Using Functional Reactive Programming for Robotic Art

Graduate

Eliane Irène Schmidli

Introduction: Usually, the control software for robotics applications is written in a low-level imperative style, as shown in Fig. 1. This method intertwines the program sequence with commands for motors and sensors. To describe the program's behavior, it is typically divided into different states, each representing a specific system condition. It can be challenging to recognize the points at which state transitions are triggered. This way of programming complicates the comprehension of the code, making changes to the program flow a tedious task.

Functional Reactive Programming (FRP) offers a composable, modular approach for developing reactive applications. To compare FRP with the conventional imperative style, the control software for the robotic artwork Edge Beings by Pors & Rao was developed. The design uses Yampa, an FRP implementation in Haskell.

Approach: The artwork Edge Beings involves a motion sensor and motors, enabling figures to crawl over the edges of panels (see Fig. 3). The beings' behavior varies based on the viewer's proximity to the artwork. If the viewer is too close, the beings only peek over the edge. Otherwise, social dynamics can be observed between them. The beings belong to distinct groups, with only one allowed to emerge completely at a time. Each group has a suppressor that can drive away weaker groups, making room for its own group.

In Yampa, the problem can be implemented similarly to the described scenario (see Fig. 2). The 'behavior' function switches from social to disturbed behavior when a motion event is registered by the sensor and reverts to social behavior when no motion is detected. The 'social' function defines the behavior corresponding to each being's role. For a suppressor the 'suppressor' function initiates 'creepOut' if it outranks the current group, allowing the suppressor to fully emerge. This triggers the 'currentGroupEv' event for the suppressor's group, enabling it to emerge as well.

Instead of communicating directly with the motors, the FRP approach produces a signal indicating the current positions of the beings during runtime. This signal can be consumed by the motors that move to the corresponding position. This decouples the behavior of the beings from the control of motors.

Conclusion: The design allows to modify the behavior without adapting the hardware control. In addition, the produced position signal can be used for a simulation of the artwork. This enables users to visually observe the effects of modifications in the program without access to the physical hardware.

Furthermore, the Yampa implementation presents state transitions more clearly, resulting in a more understandable code, especially for people with little programming experience. Nevertheless, getting started with FRP can be challenging as there are distinct definitions of FRP and frameworks based on different concepts.

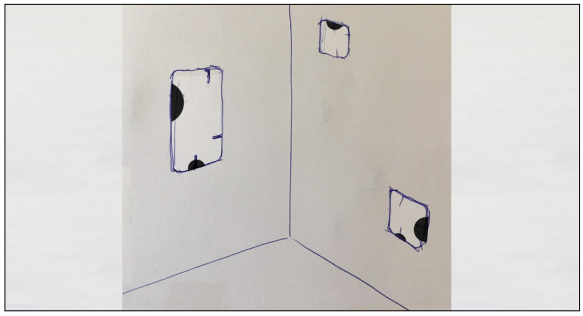
Figure 1: Implementation of the behavior of Edge Beings using imperative style (simplified)
Own presentment

```python
def step():

    if state == State.HIDE:
        motor.go_to_position(Pos.HIDE)
        state = State.PEEK

    elif state == State.PEEK:
        motor.go_to(Pos.PEEK)
        if (not motion and is_stronger_suppressor):
            suppressor_events.insert(rank)
            state = State.STAND
        elif (group_is_allowed_to_go_out):
            state = State.STAND
        else:
            state = State.HIDE

    elif state == State.STAND:
            motor.go_to(Pos.STAND)
            state = State.HIDE

def event_handling():
   # Sets states depending on events
```

Figure 2: Implementation of the behavior of Edge Beings using FRP (simplified)
Own presentment

```haskell
check    :: SF Input Velocity -- hide -> peek -> hide
creepOut :: SF Input Velocity -- hide -> peek -> stand -> hide

suppressor :: SF Input Velocity
suppressor = check `doUntil` higherRankEv
    `switch` \_ -> (creepOut `doUntil` notHigherRankEv)
    `switch` const suppressor

being :: SF Input Velocity
being = check `doUntil` currentGroupEv
    `switch` \_ -> (creepOut `doUntil` notCurrentGroupEv)
    `switch` const being

social :: SF Input [Position]-- starts 'being' and 'suppressor' depending on the beings' role,
                             -- calculates position from velocity

disturbed :: SF Input [Position] -- runs 'check' for all, calculates position from velocity

behavior :: SF Input [Position]
behavior = social `doUntil`` motionEv
    `switch` \_ -> disturbed `doUntil`` noMotionEv
    `switch` const behavior
```

Figure 3: Sketch of the artwork Edge Beings
Source: Pors & Rao Studio



Advisor
Prof. Dr. Farhad D. Mehta

Co-Examiner
Dr. Joachim Breitner

Subject Area
Computer Science, Software and Systems

Project Partner
Pors & Rao Studio, Bangalore, India