A few notes on the textbook's introduction to and examples of behaviours, plus additional details on how synchronize motor commands in LeJOS

March 12, 2018

Henning Christiansen
Roskilde University
http://www.ruc.dk/~henning/, henning@ruc.dk

The introduction to behaviours in Chapter 18 of Bagnall's book is rather short and imprecise, and the example program in many respects, do no follow the book's own guidelines for good practice. Here we discuss the issues and show some example programs, and we can also demonstrate that the LeJOS implementation of behaviours is not stable. However, these are the premises when using open source, freeware software platforms, so it is always a matter of experimentation. – In the following, we assume that you are familiar with this chapter and that you have the text, including examples.

This note uses example programs that are variants of those in the book; however, we do not not include the remote control.

1 On the meaning of behaviours under LeJOS's `subsumption` class

As described in the book, a behaviour is defined in terms of three methods, `takeControl()`, `action()` and `suppress()`.

The overall control algorithm, hidden inside the `Arbitrator` class, is testing from time to time the different behaviour's `takeControl()` method, and from these results and the priorities of these behaviours, it may decide to stop the current behaviour (call it B_1) and give control to another (call it B_2). This may involve the following events.

- B_1.`takeControl()` is either running still, or it has terminated. It may have initiated some motor commands that are still running, even if `takeControl()` itself has terminated. The `takeControl()` method may also be parked in a busy-waiting loop, e.g., of the form `while(···) Thread.yield()`.
- Now assume the system decides to start B_2 by calling first B_1.`suppress()`:
  - the programmer should have made B_1.`suppress()` in such a way that it cleans up after B_1 which involves stopping B_1.`action()` if it is still running (e.g., in a busy-waiting

loop as shown above), update various data structures if that is relevant, and stop any other threads and motors that it has initiated, in case these otherwise would interfere with another behaviour;

and next B_2.`action()`:

– the programmer should have made B_2.`action()` in such a way that it can start in the intended way, independently of which (if any) behaviour running before it; in should initiate data structures (if relevant), perhaps start some motors and perhaps enter a busy-waiting loop ((and we can go to the first bullet, with B_2 taking the role as a new B_2)).

2 Good practice and the lack of it

The book mentions an important advantage of using behaviours as they are found in LeJOS can be programmed independently of any other behaviour.

The `BehaviorForward` method shown p. 273 does not quite obey live up to this standard.

1. The method `public boolean takeControl() {return true;}` does only give sense because *it is expected to be the behaviour with the lowest priority.* If it has, say, the highest priority, it would block for any other behaviour.

2. The `suppress()` method is empty, which means that that the two motors that were started by the `action()` will keep running. An inspection of *the entire program* can show that this does not give problems, because all behaviours in the program give commands to both motors (or terminates the run), so they are reset that way.

. Point 2 is obviously problematic, as you cannot expect that a new behaviour (referring to, say, only the left motor) added to the program will behave as expected; we will consider different ways of repairing the program.

Point 1 invites to a more interesting discussion. As it appears it is not possible to give this behaviour a higher priority by changing its position in the array of behaviours given to the `Arbitrator` object. On the other hand, such a `takeControl()` methods seems to be the only relevant for the behaviour of the lowest priority in a well-designed robot, so that it would never get stuck in a situation, where it could not find out what to do.

3 Our first version of the book's example program

Our example program does not use the infrared remote control, and instead there is a behaviour for stopping the robot by either clicking the touch sensor or pressing the so-called escape button (up and to the left) on the EV3 brick. Furthermore, we have used the infrared sensor as described in our previous course not on high-level sensors. All files can be found on moodle.

BehaviourForward.java

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Behavior;

// Adapted from Bagnall p. 273.

public class BehaviourForward implements Behavior {
   RegulatedMotor leftMotor;
   RegulatedMotor rightMotor;

   public BehaviourForward(RegulatedMotor left, RegulatedMotor right) {
      this.leftMotor = left; this.rightMotor = right;
   }

   public boolean takeControl() {
      return true;
   }

   public void action() {
      leftMotor.forward();
      rightMotor.forward();
   }

   public void suppress() {
      // nothing to do
   }
}
```

InfraredAdapter.java

```
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3IRSensor;
import lejos.robotics.SampleProvider;

// adapted from Bagnall p. 275-276; this is an example of a high-level sensor
public class InfraredAdapter extends Thread {
    int objectDistance = 1000;
   EV3IRSensor irSensor = new EV3IRSensor(SensorPort.S2);
   SampleProvider sp = irSensor.getDistanceMode();

   public InfraredAdapter() {this.setDaemon(true);this.start();}

   public void run() {
```

```
      while(true) {
         float [] sample = new float[sp.sampleSize()];
         sp.fetchSample(sample, 0);
         if((int)sample[0]==0) objectDistance=1000;
         else objectDistance=(int)sample[0];
         Thread.yield();
      }
   }

   public int getObjectDistance() {return objectDistance;}
}
```

BehaviourAvoidObject.java

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Behavior;

// adapted from Bagnall p. 274
public class BehaviourAvoidObject implements Behavior {
   RegulatedMotor leftMotor;
   RegulatedMotor rightMotor;
   InfraredAdapter irAdapter;
    boolean backing_up=false;

   public BehaviourAvoidObject(RegulatedMotor left, RegulatedMotor right, InfraredAdapter irA) {
      this.leftMotor = left; this.rightMotor = right; this.irAdapter=irA;
   }

   public boolean takeControl() {
      return irAdapter.objectDistance < 25; // global variable
   }

   public void action() {
      backing_up=true;
      leftMotor.rotate(-600,true); rightMotor.rotate(-600);
      leftMotor.rotate(450,true); rightMotor.rotate(-450);
      backing_up=false;
   }

   public void suppress() {
      // The following from the book looks weird; not in accordance with the specification
      while(backing_up) Thread.yield();
   }
```

```
}
```

BehaviourStopByTouch.java

```
import lejos.hardware.Button;
import lejos.hardware.Sound;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3TouchSensor;
import lejos.robotics.SampleProvider;
import lejos.robotics.subsumption.Behavior;

public class BehaviourStopByTouch implements Behavior {
   EV3TouchSensor touch = new EV3TouchSensor(SensorPort.S1);
   SampleProvider touched = touch.getTouchMode();
   float[] sample = new float[touched.sampleSize()];

   public boolean takeControl() {
       touched.fetchSample(sample,0);
       int t = (int) sample[0];
      return t==1 || Button.ESCAPE.isDown();
   }

   public void action() { System.exit(1); }

   public void suppress() {
   }
}
```

TestingBehavioursMainNewName.java including main method

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;
import lejos.hardware.motor.*;
import lejos.hardware.port.MotorPort;

public class TestingBehavioursMainNewName {
   public static void main(String[] args) {
      RegulatedMotor leftMotor = new EV3LargeRegulatedMotor(MotorPort.B);
      RegulatedMotor rightMotor = new EV3LargeRegulatedMotor(MotorPort.A);
      Behavior b1 = new BehaviourForward(leftMotor,rightMotor);
      InfraredAdapter ir = new InfraredAdapter();
      Behavior b2 = new BehaviourAvoidObject(leftMotor,rightMotor,ir);
```

```
      Behavior b3 = new BehaviourStopByTouch();
      Behavior[] b1b2b3 = {b1,b2,b3};
      Arbitrator arby = new Arbitrator(b1b2b3);
      arby.go();
   }
}
```

4 Different attempts to improve the moving forward behaviour

At we explained above, one of the problems with the `BehaviourForward` behaviour, was that it did not stop the motors, that it had started, so it would easily conflict with another, new behaviour supplied by the programmer. We fix this by adding `stop()` commands as follows.

BehaviourForward_2.java

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Behavior;

public class BehaviourForward_2 implements Behavior {
   RegulatedMotor leftMotor;
   RegulatedMotor rightMotor;
....
   public void suppress() {
       leftMotor.stop();
      rightMotor.stop();
   }
}
```

To test it, you need to change the following line in the `main()` method ,

```
    Behavior b1 = new BehaviourForward(leftMotor,rightMotor);
```

into

```
    Behavior b1 = new BehaviourForward_2(leftMotor,rightMotor);
```

According to our best understanding, this ought to work. We can observe that it works for a while (!!), but many activations in a row of `BehaviourAvoidObject` may stress the system, so that the `BehaviourForward_2` cannot be interrupted by the two other behaviours.

We may hypothesize that there is some thread left over which is consuming all the available CPU time, but we have no good explanation of this overall pattern.

Our next, and more successful attempt is to let `suppress()`use a flag (i.e., a common boolean variable) to communicate from to `action()` that it should get out of its busy waiting loop, clean up and exit. It looks as follows.

BehaviourForward_3.java

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Behavior;

public class BehaviourForward_3 implements Behavior {
   RegulatedMotor leftMotor;
   RegulatedMotor rightMotor;

   public BehaviourForward_3(RegulatedMotor left, RegulatedMotor right) {
      this.leftMotor = left; this.rightMotor = right;
   }

   public boolean takeControl() {
      return true;
   }

   private boolean suppressed = false;

   public void action() {
      suppressed = false;
      leftMotor.forward();
      rightMotor.forward();
      while(!suppressed) Thread.yield();
      leftMotor.stop();
      rightMotor.stop();
   }

   public void suppress() {
      suppressed=true;
   }
}
```

To test it, you need to change the critical line in the `main()` method into

```
Behavior b1 = new BehaviourForward_3(leftMotor,rightMotor);
```

This works, it is much cleaner from a structural point of view, and it does not suffer from the strange stress problems as did the BehaviourForward_2.

However, it displays another problem concerning the motor commands: the two motors do not stop at the same time. It appears that `leftMotor.stop()` issues a command to the motor, and wait until it has stopped, and first then, it begins executes `rightMotor.stop()`.

5 Synchronizing motors

LeJOS has recently been added facilities to synchronize motors, so we avoid the problems of having them not to stop or start at the same time, which introduces a lot of noise.

The methods that we can use for this is are included in the class `lejos.hardware.motor.BaseRegulatedMotor` that implements the interface `RegulatedMotor` that we have been using in these examples. We copy-paste from the documentation:

void	**synchronizeWith(RegulatedMotor[] syncList)** Specify a set of motors that should be kept in synchronization with this one.
void	**startSynchronization()** Begin a set of synchronized motor operations
void	**endSynchronization()** Complete a set of synchronized motor operations.

Find all files on moodle packed as `SynchMotors.zip`; file names are similar to those used so far, but this new files should be put into a new "project". The way it works can be seen in this new version of the behaviour for moving forward.

BehaviourForward.java – with synchronization

```
import lejos.robotics.RegulatedMotor;
import lejos.robotics.subsumption.Behavior;

// Modified version
// use synchronization to have motors work more symetrically

public class BehaviourForward implements Behavior {
   RegulatedMotor leftMotor;
   RegulatedMotor rightMotor;

   public BehaviourForward(RegulatedMotor left, RegulatedMotor right) {
      this.leftMotor = left; this.rightMotor = right;
   }

   public boolean takeControl() {
      return true;
   }

   private boolean suppressed = false;

   public void action() {
```

```
      suppressed = false;
      // NEW STUFF: synchronizing motors (apoligize Java syntax)
      RegulatedMotor[] syncList = {leftMotor};
      rightMotor.synchronizeWith(syncList );
      rightMotor.startSynchronization();
        leftMotor.forward();
        rightMotor.forward();
      rightMotor.endSynchronization();
      while(!suppressed) Thread.yield();
      rightMotor.startSynchronization();
        leftMotor.stop();
        rightMotor.stop();
      rightMotor.endSynchronization();
   }

   public void suppress() {
      suppressed=true;
   }
}
```

The file BehaviourAvoidObject.java is modified in an analogous way.

Conclusion: motors do stop and start at then same time.