communication is committed
    • changes coordinatorState to $Q^1$
  * else if the reply is CONFIRM_NO,
    • resume all other tasks and the communication is aborted
    • changes coordinatorState to $Q^1$

– $T$
  * This state will be reached whenever there is timeout or rendezvous is done via PassiveTask. Here it just come out of the rendezvous without doing anything.

The PassiveTask perform the following actions.

• receives message from a node
  – if the message is REQUEST and the status of rendezvous is trying, then
    * if the node is not yet captured then, sends the YES message to the other end
    * else, sends the NO message to the other end

• receives message from a node
  – if the message is CONFIRM
* suspends all other tasks, checks the status of rendezvous and then resumes all other tasks
* if status of rendezvous is *trying*, then sends the CONFIRM_YES message to the node
* else sends the CONFIRM_NO message to the node

5.5 Usage

To use this CRP Rendezvous communication primitives in your application, just write your application as one or more tasks in the code provided by us and call the functions rendezvousSend() and rendezvousReceive() appropriately. Update the number of tasks N_TASKS to total number of tasks and give less priority (i.e., numerically higher value) to your tasks than that of InitializationTask, CoordinatorTask and PassiveTask.

5.6 Network Problems

- Message Losses and Reordering: ZigBee 802.15.4 protocol guarantees the reliable communication. So we can safely assume that there won’t be any message losses and reordering.

- Interference: CSMA/CA option of XBee module is enabled. So there is very less chances of interference

But still there is a chance for message corruption when more than one node senses the medium free at the same time. This is handled in our implementation as below.

- If a Task receives a corrupted message while protocol is running, it just discards that message, sends IGNORE message to the partner and restarts the rendezvous

- If a Task receives a IGNORE message it discards that message and restarts the rendezvous

- So, the state of the robot will be same before and after a corrupted message is received
5.7 Flowcharts

Let us consider an example which involves two nodes namely $A$ and $B$ with $id(A) < id(B)$. Assume the scenario where $A$ wants to communicate with $B$ over a channel $CH$. In CRSM it will be represented as below.

![Flowchart diagram](image)

In CRP Rendezvous communication protocol, the conversation will be either between CoordinatorTask of node $A$ and PassiveTask of node $B$ or between CoordinatorTask of node $B$ and PassiveTask of node $A$.

After rendezvous requests are registered, the rendez lists of the nodes look as below.

```
from : id(A)
to : id(B)
status : UNTRIED
type : SEND
```

```
from : id(B)
to : id(A)
status : UNTRIED
type : RECEIVE
```

rendezvous on Node A   rendezvous on Node B
Here are the flowcharts for CoordinatorTask and PassiveTask

Figure 5.3: CoordinatorTask Flowchart
Figure 5.4: PassiveTask Flowchart
Chapter 6

Case Study

6.1 Treasure Hunt

The system consists of more than one Fire Bird. Each Fire Bird will go in its own direction to find the treasure (obstacle is a treasure in our case). It is assumed that there will be only one treasure so that only one Fire Bird will find it. Whenever a Fire Bird finds the treasure it stops there and sends TreasureFound messages to all other Fire Birds. If a Fire Bird receives TreasureFound message from other Fire Bird, it stops moving in its own direction and moves towards the Fire Bird which has sent the TreasureFound message.

In this example, each node will have three tasks excluding CoordinatorTask and PassiveTask. They are Task1, Task2 and Task3. Task1 is responsible for finding out the treasure. As and when it finds the treasure, it stops and sends TreasureFound messages to other two nodes over separate channels. Task2 and Task3 are responsible for receiving the TreasureFound message from other nodes. Whenever a Fire Bird receives a TreasureFound message, it stops moving in its direction and starts moving towards the Fire Bird which has sent the TreasureFound message. The following CRSM descriptions will explain it more clearly.
Figure 6.1: Abstract View Of Demo

Figure 6.2: CRSM Representation Of Node1
Figure 6.3: CRSM Representation Of Node2

Figure 6.4: CRSM Representation Of Node3
Chapter 7

Conclusions

7.1 Summary

To summarize, we have started with implementing a translator which takes CRSM model description file (.crsm) as input and generates corresponding C code for Fire Bird. But there is no inbuilt channel communication available for Fire Bird. So we thought of exploring about implementing CRP rendezvous for Fire Bird using uC/OS. Finally we have ended with developing a full fledged CRP rendezvous communication primitives for Fire Bird using uC/OS which can be used for communication between multiple Fire Birds.

7.2 Problems Faced

- It is very hard to debug the code as there is only limited display space available on Fire Bird
- As this project involves testing the code on hardware, if there is any problem its difficult to find out whether the problem is in code or in hardware
- There are some implementation specific changes required for algorithm proposed by Prof. Ramesh S in [2]
- Sometimes interference between XBee communications lead to message corruption

7.3 Future Extensions

- From the experiments it is found that the CRP Rendezvous communication protocol is taking approximately 500 milliseconds time. This can be improved further by
tuning the delays involved in its implementation.

- Implementing the CRSM to C translator using this communication primitives. Then that will become the complete implementation of CRSM for a distributed robotic system.

- Integrating CRP rendezvous implementation with Esterel2C implementation by Shashidhar[15]
Bibliography

[1] www.nex-robotics.com


