Behavior-based robots

Henning Christiansen

Robotics course, RUC
March 12, 2018

Today's program

- ....
- (No assignment this week)

A video and a discussion

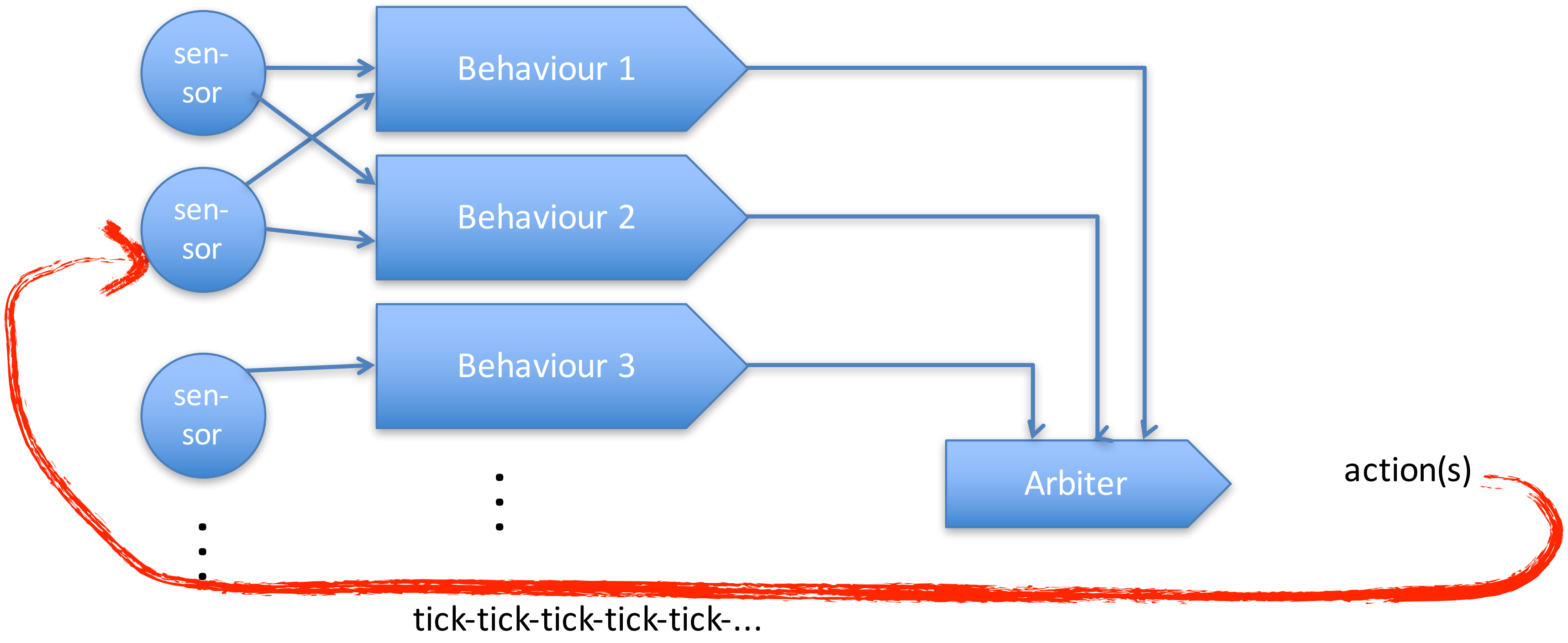
- https://www.youtube.com/watch?v=QzkfGIvYJEg

- Robots needs capable sensors and appropriate ways of reacting designed for their expected (and unexpected) environment

- High-level sensors (cf. last week) simplifies code and helps modularization

- Describing the robot's overall doings in terms of a number of separate behaviours – as above

Behaviours-based robots: the basic idea



- Sensors: Basic or high-level
- Behaviour: Servo or ballistic (≈ simple or complex)
- Arbiter/arbitrator: How to select and combine different behaviours

Background

- Attributed to R.A. Brooks, 1986
- Inspired by insects; survival instincs
- A fundamental layer in "all" interesting robots
- May be complemented with advanced AI, planning algorithms etc.
- An "intelligent" layer may interact with the behavioural≈instictive level.
    - "Stop thinking, run!!!!!"
    - Background "intelligent" thread may plan and re-plan dynamically; one among other behaviours: "follow the current plan"

Different versions

- Most general: all behaviours calculating in parallel, suggesting actions, arbitrator selects optimal combinations of those actions

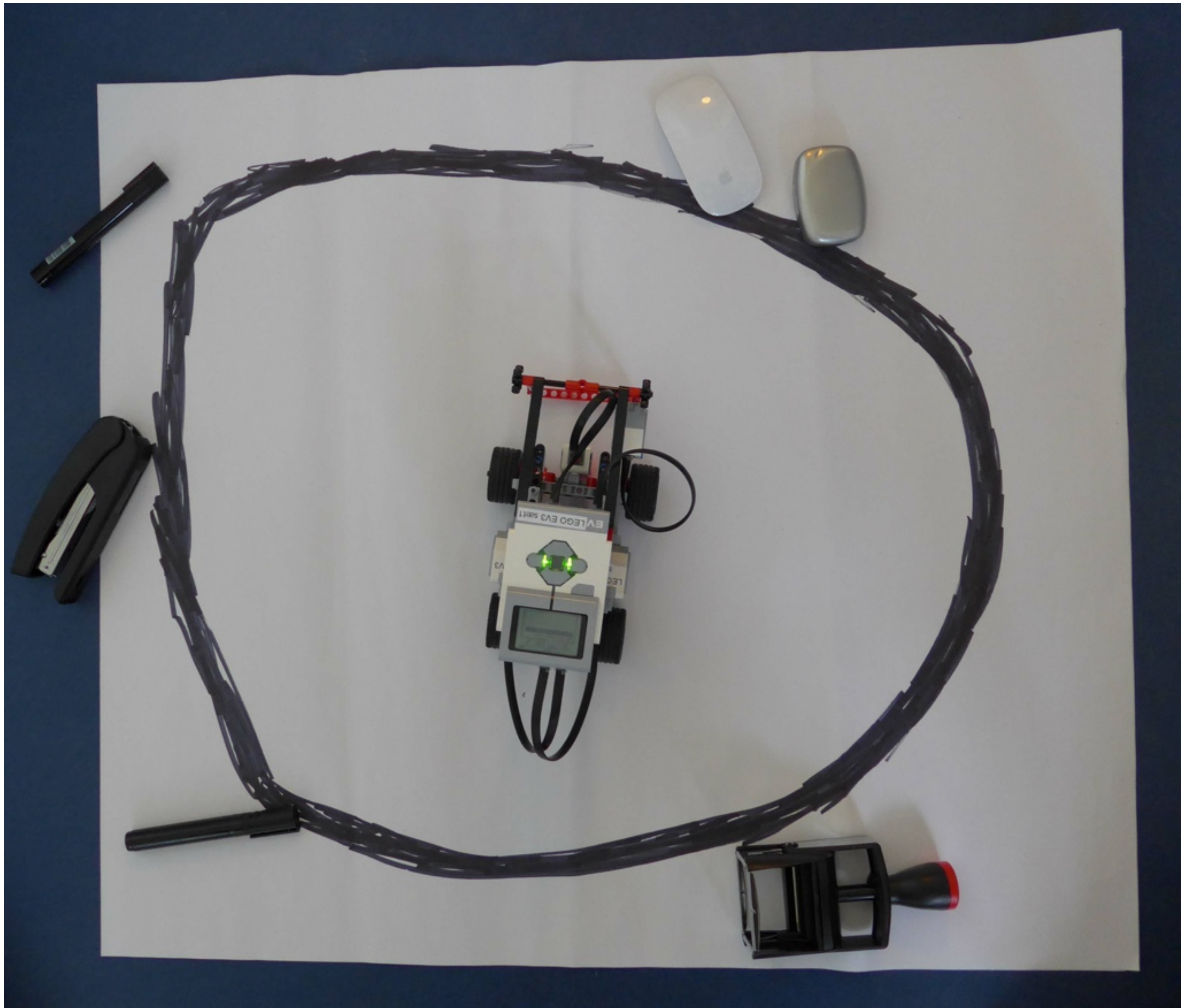
- Brooks' article defines "subsumption architecture": behaviors ordered by priority; higher priority actions may interrupt lower priority (a bit complicated; description not obvious)

- Simplistic but effective: Drop arbitrator, decision tree determins next behaviour froim censors (next slides)

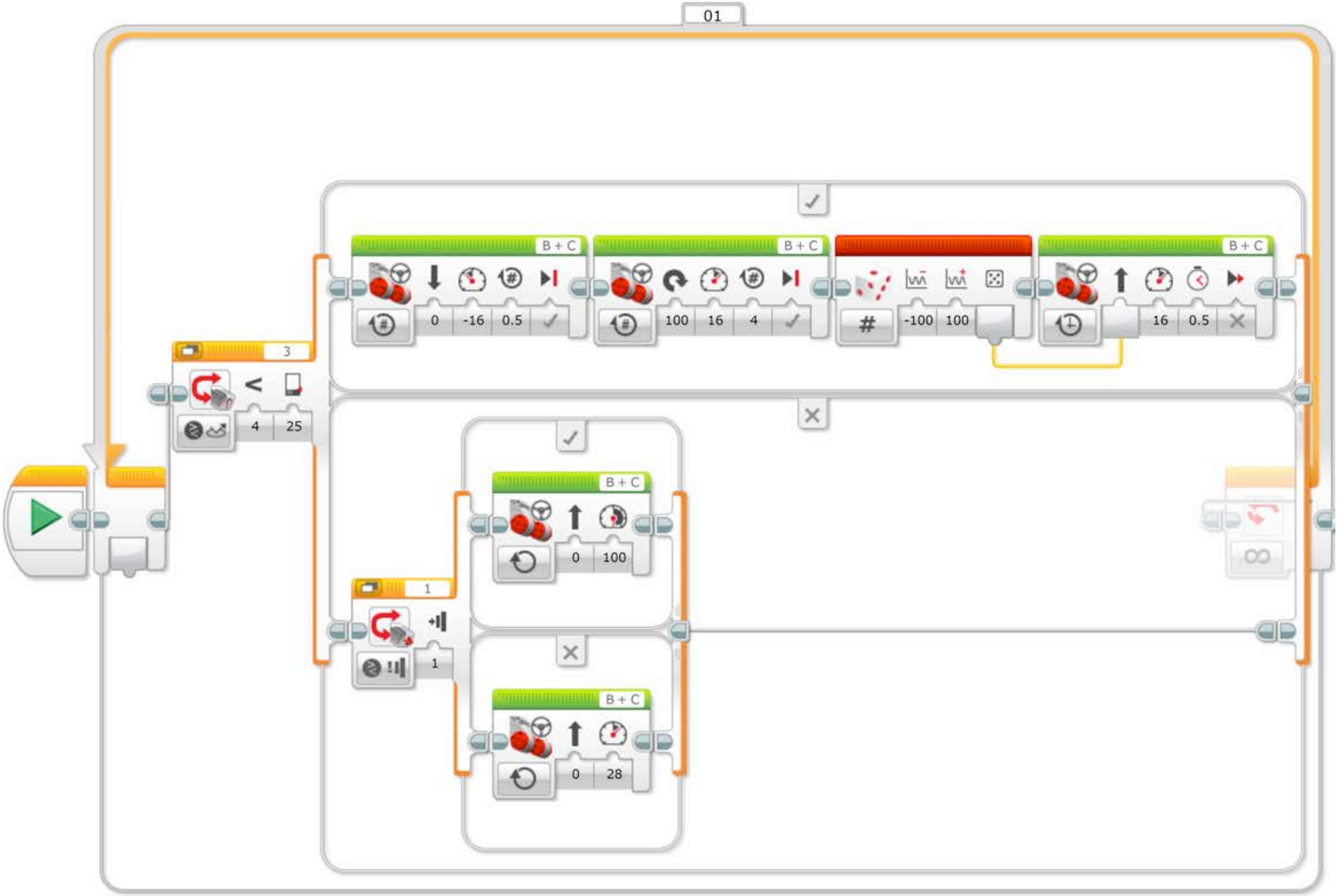
- LeJOS, ...

Using decision trees; used in my HUMTEK course

- Example: A cleaning robot



Programming it with Lego Mindstorm's in-the-box graphical programming language



Behaviour-based robotic in LeJOS

Comments on the names

- **`lejos.robotics.subsumption` is *no*t subsumption but simple, priority-based scheduling**

- **the class `Arbitrator` is *not really an arbitrator*, but a choice mechanism of which behaviour is selected (and is allowed to compute)**

Defining a behaviour in LeJOS

```
class MyNewBehaviour implements Behav

  ....

  public boolean takeControl() {...}

  public void action(} {...}

  public void suppress() {...}

}
```

must be true for this B. to be selected

what this B. is supposed to be doing

what this B. needs to do to clean up when stopped

The more precise version: based on threads

Consider that *B1* is running and *B2* takes over:
B2`.takeControl() == true`, and system has decided for B2

- *B1* considered <u>current</u>: Either *B1*`.action()` running in its own thread *T1* <u>or</u> has terminated

- System calls *B1*`.suppress()`, which *should be* programmed in such a way that a possible *T1* is terminated as well as other threads started, perhaps indirectly, by it.

- System starts new thread *T2* in which *B2*`.action()` is started; now *B2* is considered current

Putting several behaviours together

```
Behaviour b1 = new MyBehaviour1(...);
Behaviour b2 = new MyBehaviour2(...);
...
Behaviour bn = new MyBehaviourn(...);

Behavior [] allBehavs = {b1,b2,...,bn};

Arbitrator arbit = new Arbitrator(allBehavs);

arbit.go();    // and not .start() as shown in the book (sic!)
```

Overall control is a thread running a loop:

- check **b***i***.takeControl()** in the order *i*=*n*, *n*-1, ..., 1 until a **b***k* is found
- If **b***k* is different from current, let it be the new current and do as in the previous slide

Good practices

- Program your behaviours so the they can be understood independently of any other behaviour
  - Perhaps relaxed: defining your own protocal for how each behaviour should expect the state it takes over and the state it leaves behind
- **`suppress()`** should do its job in an instant (i.e., no loops!!), so that an urgent action of higher priority can take over immediately
  - Perhaps relaxed: if this behaviour is known to have highest priority off all, **`suppress()`** may be allowed to consume some time
- Each behaviour should be programmed with no assumption of its priority
  - Perhaps relaxed: if this behaviour is known to always to have highest or loweste priority off all, you may (need to take) that into account

Program example

See course note and sample files on moodle

- A robot that drives around and uses an infrared sensor for avoiding obstacles and a touch sensor for stopping
- Infrared sensor wrapped as a high-level sensor, accessed by a public variable, e.g.:

```
irAdapter.objectDistance < 25
```

- Three behaviours

```
BehaviourForward
BehaviourAvoidObject
BehaviourStopByTouch
```

- Course note discusses "good practice" and shows different variations of BehaviourForward.java
- We will show a few details + add an introduction to motor synchronication

Main method

```
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();
    }
}
```

A closer look at `BehaviourForward`

```
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() { }

}
```

A closer look at BehaviourForward_3

```
public class BehaviourForward implements Behavior {
    ....
    private boolean suppressed = false;

    public boolean takeControl() {return true; }


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


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

Final detail: having motors to start and stop at the same time

```
public class BehaviourForward implements Behavior {
    ....
        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();
    }
}
```

Conclusion

- Behaviour.based programming in LeJOS
    - based on a simple priority-based principle
    - fairly easy to use if you remember things are running in thraeds!!!!
- Drawback: Only one behaviour at a time
    - later we may consider how to relax this