This directory contains the Equalizer example, used in Sander's PhD Thesis [1], and modelled using shallow-embedded signals.

0.1 Overview

The main task of the equalizer system is to adjust the audio signal according to the `Button Control`, that works as a user interface. In addition, the bass level must not exceed a predefined threshold to avoid damage to the speakers.

This specification can be naturally decomposed into four functions shown in Figure **??**. The subsystems `Button Control` and `Distortion Control`, are control dominated (grey shaded), while the `Audio Filter` and the `Audio Analyzer` are data flow dominated subsystems.

The `Button Control` subsystem monitors the button inputs and the override signal from the subsystem `Distortion Control` and adjusts the current bass and treble levels. This information is passed to the subsystem `Audio Filter`, which receives the audio input, and filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the `Audio Analyzer` subsystem, which determines, whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem `Distortion Control`, which decides, if a minor or major violation is encountered and issues the necessary commands to the `Button Control` subsystem.

The frequency characteristics of the `Equalizer` is adjusted by the coefficients for the three FIR-filters in the `AudioFilter`.

```
module Equalizer(equalizer) where

import ForSyDe.Shallow

import ButtonControl
import DistortionControl
import AudioAnalyzer
import AudioFilter
```

The structure of the equalizer is expressed as a network of blocks:

```
equalizer lpCoeff bpCoeff hpCoeff dftPts
          bassUp bassDn trebleUp trebleDn input = (bass, treble)
  where
    (bass, treble)  = buttonControl overrides bassUp bassDn
                                    trebleUp trebleDn
    output          = audioFilter lpCoeff bpCoeff hpCoeff bass
                                  treble input
    distFlag        = audioAnalyzer dftPts output
    overrides       = distortionControl delayedDistFlag
    delayedDistFlag = delaySY Abst distFlag
```

Since the equalizer contains a feedback loop, the signal `DistFlag` is delayed one event cycle using the initial value $\perp$.

0.2 Overview

The subsystem `Button Control` works as a user interface in the equalizer system. It receives the four input signals $\overline{\texttt{BassDn}}$, $\overline{\texttt{BassUp}}$, $\overline{\texttt{TrebleDn}}$, $\overline{\texttt{TrebleUp}}$ and the override signal `Override` from the `Distortion Control` and calculates the new bass and treble values for the output signals $\overline{\texttt{Bass}}$ and $\overline{\texttt{Treble}}$. The subsytem contains the main processes `Button Interface` and `Level Control`. The process `Level Control` outputs a new value, if either the signal $\overline{\texttt{Button}}$ or the signal $\overline{\texttt{Overr}}$ is present, otherwise the output value is absent. The process `Hold Level` is modeled by means of `holdSY (0.0, 0.0)` that outputs the last present value, if the input value is absent. The process `unzipSY` transforms a signal of tuples (the current bass and treble level) into a tuple of signals (a bass and a treble signal).

```
module ButtonControl (buttonControl) where

import ForSyDe.Shallow
import EqualizerTypes
--import Combinators

data State   = Operating
             | Locked deriving(Eq, Show)
type Level   = Double
type Bass    = Level
type Treble  = Level

buttonControl :: Signal (AbstExt OverrideMsg) -> Signal (AbstExt
     Sensor)
              -> Signal (AbstExt Sensor) -> Signal (AbstExt
                  Sensor)
              -> Signal (AbstExt Sensor) -> (Signal Bass,Signal
                  Treble)
buttonControl overrides bassDn bassUp trebleDn trebleUp
    = (bass, treble)
      where (bass, treble) = unzipSY levels
            levels = holdSY (0.0, 0.0) $ levelControl button
                overrides
            button = buttonInterface bassDn bassUp trebleDn
                trebleUp
```

0.3 The Process `Button Interface`

The `Button Interface` monitors the four input buttons $\overline{\texttt{BassDn}}$, $\overline{\texttt{BassUp}}$, $\overline{\texttt{TrebleDn}}$, $\overline{\texttt{TrebleUp}}$ and indicates if a button is pressed. If two or more buttons are pressed the conflict is resolved by the priority order of the buttons.

```
buttonInterface :: Signal (AbstExt Sensor) -> Signal (AbstExt
    Sensor)
                -> Signal (AbstExt Sensor) -> Signal (AbstExt
                    Sensor)
                -> Signal (AbstExt Button)
buttonInterface bassUp bassDn trebleUp trebleDn
   = zipWith4SY f bassUp bassDn trebleUp trebleDn
     where f (Prst Active) _ _ _ = Prst BassUp
```

```haskell
            f _ (Prst Active) _ _ = Prst BassDn
            f _ _ (Prst Active) _ = Prst TrebleUp
            f _ _ _ (Prst Active) = Prst TrebleDn
            f _ _ _ _             = Abst
```

0.4 The Process `Level Control`

The process has a local state that consists of a mode and the current values for the bass and treble levels (Figure **??**). The `Level Control` has two modes, in the mode `Operating` the bass and treble values are stepwise changed in 0.2 steps. However, there exists maximum and minimum values which are -5.0 and +5.0. The process enters the mode `Locked` when the `Override` input has the value `Lock`. In this mode an additional increase of the bass level is prohibitet and even decreased by 1.0 in case the `Override` signal has the value `CutBass`. The subsystem returns to the `Operating` mode on the override value `Release`. The output of the process is an absent extended signal of tuples with the current bass and treble levels.

```haskell
levelControl :: Signal (AbstExt Button) -> Signal (AbstExt
    OverrideMsg)
             -> Signal (AbstExt (Bass,Treble))
levelControl button overrides
  = mealy2SY nextState output (initState, initLevel) button
      overrides

nextState :: (State,(Double,Double)) -> AbstExt Button
          -> AbstExt OverrideMsg -> (State,(Double,Double))
nextState (state, (bass, treble)) button override
  = (newState, (newBass, newTreble)) where
     newState = if state == Operating then
                   if override == Prst Lock then
                      Locked
                   else
                      Operating
                else
                   if override == Prst Release then
                      Operating
                   else
                      Locked

     newBass = if state == Locked then
                  if override == Prst CutBass then
                     decreaseLevel bass cutStep
                  else
                     if button == Prst BassDn then
                        decreaseLevel bass step
                     else
                        bass
               else -- state = Operating
                  if button == Prst BassDn then
                     decreaseLevel bass step
                  else
                     if button == Prst BassUp then
                        increaseLevel bass step
```

```
                     else
                        bass

      newTreble = if button == Prst TrebleDn then
                     decreaseLevel treble step
                  else
                     if button == Prst TrebleUp then
                        increaseLevel treble step
                     else
                        treble

output :: (a, (Bass, Treble)) -> AbstExt Button -> AbstExt
    OverrideMsg
          -> AbstExt (Bass,Treble)
output _            Abst Abst = Abst
output (_, levels) _     _    = Prst levels
```

The process uses the following initial values.

```
initState =  Operating
initLevel =  (0.0, 0.0)
maxLevel  =  5.0
minLevel  = -5.0
step      =  0.2
cutStep   =  1.0
```

The process uses the following auxiliary functions.

```
decreaseLevel :: Level -> Level -> Level
decreaseLevel level step = if reducedLevel >= minLevel then
                              reducedLevel
                           else
                              minLevel
                           where reducedLevel = level - step

increaseLevel :: Level -> Level -> Level
increaseLevel level step = if increasedLevel <= maxLevel then
                              increasedLevel
                           else
                              maxLevel
                           where increasedLevel = level + step
```

The block `Distortion Control` is directly developed from the SDL-specification, that has been used for the MASCOT-model [**BjJa2000**]. The specification is shown in Figure **??**.

Figures/DistortionControl-eps-converted-to.pdf

Figure 1: SDL-description of *Distortion Control*

The `Distortion Control` is a single FSM, which is modeled by means of the skeleton `mealySY`. The global state is not only expressed by the explicit states `-Passed`, `Failed` and `Locked -`, but also by means of the variable cnt. The state machine has two possible input values, `Pass` and `Fail`, and three output values, `Lock`, `Release` and `CutBass`.

The `mealySY` creates a process that can be interpreted as a Mealy-machine. It takes two functions, nxtSt to calculate the next state and out to calculate the output. The state is represented by a pair of the explicit state and the variable cnt. The initial state is the same as in the SDL-model, given by the tuple `(Passed, 0)`. The nxtSt function uses pattern matching. Whenever an input value matches a pattern of the nxtSt function the corresponding right hand side is evaluated, giving the next state. An event with an absent value leaves the state unchanged. The output function is modeled in a similar way. The output is absent, when no output message is indicated in the SDL-model.

```haskell
module DistortionControl (distortionControl) where

import ForSyDe.Shallow
import EqualizerTypes

data State =  Passed
            | Failed
            | Locked

distortionControl ::  Signal (AbstExt AnalyzerMsg)
                   -> Signal (AbstExt OverrideMsg)

distortionControl distortion
   = mealySY nxtSt out (Passed, 0) distortion

lim = 3

--      State        Input        NextState
nxtSt (state, cnt) (Abst)       = (state,cnt)
nxtSt (Passed,cnt) (Prst Pass) = (Passed,cnt)
nxtSt (Passed,_  ) (Prst Fail) = (Failed,lim)
nxtSt (Failed,cnt) (Prst Pass) = (Locked,cnt)
nxtSt (Failed,cnt) (Prst Fail) = (Failed,cnt)
nxtSt (Locked,_  ) (Prst Fail) = (Failed,lim)
nxtSt (Locked,cnt) (Prst Pass) = (newSt,newCnt)
   where newSt = if (newCnt == 0) then Passed
                                  else Locked
         newCnt = cnt - 1

--    State          Input        Output
out (Passed,_)    (Prst Pass) = Abst
out (Passed,_)    (Prst Fail) = Prst Lock
out (Failed,_)    (Prst Pass) = Abst
out (Failed,_)    (Prst Fail) = Prst CutBass
out (Locked,_)    (Prst Fail) = Abst
out (Locked,cnt) (Prst Pass) =
   if (cnt == 1) then Prst Release
                 else Abst
out _              Abst        = Abst
```

```haskell
module TestEqualizer where

import System.IO
```

```haskell
import Equalizer
import EqualizerTypes
import ForSyDe.Shallow
import AudioFilter
import AudioAnalyzer

audioIn = takeS (pts * 4) $ infiniteS (id) 1.0

bassUp = signal [Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst]

bassDn = signal [Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst]

trebleUp = signal [Abst, Abst, Abst, Abst,
                 Abst, Abst, Abst, Prst Active,
                 Abst, Abst, Abst, Abst,
                 Abst, Abst, Abst, Prst Active]

trebleDn = signal [Abst, Abst, Abst, Prst Active,
                 Abst, Abst, Abst, Prst Active,
                 Abst, Abst, Abst,  Prst Active,
                 Abst, Abst, Abst, Prst Active]

overrides = signal [Abst, Abst, Abst, Abst,
                    Prst Lock, Abst, Abst, Abst,
                    Abst, Prst CutBass, Abst, Abst,
                    Prst Release,  Abst, Abst, Abst]
bass = infiniteS id 0.0
treble = infiniteS id 0.0
dataflow = audioAnalyzer 2 (audioFilter lpCoeff bpCoeff hpCoeff
    bass treble audioIn)

testEqualizer = equalizer lpCoeff bpCoeff hpCoeff 2 bassUp
    bassDn trebleUp trebleDn audioIn
--testButtonInterface = buttonInterface  bassUp bassDn trebleUp
    trebleDn
```

```haskell
lpCoeff = vector
     [ 0.01392741661548, 0.01396895728902,
       0.01399870011280, 0.01401657422649,
       0.01402253700635, 0.01401657422649,
       0.01399870011280, 0.01396895728902,
       0.01392741661548 ]

bpCoeff = vector
     [ 0.06318761339784, 0.08131651217682,
       0.09562326700432, 0.10478344432968,
       0.10793629404886, 0.10478344432968,
       0.09562326700432, 0.08131651217682,
       0.06318761339784 ]

hpCoeff = vector
     [ -0.07883878579454, -0.09820015927379,
```

```haskell
        -0.11354603917221,  -0.12339860164118,
         0.87320570334018,  -0.12339860164118,
        -0.11354603917221,  -0.09820015927379,
        -0.07883878579454 ]

zeros = infiniteS id 0.0

audioFilterD = audioFilter lpCoeff bpCoeff hpCoeff zeros zeros
pts = 4

main = do
          -- Test AudioFilter
          putStr "\n-->Test AudioFilter \n"
          filterInfile <- openFile "Test/AudioIn.mat" ReadMode
          filterContents <- hGetContents filterInfile
          putStr (show (audioFilterD (readS filterContents)))
          writeFile "Test/audioOut.dat" (writeS (audioFilterD (
              readS filterContents)))
          -- Test AudioAnalyzer
          putStr "\n--> Test AudioAnalyzer \n"
          analyzerInfile <- openFile "Test/audioOut.dat"
              ReadMode
          analyzerOutfile <- openFile "Test/analyzerOut.dat"
              WriteMode
          analyzerContents <- hGetContents analyzerInfile
          putStr (show (audioAnalyzer pts ((readS
              analyzerContents) :: Signal Double)))
          hPutStr analyzerOutfile (writeS (audioAnalyzer pts ((
              readS analyzerContents) :: Signal Double)))

          --fftInfile <- openFile "audioOut.txt" ReadMode
          --fftOutfile <- openFile "fftOut.txt" WriteMode
          --contents <- hGetContents fftInfile
          --putStr (writeS (((readS contents) :: Signal Double))
              )
          --hPutStr fftOutfile (writeS (((readS contents) ::
              Signal Double)))
          putStr "\nDone.\n"
```

```haskell
module TestButtonControl where

import ButtonControl
import EqualizerTypes
import ForSyDe.Shallow




bassUp = signal [Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst,
                 Prst Active, Prst Active, Abst, Abst]

bassDn = signal [Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst,
                 Abst, Abst, Prst Active, Abst]
```

```haskell
trebleUp = signal [Abst, Abst, Abst, Abst,
                   Abst, Abst, Abst, Prst Active,
                   Abst, Abst, Abst, Abst,
                   Abst, Abst, Abst, Prst Active]

trebleDn = signal [Abst, Abst, Abst, Prst Active,
                   Abst, Abst, Abst, Prst Active,
                   Abst, Abst, Abst,  Prst Active,
                   Abst, Abst, Abst, Prst Active]

overrides = signal [Abst, Abst, Abst, Abst,
                    Prst Lock, Abst, Abst, Abst,
                    Abst, Prst CutBass, Abst, Abst,
                    Prst Release,  Abst, Abst, Abst]

testButtonControl = buttonControl overrides bassUp bassDn
    trebleUp trebleDn
--testButtonInterface = buttonInterface  bassUp bassDn trebleUp
    trebleDn
```

```haskell
module TestDistortionControl where

import DistortionControl
import EqualizerTypes
import ForSyDe.Shallow

flag = signal
       [Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Fail,
        Abst, Abst, Abst, Prst Pass,
        Abst, Abst, Abst, Prst Pass]

testDistortionControl = distortionControl flag
```

References

[1] Ingo Sander. “System Modeling and Design Refinement in ForSyDe”. NR 20140805. PhD thesis. KTH, Microelectronics and Information Technology, IMIT, 2003, pp. xvi, 228. ISBN: 91-7283-501-X.