Gürtler Fabienne 12-935-508

Robotics Series 4

Exercise 1. State Machine: advanced lover and explorer behavior revisited

a)
```
if counter == 200 then
    emit execAdvLover
elseif counter == 400 then
    emit stop
elseif counter == 600 then
    emit execExplorer
elseif counter == 800 then
    emit stop
end
```

The e-puck emits events every 200 steps. It has a counter that counts from 0 to 800, is then reseted to 0 and starts anew. The other e-puck is the state machine receiving the events. This state machine has three states: *STOP, ADVANCED_LOVER, EXPLORER* and three events: *stop, execAdvLover, execExplorer.* Once it receives an event it goes into the corresponding state. It starts in the idle *STOP* state but when it receives the *execAdvLover* event it goes to the *ADVANCED_LOVER* state. After 200 steps it receives the *stop* event and thus goes back to the *STOP* state. It receives events in the following order: *execAdvLover → stop → execExplorer → stop.* And then the cycle repeats.

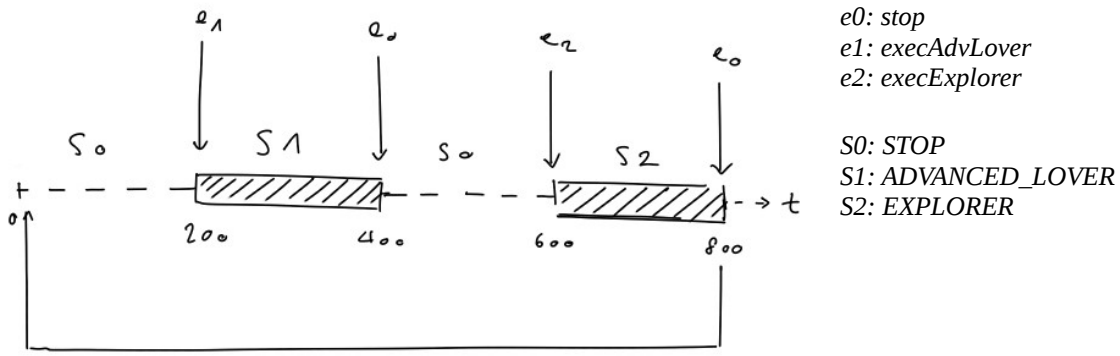

Illustration 1: Events

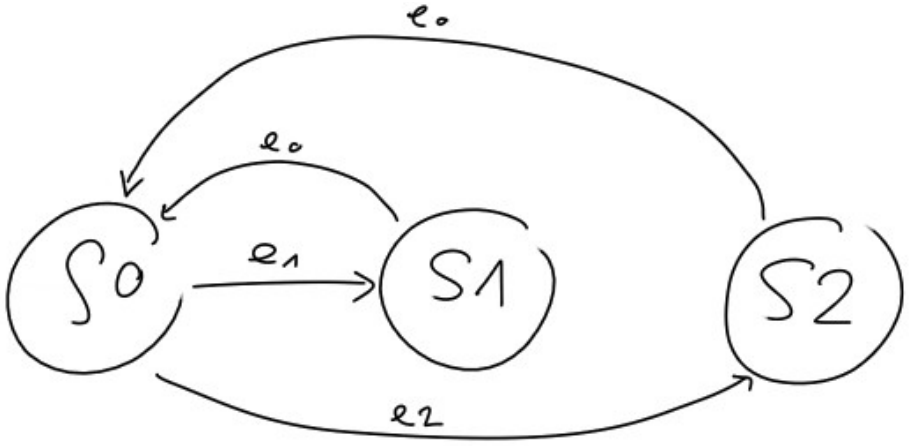

Illustration 2: State machine

b) Interestingly, with this code it worked when the state *STOP* was removed. The resulting state machine looked like this:

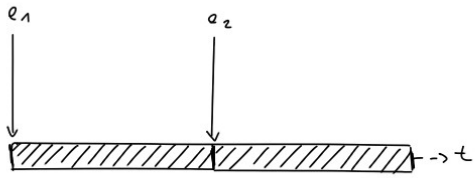

Illustration 3: Reduced Events

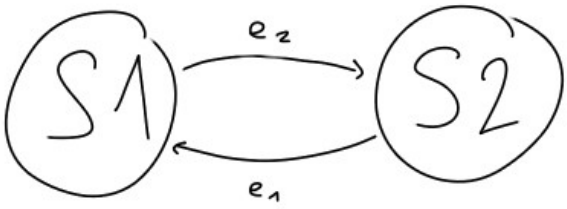

Illustration 4: Reduced state machine

Even tough it works designing a state machine like this should be avoided because there is no guarantee it actually will work as intended. If the e-puck is in an active state – executing a code - and receives an event he will immediately react to this new event without finishing the code he was working on. For example: the e-puck is supposed to calculate some variables in state A and then use them in state B but if he is interrupted in state A he might not yet have calculated the correct values for the variables. This unsafe behavior might lead to an incorrect behavior. In this case, because both active states calculate their values separately he does not have that problem. But generally one should not interrupt states because it is hard to predict what is going to happen. In the explorer and the advanced lover state the event gets interrupted in either case, because they both run infinitely. So either they get interrupted by each other (Illustration 3) or by the *stop* event (Illustration 1).
There is an other problem that will cause an error. If two events are send out immediately after one another, the e-puck receives them at virtually the same time. In the simulation this results in an *Event Execution killed* error and the e-puck gets stuck in the second event. In the area Aseba Studio did not display an error message, but the e-puck also got stuck.

Note: For this exercise we used the solutions of series 3 for the implementations of the advanced love and the explorer behavior. It is shorter and simpler than the one we came up with.

Gürtler Fabienne 12-935-508

Exercise 2

a)

Emmiting events:

```
# Safe behavior
emit go(SPEED_NORM)
for i in 0:DELAY_SAFE do end
emit turn
```

```
# Unsafe behavior:
emit go(SPEED_NORM)
for i in 0:DELAY_UNSAFE do end
emit turn
```

Implementation of events

```
onevent  go
    for counter in 0:DELAY do
        speed.left = args[0]
        speed.right = args[0]
    end
    speed.left = 0
    speed.right = 0
```

```
onevent turn
    for counter in 0:DELAY do
        speed.left = args[0]
        speed.right = -args[0]
    end
    speed.left = 0
    speed.right = 0
```

The events *go* needs a little over DELAY=5000 to complete movement: t(go)= 5000. If next event is sent before that it produces an *Event Execution killed* error. When t(go) > t(turn) it is unsafe behavior and when t(go) < t(turn) it is safe behavior. So *DELAY_SAFE* need to be over 5000, I set it to 8000. *DELAY_UNSAFE* = 4000 < *DELAY = 5000* is unsafe.

b) To ensure that the execution of a code associated with an incoming event is delayed until no other code is executed, the various events would have to “communicate”. In Java there are threads which can have different states:

NEW: thread that has not yet started.
RUNNABLE: thread executing in the Java VM is in this state.
BLOCKED: thread that is blocked waiting for a monitor lock is in this state.
WAITING: thread waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING: like waiting, but only waits for specified waiting time.
TERMINATED: thread that has exited.

There are different ways to handle threads in Java. For example you can make a thread wait for an other thread to finish with the *join()* method. There is also the *wait()* and *notify()* concept which would be useful. You could tell the first e-puck to wait until the second e-puck sends a signal to notify him that he is now done with the code and ready for new instructions. There could still be problems, like with infinite loops. If first event is an infinite loop, and e-puck is told to wait with next event until the first one is finished, it never executes the second code. It may also be useful to have an event queue, that stores all the events and when it gets notified that the event was completed, it takes the next event from the queue and so on. This is all theoretical but I am sure that there are many ways to handle the scheduling of events.