SWE 760

Lecture 11 –
Detailed Real-Time Software Design

Reference:

Hassan Gomaa
Dept of Computer Science
George Mason University
Fairfax, VA

Figure 4.1 COMET/RTE life cycle model
Software Modeling for RT Embedded Systems

1 Develop RT Software Requirements Model
   – Develop Use Case Model

2 Develop RT Software Analysis Model
   – Develop state machines for state dependent objects
   – Structure software system into objects
   – Develop object interaction diagrams for each use case

3 Develop RT Software Design Model
   – Design of Software Architecture for RT Embedded Systems
   – Apply RT Software Architectural Design Patterns
   – Design of Component-Based RT Software Architecture
   – Design Concurrent RT Tasks
   – **Develop Detailed RT Software Design**
   – Analyze Performance of Real-Time Software Designs

Detailed Software Design

• Detailed Design of composite tasks
  – Active object that contain nested passive objects
  – Designed using task clustering criteria
  – Internal design of composite task
  – Design class interfaces of nested classes

• Design details of task synchronization
  • Passive objects accessed by more than one task

• Design connector classes
  • Address details of inter-task communication
Example of Design of Composite Task Containing Passive Objects

- Temporal clustering and device I/O objects
  - Temporal clustering task
    - Polled I/O
    - Activated periodically
  - Two passive devices
- Information hiding objects
  - Device I/O Objects
    - Hide details of how to read from device
    - Operations executed in thread of control of task

Fig 14.3a Before task clustering
Figure 13.2a Example of temporal clustering
- periodic I/O tasks before temporal clustering
Device I/O Class

Originally determined in Analysis Model
- Provides virtual interface to device
Hides actual interface to real world device
Insulate users of class from changes to real world device
- Input, Output, I/O class
Supports virtual interface via operations
Impact of changes to real world interface
- Specification of operations is unchanged
- Internals of operations change
Figure 14.3c Example of temporal clustering and input objects

- TemperatureSensorInput
  - initialize()
  - read(out-tempData)

- PressureSensorInput
  - initialize()
  - read(out-pressureData)

Figure 14.3d Temporal clustering task with nested input objects

- timerResource
  - hwDevice
    - DigitalClock

- coordinator
  - SensorMonitor
    - read(out-tempData)
    - read(out-pressureData)

- entity
  - sharedDataCommitResource
    - sharedMutualExclusionResource
      - SensorDataRepository
Example of Design of Composite Task Containing Passive Objects

- Control clustering task and passive objects
  - Control clustering task
  - Information hiding objects
    - State dependent control object
    - Device I/O Object
  - Operations executed in thread of control of control clustering task
- Concurrent access to classes
  - Classes inside task do not need synchronization
  - Classes outside task need synchronization

Figure 14.4 Example of Control Clustering with passive objects
State Machine Class
Hides contents of statechart / state transition table
  • Maintains current state of object
Process Event Operation
  • Called to process input event
  • Depending on current state and conditions
    • Might change state of object
    • Might return action(s) to be performed
Current State Operation
  • Returns the state stored in state transition table
If state transition table changes
Only this class is impacted

Fig. 14.2: Example of State Machine class

<table>
<thead>
<tr>
<th>MicrowaveStateMachine</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ initializeSTM ()</td>
</tr>
<tr>
<td>+ processEvent (in event, out action)</td>
</tr>
<tr>
<td>+ currentState () : State</td>
</tr>
</tbody>
</table>

Figure 14-4: Design of nested state machine and output classes

<table>
<thead>
<tr>
<th>PumpControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ initializeSTM ()</td>
</tr>
<tr>
<td>+ processEvent (in event, out action)</td>
</tr>
<tr>
<td>+ currentState () : State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PumpEngineOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ initialize ()</td>
</tr>
<tr>
<td>+ start ()</td>
</tr>
<tr>
<td>+ stop ()</td>
</tr>
</tbody>
</table>
Synchronization of Tasks Interacting via Passive Objects

Task interaction via shared data
  • Needs synchronization

Task interaction via passive data abstraction object
  • Hides structure of data repository
  • Hides synchronization from tasks
  • Two possible solutions
    • Mutual exclusion
    • Multiple readers / multiple writers
Information Hiding Objects  
Synchronization of Access

- Each information hiding object
  - Designed for application
- Mutually exclusive access to data repository
  - Only one task can access data repository at a time
  - Use binary semaphore
- Access by multiple readers / writers
  - Allows access to data repository
  - By many readers concurrently
    - Only one writer
Interaction Between Concurrent Tasks

- Mutual exclusion
  - Two or more tasks need to access shared data
  - Access must be mutually exclusive
- Binary semaphore
  - Boolean variable that is only accessed by means of two atomic (indivisible) operations
  - **acquire** (semaphore)
    - if the resource is available, then get the resource
    - if resource is unavailable, wait for resource to become available
  - **release** (semaphore)
    - signals that resource is now available
    - if another task is waiting for the resource, it will now acquire the resource

Example of Mutual Exclusion (Page 276)

- Solution uses one binary semaphore

```plaintext
public readAnalogSensor (in sensorID, out sensorValue, out upperLimit, out lowerLimit, out alarmCondition)
-- Critical section for read operation.
acquire (sensorDataStoreSemaphore)
sensorValue := sensorDataStore (sensorID, value)
upperLimit := sensorDataStore (sensorID, upLim)
lowerLimit := sensorDataStore (sensorID, loLim)
alarmCondition := sensorDataStore (sensorID, alarm)
release (sensorDataStoreSemaphore)
end readAnalogSensor;
```
Example of Mutual Exclusion (Pages 276-277)

```java
public updateAnalogSensor (in sensorID, in sensorValue)
-- Critical section for write operation.
acquire (sensorDataStoreSemaphore)
    sensorDataStore (sensorID, value) := sensorValue
    if(sensorValue >= sensorDataStore (sensorID, upLim))
        then sensorDataStore (sensorID, alarm) := high
    elseif(sensorValue <= sensorDataStore (sensorID, loLim))
        then sensorDataStore (sensorID, alarm) := low
    else sensorDataStore (sensorID, alarm) := normal
endif
release (sensorDataStoreSemaphore)
end updateAnalogSensor;
```

Example of Multiple Readers / Multiple Writers (Pages 277-278)

- Solution uses two binary semaphores and one integer

```java
public readAnalogSensor (in sensorID, out sensorValue, out upperLimit, out lowerLimit, out alarmCondition)
-- Read operation called by reader tasks. Several readers are allowed
-- to access the data store providing there is no writer accessing it.
acquire (readerSemaphore)
    Increment numberOfReaders
    if numberOfReaders = 1 then acquire (sensorDataStoreSemaphore)
release (readerSemaphore)
    sensorValue := sensorDataStore (sensorID, value)
    upperLimit := sensorDataStore (sensorID, upLim)
    lowerLimit := sensorDataStore (sensorID, loLim)
    alarmCondition := sensorDataStore (sensorID, alarm)
acquire (readerSemaphore)
    Decrement numberOfReaders
    if numberOfReaders = 0 then release (sensorDataStoreSemaphore)
release (readerSemaphore)
end readAnalogSensor
```
Example of Multiple Readers / Multiple Writers (Pages 277-278)

```java
public updateAnalogSensor (in sensorID, in sensorValue)
-- Critical section for write operation.
acquire (sensorDataStoreSemaphore)
    sensorDataStore (sensorID, value) := sensorValue
    if sensorValue >= sensorDataStore (sensorID, upLim)
        then sensorDataStore (sensorID, alarm) := high
    elseif sensorValue <= sensorDataStore (sensorID, loLim)
        then sensorDataStore (sensorID, alarm) := low
    else sensorDataStore (sensorID, alarm) := normal
endif;
release (sensorDataStoreSemaphore)
end updateAnalogSensor;
```

Connector Classes

- Classes designed to provide inter-task communication and synchronization
- Message buffering monitor classes
  - Synchronized (mutually exclusive) operations
• Asynchronous message communication
  - Message Queue connector class
• Synchronous message communication without reply
  - Message Buffer connector class
• Synchronous message communication with reply
  - Message Buffer and Response connector class
Asynchronous message communication
- Use message queue connector class
- Encapsulates message queue

Synchronization within Connector Object
- Synchronization between tasks (Java threads)
  - When task enters synchronized operation, it acquires semaphore
  - Synchronization methods
    - Wait
      - Task is suspended, releases semaphore
    - Signal (Notify in Java)
      - Wake up a suspended task
  - Condition wait
    - Check condition for waiting, e.g.,
      - while messageCount = 0 do wait
Message Queue Connector Class (Pages 285 – 286)

```java
monitor MessageQueue
-- Encapsulate message queue that holds max of maxCount messages
-- Monitor operations are executed mutually exclusively;
private maxCount : Integer;
private messageCount : Integer = 0;

public send (in message)
  while messageCount = maxCount do wait;
  place message in buffer;
  Increment messageCount;
  if messageCount = 1 then notify;
end send;

public receive (out message)
  while messageCount = 0 do wait;
  remove message from buffer;
  Decrement messageCount;
  if messageCount = maxCount-1 then notify;
end receive;
```

Figure 14.7 Design of message buffer connector

- Synchronous message communication without reply
  - Encapsulates a message buffer
  - Holds at most one message
**Message Buffer Connector Class (Page 287)**

```java
monitor MessageBuffer
-- Encapsulate a message buffer that holds at most one message.
-- Monitor operations are executed mutually exclusively
private messageBufferFull : Boolean = false;
public send (in message)
    place message in buffer;
    messageBufferFull := true;
    notify;
    while messageBufferFull = true do wait;
end send;

public receive (out message)
    while messageBufferFull = false do wait;
    remove message from buffer;
    messageBufferFull := false;
    notify;
end receive;
end MessageBuffer;
```

**Figure 14.8 Design of message buffer and response connector**

- Synchronous message communication with reply
  - Encapsulates a message buffer
  - Holds one message
  - Encapsulates a response buffer
  - Holds one response
Message Buffer & Response Connector Class (Pages 288 – 289)

monitor MessageBuffer&Response  
-- Encapsulates a message buffer that holds at most one message  
-- and a response buffer that holds at most one response.  
-- Monitor operations are executed mutually exclusively.

private messageBufferFull : Boolean = false;
private responseBufferFull : Boolean = false;

public send (in message, out response)  
place message in buffer;  
messageBufferFull := true;  
notify;  
while responseBufferFull = false do wait;  
remove response from response buffer;  
responseBufferFull := false;  
end send;

public receive (out message)  
while messageBufferFull = false do wait;  
remove message from buffer;  
messageBufferFull := false;  
end receive;

public reply (in response)  
Place response in response buffer;  
responseBufferFull := true;  
notify;  
end reply;
end MessageBuffer&Response;

Figure 145b Example of cooperating tasks using message queue connectors
Software Modeling for RT Embedded Systems

1 Develop RT Software Requirements Model
   – Develop Use Case Model
2 Develop RT Software Analysis Model
   – Develop state machines for state dependent objects
   – Structure software system into objects
   – Develop object interaction diagrams for each use case
3 Develop RT Software Design Model
   – Design of Software Architecture for RT Embedded Systems
   – Apply RT Software Architectural Design Patterns
   – Design of Component-Based RT Software Architecture
   – Design Concurrent RT Tasks
   – **Develop Detailed RT Software Design**
   – Analyze Performance of Real-Time Software Designs