The ASK transceiver example in ForSyDe

Ingo Sander, Jun Zhu and Axel Jantsch
Kungliga Tekniska Högskolan
Stockholm, Sweden

January 18, 2017

Contents

Chapter 1

Introduction

1.1 Overview

The ForSyDe model of the ASK uses the following computational models.

- Untimed model (SDF)
- Synchronous model (SR)
- Continuous time model (CT)

The example can be downloaded from the ANDRES homepage. In order to run the model we need a version of the ForSyDe standard library that includes the new library for continuous time models. Since the CT-library is still under development, there is no stable release with it available. However, an intermediate version can be downloaded from the ANDRES homepage.

The structure of the ForSyDe model is shown in Figure 1.1.

All signal names shown in the figure can be accessed, if the ForSyDe model is executed with a Haskell interpreter like `hugs`[1] or `ghci`[2].

1.1.1 Installation

We assume in the following that you are using a UNIX environment.

1. Make a directory `ForSyDeExample` somewhere in your directory structure. Enter this directory.
2. Download the file `ForSyDeStdLib.zip` from the ANDRES homepage
3. Download the file `ToyExample.zip` from the ANDRES homepage

[1] `hugs` can be downloaded from `http://haskell.org/hugs`

[2] `ghci` is part of the Glasgow Haskell Compiler, which can be downloaded from `http://haskell.org/ghc/`

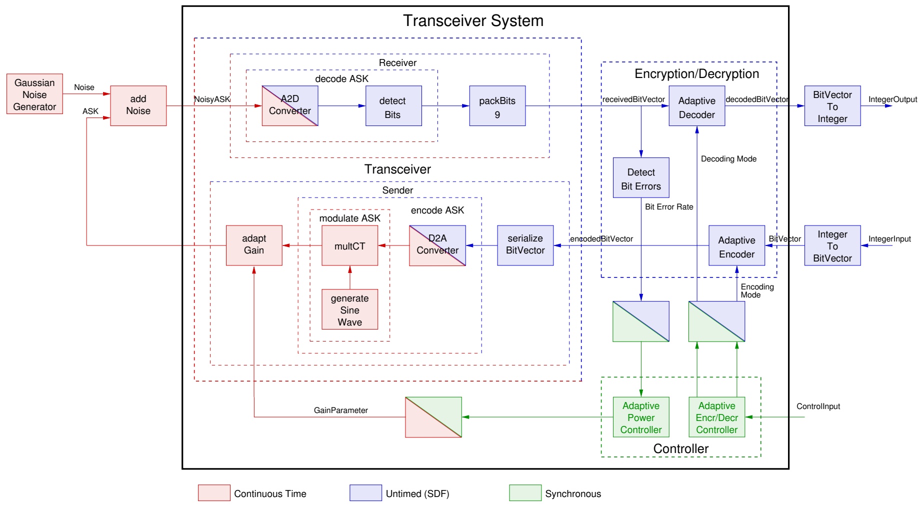

Figure 1.1: The Structure of ASK transceiver case study

4. Extract the ForSyDe library so that it is located under the directory `ForSyDeExample`

5. Extract the toy example so that it is located under the directory `ForSyDeExample`

6. Create an environment variable `$FORSYDELIB` that points to the directory `ForSyDeStdLib`

7. Move to the directory for the toy example

8. Start your Haskell interpreter with

   - `hugs -P:$FORSYDELIB` (for `hugs`)
   - `>ghci -i$FORSYDELIB` (for `ghci`)

9. Then load the testbench, which includes all the modules with `:l TestBench`.

10. Then start the simulation by `main`.

11. You should now see something like this.

```
>hugs -P:\$FORSYDELIB
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: May 2006       _________________________________________
```

```
Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :l Main
Main> main
Testing ...
Input signal of integers
input = {0,1,2,3,4}
Output signal of integers
output = {0,40,2,3,4}
Signal plotted.
Signal plotted.
Signal plotted.
Done!
```

12. Three plots are generated during the execution of the model.

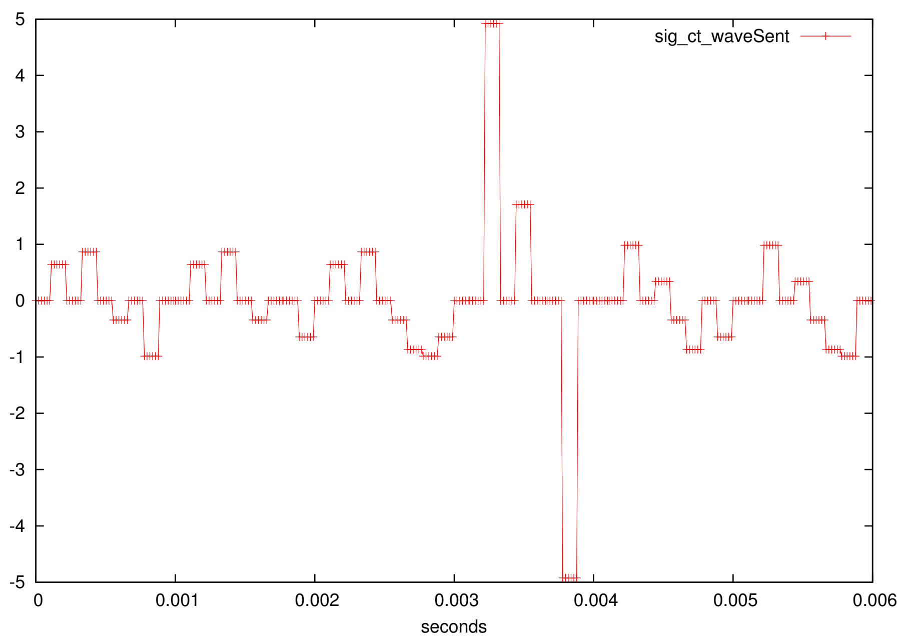

Figure 1.2: The signal `sig_ct_waveSent`

As can be seen in the figures, the gain is increased to a higher level after an increase of the bit error rate. The 2nd input signal '1' was received as a '40'.

13. You can also look at exported internal signals like the interfaces between the subsystems of Figure 1.1.

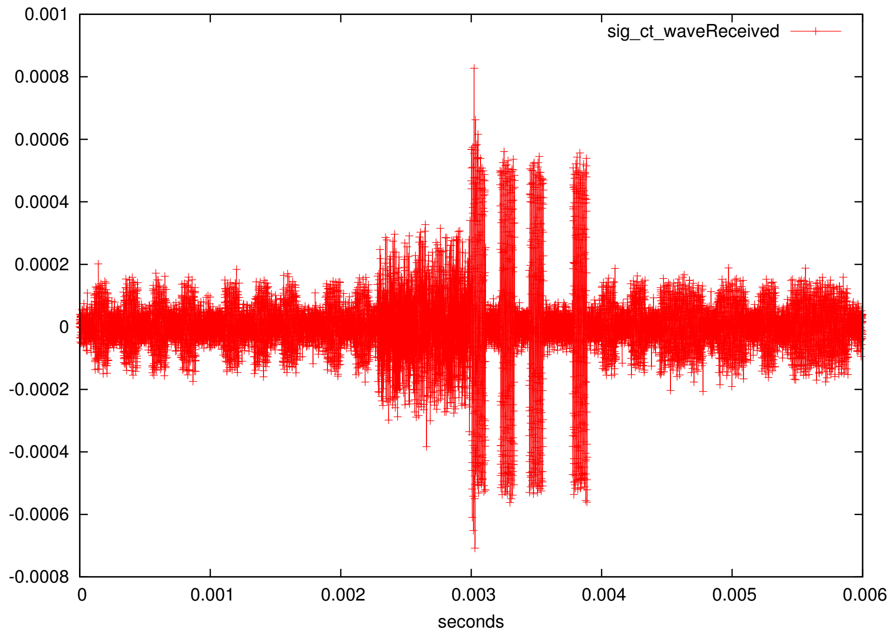

Figure 1.3: The signal `sig_ct_waveReceived`

```
Main> bitErrorRate
{0,1,0,0,0}
```

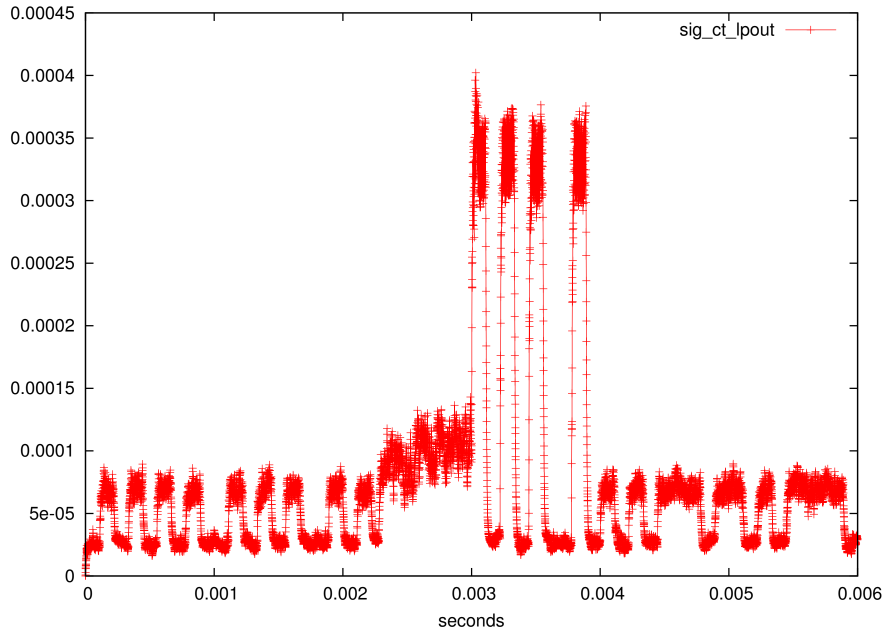

Figure 1.4: The signal `sig_ct_lpout`

Chapter 2

The module `TransceiverSystem`

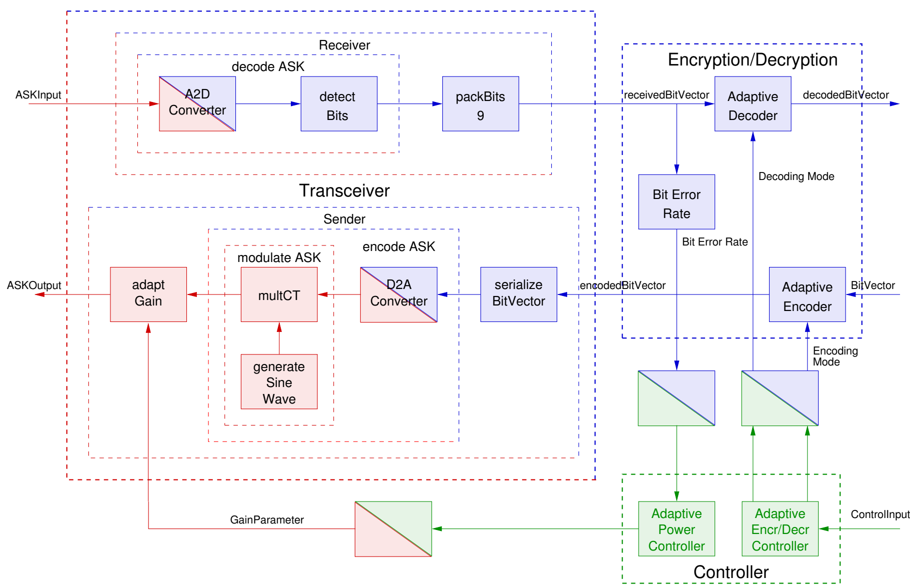

Figure 2.1: The Structure of the module `TransceiverSystem`

The transceiver system contains the Transceiver, Encryption, Decryption, Controller and interfaces between different model of computation domains. It is the system module under test.

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.TransceiverSystem
    where
```

```haskell
-- import ForSyDeMoCLib
-- import CTLib
-- import BitVector

import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib
import ForSyDe.Shallow.BitVector

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Transceiver
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.EncDec
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Controller


transceiverSystem ::
  Signal (SubsigCT Double) -- The input CT signal to ASK receiver
  -> Signal (Vector Integer) -- The input SDF signal to Encyption module
  -> Signal Integer      -- The input SR signal to control the Enc/Dec algorithms
  -> (Signal (Vector Integer),
      Signal (SubsigCT Double),
      Signal (SubsigCT Double),
      Signal (Vector Integer),
      Signal (Vector Integer),
      Signal Integer
--       , Signal Double
     )
transceiverSystem sig_ct_waveReceived sig_sr_testIn sig_sr_testCryptoMode =
  (sig_sr_testOut, sig_ct_waveSent, sig_ct_lpout, sig_sr_Rx,
                   sig_sdf_Tx, sig_sr_bitError
--                   ,
  )
  where
    -- Transceiver module
    (sig_sdf_Rx, sig_ct_lpout, sig_ct_waveSent) =
            transceiver sig_ct_waveReceived sig_sdf_Tx sig_ct_powerMode sig_sdf_thresh'
    -- Encryption/decryption module
    (sig_sr_bitError, sig_sr_testOut, sig_sr_Tx)
         = moduleEncDec sig_sr_Rx sig_sr_testIn sig_sr_cryptoMode
    -- Controller module
    (sig_sr_cryptoMode, sig_sr_powerMode,sig_sdf_thresh')
         = moduleController sig_sr_bitError sig_sr_testCryptoMode
    -- interfaces
    sig_ct_powerMode = sync2CTInterface sync2CTClockTime sig_sr_powerMode
    sig_sr_Rx = sig_sdf_Rx
    sig_sdf_Tx = sig_sr_Tx
```

Chapter 3

The module `Transceiver`

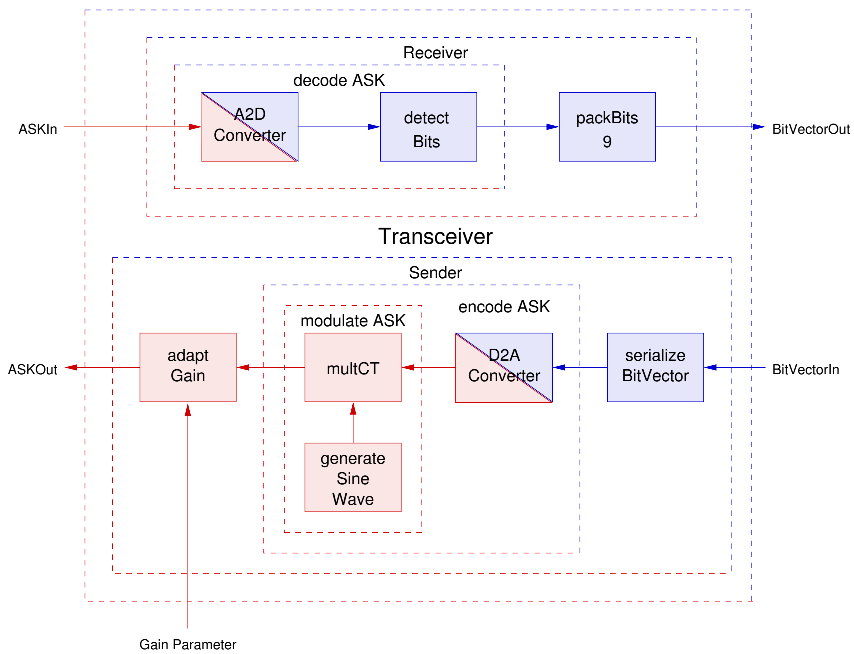

Figure 3.1: The structure of the Transceiver module

The transceiver contains a receiver and a sender. The receiver receives an ASK-signal of the continuous time domain and outputs a signal of bitvectors that is modeled in the untimed domain. The sender conducts the opposite operation, but in addition, it also adapts the gain based on the input of a control signal.

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Transceiver
    where
```

```haskell
-- import ForSyDeMoCLib
-- import CTLib
-- import BitVector
-- import FilterLib

import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib
import ForSyDe.Shallow.BitVector
import ForSyDe.Shallow.FilterLib

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities

transceiver :: -- (Fractional a) =>
               Signal (SubsigCT Double) -- Input: ASK Signal (to receiver)
             -> Signal (Vector Integer) -- Input: Signal of bitvectors
                                       --        (to sender)
             -> Signal ((Rational,Rational),
                        (Rational -> Double)
                        -> Rational -> Double)
                                       -- Input: Signal of analog functions
                                       -- (to sender)
            -> Signal Double           -- Adaptive threshold value
            -> (Signal (Vector Integer),
                        Signal (SubsigCT Double),
                        Signal (SubsigCT Double))
                                       -- Output: (Signal of bitvectors
                                       --         (from receiver),
                                       --          ASK Signal (from sender))
transceiver sig_ct_waveReceived sig_sdf_Tx sig_ct_powerMode sig_sdf_thresh' =
                                    (sig_sdf_Rx, sig_ct_lpout, sig_ct_waveSent)
    where (sig_sdf_Rx,sig_ct_lpout) = receiver sig_ct_waveReceived sig_sdf_thresh'
          sig_ct_waveSent = sender sig_ct_powerMode sig_sdf_Tx

receiver :: -- (Fractional a) =>
             Signal (SubsigCT Double)         -- Input: ASK Signal
         ->  Signal Double                    -- Adaptive threshold value
         -> (Signal (Vector Integer),Signal (SubsigCT Double)) -- Output
receiver sig_ct_waveReceived sig_sdf_thresh' = (sig_sdf_Rx,sig_ct_lpout)
    where
      (sig_sdf_Rx, sig_ct_lpout) =
            (packBits 9 $ snd $ decodeASK sig_ct_waveReceived sig_sdf_thresh',
             fst $ decodeASK sig_ct_waveReceived sig_sdf_thresh')

sender :: -- (Fractional a) =>
          Signal ((Rational,Rational),(Rational -> Double) -> Rational -> Double)
                                  -- Input: Signal of analog functions
       -> Signal (Vector Integer) -- Input: Signal of bitvectors
       -> Signal (SubsigCT Double)         -- Output: ASK Signal
```

```
sender sig_ct_powerMode sig_sdf_Tx = sig_ct_waveSent
    where sig_ct_waveSent = adaptGain sig_ct_powerMode sig_ct_wave
          sig_ct_wave = -- mapSY toRational $
                 encodeASK period_bit radian_sin timeInterval sig_sdf_bit
          sig_sdf_bit = mapU 1 (fromVector . strip) sig_sdf_Tx
```

The process adaptGain is an adaptive process that amplifies an input signal according to the gain signal.

```
adaptGain :: -- (Fractional a) =>
          Signal ((Rational,Rational),(Rational -> Double) -> Rational -> Double)
                                   -- Input: Signal of analog functions
          -> Signal (SubsigCT Double) -- Input: Analog signal
          -> Signal (SubsigCT Double) -- Output: Analog signal
adaptGain gainSignal ctSignal = pGainApplyfCT gainSignal ctSignal

pGainApplyfCT :: -- (Fractional a) =>
                  Signal ((Rational, Rational),
                          (Rational -> Double) -> Rational -> Double)
               -> Signal (SubsigCT Double)
               -> Signal (SubsigCT Double)
pGainApplyfCT = applyfCT sync2CTClockTime -- plotStepSize
```

The process encodeASK takes a signal of bits as input and two parameters, which describe the length of the period in seconds for one bit and the frequency of the sine carrier wave in Hertz as input and produces the ASK-signal for the given time interval.

```
encodeASK period_bit freq_sin timeInterval sig_sdf_bit = sig_ct_wave
    where
        sig_ct_wave = modulateASK sig_ct_sine sig_ct_convBit
        sig_ct_convBit =
          d2aConverter DAhold period_bit (mapSY fromInteger sig_sdf_bit)
        sig_ct_sine = generateSineWave freq_sin timeInterval

generateSineWave freq_sin timeInterval
    = signal [SubsigCT (sineFunction, timeInterval)]
      where
         sineFunction x = sin (fromRational $ x * freq_sin)

modulateASK sig_ct_sine sig_ct_convBit = multCT sig_ct_sine sig_ct_convBit
```

The process decodeASK takes a continuous time input signal and converts it to a signal of bits. It consists of two parts. The process a2dConverter converts the analog signal into a digital signal, where a single bit is represented by samplesPerBit values. The process detectBitLevel calculates the level of one bit out of samplesPerBit values.

```
decodeASK :: -- (Fractional a) =>
             Signal (SubsigCT Double) -- Input signal (continuous time)
          ->  Signal Double                    -- Adaptive threshold value
```

```
          -> (Signal (SubsigCT Double),Signal Integer)  -- Output signal
decodeASK sig_ct_waveReceived sig_sdf_thresh' =
    (sig_ct_lpout,zipWithU 1 samplesPerBit detectBitLevel sig_sdf_thresh
         $ a2dConverter (a2dResolution)
         $ sig_ct_lpout)
  where
    sig_ct_lpout = sLinearFilter RK4 a2dResolution' [1] filterDemCoef $
                                 absCT sig_ct_waveReceived
    sig_sdf_thresh = comb_sdf_adaptThresh sig_sdf_thresh'

comb_sdf_adaptThresh = combU 1 adaptThreshF
  where
    adaptThreshF [x] = repeatN numberOfBits x
```

Chapter 4

The module `Controller`

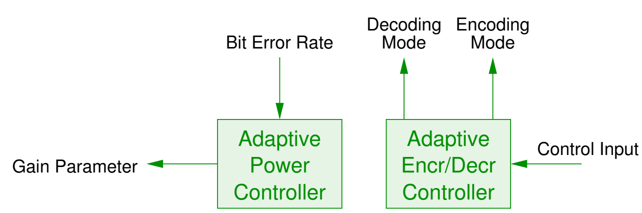

Figure 4.1: The Structure of the Controller module

The `Controller` is modelled in the synchronous domain. There is one input to control the encoding and decoding algorithms. The bit error rate input will be analysed by the adaptive power controller to set the gain, which is an input to the sender in the transceiver module.

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Controller
    where

-- import ForSyDeMoCLib
-- import CTLib

import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters

moduleController ::
     Signal Integer
  -> Signal Integer
  -> (Signal Integer,
      Signal ((Rational -> Double) -> Rational -> Double),
      Signal Double)
```

```
moduleController sig_sr_bitError sig_sr_testCryptoMode =
     (sig_sr_cryptoMode, sig_sr_powerMode,sig_sdf_thresh)
  where
    -- Encoding/decoding mode signal
    sig_sr_cryptoMode = sig_sr_testCryptoMode
    -- Power mode signal
    sig_sr_powerMode = adaptivePowerController ((signal [0])  +-+ sig_sr_bitError )
    -- Threshold mode signal
    sig_sdf_thresh = --  signal $ repeat threshVal
            adaptiveThresholdController ((signal [0])  +-+ sig_sr_bitError )

-- Some adaptive implementations:
--    Encoding function signal
--     (Signal ([Vector Integer] -> [Vector Integer]),
--    encodingAlgorithm' = adaptiveEncController controllInput
--    Decoding function signal
--      Signal ([Vector Integer] -> [Vector Integer]),
--    decodingAlgorithm' = adaptiveDecController controllInput
```

Chapter 5

The module `Encryption and Decryption`

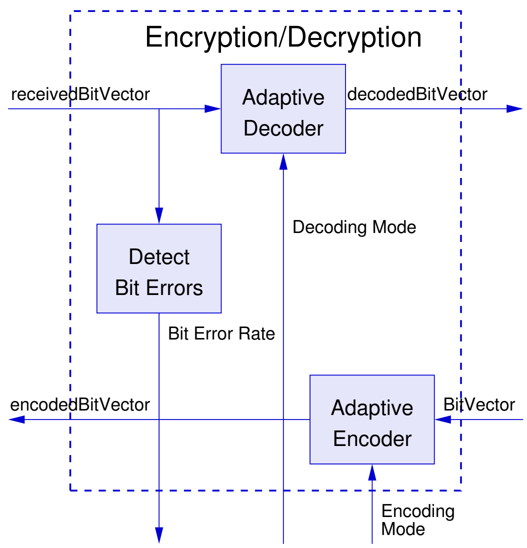

Figure 5.1: The Structure of the Encryption and Decryption module

The main module for encryption and decryption module in SDF domain.

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.EncDec
    where

-- import ForSyDeMoCLib
-- import AdaptivityLib
-- import CTLib
```

```
-- import BitVector

import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib
import ForSyDe.Shallow.AdaptivityLib
import ForSyDe.Shallow.BitVector

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities

moduleEncDec sig_sr_Rx sig_sr_testIn sig_sr_cryptoMode
                 = (sig_sr_bitError, sig_sr_testOut, sig_sr_Tx)
  where
    sig_sr_bitError = detectBitErrors sig_sr_Rx
    sig_sr_testOut = adaptiveDecoder sig_sr_Rx
    sig_sr_Tx = adaptiveEncoder sig_sr_testIn
    -- Processes used
    adaptiveDecoder = pAdaptiveSR (adaptiveDecController sig_sr_cryptoMode)
                                                . pBitsStripper
    adaptiveEncoder = pEvenParityWrapper . pAdaptiveSR
                                  (adaptiveEncController sig_sr_cryptoMode)
```

The processes used by this module.
The process pBitsStripper is to strip off the even parity bit from the 9-bit bitVector Signal.

```
pBitsStripper :: Signal (Vector Integer) -> Signal (Vector Integer)
pBitsStripper =  mapSY removeParityBit
-- pBitsStripper = combU 1 $ wrap . removeParityBit . strip
```

The process pEvenParityWrapper is to wrap an even parity bit in the head for the 8-bit bitVector Signal.

```
pEvenParityWrapper :: Signal (Vector Integer) -> Signal (Vector Integer)
pEvenParityWrapper = mapSY $ addParityBit Even
-- pEvenParityWrapper = combU 1 $ wrap . addParityBit Even . strip
```

The process bitErrorRate is to validation of the even parity bit in the head for the 9-bit bitVector Signal.

```
detectBitErrors :: Signal (Vector Integer) -> Signal Integer
detectBitErrors = mapSY isEvenParity'
-- detectBitErrors = combU 1 $ wrap . isEvenParity' . strip
  where
    isEvenParity' x | isEvenParity x = 0
                    | otherwise = 1
```

The process pAdaptiveSR is the adaptive process, which applies the functions defined in the first signal on the second input signal.

```
pAdaptiveSR :: Signal (a->b) -> Signal a -> Signal b
pAdaptiveSR = applyfSY

-- Some adaptive implementations:
```

```
--    adaptiveDecoder = pAdaptiveSDF decodingAlgorithm . pBitsStripper
--    adaptiveEncoder = pEvenParityWrapper . pAdaptiveSDF encodingAlgorithm
```

Chapter 6

The module `Main`

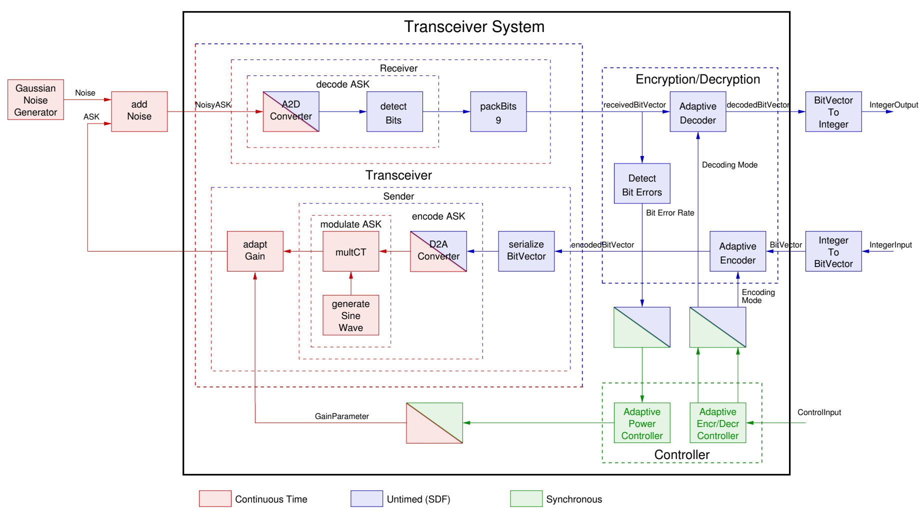

Figure 6.1: The Structure of the module `Main`

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Main where

import System.IO
import System.IO.Unsafe


-- import ForSyDeMoCLib
-- import CTLib
-- import FilterLib
-- import BitVector
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Gaussian
```

```
import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib
import ForSyDe.Shallow.FilterLib
import ForSyDe.Shallow.BitVector
```

```
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Transceiver
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Controller
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.EncDec
import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.TransceiverSystem
```

The main function for TestBench module in CT, synchronous and SDF domain.
Type "main" in GHCI to run the test.

```
main = do
   putStrLn "Testing␣..."
   putStrLn "Input␣signal␣of␣integers"
   putStrLn ("input␣=␣" ++ (show $ integerInput))
   putStrLn "Output␣signal␣of␣integers"
   putStrLn ("output␣=␣" ++ (show $ integerOutput))

   plotCT' plotStepSize [(subCT sig_ct_waveReceived sig_ct_waveSentAttenuated, "sig_ct_noise")]
   plotCT' plotStepSize [(sig_ct_waveSent, "sig_ct_waveSent")]
   plotCT' plotStepSize [(sig_ct_waveReceived, "sig_ct_waveReceived")]
   plotCT' plotStepSize [(sig_ct_lpout, "sig_ct_lpout")]
   putStrLn "Done!"

{-
-}
sig_sdf_lpin = a2dConverter dataPeriod sig_ct_waveReceived
sig_sdf_lpout = a2dConverter dataPeriod sig_ct_lpout
```

To test the transceiver system.

```
integerInput = mapSY bitVectorToInt sig_sr_testIn
integerOutput = mapSY bitVectorToInt sig_sr_testOut

(sig_sr_testOut, sig_ct_waveSent, sig_ct_lpout,
 sig_sr_Rx, sig_sdf_Tx, sig_sr_bitError) =
       transceiverSystem sig_ct_waveReceived sig_sr_testIn sig_sr_testCryptoMode
```

Sub-module 1 of the testbench is just to generate the stimuli signal which will be sent into the Encryption module. It is in SDF domain.

```
sig_sr_testIn = mapSY (intToBitVector 8) $ signal integerList
```

Sub-module 2 of the testbench is to initialize the signal to control the enc-dec algorithms. It is in synchronous domain.

```
sig_sr_testCryptoMode :: Signal Integer
sig_sr_testCryptoMode = signal (zeros3++ones5++zeros2)
  where
    zeros2 = take 2 $ repeat 0
    zeros3 = take 3 $ repeat 0
    ones5 = take 5 $ repeat 1
```

Sub-module 3 of the testbench is to add Gaussian noise into the ouput from the ASK Sender. The new signal with noise will be sent back into the ASK receiver. The 'sigVarivance' with possible variable variances is used to generate the gaussian noise.

```
sig_ct_waveReceived = addCT sig_ct_waveSentAttenuated sig_ct_noise
-- Here is the signal called 'noisy_signal' in SystemC group.
sig_sdf_waveReceived = a2dConverter a2dResolution'  sig_ct_waveReceived

-- To attenuate the ouput signal from ASK
sig_ct_waveSentAttenuated = scaleCT attenuation sig_ct_waveSent
-- The gaussian noise
sig_ct_noise = d2aConverter DAlinear a2dResolution' sig_GaussianDouble2 -- gaussianNoise
--gaussianNoise =  gaussianNoiseGen 0.0 sigVarivance myRandDouble0
sig_GaussianDouble2 = zipWithU 1 2 (gaussianF2 0 ) sigVarivance sig_myRand

-- The variance of the gaussian noise
sigVarivance | change_variance = signal $ repeatN change_var_start gaussianVar
                                   ++ repeatN (change_var_end- change_var_start)
                                              (gaussianVar*change_var_factor)
                                   ++ repeat gaussianVar
             | otherwise = signal $ repeat gaussianVar
```

Chapter 7

The module `Utilities`

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Utilities where

-- import CTLib
-- import ForSyDeMoCLib
-- import BitVector

import ForSyDe.Shallow
import ForSyDe.Shallow.CTLib
import ForSyDe.Shallow.BitVector

import ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters

-- |'subCT' adds two input signals together.
subCT :: (Show a, Num a) =>
         Signal (SubsigCT a) -- ^The first input signal
      -> Signal (SubsigCT a) -- ^The second input signal
      -> Signal (SubsigCT a) -- ^The output signal
subCT s1 s2 = applyF2 f s1' s2'
    where (s1',s2') = cutEq s1 s2
          f g1 g2 = f'
              where f' x = (g1 x) - (g2 x)


constDoubleF :: Double -> Double -> Double
constDoubleF = (\x y->x)

linearDoubleF :: Double -> Double -> Double -> Double -> Double -> Double
linearDoubleF c holdT m n x = (1-alpha)*m + alpha*n
  where alpha = (x-holdT)/c
```

The function detectBitLevel is used to detect the level of a bit in the ASK signal. The algorithm output a '1', if more that 3 values are larger than the noise threshold value `noiseThreshVal`. The algorithm should be checked with TU Vienna!

```
detectBitLevel :: (Num a, Ord a)=> [a] -> [a] -> [Integer]
detectBitLevel  [x0] [x1,x2]| x2 > x0 = [1]
                            | otherwise = [0]
{-
detectBitLevel x0 xs | sum ( map (isOne x0) xs) > detectOneNum = [1]
                     | otherwise = [0]
  where
    isOne :: (Num a, Ord a) => a -> a -> Int
    isOne x0 x | (abs x) > x0 = 1
            | otherwise = 0
-}
```

The functions wrap converts a signal value to a singleton list, and strip makes the opposite operation.

```
wrap = \x -> [x]
strip [x] = x
```

The process applyfCT is an adptive process, where the functionality is controlled by a signal op parameters. The input and output signal are continuous time signals. The first parameter gives the stepsize.

```
applyfCT :: (Show a, Num a) => Rational
         -> Signal ((Rational, Rational), (Rational -> a) -> Rational -> a)
         -> Signal (SubsigCT a)-> Signal (SubsigCT a)
applyfCT c NullS _ = NullS
applyfCT c _ NullS = NullS
applyfCT c (f:-fs) ss = combCTInterval c f (takeCT st'' ss)
                         +-+ applyfCT c (fs) (dropCT st' ss)
  where
    t' = stopT' f
    s' = startT' f
    st' = t' - s'
    st'' = st'
    stopT' ((_,t),_) = t
    startT' ((s,_),_) = s

combCTInterval :: (Show a, Num a) => Rational
               -> ((Rational,Rational),(Rational -> a) -> (Rational -> a))
               -> Signal (SubsigCT a)-> Signal (SubsigCT a)
combCTInterval c ((s,t), f) NullS = NullS
combCTInterval c ((s,t), f) ss
               | (duration (takeCT c ss)) < c = NullS
               | tStart' < s = (takeCT c ss) +-+
                                combCTInterval c ((s,t), f) (dropCT c ss)
               | tStart' > t = NullS
               | otherwise = applyF1 f (takeCT (t-tStart') ss)
  where
    tStart' = startTime $ takeCT c ss
```

The process adaptiveEncController is to generate the encoding algorithms. It uses the input signal to control the selection of the output encoding functions.

```
adaptiveEncController :: Signal Integer
                -> Signal (Vector Integer->Vector Integer)
adaptiveEncController NullS = NullS
adaptiveEncController (x:-xs) | x == 1 = encAlgorithm1 :- adaptiveEncController xs
                       | otherwise = encAlgorithm2 :- adaptiveEncController xs
  where
    -- DES
    encAlgorithm1 xs = zipWithV togglingBit xs (vector [1,0,1,0,1,0,1,0])
    -- Blowfish
    encAlgorithm2 xs = zipWithV togglingBit xs (vector [0,1,0,1,0,1,0,1])

togglingBit :: Integer -> Integer -> Integer
togglingBit 0 0 = 0
togglingBit 1 0 = 1
togglingBit 0 1 = 1
togglingBit 1 1 = 0
```

The process `adaptiveDecController` is to generate the decoding algorithms. It uses the input signal to control the selection of the output decoding functions.

```
adaptiveDecController :: Signal Integer
                 -> Signal (Vector Integer->Vector Integer)
adaptiveDecController NullS = NullS
adaptiveDecController (x:-xs) | x == 1 = decAlgorithm1 :- adaptiveDecController xs
                       | otherwise = decAlgorithm2 :- adaptiveDecController xs
  where
    -- DES
    decAlgorithm1 xs = zipWithV togglingBit xs (vector [1,0,1,0,1,0,1,0])
    -- Blowfish
    decAlgorithm2 xs = zipWithV togglingBit xs (vector [0,1,0,1,0,1,0,1])
```

The process `pAdaptivePowerF` is to generate the power adjustment algorithms. It uses the input signal to control the selection of the different amplititude gain functions to reflect voltage changes.

```
adaptivePowerController :: Signal Integer
                        -> Signal ((Rational -> Double) -> Rational -> Double)
adaptivePowerController = mapSY powerF
  where
    powerF i | i==0 = fVL
             | otherwise = fVH
    fVH = \f x -> gainPower * (f x)
    fVL = id

adaptiveThresholdController :: Signal Integer
                            -> Signal Double
adaptiveThresholdController = mapSY threshF
  where
    threshF i | i==0 = threshVal
             | otherwise = gainThreshold * threshVal
```

`sync2CTInterface` is an interface of the signal from the SR domain to CT domain.

```
sync2CTInterface :: Rational   -- timestep
                 -> Signal a -- A signal in SR domain
                 -> Signal ((Rational, Rational), a) -- A signal in CT domain
sync2CTInterface c xs = mealySY g f (0::Rational) xs
  where
    g :: Rational -> a -> Rational
    g x _ = x +c
    f :: Rational -> a -> ((Rational, Rational), a)
    f x y = ((x,x+c), y)
```

`packBits` is to pack the bits in the signal into bit-vector.

```
packBits :: Int -> Signal a -> Signal (Vector a)
packBits n bitSignal = mapU n (wrap . vector) bitSignal
```

A Reg list with finite bit size.

```
finiteRegList :: Int -- The specified space of the Fifo
               -> [a] -- Previous Fifo state
               ->  a  -- New input
               -> [a] -- Current Fifo state
finiteRegList n xs y = take n (y:xs)
```

Some helper functions

```
atL n = head . drop (n-1)
-- replaceL n y xs = take (n-1) xs ++ [y] ++ drop n xs
repeatN n = take n . repeat

foldlSY :: (a->b->a) -> a -> Signal b -> a
foldlSY f z NullS     = z
foldlSY f z (x:-xs) = foldlSY f (f z x) xs
```

/

Chapter 8

The Module Parameters

The module `Parameters` gathers the parameters used in the ASK transceiver system example.

```
module ForSyDe.Shallow.Example.Heterogeneous.ASKTransceiver.Parameters where

import Data.Ratio

freq_byte = 1000  -- 0.001 MHZ
period_byte = 1%freq_byte -- Clock period of one byte

freq_bit = 9000  -- 0.009 MHZ,
period_bit =  1%freq_bit -- Clock period of one bit in the digital domain

-- Frequency of ASK sine signal is 10MHZ
freq_ASK = 100000 -- 00 -- 1e7
radian_sin = toRational $ 2*pi* (fromInteger freq_ASK)

-- The attenuation of the ouput analog signal as it is sent back by the TB
attenuation = 0.0001 :: Double

-- Threshold value of detect one
threshVal = 0.000045

-- It is used to only detect the 2nd sample, which is in the middle of the bit
a2dResolution = 1/2 * period_bit
samplesPerBit = 2 :: Int

-- Special stuff to trigger bit errors during simulation
change_variance = True
change_var_start = 3 * dataRatePerByte :: Int -- ms
change_var_end = 4* dataRatePerByte :: Int -- ms
change_var_factor = 10.0 -- 1.04 -- 1.07 -- 1.08 -- 1.1  1.04

gaussianVar = 1e-9 -- 3e-9
```

```haskell
freq_cutoff = 3.0*9.0* fromIntegral freq_byte
-- Numberator coefficients of the s-filter is always [1].
-- Denominator coefficients of the s-filter
filterDemCoef :: (Fractional a) => [a]
filterDemCoef = map realToFrac
                [1/(2*pi*freq_cutoff),1]

-- One clock cycle in synchronous domain corresponds to 1ms in CT domain
numberOfBits = 9 :: Int         -- number of Bits in a byte
sync2CTClockTime = (toRational numberOfBits ) * period_bit

-- Gain of the higher power mode
gainPower = 5.0 :: Double
gainThreshold = 2.5 :: Double -- gainPower

----------------------------------------------
-- Below are some parameters we used different values


----------------------------------------------
-- Below are some parameters especially used in Our model
-- The input data for test.
integerList = [0..5]

-- Time Interval for the sine source
timeInterval = (0%10,100%1000)

-- Resolution for graphical plots
plotStepSize = 2%100000 -- 000

-- Sampling period of the filter
-- Now it is 2e-8, that means for each sine period there are 5 samples
a2dResolution' = -- 2%100000000
                 1%dataRate -- 199998000
dataRatePerBit = (floor $ 20 * fromInteger freq_ASK / fromInteger freq_bit) :: Int
dataRatePerByte = 9 * dataRatePerBit :: Int
dataRate = freq_bit * fromIntegral dataRatePerBit
dataPeriod = 1 % dataRate
```