

# The ForSyDe Standard Library<sup>1</sup>

I. Sander      A. Jantsch      A. K. Singh  
                  T. Raudvere  
Royal Institute of Technology, Sweden

November 19, 2001

<sup>1</sup>Version 2.0

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Overview . . . . .	4
<b>2</b>	<b>The module ForSyDeStdLib</b>	<b>5</b>
2.1	Overview . . . . .	5
<b>3</b>	<b>The module AbsentExt</b>	<b>6</b>
3.1	Overview . . . . .	6
3.2	Functions on the data type AbsentExt . . . . .	6
3.3	Implementation of library functions . . . . .	6
<b>4</b>	<b>The module Signal</b>	<b>8</b>
4.1	Overview . . . . .	8
4.2	The data type Signal . . . . .	8
4.3	Functions on the data type Signal . . . . .	8
4.4	Implementation of the library functions . . . . .	9
4.4.1	Overloading of Show and Read for the data type Signal . . . . .	9
4.4.2	Functions on the data type Signal . . . . .	10
<b>5</b>	<b>The module SynchronousLib</b>	<b>11</b>
5.1	Overview . . . . .	11
5.2	Skeletons for Combinatorial Processes . . . . .	11
5.3	Skeletons for Sequential Processes . . . . .	12
5.4	Processes . . . . .	13
5.5	Implementation of Library Functions . . . . .	13
5.5.1	Skeletons for Combinatorial Processes . . . . .	13
5.5.2	Skeletons for Sequential Processes . . . . .	14
5.5.3	Processes . . . . .	15
<b>6</b>	<b>The module SynchronousProcessLib</b>	<b>17</b>
6.1	Overview . . . . .	17
6.2	Processes . . . . .	17
6.3	Implementation of Processes . . . . .	18
<b>7</b>	<b>The module DataflowLib</b>	<b>19</b>
7.1	Overview . . . . .	19
7.2	Data Types . . . . .	19
7.3	Skeletons . . . . .	20
7.3.1	Skeletons for Combinatorial Processes . . . . .	20
7.3.2	Skeletons for Sequential Processes . . . . .	20
7.4	Implementation . . . . .	21
7.4.1	Combinatorial Skeletons . . . . .	21
7.4.2	Sequential Skeletons . . . . .	22

7.4.3	Supporting Functions . . . . .	23
<b>8</b>	<b>The module UntimedLib</b>	<b>26</b>
8.1	Overview . . . . .	26
8.2	Skeletons . . . . .	26
8.2.1	Skeletons for Combinatorial Processes . . . . .	26
8.2.2	Skeletons for Sequential Processes . . . . .	26
8.2.3	Zip and Unzip based Skeletons . . . . .	27
8.2.4	Source and Sink Skeletons . . . . .	28
8.3	Helper Functions . . . . .	28
<b>9</b>	<b>The module DiscreteEventLib</b>	<b>29</b>
9.1	Overview . . . . .	29
9.2	Data Types . . . . .	29
9.3	Skeltons . . . . .	30
9.4	Skeltons for combinatorial processes . . . . .	30
9.5	Skeltons for sequential processes . . . . .	31
9.6	implementation . . . . .	33
9.6.1	Combinatorial Skeltons . . . . .	33
9.6.2	Sequential Skeltons . . . . .	34
9.6.3	supporting function . . . . .	36
<b>10</b>	<b>The module StochasticLib</b>	<b>40</b>
<b>11</b>	<b>The Stochastic Library</b>	<b>41</b>
11.1	Overview . . . . .	41
11.2	Skeletons for Stochastic Porcesses . . . . .	41
11.3	Implementation of Skeletons . . . . .	41
11.3.1	Skeletons for Stochastic Porcesses . . . . .	41
11.4	Supporting Functions . . . . .	42
<b>12</b>	<b>The module FiringRules</b>	<b>44</b>
12.1	Overview . . . . .	44
<b>13</b>	<b>The module Vector</b>	<b>45</b>
13.1	Overview . . . . .	45
13.2	The Data Type Vector . . . . .	45
13.3	Functions on the Data Type vector . . . . .	45
13.4	Implementation . . . . .	48
13.4.1	The Data Type vector . . . . .	48
13.4.2	Functions on the Data Type Vector . . . . .	48
<b>14</b>	<b>The module Memory</b>	<b>51</b>
14.1	Overview . . . . .	51
14.2	Data Structure . . . . .	51
14.3	Functions on the data type Memory . . . . .	51
14.4	Implementation of Functions . . . . .	52
<b>15</b>	<b>The module Queue</b>	<b>53</b>
15.1	Overview . . . . .	53
15.2	The data type Queue . . . . .	53
15.3	Functions on the data types Queue and FiniteQueue . . . . .	53
15.4	Implementation . . . . .	54

<b>16 Possible Extensions to the ForSyDe Standard Library</b>	<b>55</b>
16.1 Overview . . . . .	55

# Chapter 1

## Introduction

### 1.1 Overview

The ForSyDe standard library ForSyDeStdLib provides data types and functions to model systems according to the ForSyDe methodology.

The structure of the library is shown in Figure 1.1.

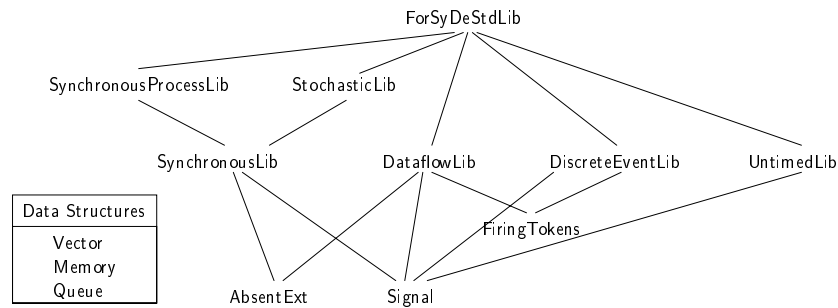


Figure 1.1: The Structure of the ForSyDeStdLib

The document presents version 2.0, but the library is under development. However, we try to keep new versions of the library as compatible with older versions as possible.

Additional application libraries can be built on top of this library.

## Chapter 2

# The module **ForSyDeStdLib**

### 2.1 Overview

The ForSyDe Standard Library contains the data types and functions for the ForSyDe design methodology.

The module ForSyDeStdLib works as a container and exports the other libraries.

```
module ForSyDeStdLib(  
    module StochasticLib,  
    module SynchronousProcessLib,  
    module SynchronousLib, module DataflowLib,  
    module DiscreteEventLib, module UntimedLib,  
    module Vector, module Signal, module Memory,  
    module AbsentExt, module Queue)  
    where  
  
    import StochasticLib  
    import UntimedLib  
    import DataflowLib  
    import SynchronousProcessLib  
    import SynchronousLib  
    import DiscreteEventLib  
    import Vector  
    import Signal  
    import Memory  
    import AbsentExt  
    import Queue  
    import FiringRules
```

## Chapter 3

# The module **AbsentExt**

### 3.1 Overview

The module `AbsentExt` is used to extend existing data types with the value "absent" ( $\perp$ ).

```
module AbsentExt (  
    AbstExt (Abst, Prst), fromAbstExt, abstExt,  
    isAbsent, isPresent)  
  where
```

The data type `AbstExt` has two constructors. The constructor `Abst` is used to model the absence of a value, while the constructor `Prst` is used to model present values.

```
data AbstExt a = Abst  
              | Prst a deriving (Eq)
```

The data type `AbstExt` is defined as an instance of `Show` and `Read`. `'⊥'` represents the value `Abst` while a present value is represented with its value, e.g. `Prst 1` is represented as `'1'`.

### 3.2 Functions on the data type **AbsentExt**

The module defines the following functions:

```
abstExt      :: a → AbstExt a  
fromAbstExt :: a → AbstExt a → a  
isPresent   :: AbstExt a → Bool  
isAbsent    :: AbstExt a → Bool
```

The function `abstExt` converts a value into an extended value. The function `fromAbstExt` converts a value from a extended value.

The functions `isPresent` and `isAbsent` check for the presence or absence of a value.

### 3.3 Implementation of library functions

```
instance Show a ⇒ Show (AbstExt a) where  
  showsPrec _ x = showsAbstExt x  
  
showsAbstExt Abst = (++) "⊥"  
showsAbstExt (Prst x) = (++) (show x)
```

```

instance Read a => Read (AbstExt a) where
  readsPrec _ x      = readsAbstExt x

  readsAbstExt
  readsAbstExt s     :: (Read a) => ReadS (AbstExt a)
                    = [(Abst, r1) | ("_", r1) ← lex s]
                      ++ [(Prst x, r2) | (x, r2) ← reads s]

  abstExt v          = Prst v

  fromAbstExt x Abst = x
  fromAbstExt _ (Prst y) = y

  isPresent Abst     = False
  isPresent (Prst _) = True

  isAbsent           = not . isPresent

```

# Chapter 4

## The module **Signal**

### 4.1 Overview

The module `Signal` defines the data type `Signal` and functions operating on this data type.

```
module Signal( Signal (NullS, (:-)), (-:), (+++), (!-),
               signal, fromSignal,
               unitS, nullS, headS, tailS, atS, takeS, dropS,
               lengthS
             )
where

infixr 5      :-
infixr 5      -:
infixr 5      +++
infixr 5      !-
```

### 4.2 The data type **Signal**

A signal is defined as a list of events. An event has a tag and a value. The tag of an event is defined by the position in the list.

```
data Signal a = NullS
      | a :- Signal a deriving (Eq)
```

A signal is defined as an instance of the classes `Read` and `Show`. The signal `1 :- 2 :- NullS` is represented as `{1,2}`.

### 4.3 Functions on the data type **Signal**

The module defines the following on the data type `Signal`:

```
signal          :: [a] → Signal a
fromSignal      :: Signal a → [a]
unitS           :: a → Signal a
nullS           :: Signal a → Bool
headS           :: Signal a → a
tailS           :: Signal a → Signal a
atS             :: Int → Signal a → a
takeS           :: Int → Signal a → Signal a
dropS           :: Int → Signal a → Signal a
```

```

lengthS      :: Num a => Signal b -> a
(-:)         :: Signal a -> a -> Signal a
(+++)        :: Signal a -> Signal a -> Signal a

```

The functions `signal` and `fromSignal` convert a list into a signal and vice versa. The function `unitS` creates a signal with one value. The function `nullS` checks if a signal is empty. The function `headS` gives the first value - the head of a signal, while `tailS` gives the rest of the signal - the tail. The function `atS` gives returns the  $n$ -th event in a signal. The numbering of events in a signal starts with 0. There is also an operator version of this function, `(!)`. The function `takeS` returns the first  $n$  values of a signal, while the function `dropS` drops the first  $n$  values from a signal. New signals can be created by means of the following functions. The data constructor `(:-)` adds an element to the signal at the head of the signal. The function `lengthS` returns the length of a *finite* stream. The operator `(-)` adds an element to a signal at the tail. Finally the operator `(+++)` concatenates two signals into one signal.

The combinator `fanS` takes two processes `p1` and `p2` and generates a process network, where a signal is split and processed by the processes `p1` and `p2`.

```

fanS :: (Signal a -> Signal b) -> (Signal a -> Signal c)
      -> Signal a -> (Signal b, Signal c)

```

## 4.4 Implementation of the library functions

### 4.4.1 Overloading of **Show** and **Read** for the data type **Signal**

```

instance (Show a) => Show (Signal a) where
  showsPrec p NullS = showParen (p > 9) (
    showString "{}")
  showsPrec p xs    = showParen (p > 9) (
    showChar '{' . showSignal1 xs)
  where
    showSignal1 NullS
      = showChar '}'
    showSignal1 (x:-NullS)
      = shows x . showChar '}'
    showSignal1 (x:-xs)
      = shows x . showChar ','
        . showSignal1 xs

```

```

instance Read a => Read (Signal a) where
  readsPrec _ s = readsSignal s

```

```

readsSignal      :: (Read a) => ReadS (Signal a)
readsSignal s    = [((x:-NullS), rest)
  | ("{" , r2) <- lex s,
    (x, r3)   <- reads r2,
    ("}" , rest) <- lex r3]
++ [(NullS, r4)
  | ("{" , r5) <- lex s,
    ("}" , r4) <- lex r5]
++ [((x:-xs), r6)
  | ("{" , r7) <- lex s,
    (x, r8)   <- reads r7,
    ("," , r9) <- lex r8,
    (xs, r6)  <- readsValues r9]

```

```

readsValues      :: (Read a) => ReadS (Signal a)
readsValues s    = [((x:-NullS), r1)
                   | (x, r2) <- reads s,
                     ("}", r1) <- lex r2]
                ++ [((x:-xs), r3)
                   | (x, r4) <- reads s,
                     ("", r5) <- lex r4,
                     (xs, r3) <- readsValues r5]

```

#### 4.4.2 Functions on the data type **Signal**

```

signal []          = NullS
signal (x:xs)     = x :- signal xs

fromSignal NullS  = []
fromSignal (x:-xs) = x : fromSignal xs

unitS x           = x :- NullS

nullS NullS      = True
nullS _          = False

headS NullS      = error "headS: Signal is empty"
headS (x:-_)    = x

tailS NullS      = error "tailS: Signal is empty"
tailS (_:-xs)    = xs

atS _ NullS      = error "atS: Signal has not enough elements"
atS 0 (x:-_)    = x
atS n (_:-xs)    = atS (n-1) xs

(!-) xs n        = atS n xs

takeS 0 _        = NullS
takeS _ NullS    = NullS
takeS n (x:-xs) | n <= 0 = NullS
                 | otherwise = x :- takeS (n-1) xs

dropS 0 NullS    = NullS
dropS _ NullS    = NullS
dropS n (x:-xs) | n <= 0 = x:-xs
                 | otherwise = dropS (n-1) xs

(-:) xs x        = xs ++ (x :- NullS)

(+++) NullS ys   = ys
(+++) (x:-xs) ys = x :- (xs +++ ys)

lengthS NullS    = 0
lengthS (_:-xs) = 1 + lengthS xs

fanS p1 p2 xs    = (p1 xs, p2 xs)

```

# Chapter 5

## The module **SynchronousLib**

### 5.1 Overview

The synchronous library `SynchronousLib` defines skeletons and processes for the synchronous computational model. A skeleton is a higher order function which together with combinatorial function(s) and values as arguments constructs a process. Thus a skeleton can also be viewed as a *process constructor*.

```
module SynchronousLib(  
    module Vector, module Signal, module AbsentExt,  
    module Memory, mapSY,  
    zipWithSY, zipWith3SY, zipWith4SY, scan1SY,  
    scanl2SY, scanl3SY, scanlDelaySY, scanlDelay2SY,  
    scanlDelay3SY, delaySY, delaynSY, whenSY,  
    fillSY, holdSY, zipSY, zip3SY, unzipSY,  
    unzip3SY, zipxSY, unzipxSY, mapxSY, mooreSY,  
    moore2SY, moore3SY, mealySY, mealy2SY,  
    mealy3SY, fstSY, sndSY, groupSY  
) where  
  
import Signal  
import Vector  
import AbsentExt  
import Memory
```

### 5.2 Skeletons for Combinatorial Processes

Combinatorial processes do not possess an internal state, so that the output only depends on input signals.

The module includes the following skeletons for combinatorial processes:

```
mapSY      :: (a → b) → Signal a → Signal b  
zipWithSY  :: (a → b → c) → Signal a → Signal b → Signal c  
zipWith3SY :: (a → b → c → d) → Signal a → Signal b  
           → Signal c → Signal d  
zipWith4SY :: (a → b → c → d → e) → Signal a → Signal b  
           → Signal c → Signal d → Signal e  
mapxSY     :: (a → b) → Vector (Signal a) → Vector (Signal b)
```

The skeleton `mapSY` takes a combinatorial function as argument and returns a process with one input signal and one output signal. This is shown in the following, where `mapSY (+1)` is a process which increments all values of an input signal.

In a similar way `zipWithSY`, `zipWith3SY` and `zipWith4SY` apply a combinatorial function on a number of input signals.

The skeleton `mapxSY` creates a process network that maps a function onto all signals in a vector of signals.

### 5.3 Skeletons for Sequential Processes

Sequential processes have a local state. Skeletons that construct such processes take not only functions but also values as arguments to express the value of the local state of the process. The output of sequential processes is deterministic and depends on the initial state and the input signals.

The module includes the following skeletons for sequential processes:

```
scan1SY      :: (a → b → a) → a → Signal b → Signal a
scan12SY     :: (a → b → c → a) → a → Signal b → Signal c
              → Signal a
scan13SY     :: (a → b → c → d → a) → a → Signal b
              → Signal c → Signal d → Signal a
scan1DelaySY :: (a → b → a) → a → Signal b → Signal a
scan1Delay2SY :: (a → b → c → a) → a → Signal b → Signal c
              → Signal a
scan1Delay3SY :: (a → b → c → d → a) → a → Signal b
              → Signal c → Signal d → Signal a
mooreSY      :: (a → b → a) → (a → c) → a → Signal b → Signal c
moore2SY     :: (a → b → c → a) → (a → d) → a → Signal b
              → Signal c → Signal d
moore3SY     :: (a → b → c → d → a) → (a → e) → a → Signal b
              → Signal c → Signal d → Signal e
mealySY      :: (a → b → a) → (a → b → c) → a → Signal b
              → Signal c
mealy2SY     :: (a → b → c → a) → (a → b → c → d) → a
              → Signal b → Signal c → Signal d
mealy3SY     :: (a → b → c → d → a) → (a → b → c → d → e) → a
              → Signal b → Signal c → Signal d → Signal e
delaySY      :: a → Signal a → Signal a
delaynSY     :: a → Int → Signal a → Signal a
filterSY     :: (a → Bool) → Signal a → Signal (AbstExt a)
```

We define two different basic skeletons to construct sequential processes, `scan1SY` and `scan1DelaySY`. Both skeletons take a function `ns` and a state `m` as arguments. Both skeletons use the function `ns` to calculate the next state, but calculate the output in a different way. `scan1SY` behaves like the Haskell prelude function `scanl` and has the value of the new state as its output value, while `scan1DelSY` has the current state value as output. The following example exemplifies this:

```
SynchronousLib> scan1SY (+) 0 (signal [1,2,3,4])
{1,3,6,10} :: Signal Integer
SynchronousLib> scan1DelaySY (+) 0 (signal [1,2,3,4])
{0,1,3,6} :: Signal Integer
```

Skeletons like `scan12SY`, `scan12DelaySY` are used in the same way for several input signals.

The skeletons `mooreSY` and `mealySY` are used to model state machines. These skeletons are based on the skeleton `scan1DelaySY` as is naturally for state machines in hardware, that the output operates on the current state and not on the next state.

These skeletons take a function `ns` to calculate the next state, a function `o` to calculate the output and an value `m` for the initial state. In a process based on

the `mooreSY` skeleton the output function `o` operates only on the current state of the process. In contrast the output function of a process based on the `mealySY` skeleton operates on both the current state and the input.

The skeleton `delaySY` delays the signal one event cycle by introducing an initial value at the beginning of the output signal. The skeleton `delaynSY` delays the signal  $n$  events by introducing  $n$  identical default values.

The skeleton `filterSY` takes a predicate `p` and produces a process, that discards all values that do not fulfill the predicate `p`. In this case the output is  $\perp$ .

## 5.4 Processes

The module also contains the following synchronous processes:

```

whenSY      :: Signal (AbstExt a) → Signal (AbstExt b) → Signal (AbstExt a)
fillSY      :: AbstExt a → Signal (AbstExt a) → Signal (AbstExt a)
holdSY      :: AbstExt a → Signal (AbstExt a) → Signal (AbstExt a)
zipSY       :: Signal a → Signal b → Signal (a,b)
zip3SY      :: Signal a → Signal b → Signal c → Signal (a,b,c)
unzipSY     :: Signal (a,b) → (Signal a,Signal b)
unzip3SY    :: Signal (a, b, c) → (Signal a, Signal b, Signal c)
zipxSY      :: Vector (Signal a) → Signal (Vector a)
unzipxSY    :: Signal (Vector a) → Vector (Signal a)
groupSY     :: Int → Signal a → Signal (AbstExt (Vector a))
fstSY       :: Signal (AbstExt (a,b)) → Signal (AbstExt a)
sndSY       :: Signal (AbstExt (a,b)) → Signal (AbstExt b)

```

The skeleton `whenSY` creates a process that synchronizes a signal of timed values with another signal of timed values. The output signal has the value of the first signal whenever an event has a present value and  $\perp$  when the event has an absent value.

The skeleton `fillSY` creates a process that 'fills' a signal with timed values by replacing absent values with a given present value.

The skeleton `holdSY` creates a process that 'fills' a signal with timed values by replacing absent values by the preceding present value. Only in cases, where no preceding value exists, the absent value is replaced by a supplied present value.

The process `zipSY` 'zips' two incoming signals into one signal of tuples, while the process `unzipSY` 'unzips' a signal of tuples into two signals. The functions `zip3SY` and `unzip3SY` perform the corresponding function for three signals.

The process `zipxSY` 'zip' a signal of vectors into a vector of signals. The process `unzipxSY` 'unzips' a vector of signals into a signal of vectors.

The function `groupSY` groups values into a vector of size  $n$ , which takes  $n$  cycles. While the grouping takes place the output from this process consists of absent values.

The processes `fstSY` and `sndSY` select the always the first or second value from a signal of timed values of pairs.

## 5.5 Implementation of Library Functions

### 5.5.1 Skeletons for Combinatorial Processes

```

mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)

```

```

zipWithSY _ NullS _ = NullS
zipWithSY _ _ NullS = NullS

```

zipWithSY f (x:-xs) (y:-ys) = f x y :- (zipWithSY f xs ys)

zipWith3SY \_ NullS \_ \_ = NullS  
zipWith3SY \_ \_ NullS \_ = NullS  
zipWith3SY \_ \_ \_ NullS = NullS  
zipWith3SY f (x:-xs) (y:-ys) (z:-zs) = f x y z :- (zipWith3SY f xs ys zs)

zipWith4SY \_ NullS \_ \_ \_ = NullS  
zipWith4SY \_ \_ NullS \_ \_ = NullS  
zipWith4SY \_ \_ \_ NullS \_ = NullS  
zipWith4SY \_ \_ \_ \_ NullS = NullS  
zipWith4SY f (w:-ws) (x:-xs) (y:-ys) (z:-zs)  
= f w x y z  
:- (zipWith4SY f ws xs ys zs)

mapxSY f = mapV (mapSY f)

## 5.5.2 Skeletons for Sequential Processes

scan1SY \_ \_ NullS = NullS  
scan1SY f mem (x:-xs) = f mem x :- (scan1SY f newmem xs)  
**where** newmem = f mem x

scan12SY \_ \_ NullS \_ = NullS  
scan12SY \_ \_ \_ NullS = NullS  
scan12SY f mem (x:-xs) (y:-ys) = f mem x y  
:- (scan12SY f newmem xs ys)  
**where** newmem = f mem x y

scan13SY \_ \_ NullS \_ \_ = NullS  
scan13SY \_ \_ \_ NullS \_ = NullS  
scan13SY \_ \_ \_ \_ NullS = NullS  
scan13SY f mem (x:-xs) (y:-ys) (z:-zs)  
= f mem x y z :- (scan13SY f newmem xs ys zs)  
**where** newmem = f mem x y z

scan1DelaySY \_ \_ NullS = NullS  
scan1DelaySY f mem (x:-xs) = mem :- (scan1DelaySY f newmem xs)  
**where** newmem = f mem x

scan1Delay2SY \_ \_ NullS \_ = NullS  
scan1Delay2SY \_ \_ \_ NullS = NullS  
scan1Delay2SY f mem (x:-xs) (y:-ys) = mem :- (scan1Delay2SY f newmem xs ys)  
**where** newmem = f mem x y

scan1Delay3SY \_ \_ NullS \_ \_ = NullS  
scan1Delay3SY \_ \_ \_ NullS \_ = NullS  
scan1Delay3SY \_ \_ \_ \_ NullS = NullS  
scan1Delay3SY f mem (x:-xs) (y:-ys) (z:-zs)  
= mem :- (scan1Delay3SY f newmem xs ys zs)  
**where** newmem = f mem x y z

delaySY e es = e:-es

delaynSY e n xs | n <= 0 = xs  
| otherwise = e :- delaynSY e (n-1) xs

mooreSY nextState output initial  
= mapSY output . (scan1DelaySY nextState initial)

```

moore2SY nextState output initial inp1 inp2 =
  mapSY output (scanlDelay2SY nextState initial inp1 inp2)

moore3SY nextState output initial inp1 inp2 inp3 =
  mapSY output (scanlDelay3SY nextState initial inp1 inp2 inp3)

mealySY nextState output initial signal =
  zipWithSY output (scanlDelaySY nextState initial signal) signal

mealy2SY nextState output initial inp1 inp2 =
  zipWith3SY output (scanlDelay2SY nextState initial inp1 inp2)
  inp1 inp2

mealy3SY nextState output initial inp1 inp2 inp3 =
  zipWith4SY output (scanlDelay3SY nextState initial inp1 inp2 inp3)
  inp1 inp2 inp3

filterSY p NullS      = NullS
filterSY p (x:-xs)    = if (p x == True) then
                        Prst x :- filterSY p xs
                        else
                        Abst :- filterSY p xs

```

### 5.5.3 Processes

```

whenSY NullS _          = NullS
whenSY _ NullS          = NullS
whenSY (_:-xs) (Abst:-ys) = Abst :- (whenSY xs ys)
whenSY (x:-xs) (_:-ys)   = x     :- (whenSY xs ys)

fillSY a xs = mapSY (replaceAbst a) xs
              where replaceAbst a Abst    = a
                    replaceAbst _ (Prst x) = (Prst x)

holdSY a xs = scanlSY hold a xs
              where hold a Abst    = a
                    hold _ (Prst x) = Prst x

zipSY (x:-xs) (y:-ys) = (x, y) :- zipSY xs ys
zipSY _ _             = NullS

zip3SY (x:-xs) (y:-ys) (z:-zs) = (x, y, z) :- zip3SY xs ys zs
zip3SY _ _ _                 = NullS

unzipSY NullS          = (NullS, NullS)
unzipSY ((x, y):-xys) = (x:-xs, y:-ys) where (xs, ys) = unzipSY xys

unzip3SY NullS          = (NullS, NullS, NullS)
unzip3SY ((x, y, z):-xyzs) = (x:-xs, y:-ys, z:-zs) where
  (xs, ys, zs) = unzip3SY xyzs

zipxSY NullV          = NullS
zipxSY (NullS :> xss) = zipxSY xss
zipxSY ((x:-xs) :> xss) = (x :> (mapV headS xss))
  :- (zipxSY (xs :> (mapV tailS xss)))

unzipxSY NullS          = NullV

```

```

unzipxSY (NullV :- vss) = unzipxSY vss
unzipxSY ((v:>vs) :- vss) = (v :- (mapSY headV vss))
                             :-> (unzipxSY (vs :- (mapSY tailV vss)))

groupSY n xs = mooreSY (addElement n) (output n) (NullV, 0) xs
  where addElement m (vs, n) x | n < m = (vs <: x, n+1)
                                | n == m = (unitV x, 1)
                                | m /> n = Abst
        output m (vs, n) | m == n = Prst vs
                          | m /> n = Abst

fstSY = mapSY fst1
  where fst1 Abst = Abst
        fst1 (Prst (a, _)) = Prst a

sndSY = mapSY snd1
  where snd1 Abst = Abst
        snd1 (Prst (_, b)) = Prst b

```

# Chapter 6

## The module SynchronousProcessLib

### 6.1 Overview

The synchronous process library `SynchronousProcessLib` defines processes for the synchronous computational model. It is based on the synchronous library `SynchronousLib`.

```
module SynchronousProcessLib(  
    module SynchronousLib,  
    module Signal,  
    module AbsentExt,  
    fifoDelaySY, finiteFifoDelaySY,  
    memorySY, mergeSY, counterSY  
    ) where  
  
import SynchronousLib  
import Signal  
import AbsentExt  
import Queue
```

### 6.2 Processes

The library defines the following processes:

```
fifoDelaySY      :: Signal [a] → Signal (AbstExt a)  
finiteFifoDelaySY :: Int → Signal [a] → Signal (AbstExt a)  
memorySY        :: Int → Signal (Access a) → Signal (AbstExt a)  
mergeSY         :: Signal (AbstExt a) → Signal (AbstExt a)  
                → Signal (AbstExt a)  
counterSY       :: (Enum a, Ord a) ⇒ a → a → Signal a
```

The process `fifoDelaySY` implements a synchronous model of a FIFO with infinite size, while the process `finiteFifoDelaySY` implements a FIFO with finite size. Both FIFOs take a list of values at each event cycle and output one value. There is a delay of one cycle. The process `memorySY` implements a synchronous memory. It uses access functions of the type `Read adr` and `Write adr value`. The process `mergeSY` merges two input signals into a single signal. The process has an internal buffer in order to prevent loss of data. The process is deterministic and outputs events according to their time tag. If there are two valid values at on both signals. The value of the first signal is output first. The process `counter` implements a counter,

that counts from min to max. The process counterS has no input and its output is an infinite signal.

### 6.3 Implementation of Processes

```

fifoDelaySY xs          = mooreSY fifoState fifoOutput (queue []) xs

fifoState              :: Queue a → [a] → Queue a
fifoState (Q []) xs   = (Q xs)
fifoState q xs        = fst (popQ (pushListQ q xs))

fifoOutput             :: Queue a → AbstExt a
fifoOutput (Q [])     = Abst
fifoOutput (Q (x:xs)) = Prst x

finiteFifoDelaySY n xs = mooreSY fifoStateFQ fifoOutputFQ (finiteQueue n []) xs

fifoStateFQ :: FiniteQueue a → [a] → FiniteQueue a
fifoStateFQ (FQ n []) xs = (FQ n xs)
fifoStateFQ q xs        = fst (popFQ (pushListFQ q xs))

fifoOutputFQ :: FiniteQueue a → AbstExt a
fifoOutputFQ (FQ n []) = Abst
fifoOutputFQ (FQ n (x:xs)) = Prst x

memorySY size xs      = mealySY ns o (newMem size) xs
  where
    ns mem (Read x)    = memState mem (Read x)
    ns mem (Write x v) = memState mem (Write x v)
    o mem (Read x)     = memOutput mem (Read x)
    o mem (Write x v)  = memOutput mem (Write x v)

mergeSY xs ys         = moore2SY mergeState mergeOutput [] xs ys
  where
    mergeState []      Abst Abst = []
    mergeState []      Abst (Prst y) = [y]
    mergeState []      (Prst x) Abst = [x]
    mergeState []      (Prst x) (Prst y) = [x, y]
    mergeState (u:us) Abst Abst = us
    mergeState (u:us) Abst (Prst y) = us ++ [y]
    mergeState (u:us) (Prst x) Abst = us ++ [x]
    mergeState (u:us) (Prst x) (Prst y) = us ++ [x, y]

    mergeOutput []     = Abst
    mergeOutput (u:us) = Prst u

counterSY min max = counterSY' min min max
  where counterSY' x min max | x /= max = x :- counterSY' (succ x) min max
        | x == max = x :- counterSY' min min max

```

# Chapter 7

## The module **DataflowLib**

### 7.1 Overview

This library defines data types, skeletons and functions to model dataflow process networks, as they are described by Lee and Parks in [2]).

Each process is defined by a set of firing rules and corresponding actions. A process fires, if the incoming signals match a firing rule. Then the process consumes the matched tokens and executes the action corresponding to the firing rule.

We follow their definition of data flow process networks for the combinatorial processes.

```
module DataflowLib(  
    module Signal, FiringToken(Wild, Value), mapDF,  
    zipWithDF, zipWith3DF, mealyDF, mooreDF, scanlDF  
)  
where  
  
import FiringRules  
import Signal
```

### 7.2 Data Types

The data type `FiringToken` (defined in the module `FiringRules`) defines the data type for tokens. The constructor `Wild` constructs a token wildcard, the constructor `Value a` constructs a token with value `a`.

A sequence (pattern) matches a signal, if the sequence is a prefix of the signal. Table 7.1 illustrates the use of tokens in firing rules:

Token	ForSyDe Expression	Matches
$\perp$	<code>NullS</code>	matches always
<code>*</code>	<code>[Wild]</code>	matches signal with at least one token
<code>v</code>	<code>[Value v]</code>	matches signal with <code>v</code> as its first value
<code>*,*</code>	<code>[Wild, Wild]</code>	matches signals with at least two tokens

Table 7.1: Matching of Tokens

## 7.3 Skeletons

### 7.3.1 Skeletons for Combinatorial Processes

Combinatorial processes do not have an internal state. This means, that the output signal only depends on the input signals. The library contains the following combinatorial skeletons:

```
mapDF           :: Eq a => [[FiringToken a]]
                → (Signal a → [[b]]) → Signal a → Signal b
zipWithDF       :: (Eq a, Eq b) =>
                [([FiringToken b], [FiringToken a])]
                → (Signal b → Signal a → [[c]]) → Signal b
                → Signal a → Signal c
zipWith3DF      :: (Eq a, Eq b, Eq c) =>
                [([FiringToken a],[FiringToken b],[FiringToken c])]
                → (Signal a → Signal b → Signal c → [[d]])
                → Signal a → Signal b → Signal c → Signal d
```

The skeleton `mapDF` takes a list of firing rules, a list of corresponding output functions to generate a data flow process with one input and one output signal. The skeleton `zipWithDF` and `zipWith3DF` work in the same way, but have two and three input signals. To illustrate the concept of data flow processes, we create a process that selects tokens from two inputs according to a control signal. The process has the following firing rules [2]:

$$\mathbf{R}_1 = \{[*], \perp, [T]\} \quad (7.1)$$

$$\mathbf{R}_2 = \{\perp, [*], [F]\} \quad (7.2)$$

The corresponding ForSyDe formulation of the firing rules is:

```
selectRules     = [ ([Wild], [], [Value True]),
                   ([], [Wild], [Value False]) ]
```

For the output we formulate the following set of output functions:

```
selectOutput xs ys _ = [ [headS xs],
                        [headS ys] ]
```

The select process `selectDF` is then defined by:

```
selectDF       :: Eq a => Signal a → Signal a
                → Signal Bool → Signal a
selectDF       = zipWith3DF selectRules selectOutput
```

Given the signals `s1`, `s2` and `s3`,

```
s1 = signal [1,2,3,4,5,6]
s2 = signal [7,8,9,10,11,12]
s3 = signal [True, True, False, False, True, True]
```

the executed process gives the following results:

```
DataflowLib> selectDF s1 s2 s3
{1,2,7,8,3,4} :: Signal Integer
```

### 7.3.2 Skeletons for Sequential Processes

The `mealyDF` skeleton implements the most general state machine in the ForSyDe methodology. It takes a set of firing rules, a set of corresponding next state functions and a set of output functions as argument. A firing rule is a tuple. The first value is a pattern for the state, the second value corresponds to an input pattern.

When a pattern matches, the process fires, the corresponding next state and output functions are executed, and the tokens matching the pattern are consumed.

As an example we can view a process calculating the running sum of the input tokens. It has only one firing rule, which is illustrated in Table 7.3.2. A dataflow

Firing Rule	Next State	Output
(*,[*])	state + x	{state}

Table 7.2: Firing Rules for a Running Sum Example

process using these firing rules and the initial state 0 can be formulated in ForSyDe as

```
rs xs = mealyDF firingRule nextState output initState xs
  where firingRule = [(Wild, [Wild])]
        nextState state xs = [(state + headS xs)]
        output state _ = [[state]]
        initState = 0
```

Execution of the process gives:

```
DataflowLib> rs (signal [1,2,3,4,5,6])
{0,1,3,6,10,15} :: Signal Integer
```

Another 'running sum' process rs2 takes two tokens, pushes them into a queue of five elements and calculates the sum as output.

```
rs2 = mealyDF fs ns o init
  where init = [0,0,0,0,0]
        fs = [(Wild, ([Wild, Wild]))]
        ns state xs = [drop 2 state ++ fromSignal (takeS 2 xs)]
        o state _ = [(sum state)]
```

```
DataflowLib>rs2 (signal [1,2,3,4,5,6,7,8,9,10])
{0,3,10,20,30} :: Signal Integer
```

Besides the skeleton mealyDF there are two more skeletons mooreDF, which models a Moore automaton and scanDF, which models an automaton without an output decoder.

```
mealyDF :: (Eq a, Eq b) => [(FiringToken b,[FiringToken a])]
  -> (b -> Signal a -> [b]) -> (b -> Signal a -> [[c]])
  -> b -> Signal a -> Signal c
mooreDF :: (Eq a, Eq b) => [(FiringToken b,[FiringToken a])]
  -> (b -> Signal a -> [b]) -> (b -> [c])
  -> b -> Signal a -> Signal c
scanDF :: (Eq a, Eq b) => [(FiringToken b,[FiringToken a])]
  -> (b -> Signal a -> [b])
  -> b -> Signal a -> Signal b
```

## 7.4 Implementation

### 7.4.1 Combinatorial Skeletons

```
mapDF _ _ NullS = NullS
mapDF rs as xs = output ++ mapDF rs as xs'
  where xs' = if matchedRule < 0 then
```

```

                                NullS
                                else
                                consumeDF rule xs
matchedRule = (matchDF rs xs)
rule        = rs !! matchedRule
output      = if matchedRule < 0 then
              NullS
              else
              signal ((as xs) !! matchedRule)

zipWithDF _ _ NullS NullS = NullS
zipWithDF rs as xs      ys = output ++ zipWithDF rs as xs' ys'
  where
    (xs', ys') = if matchedRule < 0 then
                  (NullS, NullS)
                  else
                  consume2DF rule xs ys
matchedRule = (match2DF rs xs ys)
rule        = rs !! matchedRule
output      = if matchedRule < 0 then
              NullS
              else
              signal ((as xs ys) !! matchedRule)

zipWith3DF _ _ NullS NullS NullS = NullS
zipWith3DF rs as xs ys zs        = output ++ zipWith3DF rs as xs' ys' zs'
  where
    (xs', ys', zs') = if matchedRule < 0 then
                       (NullS, NullS, NullS)
                       else
                       consume3DF rule xs ys zs
matchedRule = (match3DF rs xs ys zs)
rule        = rs !! matchedRule
output      = if matchedRule < 0 then
              NullS
              else
              signal ((as xs ys zs) !! matchedRule)

```

## 7.4.2 Sequential Skeletons

```

mealyDF _ _ _ _ NullS = NullS
mealyDF fs ns o state xs = output ++ mealyDF fs ns o state' xs'
  where
    xs' = if matchedRule < 0 then
           NullS
           else
           consumeDF rule xs
matchedRule = matchStDF fs state xs
rule        = snd (fs !! matchedRule)
output      = signal ((o state xs) !! matchedRule)
state'      = if matchedRule < 0 then
              error "No rule matches the pattern!"
              else
              (ns state xs) !! matchedRule

mooreDF _ _ _ _ NullS = NullS
mooreDF fs ns o state xs = output ++ mooreDF fs ns o state' xs'
  where
    xs' = if matchedRule < 0 then

```

```

        NullS
      else
        consumeDF rule xs
    matchedRule = matchStDF fs state xs
    rule        = snd (fs !! matchedRule)
    output      = signal (o state)
    state'     = if matchedRule < 0 then
                  error "No rule matches the pattern!"
                else
                  (ns state xs) !! matchedRule

scanlDF _ _ _ NullS = NullS
scanlDF fs ns state xs = (unitS state)
  +-+ scanlDF fs ns state' xs'

  where
    xs' = if matchedRule < 0 then
          NullS
        else
          consumeDF rule xs
    matchedRule = matchStDF fs state xs
    rule        = snd (fs !! matchedRule)
    state'     = if matchedRule < 0 then
                  error "No rule matches the pattern!"
                else
                  (ns state xs) !! matchedRule

```

### 7.4.3 Supporting Functions

The function `prefixDF` takes a pattern and a signal and returns `True`, if the pattern is a prefix from the signal.

```

prefixDF :: Eq a => [FiringToken a] -> Signal a -> Bool
prefixDF [] _ = True
prefixDF _ NullS = False
prefixDF (Wild:ps) (_:-xs) = prefixDF ps xs
prefixDF ((Value p):ps) (x:-xs) = if p == x then
  prefixDF ps xs
  else
  False

```

The function `consumeDF` takes a pattern and a signal and 'consumes' the pattern from the signal. The functions `consume2DF` and `consume3DF` work in the same way as `consumeDF`, but with two and three input signals.

```

consumeDF :: Eq a => [FiringToken a]
  -> Signal a -> Signal a
consumeDF _ NullS = NullS
consumeDF [] xs = xs
consumeDF (Wild:ts) (_:-xs) = consumeDF ts xs
consumeDF (Value t:ts) (x:-xs) = if t == x then
  consumeDF ts xs
  else
  error "Tokens not correct"

consume2DF :: (Eq a, Eq b) =>
  ([FiringToken a], [FiringToken b])
  -> Signal a -> Signal b -> (Signal a, Signal b)
consume2DF (px, py) xs ys = (consumeDF px xs,
  consumeDF py ys)

consume3DF :: (Eq a, Eq b, Eq c) =>

```

```

                                ([FiringToken a], [FiringToken b], [FiringToken c])
                                → Signal a → Signal b → Signal c
                                → (Signal a,Signal b,Signal c)
consume3DF (px, py, pz) xs ys zs = (consumeDF px xs,
                                   consumeDF py ys,
                                   consumeDF pz zs)

```

The function `matchDF` checks, which firing rule, starting from 0, is matched by the input signal. If no firing rule matches, the output is '-1'. The functions `match2S` and `match3DF` work in the same way for two and three inputs.

```

matchDF                                :: (Num a, Eq b) =>
                                [[FiringToken b]] → Signal b → a
matchDF rs xs                          = matchDF' 0 rs xs
  where matchDF' _ [] _                = -1
        matchDF' n (r:rs) xs          = if prefixDF r xs then
                                        n
                                        else
                                        matchDF' (n+1) rs xs

match2DF                                :: (Num a, Eq b, Eq c) =>
                                [[FiringToken b], [FiringToken c]]
                                → Signal b → Signal c → a
match2DF rs xs ys                      = match2DF' 0 rs xs ys
  where match2DF' _ [] _ _              = -1
        match2DF' n ((rx, ry):rs) xs ys
                                        = if prefixDF rx xs &&
                                            prefixDF ry ys
                                        then
                                            n
                                        else
                                            match2DF' (n+1) rs xs ys

match3DF                                :: (Num a, Eq b, Eq c, Eq d) =>
                                [[FiringToken b], [FiringToken d], [FiringToken c]]
                                → Signal b → Signal d → Signal c → a
match3DF rs xs ys zs                   = match3DF' 0 rs xs ys zs
  where match3DF' _ [] _ _ _            = -1
        match3DF' n ((rx, ry, rz):rs) xs ys zs
                                        = if prefixDF rx xs &&
                                            prefixDF ry ys &&
                                            prefixDF rz zs
                                        then
                                            n
                                        else
                                            match3DF' (n+1) rs xs ys zs

```

The function `matchStDF` works in the same way as `matchDF`, but it looks on patterns that include the state.

```

matchStDF                               :: (Num a, Eq b, Eq c) =>
                                [(FiringToken c,[FiringToken b])]
                                → c → Signal b → a
matchStDF rs state xs                  = matchStDF' 0 rs state xs
  where matchStDF' _ [] _ _             = -1
        matchStDF' n (r:rs) state xs
                                        = if prefixDF (snd r) xs &&
                                            matchState (fst r) state
                                        then
                                            n
                                        else

```

```
matchStDF' (n+1) rs state xs
matchState                                     :: Eq a => FiringToken a → a → Bool
matchState Wild                               = True
matchState (Value v) x                       = x == v
```

# Chapter 8

## The module **UntimedLib**

### 8.1 Overview

The untimed library follows the definition of the process constructors in the paper [1].

We only need the signal models from the ForSyDe libraries.

```
module UntimedLib(  
    module Signal, mapU, scanU, zipU, zipUs,  
    zipWithU, mealyU, mooreU, sourceU, sinkU, initU,  
    unzipU  
)  
  
where  
  
import Signal
```

### 8.2 Skeletons

#### 8.2.1 Skeletons for Combinatorial Processes

```
mapU :: Int -> ([a] -> [b]) -> Signal a -> Signal b  
mapU c f NullS = NullS  
mapU c f xs | lenS (takeSp c xs) < c = NullS  
            | lenS (takeSp c xs) == c  
            = signal (f (takeL c xs)) ++ (mapU c f (dropSp c xs))
```

The first parameter of `mapU` is a constant integer defining the number of tokens consumed in every evaluation cycle. The second argument is a function on lists of the input type and returning a list of the output type. For instance,

```
r2 = mapU 1 f  
    where f :: [Int] -> [Int]  
          f [x] = [2*x]
```

defines a process `r2` which consumes one token in each evaluation cycle and multiplies it by two.

#### 8.2.2 Skeletons for Sequential Processes

`scanU` has an internal state which is visible at the output. The first argument is a function  $\gamma$  which, given the state returns the number of tokens consumed next. The second argument is the next state function and the third is the initial state.

```

scanU :: (b→Int) → (b→[a]→b) → b → Signal a → Signal b
scanU gamma g state NullS = NullS
scanU gamma g state xs
  | lenS (takeSp c xs) == c = newstate
                                :- scanU gamma g newstate (dropSp c xs)
  | lenS (takeSp c xs) < c = NullS
  where c          = gamma state
        newstate = g state (takeL c xs)

```

The next two skeletons create Moore and Mealy based state machines. In addition to the next state function they also have an output encoding function.

```

mooreU :: (b→Int) → (b→[a]→b) → (b → [c]) → b
        → Signal a → Signal c
mooreU _ _ _ NullS = NullS
mooreU gamma g f state xs
  | length as == c = signal (f state)
                    ++ mooreU gamma g f newstate (dropSp c xs)
  | otherwise      = NullS
  where c          = gamma state
        as         = takeL c xs
        newstate = g state as

```

```

mealyU :: (b→Int) → (b→[a]→b) → (b → [a] → [c]) → b
        → Signal a → Signal c
mealyU _ _ _ NullS = NullS
mealyU gamma g f state xs
  | length as == c = signal (f state as)
                    ++ mealyU gamma g f newstate (dropSp c xs)
  | otherwise      = NullS
  where c          = gamma state
        as         = takeL c xs
        newstate = g state as

```

### 8.2.3 Zip and Unzip based Skeletons

```

zipU :: Signal (Int,Int) → Signal a → Signal b → Signal ([a],[b])
zipU NullS _ _ = NullS
zipU _ NullS _ = NullS
zipU _ _ NullS = NullS
zipU ((c1,c2):-cs) xs ys
  | lenS (takeSp c1 xs) == c1 && lenS (takeSp c2 ys) == c2
  = (takeL c1 xs, takeL c2 ys) :- zipU cs (dropSp c1 xs) (dropSp c2 ys)
  | (lenS (takeSp c1 xs) < c1 || lenS (takeSp c2 ys) < c2)
  = NullS

```

```

zipUs :: Int → Int → Signal a → Signal b → Signal ([a],[b])
zipUs _ _ NullS _ = NullS
zipUs _ _ _ NullS = NullS
zipUs c1 c2 xs ys
  | lenS (takeSp c1 xs) == c1 && lenS (takeSp c2 ys) == c2
  = (takeL c1 xs, takeL c2 ys)
  :- zipUs c1 c2 (dropSp c1 xs) (dropSp c2 ys)
  | otherwise = NullS

```

```

zipWithU :: Int → Int → ([a]→[b]→[c]) → Signal a → Signal b → Signal c
zipWithU _ _ f NullS _ = NullS
zipWithU _ _ f _ NullS = NullS
zipWithU c1 c2 f xs ys

```

```

| lenS (takeSp c1 xs) == c1 && lenS (takeSp c2 ys) == c2
  = signal (f (takeL c1 xs) (takeL c2 ys))
    +++ zipWithU c1 c2 f (dropSp c1 xs) (dropSp c2 ys)
| otherwise = NullS

```

```

unzipU :: Signal ([a],[b]) → (Signal a,Signal b)
unzipU NullS = (NullS,NullS)
unzipU ((as,bs):-xs) = (signal as +++ ass,
                        signal bs +++ bss)
                        where (ass,bss) = unzipU xs

```

### 8.2.4 Source and Sink Skeletons

```

sourceU :: (a→a) → a → Signal a
sourceU g state = newstate :- sourceU g newstate
                where newstate = g state

```

```

sinkU :: (a→Int) → (a→a) → a → Signal b → Signal b
sinkU _ _ _ NullS = NullS
sinkU gamma g state xs
  | length as == c = sinkU gamma g newstate (dropSp c xs)
  | otherwise      = NullS
  where as         = takeL c xs
        c         = gamma state
        newstate  = g state

```

initU is used to initialise a signal. Its first argument is prepended to its second argument, a signal.

```

initU :: [a] → Signal a → Signal a
initU init s = (signal init) +++ s

```

## 8.3 Helper Functions

```

lenS :: Signal a → Int
lenS NullS = 0
lenS (x:-xs) = 1 + lenS xs

takeSp 0 _      = NullS
takeSp _ NullS = NullS
takeSp n (x:-xs) = x :- takeSp (n-1) xs

dropSp 0 xs     = xs
dropSp _ NullS = NullS
dropSp n (x:-xs) = dropSp (n-1) xs

takeL c = fromSignal . (takeSp c)

```

# Chapter 9

## The module `DiscreteEventLib`

### 9.1 Overview

This library defines data types, skeletons and functions to model dataflow process networks, as they are described by Lee and Parks in [2]).

Each process is defined by a set of firing rules and corresponding actions. A process fires, if the incoming signals match a firing rule. Then the process consumes the matched tokens and executes the action corresponding to the firingrule. However it outputs only those events which have `time_tag` smaller than or equal to the maximum consumed input `time_tag`. The rest is stored in memory and remaining there until some input time tag is consumed which is greater than that.

```
module DiscreteEventLib (  
    module Signal, DiscreteEvent (DE),  
    mapDE, zipWithDE, zipWith3DE, mealyDE, mooreDE,  
    scanlDE)  
where  
import Signal  
import FiringRules
```

### 9.2 Data Types

The data type `DiscreteEvent` defines the data type for events of a `DiscreteEvent` `Signal`.

Two functions are defined on this type as well, `give_val` returns the value and `give_tag` returns the timing tag of `DiscreteEvent`

```
type Time_tag = Int  
  
data DiscreteEvent a = DE a Time_tag deriving (Eq, Show)  
  
give_val :: DiscreteEvent a → a  
give_val (DE x _) = x  
  
give_tag :: DiscreteEvent a → Time_tag  
give_tag (DE _ i) = i
```

## 9.3 Skeltons

### 9.4 Skeltons for combinatorial processes

The library contain the following combinatorial skeltons.

```
mapDE :: Eq a => [[FiringToken a]] → (Signal (DiscreteEvent a) →
  [Signal (DiscreteEvent b)]) → Signal (DiscreteEvent a)
  → Signal (DiscreteEvent b)
```

```
zipWithDE :: (Eq a, Eq b) => ([[FiringToken a],[FiringToken b]])
  → (Signal (DiscreteEvent a) → Signal (DiscreteEvent b)
  → [[DiscreteEvent c]]) → Signal (DiscreteEvent a)
  → Signal (DiscreteEvent b) → Signal (DiscreteEvent c)
```

```
zipWith3DE :: (Eq a, Eq b, Eq c) =>
  ([[FiringToken a],[FiringToken b],[FiringToken c]])
  → (Signal (DiscreteEvent a) → Signal (DiscreteEvent b)
  → Signal (DiscreteEvent c) → [[DiscreteEvent d]])
  → Signal (DiscreteEvent a) → Signal (DiscreteEvent b)
  → Signal (DiscreteEvent c) → Signal (DiscreteEvent d)
```

The skeleton `mapDE` takes a list of firing rules, a list of corresponding output functions to generate a `DiscreteEvent` process with one input and one output signal. The firing rules matches only the value of the event and not the time tag. The skeleton `zipWithDE` and `zipWith3DE` work in the same way, but have two and three input signals. To illustrate the concept of data flow processes, we create a process that selects tokens from two inputs according to a control signal. The process has the following firing rules

$$\mathbf{R} = \{[*,'a',*,'b'],[*,'b','b'],[*,*,'a']\} \quad (9.1)$$

The corresponding `ForSyDe` formulation of the firing rules is:

```
selectRules = [[Wild, Value 'a', Wild, Value 'b'],
  [Wild, Value 'b', Value 'b'],
  [Wild, Wild, Value 'a']]
```

For the output we formulate the following set of output functions: The function `max_tag i s`, returns the maximum timing tag of first `i` event of signal `s`.

```
selectOutput xs = [f1 xs, f2 xs, f3 xs]
```

```
f1 x = DE 'c' (t+1):-DE 'c' (t+2):-DE 'c' (t+3):-NullS
  where
  t = max_tag 4 x
```

```
f3 x = DE 'c' (2*t):-DE 'c' (4*t):-DE 'c' (4*t+2):-NullS
  where
  t = max_tag 3 x
```

```
f2 x = DE 'c' (t+1):-DE 'c' (t+2):-DE 'c' (t+4):-NullS
  where
  t = max_tag 3 x
```

```
max_tag i NullS = -1
max_tag 1 (DE p j:-xs) = j
max_tag i (DE p j:-xs) = max j (max_tag (i-1) xs)
```

Now the process `proc` is defined as follows.

```
proc :: Signal (DiscreteEvent Char) → Signal (DiscreteEvent Char)
proc = mapDE selectRules selectOutput
```

Given the input signal s

```
s = (DE 'a' 1:- DE 'b' 2:- DE 'a' 3:- DE 'a' 4:-
      DE 'a' 5:- DE 'b' 6:-DE 'b' 7:-DE 'a' 7:-
      DE 'b' 8:- DE 'b' 9:-NullS)
```

The executed process gives the following result:

```
DiscreteEvent> proc s
{DE 'c' 6,DE 'c' 8,DE 'c' 9,DE 'c' 10,DE 'c' 10,
DE 'c' 11,DE 'c' 12,DE 'c' 13,DE 'c' 14} :: Signal DiscreteEvent Char
```

## 9.5 Skeltons for sequential processes

The mealyDF skeleton implements the most general state machine in the ForSyDe methodology. It takes a set of firing rules, a set of corresponding next state functions and a set of output functions as argument. A firing rule is a tuple. The first value is a pattern for the state, the second value corresponds to an input pattern. When a pattern matches, the process fires, the corresponding next state and output functions are executed, and the tokens matching the pattern are consumed. However the actual output in that firing cycle is of only those output events for which the timing tag is smaller than the maximal consumed input token so far. This has to be done, so that the final output signal have tags in globally increasing order.

The library contains the following sequential skeltons.

```
mealyDE :: (Eq a, Eq b) ⇒ [(FiringToken b,[FiringToken a])]
→ (b → Signal (DiscreteEvent a) → [b])
→ (b → Signal (DiscreteEvent a) → [[DiscreteEvent c]])
→ b → Signal (DiscreteEvent a) → Signal (DiscreteEvent c)
```

```
mooreDE :: (Eq a, Eq b) ⇒ [(FiringToken b,[FiringToken a])]
→ (b → Signal (DiscreteEvent a) → [b])
→ (b → [DiscreteEvent c]) → b → Signal (DiscreteEvent a)
→ Signal (DiscreteEvent c)
```

```
scanlDE :: (Eq a, Eq b)
⇒ [(FiringToken (DiscreteEvent b),[FiringToken a])]
→ (DiscreteEvent b → Signal (DiscreteEvent a)
→ [DiscreteEvent b]) → DiscreteEvent b
→ Signal (DiscreteEvent a) → Signal (DiscreteEvent b)
```

As an example consider a Mealy machine defined by following -

```
output_fun state xs = [ff1 state xs, ff2 state xs, ff3 state xs, ff4 state xs]
ff1 state NullS = []
ff1 1 (DE x i:-xs) = (DE x (i*t):ff1 1 xs)
  where
    t = max_tag 3 (DE x i:-xs)

ff1 0 _ = []

ff2 0 x = [DE 'c' (t+2), DE 'c' (t+3), DE 'c' (t+11)]
  where
    t = max_tag 3 x
ff2 1 NullS = []
```

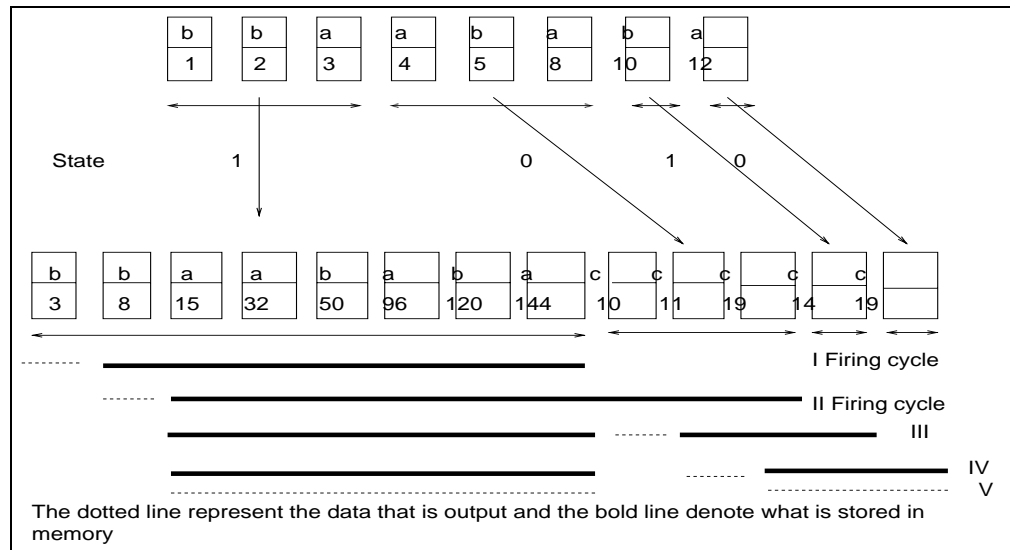


Figure 9.1: System Layer

```
ff3 _ x = [DE 'c' (4+t)]
```

**where**

```
t = max_tag 1 x
```

```
ff4 _ x = [DE 'c' (t+7)]
```

**where**

```
t = max_tag 1 x
```

```
nexstate_fun state xs = [o1 state xs, o2 state xs, o1 state xs, o2 state xs]
```

```
o1 state NullS = 0
```

```
o1 1 (DE 'b' _ :- xs) = 0
```

```
o1 _ _ = 1
```

```
o2 0 (DE 'a' _ :- xs) = 1
```

```
o2 _ _ = 0
```

```
firing_rules = [(Value 1, [Wild, Wild, Value 'a']),
                (Value 0, [Wild, Wild, Value 'a']),
                (Value 1, [Wild]), (Value 0, [Wild])]
```

```
inputt = (DE 'b' 1:-DE 'b' 2:-DE 'a' 3:-DE 'a' 4:-DE 'b' 5:-DE 'a' 8:-DE 'b' 10
          :-DE 'a' 12:-NullS)
```

Now we can define a process pro, by using the mealyDE skelton

```
pro = mealyDE firing_rules nexstate_fun output_fun 1
```

Execution of the process gives:

```
DiscreteEvent> pro inputt
```

```
{DE 'b' 3,DE 'b' 8,DE 'c' 10,DE 'c' 11,DE 'c' 14,DE 'a' 15,
DE 'c' 19,DE 'c' 19,DE 'a' 32,DE 'b' 50,DE 'a' 96,DE 'b' 120,
DE 'a' 144}:: Signal (DiscreteEvent Char)
```

The figure explains how this is happening -

## 9.6 implementation

### 9.6.1 Combinatorial Skeltons

```
mapDE rs as xs = mapDD rs as xs NullS
zipWithDE rs as xs ys = zipWithDD rs as xs ys NullS
zipWith3DE rs as xs ys zs = zipWith3DD rs as xs ys zs NullS
```

The function `mapDE` is implemented by calling another function `mapDD`. The functionality of `mapDD` can be explained in following steps -

1. The local variable `matchedRule` carries the index of the rule which matches the current input signal.
2. using the output function `as`, and the index `matchedRule`, we get the value of output in the current cycle.
3. The variable `s` is used to carry information about the output produced in old cycles which is not yet outputted.
4. Combining the signal outputted in the present cycle and that which is coming from the old cycles, those events are selected which have tag less than the largest consumed input tags and are actually outputted.
5. The largest consumed input tag is given by variable `i`
6. Those events which are not outputted are send to the next cycle by the variable `haskellremaining`.

All the functions in DiscreteEvent Library essentially follows the theme described above.

```
mapDD _ _ NullS s = s
mapDD rs as xs s = put ++ mapDD rs as xs' remaining
  where
    xs' = if matchedRule < 0 then
          NullS
        else
          fst l
    i = snd l
    l = consumeD rule xs
    matchedRule = (matchD rs xs)
    rule = rs !! matchedRule
    put = if matchedRule < 0 then
          NullS
        else
          fst m
    remaining = if matchedRule < 0 then
                 s
               else
                 snd m
    m = cutting s ((as xs) !! matchedRule) i
```

```

zipWithDD _ _ NullS NullS s = s
zipWithDD rs as xs ys s = put ++ zipWithDD rs as xs' ys' remaining
  where
    matchedRule = match2D rs xs ys
    rule = rs !! matchedRule
    (l,m,n) = consume2D rule xs ys
    xs' = if matchedRule < 0 then
          NullS
        else
          |
    ys' = if matchedRule < 0 then
          NullS
        else
          m
    put = if matchedRule < 0 then
          NullS
        else
          fst k
    remaining = if matchedRule < 0 then
                 s
               else
                 snd k
    k = cutting s (signal ((as xs ys) !! matchedRule)) n

zipWith3DD _ _ NullS NullS NullS s = s
zipWith3DD rs as xs ys zs s = put ++ zipWith3DD rs as xs' ys' zs' remaining
  where
    matchedRule = match3D rs xs ys zs
    rule = rs !! matchedRule
    (l,m,o,n) = consume3D rule xs ys zs
    xs' = if matchedRule < 0 then
          NullS
        else
          |
    ys' = if matchedRule < 0 then
          NullS
        else
          m
    zs' = if matchedRule < 0 then
          NullS
        else
          o
    put = if matchedRule < 0 then
          NullS
        else
          fst k
    remaining = if matchedRule < 0 then
                 s
               else
                 snd k
    k = cutting s (signal ((as xs ys zs) !! matchedRule)) n

```

## 9.6.2 Sequential Skeltons

```
mealyDE fs ns o state xs = mealyDD fs ns o state xs NullS
```

```

mealyDD fs ns o state NullS s = s
mealyDD fs ns o state xs s = put ++ mealyDD fs ns o state' xs' remaining
  where
    matchedRule = matchStD fs state xs
    rule = snd (fs !! matchedRule)
    state' =
      if matchedRule < 0 then
        error "No_rule_matches_the_pattern!!"
      else
        (ns state xs) !! matchedRule
    l = consumeD rule xs
    xs' = if matchedRule < 0 then
      NullS
    else
      fst l
    i = snd l
    m = cutting s (signal ((o state xs) !! matchedRule)) i
    put = if matchedRule < 0 then
      NullS
    else
      fst m
    remaining = if matchedRule < 0 then
      NullS
    else
      snd m

```

```

mooreDE fs ns o state xs = mooreDD fs ns o state xs NullS

```

```

mooreDD fs ns o state NullS s = s
mooreDD fs ns o state xs s = put ++ mooreDD fs ns o state' xs' remaining
  where
    matchedRule = matchStD fs state xs
    rule = snd (fs !! matchedRule)
    state' =
      if matchedRule < 0 then
        error "No_rule_matches_the_pattern!!"
      else
        (ns state xs) !! matchedRule
    l = consumeD rule xs
    xs' = if matchedRule < 0 then
      NullS
    else
      fst l
    i = snd l
    m = cutting s (signal (o state)) i
    put = if matchedRule < 0 then
      NullS
    else
      fst m
    remaining = if matchedRule < 0 then
      NullS
    else
      snd m

```

```

scanlDE fs ns state xs = scanlDD fs ns state xs NullS

```

```

scanlDD fs ns state NullS s = s
scanlDD fs ns state xs s = put ++ scanlDD fs ns state' xs' remaining

```

```

where
matchedRule = matchStD fs state xs
rule = snd (fs !! matchedRule)
state' =
  if matchedRule < 0 then
    error "No rule matches the pattern!!"
  else
    (ns state xs) !! matchedRule
l = consumeD rule xs
xs' = if matchedRule < 0 then
  NullS
  else
    fst l
i = snd l
m = cutting s (unitS state) i
put = if matchedRule < 0 then
  NullS
  else
    fst m
remaining = if matchedRule < 0 then
  NullS
  else
    snd m

```

### 9.6.3 supporting function

The function `prefixD` takes a pattern and a signal of discrete events and return true if the pattern matches the signal. The pattern have only values and not the time tag.

```

prefixD :: Eq a => [FiringToken a] → Signal (DiscreteEvent a) → Bool

```

```

prefixD [] _ = True
prefixD _ NullS = False
prefixD (Wild:ps) (_:-xs) = prefixD ps xs
prefixD ((Value p):ps) (x:-xs) = if p == (give_val x) then
  prefixD ps xs
  else
    False

```

The function `cutting` is taking two signals and an integer `i` as argument and it is returning two sorted signals `put` and `remaining`. The signal `put` is having those values which are less than `i` and the signal `remaining` has those values which are greater than `i`.

```

cutting :: Signal (DiscreteEvent a) → Signal (DiscreteEvent a) →
  Int → (Signal (DiscreteEvent a), Signal (DiscreteEvent a))

```

```

cutting s t i = (put, remaining)
where
put = fst l
l = (lessThan s i)
remaining = sorrt (snd l) t

```

```

sorrt :: Signal (DiscreteEvent a) → Signal (DiscreteEvent a) → Signal (DiscreteEvent a)

```

```

sorrt NullS t = t
sorrt s NullS = s
sorrt (x:-xs) (y:-ys)

```

```

| (give_tag x) <= (give_tag y) = (x:-(sorrt xs (y:-ys)))
| otherwise = (y:-(sorrt (x:-xs) ys))

```

```

lessThan :: Signal (DiscreteEvent a) → Int
          → (Signal (DiscreteEvent a), Signal (DiscreteEvent a))
lessThan NullS i = (NullS, NullS)
lessThan (x:-xs) i
| (give_tag x) <= i = (x:-p,q)
| otherwise = (NullS,(x:-xs))
  where
    (p,q) = lessThan xs i

```

The function `consumeT` takes a pattern and a signal and 'consumes' the pattern from the signal. The functions `consume2T` and `consume3T` work in the same way as `consumeT`, but with two and three input signals.

```

consumeD :: Eq a ⇒ [FiringToken a] → Signal (DiscreteEvent a)
          → (Signal (DiscreteEvent a), Int)

```

```

consumeD rule xs = consumedD' 0 rule xs
  where
    consumedD' i _ NullS = (NullS, i)
    consumedD' i [] xs = (xs, i)
    consumedD' i (Wild:ts) (a:-xs) = consumedD' (give_tag a) ts xs
    consumedD' i (Value t:ts) (a:-xs) = if t == (give_val a) then
      consumedD' (give_tag a) ts xs
    else
      error "Token_not_correct!"

```

```

consume2D :: (Eq a, Eq b) ⇒ ([FiringToken b],[FiringToken a]) →
           Signal (DiscreteEvent b) → Signal (DiscreteEvent a) →
           (Signal (DiscreteEvent b), Signal (DiscreteEvent a), Int)

```

```

consume2D rule xs ys = consume2D' 0 rule xs ys

```

```

  where
    consume2D' i _ NullS NullS = (NullS, NullS, i)
    consume2D' i ([],[ ]) xs ys = (xs, ys, i)
    consume2D' i ([],[ ]) xs (b:-ys) = if give_tag b < i then
      consume2D' i ([],[ ]) xs ys
    else
      consume2D' (give_tag b) ([],[ ]) xs ys
    consume2D' i ([],[ ]) xs (b:-ys) = if a == (give_val b) then
      if give_tag b < i then
        consume2D' i ([],[ ]) xs ys
      else
        consume2D' (give_tag b) ([],[ ]) xs ys
    else
      error "Token_not_correct!!!"
    consume2D' i ((Value a:ts),[ ]) (b:-xs) ys = if a == (give_val b) then
      if (give_tag b) < i then
        consume2D' i (ts,[ ]) xs ys
      else
        consume2D' (give_tag b) (ts,[ ]) xs ys
    else
      error "Token_not_correct!!!"
    consume2D' i ((Value a:ts),(Value b:us)) (p:-xs) (q:-ys) =
      if (a == (give_val p) && b == give_val q) then
        if (give_tag p) < (give_tag q) then
          consume2D' (give_tag q) (ts,us) xs ys
        else

```

```

        consume2D' (give_tag p) (ts,us) xs ys
    else
        error "Token_not_corect_!!!"
consume2D' i ((Wild:ts),(Value b:us)) (p:-xs) (q:-ys) =
    if (b == give_val q) then
        if (give_tag p) < (give_tag q) then
            consume2D' (give_tag q) (ts,us) xs ys
        else
            consume2D' (give_tag p) (ts,us) xs ys
    else
        error "Token_not_corect_!!!"
consume2D' i ((Value a:ts),(Wild:us)) (p:-xs) (q:-ys) =
    if (a == give_val p) then
        if (give_tag p) < (give_tag q) then
            consume2D' (give_tag q) (ts,us) xs ys
        else
            consume2D' (give_tag p) (ts,us) xs ys
    else
        error "Token_not_corect_!!!"
consume3D :: (Eq a, Eq b, Eq c) => ([FiringToken c],[FiringToken b],[FiringToken a])
    → Signal (DiscreteEvent c) → Signal (DiscreteEvent b)
    → Signal (DiscreteEvent a)
    → (Signal (DiscreteEvent c),Signal (DiscreteEvent b),Signal (DiscreteEvent a),Int)

consume3D (px,py,pz) xs ys zs = (xs',ys',zs',k)
    where
        (xs',p) = consumeD px xs
        (ys',q) = consumeD py ys
        (zs',r) = consumeD pz zs
        k = max (max p q) r

matchD :: (Num a, Eq b) => [[FiringToken b]] → Signal (DiscreteEvent b) → a
matchD rs xs = matchD' 0 rs xs
    where
        matchD' _ [] _ = -1
        matchD' n (r:rs) xs = if prefixD r xs then
            n
            else
                matchD' (n+1) rs xs

match2D :: (Num a, Eq b, Eq c) => [[[FiringToken b],[FiringToken c]]]
    → Signal (DiscreteEvent b) → Signal (DiscreteEvent c) → a

match2D rs xs ys = match2D' 0 rs xs ys
    where
        match2D' _ [] _ _ = -1
        match2D' n ((rx,ry):rs) xs ys
            = if prefixD rx xs && prefixD ry ys then
                n
            else
                match2D' (n+1) rs xs ys

match3D :: (Num a, Eq b, Eq c, Eq d)
    => [[[FiringToken b],[FiringToken d],[FiringToken c]]]
    → Signal (DiscreteEvent b) → Signal (DiscreteEvent d)
    → Signal (DiscreteEvent c) → a

```

```

match3D rs xs ys zs = match3D' 0 rs xs ys zs
  where
    match3D' _ [] _ _ = -1
    match3D' n ((rx, ry, rz):rs) xs ys zs =
      if prefixD rx xs &&
        prefixD ry ys &&
        prefixD rz zs
      then
        n
      else
        match3D' (n+1) rs xs ys zs

```

```

matchStD :: (Num a, Eq b, Eq c) => [(FiringToken c, [FiringToken b])]
  → c → Signal (DiscreteEvent b) → a

```

```

matchStD rs state xs = matchStD' 0 rs state xs
  where
    matchStD' _ [] _ _ = -1
    matchStD' n (r:rs) state xs
      = if prefixD (snd r) xs &&
        matchSttD (fst r) state
      then
        n
      else
        matchStD' (n+1) rs state xs
    matchSttD :: Eq a => FiringToken a → a → Bool
    matchSttD Wild _ = True
    matchSttD (Value a) x = a==x

```

## Chapter 10

# The module **StochasticLib**

# Chapter 11

## The Stochastic Library

### 11.1 Overview

The stochastic library provides a few stochastic skeletons, which are relatives to the skeletons of the synchronous library. These skeletons are based on two elementary functions, `sigmaUn` and `sigmaGe` which provide stochastic signals. The background and motivation for this approach is described in the paper [1]. Unfortunately, not all of the suggested skeletons are implemented.

```
module StochasticLib(module Signal, selMapSY, selScanSY, sigmaUn, sigmaGe)
  where
```

```
import SynchronousLib
import Signal
import Random
```

### 11.2 Skeletons for Stochastic Processes

The module contains the following skeletons to construct stochastic processes:

```
selMapSY  :: Int → (a → b) → (a → b) → Signal a → Signal b
selScanSY :: Int → (a → b → a) → (a → b → a) → a
           → Signal b → Signal a
```

The skeleton `selMapSY` is a stochastic variant of `mapSY`. It has an internal stochastic process and selects one out of two combinatorial functions depending on the output of the stochastic process. The skeleton `selScanSY` is a stochastic variant of `scanSY`. `sigmaUn` generates a signal list of uniformly distributed `Int` within the given range and with a given seed.

### 11.3 Implementation of Skeletons

#### 11.3.1 Skeletons for Stochastic Processes

```
select1          :: Int → (a → b) → (a→b) → a → b
select1 0 f0 _ x = f0 x
select1 1 _ f1 x = f1 x

select2          :: Int → (a → b → c) → (a→b→c)
                → a → b → c
select2 0 f0 _ x y = f0 x y
```

```

select2 1 _ f1 x y = f1 x y

selMapSY _ _ _ NullS = NullS
selMapSY seed f0 f1 xs = selmap1 f0 f1 (sigmaUn seed (0,1)) xs
  where selmap1 :: (a→b)→(a→b)→(Signal Int) → Signal a → Signal b
        selmap1 _ _ _ NullS = NullS
        selmap1 f0 f1 (s:-ss) (x:-xs)
          = (select1 s f0 f1 x) :- (selmap1 f0 f1 ss xs)

selScan1SY _ _ _ _ NullS = NullS
selScan1SY seed f0 f1 mem xs = selscan1 f0 f1 mem (sigmaUn seed (0,1)) xs
  where selscan1 :: (a → b → a) → (a → b → a) → a
                → Signal Int → Signal b → Signal a
        selscan1 _ _ _ _ NullS = NullS
        selscan1 f0 f1 mem (s:-ss) (x:-xs)
          = newmem :- (selscan1 f0 f1 newmem ss xs)
          where newmem = (select2 s f0 f1 mem x)

```

## 11.4 Supporting Functions

sigmaGe is a more general stochastic process. The first argument is a function  $f$  which describes the distribution. For each value  $v$  in the given range  $(r1,r2)$ ,  $f(v)$  is the probability that  $v$  is generated.

Note, that the user has to make sure that  $\text{sum}(f(v))=1$  for  $v$  in  $(r1,r2)$ .

```

sigmaUn :: Int → (Int, Int) → Signal Int
sigmaUn seed range = signal (stoch range (mkStdGen seed))
  where stoch :: (Int, Int) → StdGen → [Int]
        stoch range g = (fst (randomR range g))
                       : (stoch range (snd (next g)))

sigmaGe :: (Float → Float) → Int → (Int, Int) → Signal Int
sigmaGe f seed (r1,r2) = sigma2 (checkSum f (fromIntegral r1)
                                (fromIntegral r2)) f seed (r1,r2)
  where sigma2 s f seed (r1,r2)
        | s > 0.999 = signal (sigma1 (mkStdGen seed)
                                    (mkdlist f (fromIntegral (r2-r1))))
        | otherwise = error
                    ("sigmaGe: sum of probabilities is "
                     ++ (show s) ++ ". It must be 1.")

checkSum :: (Float → Float) → Float → Float → Float
checkSum f c max | c == max = f c
                 | otherwise = f(c) + (checkSum f (c+1) max)

sigma1 :: StdGen → [Float] → [Int]
sigma1 g fl = (findk (fst (randomR (0.0,1.0) g)) fl)
             : (sigma1 (snd (next g)) fl)

findk :: Float → [Float] → Int
findk r fs = findk1 0 r fs

findk1 k r (f:fs) | r < f = k
                 | otherwise = findk1 (k+1) r fs
findk1 k _ [] = k

mkdlist :: (Float → Float) → Float → [Float]
mkdlist f d = scanl (sumf f) 0.0 [1..d]

sumf :: (Float → Float) → Float → Float → Float

```

$$\text{sumf } g \times y = x + (g \ y)$$

For illustration consider the following example.

```
{-  
  pdist :: Float → Float  
  pdist d = 1/(2**d)  
  
  pdistsum 1 = pdist 1  
  pdistsum d = (pdist d) + (pdistsum (d-1))  
  
  pdistnorm :: Float → Float → Float  
  pdistnorm dmax d = 1/((pdistsum dmax) * (2**d))  
-}
```

# Chapter 12

## The module `FiringRules`

### 12.1 Overview

```
module FiringRules(  
    FiringToken (Wild, Value)  
)
```

**where**

This module contains the common datatype used in `DiscreteEvent` and `Dataflow` module.

The data type `FiringToken` defines the data type for tokens. The constructor `Wild` constructs a token wildcard, the constructor `Value a` constructs a token with value `a`

```
data FiringToken a = Wild  
    | Value a deriving (Eq, Show)
```

A sequence (pattern) matches a signal, if the sequence is a prefix of the signal. Table ?? illustrates the use of tokens in firing rules:

Token	ForSyDe Expression	Matches
<code>[⊥]</code>	<code>NullS</code>	matches always
<code>[*]</code>	<code>[Wild]</code>	matches signal with at least one token
<code>[v]</code>	<code>[Value v]</code>	matches signal with <code>v</code> as its first value
<code>[*,*]</code>	<code>[Wild, Wild]</code>	matches signals with at least two tokens

Table 12.1: Matching of Tokens

# Chapter 13

## The module `Vector`

### 13.1 Overview

The module `Vector` defines the data type `Vector` and the corresponding functions. It is a development of the module `Vector` defined by Reekie in [3]. Though the vector is modeled as a list, it should be viewed as an array, i.e. a vector has a *fixed size*. Unfortunately, it is not possible to have the size of the vector as a parameter of the vector data type, due to restrictions in Haskell's type system. Still most operations are defined for vectors with the same size.

```
module Vector (
    Vector (..), vector, fromVector, unitV, nullV, lengthV,
    atV, replaceV, headV, tailV, lastV, initV, takeV, dropV,
    selectV, groupV, (<+>), (<:), mapV, foldlV, foldrV, scanlV,
    scanrV, meshlV, meshrV, zipWithV, filterV, zipV, unzipV,
    concatV, reverseV, shiftlV, shiftrV, rotrV, rotlV,
    generateV, iterateV, copyV
) where

infixr 5 :>
infixl 5 <:
infixr 5 <+>
```

### 13.2 The Data Type `Vector`

The data type `Vector` is modeled similar to a list. It has two data type constructors. `NullV` constructs the empty vector, while `(:i)` a vector by adding an value to an existing vector. Using the inheritance mechanism of Haskell we have declared `Vector` as an instance of the classes `Read` and `Show`.

This means that the vector `1 :i 2 :i 3 :i 4 :i NullV` is shown as `|1,2,3,4i`.

```
data Vector a =          NullV
                | a :> (Vector a) deriving (Eq)
```

### 13.3 Functions on the Data Type `vector`

The function `vector` converts a list into a vector, while the function `fromVector` converts a vector into a list.

```
vector    :: [a] → Vector a
fromVector :: Vector a → [a]
```

The function `unitV` creates a vector with one element. The function `nullV` returns `True` if a vector is empty. The function `lengthV` returns the number of elements in a value. The function `atV` returns the  $n$ -th element in a vector, starting from zero. The function `replaceV` replaces an element in a vector.

```
unitV    :: a → Vector a
nullV    :: Vector a → Bool
lengthV  :: Num a ⇒ Vector b → a
replaceV :: Vector a → Int → a → Vector a
atV      :: Num a ⇒ Vector b → a → b
```

The functions `headV` and `lastV` return the first element or the the last element of a vector. The functions `tailV` returns all, but the first element of a vector, while `initV` returns all but the last elements of a vector. The function `takeV` returns the first  $n$  elements of a vector while the function `dropV` drops the first  $n$  elements of a vector.

```
headV :: Vector a → a
tailV :: Vector a → Vector a
lastV :: Vector a → a
initV :: Vector a → Vector a
takeV :: (Num a, Ord a) ⇒ a → Vector b → Vector b
dropV :: (Num a, Ord a) ⇒ a → Vector b → Vector b
```

The function `selectV` selects elements in the vector. The first argument gives the initial element, starting from zero, the second argument gives the stepsize between elements and the last argument gives the number of elements.

```
selectV :: (Num a, Ord a) ⇒ a → a → a → Vector b → Vector b
```

The function `groupV` groups a vector into a vector of vectors of size  $n$ .

```
groupV :: (Num a, Ord a) ⇒ a → Vector b → Vector (Vector b)
```

The data constructor `(:i)` adds an element add the front of the vector, while the operator `(:j)` adds an element at the end. The operator `j+i` concatenates two vectors. The function `concatV` concats a vector of vectors into a single vector.

```
(<<=>) :: Vector a → Vector a → Vector a
(<:)   :: Vector a → a → Vector a
```

The higher-order function `mapV` applies a function on all elements of a vector.

```
mapV :: (a → b) → Vector a → Vector b
```

The higher-order function `zipWithV` applies a function pairwise on to vectors.

```
zipWithV :: (a → b → c) → Vector a → Vector b → Vector c
```

The higher-order functions `foldlV` and `foldrV` fold a function from the right or from the left over a vector using an initial value.

```
foldlV :: (a → b → a) → a → Vector b → a
foldrV :: (b → a → a) → a → Vector b → a
```

The higher-function `filterV` takes a predicate function and a vector and creates a new vector with the elements for which the predicate is true.

```
filterV :: (a → Bool) → Vector a → Vector a
```

The function `zipV` zips two vectors into a vector of tuples. The function `unzipV` unzips a vector of tuples into two vectors.

```
zipV    :: Vector a → Vector b → Vector (a, b)
unzipV  :: Vector (a, b) → (Vector a, Vector b)
```

The function `shiftlV` shifts a value from the left into a vector. The function `shiftrV` shifts a value from the right into a vector. The functions `rotlV`, `rotrV` rotates a vector to the left or to the right. Note that these functions do not change the size of a vector.

```

shiftlV :: Vector a -> a -> Vector a
shiftrV :: Vector a -> a -> Vector a
rotrV   :: Vector a -> Vector a
rotlV   :: Vector a -> Vector a

```

The following functions are still undocumented: The function `concatV` transforms a vector of vectors to a single vector. The function `reverseV` reverses the order of elements in a vector.

```

concatV  :: Vector (Vector a) -> Vector a
reverseV :: Vector a -> Vector a

```

The function `iterateV` generates a vector with a given number of elements starting from an initial element using a supplied function for the generation of elements. The function `generateV` behaves in the same way, but starts with the application of the supplied function to the supplied value. The function `copyV` generates a vector with a given number of copies of the same element.

```

Vector> iterateV 5 (+1) 1
<1,2,3,4,5> :: Vector Integer
Vector> generateV 5 (+1) 1
<2,3,4,5,6> :: Vector Integer
Vector> copyV 7 5
<5,5,5,5,5,5,5> :: Vector Integer

```

```

iterateV :: Num a => a -> (b -> b) -> b -> Vector b
generateV :: Num a => a -> (b -> b) -> b -> Vector b
copyV     :: Num a => a -> b -> Vector b

```

The functions `scanlV` and `scanrV` "scan" a function through a vector. The functions take an initial element apply a function recursively first on the element and then on the result of the function application.

```

scanlV  :: (a -> b -> a) -> a -> Vector b -> Vector a
scanrV  :: (b -> a -> a) -> a -> Vector b -> Vector a

```

The following code is still not documented!

Reekie also proposed the `meshlV` and `meshrV` iterators. They are like a combination of `mapV` and `scanlV` or `scanrV`. The argument function supplies a pair of values: the first is input into the next application of this function, and the second is the output value. As an example consider the expression:

```
f x y = (x+y, x+y)
```

```
s1 = vector [1,2,3,4,5]
```

Here `meshlV` can be used to calculate the running sum.

```

Vector> meshlV f 0 s1
(15,<1,3,6,10,15>)

```

```

meshlV  :: (a -> b -> (a, c)) -> a -> Vector b -> (a, Vector c)
meshrV  :: (a -> b -> (c, b)) -> b -> Vector a -> (Vector c, b)

```

## 13.4 Implementation

### 13.4.1 The Data Type vector

```
instance (Show a) => Show (Vector a) where
  showsPrec p NullV = showParen (p > 9) (
    showString "<"
  )
  showsPrec p xs    = showParen (p > 9) (
    showChar '<' . showVector1 xs
  )
  where
    showVector1 NullV
      = showChar '>'
    showVector1 (x:>NullV)
      = shows x . showChar '>'
    showVector1 (x:>xs)
      = shows x . showChar ',' . showVector1 xs
```

```
instance Read a => Read (Vector a) where
  readsPrec _ s = readsVector s
```

```
readsVector :: (Read a) => ReadS (Vector a)
readsVector s = [(x:>NullV), rest] | ("<", r2) <- lex s,
                                   (x, r3) <- reads r2,
                                   (">", rest) <- lex r3]
  ++
  [(NullV, r4) | ("<", r5) <- lex s,
                 (">", r4) <- lex r5]
  ++
  [(x:>xs), r6) | ("<", r7) <- lex s,
                 (x, r8) <- reads r7,
                 ("", r9) <- lex r8,
                 (xs, r6) <- readsValues r9]
```

```
readsValues :: (Read a) => ReadS (Vector a)
readsValues s = [(x:>NullV), r1] | (x, r2) <- reads s,
                                   (">", r1) <- lex r2]
  ++
  [(x:>xs), r3) | (x, r4) <- reads s,
                 ("", r5) <- lex r4,
                 (xs, r3) <- readsValues r5]
```

### 13.4.2 Functions on the Data Type Vector

```
vector [] = NullV
vector (x:xs) = x :> (vector xs)
```

```
fromVector NullV = []
fromVector (x:>xs) = x : fromVector xs
```

```
unitV x = x :> NullV
```

```
nullV NullV = True
nullV _ = False
```

```
lengthV NullV = 0
lengthV (_:>xs) = 1 + lengthV xs
```

```
replaceV vs n x
```

```

    | n <= lengthV vs && n >= 0 = takeV n vs <+> unitV x <+> dropV (n+1) vs
    | otherwise                    = vs

NullV 'atV' _ = error "atV:_Vector_has_not_enough_elements"
(x:>_) 'atV' 0 = x
(_:>xs) 'atV' n = xs 'atV' (n-1)

headV NullV = error "headV:_Vector_is_empty"
headV (v:>_) = v

tailV NullV = error "tailV:_Vector_is_empty"
tailV (_:>vs) = vs

lastV NullV = error "lastV:_Vector_is_empty"
lastV (v:>NullV) = v
lastV (_:>vs) = lastV vs

initV NullV = error "initV:_Vector_is_empty"
initV (_:>NullV) = NullV
initV (v:>vs) = v :> initV vs

takeV 0 _ = NullV
takeV _ NullV = NullV
takeV n (v:>vs) | n <= 0 = NullV
                 | otherwise = v :> takeV (n-1) vs

dropV 0 _ = NullV
dropV _ NullV = NullV
dropV n (v:>vs) | n <= 0 = v :> vs
                 | otherwise = dropV (n-1) vs

selectV f s n vs | n <= 0
                  = NullV
                  | (f+s*n-1) > lengthV vs
                  = error "selectV:_Vector_has_not_enough_elements"
                  | otherwise
                  = atV vs f :> selectV (f+s) s (n-1) vs

groupV n v
  | lengthV v < n = NullV
  | otherwise     = selectV 0 1 n v :> groupV n (selectV n 1 (lengthV v-n) v)

NullV <+> ys = ys
(x:>xs) <+> ys = x :> (xs <+> ys)

xs <: x = xs <+> unitV x

mapV _ NullV = NullV
mapV f (x:>xs) = f x :> mapV f xs

zipWithV f (x:>xs) (y:>ys) = f x y :> (zipWithV f xs ys)
zipWithV _ _ _ = NullV

foldIV _ a NullV = a
foldIV f a (x:>xs) = foldIV f (f a x) xs

foldrV _ a NullV = a
foldrV f a (x:>xs) = f x (foldrV f a xs)

```

```

filterV _ NullV = NullV
filterV p (v:>vs) = if (p v) then
    v :> filterV p vs
    else
    filterV p vs

zipV (x:>xs) (y:>ys) = (x, y) :> zipV xs ys
zipV _ _ = NullV

unzipV NullV = (NullV, NullV)
unzipV ((x, y) :> xys) = (x:>xs, y:>ys) where (xs, ys) = unzipV xys

shiftlV vs v = v :> initV vs

shiftrV vs v = tailV vs <: v

rotrV NullV = NullV
rotrV vs = tailV vs <: headV vs

rotlV NullV = NullV
rotlV vs = lastV vs :> initV vs

concatV = foldrV (<+>) NullV

reverseV NullV = NullV
reverseV (v:>vs) = reverseV vs <: v

generateV 0 _ _ = NullV
generateV n f a = x :> generateV (n-1) f x
    where x = f a

iterateV 0 _ _ = NullV
iterateV n f a = a :> iterateV (n-1) f (f a)

copyV k x = iterateV k id x

scanlV _ _ NullV = NullV
scanlV f a (x:>xs) = q :> scanlV f q xs
    where q = f a x

scanrV _ _ NullV = NullV
scanrV f a (x:>NullV) = f x a :> NullV
scanrV f a (x:>xs) = f x y :> ys
    where ys@(y:>_) = scanrV f a xs

meshlV _ a NullV = (a, NullV)
meshlV f a (x:>xs) = (a'', y:>ys)
    where (a', y) = f a x
    (a'', ys) = meshlV f a' xs

meshrV _ a NullV = (NullV, a)
meshrV f a (x:>xs) = (y:>ys, a'')
    where (y, a'') = f x a'
    (ys, a') = meshrV f a xs

```

# Chapter 14

## The module `Memory`

### 14.1 Overview

This module contains the data structure and access functions for the memory model.

```
module Memory (  
    module AbsentExt, Memory (..), Access (..),  
    MemSize, Adr, newMem, memState, memOutput  
) where  
  
import Vector  
import AbsentExt
```

### 14.2 Data Structure

The data type `Memory` is modeled as a vector. The data type `Access` defines two access patterns, `Read adr` and `Write adr val`, where `adr` can be of any type..

```
type Adr                = Int  
type MemSize            = Int  
  
data Memory a           = Mem Adr (Vector (AbstExt a))  
                        deriving (Eq, Show)  
  
data Access a           = Read Adr  
                        | Write Adr a  
                        deriving (Eq, Show)
```

### 14.3 Functions on the data type `Memory`

The module defines the following access functions for the memory:

```
newMem                :: MemSize → Memory a  
memState              :: Memory a → Access a → Memory a  
memOutput             :: Memory a → Access a → AbstExt a
```

The function `newMem` creates a new memory, where the number of entries is given by a parameter. The function `memState` gives the new state of the memory, after an access to a memory. A `Read` operation leaves the memory unchanged. The function `memOutput` gives the output of the memory after an access to the memory. A `Write` operation gives an absent value as output.

## 14.4 Implementation of Functions

```
newMem size                = Mem size (copyV size Abst)

writeMem                   :: Memory a → (Int, a) → Memory a
writeMem (Mem size vs) (i, x)
  | i < size && i ≥ 0      = Mem size (replaceV vs i (abstExt x))
  | otherwise              = Mem size vs

readMem                    :: Memory a → Int → (AbstExt a)
readMem (Mem size vs) i
  | i < size && i ≥ 0      = vs 'atV' i
  | otherwise              = Abst

memState mem (Read _)     = mem
memState mem (Write i x) = writeMem mem (i, x)

memOutput mem (Read i)    = readMem mem i
memOutput _ (Write _ _)   = Abst
```

# Chapter 15

## The module Queue

### 15.1 Overview

The module `Queue` provides two data types, that can be used to model queue structures, such as FIFOs. There is a data type for an queue of infinite size `Queue` and one for finite size `FiniteQueue`.

### 15.2 The data type Queue

A queue is modeled as a list. The data type `FiniteQueue` has an additional parameter, that determines the size of the queue.

```
module Queue where
```

```
import AbsentExt
```

```
data Queue a      = Q [a] deriving (Eq, Show)
data FiniteQueue a = FQ Int [a] deriving (Eq, Show)
```

### 15.3 Functions on the data types Queue and FiniteQueue

Table 15.3 shows the functions an the data types `Queue` and `FiniteQueue`.

```
pushQ           :: Queue a → a → Queue a
pushListQ      :: Queue a → [a] → Queue a
popQ           :: Queue a → (Queue a, AbsentExt a)
queue          :: [a] → Queue a
pushFQ         :: FiniteQueue a → a → FiniteQueue a
pushListFQ     :: FiniteQueue a → [a] → FiniteQueue a
popFQ         :: FiniteQueue a → (FiniteQueue a, AbsentExt a)
finiteQueue    :: Int → [a] → FiniteQueue a
```

infinite	finite	description
pushQ	pushFQ	pushes one element on the queue
pushListQ	pushListFQ	pushes a list of elements on the queue
popQ	popFQ	pops one element from the queue
queue	finiteQueue	transforms a list into a queue

Table 15.1: Functions on the data types `Queue` and `FiniteQueue`

## 15.4 Implementation

```
pushQ (Q q) x           = Q (q ++ [x])
pushListQ (Q q) xs      = Q (q ++ xs)
popQ (Q [])             = (Q [], Abst)
popQ (Q (x:xs))        = (Q xs, Prst x)
queue xs                = Q xs
pushFQ (FQ n q) x       = if length q < n then
                          (FQ n (q ++ [x]))
                          else
                          (FQ n q)
pushListFQ (FQ n q) xs  = FQ n (take n (q ++ xs))
popFQ (FQ n [])         = (FQ n [], Abst)
popFQ (FQ n (q:qs))    = (FQ n qs, Prst q)
finiteQueue n xs       = FQ n (take n xs)
```

## Chapter 16

# Possible Extensions to the ForSyDe Standard Library

### 16.1 Overview

This module includes proposed extensions to the ForSyDe standard library. The Extensions are structured according to their corresponding library. The idea is, that this module serves as a collection of candidates of functions that can be included into the ForSyDe standard library in later releases.

This means, that this library is *not* stable!

```
module ForSyDeStdLibExtensions where
```

```
import ForSyDeStdLib
```

# Bibliography

- [1] Axel Jantsch, Ingo Sander, and Wenbiao Wu. The usage of stochastic processes in embedded system specifications. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, April 2001.
- [2] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *IEEE Proceedings*, 1995.
- [3] H. J. Reekie. *Realtime Signal Processing*. PhD thesis, University of Technology at Sydney, Australia, 1995.