Prof. Dr. A. Podelski, Dr. B. Westphal
S. Feo Arenis

Sommersemester 2015

Softwaretechnik/Software Engineering

http://swt.informatik.uni-freiburg.de/teaching/SS2015/swtvl

Exercise Sheet 5

Early submission: Friday, 2015-07-03, 14:00 Regular submission: Monday, 2015-07-06, 14:00

Exercise 1 – (Object and Class Diagrams) (5/20 Points)

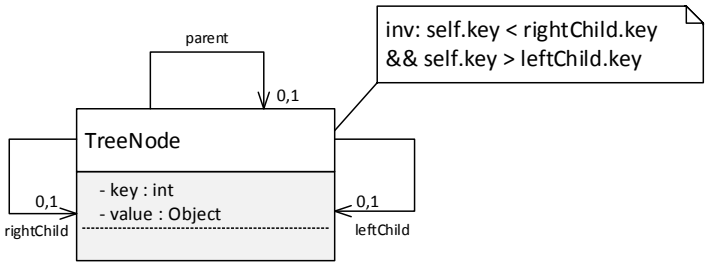

Figure 1: Class diagram for a binary search tree.

Consider the class diagram shown in Figure 1.

(i) Present the class diagram as a *signature*. Provide the **abstract syntax** of the diagram. (1)

(ii) Consider the formula

$$F := \forall\, self : Node \bullet key(self) > key(leftChild(self)) \wedge key(self) < key(rightChild(self))$$

shown in OCL-notation in Figure 1.

Give **system states** $\sigma_1, \sigma_2, \sigma_3$ and their corresponding object diagrams such that

a) formula F evaluates to *true* for σ_1,

b) formula F evaluates to *false* for σ_2,

c) formula F evaluates to $\perp$ (undefined) for σ_3. (3)

(iii) Choose one of your system states from (ii) and **prove**, using the interpretation function as defined in the lecture, that the formula actually evaluates to the value you claimed. (1)

Exercise 2 – (Object Diagrams for Documentation) (5 Bonus)

A *binary search tree* is a data structure that allows finding whether a key is contained and retrieve the value associated to that key in $O(log_2\ n)$, where n is the number of keys stored. Keys are stored in the tree such that for every subtree, all the keys reachable through the left child are strictly smaller than the key of the root and all the keys reachable trough the right child are strictly larger than the key of the root.
Explain, using appropriate illustrative object diagrams, how the data structure defined by the class diagram in Figure 1 is supposed to be used to implement binary search trees.
Provide in particular object diagrams which illustrate corner cases and give negative examples that show how the data structure should *not* be used. (5 bonus)

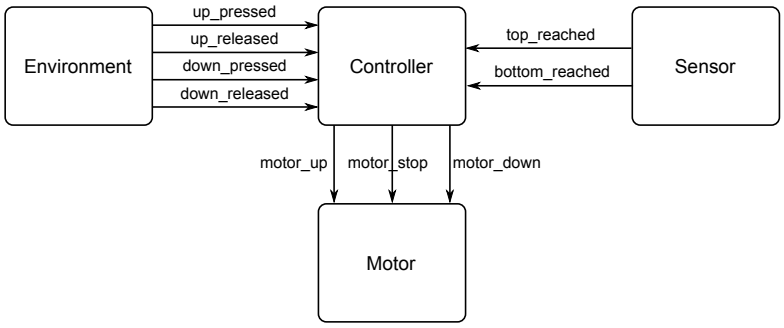

Figure 2: Architecture diagram of the UPPAAL model of the power window.

Exercise 3 – (Design Modeling) (10/20 Points)

In this exercise, you assume the role of a development engineer for the power window project of the previous exercise sheets. The system architect gives you the task of designing a controller for the power window.
You are given a model of the power window system (including an environment model). It consists of four communicating finite automata (CFAs):

- `Environment`: sends events using the channels `up_pressed`, `up_released`, `down_pressed` and `down_released` to notify the controller about the user's interaction with the window button.
- `Sensor`: sends the events `top_reached` and `bottom_reached` to notify the controller about the window reaching its extreme positions. The current position of the window can be read from the variable `sensor_position` that can take the values `BOTTOM` for the lowermost position, `BELOW_THRESHOLD` for a position above the bottom and below the auto full close threshold, `ABOVE_THRESHOLD` for a position above the auto full close threshold and `TOP` for the uppermost position.
- `Motor`: accepts movement commands on the channels `motor_up`, `motor_down` and `motor_stop`. Issuing a movement command in a direction that exceeds the range of motion of the window brings the motor to the *broken* state, where it ceases to operate.
- `Controller`: is empty; it just consists of one initial state. Your task is to design it and to verify that your `Controller` CFA complies with the customer requirements.

The files necessary for this task are available for download together with this exercise sheet.

(i) **Design** a controller for the power window with **auto full close** using UPPAAL. Use the file `powerwindow.xml` and fill out the `Controller` template. Provide a thorough documentation of your design. Convince your readers that your design satisfies the power windows requirements discussed so far. Also submit the XML file of your model. (4)

Hint: In particular the (anti-)scenarios represented by the LSCs in the previous exercise sheet are power window requirements. Creating and saving (and submitting) simulation traces (cf. (ii) below) may also be useful for this task.

Hint: UPPAAL *provides options to export automata as pictures as, e.g., as PDF files, and to color the states of automata. Consider to use these facilities to improve your documentation.*

(ii) **Demonstrate** the auto full close function by providing a trace using the simulator module of UPPAAL. Document the trace while describing the strategy of your controller for the auto full close function. Also save the trace file and submit it. (1)

(iii) **Verify** that your design allows reaching all sensor positions, does not break the motor and that your model does not contain deadlocks. During development, you can import the queries

in the file `powerwindow.q` and use the verifier module of UPPAAL. A good design should satisfy all queries.

Document the verification process by using the command line UPPAAL verifier (see usage instructions below). Report the output (in appropriate, well-readable form) and discuss the number of states explored during verification. What are the maximum and minimum number of states explored? Why is the minimum so low and the maximum so high? Are the numbers plausible to you? (1)

Now, perform the design, verification and demonstration tasks for a power window that also supports obstacle detection. In the extended model, the `Sensor` CFA additionally sends events `obstacle_detected` and `obstacle_removed`. Moving the motor up when an obstacle is present breaks the motor.

(iv) **Design** a controller for the power window with **auto full close** using UPPAAL. Use the file `powerwindow_obstacle.xml` and fill out the `Controller` template. Document your design, possibly as an extension of the documentation provided for task (i). Also submit the XML file of your model.

(v) **Demonstrate** the obstacle detection function by providing a trace using the simulator module of UPPAAL. Document the trace while describing the behavior of your controller when an obstacle is detected. Also save the trace file and submit it. (1)

(vi) **Verify** that your design allows reaching all sensor positions, does not break the motor and that your model does not contain deadlocks. During development, you can import the queries in the file `powerwindow.q` and use the verifier module of UPPAAL. A good design should satisfy all queries.

Document and discuss the verification process similar to task (iii). (1)

Hint: If you want to reuse the controller of task (i), you can use the option File → Import Template from the menu bar of UPPAAL. *But note that you are supposed to submit* two *XML files, one for (i) and one for (iv).*

Uppaal Usage Instructions

- **Editor and Simulator**

  UPPAAL is installed in the Linux machines of the computer pool. To execute it, use the following command line:

  ```
  /usr/local/ufrb/uppaal/uppaal-4.0.14/uppaal
  ```

  You can use it directly or by logging in remotely using Secure Shell (SSH), e.g.

  ```
  ssh -X username@aschgabad.informatik.uni-freiburg.de
  ```

  where `username` is your pool account name. Note that X11 forwarding (`-X`) is necessary for the editor and simulator front-end; the verifier is a pure command line tool and does not require this option.

- **Stand-alone Verifier**

  To run the verifier from the command line on one of the pool's Linux hosts, use, e.g., the command

  ```
  /usr/local/ufrb/uppaal/uppaal-4.0.14/bin-Linux/verifyta -u
                                              powerwindow.xml powerwindow.q
  ```

  (one line!).

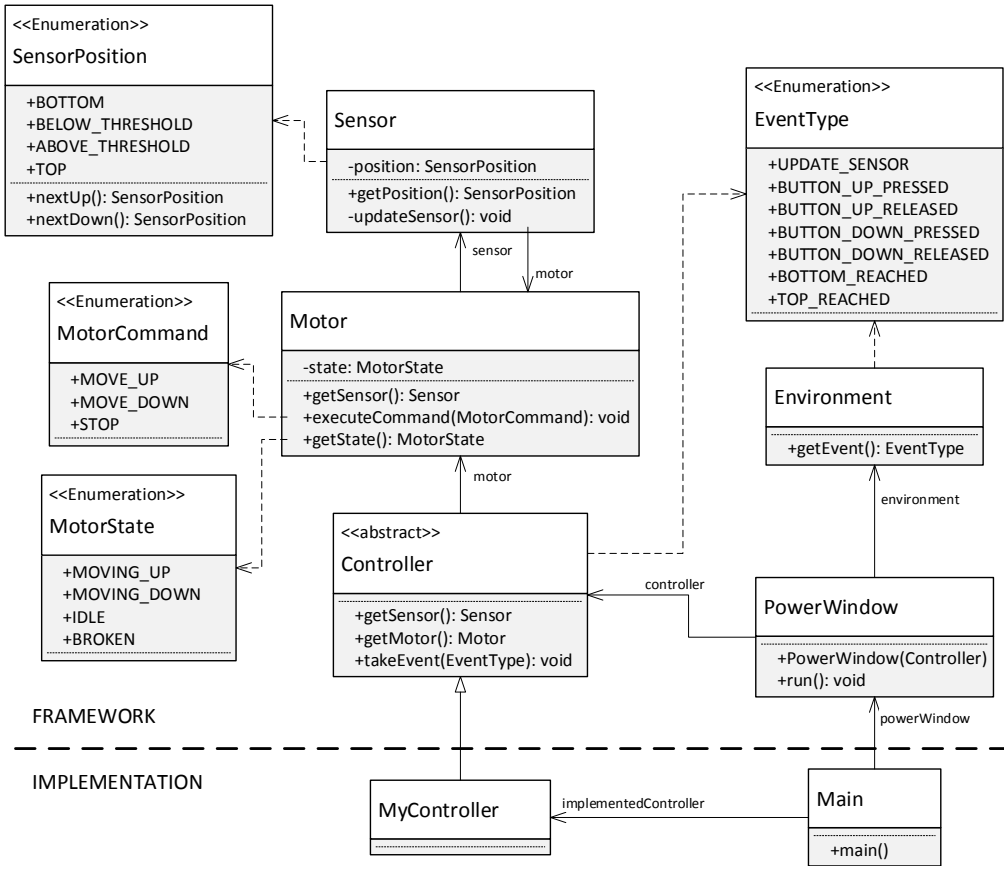

Figure 3: Partial class diagram of the simulator framework. You are required to provide the implementation of `MyController`.

Exercise 4 – (From Design to Implementation) (5/20 Points)

In this exercise, you are required to implement in software the controller from exercise 3(i). For that, we have provided a Java framework that simulates the power window with auto full close using a controller implementation provided by you. For your reference, a class diagram of the framework is shown on Figure 3. Additionally, an example interaction of the user with the simulator is shown on Figure 4.

(i) Provide **an implementation** of the controller you designed in the previous exercise by *extending* the class `Controller`. See, e.g. Figure 5. Also make sure to provide an entry point for the simulator that initializes it using your controller implementation. For this, create a `main` method that invokes the `run()` method of `PowerWindow`. See, e.g., Figure 6. Submit the files containing the source code of your solution.

Convince your readers that you *faithfully* implemented your design from Exercise 3, i.e. that your implementation can do what the design does and does not do anything else in addition.

Discuss: what (informal) definition of "program implements model" did you use in your argumentation? (5)

Hint: concrete computation paths of your implementation (obtained from executing it) may be useful for this task.

To assign the points, your (appropriately documented) solution code will be reviewed and tested to check whether it fulfills the requirements of the power window, i.e., that your controller imple-

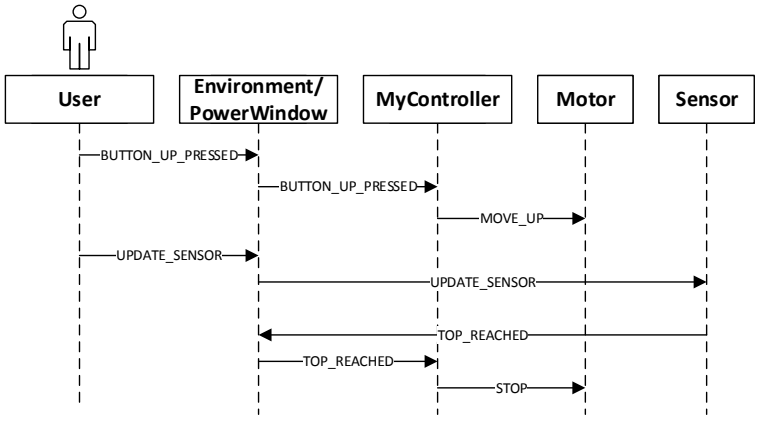

Figure 4: Sequence diagram showing the expected behavior of the power window for an example interaction with the simulator. The user sends event `BUTTON_UP_PRESSED` first, it is forwarded by the simulation loop to the controller implementation, which correctly responds by sending the command `MOVE_UP` to the motor. Then, the user sends event `UPDATE_SENSOR`. The event is forwarded to the sensor. The sensor updates its state and replies with event `TOP_REACHED`, which is forwarded immediately to the controller. The controller responds by sending command `STOP`.

mentation moves the window according to the button commands and never breaks the motor.

Note: the easiest way to solve this task may be to use Java in the provided simulation framework. It allows you (and your tutor) to compile, run, and test your solution.
Using Java is not required though. Yet if you choose to solve this task in any other programming language, provide sufficient explanation and links to resources that allow your tutor to check that your implementation runs. (It is at the discretion of your tutor to consider the effort too high; if it must be as exotic as, e.g., "Turbo Pascal 3.0 for CP/M", you may want to negotiate with your tutor first.)
Plus, when not using Java, provide a very convincing explanation (including the used language constructs) of your implementation — you can only assume that your tutor knows Java.

Usage Instructions

The simulator framework is found in the file `powerwindow.jar`, available together with this exercise sheet. Make sure to read the documentation of the package which is found in the file `powerwindow-doc.zip`. To read it, extract the contents of the file in a directory of your choice and open the file `index.html`.

We will assume that your solution consists of the files `Main.java` and `MyController.java`. To test your solution on a Linux host in the computer pool, perform the following steps:

1. Make sure `powerwindow.jar` is in the current directory, along with your solution files.
2. Compile your files using the command

   ```
   javac -cp powerwindow.jar Main.java MyController.java
   ```

3. Run your program using the command

   ```
   java -cp "powerwindow.jar:." Main
   ```

   On windows platforms, replace the colon ':' with a semicolon ';'.
4. You will see the simulator prompt where you can enter a number to select the event you like to send to your controller.

   ```
   Motor state: IDLE
   Window position: BELOW_THRESHOLD
   Select an event to send
   0: UPDATE_SENSOR
   ```

```
1: BUTTON_UP_PRESSED
2: BUTTON_UP_RELEASED
3: BUTTON_DOWN_PRESSED
4: BUTTON_DOWN_RELEASED
5: BOTTOM_REACHED
6: TOP_REACHED
Your choice: 0
```

The chosen event `UPDATE_SENSOR` (number 0) causes the sensor to move to the next position according to the state of the motor and send the appropriate events to your controller when reaching the extreme positions. This particular event is *never* sent to your controller, it is only used to operate the environment model.

After making a choice, the state of the simulator is updated by in particular calling the `takeEvent` method of the controller implementation. You will see a new prompt indicating the new state of the system.

```
>> Motor received command MOVE_UP
Motor state: MOVING_UP
Window position: BELOW_THRESHOLD
Select an event to send
0: UPDATE_SENSOR
1: BUTTON_UP_PRESSED
2: BUTTON_UP_RELEASED
3: BUTTON_DOWN_PRESSED
4: BUTTON_DOWN_RELEASED
5: BOTTOM_REACHED
6: TOP_REACHED
Your choice:
```

Figure 5: Sample file `MyController.java`

```
import de.unifreiburg.powerwindow.*;

public class MyController extends Controller {

  public void takeEvent(EventType event) {
    // Read the position of the sensor
    SensorPosition position = getSensor().getPosition();

    // Write the code to process the events
    switch (event) {
    case TOP_REACHED:
    case BOTTOM_REACHED:
    case BUTTON_DOWN_RELEASED:
    case BUTTON_UP_RELEASED:
    case BUTTON_DOWN_PRESSED:
    case BUTTON_UP_PRESSED:
      // Send a command to the motor
      getMotor().executeCommand(MotorCommand.STOP);
      break;
    default:
      break;
    }
  }
}
```

Figure 6: Sample file `Main.java`

```
import de.unifreiburg.powerwindow.*;

public class Main {
  static Controller implementedController = new MyController();
  public static void main(String[] args) {
    new PowerWindow(implementedController).run();
  }
}
```