Using Functional Reactive Programming in Yampa to Redesign the Control Software for the Robotic Artwork "Islanders"

Eliane I. Schmidli
OST – Eastern Switzerland University of Applied Sciences
MSE Projects 1 & 2
Supervisor: Prof. Dr. Farhad Mehta
Spring 2023

Abstract

The Islanders artwork by Pors & Rao is a robot application reacting to the sounds in the room and controlling actuators. The control software is written in a low-level imperative style, intertwining the program sequence and commands for the actuators. Furthermore, the program code describes the behavior of the program depending on the current state of the actuators. It is difficult to comprehend where the state transitions are initiated. This way of programming complicates the comprehension of the code making modifications in the program sequence arduous. Functional Reactive Programming (FRP) is a composable, modular way to program reactive applications. With FRP, the code was redesigned, making it more customizable, especially for people with little programming experience. The redesign uses Yampa, an FRP implementation in Haskell using Arrows as the basic structuring framework. The resulting design is very promising. The state transitions clearly show which event triggers which reaction, making the code much more understandable. The new design separates the control of the hardware from the implementation of the program flow. This makes it possible to use the same code with different peripheral devices.

1 Introduction

The Islanders[1] artwork was created by the artist duo Pors & Rao [Rao20]. It is a composition of different-sized panels mounted on three walls. When the room becomes quiet, black silhouettes of tiny figures slowly start to creep out from behind the panels. First, they peep over the edge, and after a while, they move further out. A loud noise in the room causes the creatures to retreat and hide behind the panel again. At a soft sound, a few creatures do not hide completely but still peek over the edge [Rao06].

The movements of the figures appear life-like due to a behavioral algorithm. A random factor makes the islanders move differently from each other. Depending on the sound volume, the figures move faster or slower. If the sound level in the room is high, the figures are more likely to hide completely and stay hidden longer.

The current control software is written in a state machine style. Depending on the state of the islanders, different instructions are sent to the actuators. But it is difficult to understand where in the code the state changes are triggered. This makes changes in the program sequence error-prone. Section 3.1 describes the original implementation of the control software in more detail.

FRP is a composable, modular way to code reactive applications. With the help of FRP, the control software was redesigned leading to better understandable code. Instead of sending commands directly to the actuators, in FRP a signal is created that indicates the current positions of the islanders. This can then be interpreted by various outputs. In the real artwork, commands can be transmitted to the actuators to move the islanders to the current values of the position signal. The same signal can also be used for a simulation where the Graphical User Interface (GUI) places the islanders at their current positions. A change of position then has the effect of an animation.

Section 2 provides a first introduction to the concept of FRP. The new design of the control software is presented in section 3. Section 4 provides some implementation details and highlights the strengths and weaknesses of using the FRP concept.

2 FRP and Yampa

This section introduces the main concepts of FRP and Yampa [HCNP03] that are relevant to understand this paper.

2.1 FRP

FRP is the intersection of functional programming and reactive programming. Reactive programming means that a program is event-based and acts in response to its input. Unlike the imperative style, FRP proposes a declarative approach to program reactive applications.

A simplified example of how to think of FRP is a spreadsheet. A spreadsheet consists of cells that can

[1]The artwork is presented in the talk: "High-tech art (with a sense of humor)" by Aparna Rao: https://youtu.be/kJLDl2uDNaA?t=111

contain values or functions. For example, a function can check whether the value in a particular cell is larger than the value in another particular cell. When programming the function, the user describes the dependency of the result on the two cells (`result = cell1 > cell2`). If the value in a cell is changed, the result is automatically updated. The refreshing is done by the spreadsheet program. In imperative programming, the refreshing of the values has to be specified by the programmer [BJ16, Chapter 1].

In 1997, the concept of FRP was implemented by Conal Elliott and Paul Hudak in Fran. Fran is a domain-specific language (DSL) for animations in Haskell. Fran makes it possible to separate the presentation of animations from the description of the animations [EH97].

Two years later, the FRP concepts were applied to Frob, a DSL for use in robotic systems. Frob hides the details of low-level robot programming and makes programming more hardware independent. Unlike Fran, Frob must also manage hardware control, which adds an additional layer of complexity [PHE99].

The following sentence from [PHE99] makes this difference clear.

> "For one thing, an animated figure will always do what you ask; but a robot will not!"

2.2 Yampa

Yampa is an FRP implementation inspired by Fran and Frob. Yampa is more application-independent and is used in various areas such as robotics, GUI applications, and games [HCNP03]. For instance, Pembeci, Nilsson, and Hager [PNH02] have built vision-guided, semi-autonomous robots with FRP using Yampa.

2.2.1 Signals and Events The most important concepts of FRP in Yampa are `Signals` and `Events`. A `Signal` is a continuous, time-varying value. It can be understood as a mapping of a value of type `Time` to a value of type `a`: `Time -> a`

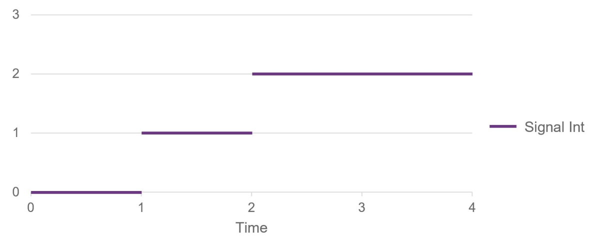

Figure 1. An example of an integer `Signal` that changes its value over time.

Figure 1 visualizes an example of an integer `Signal`. During runtime, a `Signal` always contains a value. The program code defines at which event the value of a `Signal` changes and how. A `Signal` automatically keeps its value up to date.

Events in Yampa (for example, when a mouse click happens) have a similar structure to Haskell's Maybe data type. An event stream can be understood as a `Signal (Event b)`, which is a `Signal` that yields either nothing or an `Event` with a value of type `b`. The value of type `b` is generated each time the event occurs. For example, the current position of the mouse can be attached to the event. A visualization of an event stream `Signal (Event Int)` can be found in Figure 2.

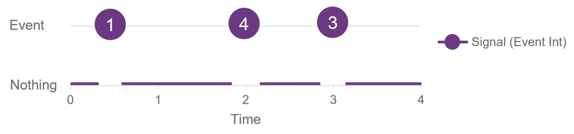

Figure 2. An example of a `Signal` that yields nothing or an `Event` containing a value of type `Int`.

The concept of `Signal` enables the writing of programs that have time[2] and space leaks[3]. The reason for this is beyond the scope of this paper. Yampa solves this problem by not allowing Signals as first-class values. The value contained in a `Signal` can only be modified using signal functions. These cannot be constructed directly, but only using a set of given combinators. The given combinators ensure that time and space leaks are avoided. In Yampa, this concept was implemented with the help of arrows, a generalization of monads proposed by John Hughes [Hug00].

A signal function `SF a b` is an instance of the `Arrow` class and can be thought of as a mapping of Signals to Signals: `Signal a -> Signal b`.

2.2.2 Example To better understand how signal functions work, this section provides a small example. It creates an animation of a straightforward motion of an object. The animation can be started and controlled using mouse clicks. Depending if the left or the right button is clicked, the current speed is slow or fast. The result is a signal that represents the position. Its value should be zero at the beginning and after a mouse click, increase with the corresponding speed. The resulting signal can be interpreted by a GUI to create an animation of an arbitrary object as shown in Figure 3.

[2] [HCNP03] includes a definition of a time leak in a real-time system.

[3] The section "Introduction" of [Kri13] provides a good example of how a space leak can occur.

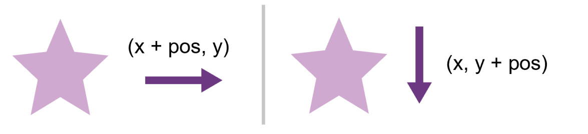

Figure 3. Animation of a star depending on the position `pos`. The direction is determined by the definition of the coordinates in the GUI. E.g. the coordinates on the left correspond to a movement along the x-axis, whereas those on the right result in a movement along the y-axis.

2.2.2.1 Creation of a Position Signal

The example starts with the creation of a signal function with `constant` to represent the velocity. It is of type `constant :: b -> SF a b` and creates a signal function providing an output signal with a constant value.

```haskell
type Velocity = Double

velocity :: Velocity -> SF () Velocity
velocity v = constant v
```

To calculate the position based on a constant velocity, the value of the input signal must be integrated with the signal function `integral` (see Figure 4).

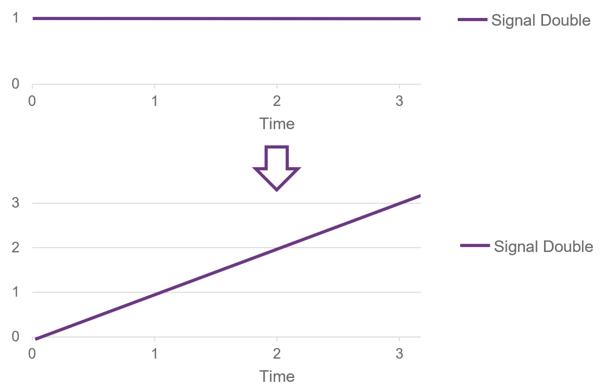

Figure 4. Visualization of `integral` accumulating a constant input value over time.

To pass the output signal of `velocity` to the `integral` signal function, as in Figure 5, a combinator is needed. One possibility to link a signal function with another is the operator `>>>`.

```haskell
(>>>) :: SF a b -> SF b c -> SF a c
```

Now the function `positionSF` can be defined that will increase the value of the resulting signal steadily. This can be interpreted as a movement with a constant speed `v`.

```haskell
type Position = Double

positionSF :: Velocity -> SF () Position
positionSF v = velocity v >>> integral
```

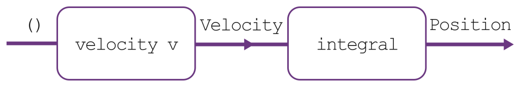

Figure 5. Visualization of `positionSF`. The arrow represents the signal that is modified by the signal functions `velocity` and `integral`.

2.2.2.2 Start on Mouse Click

Now the movement should start when a mouse click occurs. So, the velocity should first be zero, and after the mouse click set to the slow speed (here ten). For this, the function `hold` is needed. This function receives a start value as a parameter and creates a signal function from it. This signal function takes an event stream as input and produces an output signal. In the beginning, the output signal holds the start value. When an event occurs, the value is replaced by the value of the event.

```haskell
hold :: a -> SF (Event a) a
```

The output signal of `hold` should indicate the new speed after the click. For this, the velocity must be attached to the event using the function `tag`.

```haskell
tag :: (Event a) -> b -> (Event b)
```

Figure 6 visualizes the transformation of the event stream to the position signal. First, the input signal yields nothing and `hold` produces a signal containing the value zero. Therefore, the output signal of `integral` also indicates zero. When a mouse click event occurs, ten is attached to it, and `hold` changes the value accordingly to ten. After the change, `integral` starts increasing the value of the position.

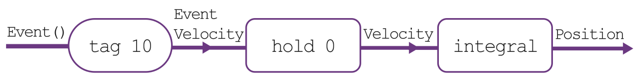

Figure 6. Visualization of `positionSF` transforming the event stream to the position signal. Note that `tag` is not a signal function.

The new version of `positionSF` receives the mouse event of type `Event()` as an input signal. To apply `tag`, the event has to be extracted from the signal. This is possible in the arrow notation. The structure is similar to the do notation as an alternative monad syntax and looks like this:

```
proc input -> do
    result1 <- signalFunction1 -< input1
    result2 <- signalFunction2 -< input2
    returnA -< result
```

The new version of positionSF in arrow notation looks like this:

```
positionSF' :: SF (Event ()) Position
positionSF' = proc click -> do
    v <- hold 0 -< click `tag` 10
    pos <- integral -< v
    returnA -< pos
```

The extracted value of the input signal is click of type Event(). The event is tagged with ten and is passed to the signal function (hold 0). This produces an output signal containing the current velocity. The velocity v can be passed to the function integral which produces the output position pos. With returnA the value pos is wrapped in the outgoing signal.

2.2.2.3 Switch between Speeds To switch between two different speeds depending on the left and right click of the mouse, two event streams must be used. This can be realized by changing the input signal of programSF' to the type (Event(), Event()). Then a different speed is tagged to each of the two click events. To use hold, both event streams have to be merged into one. For this, we have to decide which of the two events has priority when both occur simultaneously. With rMerge, the right event is always preferred[4] as in Figure 7.

```
rMerge :: (Event a) -> (Event a) -> (Event a)
```

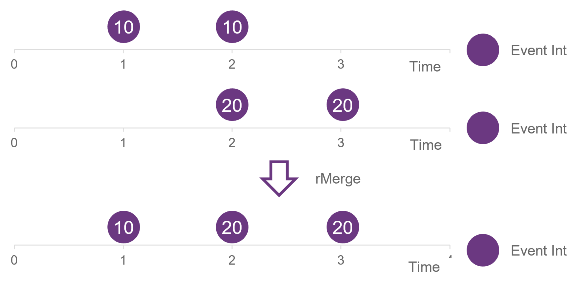

Figure 7. Visualization of rMerge. It merges the two event streams preferring the events from the second stream.

Here is the new implementation of positionSF', where lbp corresponds to a left mouse click, and rbp to a right mouse click. Each time the left mouse button is pressed, v corresponds to the value 10, and when the right button is pressed to the value 20. The position is increased accordingly faster or slower.

[4]Note that there are different versions of merges: https://hackage.haskell.org/package/Yampa-0.14/docs/FRP-Yampa.html#g:10

```
positionSF'' :: SF (Event (), Event()) Position
positionSF'' = proc (lbp, rbp) -> do
    v <- hold 0 -< lbp `tag` 10
                  `rMerge` rbp `tag` 20
    pos <- integral -< v
  returnA -< pos
```

3 Design Concept

The core of the control software is the description of the behavior of the islander. This section covers how the code of the current software could be implemented in Yampa.

3.1 Current Implementation

The current code of the Islanders artwork is written in imperative style. The program is running in a loop and at every iteration, the control software of every actuator is executed. It depends on the current state of the actuator instance which instructions will be executed.

In this code section, a simplified version of such an actuator control program is shown. In the hide state, the actuator moves the islander to the hiding position. In the go_peeking state, the islander waits for a specified time and then starts to creep out to the peeking position. In the go_standing state, the islander waits again and then moves to the standing position. In the 'wait' state, nothing happens.

```
def program(state):
  switch state

    case 'hide':
       actuator.move(0, hidingV)
       waitUntil = currentTime + waitingTime
       state = 'go_peeking'

    case 'go_peeking':
       if currentTime > waitUntil:
          actuator.move(peekPos, peekV)
          state = 'go_standing'
          waitUntil = currentTime + waitingTime

    case 'go_standing':
       if currentTime > waitUntil:
          actuator.move(standPos, standV)
          state = 'wait'

    case 'wait':
       # do nothing
       pass
```

This program describes the behavior of an islander and at the same time passes the instructions to the actuator. It is well-visible what happens in each state. However, it is unclear when and where a state is set. For example, hide is set outside the program function. When making changes in the program flow, it must first be exactly understood where these transitions occur to make changes in the correct places. Therefore, all occurrences of the state name have to be searched in the project.

3.2 New Design in FRP

In FRP it is possible to separate the behavior of the islander and the control of the actuators or the animations. The behavior can be implemented in a function producing a signal that indicates the current position of each islander. The function changes the value of this signal according to the input signal providing the sound level in the room. For example, if the noise exceeds a certain level, the islanders hide at position zero, and if it is quiet, they move out and the position is increased accordingly. Each islander has its own motion parameters, thus the positions of the islanders in the output signal will differ from each other.

The output signal containing the islanders' positions can be consumed by actuators. The actuators always move to the current position in the signal. At the value zero, the islander is hidden and the actuator is fully retracted. At the maximum position value, the islander is standing and the motor is fully extended.

The control of the microphone in the artwork produces events when certain sound levels are exceeded. In the new design, a sound signal is used that displays the values zero to two and changes depending on the events from the microphone. When it is quiet, the signal shows zero and the islanders start to crawl out. When there is a small noise, the signal changes the value to one and the islanders move backward and hide. A few islanders will still peek over the edge. If there is a loud noise, the value is set to two and all islanders hide.

This structure enables the separation of the islanders' behavior from the representation in the output. Changes to the behavior of the islanders are therefore automatically visible without modifying the control of the actuators. Furthermore, the actuators can be replaced with another output device. So, a simulation of the artwork was created using the keyboard as input and a GUI as output. The behavior of the islanders is defined with the same code as the actuators use. Figure 8 shows a screenshot of the GUI from the resulting application.

In the simulation, the events from the microphone are imitated with the keyboard number keys zero to two. The output signal is passed to the GUI that always draws the islander at the current position. Depending on which side of the panel the islander is located, the GUI changes the coordinates of the x or y axis based on the position from the output signal. So, the islander moves along the corresponding axis. If an islander is stationary, its position remains at the same value and the coordinates do not change.

The following sections further describe the implementation of the behavior of the islanders. At the end of section 4, an excerpt from the islanders code is provided.

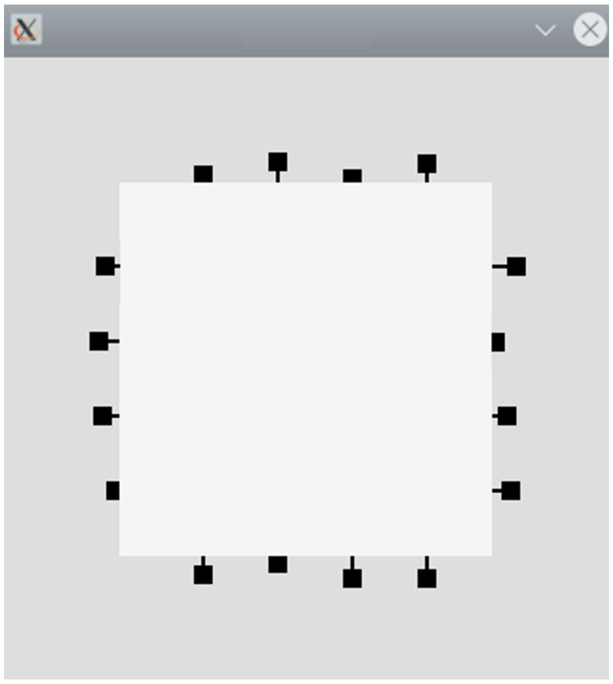

Figure 8. GUI of the Islanders Simulation

3.2.1 Execution

In order to execute the function defining the behavior indefinitely, Yampa offers the function `reactimate`. It allows receiving new inputs and processing the outputs.

```haskell
reactimate :: Monad m
  -- Initialization action
  => m a
  -- Input sensing action
  -> (Bool -> m (DTime, Maybe a))
  -- Output processing action
  -> (Bool -> b -> m Bool)
  -- Signal function
  -> SF a b
  -> m ()
```

```haskell
reactimate
    (initializationAction)
    (\_ -> inputSensingAction)
    (\_ positions -> outputProcessingAction
     positions)
    islandersBehavior
```

The `initializationAction` function defines the first input at the beginning of the execution. In this case, the sound should be zero. After the start of the execution, the function `inputSensingAction` returns either new keyboard input or sound events from the microphone. The discarded Boolean argument indicates if the `inputSensingAction` can block. The function `islandersBehavior` of type `SF a [Pos]` creates a sound signal that is altered depending on the input. Then it starts for each islander the `islander` signal function and passes the sound signal as input. The resulting positions of each islander are gathered together and passed to the outputProcessingAction function, which sends them to the GUI or the actuators. The first argument indicates if the output has changed. The outputProcessingAction should return True if the function reactimate should stop otherwise the execution continues.

The function islander defines the current behavior of an islander. It takes the sound as an input. Then it calculates the current velocity with the function behavior and then integrates the velocity to calculate the current position.

```
islander :: SF Sound Pos
islander = proc sound -> do
    rec
        v <- behavior -< (sound, pos)
        pos <- integral -< v
    returnA -< pos
```

The behavior function needs the sound and the current position as input. This allows the function to adjust the speed according to the sound level and to stop the islander's movement at certain positions. The position is an input and output which leads to a recursive call (see Figure 9). To enable this, the keyword rec is used in the arrow notation.

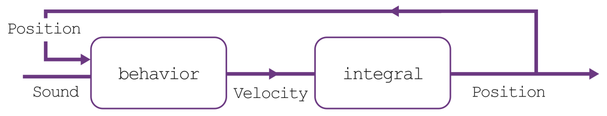

Figure 9. Visualization of islander transforming the sound signal to a position signal.

3.2.2 Behavior of the Islander The state transitions of the islander's behavior can be implemented with switch. It switches once between two signal functions of type SF a b when an event occurs in the given event stream[5] of type Event c.

```
switch
    :: SF a (b, Event c)
    -> (c -> SF a b)
    -> SF a b
```

The first parameter is a signal function combining the current behavior Signal b with the event stream Signal (Event c). If an event occurs, its value is passed to the function in the second parameter of switch that creates the signal function with the new behavior. After the switch, the new signal function is applied.

For example, the state transition to hide would be implemented as follows:

[5]Note that there are different versions of the function switch: https://hackage.haskell.org/package/Yampa-0.14/docs/FRP-Yampa.html#g:11

```
goHiding :: SF (Sound, Pos) (Vel, Event ())

behavior :: SF (Sound, Pos) Vel
behavior = (creepOut `doUntil` danger)
    `switch` (\_ -> goHiding
    `switch` const behavior)
```

The function creepOut describes the normal sequence of an islander moving out. The function delivers the corresponding velocity signal. The implementation of creepOut is given further below. The signal function danger produces events when the sound value in the input signal is too high. The danger events and the velocity of creepOut are combined into a tuple with the function doUntil.

```
doUntil
    :: SF a b
    -> SF a (Event c)
    -> SF a (b, Event c)
doUntil behavior event = behavior &&& event
```

The operator &&& combines the behavior and the event stream and is of the following type:

```
(&&&) :: SF b c -> SF b c' -> SF b (c, c')
```

When an event occurs in danger, it will trigger the switch. From this moment the velocity will be defined by the function goHiding, which sets the velocity to a high negative value until the islander arrives at the hiding position. The underscore in the lambda function after the switch means, that the input parameter (the value of the event triggering the switch) is discarded. As soon as the islander arrives, the behavior will recursively switch to the behavior function to restart the crawling process.

The moving out of an islander in the creepOut signal function would look like this.

```
creepOut :: SF (Sound, Pos) Vel
creepOut = (stop `doUntil` safe)
    `switch` (\_ -> waitBeforePeek
    `switch` (\_ -> goPeeking
    `switch` (\_ -> waitBeforeStand
    `switch` (\_ -> goStanding
    `switch` const stop))))
```

With the implementation using FRP, it becomes clear as to which event is causing which reaction. First, the islander waits until it is quiet or safe. Then it waits for its individual waiting time. Up to this moment, the output signal for the speed contains the value zero. When the time is up, the behavior changes to forward motion until the peeking position has been reached. The output signal holds the resulting velocity value during this time. The behavior changes again when the arrival event occurs, and the velocity is set to zero. The movement to the standing position then works analog to the previous sequence. Upon arrival there, the velocity value remains at zero.

3.2.3 Movement To move the islander to a position, a signal indicating the speed and an event stream are needed. When the islander reaches the target position, the event stream generates an event triggering the switch in creepOut. Then, one of the wait signal functions will be executed which sets the velocity to zero.

The function goPeeking combines the velocity with the arrival event. As seen in the Yampa example, a velocity signal can be created with constant.

```
goPeeking :: SF (Sound, Pos) (Vel, Event ())
goPeeking = constant peekV `doUntil`
    arrivalPeeking
```

In the function arrivalPeeking, the condition hasArrived is checked. Therefore, the current position of the islander needs to be extracted from the input signal using the arrow notation. As soon as the condition is true, the signal function created by edge of type SF Bool (Event ()) will generate an event.

```
arrivalPeeking :: SF (Sound, Pos) (Event())
arrivalPeeking = proc (_, pos) -> do
   let hasArrived = peekPos <= pos
   event <- edge -< hasArrived
   returnA -< event
```

3.2.4 Waiting To stay in a position for a certain time, the functions stop and timeToPeek are needed.

```
waitBeforePeek :: SF (Sound, Pos) (Vel, Event())
waitBeforePeek = stop `doUntil` timeToPeek
```

The signal function stop produces a signal with a value of zero.

```
stop :: SF () Vel
stop = constant 0
```

In order to generate an event at a specific time, there is the function after. It takes a time and a value and then produces the event with the corresponding value at the corresponding time.

```
after :: Time -> b -> SF a (Event b)
```

The implementation of the function timeToPeek is given below. Since the event value does not matter () is used.

```
waitingTime :: Time

timeToPeek :: SF (Sound, Pos) (Event())
timeToPeek = after waitingTime ()
```

4 Implementation Details

This section describes how the code was adapted to make the islanders life-like and how the hardware connection and another version of the artwork were implemented. Also, the advantages and difficulties of FRP encountered during these tasks are presented.

4.1 Making the Islanders life-like

The artwork is characterized by the fact that the figures look life-like. This is because the figures move differently due to randomly generated parameters. Furthermore, according to a behavior algorithm, different ranges for waiting times and speeds are used depending on the sound level. For instance, at a high sound level, the figures move more nervously or faster and remain longer in a position.

In order to keep the first implementation as simple as possible, the algorithm was simplified. There are predefined value ranges in which the state variables of an islander can lie.

```
type VelRange = (Vel, Vel)
type PosRange = (Pos, Pos)
type TimeRange = (Time, Time)

data IslanderState = IslanderState
{
  waitToPeek  :: TimeRange,
  waitToStand :: TimeRange,
  hiding      :: (Pos, VelRange),
  peeking     :: (PosRange, VelRange),
  standing    :: (Pos, VelRange),
  isHidingComplete :: Bool
}
```

In the beginning, the islanders' states are created with random values in the corresponding ranges. Instead of increasing these ranges for loud noises and setting new random values, the values are increased by a factor for each loud noise. Every time it is quiet, the values are decreased by a factor again. The original values serve as a lower limit. The following sections describe difficulties encountered during implementation.

4.1.1 Missed Events During the implementation of the differing behavior of the islanders, the application began to behave strangely. The problem was finally found in the signal function safe, which triggers the first switch in creepOut. The function always generated an event when the sound signal changed to zero.

```
edge :: SF Bool (Event ())

-- old implementation of safe
safe :: SF (Sound, Pos) (Event())
safe  = proc (sound, _) -> do
    event <- edge -< sound == 0
    returnA -< event
```

For all islanders that were waiting in the hiding position at this time, this event triggered the switch. However, if an islander was not yet in the hiding position, it missed the event and did not reappear until it became quiet again after a loud event.

To solve the problem, events must be generated repeatedly while it is quiet. Therefore, a signal function quiet of type SF Sound Bool was added to safe. When the sound signal becomes zero, the output signal of quiet is set to True, and when the sound increases to False again. To repeatedly create events while the sound is zero, repeatedly is used and combined with gate. The function repeatedly creates repeating events and gate disables all events during the time quiet has the value False.

```
gate :: Event a -> Bool -> Event a
repeatedly :: Time -> b -> SF a (Event b)

-- new implementation of safe
safe :: SF (Sound, Pos) (Event())
safe  = proc (sound, _) -> do
    isQuiet <- quiet -< sound
    clockEvent <- repeatedly 1 () -< ()
    returnA -< clockEvent `gate` isQuiet
```

4.1.2 Islander State as a Signal To change the behavior of the islanders over time (e.g. increase the speed with louder sounds) the islander's state should be passed as an input signal. During the realization, the function creepOut had to be adapted. The problem was that the function creepOut uses functions like after and constant. Based on the parameters passed, they create a signal function at the beginning of the execution. For example, after creates a signal function that creates an event as soon as the time is up. In the example below, the event stream created by after creates an event at time waitToPeek stored in the islander's state.

```
timeToPeek
    :: IslanderState
     -> SF (Sound, Pos) (Event())
timeToPeek state = after (waitToPeek state) ()
```

If the state parameter is passed in the input signal to timeToPeek, it can no longer be used as a function parameter as after requires. The reason for this is that the input signal's value can change over time, which requires repeated creation of signal functions. The value of an input signal can only be passed again as an input signal to other signal functions. So, the following code is therefore not possible:

```
timeToPeek
   :: SF (IslanderState, Sound, Pos) (Event ())
timeToPeek = proc (state, _, _) -> do
   -- this does not compile:
   event <- after (waitToPeek state) () -< ()
   returnA -< event
```

The problem was solved by passing the parameter at the switch. When the event that triggers the switch occurs, the current islander's state is read from the input signal and attached to the event. The function switch passes the value of the event as a parameter to the creation of the new signal function that should be executed now. Since the state is passed as a function parameter and not as an input signal, functions like after can be used to create the new signal function. Here is the function safe as an example:

```
safe = proc (state, _, sound) -> do
    -- ...
    returnA -< event `tag` (waitToPeek state)
```

The passing of the value after the switch looks like this:

```
creepOut :: SF (IslanderState, Pos, Sound) Vel
creepOut = (stop `doUntil` safe)
   `switch` (\state -> waitBeforePeek state
   `switch` (\state-> goPeeking state
   `switch` (\state -> waitBeforeStand state
   `switch` (\state -> goStanding state
   `switch` const stop ))))
```

The relationships between the state transitions are still clearly visible. However, if changes are made to the sequence, it must be ensured that all the event values are set in the correct order.

4.2 Connect Hardware

To see how the new design would look in FRP, the application was first created as a simulation. The sound level is simulated with the keyboard and as output, the islanders are displayed in the GUI. Thanks to FRP's modular approach, only a few changes were needed to connect to the real artwork. The artwork uses a framework that provides an interface for the microphone and the actuators. A socket connection was implemented to enable communication between the framework and the Haskell code.

The biggest change was rewriting the call to the function reactimate. Instead of the keyboard, the microphone data is now used as input. And instead of being displayed in the GUI, the resulting positions are sent to the framework. There the actuators are moved to the new position. The difficulty was to determine the right parameters for interpreting the current position to ensure that the islanders do not stumble or move differently than in the simulation.

Now, the same code can be connected to both the simulation and the hardware. This makes it easy to adjust and test the behavior of the islanders in the simulation before running the code on the artwork. This also makes it easier to identify the cause of an error. If an error occurs in both the artwork and the simulation, it is most likely an issue in the behavior program. Otherwise, the problem lies in the input or output management.

4.3 Add a new Version of the Artwork

There is another artwork by Pors & Rao called "Lone Islander" which is a variation of the Islanders artwork. This artwork has one islander on each side of the panel. Only one of the four islanders crawls out from behind the panel at a time. As soon as it hides, the side is randomly changed. This gives the viewer the feeling that it is always the same islander.

This version was also added to the software. Due to the modular structure, only the function `islandersBehavior` had to be adapted, which starts the `islander` function for each islander. For the lone islander, it is started only once. Every time the islander reaches the hiding position, the new panel side is determined. The function then returns the result of the `islander` function for the current panel side and zero for the other positions.

The new panel side is determined using the function `noiseR`. This generates a random signal containing values in the specified range. It is based on the pseudo-random function `randomRs` from the random library [oG22].

```
noiseR
   :: (RandomGen g, Random b)
   => (b, b)
   -> g
   -> SF a b
```

Here is a snippet of the implementation of `islander`:

```
type In = (IslanderState, Pos, Sound)
type Mov = (Pos, Vel)

danger :: SF In (Event Mov)
danger = proc (p, y, sound) -> do
  small <- eventInRange smallRange -< sound
  loud <- eventInRange loudRange -< sound
  returnA -<  (small `tag` hideOrPeekMov)
     `rMerge` (loud `tag` hideMov)

quiet :: SF Sound Bool
quiet = proc sound -> do
  quietEvent <- eventInRange quietRange -< sound
  soundEvent <- eventInRange noiseRange -< sound
  quiet <- hold False -< quietEvent `tag` True
              `rMerge` soundEvent `tag` False
  returnA -< quiet

safe :: SF In (Event Time)
safe = proc (p, _, sound) -> do
  isQuiet <- quiet -< sound
  clockEvent <- repeatedly 1 () -< ()
  let onlyQuiet = (clockEvent `gate` isQuiet)
  returnA -< onlyQuiet `tag` waitToPeek p

arrivalStanding :: Pos -> SF In (Event ())
arrivalHiding  :: Pos  -> SF In (Event ())
arrivalPeeking :: Pos -> SF In (Event Time)
arrivalPeeking target = proc (p, pos, _) -> do
  let hasArrived = target <= pos
  event <- edge -< hasArrived
  returnA -< event `tag` waitToStand p

stop :: SF In Vel
stop = constant 0

goHiding   :: Mov -> SF In (Vel, Event())
goStanding :: Mov -> SF In (Vel, Event())
goPeeking  :: Mov -> SF In (Vel, Event())
goPeeking (pos, v) = constant v `doUntil`
    arrivalPeeking pos

timeToPeek :: Time -> SF In (Event Mov)
timeToPeek t = proc (state, _, _) -> do
  event <- after t () -< ()
  returnA -< event `tag` peeking state

waitBeforePeek :: Time -> SF In (Vel, Event Mov)
waitBeforeStand :: Time -> SF In (Vel, Event Mov)
waitBeforePeek t = stop `doUntil` timeToPeek t

creepOut  :: SF In Vel
creepOut = (stop `doUntil` safe)
  `switch` (\state -> waitBeforePeek state
  `switch` (\state -> goPeeking state
  `switch` (\state -> waitBeforeStand state
  `switch` (\state -> goStanding state
  `switch` const stop))))

behavior  :: SF In Vel
behavior = (creepOut `doUntil` danger)
  `switch` (\state -> goHiding state
  `switch` const behavior)

islander :: SF (IslanderState, Sound) Pos
islander = proc (state, sound) -> do
  rec
    v <- behavior -< (state, pos, sound)
    pos <- integral -< v
  returnA -< pos
```

5 Conclusions and Future Work

The implementation of the control software of the Islanders artwork with FRP brings many advantages. Due to the modularization and the more visible state transitions, the code is more comprehensible, and thus it is simpler to make changes in the code. However, care must be taken when adapting code responsible for generating events. For example, a missed arrival event will cause the islanders to move off-screen. Furthermore, it is a prerequisite to know the `Arrow` concept to understand and adapt the code. In a further development, an abstraction would have to be offered that allows non-programmers to adjust the behavior of the islanders without the use of `Arrows`.

The implementation in FRP comes close to the original implementation. The figures do not yet appear as life-like as in the original, but this is due to the simplified implementation of the behavior algorithm. However, the current implementation shows that it is possible to implement the behavior algorithm in FRP in a similar form.

Thanks to the separation of the control from the peripherals, it was easy to replace the GUI output with the actuator control. Both versions receive the same position signal resulting in similar behavior of the simulation and the artwork. Changes to the program can thus be easily tested in the simulation before connecting the hardware.

6 Acknowledgment

Many thanks to Pors & Rao for providing their artwork for this project.

References

[BJ16] Stephen Blackheath and Anthony Jones. *Functional Reactive Programming*. Manning, 2016.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, aug 1997.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111, 2000.

[Kri13] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, 48(9):221–232, sep 2013.

[oG22] The University of Glasgow. Hackage package: random. https://hackage.haskell.org/package/random-1.2.1.1/docs/System-Random.html, 2022. Accessed: 2023-06-14.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, page 91–105, Berlin, Heidelberg, 1999. Springer-Verlag.

[PNH02] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, page 168–179, New York, NY, USA, 2002. Association for Computing Machinery.

[Rao06] Pors & Rao. Islanders. http://www.porsandrao.com/work/?workid=21, 2006. Accessed: 2022-12-06.

[Rao20] Pors & Rao. Pors & Rao. http://www.porsandrao.com/, 2020. Accessed: 2023-01-05.