All you need is love?

1 Complexish behaviours from simple systems

In these exercises we'll create several programs in the spirit of Valentino Braitenberg's *Vehicles*. We are looking to create the kinds of behaviours which saw in the lecture on Monday, such as Fear, Caution, Love, Distaste... and of course anything else you might find exciting.

2 Main

We have provided a small occam framework for you to work within. It provides the `main PROC` for your robots, and a bit of code to get you started with the implementation of Caution. You should open this file:

- Start jEdit (double click the jEdit icon)
- Open the diku.occ file (`File` → `Open`, navigate to diku folder, select diku.occ)

To compile your programs, you can either skim the "quickstart guide" we have on the VM (click the "Documentation" icon on the desktop), or you can do the following:

- Go to `Plugins` → `occPlug` → `Start occPlug`
- Click the swoopy circular thing to compile programs, and the running dude to execute them.

You will need to launch a simulation world (icons on the bottom of the desktop) before executing your occam program, however.

2.1 A Picture is Worth a Thousand Words

When writing occam programs, diagrams can be extremely useful. There is a very direct mapping from a diagram describing an occam network, to the code that implements it. Figure 1 depicts the `main PROC` which has been provided in the starter file.

We can think of `main` as a black box with some wires. These wires allow the box to communicate with the external world. In the case of `main`, this

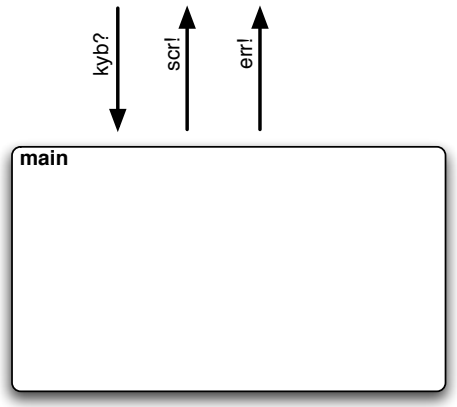

Figure 1: The Main Process

world is your keyboard and the screen output on your terminal (as it turns out, we are not going to be using these channels in the exercises presented here).

Lets open up `main` and have a look at what's inside (Figure 2). We have now taken the lid off `main`, and we can see that `main` contains other processes. The `brain.stem.ML` process is what allows us to interface with a robot running in the simulator, and as it comes from a library, we are not going to open it up and look inside. Looking at`brain.stem.ML`, we can see that it can communicate with the world using two wires, one for laser data and one for motor commands.

In in Figure 2 we can see that it is the `caution` process which receives the laser data, as it is connected to the `brain.stem.ML` process using the `sick` channel. It is also connected to the `caution` process which provides motor commands back to the `brain.stem.ML` process, probably in reponse to the laser data it receives.

2.2 A Picture is Worth a Lot of Code

Lets look at how the diagram translates into code. The outer box labeled `main`, which has three channels sticking out of it (which we happen to know are of type `BYTE`) become the following line of code (also known as a `PROC` header): `PROC main(CHAN BYTE keyboard?, screen!, error!)`. The funny ! and ? you see in the channel declerations specify that the main process can only read(?) from the keyboard channel, and

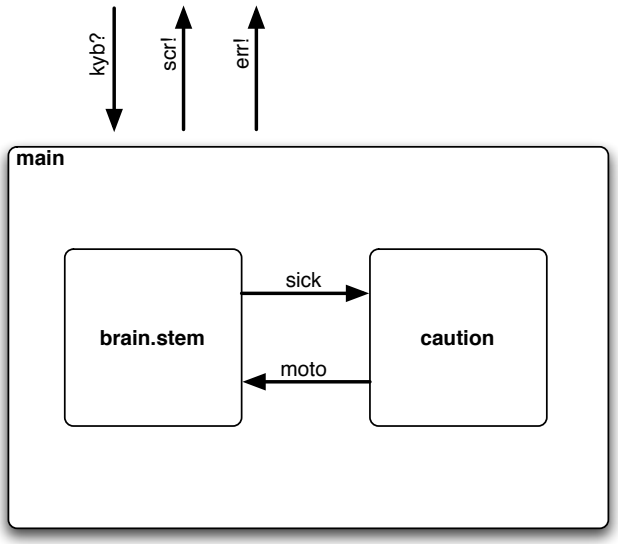

Figure 2: Main, opened up

write(!) to the screen and error channel.

Inside the `main` box, we have two processes communicating with each other, using the `sick` and `moto` wires on the diagram. These become channels in occam: `CHAN LASER sick:` and `CHAN MOTORS moto:`.

When we draw processes inside a box on a diagram, this means that the containing box is going to start those processes up and run them in parallel. occam uses the `PAR` construct for this:

```
PAR
  brain.stem.ML(moto?, sick!)
  caution(moto!, sick?)
```

The wires in the diagrams we use are directional, this means that each process has a particular and of a channel, either the *output* end, or the *input* end. `caution`, for example, has the *output* end (denoted by the !) of the 'moto' channel (`moto!`), and the input end (denoted by the ?) of the 'sick' channel (`sick?`). `brain.stem.ML` has the other end of these channels: the input end of moto (`moto?`) and the output end of sick (`sick!`).

We have now translated all the components of the 'main' diagram into occam code, and the only thing missing is ending the definition of PROC 'main' with a colon (`:`).

The full PROC becomes:

```
1 PROC main(CHAN BYTE keyboard?, screen!, error!)
2   CHAN LASER sick:
3   CHAN MOTOR moto:
4   PAR
5     brain.stem.ML(moto?, sick!)
6     caution(moto!, sick?)
7 :
```

Notice that the indentation (2 spaces) is important, as it denotes scope. The scope of the PAR for example, is everything indented two spaces from it.

3 Caution

Figure 3 depicts the internals of the `caution` process. When writing code in a CSP-based language, you generally want to start with a diagram, and from that diagram declare the channels and wire up the processes in your diagram.

Caution only requires library procedures connected together by channels of the right types. While we haven't explicitly given you the types in this diagram, you should be able to figure them out from the headers of the `PROC`esses involved (documented later).

4 Fear

We have not given you a `fear` diagram; draw one, using the library procedures defined in this document. Then, write the code (process header, channel declarations, and several library processes running in parallel) to actually create this network.

5 Lurve

"Love" is a more complex emotion than either "caution" or "fear". That is why we have Valentines Day, and why so many of us are single, and can't seem to just, you know, get together and be excellent to each-other.

...

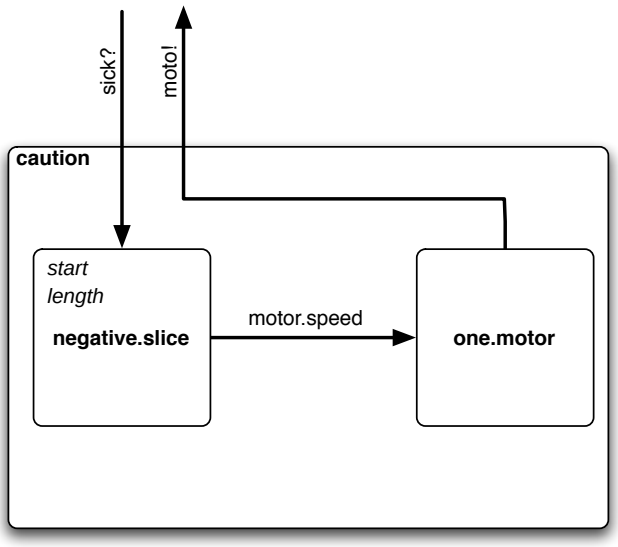

Figure 3: Process network for Caution

Right. Anyway. Figure 4 details the inner workings of the `love` process; using it, you should be able to implement love in your robot.

6 Distaste

We didn't want to use the word *hate* as the opposite of *love*, so we chose "distaste." The network should be similar; draw a network that you believe will cause your robot to wander away from things, instead of towards them. From this, then implement the code to connect things up.

7 Additional ideas

There are a number of other things you might explore:

- A robot that is exhibiting `distaste` has a hard time in some situations; what can you do to make it more robust in wandering around some of the more complex spaces?
- Can you use a replicated `PAR` (see the occam tutorial linked in from the RoboDeb site) to run more than one robot? Can you run one

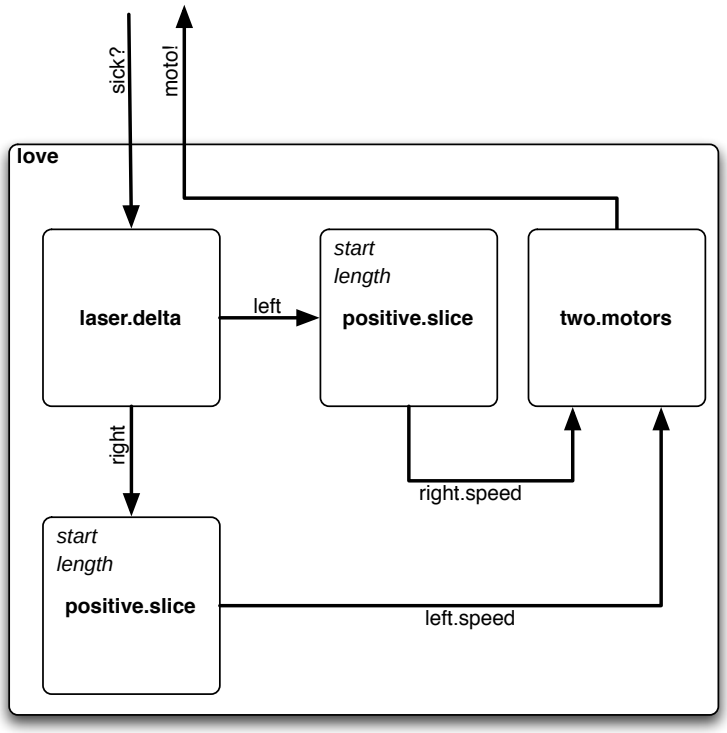

Figure 4: Process network for Love

robot that is in "love" and one that is in a state of "fear," or "distaste"?

- We depicted a bump-and-wander bot in the lecture; can you write a wall follower? Can you do it with a minimum of sensor input?
- Can you build up some kind of internal representation of a map?

There's much more that is possible with the environment; if you're interested in finding out when we upload the image to our website (probably by the end of this week), drop an email to matt@transterpreter.org and I'll add you to our announcement list.

8 More

(The previous section was written after this one; so, there's some repetition. My bad... —MCJ)

You should now have some idea of how to wire up simple behaviours for robots using occam. If you want to explore some more complex behaviours, which require more than just wiring up existing processes, have a look at the RoboDeb website, http://robodeb.transterpreter.org/. Here you will find a further tutorial on occam and robotics (the Interesting Bump and Wander), an occam primer, and various other bits of useful information and documentation.

9 A quick reference to the vehicles library

9.1 positive.slice

```
PROC positive.slice(CHAN LASER sick?, CHAN INT motor.speed!,
VAL INT start, length)
```

The closer the robot is to something in the slice, the faster the motor goes. The start and length parameters allow you to define the start and length of the slice that this process watches. To watch the entire range of the laser, use 0 for the start, and 180 for the length. To watch right 90 degree quadrant, use 90 for the start, and 90 for the length.

9.2 negative.slice

```
PROC negative.slice(CHAN LASER sick?, CHAN INT motor.speed!,
VAL INT start, length)
```

The closer the robot is to something in the slice, the slower the motor goes.
The start and length parameters work the same as for positive.slice.

9.3 laser.delta

```
PROC laser.delta(CHAN LASER sick.in?, sick.out1!, sick.out2!)
```

This process takes laser data on sick.in, and produces two identical copies
on sick.out1, and sick.out2.

9.4 one.motor

```
PROC one.motor(CHAN INT motor.speed?, CHAN MOTOR motor.command!)
```

This PROC turns integer motor speeds (in the range -300 to 300) being sent
to it, into commands which a brain.stem can understand.

9.5 two.motor

```
PROC two.motors(CHAN INT motor.speed.left?, motor.speed.right?,
CHAN MOTOR motor.command!)
```

This PROC turns integer motor speeds (in the range -300 to 300) being sent
to it, into commands which a brain.stem can understand.