Design of Real-Time Software Workshop

Part 3: Workshop Solutions “Mine Pump”

Onno van Roosmalen
O.S.vanRoosmalen@chello.nl

© O.S. van Roosmalen, 2010
Analysis Class Diagram

MinePump

- WaterSensor
  - value
  - getValue()

- MethaneSensor
  - value
  - getValue()

- FlowSensor
  - value
  - getValue()

- Pump
  - setOn()
  - setOff()
Analysis State Diagram

- **water**
  - **NotTooLow**
  - when(water.value ≤ L)
  - when(water.value ≥ H)
  - NotTooHigh

- **methane**
  - **TooHigh**
  - when(methane.value < M)
  - when(methane.value ≥ M)
  - NotTooHigh

- **flow**
  - **Faulty**
  - when(flow.value = 0 and pump.On)
  - Ok

- **pump**
  - **On**
  - when(water.NotTooHigh) /pump.setOff()
  - when(methane.TooHigh) /pump.setOff()
  - when(flow.Faulty) /pump.setOff()
  - when(water.NotTooLow and methane.Ok and flow.Ok) /pump.setOn()
  - Off

- **MinePump**
  - Ok

- **MinePump**
  - Ok
Design: Composite Structure with Ports

MinePump

- high : ReceivePort[1]
- low : ReceivePort[1]
- methane : SensorPort[1]
- flow : SensorPort[1]
- pump : ActuatorPort[1]
Refinement Step

- MinePump
  - high : ReceivePort[1]
  - WaterSensor[1]
  - Pump[1]
  - low : ReceivePort[1]
  - MethaneSensor[1]
  - FlowSensor[1]
  - methane : SensorPort[1]
  - flow : SensorPort[1]
  - pump : ActuatorPort[1]
  - water[1]
  - methane[1]
  - flow[1]
∀t_1, t_2 : (methane ≥ M) during [t_1, t_2] → pump = off during [t_1 + dc, t_2]
class MethaneSensor extends Thread {
    ...
    public void run() {
        state = Ok;
        while (true) {
            Value x = methane.getValue();
            if ((x >= M) && (state==Ok)) {
                state = TooHigh;
                pump.send(M)
            } else if (x < M && (state==TooHigh)) {
                state = Ok;
                pump.send(0);
            }
        }
    }
}
Water Sensor behavior

∀t₁, t₂ : (water ≥ H ∧ safe) during [t₁, t₂] → pump = on during [t₁ + dh, t₂]

∀t₁, t₂ : (water ≤ L) during [t₁, t₂] → pump = off during [t₁ + dl, t₂]
class WaterSensor extends Thread {

    public void run() {
        while (true) {
            high.receive();
            pump.send(H)
            low.flush();
            low.receive();
            pump.send(L);
            high.flush();
        }
    }
}
Flow Sensor Behavior

\[
\text{faulty at } t \equiv \exists t_1 < t : (\text{flow} < \text{min} \land \text{pump} = \text{on}) \text{ at } t_1 \\
\forall t_1, t_2 : (\text{faulty} \land \text{pump} = \text{on}) \text{ during } [t_1, t_2] \rightarrow \text{pump = off during } [t_1 + \alpha, t_2]
\]
Naive Mapping FlowSensor

- The flow and pump status must hold simultaneously:

```java
class FlowSensor extends Thread {
    ...
    public void run() {
        state = Ok;
        while (Ok) {
            if (pump.state == On) {
                Value x = flow.getValue();
                if ((x = 0) {
                    state = Faulty;
                    pump.send(0);
                }
            }
        }
    }
}
```
Pump Process

• The Pump process must handle receptions from various ports
  – methane
  – flow
  – water
• It can not be occupied in an activity checking flow because it must receive signals
• It can not be blocking on reception of signals because it must check flow (when pump is on)
• We must also take care that pump state does no change during flow measurement
Pump Behavior?

```
receive(L) <<from>> water / high=false
receive(M) <<from>> methane / safe=false
receive(H) <<from>> water / high=true
receive(0) <<from>> methane / safe=true
when(flow.getValue()==0) / error=true

On
entry / pump.setOn()
exit / pump.setOff()

[safe and not error] [else]
[safe and not error] [else]
[high and not error] [else]
```
class Pump extends Thread {
    
    public void run() {
        while (not error) {
            port = receiveAll();
            if (port == water) {
                high = port.receive() == H;
            } else if (port == methane) {
                safe = port.receive() == M;
            } else if (?) {
                if (state == On) error = flow.getValue == 0;
            }
            if (state == Off && safe && high && !error) {
                state = On; pump.setOn();
            } else if (state == On && (!safe || !high || error)) {
                state = Off; pump.setOff();
            }
        }
    }
}
Refinement Step (2)
class Pump extends Thread {
    ...
    
    public void run() {
        error = false;
        while (!error) {
            if (state==On) {
                Value x = flow.getValue();
                if ((x =0) {
                    error = true;
                    pump.setOff()
                }
            }
        }
    }
}
class Pump extends Thread {
    
    public void run() {
        error = false;
        while (!error) {
            if (state==On) {
                if (x == 0) {
                    error = true;
                    pump.setOff()
                }
                flow.send(); flow.receive(x);
            }
        }
    }
}
FlowSensor

class FlowSensor extends Thread {
    ...
    public void run() {
        while (true) {
            pump.receive();
            pump.send(flow.getValue());
        }
    }
}

class Pump extends Thread {
    ...
    public void run() {
        error = false;
        while (!error) {
            if (state==On) {
                flow.send(); flow.receive(x);
                if ((x =0) {
                    error = true;
                    pump.setOff()
                }
            } else {
                ...}
        }
    }
}
class Aux extends Thread {
  ...
  public void run() {
    while (true) {
      pump.send();
    }
  }
}
... Sequential Cohesion

class Pump extends Thread {
    ...
    public void run() {
        error = false;
        while (!error) {
            aux.receive();
            if (state==On) {
                flow.send(); flow.receive(x);
                if ((x =0) {
                    error = true;
                    pump.setOff()
                }
            }
        }
    }
}
Refinement Step (3)
class Pump extends Thread {
    …
    public void run() {
        while (not error) {
            port=receiveAll();
            if (port=water)  high = (port.receive()==H);
            if(port=methane) safe = (port.receive()==0);
            if(port=aux) {
                port.receive();
                if (state==On) { flow.send(); error = (flow.receive()==0);}
            }
            if(state=Off && high && safe && !error ) {
                state=On; pump.setOn();
            } else if(state=On && (!high || !safe || error)) {
                state=Off; pump.setOff();
            }
        }
    }
}
Execution (Scheduling) Directives

• The program can not be executed straightforwardly
  – The programming language doe not contain any reference to time
    • no wait(time) instruction
    • no indication of periods and deadlines

• These aspects pertain to the way the program is executed over time, not the program logic

• Periodic execution of instances of active classes MethaneSensor and Aux.

• Message (sporadic-event) driven execution of instance of active class Water.

• Also message driven execution of instances of active class Pump and FlowSensor.

• What are the tasks for schedulability analysis??
Task assignment

• Periodic task (time triggered)
  – Methane measurement
    • Body *MethaneSensor* and message handling *Pump*
  – Flow measurement
    • Body *Aux* and message handling *Pump* and *FlowSensor*

• Sporadic task (message triggered)
  – Water measurement
    • Body *WaterSensor* and message handling *Pump*

• Resources (these will cause blocking!!)
  – Shared by all tasks
    • Message handling of *Pump* and *FlowSensor*
## Task parameters

<table>
<thead>
<tr>
<th>tasks/resources</th>
<th>type</th>
<th>period</th>
<th>deadline</th>
<th>worst-case execution times</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{\text{Methane}}$</td>
<td>periodic</td>
<td>$T_{\text{Methane}}$</td>
<td>$dc - T_{\text{Methane}}$</td>
<td>$C_{\text{Methane}}$</td>
</tr>
<tr>
<td>$\tau_{\text{Flow}}$</td>
<td>periodic</td>
<td>$T_{\text{Flow}}$</td>
<td>$df - T_{\text{Flow}}$</td>
<td>$C_{\text{Flow}}$</td>
</tr>
<tr>
<td>$\tau_{\text{Water}}$</td>
<td>sporadic</td>
<td>$dlh$</td>
<td>$\min(dh, dl)$</td>
<td>$C_{\text{Water}}$</td>
</tr>
<tr>
<td>Pump</td>
<td>shared</td>
<td></td>
<td></td>
<td>$C_{\text{Pump,Flow}}$ included in $C_{\text{Pump,Flow}}$</td>
</tr>
<tr>
<td>FlowSensor</td>
<td>shared</td>
<td></td>
<td></td>
<td>$C_{\text{Pump,Water}}$ $C_{\text{Pump,Methane}}$</td>
</tr>
</tbody>
</table>

- $T_{\text{Methane}}$: worst-case execution time for tasks related to Methane.
- $T_{\text{Flow}}$: worst-case execution time for tasks related to Flow.
- $dlh$: worst-case execution time for tasks related to Water.
- $dh, dl$: worst-case execution times for Pump and Flow sensors.
- $C_{\text{Methane}}$, $C_{\text{Flow}}$, $C_{\text{Water}}$, $C_{\text{Pump,Flow}}$, $C_{\text{Pump,Water}}$, $C_{\text{Pump,Methane}}$: costs associated with different resources and tasks.
Schedulability Formula

• From earlier schedulability theory (see also Burns and Wellings, 1996)
  – Recursive relation for the response time of a task

\[ r_i = \sum_{j=1}^{i-1} C_j \left[ \frac{r_j}{T_j} \right] + C_i + \max_{j=i+1}^m C_{Mine,j} \]
## Task values

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_i$</th>
<th>$C_{Pump,i}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{\text{Methane}}$</td>
<td>12u</td>
<td>6u</td>
<td>$30 - T_{\text{Methane}}$</td>
<td>$T_{\text{Methane}}$</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_{\text{Flow}}$</td>
<td>14u</td>
<td>13u</td>
<td>$100 - T_{\text{Flow}}$</td>
<td>$T_{\text{Flow}}$</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_{\text{Water}}$</td>
<td>9u</td>
<td>5u</td>
<td>10,000</td>
<td>100,000</td>
<td>3</td>
</tr>
</tbody>
</table>

- We are trying to find the slowest processor, i.e. the largest $u$ (in milliseconds per statement), under which deadlines are still guaranteed.
Schedulability Analysis (DM)

\[ r^*_i = \sum_{j=1}^{i-1} C_j \left[ \frac{r_i}{T_j} \right] + C_i + \max_{j=i+1}^{m} C_{\text{Mine},j} \]

- Consider task with shortest deadline \((\tau_{\text{Methane}})\) \(t = 30 - T_{\text{Methane}}\)
  - \(W_M(t) = C_1 + B_1 = 12u + 13u = 25u \leq 30 - T_{\text{Methane}}\) (finish within deadline)
  - \(\leq T_{\text{Methane}}\) (finish within period)

- Maximum value of \(u = 0.6\) when \(T_{\text{Methane}} = 15\) msec
Schedulability Analysis (cont’d)

– Consider this $u_{max}$. Is it also ok for the other deadlines?

Check next shortest deadline task ($\tau_{Flow}$)

t = 15, 30, 45, ... $T_{Flow}$ (msec, order not certain)

- $W_C(15) = C_1 + C_2 + B_2 = 12u + 14u + 5u = 31u \leq 15$ (finish within period)
  
  $18.6 \leq 15$ \textbf{no}

- $W_C(30) = 2C_1 + C_2 + B_2 = 24u + 14u + 5u = 42u \leq 30$ (finish within period)
  
  $25.2 \leq 30$ \textbf{ok}

Also finish within deadline: $25.2 \leq 100 - T_{Flow}

\textbf{Maximum $T_{Flow} = 74.8$ msec} (gives lowest processor utilization)

Check next task ($\tau_{Water}$)

t = 15, 30, 45, 60, 74.8, 75, 90 ... 100,000 (msec)

- $W_W(15) = C_1 + C_2 + C_3 = 12u + 14u + 9u = 35u \leq 15$ \textbf{no}

- $W_W(30) = 2C_1 + C_2 + C_3 = 24u + 14u + 9u = 47u \leq 30$ \textbf{ok}
class Pump extends Thread {
    private high=false, safe=true, error=false, state=Off
    public synchronized void setWater(Value value) {
        high = (value==H);
        generateResponse();
    }
    public synchronized void setMethane(Value value) {
        safe = (value==0);
        generateResponse();
    }
    public synchronized void run() {
        while (waitForNextPeriod(Tf) {
            if (state==On) error = (flow.getValue()==0);
            generateResponse();
        }
    }
    private void generateResponse() {
        if(state=Off && high && safe && !error ) { state=On; }
        else if(state=On && (!high || !safe || error)) { state=Off; }
    }
}