Assignment 5: Lustre and Real-time Java

1DT056: Programming Embedded Systems
Uppsala University

February 23rd, 2012

You can achieve a maximum number of 20 points in this assignment. 12 out of the 20 points are required to pass the assignment. Assignments are to be solved by students individually.

Exercise 1 Basic Lustre nodes (6p)

In this exercise, you will implement a few basic programs in the Lustre programming language. Before you submit your solutions, simulate each program using the Luke tool, in order to identify and eliminate potential bugs.

Luke binaries for various platforms and further resources are provided on the following page: `http://www.it.uu.se/edu/course/homepage/pins/vt12/lustre`.

1. In the lecture, a node Sum has been implemented that returns, at each instant, the sum of all inputs that the node has seen up to that point. Now implement SumReset, a version of Sum with an additional Boolean input Reset that makes Sum start over from 0: **(2p)**

   ```
   node SumReset(X : int; Reset : bool)
     returns (S : int);
   ```

2. Implement a node Average with the signature **(2p)**

   ```
   node Average(X : int; Reset : bool)
     returns (A : int);
   ```

   that outputs the average of the stream X since the last Reset.

3. Implement a node HasHappenedWithin with signature **(2p)**

   ```
   node HasHappenedWithin(X : bool; N : int)
     returns (Y : bool);
   ```

   that returns whether X has happened within the last N execution steps.

Exercise 2 Door control system in Lustre (8p)

We will now use Lustre to develop a simple system that controls the entrance and exit of a laboratory dealing with bio-hazardous materials. The idea is that, in order to enter or leave the laboratory, you have to go through two doors in sequence. However, these doors cannot be open at the same time, because otherwise air infected with bacteria or viruses might freely flow out of the laboratory. A biosensor reports to the door system if any dangerous organisms exist in the area between the doors.

The physical shape of the door system is as follows: The inner door to the laboratory is called door 1; the outer door is called door 2. Each door has a door sensor, a button, and a lock. The door sensors check if the corresponding door is open, producing the boolean signals `Open1` for door 1 and `Open2` for door 2. The buttons can be pushed by a human to unlock the corresponding door, producing the boolean signals `Button1` and `Button2`. The locks are controlled by the system; they listen to the boolean signals `Unlock1` and `Unlock2`. Finally, the biosensor produces a Boolean signal `BioHazard`, that is true precisely when there are dangerous organisms in the room between the two doors.

The intended use of the system is as follows. In the beginning, both doors are closed and locked. When a person wants to leave the laboratory, she pushes the button belonging to door 1. If the outer door is not open, the inner door will be unlocked and can be pushed open by the human. Then, she waits until the door closes, before she pushes the button of the outer door. The outer door only unlocks if there is no bio-hazardous situation. If it unlocks, the human can push the door open and walk out. People can walk into the laboratory in a similar way.

The following three requirements have to hold for the system:

- **R1** A door should not be locked when it is open.
- **R2** The two doors should not be open at the same time.
- **R3** When there is a bio-hazardous situation, the outer door must remain closed and locked.

In order to ensure these requirements, the control system has to rely on a *environment assumption* describing the behaviour of a physical door:

- **E1** A door cannot be opened when it is locked.

The door controlling node has the following Lustre interface:

```
node System( Open1, Button1, Open2, Button2, BioHazard : bool )
  returns ( Unlock1, Unlock2 : bool );
```

1. Give an implementation of the node `System`. Simulate your implementation using Luke and make sure that it behaves consistently with the description given above. **(4p)**

2. We now want to verify that the `System` implementation satisfies the requirements **R1**, **R2**, **R3**. As explained in the lecture, this is done by constructing a synchronous observer containing the requirements in form of Lustre expressions. The observer will also assume that **E1** holds (for both doors) during the entire execution of the system, which is done with the help of the `Sofar` node. **(4p)**

   This observer has the form:

```
node EnvironmentDoor( Open, Unlock : bool )
  returns ( E1 : bool );
let
  E1 = ...;
tel

node ReqSystem( Open1, Button1, Open2, Button2,
                BioHazard : bool )
  returns ( R1, R2, R3 : bool );
  var Env, Unlock1, Unlock2 : bool;
let
  (Unlock1, Unlock2) =
    System( Open1, Button1, Open2, Button2, BioHazard );
  Env =
    Sofar(     EnvironmentDoor( Open1, Unlock1 )
           and EnvironmentDoor( Open2, Unlock2 ) );

  R1 = Env => ...;
  R2 = Env => ...;
  R3 = Env => ...;
tel
```

   Replace the `...` in the code with Lustre expressions faithfully formalising **E1**, **R1**, **R2**, **R3**. Verify that your `System` implementation satisfies the requirements using Luke; if verification fails, you have to correct either your implementation or your formalisation of the requirements.

   Note: Be careful when formalising **E1**! For instance, the translation as `(not Unlock) => (not Open)` is incorrect, since it does not faithfully capture the phrase "door cannot be opened."

Exercise 3 RTSJ Communication (6p)

A naïve real-time developer has developed the following embedded program, using the Real-Time Specification for Java (RTSJ). The program is supposed to wait until a button connected to the system is pressed, and then perform a lengthy computation task, in the end of which some data is output (to a serial port (UART), accessed using the stream System.out). The programmer has introduced two real-time threads to implement this behaviour: a thread "buttonTask" that periodically checks the status of the button, and a thread "actionTask" that performs the length computation once the button has been pressed. The communication between the two tasks is realised using a shared variable "buttonFlag:"

```
import javax.realtime.*;

public class ButtonHandlers {
  static boolean buttonFlag = false;

  private static class ButtonTask extends RealtimeThread {
    private boolean getButtonStatus() { ... }

    ButtonTask() {
      super(new PriorityParameters(PriorityScheduler.instance()
                                   .getMinPriority()+1),
            new PeriodicParameters(new RelativeTime(20, 0)));
    }

    public void run() {
      for (;;) {
        if (getButtonStatus()) {
          buttonFlag = true;
          //  wait until signal has been received
          while (buttonFlag);
        }
        waitForNextPeriod();
      }
    }
  }

  private static class ActionTask extends RealtimeThread {
    ActionTask() {
      super(new PriorityParameters(PriorityScheduler.instance()
                                   .getMinPriority()),
            new PeriodicParameters(new RelativeTime(50, 0)));
    }

    public void run() {
      for (;;) {
        // check whether button was pressed
        if (buttonFlag) {
          // reset the flag
          buttonFlag = false;
```

```
            System.out.println("action!");
          }
          waitForNextPeriod();
        }
      }
    }

    public static void main(String[] args) {
      new ButtonTask().start();
      new ActionTask().start();
    }
}
```

Sadly, the program is incorrect: on a particular (single-core) microcontroller, it never outputs anything on UART, even if the button is pressed.

1. Explain why no output is sent to the serial port. **(2p)**
2. We want to replace the inefficient communication via the variable "buttonFlag" by introducing a semaphore that counts the number of times the button has been pressed. Since RTSJ does not provide counting semaphores out of the box, first show how such a semaphore can be implemented using the normal Java synchronisation features. The semaphore class should have the following signature: **(4p)**

```
class CountingSemaphore {
  /**
   * Precondition: initialCount >= 0
   */
  CountingSemaphore(int initialCount) { ... }
  /**
   * Decrement the semaphore, block if the semaphore
   * is already at zero.
   */
  public synchronized void take() { ... }
  /**
   * Increment the semaphore, unblock (at most one)
   * thread possibly blocking at "take".
   */
  public synchronized void give() { ... }
}
```

Then, rewrite the threads "buttonTask" and "actionTask" to use a semaphore instead of the variable "buttonFlag." Instead of polling, the "actionTask" should become an aperiodic thread that uses the semaphore operations for blocking until a button press occurs.

Note: you can test your semaphore implementation using any standard Java installation. If you want to experiment with RTSJ, you can use the reference implementation provided at `http://www.timesys.com/java/`. However, since the reference implementation is only available for Linux, we don't provide a central installation on the lab computers.

Submission

Solutions to this assignment are to be submitted
via email to othmane.rezine@it.uu.se before the exercise session on
Friday, March 2nd, 2012, 10:15 (room 1245).

Make sure that your solution is clearly marked with your name and personal number. No solutions will be accepted after the exercise session.