ForSyDe-Atom

User Manual

George Ungureanu
Department of Electronics
KTH Royal Institute of Technology
ugeorge@kth.se

Contents

List of Figures

List of Case Studies

CHAPTER 1

Introduction

In this chapter we introduce the purpose, organization and usage of this document, as well as brief instructions and references for helping to set up the ForSyDe-Atom libraries. The scope is to facilitate the reader's progression through this document.

1.1 Purpose & organization

This book is a living document which gathers material related to ForSyDe-Atom and binds it in form of a user manual. Most of the text contained by this book originates from actual inline or literate source code documentation, in form of examples, tutorials, reports and even library API documentation. This means that this document evolves with the ForSyDe-Atom project itself and is periodically updated.

ForSyDe-Atom is a shallow-embedded DSL in the functional programming language Haskell for modeling cyber-physical and parallel systems. It enforces a disciplined way of modeling by separating the manifold concerns of systems into orthogonal *layers*. The ForSyDe-Atom formal framework aims to providing (where possible) a minimum set of primary common building blocks for each layer called *atoms*, capturing elementary semantics. Even so, the modeling framework provides library blocks and modules commonly used in CPS defined in terms of *patterns* of atoms. For more information on ForSyDe-Atom itself, please consult the associated scientific publications or the API extended documentation[1].

This document is structured in two parts. The first part gathers examples, tutorials and experiments in a learning progression. Each chapter associated with (and actually generated from) a Haskell Cabal project included in the `forsyde-atom-examples`[2] repository. This means that the first part of the book is meant to be read in parallel with running the associated example project which conveniently exports functions to test the listed code "on-the-fly". The second part of the book is the actual inline API documentation of ForSyDe-Atom generated with Haddock, which serves also as an extended library report and provides information both on theoretical and implementation issues.

If for any reason you have difficulties following the document or you encounter bugs or discrepancies in the code and text do not hesitate to contact the author(s) or maintainer(s) on GitHub or by email.

1.2 Getting ForSyDe-Atom

The ForSyDe-Atom EDSL can be downloaded from `https://github.com/forsyde/forsyde-atom`. The main page contains enough information for acquiring the dependencies and installing the library on your own. However, each example project from the `forsyde-atom-examples` repository

[1] currently available online at `https://forsyde.github.io/forsyde-atom/`
[2] available online at `https://github.com/forsyde/forsyde-atom-examples`

comes with a set of installation scripts written for user convenience, which should be enough for traversing this manual.

Attention!

Be advised that different chapters of the book have been written during different development stages of ForSyDe-Atom and are compatible with different library releases. The code listed throughout the book *might* be incompatible with the latest release. We strongly recommend using the installation scripts included in each project associated with a chapter, which acquire and install the right dependencies in a local sandbox.

Provided you have an OS installation where the minimum dependencies (GNU make, a Git CLI client, Haskell Platform and cabal-install) are working and accessible by your user profile, to run the `getting-started` example associated with chapter 2, you simply need to type in the terminal:

```
# download the examples
git clone https://github.com/forsyde/forsyde-atom-examples.git

# change directory to the desired project folder
cd forsyde-atom-examples/getting-started

# install the project (and dependencies) in a sandbox
make install

# open an interpreter session with the examples loaded
cabal repl
```

1.3 Using this document

Disclaimer

The document assumes that the reader is familiar with the syntax of Haskell and the usage of a Haskell interpreter (e.g. `ghci`). Otherwise, we recommend consulting at least the introductory chapters of one of the following books by Lipovača, 2011 and Hutton, 2016 or other recent books in Haskell.

Most of this document has been created using literate programming. This means that all code shown in the listings is compatible with the ForSyDe-Atom library version mentioned at the beginning of each chapter. Throughout the document you will find two listing styles. This style

```
-- | API documentation comment
myIdFunc :: a → a
myIdFunc = id
```

shows *source code* as it is found in the implementation files. We have taken the liberty to display some code characters as their literate equivalent (e.g. `->` is shown as →, `\` is shown as λ, and so on).

This style

```
Prelude> 1 + 1
2
```

suggests *interactive commands* given by the user in a terminal or an interpreter session. The listing above shows a typical `ghci` session, where the string after the prompter symbol `>` suggests the user input (e.g. `1 + 1`). Whenever relevant, the expected output is printed one row below that (e.g. `2`).

The code examples are bundled as separate Cabal packages and is provided as libraries meant to be loaded in an interpreter session in parallel with reading this document. Detailed instructions

on how to install the packages can be found in the `README.md` file in each project. The best way to install the packages is within sandboxed environments with all dependencies taken care of, usually scripted within the `make` commands. After a successful installation, to open an interpreter session pre-loaded with the main sandboxed library, you just need to type in the following command in a terminal from the package root path (the one containing the `.cabal` file):

```
# cabal repl
```

Each section of this document contains a small example written within a library *module*, like:

```
module X where
```

One can access all functions in module `X` by importing it in the interpreter session, unless otherwise noted (e.g. library `X` is re-exported by `Y`).

```
*Y> import X
```

Now suppose that function `myIdFunc` above was defined in module `X`, then one would have direct access to it, e.g.:

```
*Y X> :t myIdFunc
myIdFunc :: a -> a
*Y X> myIdFunc 3
3
```

By all means, the code for `myIdFunc` or any source code for that matter can be copied/pasted in a custom `.hs` file and compiled or used in any relevant means. The current format was chosen because it is convenient to "get your hands dirty" quickly without thinking of issues associated with compiler suites.

A final tip: if you think that the full name of `X` is polluting your prompter or is hard to use, then you can import it using an alias:

```
*Y> import Extremely.Long.Full.Name.For.X as ShortAlias
*Y ShortAlias>
```

1.4 References

Hutton, Graham (2016). *Programming in Haskell*. 2nd. New York, NY, USA: Cambridge University Press. ISBN: 1316626229, 9781316626221.

Lipovača, Miran (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. 1st. San Francisco, CA, USA: No Starch Press. ISBN: 1593272839, 9781593272838.

Part I

Examples & Reports

CHAPTER 2

Getting Started with ForSyDe-Atom

This chapter is meant to introduce the reader to using the basic features of ForSyDe-Atom library as a Haskell EDSL for modeling and simulating embedded and cyber-physical systems. It is not meant to substitute the API documentation nor provide any detail on the theoretical foundation, and it references external documents whenever necessary. It starts from modeling basics, goes through a toy example seen from different perspectives, and into more advanced features like creating custom patterns.

Contents

Info

Compatibility : ForSyDe-Atom version 0.2.1
Repository : https://github.com/forsyde/forsyde-atom-examples
Path : <repo_root>/getting-started
Other deps. : none

2.1 Goals

The main goals of this chapter are:

- introduce the reader to basic modeling features such as: importing library modules, using a Haskell interpreter, using helper functions, composing functions, designing with layers, understanding type signatures, using basic input/output.

- provide a step-by-step guide for modeling a toy system expressing concerns from four layers: function, extended behavior, model of computation and recursive/parallel composition.

- describe the above system as executing with the semantics dictated by four MoCs: synchronous dataflow (SDF), synchronous (SY), discrete event (DE) and continuous time (CT). For this purpose the system will be first instantiated multiple times using specialized helpers, and then described as a network of patterns overloaded with MoC semantics by injecting the right data types, thus exposing the polymorphism of atoms.
- briefly introduce the concepts of atoms and patterns and their usage and guide through creating custom patterns and behaviors.

2.2 The basics

This section introduces some basic modeling features of ForSyDe-Atom, such as helpers and process constructors. The module is re-exported by `AtomExamples.GettingStarted` which is pre-loaded in a `repl` session, so there is no need to import it manually.

```
module AtomExamples.GettingStarted.Basics where
```

We usually start a ForSyDe-Atom module by importing the `ForSyDe.Atom` library which provides some commonly used types and utilities.

```
import ForSyDe.Atom
```

In this section we will only test synchronous processes as patterns defined in the MoC layer. An extensive library of types, utilities and helpers for SY process constructors can be used by importing the `ForSyDe.Atom.MoC.SY` module.

```
import ForSyDe.Atom.MoC.SY
```

Next we import the `Absent` extended behavior, defined in the ExB layer, to get a glimpse of modeling using multiple layers. As with the previous, we need to specifically import the `ForSyDe.Atom.ExB.Absent` library to access the helpers and types.

```
import ForSyDe.Atom.ExB (res11, res21)
import ForSyDe.Atom.ExB.Absent
```

The *signal* is the basic data type defined in the MoC layer, and it encodes a *tag system* which describes time, causality and other key properties of CPS. In the case of SY MoC, a signal defines a total order between events. There are several ways to instantiate a signal in ForSyDe-Atom. The most usual one is to create it from a list of values using the `signal` helper. By studying its type signature in the online API documentation, one can see that it needs a list of elements of type `a` as argument, so let us create a test signal `testsig1`:

```
testsig1 = signal [1,2,3,4,5]
```

You can print or check the type of `testsig1`

```
*AtomExamples.GettingStarted> testsig1
{1,2,3,4,5}
*AtomExamples.GettingStarted> :t testsig1
testsig1 :: ForSyDe.Atom.MoC.SY.Core.Signal Integer
```

The type of `testsig1` tells us that the `signal` helper created a SY `Signal` carrying `Integer` values. If you are curious, you can print some information about this mysterious type

```
*AtomExamples.GettingStarted> :info ForSyDe.Atom.MoC.SY.Core.Signal
type ForSyDe.Atom.MoC.SY.Core.Signal a =
  Stream (ForSyDe.Atom.MoC.SY.Core.SY a)
        -- Defined in ForSyDe.Atom.MoC.SY.Core
```

which shows that it is in fact a type alias for a `Stream` of `SY` events. If this became too confusing, please read the MoC layer overview in this online API documentation page. Unfortunately the names printed as interactive information are verbose and show their exact location in the structure of ForSyDe-Atom. We do not care about this in the source code, since we imported the SY library properly. To benefit from the same treatment in the interpreter session, we need to do the same:

```
*AtomExamples.GettingStarted> import ForSyDe.Atom.MoC.SY as SY
*AtomExamples.GettingStarted SY> :info Signal
type Signal a = Stream (SY a)
        -- Defined in ForSyDe.Atom.MoC.SY.Core
```

Another way of creating a SY signal is by means of a `generate` process, which generates an infinite signal from a kernel value. By studying the `online API documentation`, you can see that the SY library provides a number of helpers for this particular process constructor, the one generating one output signal being `generate1`. Let us first check the type signature for this helper function:

```
*AtomExamples.GettingStarted SY> :t generate1
generate1 :: (b1 -> b1) -> b1 -> Signal b1
```

So basically, as suggested in the `online API documentation`, this helper takes a "next state" function of type `a -> a`, a kernel value of type `a`, and it generates a signal of tyle `Signal a`. With this in mind, let us create `testsig2`:

```
testsig2 = generate1 (+1) 0
```

Printing it would jam the terminal... we were serious when we said "infinite"! This is why you need to select a few events from the beginning to see whether the signal generator behaves correctly. To do so, we use the `takeS` utility:

```
*AtomExamples.GettingStarted SY> takeS 10 testsig2
{0,1,2,3,4,5,6,7,8,9}
*AtomExamples.GettingStarted SY> :t testsig2
testsig2 :: Signal Integer
```

`generate` was a process with no inputs. Now let us try a process that takes the two signals `testsig1` and `testsig2` and sums their synchronous events. For this we use the combinatorial process `comb`, and the SY constructor we need, with two inputs and one output is provided by `comb21`. Again, checking the type signature confirms that this is the helper we need:

```
*AtomExamples.GettingStarted SY> :t comb21
comb21 :: (a1 -> a2 -> b1) -> Signal a1 -> Signal a2 -> Signal b1
```

So let us instantiate `testproc1`:

```
testproc1 = comb21 (+)
```

Calling it in the interpreter with `testsig1` and `testsig2` as arguments, it returns:

```
*AtomExamples.GettingStarted SY> testproc1 testsig2 testsig1
{1,3,5,7,9}
```

which is the expected output, as based on the definition of the SY MoC, all events following the sixth one from `testsig2` are not synchronous to any event in `testsig1`.

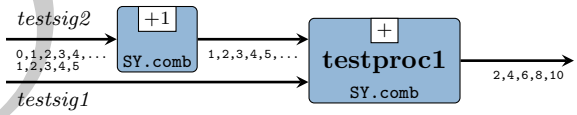


Figure 2.1: Simple process network as composition of processes

Suppose we want to increment every event of `testsig2` with 1 before summing the two signals. This particular behavior is described by the process network in fig. 2.1. There are multiple ways to instantiate this process network, mainly depending on the coding style of the user:

```
testpn1       = comb21 (+) . comb11 (+1)
testpn2 s2 s1 = testproc1 (comb11 (+1) s2) s1
testpn3 s2    = testproc1 (comb11 (+1) s2)
testpn4 s2 s1 = let s2' = comb11 (+1) s2
                in  testproc1 s2' s1
testpn5 s2    = comb21 (+)  s2'
  where s2'   = comb11 (+1) s2
```

All of the above functions are equivalent. `testpn1` uses the point-free notation, i.e. the function composition operator, between two partially-applied process constructors. `testpn2` makes use of the previously-defined `testproc1` to enforce a global hierarchy. `testpn3` is practically the same, but it exposes the partial application mechanism, by not making the `s1` argument explicit. `testpn4` makes use of local hierarchy in form of a let-binding, while `testpn5` does so through a where clause. Printing them only confirms their equivalence:

```
*AtomExamples.GettingStarted SY> testpn1 testsig2 testsig1
{2,4,6,8,10}
*AtomExamples.GettingStarted SY> testpn2 testsig2 testsig1
{2,4,6,8,10}
*AtomExamples.GettingStarted SY> testpn3 testsig2 testsig1
{2,4,6,8,10}
*AtomExamples.GettingStarted SY> testpn4 testsig2 testsig1
{2,4,6,8,10}
*AtomExamples.GettingStarted SY> testpn5 testsig2 testsig1
{2,4,6,8,10}
```

One key concept of ForSyDe-Atom is the ability to model different aspects of a system as orthogonal layers. Up until now we only experimented with two layers: the MoC layer, which concerns timing and synchronization issues, and the function layer, which concerns functional aspects, such as arithmetic and data computation. Let us rewind which DSL blocks we have used an group them by which layer they belong:

- the `Integer` values carried by the two test signals (i.e. `0, 1, ...`) and the arithmetic functions (i.e. `(+)` and `(+1)`) belong to the *function layer*.
- the signal structures for `testsig1` and `testsig2` (i.e. `Signal a`) the utility (i.e. `signal`) and the process constructors (i.e. `generate1`, `comb11` and `comb21`) belong to the *MoC layer*.

It is easy to grasp the concept of layers once you understand how *higher order functions* work, and accept that ForSyDe-Atom basically relies on the power of functional programming to define structured abstractions. In the previous case entities from the MoC layer "wrap around" entities from the function layer like in fig. 2.2, "lifting" them into the MoC domain. Unfortunately it is not that straightforward to see from the code syntax which entity belongs to which layer. For now, their membership can be determined solely by the user's knowledge of where each function is defined, i.e. in which module. Later in this guide we will make this apparent from the code syntax, but for now you will have to trust us.

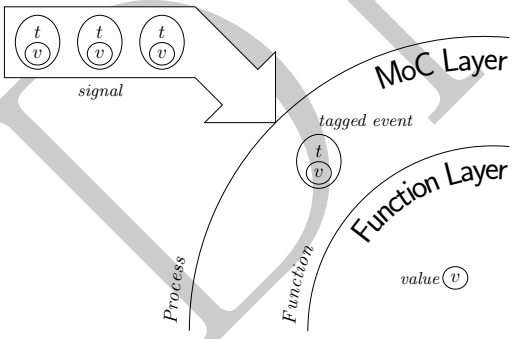


Figure 2.2: Layered structure of the processes in fig. 2.1

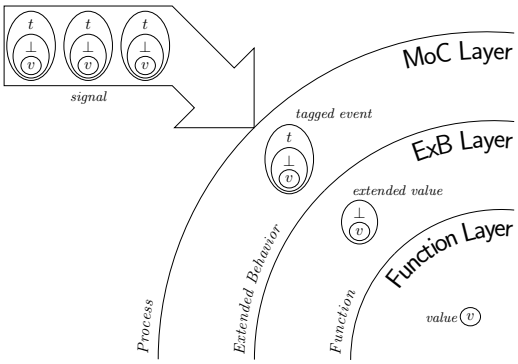


Figure 2.3: Layered structure of the processes describing absent events

As a last exercise for this section we would like to extend the behavior of the system in fig. 2.1 in order to describe whether the events are happening or not (i.e. are absent or present) and act accordingly. For this, ForSyDe-Atom defines the *Extended Behavior (ExB) layer*. As suggested in fig. 2.3, this layer extends the pool of values with symbols denoting states which would be impossible to describe using normal values, and associates some default behaviors (e.g. protocols) over these symbols.

The two processes fig. 2.1 are now defined below as `testAp1` and `testAp2`. This time, apart from the functional definition (`name = function`) we specify the type signature as well (`name :: type`), which in the most general case can be considered a specification/contract of the interfaces of the newly instantiated component. Both type signatures and function definitions expose the layered structure suggested in fig. 2.3. As specific ExB type, we use `AbstExt` and as behavior pattern constructor we choose a default behavior expressing a resolution `res`.

```
testAp1 :: Num a
        => Signal (AbstExt a) → -- ^ input signal of absent-extended values
        → Signal (AbstExt a) → -- ^ output signal of absent-extended values
testAp1 = comb11 (res11 (+1))
```

```
testAp2 :: Num a
        => Signal (AbstExt a) → -- ^ first input signal of absent-extended values
        → Signal (AbstExt a) → -- ^ second input signal of absent-extended values
        → Signal (AbstExt a) → -- ^ output signal of absent-extended values
testAp2 = comb21 (res21 (+))
```

Now all we need is to create some test signals of type `Signal (AbstExt a)`. One way is to use the `signal` utility like for `testAsig1`, but this forces to make use of `AbstExt`'s type constructors. Another way is to use the library-provided process constructor helpers, such as `filter'`, like for `testAsig2`.

```
testAsig1 = signal [Prst 1, Prst 2, Abst, Prst 4, Abst]
testAsig2 = filter' (/=4) testsig2
```

Printing out the test signals in the interpreter session this is what we get:

```
*AtomExamples.GettingStarted SY> testAsig1
{1,2,⊥,4,⊥}
*AtomExamples.GettingStarted SY> takeS 10 testAsig2
{0,1,2,3,⊥,5,6,7,8,9}
```

Trying out `testAp1` on `testAsig1`:

```
*AtomExamples.GettingStarted SY> testAp1 testAsig1
{2,3,⊥,5,⊥}
```

Everything seems all right. Now testing `testAp2` on `testAsig1` and `testAsig2`:

```
*AtomExamples.GettingStarted SY> takeS 10 $ testAp2 testAsig2 testAsig1
{1,3,*** Exception: [ExB.Absent] Illegal occurrence of an absent and present event
```

Uh oh... Actually this *is* the correct behavior of a resolution function for absent events, as defined in synchronous reactive languages such as Lustre Halbwachs et al., 1991. Let us remedy the situation, but this time using another library-provided process constructor, `when'`.

```
testAsig2' = when' mask testsig2
  where
    mask = signal [True, True, False, True, False]
```

Now printing `testAp2` looks better:

```
*AtomExamples.GettingStarted SY> testAsig2'
{0,1,⊥,3,⊥}
*AtomExamples.GettingStarted SY> testAp2 testAsig2' testAsig1
{1,3,⊥,7,⊥}
```

And recreating the process network from fig. 2.1 gives the expected result:

```
testApn1 = testAp2 . testAp1
```

```
*AtomExamples.GettingStarted SY> testApn1 testAsig2' testAsig1
{2,4,⊥,8,⊥}
```

This section has provided a crash course in modeling with FORSYDE-ATOM, with focus on a few practical matters, such as using library-provided helpers and constructors and understanding the role of layers. The following sections delve deeper into modeling concepts such as atoms and making use of ad-hoc polymorphism.

2.2.1 Visualizing your data

Up until now, we have made use of the `Show` instance of the ForSyDe-Atom data types to print out signals on the terminal screen. While this remains the main way to test if a model is working properly, there are alternative ways to plot data. This section introduces the reader to the `ForSyDe.Atom.Utility.Plot` library of utilities for visualizing signals or other data types.

The functions presented in this section are defined in the following module, which is exported by `AtomExamples.GettingStarted`.

```
module AtomExamples.GettingStarted.Plot where
```

We will be using the signals defined in the previous section, so let us import the corresponding module:

```
import AtomExamples.GettingStarted.Basics
```

And, as mentioned, we need to import the library with plotting utilities:

```
import ForSyDe.Atom.Utility.Plot
```

Upon consulting the API documentation for this module, you might notice that most utilities input a so-called `PlotData` type, which is an alias for a complex structure carrying configuration parameters, type information and data samples. Using Haskell's type classes, ForSyDe-Atom is able to provide few polymorphic utilities for converting most of the useful types into `PlotData`.

For example, the `prepare` function takes a "plottable" data type (e.g. a signal of values), and a `Config` type, and returns `PlotData`. The `Config` type is merely a record of configuration parameters useful further down in the plotting pipeline. At the time of writing this report[1], a configuration record looked like this:

```
config =
  Cfg { path    = "./fig"     -- path where the eventual data files are dumped
      , file    = "plot"      -- base name of the eventual files generated
      , rate    = 0.01        -- sampling rate if relevant (e.g. ignored by SY signals).
                              -- Useful just for e.g. explicit-timed signals.
      , xmax    = 20          -- maximum x coordinate. Necessary for infinite signals.
      , labels  = ["s1","s2"] -- labels for all signals passed to be plotted.
      , verbose = True        -- prints additional messages for each utility.
      , fire    = True        -- if relevant, fires a plotting or compiling program.
      , mklatex = True        -- if relevant, dumps a LaTeX script loading the plot.
      , mkeps   = True        -- if relevant. dumps a PostScript file with the plot.
      , mkpdf   = True        -- if relevant, dumps a PDF file with the plot.
      }
```

`ForSyDe.Atom.Utility.Plot` provides several of these pre-made configuration objects, which can be modified on-the-fly using Haskell's record syntax, as you will see further on.

Let us see again the signals `testsig1` and `testsig2` defined in the previous section:

```
λ> testsig1
{1,2,3,4,5}
λ> takeS 20 testsig2
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19}
```

The utility `showDat` prints out sampled data on the terminal, as pairs of X and Y coordinates:

```
show1 = showDat $ prepare config testsig1
show2 = let cfg = config {xmax=15, labels=["testsig1","testsig2"]}
        in  showDat $ prepareL cfg [testsig1,testsig2]
```

```
λ> show1
s1 =
        0   1.0
        1   2.0
        2   3.0
        3   4.0
        4   5.0
<
λ> show2
```

[1] ForSyDe-Atom v0.2.1

```
testsig1 =
        0   1.0
        1   2.0
        2   3.0
        3   4.0
        4   5.0

testsig2 =
        0   0.0
        1   1.0
        2   2.0
        3   3.0
        4   4.0
        5   5.0
        6   6.0
        7   7.0
        8   8.0
        9   9.0
        10   10.0
        11   11.0
        12   12.0
        13   13.0
        14   14.0
```

The function `dumpDat` dumps the data files in a path specified by the configuration object. Based on the `config` object instantiated earlier, after calling the following function you should see a new folder called `fig` in the current path, with two new `.dat` files.

```
dump2 = let cfg = config {xmax=15, labels=["testsig1","testsig2"]}
        in  dumpDat $ prepareL cfg [testsig1,testsig2]
```

```
λ> dump2
Dumped testsig1, testsig2 in ./fig
["./fig/testsig1.dat","./fig/testsig2.dat"]
```

The plot library also has a few functions which create and (optionally) fire Gnuplot scripts. In order to make use of them, you need to install the dependencies mentioned in the API documentation. For example, using the function `plotGnu` creates the following plot:

```
plot2 = let cfg = config {xmax=8, labels=["testsig1","testsig2"]}
        in  plotGnu $ prepareL cfg [testsig1,testsig2]
```

Different input data creates different types of plots, as we will see in future sections.

One can also generate LaTeX code which is meant to be compiled with the ForSyDe-LaTeX package, more specifically its signal plotting library. Check the user manual for more details on how to install the dependencies and how to use the library itself. Naturally, there is a function `showLatex` which prints out the command for a signals environment defined in ForSyDe-LaTeX:

```
latex1 = let cfg = config {xmax=8, labels=["testsig1","testsig2"]}
         in  showLatex $ prepareL cfg [testsig1,testsig2]
```

```
λ> latex1
  \begin{signalsSY}[]{8.0}
    \signalSY[]{1.0:0,2.0:1,3.0:2,4.0:3,5.0:4}
    \signalSY[]{0.0:0,1.0:1,2.0:2,3.0:3,4.0:4,5.0:5,6.0:6,7.0:7}
  \end{signalsSY}
<
```

Also, there is a command `plotLatex` for generating a standalone LaTeX document and, if possible, compiling it with `pdflatex`. For example, calling the following function generates the image from section 2.2.1.

```
latex2 = let cfg = config {xmax=8, labels=["testsig1","testsig2"]}
         in  plotLatex $ prepareL cfg [testsig1,testsig2]
```

The SY signal plot is nothing spectacular, but wraps the events in a matrix of nodes which can be embedded into a more complex TikZ figure. Other signals produce other plots. Most generated plots will need manual tweaking in order to look good. Check the user manual on how to customize each plot.

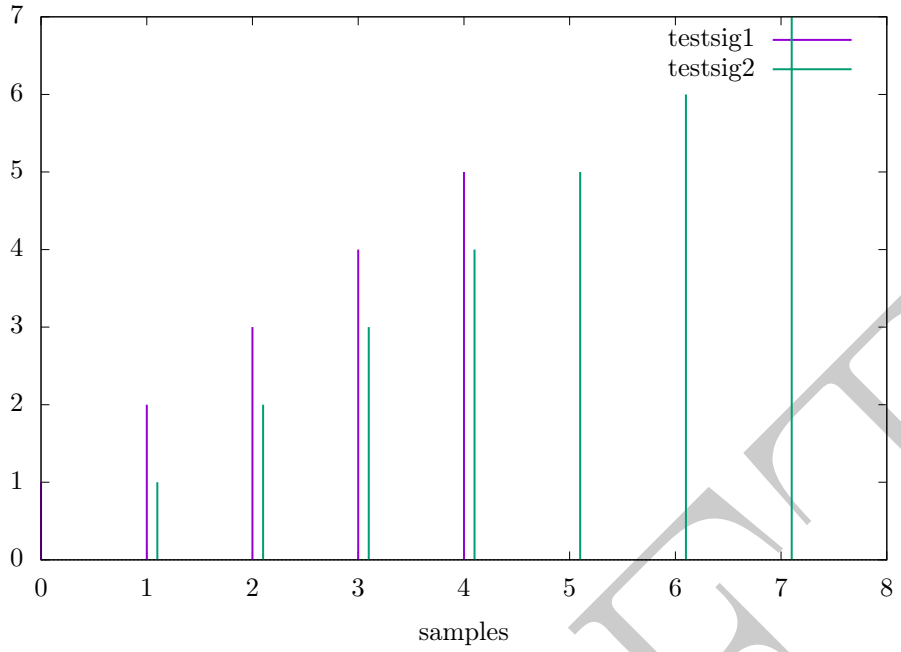


Figure 2.4: SY signal in Gnuplot as a impulse plot

1.0	2.0	3.0	4.0	5.0			
0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0

Figure 2.5: SY signal plot in ForSyDe-LaTeX as a matrix of nodes

2.3 Toy example: a focus on MoCs

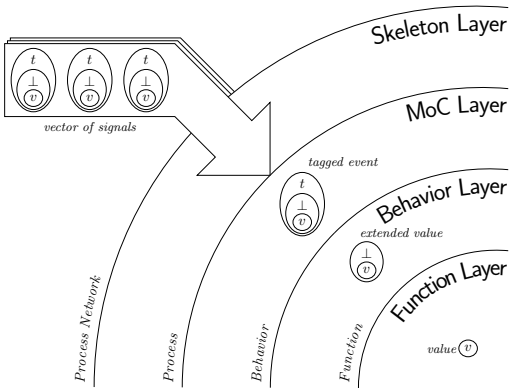
This example has been used as a case study for introducing the new concepts of ForSyDe-Atom in the paper of Ungureanu and Sander, 2017. It describes the simple system from fig. 2.6b which exposes four layers, structured like in fig. 2.6a. This system is then fed vectors of signals describing different MoCs and its response is observed. In figs. 2.6c to 2.6f some possible projections on the different layers are depicted. For now they are used just as trivia, and you need not bother with them that much.

This is a synthetic example meant to introduce as many concepts as possible in a short amount of time and, among others, it highlights:

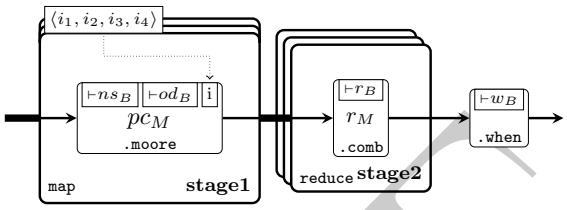
- the power of partial application for creating parameterized structures, such as the process network for `stage1`.
- alternative designs for the same `toy` system to show the effect of MoCs. First it is instantiated using different process constructor helpers defined for each MoC separately. Afterwards it is written as one single polymorphic instance using MoC layer patterns, overloaded with execution semantics in accordance with the tag system injected into the system.

For the sake of brevity, we also provide the functional description in the language introduced by Ungureanu and Sander, 2017 in eqs. (2.1)–(2.5) and table 2.1. Do not bother much about this notation either, as this exact definition will appear in the code in a more "human readable" form.

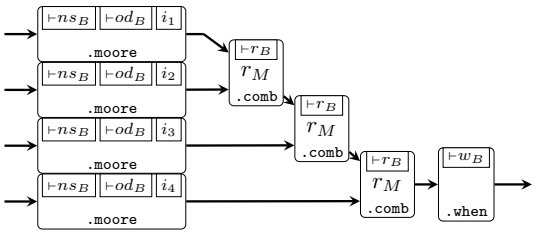
$$\texttt{toy} : \langle V \rangle \to \langle S \rangle \to S \tag{2.1}$$
$$\texttt{toy}\langle i \rangle\langle s \rangle = \texttt{when}_{\text{M}}(\Gamma_w \vdash w_{\text{B}}) \circ \texttt{reduce}_{\text{S}}(r_{\text{M}}) \circ \texttt{map}_{\text{S}}(pc_{\text{M}})\langle i \rangle\langle s \rangle$$



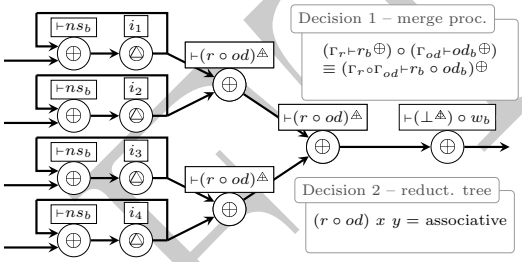
(a) The four layered structure of the toy example



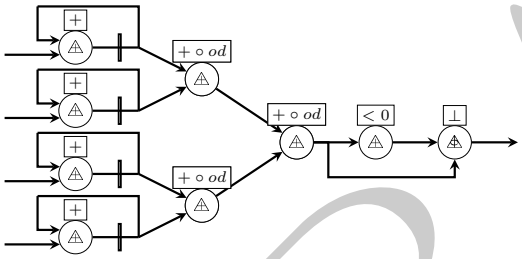
(b) View from skeleton layer



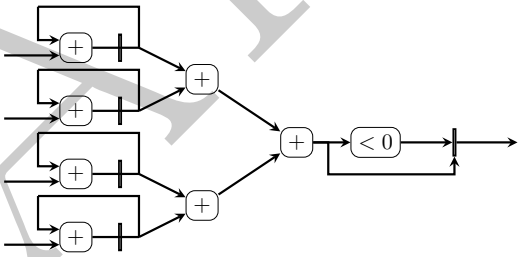
(c) Top view from process layer



(d) Flattened & refined view from process layer



(e) View from behavior layer, as projected by a timed MoC



(f) View from function layer, as projected by a timed MoC

Figure 2.6: Views and projections for the toy system

where

$$\texttt{when}_{\text{M}}(\Gamma_w \vdash w_{\text{B}})(s) = ((\bot \ \triangleq) \circ (\Gamma_w \vdash w_{\text{B}})) \oplus s \tag{2.2}$$

$$r_{\text{M}}(x, y) = \Gamma_r \vdash r_{\text{B}} \oplus (x, y) \tag{2.3}$$

$$\texttt{map}_{\texttt{s}}(pc_{\text{M}})\langle v\rangle\langle s\rangle = pc_{\text{M}} \diamond \langle v\rangle \diamond \langle s\rangle \tag{2.4}$$

$$pc_{\text{M}}(x, y) = \texttt{moore}_{\text{M}}(\Gamma_{ns} \vdash ns_{\text{B}}, \Gamma_{od} \vdash od_{\text{B}}, x)(y) \tag{2.5}$$

Table 2.1: Contexts, Functions and Initial Tokens for the System in Eq. (2.1)

MoC	$\Gamma_w \vdash \quad w_{\text{B}}(x)$	$\Gamma_r \quad \vdash \quad r_{\text{B}}(x, y)$	$\Gamma_{ns} \quad \vdash \quad ns_{\text{B}}(x, y)$	$\Gamma_{od} \vdash od_{\text{B}}(x)$	$\langle i\rangle = \langle (t, v)\rangle$
SDF[2]	$2{,}2 \vdash (x_1 < 0, x_2 < 0)^{\triangleq}$	$(1{,}1){,}1 \vdash (x_1 + y_1)^{\triangleq}$	$(1{,}2){,}1 \vdash (x_1 + y_1 + y_2)^{\triangleq}$	$1{,}1 \vdash \ x_1{}^{\triangleq}$	$\langle \ (\ ,-1) \quad (\ ,1) \quad (\ ,-1) \quad (\ ,1) \ \rangle$
SY	$\vdash \quad (x < 0)^{\triangleq}$	$\vdash \ (x + y)^{\triangleq}$	$\vdash \quad (x + y)^{\triangleq}$	$\vdash \ x \ ^{\triangleq}$	$\langle \ (\ ,-1) \quad (\ ,1) \quad (\ ,-1) \quad (\ ,1) \ \rangle$
DE	$\vdash \quad (x < 0)^{\triangleq}$	$\vdash \ (x + y)^{\triangleq}$	$\vdash \quad (x + y)^{\triangleq}$	$\vdash \ x \ ^{\triangleq}$	$\langle \ (0.5,-1) \quad (1.4,1) \quad (1.0,-1) \quad (1.4,1) \ \rangle$
CT	$\vdash \quad (x < 0)^{\triangleq}$	$\vdash \ (x + y)^{\triangleq}$	$\vdash \quad (x + y)^{\triangleq}$	$\vdash \ x \ ^{\triangleq}$	$\langle (1, \lambda t \to -1) \quad (1.4, \lambda t \to 1) \quad (1, \lambda t \to -1) \quad (1.4, \lambda t \to 1) \rangle$

[2] $\Gamma_{\text{SDF}} = (\text{consumption rate for first input}[, \text{consumption rate for second input}]), \text{production rate}$

2.3.1 Test input signals

In the following examples we will use a set of test signals defined in the following module, which is also re-exported by `AtomExamples.GettingStarted` (i.e. you don't need to import it):

```
module AtomExamples.GettingStarted.TestSignals where
```

The test signals need to define tag systems belonging to different MoCs. Each MoC has an own dedicated module under `ForSyDe.Atom.MoC` which defines atoms, patterns, types and utilities. Just like in the previous section, we need to import the needed modules. This time we name them using short aliases, to disambiguate between the different DSL items, often sharing the same name, but defined in different places.

```
import ForSyDe.Atom.ExB.Absent (AbstExt(..))
import ForSyDe.Atom.MoC.SY          as SY
import ForSyDe.Atom.MoC.DE          as DE
import ForSyDe.Atom.MoC.CT          as CT
import ForSyDe.Atom.MoC.SDF         as SDF
import ForSyDe.Atom.MoC.Time        as T
import ForSyDe.Atom.MoC.TimeStamp   as Ts
import ForSyDe.Atom.Skeleton.Vector as V
import ForSyDe.Atom.Utility.Plot
```

Let the signals `sdf1`–`sdf4` denote four SDF signals, i.e. sequences of events. Instead of using the `signal` utility, we use `readSignal` which reads a string, tokenizes it and converts it to a SDF signal. This utility function needs to be "steered" into deciding which data type to output so in order to specify the type signature we use the inline Hakell syntax `name = definition :: type`. All events, although extended, are present.

```
sdf1 = SDF.readSignal "{ 1, 1, 1, 1, 1, 1}" :: SDF.Signal (AbstExt Int)
sdf2 = SDF.readSignal "{-1, 1,-1, 1,-1, 1}" :: SDF.Signal (AbstExt Int)
sdf3 = SDF.readSignal "{ 0, 0, 1, 1, 0   }" :: SDF.Signal (AbstExt Int)
sdf4 = SDF.readSignal "{-1,-1,-1,-1,-1   }" :: SDF.Signal (AbstExt Int)
```

Similarly, let the signals `sy1`–`sy4` denote four SY signals, i.e. all events are synchronized with each other. We use the SY version of `readSignal`, also "steered" by declaring the types inline, and all events are also present.

```
sy1 = SY.readSignal "{ 1, 1, 1, 1, 1, 1}" :: SY.Signal (AbstExt Int)
sy2 = SY.readSignal "{-1, 1,-1, 1,-1, 1}" :: SY.Signal (AbstExt Int)
sy3 = SY.readSignal "{ 0, 0, 1, 1, 0   }" :: SY.Signal (AbstExt Int)
sy4 = SY.readSignal "{-1,-1,-1,-1,-1   }" :: SY.Signal (AbstExt Int)
```

For the DE signals `de1`–`de4` we need to specify for each event an explicit tag (i.e. timestamp), as required by the DE tag system. For this, the DE version of `readSignal` reads each event using the syntax `value@timestamp`. Needless to say, all events are also present.

```
de1 = DE.readSignal "{ 1@0                                  }":: DE.Signal (AbstExt Int)
de2 = DE.readSignal "{-1@0, 1@0.7,-1@1.4, 1@2.1,-1@2.8, 1@3.5}":: DE.Signal (AbstExt Int)
de3 = DE.readSignal "{ 0@0,        1@1.4,        0@2.8      }":: DE.Signal (AbstExt Int)
de4 = DE.readSignal "{-1@0                                  }":: DE.Signal (AbstExt Int)
```

Let `ct1`–`ct4` denote four CT signals. As the events in a CT signal are themselves continuous functions of time, we cannot specify them as mere strings, thus we cannot use a `readSignal` utility any more. This time we will use the CT version of the `signal` utility, where each event is specified as a tuple `(timestamp, `$f(t)$`)`, and can be considered as a continuous sub-signal. For representing time we use an alias `Time` for `Rational`, defined in the `ForSyDe.Atom.MoC.Time`. This module also contains utility functions of time, such as the constant function `const` or the sine `sin`. We define local functions to wrap the type returned by a CT subsignal into an `AbstExt` type.

```
pconst = T.const . Prst
ct1 = CT.signal [(0,pconst 1)]                                 :: CT.Signal (AbstExt Time)
ct2 = CT.signal [(0,Prst . (λt → T.sin (T.pi * t)))]           :: CT.Signal (AbstExt Time)
ct3 = CT.signal [(0,pconst 0),(1.4,pconst 1),(2.8,pconst 0)] :: CT.Signal (AbstExt Time)
ct4 = CT.signal [(0,pconst (-1))]                              :: CT.Signal (AbstExt Time)
```

Finally, we need to bundle these signals into vectors of signals, to feed into the `toy` system from fig. 2.6b and eq. (2.1). For this purpose we pass the four signals of each MoC as a list to the `vector` utility.

```
vsdf = V.vector [sdf1, sdf2, sdf3, sdf4] :: V.Vector (SDF.Signal (AbstExt Int))
vsy  = V.vector [ sy1,  sy2,  sy3,  sy4] :: V.Vector ( SY.Signal (AbstExt Int))
vde  = V.vector [ de1,  de2,  de3,  de4] :: V.Vector ( DE.Signal (AbstExt Int))
vct  = V.vector [ ct1,  ct2,  ct3,  ct4] :: V.Vector ( CT.Signal (AbstExt Time))
```

Now we need to create for each MoC the vectors with the initial states for the Moore machines, i.e. $\langle i \rangle$ from table 2.1. For SDF, SY and DE we are also making use of the fact that the defined data types are Readable. The data types that we need within the vectors are in accordance to the `moore` process constructor helpers defined in each module, so be sure to check the online API documentation to understand why we need those particular types. For example, SDF requires a partition (list) of values as initial states whereas DE apart from a value it requires also the duration of the first event. For the CT vector, as before, we cannot read functions as string, so we use the `vector` utility. `ict` also shows the usage of the `milisec` utility which converts an integer into a timestamp.

```
isdf = read "<[-1],[ 1],[-1],[ 1]>" :: V.Vector [AbstExt Int]
isy  = read "<(-1),  1, (-1),  1 >" :: V.Vector (AbstExt Int)
ide  = read "<(1,-1),(1.4, 1),(1.0,-1),(1.4, 1)>"
                                    :: V.Vector (TimeStamp, AbstExt Int)
ict  = V.vector [
  (Ts.milisec 1000, pconst (-1)),
  (Ts.milisec 1400, pconst   1 ),
  (Ts.milisec 1000, pconst (-1)),
  (Ts.milisec 1400, pconst   1 )]   :: V.Vector (TimeStamp, Time -> AbstExt Time)
```

For the polymorphic instance in section 2.3.6 we need to provide the initial states as wrapped in signals, so we create unit signals:

```
sisdf = SDF.signal <$> isdf
sisy  = SY.unit    <$> isy
side  = DE.unit    <$> ide
sict  = CT.unit    <$> ict
```

Finally, let us instantiate some plotting utilities, to test the different signals throughout the experiments:

```
plot  until lbls =    plotGnu . prepare defaultCfg {xmax=until, labels=lbls, rate=0.01}
latex until lbls =  plotLatex . prepare defaultCfg {xmax=until, labels=lbls, rate=0.01}
plotV  until lbls =    plotGnu . prepareV defaultCfg {xmax=until, labels=lbls, rate=0.01}
latexV until lbls =  plotLatex . prepareV defaultCfg {xmax=until, labels=lbls, rate=0.01}
```

2.3.2 SY instance

The SY instance of the `toy` system is created using process constructor helpers defined in `ForSyDe.Atom.MoC.SY`, and is defined in the following module (re-exported by `AtomExamples.GettingStarted`).

```
module AtomExamples.GettingStarted.SY where
```

As in the previous examples we import the modules we need, and use aliases for referencing them in the code.

```
import ForSyDe.Atom
import ForSyDe.Atom.ExB.Absent (AbstExt)
import ForSyDe.Atom.ExB             as ExB
import ForSyDe.Atom.MoC.SY          as SY
import ForSyDe.Atom.Skeleton.Vector as V
```

Although Haskell's type engine can infer these type signatures, for the sake of documenting the interfaces for each stage, we will explicitly write their types. First, `stage1` is defined as a `farm` network of `moore` processes, where the initial states are provided by a vector. Its definition makes use of partial application (i.e. arguments which are not explicitly written are supposed to be the same on the LHS as on the RHS). It is defined hierarchically, making use of local name bindings after the `where` keyword.

```
stage1SY :: V.Vector (AbstExt Int)            -- ^ vector of initial states
         -> V.Vector (SY.Signal (AbstExt Int)) -- ^ vector of input signals
         -> V.Vector (SY.Signal (AbstExt Int)) -- ^ vector of output signals
```

```
stage1SY = V.farm21 pcSY
  where
    pcSY = SY.moore11 ns od
    ns   = ExB.res21 (+)
    od   = ExB.res11 id
```

Let us print and plot the inputs against the outputs, using the test signals and plotting functions `latexV` and `plotV` defined in section 2.3.1:

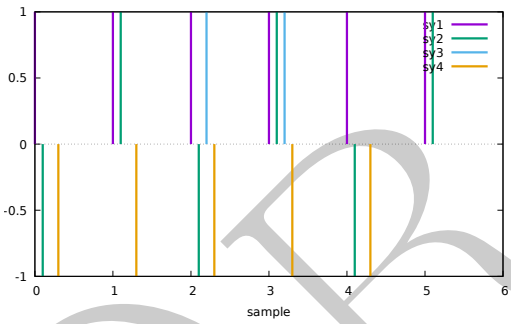
```
λ> isy
<-1,1,-1,1>
λ> vsy
<{1,1,1,1,1,1},{-1,1,-1,1,-1,1},{0,0,1,1,0},{-1,-1,-1,-1,-1}>
λ> stage1SY isy vsy
<{-1,0,1,2,3,4,5},{1,0,1,0,1,0,1},{-1,-1,-1,0,1,1},{1,0,-1,-2,-3,-4}>
λ> let latexIn = latexV 6 ["sy1","sy2","sy3","sy4"] vsy
λ> let latexS1 = latexV 7 ["sy1-1","sy2-1","sy3-1","sy4-1"] $ stage1SY isy vsy
λ> let gnuIn = plotV 6 ["sy1","sy2","sy3","sy4"] vsy
λ> let gnuS1 = plotV 7 ["sy1-1","sy2-1","sy3-1","sy4-1"] $ stage1SY isy vsy
```

1	1	1	1	1	1
-1	1	-1	1	-1	1
0	0	1	1	0	
-1	-1	-1	-1	-1	

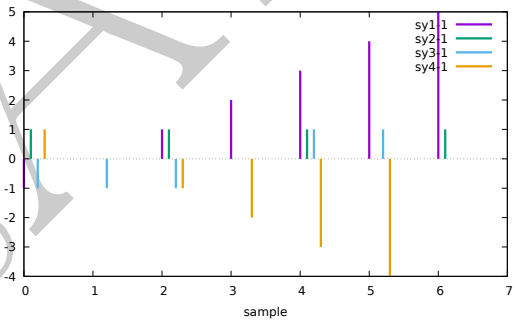
(a) `latexIn`

-1	0	1	2	3	4	5
1	0	1	0	1	0	1
-1	-1	-1	0	1	1	
1	0	-1	-2	-3	-4	

(b) `latexS1`



(c) `gnuIn`



(d) `gnuS1`

The second stage of the `toy` system in fig. 2.6b is defined as a `reduce` network of `comb` processes. As seen in its type signature, it inputs a vector of signals and it reduces it to a single signal.

```
stage2SY :: V.Vector (SY.Signal (AbstExt Int))
         → SY.Signal (AbstExt Int)
stage2SY = V.reduce rSY
  where
    rSY = SY.comb21 (ExB.res21 (+))
```

Again, let us print and plot the output signals using the test inputs and utilities defined in section 2.3.1.

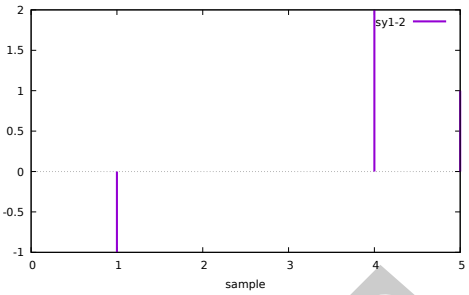
```
λ> let s2out = (stage2SY . stage1SY isy) vsy
λ> s2out
{0,-1,0,0,2,1}
λ> let latexS2 = latex 7 ["sy1-2","sy2-2","sy3-2","sy4-2"] s2out
λ> let gnuS2 = plot 7 ["sy1-2","sy2-2","sy3-2","sy4-2"] s2out
```

Finally, the last stage of the `toy` system applies a `filter` pattern on the reduced signal to mark all values less than 0 as absent.

```
stage3SY :: SY.Signal (AbstExt Int)
         → SY.Signal (AbstExt Int)
```

0 -1 0 0 2 1

(a) latexS2



(b) gnuS2

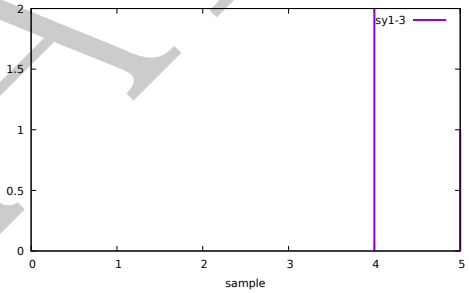
```
stage3SY = SY.filter (>=0)
>
toySY :: V.Vector (AbstExt Int)             -- ^ initial states
      → V.Vector (SY.Signal (AbstExt Int)) -- ^ input
      → SY.Signal (AbstExt Int)             -- ^ output
toySY i = stage3SY . stage2SY . stage1SY i
```

We print and plot the system response to the test signals defined in section 2.3.1.

```
λ> toySY isy vsy
{0,⊥,0,0,2,1}
λ> let latexS3 = latex 6 ["sy1-3"] $ toySY isy vsy
λ> let gnuS3   = plot  6 ["sy1-3"] $ toySY isy vsy
```

0 ⊥ 0 0 2 1

(a) latexS3



(b) gnuS3

2.3.3 DE instance

The DE instance of the `toy` looks exactly the same as the SY instance in section 2.3.2, but is created using constructors from the `ForSyDe.Atom.MoC.DE` module. This is why we will skip most of the description, and jump straight to testing it. The following file, as you are used to by now, is re-exported by `AtomExamples.GettingStarted`.

```
module AtomExamples.GettingStarted.DE where
```

As previously, we use aliases for the imported modules.

```
import ForSyDe.Atom
import ForSyDe.Atom.ExB.Absent (AbstExt)
import ForSyDe.Atom.ExB             as ExB
import ForSyDe.Atom.MoC.DE          as DE
import ForSyDe.Atom.Skeleton.Vector as V
```

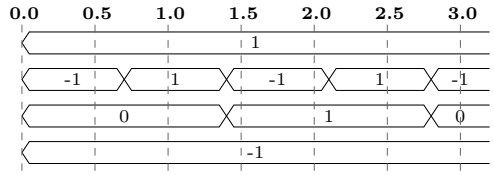
Again, we make the type signatures explicit for documentation purpose. For `stage1` we use the same `farm` network but now using DE `moore` processes.

```
stage1DE :: V.Vector (TimeStamp, AbstExt Int)  -- ^ vector of initial states
         → V.Vector (DE.Signal (AbstExt Int)) -- ^ vector of input signals
         → V.Vector (DE.Signal (AbstExt Int)) -- ^ vector of output signals
```

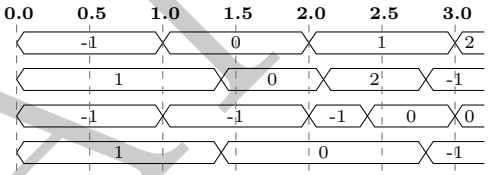
```
stage1DE = V.farm21 pcDE
  where
    pcDE = DE.moore11 ns od
    ns   = ExB.res21 (+)
    od   = ExB.res11 id
```

When printing `ide` and `vde` we can see the effects of rounding the input floating point numbers to the nearest discrete timestamp. We also have to take into account that the DE version of the Moore machine produces infinite signals when we print them out. Notice that the generated graphs may need to be tweaked in order to show the information properly.

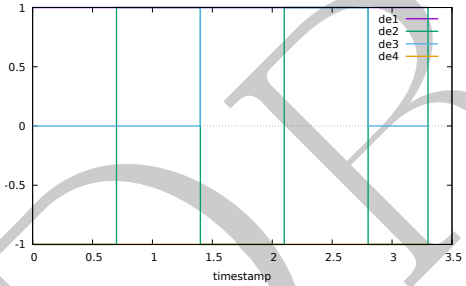
```
λ> ide
<(1s,-1),(1.399999999999s,1),(1s,-1),(1.399999999999s,1)>
λ> vde
<{ 1 @0s},{ -1 @0s, 1 @0.699999999999s, -1 @1.399999999999s, 1 @2.1s, -1 @2.799999999999s, 1 @3.5s
    },{ 0 @0s, 1 @1.399999999999s, 0 @2.799999999999s},{ -1 @0s}>
λ> fmap (takeS 6) $ stage1DE ide vde
<{ -1 @0s, 0 @1s, 1 @2s, 2 @3s, 3 @4s, 4 @5s},{ 1 @0s, 0 @1.399999999999s, 2 @2.099999999998s, -1
    @2.799999999998s, 1 @3.499999999997s, 3 @3.499999999999s},{ -1 @0s, -1 @1s, -1 @2s, 0 @2
    .399999999999s, 0 @3s, 1 @3.399999999999s},{ 1 @0s, 0 @1.399999999999s, -1 @2.799999999998s, -2
     @4.199999999997s, -3 @5.599999999996s, -4 @6.999999999995s}>
λ> let latexIn = latexV 3.3 ["de1","de2","de3","de4"] vde
λ> let latexS1 = latexV 3.3 ["de1-1","de2-1","de3-1","de4-1"] $ stage1DE ide vde
λ> let gnuIn = plotV 3.3 ["de1","de2","de3","de4"] vde
λ> let gnuS1 = plotV 3.3 ["de1-1","de2-1","de3-1","de4-1"] $ stage1DE ide vde
```



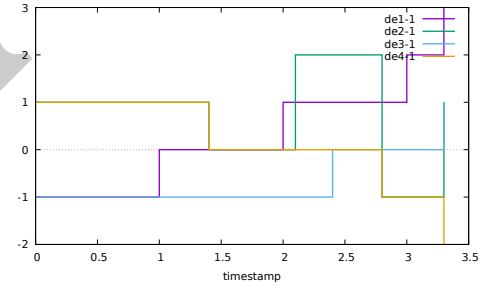
(a) `latexIn`



(b) `latexS1`



(c) `gnuIn`



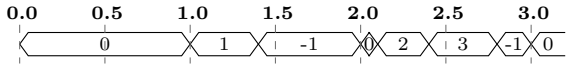
(d) `gnuS1`

The second stage according to fig. 2.6b is also defined as a `reduce` network but this time we use the DE process constructor for the `comb` processes.

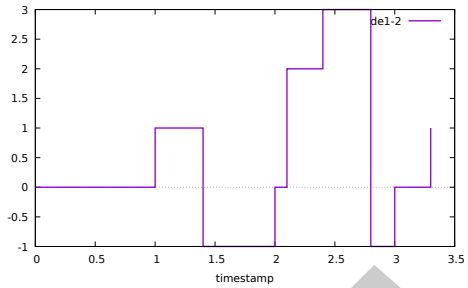
```
stage2DE :: V.Vector (DE.Signal (AbstExt Int))
         → DE.Signal (AbstExt Int)
stage2DE = V.reduce rDE
  where
    rDE = DE.comb21 (ExB.res21 (+))
```

```
λ> let s2out = (stage2DE . stage1DE ide) vde
λ> takeS 10 s2out
{ 0 @0s, 1 @1s, -1 @1.399999999999s, 0 @2s, 2 @2.099999999998s, 3 @2.399999999999s, -1 @2
    .799999999998s, 0 @3s, 1 @3.399999999999s, 3 @3.499999999997s}
λ> let latexS2 = latex 3.3 ["de1-2","de2-2","de3-2","de4-2"] s2out
λ> let gnuS2 = plot 3.3 ["de1-2","de2-2","de3-2","de4-2"] s2out
```

For the last stage of the `toy` system there is no DE process constructor in `ForSyDe.Atom.MoC.DE` so we need to create it ourselves.



(a) latexS2

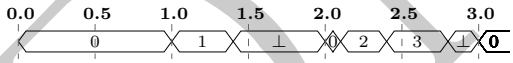


(b) gnuS2

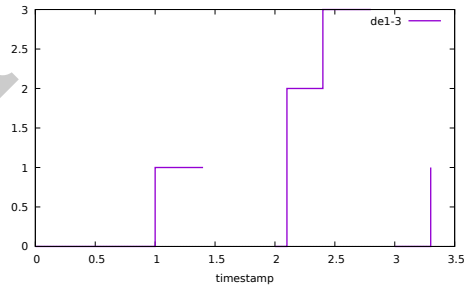
```
stage3DE :: DE.Signal (AbstExt Int)
         → DE.Signal (AbstExt Int)
stage3DE = deFilter (>=0)
  where
    deFilter p s = DE.comb21 ExB.filter (predSig p s) s
    predSig  p s = DE.comb11 (ExB.res11 p) s

toyDE :: V.Vector (TimeStamp, AbstExt Int)  -- ^ initial states
      → V.Vector (DE.Signal (AbstExt Int))  -- ^ input
      → DE.Signal (AbstExt Int)             -- ^ output
toyDE i = stage3DE . stage2DE . stage1DE i
```

```
λ> takeS 10 $ toyDE ide vde
{ 0 @0s, 1 @1s, ⊥ @1.399999999999s, 0 @2s, 2 @2.099999999998s, 3 @2.399999999999s, ⊥ @2
    .799999999998s, 0 @3s, 1 @3.399999999999s, 3 @3.499999999997s}
λ> let latexS3 = latex 3.3 ["de1-3"] $ toyDE ide vde
λ> let gnuS3   = plot  3.3 ["de1-3"] $ toyDE ide vde
```



(a) latexS3



(b) gnuS3

2.3.4 CT instance

The CT instance of the `toy` looks exactly the same as the previous ones in section 2.3.2 and **??**, but is created using constructors from the `ForSyCt.Atom.MoC.CT` module. The following file, as you are used to by now, is re-exported by `AtomExamples.GettingStarted`.

```
module AtomExamples.GettingStarted.CT where
```

As prevoiusly, we use aliases for the imported modules.

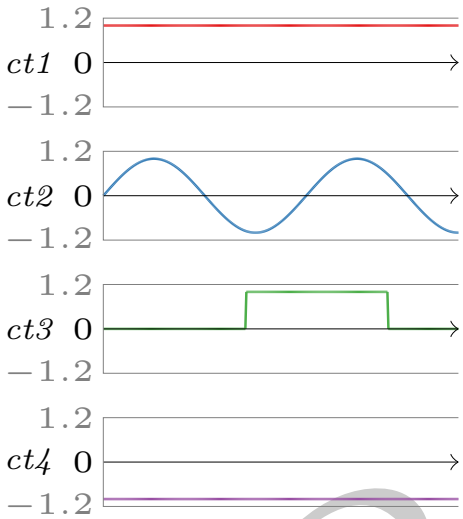
```
import ForSyDe.Atom
import ForSyDe.Atom.ExB.Absent (AbstExt)
import ForSyDe.Atom.ExB             as ExB
import ForSyDe.Atom.MoC.CT          as CT
import ForSyDe.Atom.Skeleton.Vector as V
```

Again, `stage1` is a `farm` network of CT `moore` processes.

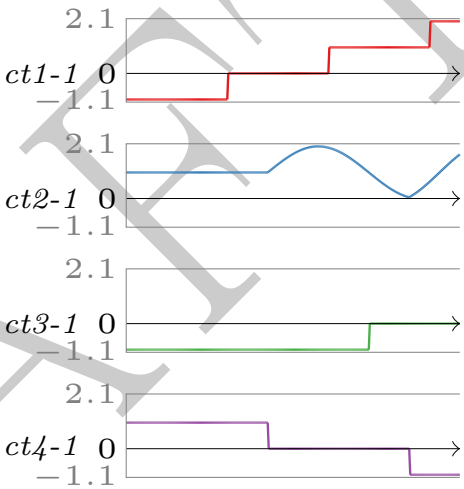
```
stage1CT :: V.Vector (TimeStamp, Time → AbstExt Time)  -- ^ vector of initial states
         → V.Vector (CT.Signal (AbstExt Time))         -- ^ vector of input signals
         → V.Vector (CT.Signal (AbstExt Time))         -- ^ vector of output signals
stage1CT = V.farm21 pcCT
  where
    pcCT = CT.moore11 ns od
    ns   = ExB.res21 (+)
    od   = ExB.res11 id
```

We cannot print `ict` nor `vct` any more, but we can plot them. Again, the generated plots might need to be tweaked.

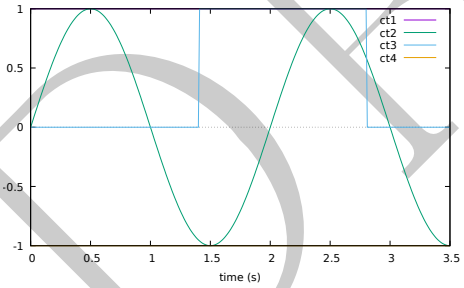
```
λ> let latexIn = latexV 3.5 ["ct1","ct2","ct3","ct4"] vct
λ> let latexS1 = latexV 3.5 ["ct1-1","ct2-1","ct3-1","ct4-1"] $ stage1CT ict vct
λ> let gnuIn = plotV 3.5 ["ct1","ct2","ct3","ct4"] vct
λ> let gnuS1 = plotV 3.5 ["ct1-1","ct2-1","ct3-1","ct4-1"] $ stage1CT ict vct
```



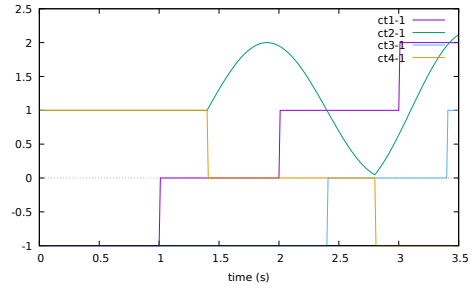
(a) `latexIn`



(b) `latexS1`



(c) `gnuIn`



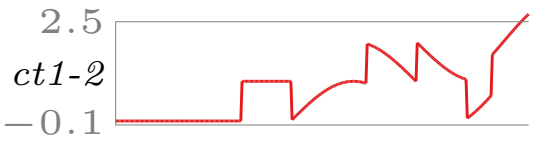
(d) `gnuS1`

The second stage is a `reduce` network of CT `comb` processes.

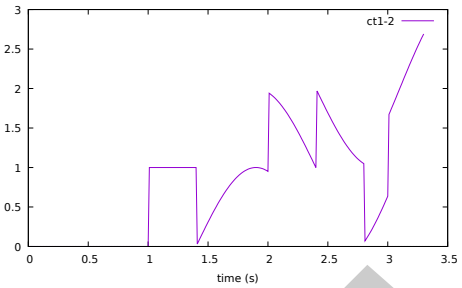
```
stage2CT :: V.Vector (CT.Signal (AbstExt Time))
         → CT.Signal (AbstExt Time)
stage2CT = V.reduce rCT
  where
    rCT = CT.comb21 (ExB.res21 (+))
```

```
λ> let s2out   = (stage2CT . stage1CT ict) vct
λ> let latexS2 = latex 3.3 ["ct1-2","ct2-2","ct3-2","ct4-2"] s2out
λ> let gnuS2   = plot 3.3 ["ct1-2","ct2-2","ct3-2","ct4-2"] s2out
```

For the last stage of the `toy` system there is no CT process constructor in `ForSyDe.Atom.MoC.CT` so we need to create it ourselves.



(a) latexS2

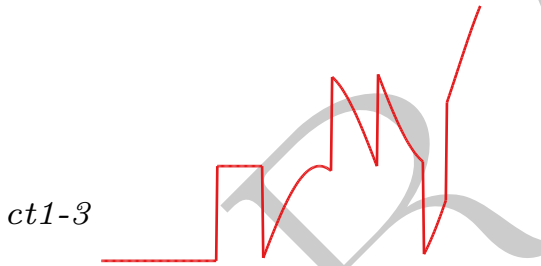


(b) gnuS2

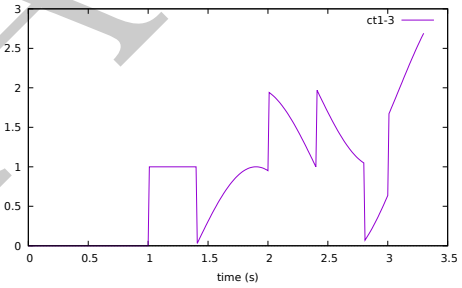
```
stage3CT :: CT.Signal (AbstExt Time)
         → CT.Signal (AbstExt Time)
stage3CT = ctFilter (>=0)
  where
    ctFilter p s = CT.comb21 ExB.filter (predSig p s) s
    predSig  p s = CT.comb11 (ExB.res11 p) s

toyCT :: V.Vector (TimeStamp, Time → AbstExt Time)  -- ^ initial states
      → V.Vector (CT.Signal (AbstExt Time))         -- ^ input
      → CT.Signal (AbstExt Time)                    -- ^ output
toyCT i = stage3CT . stage2CT . stage1CT i
```

```
λ> let latexS3 = latex 3.3 ["ct1-3"] $ toyCT ict vct
λ> let gnuS3   = plot  3.3 ["ct1-3"] $ toyCT ict vct
```



(a) latexS3



(b) gnuS3

2.3.5 SDF instance

The SDF instance of the `toy` system is created using process constructor helpers defined in `ForSdfDe.Atom.MoC.SDF`, and can be found in the following module (re-exported by `AtomExamples.GettingStarted`).

```
module AtomExamples.GettingStarted.SDF where
```

```
import ForSyDe.Atom
import ForSyDe.Atom.ExB.Absent (AbstExt)
import ForSyDe.Atom.ExB              as ExB
import ForSyDe.Atom.MoC.SDF          as SDF
import ForSyDe.Atom.Skeleton.Vector as V
```

`stage1` is defined as a `farm` network of SDF `moore` processes. As SDF Moore processes are in principle graph loops, we take the initial tokens for each loop from a vector of lists of tokens. Also, both next state and output decoder functions are defined over lists of values instead of values, and they need to be provided within a context which describes the *production* and *consumption* rates. Read more about the particularities of SDF in the API documentation.

```
stage1SDF :: V.Vector ([AbstExt Int])           -- ^ vector of initial tokens
          → V.Vector (SDF.Signal (AbstExt Int)) -- ^ vector of input signals
```

```
        → V.Vector (SDF.Signal (AbstExt Int)) -- ^ vector of output signals
stage1SDF = V.farm21 pcSDF
  where
    pcSDF = SDF.moore11 ((1,2),1,ns) (1,1,od)
    ns [x1] [y1,y2] = [ExB.res31 (λa b c → a + b + c) x1 y1 y2]
    od [x1]         = [ExB.res11 id x1]
```

Let us print and plot the inputs against the outputs, using the test signals and plotting functions `latexV` and `plotV` defined in section 2.3.1:

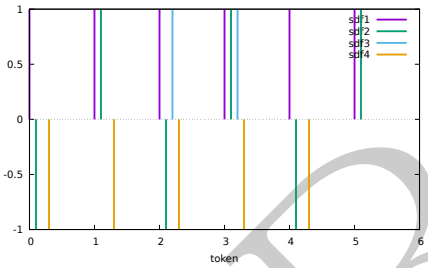
```
λ> isdf
<[-1],[1],[-1],[1]>
λ> vsdf
<{1,1,1,1,1,1},{-1,1,-1,1,-1,1},{0,0,1,1,0},{-1,-1,-1,-1,-1}>
λ> stage1SDF isdf vsdf
<{-1,1,3,5},{1,1,1,1},{-1,-1,1},{1,-1,-3}>
λ> let latexIn = latexV 6 ["sdf1","sdf2","sdf3","sdf4"] vsdf
λ> let latexS1 = latexV 7 ["sdf1-1","sdf2-1","sdf3-1","sdf4-1"] $ stage1SDF isdf vsdf
λ> let gnuIn = plotV 6 ["sdf1","sdf2","sdf3","sdf4"] vsdf
λ> let gnuS1 = plotV 7 ["sdf1-1","sdf2-1","sdf3-1","sdf4-1"] $ stage1SDF isdf vsdf
```

1.0	1.0	1.0	1.0	1.0	1.0
-1.0	1.0	-1.0	1.0	-1.0	1.0
0.0	0.0	1.0	1.0	0.0	
-1.0	-1.0	-1.0	-1.0	-1.0	

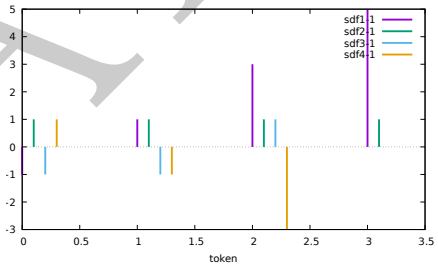
(a) `latexIn`

-1.0	1.0	3.0	5.0
1.0	1.0	1.0	1.0
-1.0	-1.0	1.0	
1.0	-1.0	-3.0	

(b) `latexS1`



(c) `gnuIn`



(d) `gnuS1`

`stage2` is again a `reduce` network of `comb` processes. As with `stage1`, we need to provide the production and consumption rates.

```
stage2SDF :: V.Vector (SDF.Signal (AbstExt Int))
          → SDF.Signal (AbstExt Int)
stage2SDF = V.reduce rSDF
  where
    rSDF = SDF.comb21 ((1,1),1,rF)
    rF [x1] [y1] = [ExB.res21 (+) x1 y1]
```

Again, let us print and plot the output signals using the test inputs and utilities defined in section 2.3.1.

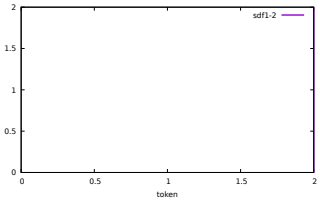
```
λ> let s2out = (stage2SDF . stage1SDF isdf) vsdf
λ> s2out
{0,0,2}
λ> let latexS2 = latex 3 ["sdf1-2"] s2out
λ> let gnuS2 = plot 3 ["sdf1-2"] s2out
```

As for DE and CT instances, a SDF `filter` process does not really make sense in practice, but for the scope of this toy system, we need to instantiate one ourselves.

```
stage3SDF :: SDF.Signal (AbstExt Int)
          → SDF.Signal (AbstExt Int)
stage3SDF = sdfFilter (>=0)
  where
```

0.0 0.0 2.0

(a) latexS2



(b) gnuS2

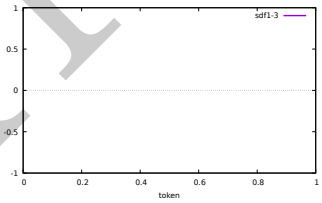
```
    sdfFilter p s   = SDF.comb21 ((2,2),2,filterF) (predSig p s) s
    filterF pl sl   = zipWith ExB.filter pl sl
    predSig   p s   = SDF.comb11 (1,1,fmap (ExB.res11 p)) s
```

```
>
toySDF :: V.Vector ([AbstExt Int])           -- ^ initial tokens
      -> V.Vector (SDF.Signal (AbstExt Int)) -- ^ input
      -> SDF.Signal (AbstExt Int)            -- ^ output
toySDF i = stage3SDF . stage2SDF . stage1SDF i
```

```
λ> toySDF isdf vsdf
{0,0}
λ> let latexS3 = latex 3 ["sdf1-3"] $ toySDF isdf vsdf
λ> let gnuS3   = plot  3 ["sdf1-3"] $ toySDF isdf vsdf
```

0.0 0.0

(a) latexS3



(b) gnuS3

2.3.6 Polymorphic instance

In the previous section you've seen how to model systems in ForSyDe-Atom using the helper functions for instantiating process constructors in different MoCs. In this section we will be instantiating the "raw" polymorphic form of the same process constructors, not overloaded with any execution semantics. The execution semantics are deduced from the tag system of the input signals, i.e. their types. These process constructors are defined as patterns of MoC atoms in the ForSdfDe.Atom.MoC module. The code below is exported by `AtomExamples.GettingStarted`.

```
module AtomExamples.GettingStarted.Polymorphic where
```

Notice that apart from the polymorphic MoC patterns, we are also using "raw" extended behavior and skeleton patterns.

```
import ForSyDe.Atom
import ForSyDe.Atom.MoC.SDF (Prod, Cons)
import ForSyDe.Atom.ExB      as ExB
import ForSyDe.Atom.MoC      as MoC
import ForSyDe.Atom.Skeleton as Skel
```

`stage1` is defined, like in all previous instances, as a farm network of moore processes.

```
stage1 :: (Skeleton s, MoC m, ExB b)
       => Fun m (b a) (Fun m (b a) (Ret m (b a))) -- ^ next state function
       -> Fun m (b a) (Ret m (b a)) -- ^ output decoder function
       -> s (Stream (m (b a))) -- ^ signals with initial tokens
       -> s (Stream (m (b a))) -- ^ vector of input signals
       -> s (Stream (m (b a))) -- ^ vector of output signals
stage1 ns od = Skel.farm21 (MoC.moore11 ns od)
```

We can immediately observe some main differences in the type signature. First, the `Vector`, `Signal` and `AbstExt` data types are not explicit any more, but suggested as type constraints. The first line in the type signature `(Skel s, MoC m, ExB b)` suggests that the type of `s` should belong to the skeleton layer, the type of `m` should belong to the MoC layer and the type of `b` should belong to the extended behavior layer. Another peculiarity is the presence of the first two structures, but there should be nothing frightening about them: e.g. a structure `Fun m a (Fun m b (Ret m c))` simply stands for the type of a function `a -> b -> c`, which was wrapped in a context specific to a MoC `m`. Read the MoC layer's API documentation for more on function contexts. This means that the functions for the next state decoder and the output decoder need to be provided as arguments for `stage1`, and they might differ depending on the MoCs. A third peculiarity is that the initial states are provided as signals and not through some specific structure any more. Indeed, the MoC atoms extract initial states from signals, and deal with them in different ways depending on the MoC they implement.

The main two classes of MoCs, based on their notion of tags, but also based on how they deal with events, are *timed* MoCs (e.g. SY, DE, CT) and *untimed* MoCs (e.g. SDF). Concerning the functions they lift from layers below, we can say that in ForSyDe-Atom timed MoCs lift functions on individual values, whereas untimed MoCs lift functions on lists of values (i.e. multiple tokens). Based on this observation, let us define the next state and output decoders for timed and untimed/SDF MoCs.

```
nsT :: (ExB b, Num a) => b a → b a → b a
odT :: (ExB b, Num a) => b a → b a
nsT  = ExB.res21 (+)
odT  = ExB.res11 id

nsSDF :: (ExB b, Num a) => (Cons, [b a] → (Cons, [b a] → (Prod, [b a])))
odSDF :: (ExB b, Num a) => (Cons, [b a] → (Prod, [b a]))
nsSDF = MoC.ctxt21 (1,2) 1 (λ[x1] [y1,y2] → [ExB.res31 (λa b c → a + b + c) x1 y1 y2])
odSDF = MoC.ctxt11 1     1 (λ[x1]         → [ExB.res11 id x1])
```

OBS: for the sake of simplicity, the ExB component has been left as part of the `nsT` and `odT`, respectively `nsSDF` and `odSDF`, and not part of `stage1`. Describing all layers within the `stage1` function would have rendered the type signature a bit more complicated and is left as an exercise for the reader.

We postpone plotting the input and output signals for later. Carrying on with instatiating `stage2` as a `reduce` network of `comb` processes:

```
stage2 :: (Skeleton s, MoC m, ExB b)
       => Fun m (b a) (Fun m (b a) (Ret m (b a))) -- ^ reduce function
       → s (Stream (m (b a)))                     -- ^ vector of input signals
       → Stream (m (b a))                         -- ^ output signal
stage2 r = Skel.reduce (MoC.comb21 r)
```

Again, the passed functions need to be specifically defined for each MoC, and for simplicity we include the ExB part as well:

```
rT :: (ExB b, Num a) => b a → b a → b a
rT  = ExB.res21 (+)

rSDF :: (ExB b, Num a) => (Cons, [b a] → (Cons, [b a] → (Prod, [b a])))
rSDF = MoC.ctxt21 (1,1) 1 (λ[x1] [y1] → [ExB.res21 (+) x1 y1])
```

Finally `stage3`, the `filter` pattern, we create it ourselves in terms of existing ones. This time we incorporate the extended behaviors in the definition of `stage3`, an we only ask for a context wrapper as input argument. Don't be alarmed by the scary type signature, the actual implementation is quite elegant.

```
stage3 :: (MoC m, ExB b, Ord a, Num a)
       => ((b a → b a)
           → Fun m (b a) (Ret m (b a))) -- ^ context wrapper for the filter behavior
       → Stream (m (b a))               -- ^ input signal
       → Stream (m (b a))               -- ^ output signal
stage3 fctx = MoC.comb11 (fctx filtF)
  where filtF a = ExB.filter (ExB.res11 (>=0) a) a
```

And now for the timed/untimed context wrappers:

```
fctxT     = id
fctxSDF f = MoC.ctxt11 2 2 (fmap f)
```

The full definition of the `toy` system:

```
toy ns od r fctx is = stage3 fctx . stage2 r . stage1 ns od is
```

And that's it! Let us plot now the test signals and the responses of the system for each stage. This time we will use only LaTeX plots. We can also plot initial states as they are wrapped as signals. The test results can be seen in fig. 2.7.

```
λ> let noLabel = ["","","",""]
λ> let iSDF = latexV 2 noLabel sisdf
λ> let iSY  = latexV 2 noLabel sisy
λ> let iDE  = latexV 2 noLabel side
λ> let iCT  = latexV 2 noLabel sict
λ>
λ> let vSDF = latexV 6   noLabel vsdf
λ> let vSY  = latexV 6   noLabel vsy
λ> let vDE  = latexV 3.3 noLabel vde
λ> let vCT  = latexV 3.3 noLabel vct
λ>
λ> let s1SDF = latexV 6   noLabel $ stage1 nsSDF odSDF sisdf vsdf
λ> let s1SY  = latexV 6   noLabel $ stage1 nsT   odT   sisy  vsy
λ> let s1DE  = latexV 3.3 noLabel $ stage1 nsT   odT   side  vde
λ> let s1CT  = latexV 3.3 noLabel $ stage1 nsT   odT   sict  vct
λ>
λ> let s2 ns od r is = stage2 r . stage1 ns od is
λ> let s2SDF = latex 6   noLabel $ s2 nsSDF odSDF rSDF sisdf vsdf
λ> let s2SY  = latex 6   noLabel $ s2 nsT   odT   rT   sisy  vsy
λ> let s2DE  = latex 3.3 noLabel $ s2 nsT   odT   rT   side  vde
λ> let s2CT  = latex 3.3 noLabel $ s2 nsT   odT   rT   sict  vct
λ>
λ> let s3SDF = latex 6   noLabel $ toy nsSDF odSDF rSDF fctxSDF sisdf vsdf
λ> let s3SY  = latex 6   noLabel $ toy nsT   odT   rT   fctxT   sisy  vsy
λ> let s3DE  = latex 3.3 noLabel $ toy nsT   odT   rT   fctxT   side  vde
λ> let s3CT  = latex 3.3 noLabel $ toy nsT   odT   rT   fctxT   sict  vct
```

As expected, the results in fig. 2.7 are exactly the same as the ones presented in section 2.3.2 and ??????. In conclusion we have succesfully instantiated a MoC-agnostic system, whose execution semantics are inferred according to the input data types. This is possible thanks to the notion of type classes, inferred from the host language Haskell. In this section, instead of MoC-specific helpers, we have used the "raw" process constructors as defined in the `ForSdfDe.Atom.MoC` module as patterns of MoC-layer atoms.

This example, used as a case study by Ungureanu and Sander, 2017, has been focused on the MoC layer. A similar approach based on atom polymorphism could target other layers as well since, as you have seen, all layers are implemented as type classes. At the moment of writing this report the extended behavior layer was represented only by the `AbstExt` type, while the skeleton layer had only `Vector`. Nevertheless, future iterations of ForSyDe-Atom will describe more types.

2.4 Making your own patterns

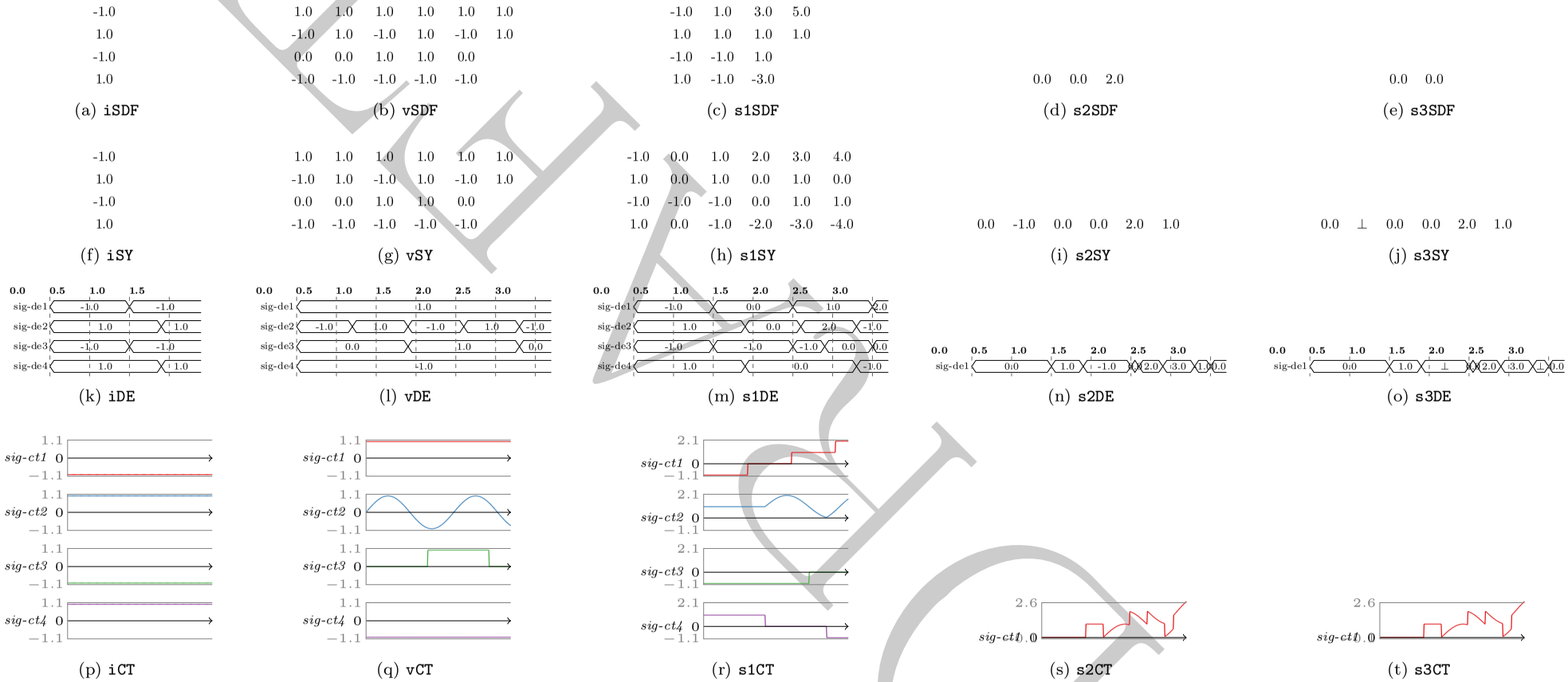
The final section of this report introduces the reader to constructing custom patterns in ForSyDe-Atom. Up until now we have been using patterns which were pre-defined as compositions of atoms. Atoms are primitive, indivizible building blocks capturing the most basic semantics in each layer.

```
{-# LANGUAGE PostfixOperators #-}
```

The code for this section is found in the following module, which is *not* re-exported, i.e. needs to be manually imported.

```
module AtomExamples.GettingStarted.CustomPattern where
```

For this exercise, we will create a custom `comb` pattern with 5 inputs and 3 outputs, as a process constructor in the MoC layer. We will test this pattern with a set of SY and a set of DE input signals, thus we need to import the following modules:



(a) `iSDF` (b) `vSDF` (c) `s1SDF` (d) `s2SDF` (e) `s3SDF`

(f) `iSY` (g) `vSY` (h) `s1SY` (i) `s2SY` (j) `s3SY`

(k) `iDE` (l) `vDE` (m) `s1DE` (n) `s2DE` (o) `s3DE`

(p) `iCT` (q) `vCT` (r) `s1CT` (s) `s2CT` (t) `s3CT`

Figure 2.7: Inputs and outputs for the polymorphic `toy` system in section 2.3.6

```
import ForSyDe.Atom
import ForSyDe.Atom.MoC
import ForSyDe.Atom.MoC.SY as SY
import ForSyDe.Atom.MoC.DE as DE
```

The best way to start building your own patterns is to study the source code for the existing patterns and see how they are made. If you don't want to dig into the source code of ForSyDe-Atom, there is a link in the API documentation for each exported element, as suggested in fig. 2.8.

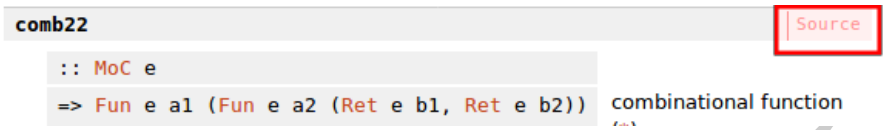


Figure 2.8: Screenshot from the API documentation. The link to the source code is marked with a red rectangle

Studying the `comb22` pattern, you can see that it is defined in terms of the `lift` and `sync` atoms which are represented by the infix operators `-.-` and `-*-` respectively, and the `unzip` utility represented by the postfix operator `-*<`. `lift` and `sync` are atoms becuse they capture an interface for exeucution semantics, whereas `unzip` is just a utility because it is merely a type traversal which alters the structure of data types and rebuilds it to describe "signals of events carrying values".

Considering the applicative nature of the `-.-` and `-*-` atoms, the `comb` pattern with 5 inputs and 3 outputs can be written as the mathematical formula below. This one-liner tells that function `f` is "lifted" into the MoC domain, and applied to the five input signals which are synchronized. The `-*«` postfix operator then "unzips" the resulting signal of triples into three synchronized signals of values. The applicative mechanism explained in the previous paragraph is depicted in fig. 2.9.

```
comb53 f s1 s2 s3 s4 s5
  = (f -.- s1 -*- s2 -*- s3 -*- s4 -*- s5 -*<<)
```

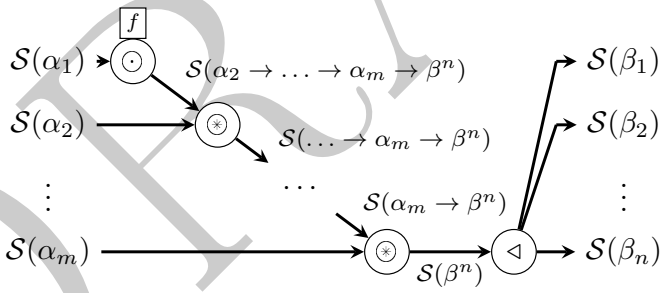


Figure 2.9: Composition of atoms forming the `comb` pattern

If we want to restrict the pattern to one specific MoC, then we must mention this in the type signature we associate it with, like in the example below.

```
comb53SY :: (a1 → a2 → a3 → a4 → a5 → (b1,b2,b3))
         → SY.Signal a1    -- ^ input signal
         → SY.Signal a2    -- ^ input signal
         → SY.Signal a3    -- ^ input signal
         → SY.Signal a4    -- ^ input signal
         → SY.Signal a5    -- ^ input signal
         → ( SY.Signal b1
           , SY.Signal b2
           , SY.Signal b3) -- ^ 3 output signals
comb53SY f s1 s2 s3 s4 s5
  = (f -.- s1 -*- s2 -*- s3 -*- s4 -*- s5 -*<<)
```

To test the output, let us create five signals and a function that needs to be lifted. For the example, the terminal printouts should suffice to test our simple pattern.

```
λ> import AtomExamples.GettingStarted.CustomPattern as CP
λ> let fun a b c d e = (a+c+e, d-b, a*e)
λ> let sy1 = SY.signal [1,2,3,4,5]
λ> let sy2 = SY.comb11 (+10) sy1
λ> let sy3 = SY.constant1 100
λ> let de1 = DE.signal [(0,1),(3,2),(7,3),(9,4),(11,5)]
λ> let de2 = DE.signal [(0,11),(3,12),(5,13),(9,14),(11,15)]
λ> let de4 = DE.constant1 100
λ>
λ> let (o1,o2,o3) = CP.comb53 fun sy1 sy1 sy2 sy2 sy3
λ> o1
λ> {112,114,116,118,120}
λ> o2
λ> {10,10,10,10,10}
λ> o3
λ> {100,200,300,400,500}
λ>
λ> let (o1,o2,o3) = CP.comb53 fun de1 de1 de2 de2 de4
λ> o1
λ> { 112 @0s, 114 @3s, 115 @5s, 116 @7s, 118 @9s, 120 @11s}
λ> o2
λ> { 10 @0s, 10 @3s, 11 @5s, 10 @7s, 10 @9s, 10 @11s}
λ> o3
λ> { 100 @0s, 200 @3s, 200 @5s, 300 @7s, 400 @9s, 500 @11s}
λ>
λ> let (o1,o2,o3) = CP.comb53SY fun sy1 sy1 sy2 sy2 sy3
λ> o1
λ> {112,114,116,118,120}
λ> o2
λ> {10,10,10,10,10}
λ> o3
λ> {100,200,300,400,500}
λ>
λ> let (o1,o2,o3) = CP.comb53SY fun de1 de1 de2 de2 de4
<interactive>:71:56-58:
    Couldn't match type 'DE Integer' with 'SY b3'
    Expected type: SY.Signal b3
      Actual type: DE.Signal Integer
    Relevant bindings include
      it :: (SY.Signal b3, SY.Signal b2, SY.Signal b3)
        (bound at <interactive>:71:1)
    In the second argument of 'c0omb53SY', namely 'de1'
    In the expression: comb53SY fun de1 de1 de2 de2 de4

...
```

2.5 Conclusion

This report has introduced the reader to the basic features of ForSyDe-Atom a framework for modeling and testing of cyber-physical systems. It has covered basic usage such as instantiating systems and plotting signals. It briefly went through concepts such as layers, atoms and patterns, and has focused on their practical usage. A step-by-step tutorial has been presented, showing alternative ways to instantiate systems and demonstrating the polymorphism of layers.

The reader is recommended to further consult the API documentation which also acts as a manual for the library, as well as the related publications listed on the project web site. Future reports will assume familiarity with using and understanding the framework and will focus mainly on results.

2.6 References

Halbwachs, Nicholas et al. (1991). "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9, pp. 1305–1320.

Ungureanu, George and Ingo Sander (2017). "A layered formal framework for modeling of cyber-physical systems". In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1715–1720.

CHAPTER 3

Hybrid CT/DT Models in ForSyDe-Atom

This chapter gathers examples and experiments which involve hybrid models focusing on the the semantics and implications of combining and translating between continuous and discrete domains. As such, the focus is mainly on exploring alternative models and analyzing the implications on fidelity, precision and performance. This chapter is also meant to support the associated scientific publications with experimental results.

Contents

Info

Compatibility : ForSyDe-Atom version 0.2.2
Repository : https://github.com/forsyde/forsyde-atom-examples
Path : <repo_root>/hybrid
Other deps. : none

3.1 Goals

The reader is assumed to have been familiarized with the ForSyDe-Atom modeling framework. A good resource for that is chapter 2. The main goals of this chapter are:

- explore alternative ForSyDe-Atom models for widely known hybrid (discrete and continuous time) systems, with the scope of analyzing the implications from the modeling and simulation perspective.
- train the reader into the decision-making process of modeling hybrid systems, and the trade-offs involved. While as per writing this report the ForSyDe-Atom modeling framework is still limited, future directions are hinted.
- support the associated scientific publications with the complete experiments and results for the models used as case studies. These publications include: **ungureanu18a**

3.2 RC Oscillator

In this section we model the RC oscillator setup in fig. 3.1. Despite its apparent simplicity, shows an interesting problem in CPS: how continuous systems react to discrete stimuli. This example was used in Ungureanu, Medeiros, and Sander, 2018, and for a study on the theoretical implications

of the chosen models, we strongly recommend reading this paper before going further. The source code for this section is found in the following module:

```
module AtomExamples.Hybrid.RCOsc where
```

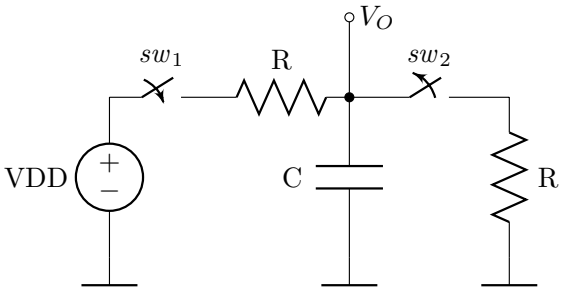


Figure 3.1: RC oscillator setup

These are the dependencies that need to be included within this module:

```
import ForSyDe.Atom                          -- general utilities
import ForSyDe.Atom.MoC.CT   as CT           -- CT MoC library
import ForSyDe.Atom.MoC.DE   as DE           -- DE MoC library
import ForSyDe.Atom.MoC.SY   as SY           -- SY MoC library
import ForSyDe.Atom.MoC.Time as T            -- utilities for functions of time
import ForSyDe.Atom.MoC.TimeStamp (milisec) -- utilities for time stamps
import ForSyDe.Atom.Utility.Plot             -- plotting utilities
```

The circuit in fig. 3.1 is characterized by two states: 1) sw_1 is closed and sw_2 open, equivalent to the situation where the capacitor C charges with VDD; 2) sw_1 is open and sw_2 is closed, equivalent to the situation where the capacitor C discharges to the ground through R. For the scope of this example, to keep the model simple and deterministic, we assume that sw_1 and sw_2 open/close alternatively at the same discrete time instants, there are no intermediary transitions, and we ignore the effects of discharge through switches.

Let us analyze case 1) above. The circuit in fig. 3.1 becomes a RC integrator, where the input signal is applied to the resistance with the output taken across the capacitor. The amount of charge that is established across the plates of the capacitor is equal to the time domain integral of the current, like in eq. (3.1). The rate at which the capacitor charges is directly proportional to the amount of the resistance and capacitance giving the time constant of the circuit like in eq. (3.2). As the capacitors current can be expressed as the rate of change of charge, Q with respect to time, we can express the charge at any instant of time like in eq. (3.3). Eq. (3.3) brings together the previous equations and uses the instant voltage charge formula to obtain the final integral formula for V_O.

$$i_C(t) = C\frac{\partial V_{C(t)}}{\partial t} = \frac{V_{DD}}{R} \tag{3.1}$$

$$RC = R\frac{Q}{V} = R\frac{i \times T}{i \times R} = T \tag{3.2}$$

$$i = \frac{\partial Q}{\partial t} \Rightarrow Q = \int i \; \partial t \tag{3.3}$$

$$V_O = V_C = \frac{Q}{C} \overset{(3)}{=} \frac{1}{C}\int i \; \partial t \overset{(1)}{=} \frac{1}{C}\int \frac{V_{DD}}{R}\partial t = \frac{1}{RC}\int V_{DD}\partial t \tag{3.4}$$

If an ideal step voltage pulse is applied, that is with the leading edge and trailing edge considered as being instantaneous, the voltage across the capacitor will increase for charging exponentially over time at a rate determined by eq. (3.5). Similarly, in case 2) the circuit becomes an RC bridge where, assuming that C has been fully charged, it now discharges to the ground at a rate

determined by eq. (3.6).

$$V_{O,\text{charge}}(t) = V_C(t) = V_{DD}\left(1 - e^{-\frac{t}{RC}}\right) \tag{3.5}$$

$$V_{O,\text{discharge}}(t) = V_C(t) = V_{DD}\left(e^{-\frac{t}{RC}}\right) \tag{3.6}$$

Translating eqs. (3.5) and (3.6) into ForSyDe-Atom Haskell code, we get the following, assuming `t0` is the moment where the switch occurred, and ignoring the factor of V_{DD} (which is considered and scaled in the output decoder anyway).

```
-- | RC time constant.
rc = 0.1

-- | V_O(t0,t) during the discharging. t0 is the instant when the switch occurred.
vcDischarge t0 = λt → T.exp (−(t−t0) / rc)

-- | V_O(t0,t) during the charging. t0 is the instant when the switch occurred.
vcCharge    t0 = λt → 1 − T.exp (−(t−t0) / rc)
```

Naturally, in ForSyDe-Atom we can model the oscillator in multiple ways, depending on what aspect we want to focus. Our first example models the RC oscillator as a Mealy finite state machine which embeds the continuous and discrete time semantics as defined in the `ForSyCt.Atom.MoC.CT` library, like in fig. 3.2. As such, we imply from the model that the switching of sw_1 and sw_2 is performed periodically after e certain τ, inferred from the initial state of the `CT.mealy` process. As such, the state machine is loaded with the voltage charging rule in eq. (3.5), and a duration τ. The configuration of the `mealy` pattern ensures that at each multiple of τ the next state function `ns` will be performed, negating the state rule (i.e. the rate for V_O), basically translatig back and forth between eqs. (3.5) and (3.6). As mentioned before, the output decoder `od` merely rescales V_O with V_{DD}.

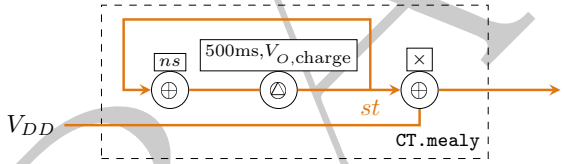


Figure 3.2: Internal pattern of the initial RC oscillator setup

```
-- | RC oscillator model as CT FSM with discrete semantics inherent in the CT model.
osc1 :: CT.Signal Rational -- ^ VDD as input signal
     → CT.Signal Rational -- ^ V_O as output signal
osc1 = CT.mealy11 ns od (milisec 500, vcCharge 0)
  where
    ns v _ = 1 + (−1 * v)
    od     = (*)
```

Plotting the output against an "arbitrary" V_{DD}, we get fig. 3.3.

```
-- | plotting configuration that will be used throughout this section
cfg = defaultCfg {xmax=3, rate=0.01}
```

```
-- | example input for testing 'osc1'
vdd1 = CT.signal [(0,λ_→2),(1,λ_→1.5),(2,λ_→1)] :: CT.Signal Rational

-- | plotting the response of 'osc1'
plot1 = plotGnu $ prepareL cfg {labels=["V_{DD}","V_O"]} $ [vdd1, osc1 vdd1]
```

The model for `osc1`, although strikes out as simple and elegant, is by all means limited. In the following paragraphs we will address these limitations one at a time, and try find solutions that overcome them in order to include more realistic or at least a family of behaviors correctly covering classes of (increasing) non-determinism.

Lee, 2016 argues that determinism is an attribute of the model, and it is dependent on what we consider as inputs and outputs. As such, the model `osc1` in fig. 3.2 is a deterministic model for the circuit in fig. 3.1 and the output in fig. 3.3 is the correct response in the following conditions:

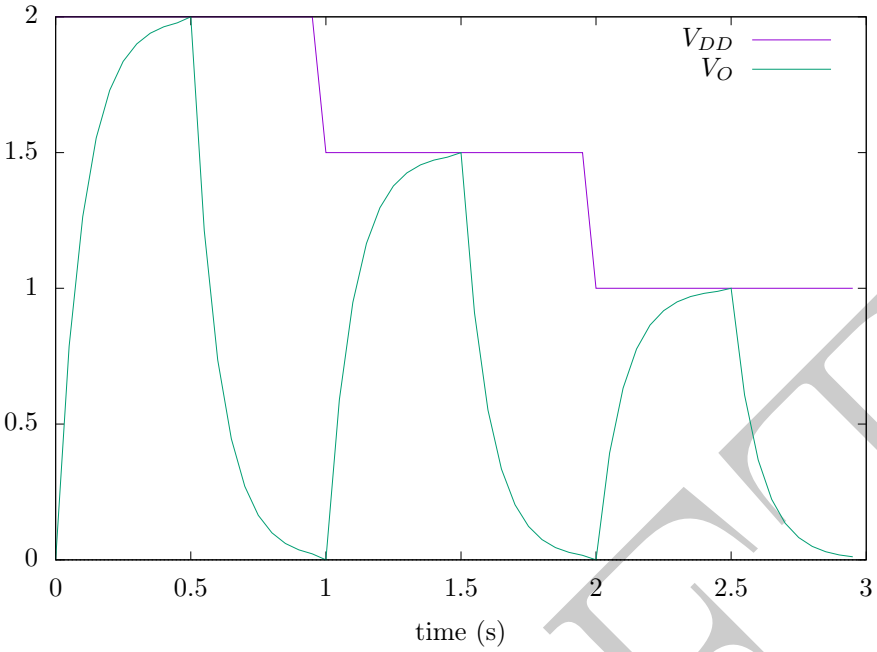


Figure 3.3: Response of `osc1`

- the changes in V_{DD} happen at multiples of τ. If a changes occur at any other time the response will be shown, but it will not describe the RC circuit in a realistic manner, as we will see shortly.
- the time constant follows roughly the rule $5RC \leq \tau$ where the chosen τ is a part of the initial state of the Mealy state machine. This says that the discharging occurrs when the capacitor is charged at least 99.3% and vice-versa, according to eqs. (3.5) and (3.6).
- $\tau > 0$. $\tau = 0$ would render the system as non-causal and would manifest Zeno behavior. In practical terms, this is the equivalent of the simulation being stuck and not advancing time.

Another property (or limitation, depending on what your intentions are) is that the switching of sw_1 and sw_2 cannot be controlled and is a property of the Mealy machine. This switching is inferred by τ which sets the (discrete) period of these events. Let us change that by giving the possibility to control the state of the capacitor as charging/discharging through a signal of discrete events. A system such as the one we propose implies some subtle changes in the modelling which require a deeper understanding on the underlying MoCs. First of all, we still require a stateful process (a state machine) which captures the notion of working modes, but which reacts to a state change instantaneously. But this implies that $\tau = 0$ which, as said above, would lead to Zeno behavior. On the other hand, the particular behavior required is described precisely by *synchronous reactive* MoC, where the response of a system is performed in "zero time".

Continuing to adhere to the school of thought of Lee, 2016, where the merit of the modeler lies in choosing the right modeling paradigm for the right problem[1], we choose to model the above described system as a synchronous reactive (SY) state machine wrapped inside a DE/CT environment like in fig. 3.4. As such, we should note a few particularities of this model:

- in CT the carried values are implicit functions of time, i.e. they carry time semantics. In DE and SY, the time semantics make no sense, thus the same functions of time are explicit (where `Time` itself is just a data type $\in V$). Therefore when converting $\mathcal{S}_{CT}(\alpha) \mapsto \mathcal{S}_{DE}(t \to \alpha)$ and vice-versa $\mathcal{S}_{DE}(t \to \alpha) \mapsto \mathcal{S}_{CT}(\alpha)$.

[1]Lee argues that the cost we pay in the loss of determinism is a property of the chosen modeling framework, and *not* of the model itself.

- in SY tags are useless, therefore DE tags and values are split into two separate SY signals upon conversion. This way we can easily recreate the DE signal without loss of information. So upon conversion $\mathcal{S}_{DE}(\alpha) \mapsto \mathcal{S}_{SY}(t) \times \mathcal{S}_{SY}(\alpha)$ and vice-versa $\mathcal{S}_{SY}(t) \times \mathcal{S}_{SY}(\alpha) \mapsto \mathcal{S}_{DE}(\alpha)$.
- the discrete control signal S_{control}, carries nothing, and it is used only for the timestamps generated by its tags, i.e. for generating $\mathcal{S}_{SY}(t)$. These timestamps are further used by the `od` function to determine the time when the switch occurred `t0` in the formulas for V_O in eqs. (3.5) and (3.6).

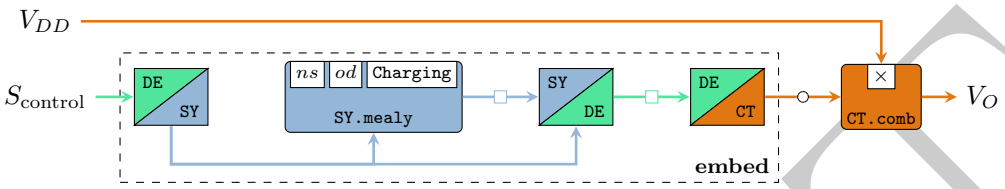


Figure 3.4: RC oscillator model which reacts to a discrete control signal. Shapes decorating signals suggest the type of carried tokens: circles = scalars; squares = functions

Like in the model from fig. 3.2, the model for `osc2` in fig. 3.4 scales the output with V_{DD} through a combinational CT process. The next state decoder function *ns* does not manipulate the rule for V_C anymore, but rather switches between the states `Charge` and `Discharge`, and the output decoder *od* selects the proper V_C rule from eq. (3.5) or eq. (3.6) respectively, based on the current state.

```
-- | Encodes the capacitor state
data CState = Charging | Discharging
```

We encode the capcitor state (i.e. the states of sw_1 and sw_2) through the `CState` data type. Thus the code for fig. 3.4 is:

```
-- | RC oscillator model as FSM which reacts to discrete impulses
osc2 :: CT.Signal Rational  -- ^ VDD input signal
     → DE.Signal ()         -- ^ control signal of discrete impulses
     → CT.Signal Rational   -- ^ output signal
osc2 s = CT.comb21 (*) s . embed (SY.mealy11 ns od Charging)
  where
    -- SY next state function
    ns Charging    _  = Discharging
    ns Discharging _  = Charging
    -- SY output decoder function
    od Charging    t0 = vcCharge (time t0)
    od Discharging t0 = vcDischarge (time t0)
    -- wrapper that embeds a SY process into a mixed DE/CT environment
    embed p de        = let (t, _) = DE.toSY de
                        in DE.toCT $ SY.toDE t (p t)
```

Now let us create two situations to test `osc2`. The control signal `sCtrl` injects events at timestamps 0, 0.6, 1.4, 2.3 and 2.8, causing the SY state machine to react each time changing its state from `Charge` to `Discharge` and vice-versa. The first V_{DD} input `vdd21` is mirroring an "ideal case", when changes occur at time instants where switching happens, meaning that it does not affect the evolution of the V_O rule. On the contrary, `vdd22` comes at arbitrary times, thus affecting the output in a way that is unrealistic, i.e. V_O changes values abruptly and instantaneously, which strays away from the acceptable behavior of a capacitor. This can be seen in fig. 3.5.

```
-- | Signal of discrete events. Carries nothing, it is used only for the time stamps.
sCtrl = DE.signal [(0,()),(0.6,()),(1.4,()),(2.3,()),(2.8,())] :: DE.Signal ()

-- | example input for testing 'osc2'
vdd21 = CT.signal [(0,λ_→2),(1.4,λ_→1.5),(2.8,λ_→1)] :: CT.Signal Rational
vdd22 = CT.signal [(0,λ_→1),(0.8,λ_→2),(1.6,λ_→1.5)] :: CT.Signal Rational

-- | plotting the example responses of 'osc2'
plot21 = plotGnu $ prepareL cfg {labels=["V_{DD}","V_O"]} $ [vdd21, osc2 vdd21 sCtrl]
plot22 = plotGnu $ prepareL cfg {labels=["V_{DD}","V_O"]} $ [vdd22, osc2 vdd22 sCtrl]
```

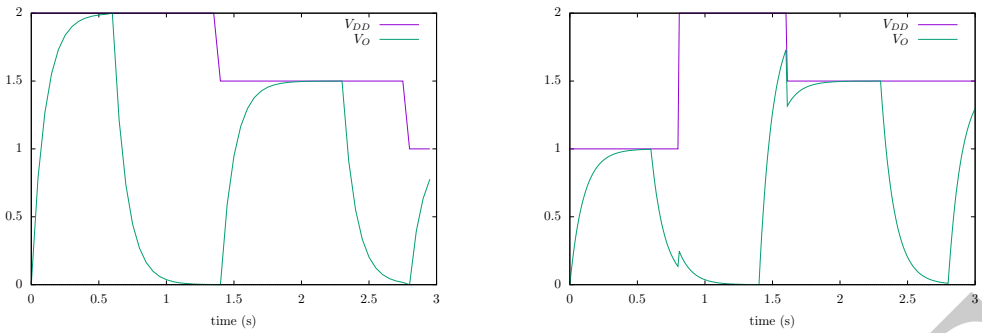


Figure 3.5: Response of `osc2`

The second case in fig. 3.5 can be explained easily if we consider the fact that V_O itself is the result of a statically pre-calculated function of time. There is no notion of feedback response to the continuous inputs, the only feedback is synchronous reactive for `osc2` and even `osc1`. In other words any change on the input will affect how the output "looks like", but it will not affect the continuous behavior which, as said, is pre-calculated. In order to influence the continuous behavior, some sort of continuous feedback is necessary, which is usually non-causal, thus uncomputable. Without going too much into detail we can say that in order to model a realistic behavior of the capacitor response in V_O, we need to embed an ordinary differential equation (ODE) solver within a synchronous reactive state machine[2], which models precisely eq. (3.4).

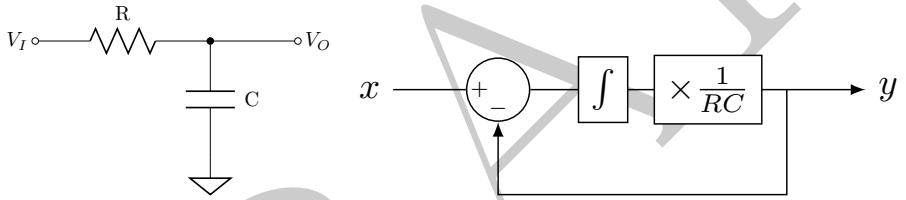


Figure 3.6: RC bridge: circuit (left); block diagram based on eq. (3.4) (right)

$$\int_{t_0}^{t_0+h} f(t, y(t))\partial t \approx h f(t_0, y(t_0)) \tag{3.7}$$

$$\dot{y}_n = \frac{hRC}{h+RC}\left(\frac{1}{RC}y_n + \frac{1}{h}y_{n-1}\right) \tag{3.8}$$

First let us model a simple RC bridge like in fig. 3.6 which acts like a low-pass filter, and acts according to eq. (3.4). As mentioned, we model this circuit as a SY state machine embedded within a CT environment. Why a state machine? Because the circuit itself, due to the capacitor, is a memory system, i.e. its output is dependent on its history as well as inputs. The previous "history" at the start of simulation is encoded inside the initial state. The next state decoder itself needs to "feed-through" the input and mirror the behavior of the block diagram fig. 3.6. On the other hand, this block diagram shows a non-causal system, which as such is uncomputable, and needs to be transformed into a solvable form. As per the writing of this report, ForSyDe-Atom[3] did not provide generic ODE solvers, thus for didactic purpose we write the following numerical solver for our RC circuit in fig. 3.6 using Euler's trapezoidal method eq. (3.7), by substituting by hand eq. (3.4) with its feed-forward version in eq. (3.8):

```
-- | ODE solver for a simple RC low-pass filter using Euler's method
euler :: TimeStamp          -- ^ time step for the solver precision
      -> (Time -> Rational) -- ^ the input function being integrated
```

[2] theoretical implications will be analyzed in the upcoming journal publications
[3] version 0.2.2

```
      → Rational           -- ^ the "history" of the integral at t0
      → TimeStamp          -- ^ t0
      → Time → Rational    -- ^ a function of time
euler step f p t0 t = iterate p t0
  where
    h = time step
    -- by-hand substitution of eq.(4) using the trapezoidal rule
    calc vp v = (h * rc)/(h + rc) * (1/rc * v + 1/h * vp)
    -- loop which calculates the integral step-wise from t0 to t
    iterate st ti
      | t < time ti = st
      | otherwise   = iterate (calc st $ f (time ti)) (ti + step)
```

As said earlier, we model the RC circuit in fig. 3.6 by including the `euler` solver into a feed-through state machine (see the `state` pattern) like in fig. 3.7. Like for `osc2`, the timestamps resulted after splitting the tags from values when interfacing between DE and SY are used for determining the t_0s for each new incoming event. The solver inputs functions of time and outputs functions of time (i.e. the solver itself is the output).

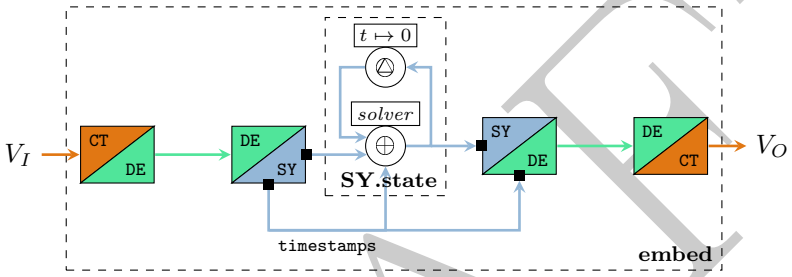


Figure 3.7: ForSyDe-Atom model of the RC bridge in fig. 3.6

```
rcfilter :: CT.Signal Rational → CT.Signal Rational
rcfilter s
  = let (ts, sy) = DE.toSY $ CT.toDE s
        out      = SY.state21 ns (λ_→0) ts sy
        ns p t s = euler 0.01 s (p (time t)) t
    in DE.toCT $ SY.toDE ts out
```

Testing the filter against a square wave signal, we get the response plotted in fig. 3.8. As can be clearly seen, the behavior is the right one and it reacts correctly to the inputs, taking unto account the current state of the system, i.e. its history, and it is a *continuous* signal in the true sense of the word.

```
-- | square wave signal for testing 'rcfilter'
vi3 = CT.signal [(0,λ_→2), (0.3,λ_→0), (1,λ_→1.5), (1.5,λ_→0), (1.7,λ_→1), (2.5,λ_→0)] :: CT.
    Signal Rational

-- | plotting the example response of 'rcfilter'
plot31 = plotGnu $ prepareL cfg {labels=["V_I","V_O"]} $ [vi3, rcfilter vi3]
```

Now let us modify `osc2` from fig. 3.4 to mirror the correct behavior of the circuit in fig. 3.1. Seems to it that there is not much to do, as `rcfilter` already responds correctly to any input. On the other hand the RC oscillator as represented in fig. 3.1 admits as inputs both V_{DD} and the control signal for sw_1 and sw_2. To correctly model that in ForSyDe-Atom we need to separate the FSM which controls the state of the capacitor as charging or discharging, which in turn will generate V_I. We do that like in fig. 3.9.

```
-- | ODE-based RC oscillator model
osc4 :: CT.Signal Rational  -- ^ VDD input signal
     → DE.Signal ()         -- ^ control signal of discrete impulses
     → CT.Signal Rational   -- ^ output signal
osc4 vdd ctl
  = let -- generator for the switch state variable
        swState = DE.embedSY11 (SY.stated11 swF Charging) ctl
```

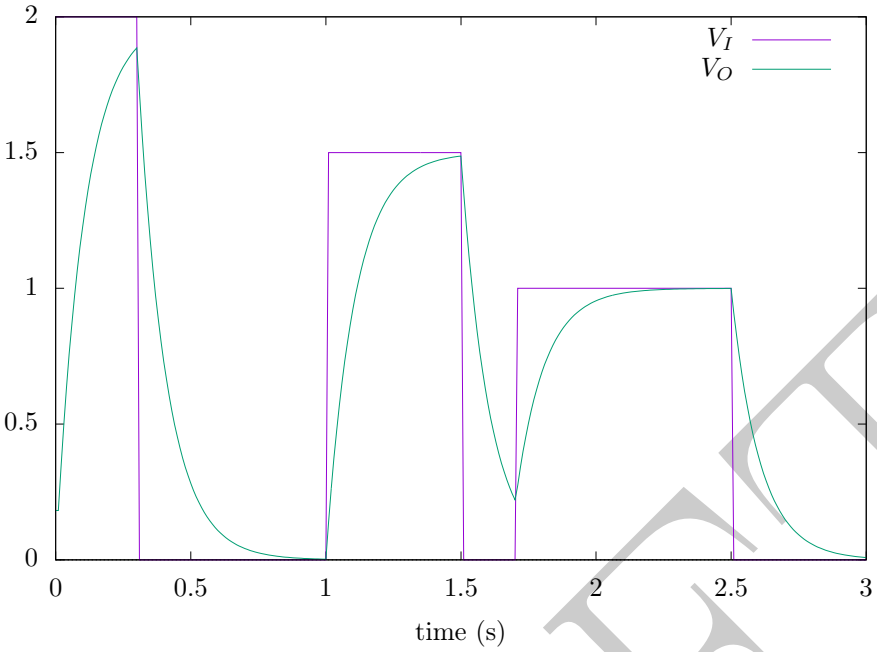


Figure 3.8: Response of `rcfilter`

```
        swF Charging    _  = Discharging
        swF Discharging _  = Charging
        -- transforms VDD into VI for the RC filter
        vddSwitched        = DE.comb21 vddF swState $ CT.toDE vdd
        vddF Charging    v = v
        vddF Discharging _ = λ_→0
        -- state machine with ODE solver modeling an RC filter
        vOut        = embed (SY.state21 nsEU (λ_→0)) vddSwitched
        nsEU p t s  = euler 0.01 s (p $ time t) t
        -- custom wrapper that embeds a SY tag-aware process into a DE environment
        embed p de  = let (t, v) = DE.toSY de
                      in SY.toDE t (p t v)
    in DE.toCT vOut
```

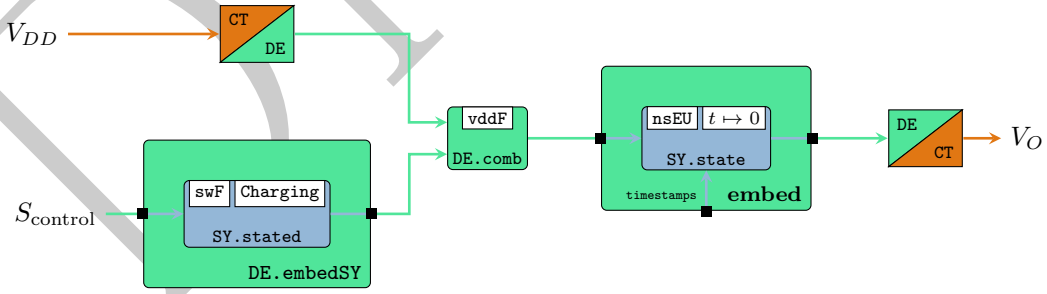


Figure 3.9: FORSYDE-ATOM model of the RC oscillator described by `osc4`

Testing `osc4` against the same inputs as `osc2` in fig. 3.5, we get the response plotted in fig. 3.10, which is now the correct behavior of the RC circuit with respect to its inputs. As expected, the output describes correctly the dynamics of the system based on the state and history of the capacitor.

```
plot41 = plotGnu $ prepareL cfg {labels=["V_{DD}","V_O"]} $ [vdd22, osc4 vdd22 sCtrl]
```

We have shown three different models of an RC oscillator circuit represented in fig. 3.1 at different levels of complexity, and a model of an RC bridge represented in fig. 3.6. The fact that

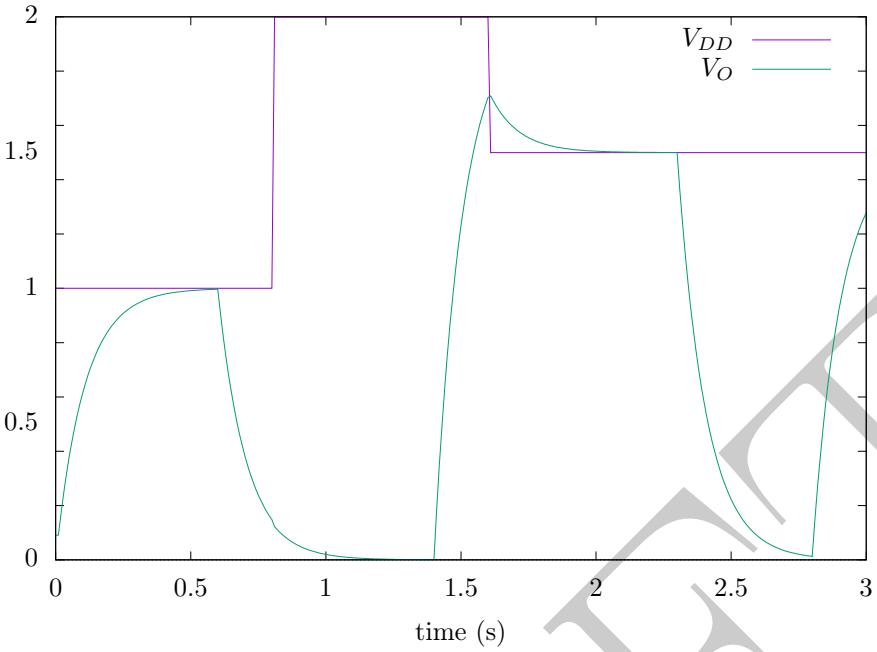


Figure 3.10: Response of `osc4`

they are different does not mean that either is "more correct/incorrect" than the other. Either of them might very well be treated as "correct" depending on what we consider or not the acceptable inputs (note that we have not defined them on purpose). As expected, the more we consider the inputs as sources of non-determinism, the more complex our model needs to be in order to cover "special" cases. As you might guess already, this comes at a severe cost of run-time performance.

Let us briefly measure this cost in performance between the four presented models. For this, we consider two situations:

Experiment 1 we need to find out V_O at $t = 2.8$ seconds.

Experiment 2 we need to sample V_O for the whole period $t = [0, 3]$ seconds, with a precision of 50 milliseconds.

We consider the same input scenarios for all 4 models:

```
-- | plotting/sampling configuration for performance testing
cfgTest = defaultCfg {xmax=3, rate=0.005}
-- | VDD input for performance testing
vddTest = CT.signal [(0,λ_→2),(1,λ_→1.5),(2,λ_→1)] :: CT.Signal Rational
-- | VI input for performance testing of the RC filter model
viTest = CT.signal [(0,λ_→2),(0.5,λ_→0),(1,λ_→1.5),(1.5,λ_→0),(2,λ_→1),(2.5,λ_→0)] :: CT.
    Signal Rational
-- | Switch control signal for performance testing
ctlTest = DE.signal [(0,()),(0.5,()),(1,()),(1.5,()),(2,()),(2.5,())] :: DE.Signal ()
```

DISCLAIMER: we have measured the performance for all experiments using the `:set +s` directive in a `ghci` interpreter session, which offers some crude information about the run-time. The measurements are by no means realistic for compiled and optimized programs, but rather give a rough idea about the raw, un-optimized functions run directly in the interpreter. In any case, this offers a common ground for comparing implementations between them to observe trends and the influences of the design decisions, but not to judge ForSyDe-Atom as such.

```
λ> :set +s
λ> fromRational $ osc1 vddTest         `CT.at` 2.8
4.9787192469339395e-2
(0.01 secs, 398,792 bytes)
λ> fromRational $ osc2 vddTest ctlTest `CT.at` 2.8
```

```
4.9787192469339395e-2
(0.01 secs, 385,952 bytes)
λ> fromRational $ rcfilter viTest      'CT.at' 2.8
5.169987618408467e-2
(0.08 secs, 34,044,128 bytes)
λ> fromRational $ osc4 vddTest ctlTest 'CT.at' 2.8
5.169987618408467e-2
(0.07 secs, 34,047,720 bytes)
```

Model	Experiment 1		Experiment 2	
	time (s)	memory (MB)	time (s)	memory (MB)
`osc1`	0.01	0.4	0.09	23
`osc2`	0.01	0.4	0.09	18
`rcfilter`	0.08	34	5.04	1930
`osc4`	0.07	34	5.56	1930

Table 3.1: Experimental results for testing the RC models

The experimental results on an computer with Intel® Core™ i7-3770 CPU @ 3.40GHz × 8 threads, and 31,4 GiB RAM are shown in table 3.1. As expected, `osc1` and `osc2` perform in almost negligible time for both experiments due to lazy evaluation which performs computation only at the requested evaluation point. However, the lazy evaluation is costly in terms of run-time memory, fact noticed especially for `rcfiler` and `osc4` due to the fact that intermediate structures need to be stored before evaluating. The complexity of `rcfilter` and `osc4` are seen both in the execution time and in the memory consumption. Apart from the cost in performance, model fidelity for unknown inputs came also with a high price in loss of precision due to chained numerical computation, fact seen from the evaluation results of **Experiment 1** in the interpreter listing above. Choosing a better solver than `euler`, e.g. a Runge-Kutta solver, or even better, a symbolic solver, could improve both performance and precision, but that is out of the scope of this report. Another, rather surprising fact is that `osc2` performs slightly better than `osc1`, probably to the lack of tag calculus in the SY domain, i.e. tags are mainly passed untouched between interfaces, whereas calculations are performed in a synchronous reactive manner on values only.

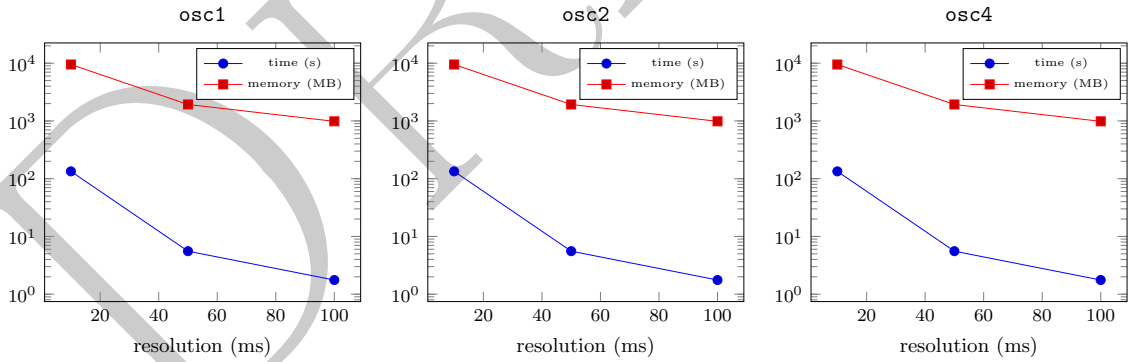


Figure 3.11: Execution time and memory consumption for **Experiment 2**

To get a feeling of the cost complexity of the models for `osc1`, `osc2` and `osc4`, we plot the execution time and memory consumption during **Experiment 2** for three resolutions: 10 milliseconds, 50 milliseconds and 100 milliseconds. This way we can observe the trends in fig. 3.11, where the most notable one is the exponential growth in execution time for `osc4`. This can be explained recalling the so-called "time leaks" observed by **hudak03** (among others), which occur due to the fact that lazy evaluation forces to recalculate all previous states of the ODE solver in order to evaluate the current sample. In functional reactive programming (FRP), a solution to time leaks is the concept of "continuation", which is roughly what happens when the synchronous reactive FSM receives a new discrete event: it stores the current state to the ODE to be used in

the future. Knowing this, let us see what happens when we double the number of discrete events within the input V_{DD} signal.

```
-- | VDD input for performance testing, having twice more discrete events than 'vddTest'
vddTest1 = CT.signal [(0,λ_→2),(0.5,λ_→2),(1,λ_→1.5),(1.5,λ_→1.5),(2,λ_→1),(2.5,λ_→1)] :: CT.
    Signal Rational
```

3.3 Conclusion

This project is still under development...

3.4 References

Lee, Edward A. (2016). "Fundamental Limits of Cyber-Physical Systems Modeling". In: *ACM Trans. Cyber-Phys. Syst.* 1.1, 3:1–3:26. ISSN: 2378-962X. DOI: 10.1145/2912149. URL: http://doi.acm.org/10.1145/2912149.

Ungureanu, George, José E. G. de Medeiros, and Ingo Sander (2018). "Bridging discrete and continuous time with Atoms". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

Part II

Tools & Libraries Documentation

CHAPTER 4

The ForSyDe-Atom Standard Library

This chapter is an extended API documentation of ForSyDe-Atom Standard Library version 0.1.1[1] which was organized to also serve the purpose of technical report. It treats both theoretical and practical aspects, justifying the implementation and providing examples of usage for most of the library-exported functions. It has been generated from the inline documentation using a Haddock[2] generator.

Contents

[1] available at: https://github.com/forsyde/forsyde-atom
[2] custom build from: https://github.com/ugeorge/haddock

4.1 Introduction

The ForSyDe (Formal System Design) methodology has been developed with the objective to move system design to a higher level of abstraction and to bridge the abstraction gap by transformational design refinement. It targets the modelling and characterization of cyber-physical systems inheriting the theory of Models of Computation (MoCs), providing both a correct-by-construction execution model and analyzable entry point for further synthesis flows. For more information about ForSyDe and its associated projects please consult the ForSyDe webpage.

The `forsyde-atom` library is a shallow-embedded DSL implementing the execution semantics of an *atom-based approach* to ForSyDe. Its purpose is to provide a modeling framework for cyber-physical systems and to serve as a proof-of-concept for a future (non-strict typed) DSL. Adhering to the formalism set by the PhD Thesis of Ingo Sander, the current framework views systems as networks of processes communicating through signals. The *atom-based approach* to ForSyDe adds some new important concepts:

- the separation of concerns through semantically-independent ***layers*** of computation, behavior, synchronization and structure. Each layer offers a different analyzable view of a given system and is described using the concept of *higher-order functions*.
- the description of each layer as a network of primitive building blocks called ***atoms***. Each atom embeds an undividable operation, and wraps functions of lower layers with the semantics dictated by a higher layer. They are described using the powerful concept of *applicative functors*.
- the complete ***autonomy*** of atoms in relation to the patterns they build. As such, process networks or constructors are nothing but *structural* ad-hoc compositions that aid in achieving complex behaviors whereas the actual behavior is dictated by the atoms alone. This is proved (for the time being) for the *Synchronization Layer* by implementing all MoCs as instances of *only one type class*.

While the host language limits the possibility of providing general (e.g. proces) constructors due to type-strictness, the `forsyde-atom` framework is by all means complete. In this sense the user can create her own custom *correct-by-design* constructors and networks as compositions of the provided atoms and utilities. Also, this haddock-generated page is organized both as an API documentation and as a technical report to facilitate te use and understanding of the formal principles behing the design process.

4.1.1 Naming convention

All multi-parameter patterns and utilities provided by the library API as higher-order functions are named along the lines of `functionMN` where `M` represents the number of ***curried*** inputs (i.e. `a1 -> a2 -> ... -> aM`), while `N` represents the number of ***tupled*** outputs (i.e. `(b1,b2,...,bN)`). To avoid repetition we only provide documentation for functions with 2 inputs and 2 outputs (i.e. `function22`), while the available ones are mentioned as a regex (i.e. `function[1-4][1-4]`). In case the provided functions do not suffice, feel free to implement your own patterns following the example in section 2.4.

4.2 ForSyDe.Atom

```
module ForSyDe.Atom (
    ExB(extend, (/.\), (/*\), (/&\), (/!\)),  Stream(NullS, (:-)),
    module ForSyDe.Atom.MoC.Stream,  Time,  TimeStamp,
    MoC(Fun, Ret, (-.-), (-*-), (-*), (-<-), (-&-)),
    Skeleton((=.=), (=*=), (=\=), (=<<=), first, last),
    module ForSyDe.Atom.Utility,  (||<)
  ) where
```

The formal foundation upon which ForSyDe I. Sander and Jantsch, 2006 defines its semantics is the *tagged signal model* Lee and Sangiovanni-Vincentelli, 1998. This is a denotational framework introduced by Lee and Sangiovanni-Vincentelli as a common meta model for describing properties of concurrent systems in general terms as sets of possible behaviors. Systems are regarded as *compositions* of *processes* acting on *signals* which are sets of *tagged events*. Signals are characterized by a *tag system* which determines causality between events, and could model time, precedence relationships, synchronization points, and other key properties. Based on how tag systems are defined, one can identify several *Models of Computations (MoCs)* as classes of behaviors dictating the semantics of execution and concurrency in a network of processes.

These concepts are the supporting pillars of ForSyDe's philosophy, and state the purpose of the `forsyde-atom` library: it is supposed to be a modelling framework used as a proof-of-concept for the atom-based approach to cyber-physical systems Ungureanu and Ingo Sander, 2017. This approach extends the tagged signal model by systematically deconstructing processes to their basic semantics and recreating them using a minimal language of primitive building blocks called *atoms*. It also tries to expand the scope of this model by exploiting more aspects than just timing, by adding primitives for parallelism, behavior, etc.

The API documentation is structured as follows: this page provides an overview of the general notions and concepts, gently introducing the separate modules and the motivation behind them. Each major module corresponds to a separate *layer* Ungureanu and Ingo Sander, 2017 implemented as a type class. The documentation pages for each layer and for each of their instances

contains in-depth knowledge and examples, and can be accessed from the contents page or by following the links suggested. For more complex examples and tutorials follow the links in the project web page.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

4.2.1 The layered process model

The `forsyde-atom` project is led by three main policies:

1. in order to cope with the complexity of cyber-physical systems (CPS) it tries to separate the concerns such as computation, timing, synchronization, parallelism, structure, behavior, etc.

2. in order to have a small, ideally minimal grammar to reason about systems correctness, it aims to provide primitive (indivisible) operators called *atoms* as building blocks for independently developing complex aspects of a system's execution through means of composition or generalization.

3. in order to express complex behaviors with a minimal grammar, it decouples structure (composition) from meaning (semantics), the only semantics carriers being atoms. Thus complex behaviors can be described in terms of *patterns of atoms*. Using ad-hoc polymorphism, atoms can be overloaded with different semantics triggered by the data type they input, whereas their composition is always the same.

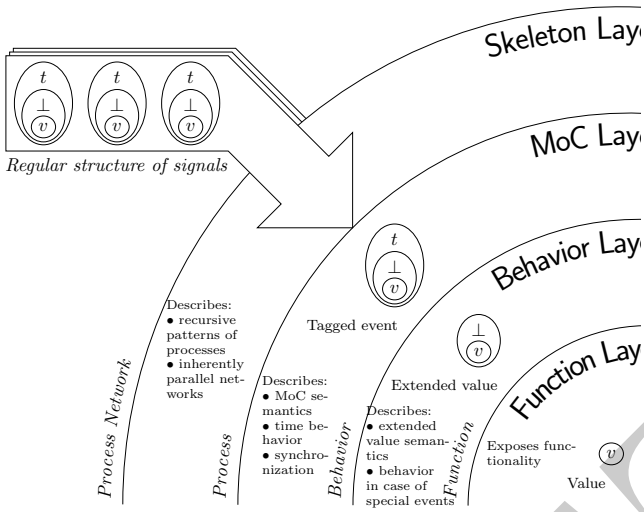
`atom` the elementary (primitive, indivisible) constructor which embeds a set of semantics relevant for their respective layer (e.g. timing, behavioural, structural, etc.)

`atom patterns` meaningful compositions of atoms. They are provided as constructors which need to be properly instantiated in order to be used. We also use the term "pattern" to differentiate atom compositions as constructors from atoms as constructors.

The first policy, i.e. the separation of concerns led to the so-called *layered process model* which is reflected in the library by providing separate major modules associated with each layer. Layers as such are independent collections of entities for modeling different aspects of CPS. These aspects interact through means of higher-order functions, wrapping each other in as structured fashion in a way which can be visualized as below.

Layers are implemented as type classes which imply:

- **atoms** as function signatures belonging to the type class;
- **patterns** which are compositions atoms, provided as constructors;
- **data types** for all the classes of behaviors concerning the aspect described by the layer in question. These types instantiate the above type class and overload the atoms with semantics in accordance to the behavior described. For example, the `MoC` layer is currently instantiated by types describing the `CT`, `DE`, `SY` and `SDF` MoCs.



In order to model interleaving aspects of CPS, layers interact with each other through means of higher order functions. As such, each layer describes some atoms as higher-order functions which take entities belonging to another layer as arguments. Intrinsically, the data types belonging to a layer may be wrapping types of other layers, as depicted in the figure above. For a short comprehensive overview on layers, please refer to Ungureanu and Ingo Sander, 2017.

By convention, the first (innermost) layer is always the *function layer* which describes arbitrary functions on data and expresses the system's functional aspects. In the following paragraphs we will give an overview of the "outer" layers currently implemented in `forsyde-atom`, which in comparison, express the extra-functional aspects of a system (timing, behavior, synchronization, and so on).

4.2.2 The Extended Behavior (ExB) Layer

As seen in layered process model, the extended behavior layer expands the set of possible behaviors implied by a layer (typically the function layer), by defining a set of symbols with *known* semantics, and adding it to (i.e. wrapping) the pool of possible values or states.

While semantically the `ExB` layer extends the value pool in order to express special events (e.g. error messages or even the complete absence of events), it practically provides an independent environment to model events with a default/known response, independently of the data path. These responses are particularly captured by atoms, thus enforcing the high-level separation of concerns between e.g. control and data paths.

This layer provides:

- a set of extended behavior atoms defining the interfaces for the resolution and response functions, as part of the `ExB` type class *(see below)*.
- a library of function wrappers as specific atom patterns (*Check the `ForSyDe.Atom.ExB` module for extensive documentation*).
- a set of data types defining classes of behaviors and instantiating the `ExB` type class (*check the links in the instances section for extensive documentation*).

`class Functor b => ExB b where`
: Class which defines the atoms for the extended behavior layer.

As its name suggests, this layer is extending the behavior of processes (or merely of functions if we ignore timing semantics completely), by expanding the domains of the wrapped layer (e.g. the set of values) with symbols having clearly-defined semantics (e.g. special events with known responses).

The types associated with this layer can simply be describes as:

$$\mathcal{B}(\alpha) = \alpha \cup b$$

where α is a base type and b is the type extension, i.e. a set of symbols with clearly-defined semantics.

Extended behavior atoms are functions of these types, defined as interfaces in the `ExB` type class.

Methods

`extend :: a -> b a`
: Extends a value (from a layer below) with a set of symbols with known semantics, as described by a type instantiating this class.

`(/.\) :: (a -> a) -> b a -> b a`
: Basic functor operator. Lifts a function (from a layer below) into the domain of the extended behavior layer.

$$\triangle : (\alpha \to \beta) \to \mathcal{B}(\alpha) \to \mathcal{B}(\beta)$$

`(/*\) :: b (a -> a) -> b a -> b a`
: Applicative operator. Defines a res between two extended behavior symbols.

$$\triangleq : \mathcal{B}(\alpha \to \beta) \to \mathcal{B}(\alpha) \to \mathcal{B}(\beta)$$

`(/&\) :: b Bool -> b a -> b a`
: Predicate operator. Generates a defined behavior based on an extended Boolean predicate.

$$\triangleq : \mathcal{B}(\texttt{Bool}) \to \mathcal{B}(\alpha) \to \mathcal{B}(\alpha)$$

`(/!\) :: a -> b a -> a`
: Degrade operator. Degrades a behavior-extended value into a non-extended one (from a layer below), based on a kernel value. Used also to throw exceptions.

$$\triangleq : \alpha \to \mathcal{B}(\alpha) \to \alpha$$

`instance ExB AbstExt`
: Implements the absent semantics of the extended behavior atoms.

4.2.3 The Model of Computation (MoC) Layer

This layer represents a major part of the `forsyde-atom` library and is concerned in modeling the timing aspects of CPS. While its foundations have been layered in the classical ForSyDe I. Sander and Jantsch, 2006, it is mainly inspired from Lee and Sangiovanni-Vincentelli, 1998 an it tries to follow the tagged signal model as closely as it is permitted by the host language, and with the adaptations require by the atom approach.

Although a short introduction of the tagged signal model has been written in the introduction of this documentation, we feel obliged to provide a primer in the classical ForSyDe theory in order to understand how everything fits together.

Signals

Lee and Sangiovanni-Vincentelli, 1998 defines signals as (ordered) sets of events where each event is composed of a tag T and a value V. Similarly, in ForSyDe a signal is defined as a (partially

or totally) *ordered sequence* of events that enables processes to communicate and synchronize. Sequencing might infer an implicit order of events, but more importantly it determines an order of evaluation, which is a key piece of a simulation engine.

$$
\begin{aligned}
& s = \{e_0, e_1, ...\} \in S \\
& \text{where } e_j = (t_j, v_j) \\
& \qquad v_j \in V,\ t_j \in T \\
& \qquad t_j \leq t_{j+1},\ \forall\ j \in \mathbb{N}
\end{aligned}
$$

In ForSyDe-Atom, sequencing is achieved using the `Stream` data type, inspired from Reekie, 1995. In ForSyDe-Atom, signals are streams that carry *events*, where each type of event is identified by a type constructor which defines its tag system. In other words, we can state that through its tag system, a signal is *bound* to a MoC.

```
data Stream e
```

`=`	`NullS`	terminates a signal
`|`	`e (:-) (Stream e)`	the default constructor appends an event to the head of the stream

Defines a stream of events, encapsulating them in a structure isomorphic to an infinite list R. S. Bird, 1987, thus all properties of lists may also be applied to `Stream`s. While, in combination with lazy evaluation, it is possible to create and simulate infinite signals, we need to ensure that the first/previous event is always fully evaluated. This can be translated into the following rule:

zero-delay feedbacks are forbidden, due to un-evaluated self-referential calls. In a feedback loop, there always has to be enough events to ensure the data flow.

This rule imposes that the stream of data is uninterrupted in order to have an evaluatable kernel every time a new event is produced (i.e. to avoid deadlocks). Thus we can add the rule:

cleaning of output events is also forbidden. In other words, for each new input at any instant in time, a process must react with *at least* one output event.

```
instance Functor Stream
```
allows for the mapping of an arbitrary function `(a -> b)` upon all the events of a `(Stream a)`.

```
instance Applicative Stream
```
enables the `Stream` to behave like a `ZipList`

```
instance Foldable Stream
```
provides folding functions useful for implementing utilities, such as `length`.

```
instance Read a => Read (Stream a)
```
signal `(1 :- 2 :- NullS)` is read using the string `"{1,2}"`.

```
instance Show a => Show (Stream a)
```
signal `(1 :- 2 :- NullS)` is represented as `{1,2}`.

```
instance Plottable a => Plot (Signal a)
```
`SY` signals.

```
instance Plottable a => Plot (Signal a)
```
`SDF` signals.

```
instance Plottable a => Plot (Signal a)
```
`DE` signals.

`instance Plottable a => Plot (Signal a)`
`CT` signals.

For extended documentation and a list of all utilities associated with the `Stream` type you can consult:

```
module ForSyDe.Atom.MoC.Stream
```

Processes

As described in Lee and Sangiovanni-Vincentelli, 1998, processes are either "set of possible behaviors" of signals or "relations" between multiple signals. One can describe complex systems by composing processes, which in this case is interpreted as the "intersection of the behaviors of each of the processes being involved".

monotonicity In order to ensure causal order and determinancy, processes need to be *monotonic* Lee and Sangiovanni-Vincentelli, 1998. A signal's tags (if explicit) *must be* a partial or total order and all tag alterations must be monotonic.

ForSyDe inherits this definition with respect to a functional view, thus a **process** p is a functional mapping over (the history of) signals. A process can *only* be instantiated using a **process constructor** pc, which is a higher order function embedding MoC semantics and/or a specific composition, but lacking functionality.

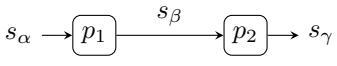
$$p : S^m \rightarrow S^n \in P \qquad pc_{\mathrm{M}} : V^n \rightarrow P \in PC$$
$$p(s, ...) = (s, ...) \qquad pc_{\mathrm{M}}(v, ...) = p$$

Since processes are functions, process composition is equivalent to function composition. This means that composing two processes `p1` and `p2` grants the process `p2 . p1`

```
p1      :: Signal α -> Signal β
p2      :: Signal β -> Signal γ
p2 . p1 :: Signal α -> Signal γ
```

This implies that there is a signal `Signal` β that "streams" the result from `p1` to `p2`, as suggested in the drawing:

$$p_1 : S_\alpha \rightarrow S_\beta$$
$$p_2 : S_\beta \rightarrow S_\gamma$$
$$p_2 \circ p_1 : S_\alpha \rightarrow S_\gamma$$
$$(p_2 \circ p_1)(s) \equiv p_2(p_1(s))$$

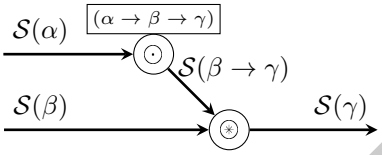


Process networks describe ForSyDe systems in terms of compositions of processes and originate from Reekie's process nets Reekie, 1995. A process network is a process itself, i.e. function from signal(s) to signal(s). The composition above `p2 . p1` can also be regarded as a process network.

In ForSyDe-Atom atoms can be regarded as process constructors as their instantiations are functions on signals of events. Instantiations of atom patterns are the exact equivalent of process networks, which themselves are also processes, depending on the level of abstraction you are working with (hierarchical blocks vs. flat structures).

To understand the versatility of composition and partial application in building process constructors, consider the example above where composition of two processes infers a signal between them. This mechanism also works when composing constructors (un-instantiated atoms), which yields another constructor. By instantiating (fully applying) the new constructor we obtain a process network equivalent to the composition of the respective primitive processes obtained by instantiating (fully applying) the component atoms, like in the example below:

$$
\begin{aligned}
p_1(s_1) &= (+) \odot s_1 \\
p_2(s_1, s_2) &= s_1 \circledast s_2 \\
pn(s_1, s_2) &= (p_2(s_2) \circ p_1)(s_1) \\
pc'(f)(s_1, s_2) &= (\circledast s_2) \circ (f \odot s_1) \\
p'(s_1, s_2) &= pc'\,(+)(s_1, s_2) \\
pn &\equiv p'
\end{aligned}
$$



Now if we visualize process networks as graphs, where processes are nodes and signals are edges, a meaningful process composition could be regarded as graph patterns. Therefore it is safe to associate process constructors as patterns in process networks.

Models of Computation

As mentioned in the introduction, *MoCs* are classes of behaviors dictating the semantics of execution and concurrency in a network of processes. Based on the definitions of their tag systems ForSyDe identifies MoCs as:

1. *timed* where T is a totally ordered set and t express the notion of physical time (e.g. continuous time `CT`, discrete event `DE`) or precedence (e.g. synchronous `SY`);
2. *untimed*, where T is a partially ordered set and t is expressed in terms of constraints on the tags in signals (e.g. dataflow, synchronous data flow `SDF`).

As concerning MoCs, ForSyDe implements the execution semantics *through process constructors*, abstracting the timing model and inferring a schedule of the process network. In ForSyDe-Atom all atoms embed operating semantics dictated by a certain MoC and are side-effect-free. This ensures the functional correctness of a system even from early design stages.

Representing Time

For explicit time representation, ForSyDe-atom provides two distinct data types.

`type Time = Rational`
: Type alias for the type to represent metric (continuous) time. Underneath we use `Rational` that is able to represent any t between $t_1 < t_2 \in T$.

`type TimeStamp = DiffTime`
: Alias for the type representing discrete time. It is inherently quantizable, the quantum being a picosecond (10^{-12} seconds), thus it can be considered order-isomorphic with a set of integers, i.e. between any two timestamps there is a finite number of timestamps. Moreover, a timestamp can be easily translated into a rational number representing fractions of a second, so the conversion between timestamps (discrete time) and rationals (analog/continuous time) is straightforward.

This type is used in the explicit tags of the `DE` MoC (and subsequently the discrete event evaluation engine for simulating the `CT` MoC).

MoC Layer Overview

This layer consists of:

- 4 atoms as infix operators, implemented as methods of the type class `MoC`. Since each MoC is determined by its tag system, we expose this which are instances of this class. Thus an event's type will trigger an atom to behave in accordance to its associated MoC.
- a library of meaningful atom patterns as process constructors. (*Check the `ForSyDe.Atom.MoC` module for extensive documentation*).
- a set of data types defining tag systems through the structure of events (i.e. $T \times V$). They are instances of the `MoC` type class and define the rules of execution that will trigger an atom to behave in accordance to its associated MoC. For each supported MoC, `forsyde-atom` provides a module which defines the signal (event) type, but also a set of utilities and process constructors as specific instantiations of atom patterns. (*Check the links in the instances section for extensive documentation*).

`class Applicative e => MoC e where`

This is a type class defining interfaces for the MoC layer atoms. Each model of computation exposes its tag system through a unique event constructor as an instance of this class, which defines $T \times V$.

To express all possible MoCs which can be described using a *tagged signal model* we need to capture the most general form of their atoms. Recall that all atoms in the layered framework are represented as higher-order functions on structured types (instances of this class), taking functions of other (lower) layers as arguments. While this principle stands also for this layer, the functions taken as arguments need to be formatted for each MoC in particular in order to capture additional information, which we can call in general terms as the *execution context*.

One typical example of additional information is the consumption and production rates of for data flow MoCs (e.g. SDF). In this case the passed functions are defined over "partitions" of events, i.e. groupings of events with the same partial order in relation to, for example, a process firing. The formal description of such a "formatted function" taken as argument by a MoC entity is:

$$\dot{a} \rightarrow \dot{b} \equiv \dot{\alpha}_1 \times \dot{\alpha}_2 \times ... \rightarrow \dot{\beta}_1 \times \dot{\beta}_2 \times ...$$

where a and b might be Cartesian products of different types, corresponding to how many signals the constructor is applied to or how many signals it yields, and each type is expressed as:

$$\dot{\alpha} = \alpha^r$$

$$r \in \begin{cases} \{1\}, & \text{for all timed MoCs (e.g. SY, DE, CT)} \\ \mathbb{N}, & \text{for static variants of SDF} \\ \mathbb{N}^n, & \text{for CSDF} \\ S \times \mathbb{N} \rightarrow \mathbb{N}, & \text{where } S \text{ is a state space,} \\ & \text{in the most general case of untimed data flow} \end{cases}$$

While, as you can see above, the execution context can be extracted from the type information, working with type-level parameters is not a trivial task in Haskell, especially if we want to describe a general and extensible type class. This is why we have chosen a pragmatic approach in implementing the `MoC` class:

- any (possible) Cartesian product of α is represented using a recursive type, namely a list $[\alpha]$.
- as the execution context cannot (or can hardly) be extracted from the recursive type, in the most general case we pass both context *and* argument as a pair (see each instance in particular). To aid in pairing contexts with each argument in a function, the `ctxt` utilities are provided (see `ctxt22`).
- this artifice was masked using the generic type families `Fun` and `Res`.

Methods

`Fun e a b`

This is a type family alias for a context-bound function passed as an argument to a MoC atom. In the most simple case it can be regarded as an enhanced -> type operator. While hiding the explicit definition of arguments, this implementation choice certainly has its advantages in avoiding unnecessary or redundant type constructors (see version 0.1.1 and prior). Aliases are replaced at compile time, thus not affecting run-time performance.

$$\dot{\alpha} \to \beta$$

`Ret e b`

As with `Fun`, this alias hides a context-bound value (e.g. function return). Although the definition seems to be redundant with `Fun`, this alias is needed for utilities to recreate clean types again (see `-*`).

$$\dot{\beta}$$

`(-.-) :: Fun e a b -> Stream (e a) -> Stream (e b)`

This atom is mapping a function on values (in the presence of a context) to a signal, i.e. stream of tagged events. As ForSyDe deals with *determinate*, *functional* processes, this atom defines the (only) *behavior* of a process in rapport to one input signal Lee and Sangiovanni-Vincentelli, 1998.

$$\odot : (\dot{\alpha} \to \beta) \to \mathcal{S}(\alpha) \to \mathcal{S}(\beta)$$

`(-*-) :: Stream (e (Fun e a b)) -> Stream (e a) -> Stream (e b)`

This atom synchronizes two signals, one carrying functions on values (in the presence of a context), and the other containing values, during which it applies the former on the latter. As concerning the process created, this atom defines a *relation* between two signals Lee and Sangiovanni-Vincentelli, 1998.

$$\circledast : \mathcal{S}(\dot{\alpha} \to \beta) \to \mathcal{S}(\alpha) \to \mathcal{S}(\beta)$$

`(-*) :: Stream (e (Ret e b)) -> Stream (e b)`

Artificial *utility* which drops the context and/or partitioning yielding a clean signal type.

$$\mathcal{S}(\dot{\beta}) \to \mathcal{S}(\beta)$$

`(-<-) :: Stream (e a) -> Stream (e a) -> Stream (e a)`

This atom appends a (partition of) events at the beginning of a signal. This atom is necessary to ensure *complete partial order* of a signal and assures the *least upper bound* necessary for example in the evaluation of feedback loops Lee and Sangiovanni-Vincentelli, 1998.

$$\circledcirc : \dot{\alpha} \to \mathcal{S}(\alpha) \to \mathcal{S}(\alpha)$$

Notice the difference between the formal and the implemented type signatures. In the implementation the value/partition is wrapped inside an event type to enable smooth composition. You might also notice the type for the initial event(s) as being wrapped inside a signal constructor. This allows defining an DSL for this layer which is centered around signals exclusively, while also enabling to define atoms as homomorphisms to certain extent R. Bird and Moor, 1997. Certain MoCs might have additional constraints on the first operand to be finite.

`(-&-) :: Stream (e a) -> Stream (e a) -> Stream (e a)`

This atom allows the manipulation of tags in a signal in a restrictive way which preserves *monotonicity* and *continuity* in a process Lee and Sangiovanni-Vincentelli, 1998, namely by phase-shifting all tags in a signal with the appropriate metric corresponding to each MoC. Thus it preserves the characteristic function intact I. Sander and Jantsch, 2006.

$$\oplus : T \to \mathcal{S}(\alpha) \to \mathcal{S}(\alpha)$$

As with the `-<-` atom, we can justify the type signature for smooth composition and the definition of atoms as homomorphisms to certain extent. This in turn allows the interpretation of the `-Methods-` operator as aligning the phases of two signals: the second operand is aligned based on the first.

`instance MoC SY`

Implenents the execution and synchronization semantics for the SY MoC through its atoms.

`instance MoC SDF`

Implenents the SDF semantics for the MoC atoms

`instance MoC DE`

Implenents the execution and synchronization semantics for the DE MoC through its atoms.

`instance MoC CT`

Implenents the execution and synchronization semantics for the CT MoC through its atoms.

4.2.4 The Skeleton Layer

The skeleton layer describes recursive and regular composition of processes which expose inherent potential for parallelism. As such, it wraps lower layer entities (i.e. processes, signals), into regular structures called *categorical types*. Most of the ground work for this layer is based on the categorical type theory R. Bird and Moor, 1997, which enable the description of algorithmic skeletons as high-level constructs encapsulating parallelism and communication with an associated cost complexity.

This layer provides:

- 3 atoms as infix operators which, as demonstrated in R. Bird and Moor, 1997 and Skillicorn, 1994, are enough to describe *all* algorithmic skeletons.
- a library of generic skeletons as specific atom patterns. (*Check the* `ForSyDe.Atom.Skeleton` *module for extensive documentation*).
- a set of different categorical types which implement these atoms, as instances of the `Skeleton` type class. These types provide additional skeletons patterns of atoms which takes as arguments their own type constructors. (*Check the links in the instances section for extensive documentation*).

`class Functor c => Skeleton c where`

Class containing all the Skeleton layer atoms.

This class is instantiated by a set of categorical types, i.e. types which describe an inherent potential for being evaluated in parallel. Skeletons are patterns from this layer. When skeletons take as arguments entities from the MoC layer (i.e. processes), the results themselves are parallel process networks which describe systems with an inherent potential to be implemented on parallel platforms. All skeletons can be described as composition of the three atoms below (`=«=` being just a specific instantiation of `=\=`). This possible due to an existing theorem in the categorical type theory, also called the Bird-Merteens formalism R. Bird and Moor, 1997:

factorization A function on a categorical type is an algorithmic skeleton (i.e. catamorphism) *iff* it can be represented in a factorized form, i.e. as a *map* composed with a *reduce*.

Consequently, most of the skeletons for the implemented categorical types are described in their factorized form, taking as arguments either:

- type constructors or functions derived from type constructors
- processes, i.e. MoC layer entities

Most of the ground-work on algorithmic skeletons on which this module is founded has been laid in R. Bird and Moor, 1997, Skillicorn, 1994 and it founds many of the frameworks collected in Fischer, Gorlatch, and Bischof, 2003.

Methods

`(=.=) :: (a -> b) -> c a -> c b`

Atom which maps a function on each element of a structure (i.e. categorical type), defined as:

$$\diamondsuit : (\alpha \to \beta) \to \mathcal{S}(\alpha) \to \mathcal{S}(\beta)$$

`=.=` together with `=*=` form the map pattern.

`(=*=) :: c (a -> b) -> c a -> c b`

Atom which applies the functions contained by as structure (i.e. categorical type), on the elements of another structure, defined as:

$$\diamondsuit : \mathcal{C}(\alpha \to \beta) \to \mathcal{C}(\alpha) \to \mathcal{C}(\beta)$$

`=.=` together with `=*=` form the map pattern.

`(=\=) :: (a -> a -> a) -> c a -> a`

Atom which reduces a structure to an element based on an *associative* function, defined as:

$$\diamondsuit : (\alpha \times \alpha \to \alpha) \to \mathcal{C}(\alpha) \to \alpha$$

`(=«=)`

Skeleton which *pipes* an element through all the functions contained by a structure. N.B.: this is not an atom. It has an implicit definition which might be augmented by instances of this class to include edge cases.

$$\diamondsuit : \mathcal{C}(\alpha \to \alpha) \times \alpha \to \alpha$$
$$\diamondsuit = (\circ)\diamondsuit$$

As the composition operation is not associative, we cannot treat pipe as a true reduction. Alas, it can still be exploited in parallel since it exposes another type of parallelism: time parallelism.

`first :: c a -> a`

Returns the first element in a structure.N.B.: this is not an atom. It has an implicit definition which might be replaced by instances of this class with a more efficient implementation.

$$\texttt{first}_S : \mathcal{C}(\alpha) \to \alpha$$
$$\texttt{first}_S = (\lambda\ a\ b \to a)\diamondsuit$$

`last :: c a -> a`

Returns the last element in a structure.N.B.: this is not an atom. It has an implicit definition which might be replaced by instances of this class with a more efficient implementation.

$$\texttt{last}_S : \mathcal{C}(\alpha) \to \alpha$$
$$\texttt{last}_S = (\lambda\ a\ b \to b)\diamondsuit$$

`instance Skeleton Vector`

Ensures that `Vector` is a structure associated with the Skeleton Layer.

4.2.5 Utilities

The `Atom` module exports a set of utility functions, mainly for aiding the designer to avoid working with zipped tuples which might pollute the design. Utilities are function without any semantical value (thus not considered atoms). They operate on and might alter the *structure* of some datum, but it does not affect its state.

For a list of all the provided utilities, please consult the following module:

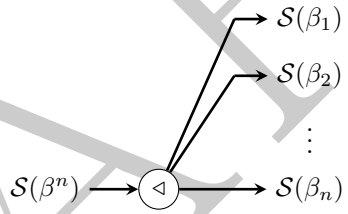
```
module ForSyDe.Atom.Utility
```

Among the most useful utilities we mentions the `unzip` function. Recall that in all our definitions for patterns, they were expressed in the most general form as functions from n-ary Cartesian products to m-ary Cartesian products. While partial application provides a versatile mechanism that can translate n-ary inputs into curried arguments (which is very powerful in combination with an applicative style), we cannot do so for return types. For the latter we must rely on tuples. But working with tuples of data wrapped in several layers of structures becomes extremely cumbersome. Take for example the case of a process constructed with *pc* in equation (1) below. Using only the provided atoms to implement *pc* would give us a process which returns only one signal of a tuple and not, as we would like, a tuple of signals of events.

$$pc : (\alpha \to \beta^n) \to \mathcal{S}(\alpha) \to \mathcal{S}(\beta^n) \quad (1)$$

$$\triangleleft : \mathcal{S}(\beta^n) \to \mathcal{S}(\beta)^n \quad (2)$$

$$(pc\, f \triangleleft) : \mathcal{S}(\alpha) \to \mathcal{S}(\beta)^n \quad (3)$$



Therefore, by implementing all data types associated with signals and events as instances of `Functor`, we were able to provide a (set of) *unzip* utility functions defined as in equation (2) above, in the `ForSyDe.Atom.Utility` module. Mind that we call *unzip* a utility and not an atom, since it has no synchronization nor behavior semantic. It just conveniently "lifts" the wrapped tuples in order to create "collections" of events and signals, and it is imposed by the mechanisms of the type system in the host language.

```
(||<) :: (Functor f1, Functor f2) =>
    f1 (f2 (a1, a2)) -> (f1 (f2 a1), f1 (f2 a2))
```

This set of utility functions "unzip" nested n-tuples, provided as postfix operators. They are crucial for reconstructing data types from higher-order functions which input functions with multiple outputs. It relies on the nested types being instances of `Functor`.

The operator convention is `(|+<+)`, where the number of `|` represent the number of layers the n-tuple is lifted, while the number of `<` + 1 is the order n of the n-tuple.

`ForSyDe.Atom.Utility` exports the constructors below. Please follow the examples in the source code if they do not suffice:

```
   |<,    |<<,    |<<<,    |<<<<,    |<<<<<,    |<<<<<<,    |<<<<<<<,    |<<<<<<<<,
  ||<,   ||<<,   ||<<<,   ||<<<<,   ||<<<<<,   ||<<<<<<,   ||<<<<<<<,   ||<<<<<<<<,
 |||<,  |||<<,  |||<<<,  |||<<<<,  |||<<<<<,  |||<<<<<<,  |||<<<<<<<,  |||<<<<<<<<,
||||<, ||||<<, ||||<<<, ||||<<<<, ||||<<<<<, ||||<<<<<<, ||||<<<<<<<, ||||<<<<<<<<,
```

Example:

```
λ> :set -XPostfixOperators
λ> ([Just (1,2,3), Nothing, Just (4,5,6)] ||<<)
([Just 1,Nothing,Just 4],[Just 2,Nothing,Just 5],[Just 3,Nothing,Just 6])
```

4.3 ForSyDe.Atom.ExB

```
module ForSyDe.Atom.ExB (
    ExB(extend, (/.\), (/*\), (/&\), (/!\)),  res22,  filter,  filter',
    degrade,  ignore22
  ) where
```

This module exports the core entities of the extended behavior layer: interfaces for atoms and common patterns of atoms. It does *NOT* export any implementation or instantiation of any specific behavior extension type.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

4.3.1 Atoms

`class Functor b => ExB b where`

Class which defines the atoms for the extended behavior layer.

As its name suggests, this layer is extending the behavior of processes (or merely of functions if we ignore timing semantics completely), by expanding the domains of the wrapped layer (e.g. the set of values) with symbols having clearly-defined semantics (e.g. special events with known responses).

The types associated with this layer can simply be describes as:

$$\mathcal{B}(\alpha) = \alpha \cup b$$

where α is a base type and b is the type extension, i.e. a set of symbols with clearly-defined semantics.

Extended behavior atoms are functions of these types, defined as interfaces in the `ExB` type class.

Methods

`extend :: a -> b a Source`

Extends a value (from a layer below) with a set of symbols with known semantics, as described by a type instantiating this class.

`(/.\) :: (a -> a) -> b a -> b a`

Basic functor operator. Lifts a function (from a layer below) into the domain of the extended behavior layer.

$$\triangle : (\alpha \to \beta) \to \mathcal{B}(\alpha) \to \mathcal{B}(\beta)$$

`(/*\) :: b (a -> a) -> b a -> b a`

Applicative operator. Defines a res between two extended behavior symbols.

$$\triangleq : \mathcal{B}(\alpha \to \beta) \to \mathcal{B}(\alpha) \to \mathcal{B}(\beta)$$

`(/&\) :: b Bool -> b a -> b a Source`

Predicate operator. Generates a defined behavior based on an extended Boolean predicate.

$$\triangleq : \mathcal{B}(\texttt{Bool}) \to \mathcal{B}(\alpha) \to \mathcal{B}(\alpha)$$

`(/!\) :: a -> b a -> a Source`
: Degrade operator. Degrades a behavior-extended value into a non-extended one (from a layer below), based on a kernel value. Used also to throw exceptions.

$$\triangleleft : \alpha \rightarrow \mathcal{B}(\alpha) \rightarrow \alpha$$

`instance ExB AbstExt`
: Implements the absent semantics of the extended behavior atoms.

4.3.2 Patterns

`res22`

`::`	ExB b	
`=>`	`(a1 -> a2 -> (a1', a2'))`	function on values
`->`	`b a1`	first input
`->`	`b a2`	second input
`->`	`(b a1', b a2')`	tupled output

$$\triangleleft : (V^n \rightarrow V^m) \rightarrow \mathcal{B}^n \rightarrow \mathcal{B}^m$$
$$f \triangleleft (b_1, b_2, ...b_n) = f \triangleleft b_1 \triangleleft b_2 \triangleleft ... \triangleleft b_n \triangleleft$$

The `res` behavior pattern lifts a function on values to the extended behavior domain, and applies a resolution between two extended behavior symbols.

Constructors: `res[1-8][1-4]`.

`filter :: ExB b => b Bool -> b a -> b a`
: Prefix name for the prefix operator `/&\`.

`filter' :: ExB b => Bool -> a -> b a`
: Same as `filter` but takes base (non-extended) values as input arguments.

`degrade :: ExB b => a -> b a -> a`
: Prefix name for the degrade operator `/!\`.

`ignore22`

`::`	ExB b	
`=>`	`(a1 -> a2 -> a1' -> a2' -> (a1, a2))`	function of `Y + X` arguments
`->`	`a1`	
`->`	`a2`	
`->`	`b a1'`	
`->`	`b a2'`	
`->`	`(a1, a2)`	

$$\texttt{ignore}_B : (V^n \times V^m \rightarrow V^n) \rightarrow V^n \times \mathcal{B}^m \rightarrow V^n$$
$$\texttt{ignore}_B(f)(a^n, b^m) = a^n \triangleleft (f(a^n) \triangleleft b^m)$$

The `ignoreXY` pattern takes a function of `Y + X` arguments, `Y` basic inputs followed by `X` behavior-extended inputs. The `X` behavior-extended arguments are subjugated to a res, and the result is then degraded using the first `Y` arguments as fallback. The effect is similar to "ignoring" a the result of a res function if $\in b$.

The main application of this pattern is as extended behavior wrapper for state machine functions which do not "understand" extended behavior semantics, i.e. it simply propagates the current state ($\in \alpha$) if the inputs (their res) belongs to the set of extended values ($\in b$).

Constructors: `ignore[1-4][1-4]`.

4.4 ForSyDe.Atom.ExB.Absent

```
module ForSyDe.Atom.ExB.Absent (
    AbstExt(Abst, Prst)
  ) where
```

This module implements the constructors and assocuated utilities of a type which extends the behavior of a function to express "absent events" (see Halbwachs et al., 1991).

The `AbstExt` type can be used directly with the atom patterns defined in `ForSyDe.Atom.ExB`, and no helpers or utilities are needed. Example usage:

```
λ> res21 (+) (Prst 1) (Prst 2)
3
λ> res21 (+) Abst     Abst
⊥
λ> filter Abst         (Prst 1)
⊥
λ> filter (Prst False) (Prst 1)
⊥
λ> filter (Prst True)  (Prst 1)
1
λ> filter' False 1 :: AbstExt Int
⊥
λ> filter' True  1 :: AbstExt Int
1
λ> degrade 0 (Prst 1)
1
λ> degrade 0 Abst
0
λ> ignore11 (+) 1 (Prst 1)
2
λ> ignore11 (+) 1 Abst
1
λ> res21 (+) (Prst 1) Abst
*** Exception: [ExB.Absent] Illegal occurrence of an absent and present event
```

```
data AbstExt a
```

`=` `Abst` ⊥ denotes the absence of a value

`|` `Prst a` ⊤ a present event with a value

The `AbstExt` type extends the base type with the '⊥' symbol denoting the absence of a value/event (see Halbwachs et al., 1991).

```
instance Functor AbstExt
```

`Functor` instance. Bypasses the special values and maps a function to the wrapped value. Provides the `(<$>)` operator.

```
instance Applicative AbstExt
```

`Applicative` instance. Check source code for the lifting rules. Provides the `(<*>)` operator

```
instance ExB AbstExt
```

Implements the absent semantics of the extended behavior atoms.

```
instance Eq a => Eq (AbstExt a)
```

```
instance Read a => Read (AbstExt a)
```

Reads the '_' character to an `Abst` and a normal value to `Prst`-wrapped one.

`instance Show a => Show (AbstExt a)`
: Shows `Abst` as '⊥', while a present event is represented with its value.

`instance (Show a, Plottable a) => Plottable (AbstExt a)`
: Absent-extended plottable types

4.5 ForSyDe.Atom.MoC

```
module ForSyDe.Atom.MoC (
    MoC(Fun, Ret, (-.-), (-*-), (-*), (-<-), (-&-)),  delay,  comb22,
    reconfig22,  state22,  stated22,  moore22,  mealy22,  ctxt22,  warg,  wres,
    (-*<)
  ) where
```

This module exports the core entities of the MoC layer: interfaces for atoms and process constructors as patterns of atoms. It does *NOT* export any implementation or instantiation of any specific MoC.

Current MoC implementations can be used by importing their respective modules:

- `ForSyDe.Atom.MoC.CT`
- `ForSyDe.Atom.MoC.DE`
- `ForSyDe.Atom.MoC.SY`
- `ForSyDe.Atom.MoC.SDF`

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

4.5.1 Atoms

`class Applicative e => MoC e where`
: This is a type class defining interfaces for the MoC layer atoms. Each model of computation exposes its tag system through a unique event constructor as an instance of this class, which defines $T \times V$.

  To express all possible MoCs which can be described using a *tagged signal model* we need to capture the most general form of their atoms. Recall that all atoms in the layered framework are represented as higher-order functions on structured types (instances of this class), taking functions of other (lower) layers as arguments. While this principle stands also for this layer, the functions taken as arguments need to be formatted for each MoC in particular in order to capture additional information, which we can call in general terms as the *execution context*.

  One typical example of additional information is the consumption and production rates of for data flow MoCs (e.g. SDF). In this case the passed functions are defined over "partitions" of events, i.e. groupings of events with the same partial order in relation to, for example, a

process firing. The formal description of such a "formatted function" taken as argument by a MoC entity is:

$$\dot{a} \rightarrow \dot{b} \equiv \dot{\alpha}_1 \times \dot{\alpha}_2 \times ... \rightarrow \dot{\beta}_1 \times \dot{\beta}_2 \times ...$$

where a and b might be Cartesian products of different types, corresponding to how many signals the constructor is applied to or how many signals it yields, and each type is expressed as:

$$\dot{\alpha} = \alpha^r$$

$$r \in \begin{cases} \{1\}, & \text{for all timed MoCs (e.g. SY, DE, CT)} \\ \mathbb{N}, & \text{for static variants of SDF} \\ \mathbb{N}^n, & \text{for CSDF} \\ S \times \mathbb{N} \rightarrow \mathbb{N}, & \text{where } S \text{ is a state space,} \\ & \text{in the most general case of untimed data flow} \end{cases}$$

While, as you can see above, the execution context can be extracted from the type information, working with type-level parameters is not a trivial task in Haskell, especially if we want to describe a general and extensible type class. This is why we have chosen a pragmatic approach in implementing the `MoC` class:

- any (possible) Cartesian product of α is represented using a recursive type, namely a list $[\alpha]$.
- as the execution context cannot (or can hardly) be extracted from the recursive type, in the most general case we pass both context *and* argument as a pair (see each instance in particular). To aid in pairing contexts with each argument in a function, the `ctxt` utilities are provided (see `ctxt22`).
- this artifice was masked using the generic type families `Fun` and `Res`.

Methods

`Fun e a b Source`

This is a type family alias for a context-bound function passed as an argument to a MoC atom. In the most simple case it can be regarded as an enhanced -> type operator. While hiding the explicit definition of arguments, this implementation choice certainly has its advantages in avoiding unnecessary or redundant type constructors (see version 0.1.1 and prior). Aliases are replaced at compile time, thus not affecting run-time performance.

$$\dot{\alpha} \rightarrow \beta$$

`Ret e b Source`

As with `Fun`, this alias hides a context-bound value (e.g. function return). Although the definition seems to be redundant with `Fun`, this alias is needed for utilities to recreate clean types again (see `-*`).

$$\dot{\beta}$$

`(-.-) :: Fun e a b -> Stream (e a) -> Stream (e b) Source`

This atom is mapping a function on values (in the presence of a context) to a signal, i.e. stream of tagged events. As ForSyDe deals with *determinate*, *functional* processes, this atom defines the (only) *behavior* of a process in rapport to one input signal Lee and Sangiovanni-Vincentelli, 1998.

$$\odot : (\dot{\alpha} \rightarrow \beta) \rightarrow \mathcal{S}(\alpha) \rightarrow \mathcal{S}(\beta)$$

`(-*-) :: Stream (e (Fun e a b)) -> Stream (e a) -> Stream (e b) Source`

This atom synchronizes two signals, one carrying functions on values (in the presence of a context), and the other containing values, during which it applies the former on the latter. As concerning the process created, this atom defines a *relation* between two signals Lee and Sangiovanni-Vincentelli, 1998.

$$\circledast : \mathcal{S}(\dot{\alpha} \rightarrow \beta) \rightarrow \mathcal{S}(\alpha) \rightarrow \mathcal{S}(\beta)$$

`(-*) :: Stream (e (Ret e b)) -> Stream (e b) Source`
Artificial *utility* which drops the context and/or partitioning yielding a clean signal type.

$$\mathcal{S}(\dot{\beta}) \to \mathcal{S}(\beta)$$

`(-<-) :: Stream (e a) -> Stream (e a) -> Stream (e a) Source`
This atom appends a (partition of) events at the beginning of a signal. This atom is necessary to ensure *complete partial order* of a signal and assures the *least upper bound* necessary for example in the evaluation of feedback loops Lee and Sangiovanni-Vincentelli, 1998.

$$\circledcirc : \dot{\alpha} \to \mathcal{S}(\alpha) \to \mathcal{S}(\alpha)$$

Notice the difference between the formal and the implemented type signatures. In the implementation the value/partition is wrapped inside an event type to enable smooth composition. You might also notice the type for the initial event(s) as being wrapped inside a signal constructor. This allows defining an DSL for this layer which is centered around signals exclusively, while also enabling to define atoms as homomorphisms to certain extent R. Bird and Moor, 1997. Certain MoCs might have additional constraints on the first operand to be finite.

`(-&-) :: Stream (e a) -> Stream (e a) -> Stream (e a) Source`
This atom allows the manipulation of tags in a signal in a restrictive way which preserves *monotonicity* and *continuity* in a process Lee and Sangiovanni-Vincentelli, 1998, namely by phase-shifting all tags in a signal with the appropriate metric corresponding to each MoC. Thus it preserves the characteristic function intact I. Sander and Jantsch, 2006.

$$\circledplus : T \to \mathcal{S}(\alpha) \to \mathcal{S}(\alpha)$$

As with the `-<-` atom, we can justify the type signature for smooth composition and the definition of atoms as homomorphisms to certain extent. This in turn allows the interpretation of the `-Methods-` operator as aligning the phases of two signals: the second operand is aligned based on the first.

`instance MoC SY`
Implenents the execution and synchronization semantics for the SY MoC through its atoms.

`instance MoC SDF`
Implenents the SDF semantics for the MoC atoms

`instance MoC DE`
Implenents the execution and synchronization semantics for the DE MoC through its atoms.

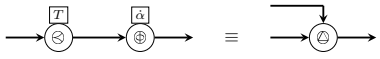
`instance MoC CT`
Implenents the execution and synchronization semantics for the CT MoC through its atoms.

4.5.2 Process constructors

Process constructors are defined as patterns of MoC atoms. Check the naming convention of the API in the page description.

```
delay :: MoC e => Stream (e a) -> Stream (e a) -> Stream (e a)
```

$$\circlearrowleft : T \times \dot{\alpha} \to \mathcal{S}(\alpha) \to \mathcal{S}(\alpha)$$



The `delay` process provides both initial token(s) and shifts the phase of the signal. In other words, it "delays" a signal with one or several events.

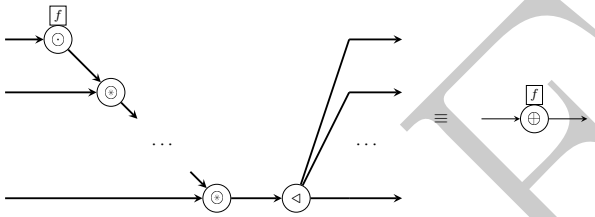
There is also an infix variant `-&>-` (`infixl 3`). To justify the first argument, see the documentation of the `-<-` atom.

`comb22`

`::`	MoC e	
`=>`	`Fun e a1 (Fun e a2 (Ret e b1, Ret e b2))`	combinational function (*)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e b1), Stream (e b2))`	two output signals

(*) to be read `a1 -> a2 -> (b1, b2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

$$\oplus : (\dot{V}^n \to \dot{V}^m) \to \mathcal{S}^n \to \mathcal{S}^m$$
$$f \oplus (s_1, s_2, ...s_n) = f \odot s_1 \circledast s_2 \circledast ... \circledast s_n \triangleleft$$



The `comb` processes takes care of synchronization between signals and maps combinatorial functions on their event values.

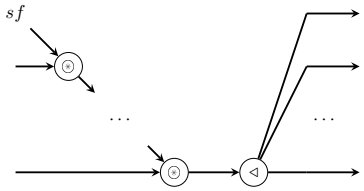
This library exports constructors of type `comb[1-8][1-4]`.

`reconfig22`

`::`	MoC e	
`=>`	`Stream (e (Fun e a1 (Fun e a2 (Ret e b1, Ret e b2))))`	signal carrying functions (*)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e b1), Stream (e b2))`	two output signals

(*) to be read `a1 -> a2 -> (b1, b2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

$$\texttt{reconfig}_M : \mathcal{S}(\dot{V}^n \to \dot{V}^m) \to \mathcal{S}^n \to \mathcal{S}^m$$
$$\texttt{reconfig}_M(sf, s_1, s_2, ...s_n) = f \odot s_1 \circledast s_2 \circledast ... \circledast s_n \triangleleft$$



The `reconfig` processes constructs adaptive processes, where the first signal carries functions, and it is synchronized with all the other signals.

This library exports constructors of type `reconfig[1-8][1-4]`.

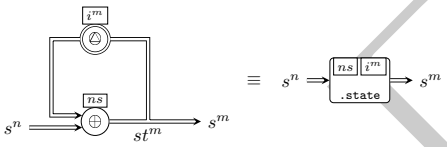
`state22`

`::`	MoC e	
`=>`	`Fun e st1 (Fun e st2 (Fun e a1 (Fun e a2 (Ret e st1, Ret e st2))))`	next state function (*)
`->`	`(Stream (e st1), Stream (e st2))`	initial state(s) (**)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e st1), Stream (e st2))`	output signals mirroring the next state(s).

(*) meaning `st1 -> st2 -> a1 -> a2 -> (st1,st2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(**) see the documentation for `-<-` *for justification of the type*

$$\texttt{state}_M : (\dot{V}^m_{st} \times \dot{V}^n \to \dot{V}^m_{st}) \times \dot{E}^m_{st} \to \mathcal{S}^n \to \mathcal{S}^m_{st}$$
$$\texttt{state}_M(ns, i^m)(s^n) = ns \oplus (st^m, s^n)$$
$$\text{where } st^m = i^m \circledcirc ns \oplus (st^m, s^n)$$



The `state` processes generate process networks corresponding to a simple state machine like in the graph above.

This library exports constructors of type `state[1-4][1-4]`.

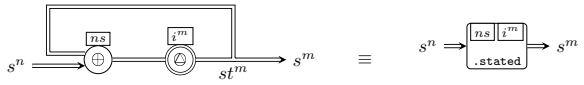
`stated22`

`::`	MoC e	
`=>`	`Fun e st1 (Fun e st2 (Fun e a1 (Fun e a2 (Ret e st1, Ret e st2))))`	next state function (*)
`->`	`(Stream (e st1), Stream (e st2))`	initial state(s) (**)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e st1), Stream (e st2))`	output signals mirroring the next state(s).

(*) meaning `st1 -> st2 -> a1 -> a2 -> (st1,st2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(**) see the documentation for `-<-` *for justification of the type*

$$\texttt{stated}_M : (\dot{V}^m_{st} \times \dot{V}^n \to \dot{V}^m_{st}) \times \dot{E}^m_{st} \to \mathcal{S}^n \to \mathcal{S}^m_{st}$$
$$\texttt{stated}_M(ns, i^m)(s^n) = st^m$$
$$\text{where } st^m = i^m \circledcirc ns \oplus (st^m, s^n)$$



The `state` processes generate process networks corresponding to a simple state machine like in the graph above. The difference between `state22` and `stated22` is that the latter outputs the current state rather than the next one. There exists a variant with 0 input signals, in which case the process is a signal generator.

This library exports constructors of type `stated[0-4][1-4]`.

`moore22`

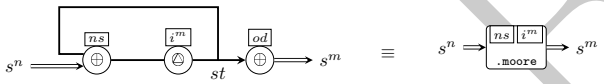
`::`	MoC e	
`=>`	`Fun e st (Fun e a1 (Fun e a2 (Ret e st)))`	next state function (*)
`->`	`Fun e st (Ret e b1, Ret e b2)`	output decoder (**)
`->`	`Stream (e st)`	initial state (***)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e b1), Stream (e b2))`	output signals

(*) meaning `st -> a1 -> a2 -> st` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(**) meaning `st -> (b1, b2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(***) see the documentation for `-<-` *for justification of the type*

$$\texttt{moore}_M : (\dot{V}_{st} \times \dot{V}^n \to \dot{V}_{st}) \times (\dot{V}_{st} \to \dot{V}^m) \times \dot{E}_{st} \to \mathcal{S}^n \to \mathcal{S}^m$$
$$\texttt{moore}_M(ns, od, i^m)(s^n) = od \oplus st^m$$
$$\text{where } st^m = i^m \oslash ns \oplus (st^m, s^n)$$



The `moore` processes model Moore state machines.

This library exports constructors of type `moore[1-4][1-4]`.

`mealy22`

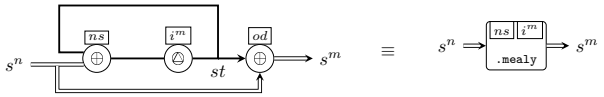
`::`	MoC e	
`=>`	`Fun e st (Fun e a1 (Fun e a2 (Ret e st)))`	next state function (*)
`->`	`Fun e st (Fun e a1 (Fun e a2 (Ret e b1, Ret e b2)))`	output decoder (**)
`->`	`Stream (e st)`	initial state (***)
`->`	`Stream (e a1)`	first input signal
`->`	`Stream (e a2)`	second input signal
`->`	`(Stream (e b1), Stream (e b2))`	output signals

(*) meaning `st -> a1 -> a2 -> st` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(**) meaning `st -> a1 -> a2 -> (b1, b2)` *where each argument and result might be individually wrapped with a context and might also express a partition.*

(***) see the documentation for `-<-` *for justification of the type*

$$\texttt{mealy}_M : (\dot{V}_{st} \times \dot{V}^n \to \dot{V}_{st}) \times (\dot{V}_{st} \times \dot{V}^n \to \dot{V}^m) \times \dot{E}_{st} \to \mathcal{S}^n \to \mathcal{S}^m$$
$$\texttt{mealy}_M(ns, od, i^m)(s^n) = od \oplus (st^m, s^n)$$
$$\text{where } st^m = i^m \oslash ns \oplus (st^m, s^n)$$



The `mealy` processes model Mealy state machines.

This library exports constructors of type `mealy[1-4][1-4]`.

4.5.3 Utilities

`ctxt22`

`::`	`(ctx, ctx)`	argument contexts (e.g. consumption rates in SDF)
`->`	`(ctx, ctx)`	result contexts (e.g. production rates in SDF)
`->`	`(a1 -> a2 -> (b1, b2))`	function on values/partitions of values
`->`	`(ctx, a1 -> (ctx, a2 -> ((ctx, b1), (ctx, b2))))`	context-wrapped form of the previous function

$$\texttt{ctxt} : \Gamma \times ([\alpha]^n \to [\beta]^m) \to (\dot{\alpha}^n \to \dot{\beta}^m)$$

Wraps a function with the context needed by some MoCs for their constructors (e.g. rates in SDF).

This library exports wrappers of type `ctxt[1-8][1-4]`.

`warg :: c -> (a -> b) -> (c, a -> b)`
Attaches a context parameter to a function argument (e.g consumption rates in SDF). Used as kernel function in defining e.g. `ctxt22`.

`wres :: p -> b -> (p, b)`
Attaches a context parameter to a function's result (e.g production rates in SDF). Used as kernel function in defining e.g. `ctxt22`.

```
(-*<) :: MoC e =>
         Stream (e (Ret e b1, Ret e b2)) -> (Stream (e b1), Stream (e b2))
```
Utilities for extending the `-*` atom for dealing with tupled outputs. This library exports operators of form `-*<{1,8}`.

4.6 ForSyDe.Atom.MoC.Stream

```
module ForSyDe.Atom.MoC.Stream (
    Stream(NullS, (:-)),  ,  ,  ,  ,  , stream, fromStream, headS, tailS,
    lastS, repeatS, takeS, dropS, takeWhileS, (+-+)
  ) where
```

This module defines the shallow-embedded `Stream` datatype and utility functions operating on it. In ForSyDe a signal is represented as a (partially or totally) *ordered* sequence of events that enables processes to communicate and synchronize. The `Stream` type is but an ordered structure to encapsulate events as infinite streams.

`data Stream e`

`=`	`NullS`	terminates a signal
`|`	`e (:-) (Stream e)`	the default constructor appends an event to the head of the stream

Defines a stream of events, encapsulating them in a structure isomorphic to an infinite list R. S. Bird, 1987, thus all properties of lists may also be applied to `Stream`s. While, in combination with lazy evaluation, it is possible to create and simulate infinite signals, we need to ensure that the first/previous event is always fully evaluated. This can be translated into the following rule:

zero-delay feedbacks are forbidden, due to un-evaluated self-referential calls. In a feedback loop, there always has to be enough events to ensure the data flow.

This rule imposes that the stream of data is uninterrupted in order to have an evaluatable kernel every time a new event is produced (i.e. to avoid deadlocks). Thus we can add the rule:

cleaning of output events is also forbidden. In other words, for each new input at any instant in time, a process must react with *at least* one output event.

`instance Functor Stream`
allows for the mapping of an arbitrary function `(a -> b)` upon all the events of a `(Stream a)`.

`instance Applicative Stream`
enables the `Stream` to behave like a `ZipList`

`instance Foldable Stream`
provides folding functions useful for implementing utilities, such as `length`.

`instance Read a => Read (Stream a)`
signal `(1 :- 2 :- NullS)` is read using the string `"{1,2}"`.

`instance Show a => Show (Stream a)`
signal `(1 :- 2 :- NullS)` is represented as `{1,2}`.

`instance Plottable a => Plot (Signal a)`
`SY` signals.

`instance Plottable a => Plot (Signal a)`
`SDF` signals.

`instance Plottable a => Plot (Signal a)`
`DE` signals.

`instance Plottable a => Plot (Signal a)`
`CT` signals.

`stream :: [a] -> Stream a`
The function `signal` converts a list into a signal.

`fromStream :: Stream a -> [a]`
The function `fromStream` converts a signal into a list.

`headS :: Stream a -> a`
Returns the head of a signal.

`tailS :: Stream e -> Stream e`
Returns the tail of a signal

`lastS :: Stream p -> p`
Returns the last event in a signal.

`repeatS :: a -> Stream a`
Returns an infinite list containing the same repeated event.

`takeS :: (Ord t, Num t) => t -> Stream e -> Stream e`
Returns the first `n` events in a signal.

`dropS :: (Ord t, Num t) => t -> Stream e -> Stream e`
Drops the first `n` events in a signal.

`takeWhileS :: (a -> Bool) -> Stream a -> Stream a`
: Returns the first events of a signal which comply to a condition.

`(+-+) :: Stream e -> Stream e -> Stream e`
: Concatenates two signals.

4.7 ForSyDe.Atom.MoC.SY

```
module ForSyDe.Atom.MoC.SY (
    SY(SY, val),  Signal,  unit2,  signal,  readSignal,  delay,  comb22,
    reconfig22,  constant2,  generate2,  stated22,  state22,  moore22,
    mealy22,  when,  when',  is,  whenPresent,  filter,  filter',  fill,  hold,
    reactAbst2,  toDE2,  toSDF2,  zipx,  unzipx,  unzipx'
  ) where
```

The SY library implements the atoms holding the sematics for the synchronous computation model. It also provides a set of helpers for properly instantiating process network patterns as process constructors.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

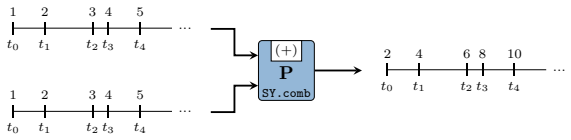
4.7.1 Synchronous (SY) event

According to Lee and Sangiovanni-Vincentelli, 1998, "two events are synchronous if they have the same tag, and two signals are synchronous if all events in one signal are synchronous to an event in the second signal and vice-versa. A system is synchronous if every signals in the system is synchronous to all the other signals in the system."

The synchronous (SY) MoC defines no notion of physical time, its tag system suggesting in fact the precedence among events. To further demystify its mechanisms, we can formulate the following proposition:

The SY MoC is abstracting the execution semantics of a system where computation is assumed to perform instantaneously (with zero delay), at certain synchronization points, when data is assumed to be available.

Below is a *possible* behavior in time of the input and the output signals of a SY process, to illustrate these semantics:



Implementing the SY tag system is straightforward if we consider the synchronous `Signal` as an infinite list. In this case the tags are implicitly defined by the position of events in a signal: t_e would correspond with the event at the head of a signal t_1 with the following event, etc... The only explicit parameter passed to a SY event constructor is the value it carries $\in V_e$. As such, we can state the following particularities:

1. tags are implicit from the position in the `Stream`, thus they are completely ignored in the type constructor.
2. the type constructor wraps only a value
3. being a *timed MoC*, the order between events is total Lee and Sangiovanni-Vincentelli, 1998.
4. from the previous statement we can conclude that there is no need for an execution context (see section 4.5) and we can ignore the formatting of functions in `ForSyDe.Atom.MoC`, thus we can safely assume:

$$\dot{a} = a$$

```
newtype SY a
    =  SY
    >  val ::  a
```

The SY event. It identifies a synchronous signal.

`instance Functor SY`
Allows for mapping of functions on a SY event.

`instance Applicative SY`
Allows for lifting functions on a pair of SY events.

`instance MoC SY`
Implenents the execution and synchronization semantics for the SY MoC through its atoms.

`instance Read a => Read (SY a)`
Reads the value wrapped

`instance Show a => Show (SY a)`
Shows the value wrapped

`instance Plottable a => Plot (Signal a)`
`SY` signals.

```
instance type Ret SY b = b
instance type Fun SY a b = a -> b
```

4.7.2 Aliases & utilities

A set of type synonyms and utilities are provided for convenience. The API type signatures will feature these aliases to hide the cumbersome construction of atoms and patters as seen in `ForSyDe.Atom.MoC`.

`type Signal a = Stream (SY a)`
Type synonym for a SY signal, i.e. "an ordered stream of SY events"

`unit2 :: (a1, a2) -> (Signal a1, Signal a2)`
Wraps a (tuple of) value(s) into the equivalent unit signal(s).

Helpers: `unit`, `unit2`, `unit3`, `unit4`.

`signal :: [a] -> Signal a`
: Transforms a list of values into a SY signal.

`readSignal :: Read a => String -> Signal a`
: Reads a signal from a string. Like with the `read` function from `Prelude`, you must specify the tipe of the signal.

```
λ> readSignal "{1,2,3,4,5}" :: Signal Int
{1,2,3,4,5}
```

4.7.3 SY process constuctors

These SY process constructors are basically specific instantiations of the patterns of atoms defined in `ForSyDe.Atom.MoC`. Some are also wrapping functions in an extended behavioural model.

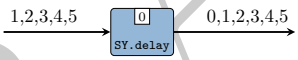
Simple

`delay`

`::`	`a`	initial value
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

The `delay` process "delays" a signal with one event. Instantiates the `delay` pattern.

```
λ> let s = signal [1,2,3,4,5]
λ> delay 0 s
{0,1,2,3,4,5}
```



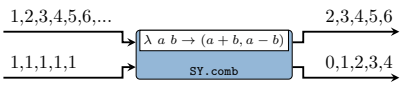
`comb22`

`::`	`(a1 -> a2 -> (b1, b2))`	function on values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	two output signals

`comb` processes map combinatorial functions on signals and take care of synchronization between input signals. It instantiates the `comb` pattern (see `comb22`).

Constructors: `comb[1-4][1-4]`.

```
λ> let s1 = signal [1..]
λ> let s2 = signal [1,1,1,1,1]
λ> comb11 (+1) s2
{2,2,2,2,2}
λ> comb22 (\a b-> (a+b,a-b)) s1 s2
({2,3,4,5,6},{0,1,2,3,4})
```



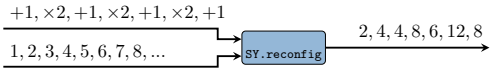
`reconfig22`

`::`	`Signal (a1 -> a2 -> (b1, b2))`	signal carrying functions
`->`	`Signal a1`	first input signal carrying arguments
`->`	`Signal a2`	second input signal carrying arguments
`->`	`(Signal b1, Signal b2)`	two output signals

`reconfig` creates an synchronous adaptive process where the first signal carries functions and the other carry the arguments. It instantiates the `reconfig` atom pattern (see `reconfig22`).

Constructors: `reconfig[1-4][1-4]`.

```
λ> let sf = signal [(+1),(*2),(+1),(*2),(+1),(*2),(+1)]
λ> let s1 = signal [1..]
λ> reconfig11 sf s1
{2,4,4,8,6,12,8}
```



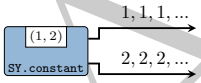
`constant2`

`::`	`(b1, b2)`	values to be repeated
`->`	`(Signal b1, Signal b2)`	generated signals

A signal generator which keeps a value constant. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `constant[1-4]`.

```
λ> let (s1, s2) = constant2 (1,2)
λ> takeS 3 s1
{1,1,1}
λ> takeS 5 s2
{2,2,2,2,2}
```



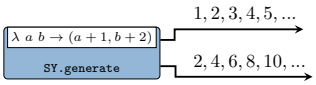
`generate2`

`::`	`(b1 -> b2 -> (b1, b2))`	function to generate next value
`->`	`(b1, b2)`	kernel values
`->`	`(Signal b1, Signal b2)`	generated signals

A signal generator based on a function and a kernel value. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `generate[1-4]`.

```
λ> let (s1,s2) = generate2 (\a b -> (a+1,b+2)) (1,2)
λ> takeS 5 s1
{1,2,3,4,5}
λ> takeS 7 s2
{2,4,6,8,10,12,14}
```



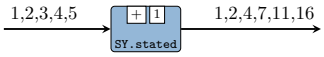
`stated22`

`::`	`(b1 -> b2 -> a1 -> a2 -> (b1, b2))`	next state function
`->`	`(b1, b2)`	initial state values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	output signals

`stated` is a state machine without an output decoder. It is an instantiation of the `state` MoC constructor (see `stated22`).

Constructors: `stated[1-4][1-4]`.

```
λ> let s1 = signal [1,2,3,4,5]
λ> stated11 (+) 1 s1
{1,2,4,7,11,16}
```



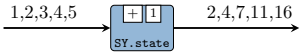
`state22`

```
::  (b1 -> b2 -> a1 -> a2 -> (b1, b2))   next state function
->  (b1, b2)                             initial state values
->  Signal a1                            first input signal
->  Signal a2                            second input signal
->  (Signal b1, Signal b2)               output signals
```

`state` is a state machine without an output decoder. It is an instantiation of the `stated` MoC constructor (see `state22`).

Constructors: `state[1-4][1-4]`.

```
λ> let s1 = signal [1,2,3,4,5]
λ> state11 (+) 1 s1
{2,4,7,11,16}
```



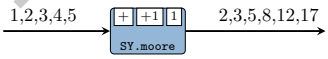
`moore22`

```
::  (st -> a1 -> a2 -> st)   next state function
->  (st -> (b1, b2))         output decoder
->  st                       initial state
->  Signal a1
->  Signal a2
->  (Signal b1, Signal b2)
```

`moore` processes model Moore state machines. It is an instantiation of the `moore` MoC constructor (see `moore22`).

Constructors: `moore[1-4][1-4]`.

```
λ> let s1 = signal [1,2,3,4,5]
λ> moore11 (+) (+1) 1 s1
{2,3,5,8,12,17}
```



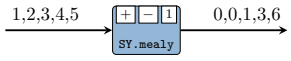
`mealy22`

```
::  (st -> a1 -> a2 -> st)          next state function
->  (st -> a1 -> a2 -> (b1, b2))    outpt decoder
->  st                              initial state
->  Signal a1
->  Signal a2
->  (Signal b1, Signal b2)
```

`mealy` processes model Mealy state machines. It is an instantiation of the `mealy` MoC constructor (see `mealy22`).

Constructors: `mealy[1-4][1-4]`.

```
λ> let s1 = signal [1,2,3,4,5]
λ> mealy11 (+) (-) 1 s1
{0,0,1,3,6}
```



Predicate behavior

These processes manipulate the behavior of a signal based on predicates on their status.

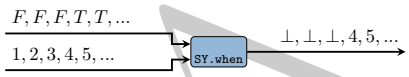
`when`

`::`	`Signal (AbstExt Bool)`	Signal of predicates
`->`	`Signal (AbstExt a)`	Input signal
`->`	`Signal (AbstExt a)`	Output signal

This process predicates the existence of values in a signal based on a signal of boolean values (conditions). It is similar to the `when` construct in the synchronous language Lustre Halbwachs et al., 1991, based on which clock calculus can be performed.

OBS: this process assumes that all signals carry absent-extended values, which is appropriate in describing multi-clock systems. For a version which inputs signals of non-extended values, check `when'`.

```
λ> let s1   = (signal . map Prst) [1,2,3,4,5]
λ> let pred = (signal . map Prst) [False,False,False,True,True]
λ> when pred s1
{⊥,⊥,⊥,4,5}
```



`when'`

`::`	`Signal Bool`	Signal of predicates
`->`	`Signal a`	Input signal
`->`	`Signal (AbstExt a)`	Output signal

Same as `when` but inputs signals of non-extended values.

```
λ> let s1   = signal [1,2,3,4,5]
λ> let pred = signal [False,False,False,True,True]
λ> when' pred s1
{⊥,⊥,⊥,4,5}
```

`is :: Signal (AbstExt a) -> (a -> Bool) -> Signal (AbstExt Bool)`

Simple wrapper for applying a predicate function on a signal of absent-extended events.

```
λ> let s1   = signal $ map Prst [1,2,3,4,5]
λ> s1 `is` (>3)
{False,False,False,True,True}
```

`whenPresent :: Signal (AbstExt a1) -> Signal (AbstExt a2) -> Signal (AbstExt a2)`

Same as `when` but triggering the output events merely based on the presence of the first input rather than a predicate function.

```
λ> let s1   = signal $ map Prst [1,2,3,4,5]
λ> let sp   = signal [Prst 1, Prst 1, Abst, Prst 1, Abst]
λ> whenPresent sp s1
{1,2,⊥,4,⊥}
```

`filter`

`::`	`(a -> Bool)`	Predicate function
`->`	`Signal (AbstExt a)`	Input signal
`->`	`Signal (AbstExt a)`	Output signal

Filters out values to `Abst` if they do not fulfill a predicate function.

OBS: this process assumes that all signals carry absent-extended values, which is appropriate in describing multi-clock systems. For a version which inputs signals of non-extended values, check `filter'`.

```
λ> let s1   = (signal . map Prst) [1,2,3,4,5]
λ> filter (>3) s1
{⊥,⊥,⊥,4,5}
```

`filter'`

`::`	`(a -> Bool)`	Predicate function
`->`	`Signal a`	Input signal
`->`	`Signal (AbstExt a)`	Output signal

Same as `filter` but inputs signals of non-extended values.

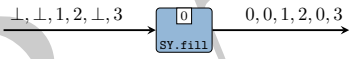
```
λ> let s1   = signal [1,2,3,4,5]
λ> filter' (>3) s1
{⊥,⊥,⊥,4,5}
```

`fill`

`::`	`a`	Value to fill with
`->`	`Signal (AbstExt a)`	Input
`->`	`Signal a`	Output

Fills absent events with a pre-defined value.

```
λ> let s1   = signal [Abst, Abst, Prst 1, Prst 2, Abst, Prst 3]
λ> fill 0 s1
{0,0,1,2,0,3}
```

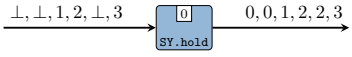


`hold`

`::`	`a`	Value to fill with in case there was no previous value
`->`	`Signal (AbstExt a)`	Input
`->`	`Signal a`	Output

Similar to `fill`, but holds the last non-absent value if there was one. It implements a `state` pattern (see `state22`).

```
λ> let s1   = signal [Abst, Abst, Prst 1, Prst 2, Abst, Prst 3]
λ> hold 0 s1
{0,0,1,2,2,3}
```



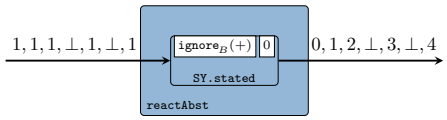
`reactAbst2`

`::`	`(Signal (AbstExt a1) -> Signal (AbstExt a2) -> Signal b)`	process which degrades the absent extension, e.g. holds present values
`->`	`Signal (AbstExt a1)`	
`->`	`Signal (AbstExt a2)`	
`->`	`Signal (AbstExt b)`	absent-extended signal, properly reacting to the inputs

Creates a wrapper enabling a reactive behvior to absent-extended signals for processes which would otherwise degrade the absent-extension (e.g. state machines with `ignore22` behavior).

Constructors: `reactAbst[1-4]`.

```
λ> let s1 = readSignal "{1,1,1,_,1,_,1}" :: Signal (AbstExt Int)
λ> let proc = stated11 (B.ignore11 (+)) 0
λ> proc s1
{0,1,2,3,3,4,4,5}
λ> reactAbst1 proc s1
{0,1,2,⊥,3,⊥,4}
```



4.7.4 Interfaces

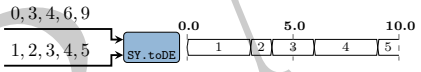
`toDE2`

`::`	`Signal TimeStamp`	SY signal carrying `DE` timestamps
`->`	`Signal a`	first input SY signal
`->`	`Signal b`	second input SY signal
`->`	`(Signal a, Signal b)`	two output `DE` signals

Wraps explicit timestamps to a (set of) `SY` signal(s), rendering the equivalent synchronized `DE` signal(s).

Constructors: `toDE`, `toDE2`, `toDE3`, `toDE4`.

```
λ> let s1 = SY.signal [0,3,4,6,9]
λ> let s2 = SY.signal [1,2,3,4,5]
λ> toDE s1 s2
{ 1 @0s, 2 @3s, 3 @4s, 4 @6s, 5 @9s}
```

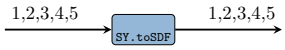


`toSDF2 :: Signal a -> Signal b -> (Signal a, Signal b)`

Transforms a (set of) `SY` signal(s) into the equivalent `SDF` signal(s). The only change is the event consructor. The total order of SY is interpreted as partial order by the next SDF process downstream.

Constructors: `toSDF`, `toSDF2`, `toSDF3`, `toSDF4`.

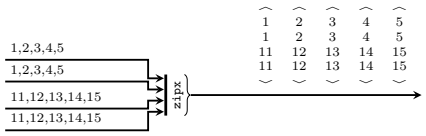
```
λ> let s = SY.signal [1,2,3,4,5]
λ> toSDF s
{1,2,3,4,5}
```



`zipx :: Vector (Signal a) -> Signal (Vector a)`

Synchronizes all the signals contained by a vector and zips them into one signal of vectors. It instantiates the `zipx` skeleton.

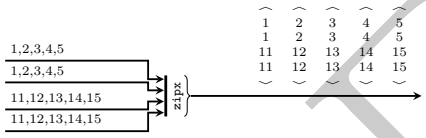
```
λ> let s1 = SY.signal [1,2,3,4,5]
λ> let s2 = SY.signal [11,12,13,14,15]
λ> let v1 = V.vector [s1,s1,s2,s2]
λ> v1
<{1,2,3,4,5},{1,2,3,4,5},{11,12,13,14,15},{11,12,13,14,15}>
λ> zipx v1
{<1,1,11,11>,<2,2,12,12>,<3,3,13,13>,<4,4,14,14>,<5,5,15,15>}
```



`unzipx :: Integer -> Signal (Vector a) -> Vector (Signal a)`

Unzips the vectors carried by a signal into a vector of signals. It instantiates the `unzipx` skeleton. To avoid infinite recurrence, the user needs to provide the length of the output vector.

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = SY.signal [v1,v1,v1,v1,v1]
λ> s1
{<1,2,3,4>,<1,2,3,4>,<1,2,3,4>,<1,2,3,4>,<1,2,3,4>}
λ> unzipx 4 s1
<{1,1,1,1,1},{2,2,2,2,2},{3,3,3,3,3},{4,4,4,4,4}>
```



`unzipx' :: Signal (Vector a) -> Vector (Signal a)`

Same as `unzipx`, but "sniffs" the first event to determine the length of the output vector. Might have unsafe behavior!

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = SY.signal [v1,v1,v1,v1,v1]
λ> s1
{<1,2,3,4>,<1,2,3,4>,<1,2,3,4>,<1,2,3,4>,<1,2,3,4>}
λ> unzipx' s1
<{1,1,1,1,1},{2,2,2,2,2},{3,3,3,3,3},{4,4,4,4,4}>
```

4.8 ForSyDe.Atom.MoC.DE

```
module ForSyDe.Atom.MoC.DE (
    TimeStamp, DE(DE, tag, val), Signal, unit2, infinite, until, signal,
    checkSignal, readSignal, delay, delay', comb22, reconfig22, sync2,
    constant2, generate2, stated22, state22, moore22, mealy22, toSY2,
    toCT2, zipx, unzipx, unzipx', embedSY22
  ) where
```

The `DE` library implements the atoms holding the sematics for the discrete event computation model. It also provides a set of helpers for properly instantiating process network patterns as process constructors.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

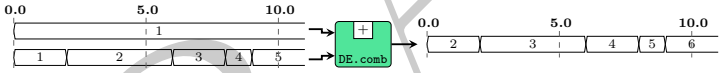
4.8.1 Discrete event (DE)

According to Lee and Sangiovanni-Vincentelli, 1998, "a discrete-event system is a timed system Q where for all $s \in Q$, the tag sytem is order-isomorphic to a subset of the integers. Order-isomorphic means simply that there exists an order-preserving bijection between the events in T and a subset of the integers (or the entire set of integers)."

The discrete event (DE) MoC does suggest the notion of physical time through its tags, also called timestamps. As the definition above implies, an important property of the DE tag system is that between any two timestamps t_u and t_v there is a ***finite*** number of possible timestamps. Based on this we can formulate the folowing specialized definition:

The DE MoC is abstracting the execution semantics of a system where synchronization is *discretized* and *time-aware*, and it is performed whenever a new event occurs.

There are many variants of discrete event simulators, each of them implementing slight variations of the semantics stated in Lee and Sangiovanni-Vincentelli, 1998. The execution model covered by the DE implementation of ForSyDe-atom may be described as a simplified "cycle simulator" with no delta-delay nor superdense time. The signals behave as "latched channels" (similar to an HDL simulator), and processes react instantaneously to any new event. While the simplicity of the execution engine is a desired one, more complex behaviors such as zero-time feedback, non-zero reaction time and communication protocols (e.g. lossy buffers) may be achieved by composing patterns from the `ForSyDe.Atom.MoC` and/or `ForSyDe.Atom.ExB` layers. Nevertheless, the DE behaviors possible within ForSyDe-Atom are included in the class of *conservative simulators* as presented in **Fujimoto00** due to the dataflow nature of the evaluation mechanisms. Below you can see an example of a simple DE process, without behavior extensions:



Below are stated a few particularities of our DE MoC implementation:

1. According to Lee and Sangiovanni-Vincentelli, 1998, our DE MoC is a one-sided system, i.e. time starts from an absolute 0. While negative time cannot be represented, signals can be phase-aligned with the help of the `-&-` atom. All signals need to start from timestamp 0, and events need to be positioned with their tags in strict ascending order. The `checkSignal` utility enforces these rules.

2. tags are explicit and a DE event will construct a type around both a tag and a value. Tags represent the start time of the event, the end time being implicit from the start time of the next event. By doing so, we ensure that the time domain is non-disjoint, i.e. a sub-case of continous time.

3. according to the previous point, events are assumed to persist from their time of arrival until the next event arrives or, if there is no incoming event, until infinity. This default behavior can be be interpret signals as being either persistent channels (e.g. latched wires), or non-blocking buffers of size 1.

4. as a consequence to the previous is that feedback loops will generate an infinite number of events (strictly preceding each other), since a loop updates the value after a certain delay, and any input is assumed to go to infinity. Thus we can now fully jutify the definition of the `delay` pattern as consisting in a *prepend* (i.e. generating the new value) and a *phase shift* (i.e. advancing time with a positive integer). This is done in order to both preserve causality *and* avoid deadlock.

5. due to the reactive and dataflow natures of the execution system, DE processes *are forbidden* to clean up events. Doing so might lead to deadlock wherever any feedback is involved. This means that a new event is created every time a new event arrives, regardless of what value it carries. This means that *all* values are propagated, justifying our system's conservative approach **Fujimoto00** Atoms themselves do not clean signals, but using interfaces that do should be treated with extreme special care, as it is considered unsafe and deadlock-prone.

6. due to the conservative approach, and to the fact that MoC atoms are independent synchronization entities, ForSyDe simulators are completely parallelizable, since processes are self-sufficient and do not depend on a global event queue (as compared to other cycle simulators).

7. any signal from outside needs to be sane (T must be a total order) before being injected into a ForSyDe process network. Helper functions are equipped with sanity checkers. Inside a ForSyDe process network, transformations are rate-monotonic, thus output signals are guaranteed to be sane.

8. since T is a total order, there is no need for an execution context (see section 4.5) and we can ignore the formatting of functions in `ForSyDe.Atom.MoC`, thus we can safely assume:

$$\dot{a} = a$$

`type TimeStamp = DiffTime`
: Alias for the type representing discrete time. It is inherently quantizable, the quantum being a picosecond (10^{-12} seconds), thus it can be considered order-isomorphic with a set of integers, i.e. between any two timestamps there is a finite number of timestamps. Moreover, a timestamp can be easily translated into a rational number representing fractions of a second, so the conversion between timestamps (discrete time) and rationals (analog/continuous time) is straightforward.

  This type is used in the explicit tags of the `DE` MoC (and subsequently the discrete event evaluation engine for simulating the `CT` MoC).

```
data DE a
  =   DE
  >   tag ::  TimeStamp   timestamp
  >   val ::  a           the value
```
: The DE event. It identifies a discrete event signal.

`instance Functor DE`
: Allows for mapping of functions on a DE event.

`instance Applicative DE`
: Allows for lifting functions on a pair of DE events.

`instance MoC DE`
: Implements the execution and synchronization semantics for the DE MoC through its atoms.

`instance Eq a => Eq (DE a)`

`instance Read a => Read (DE a)`
: Reads the string of type `v@t` as an event `DE t v`.

`instance Show a => Show (DE a)`
: Shows the event with tag `t` and value `v` as `v @t`.

`instance Plottable a => Plot (Signal a)`
: `DE` signals.

```
instance type Ret DE b = b
instance type Fun DE a b = a -> b
```

4.8.2 Aliases & utilities

A set of type synonyms and utilities are provided for convenience. The API type signatures will feature these aliases to hide the cumbersome construction of atoms and patters as seen in `ForSyDe.Atom.MoC`.

`type Signal a = Stream (DE a)`
: Type synonym for a SY signal, i.e. "a signal of SY events"

`unit2 :: ((TimeStamp, a1), (TimeStamp, a2))`
`-> (Signal a1, Signal a2)`
: Wraps a (tuple of) pair(s) `(tag, value)` into the equivalent unit signal(s), in this case a signal with one event with the period `tag` carrying `value`.

Helpers: `unit` and `unit[2-4]`.

`infinite :: a -> Signal a`
: Creates an infinite signal.

`until :: TimeStamp -> Signal a -> Signal a`
: Takes the first part of the signal util a given timestamp. The last event of the resulting signal is at the given timestamp and carries the previous value. This utility is useful when plotting a signal, to specify the interval of plotting.

`signal :: [(TimeStamp, a)] -> Signal a`
: Transforms a list of tuples `(tag, value)` into a DE signal. Checks if it is well-formed.

`checkSignal :: Stream (DE a) -> Stream (DE a)`
: Checks if a signal is well-formed or not, according to the DE MoC interpretation in `ForSyDe-Atom`.

`readSignal :: Read a => String -> Signal a`
: Reads a signal from a string and checks if it is well-formed. Like with the `read` function from `Prelude`, you must specify the type of the signal.

```
λ> readSignal "{ 1@0, 2@2, 3@5, 4@7, 5@10 }" :: Signal Int
{ 1 @0s, 2 @2s, 3 @5s, 4 @7s, 5 @10s}
λ> readSignal "{ 1@0, 2@2, 3@5, 4@10, 5@7 }" :: Signal Int
{ 1 @0s, 2 @2s, 3 @5s*** Exception: [MoC.DE] malformed signal
λ> readSignal "{ 1@1, 2@2, 3@5, 4@7, 5@10 }" :: Signal Int
*** Exception: [MoC.DE] signal does not start from global 0
```

4.8.3 DE process constuctors

The DE process constructors are basically specific instantiations of patterns defined in `ForSyDe.Atom.MoC`. Some might also be wrapping functions in an extended behavioural model.

Simple

These are mainly direct instantiations of patterns defined in `ForSyDe.Atom.MoC`, using DE-specific utilities.

`delay`

`::`	`TimeStamp`	time delay
`->`	`a`	initial value
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

The `delay` process "delays" a signal with one event. Instantiates the `delay` pattern.

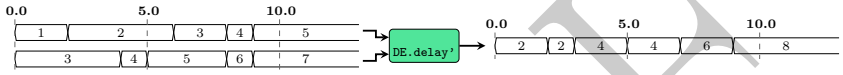
```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> delay 3 0 s
{ 0 @0s, 1 @3s, 2 @5s, 3 @9s, 4 @11s, 5 @12s}
```

`delay'`

`::`	`Signal a`	signal "borrowing" the initial event
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

Similar to the previous, but this is the raw instantiation of the `delay` pattern. It "borrows" the first event from one signal and appends it at the head of another signal.

```
λ> let s1 = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> let s2 = readSignal "{3@0, 4@4, 5@5, 6@8, 7@9}" :: Signal Int
λ> delay' s1 s2
{ 1 @0s, 3 @2s, 4 @6s, 5 @7s, 6 @10s, 7 @11s}
```



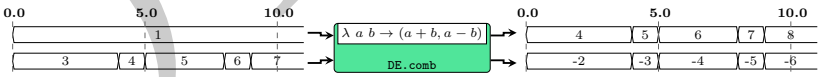
`comb22`

`::`	`(a1 -> a2 -> (b1, b2))`	function on values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	two output signals

`comb` processes map combinatorial functions on signals and take care of synchronization between input signals. It instantiates the `comb` pattern (see `comb22`).

Constructors: `comb[1-4][1-4]`.

```
λ> let s1 = infinite 1
λ> let s2 = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> comb11 (+1) s2
{ 2 @0s, 3 @2s, 4 @6s, 5 @8s, 6 @9s}
λ> comb22 (\a b-> (a+b,a-b)) s1 s2
({ 2 @0s, 3 @2s, 4 @6s, 5 @8s, 6 @9s},{ 0 @0s, -1 @2s, -2 @6s, -3 @8s, -4 @9s})
```



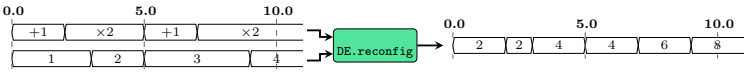
`reconfig22`

`::`	`Signal (a1 -> a2 -> (b1, b2))`	signal carrying functions
`->`	`Signal a1`	first input signal carrying arguments
`->`	`Signal a2`	second input signal carrying arguments
`->`	`(Signal b1, Signal b2)`	two output signals

`reconfig` creates a DE adaptive process where the first signal carries functions and the other carry the arguments. It instantiates the `reconfig` atom pattern (see `reconfig22`).

Constructors: `reconfig[1-4][1-4]`.

```
λ> let sf = signal [(0,(+1)),(2,(*2)),(5,(+1)),(7,(*2))]
λ> let s1 = signal [(0,1),(3,2),(5,3),(9,4)]
λ> reconfig11 sf s1
{ 2 @0s, 2 @2s, 4 @3s, 4 @5s, 6 @7s, 8 @9s}
```



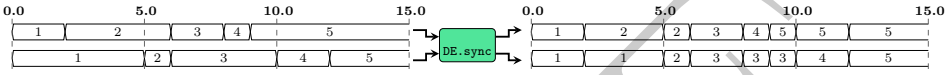
`sync2`

::	`Signal a1`	first input signal
->	`Signal a2`	second input signal
->	`(Signal a1, Signal a2)`	two output signals

`sync` synchronizes multiple signals, so that they have the same set of tags, and consequently, the same number of events. It instantiates the `comb` atom pattern (see `comb22`).

Constructors: `sync[1-4]`

```
λ> let s1 = readSignal "{1@0, 2@2, 3@6, 4@8,  5@9}"  :: Signal Int
λ> let s2 = readSignal "{1@0, 2@5, 3@6, 4@10, 5@12}" :: Signal Int
λ> sync2 s1 s2
({ 1 @0s, 2 @2s, 2 @5s, 3 @6s, 4 @8s, 5 @9s, 5 @10s, 5 @12s},{ 1 @0s, 1 @2s, 2 @5s, 3 @6s, 3
    @8s, 3 @9s, 4 @10s, 5 @12s})
```



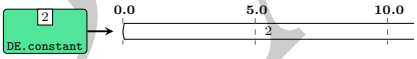
`constant2`

::	`(b1, b2)`	values to be repeated
->	`(Signal b1, Signal b2)`	generated signals

A signal generator which keeps a value constant. As compared with the `SY`, it just constructs an infinite signal with constant value (i.e. a signal with one event starting from time 0).

Constructors: `constant[1-4]`.

```
λ> constant1 2
{ 2 @0s}
```



`generate2`

::	`(b1 -> b2 -> (b1, b2))`	function to generate next value
->	`((TimeStamp, b1), (TimeStamp, b2))`	kernel values tupled with their generation rate.
->	`(Signal b1, Signal b2)`	generated signals

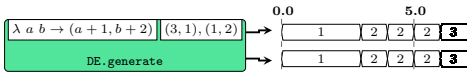
A signal generator based on a function and a kernel value. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `generate[1-4]`.

```
λ> let (s1,s2) = generate2 (\a b -> (a+1,b+2)) ((3,1),(1,2))
λ> takeS 5 s1
{ 1 @0s, 2 @3s, 2 @4s, 2 @5s, 3 @6s}
λ> takeS 7 s2
{ 2 @0s, 4 @1s, 6 @2s, 8 @3s, 10 @4s, 12 @5s, 14 @6s}
```

`stated22`

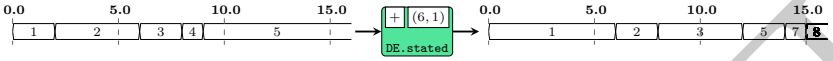
::	`(b1 -> b2 -> a1 -> a2 -> (b1, b2))`	next state function
->	`((TimeStamp, b1), (TimeStamp, b2))`	initial state values tupled with their initial delay
->	`Signal a1`	first input signal
->	`Signal a2`	second input signal
->	`(Signal b1, Signal b2)`	output signals



`stated` is a state machine without an output decoder. It is an instantiation of the `state` MoC constructor (see `stated22`).

Constructors: `stated[1-4][1-4]`.

```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> takeS 7 $ stated11 (+) (6,1) s
{ 1 @0s, 2 @6s, 3 @8s, 5 @12s, 7 @14s, 8 @15s, 10 @18s}
```



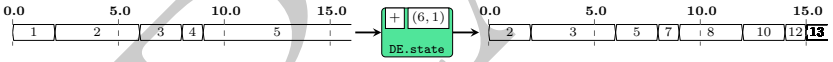
`state22`

`::`	`(b1 -> b2 -> a1 -> a2 -> (b1, b2))`	next state function
`->`	`((TimeStamp, b1), (TimeStamp, b2))`	initial state values tupled with their initial delay
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	output signals

`state` is a state machine without an output decoder, and the state non-transparent. It is an instantiation of the `state` MoC constructor (see `state22`).

Constructors: `state[1-4][1-4]`.

```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> takeS 7 $ state11 (+) (6,1) s
{ 2 @0s, 3 @2s, 5 @6s, 7 @8s, 8 @9s, 10 @12s, 12 @14s}
```



`moore22`

`::`	`(st -> a1 -> a2 -> st)`	next state function
`->`	`(st -> (b1, b2))`	output decoder
`->`	`(TimeStamp, st)`	initial state: tag and value
`->`	`Signal a1`	
`->`	`Signal a2`	
`->`	`(Signal b1, Signal b2)`	

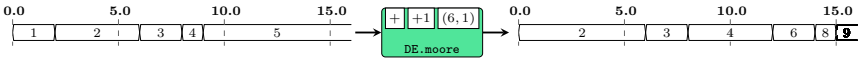
`moore` processes model Moore state machines. It is an instantiation of the `moore` MoC constructor (see `moore22`).

Constructors: `moore[1-4][1-4]`

```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> takeS 7 $ moore11 (+) (+1) (6,1) s
{ 2 @0s, 3 @6s, 4 @8s, 6 @12s, 8 @14s, 9 @15s, 11 @18s}
```

`mealy22`

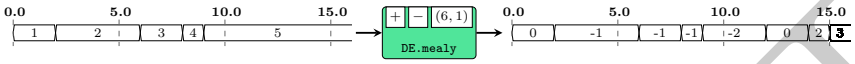
`::`	`(st -> a1 -> a2 -> st)`	next state function
`->`	`(st -> a1 -> a2 -> (b1, b2))`	outpt decoder
`->`	`(TimeStamp, st)`	initial state: tag and value
`->`	`Signal a1`	
`->`	`Signal a2`	
`->`	`(Signal b1, Signal b2)`	



`mealy` processes model Mealy state machines. It is an instantiation of the `mealy` MoC constructor (see `mealy22`).

Constructors: `mealy[1-4][1-4]`

```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: Signal Int
λ> takeS 7 $ mealy11 (+) (-) (6,1) s
{ 0 @0s, -1 @2s, -1 @6s, -1 @8s, -2 @9s, 0 @12s, 2 @14s}
```



Interface processes

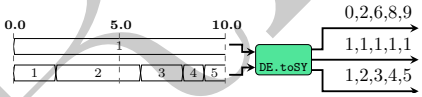
`toSY2`

`::`	`Signal a`	first input DE signal
`->`	`Signal b`	second input DE signal
`->`	`(Signal TimeStamp, Signal a, Signal b)`	signal carrying timestamps tupled with the two output `SY` signals

Synchronizes a (set of) `DE` signal(s) an strips off their explicit tags, outputting the equivalent `SY` signal(s), tupled with an SY signal carrying the timestamps for the synchronization points.

Constructors : `toSY[1-4]`.

```
λ> let s1 = DE.infinite 1
λ> let s2 = DE.readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: DE.Signal Int
λ> toSY2 s1 s2
({0s,2s,6s,8s,9s},{1,1,1,1,1},{1,2,3,4,5})
```



`toCT2`

`::`	`Signal (Time -> a)`	first input DE signal
`->`	`Signal (Time -> b)`	second input DE signal
`->`	`(Signal a, Signal b)`	two output `CT` signals

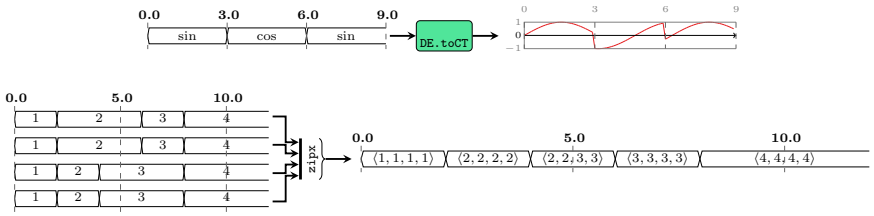
Semantic preserving transformation between a (set of) DE signal(s) and the equivalent CT signals. The `DE` events must carry a function of `Time` which will be lifted by providing it with `CT` implicit time semantics.

Constructors: `toCT[1-4]`.

`zipx :: Vector (Signal a) -> Signal (Vector a)`

Synchronizes all the signals contained by a vector and zips them into one signal of vectors. It instantiates the `zipx` skeleton.

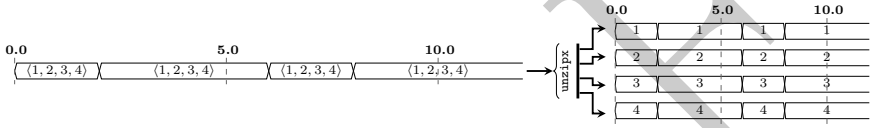
```
λ> let s1 = DE.readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: DE.Signal Int
λ> let s2 = DE.readSignal "{1@0, 2@2, 3@4, 4@8, 5@9}" :: DE.Signal Int
λ> let v1 = V.vector [s1,s1,s2,s2]
λ> v1
<{ 1 @0s, 2 @2s, 3 @6s, 4 @8s, 5 @9s},{ 1 @0s, 2 @2s, 3 @6s, 4 @8s, 5 @9s},{ 1 @0s, 2 @2s, 3
    @4s, 4 @8s, 5 @9s},{ 1 @0s, 2 @2s, 3 @4s, 4 @8s, 5 @9s}>
λ> zipx v1
{ <1,1,1,1> @0s, <2,2,2,2> @2s, <2,2,3,3> @4s, <3,3,3,3> @6s, <4,4,4,4> @8s, <5,5,5,5> @9s}
```



`unzipx :: Integer -> Signal (Vector a) -> Vector (Signal a)`

Unzips the vectors carried by a signal into a vector of signals. It instantiates the `unzipx` skeleton. To avoid infinite recurrence, the user needs to provide the length of the output vector.

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = DE.signal [(0,v1),(2,v1),(6,v1),(8,v1),(9,v1)]
λ> s1
{ <1,2,3,4> @0s, <1,2,3,4> @2s, <1,2,3,4> @6s, <1,2,3,4> @8s, <1,2,3,4> @9s}
λ> unzipx 4 s1
<{ 1 @0s, 1 @2s, 1 @6s, 1 @8s, 1 @9s},{ 2 @0s, 2 @2s, 2 @6s, 2 @8s, 2 @9s},{ 3 @0s, 3 @2s, 3
    @6s, 3 @8s, 3 @9s},{ 4 @0s, 4 @2s, 4 @6s, 4 @8s, 4 @9s}>
```



`unzipx' :: Signal (Vector a) -> Vector (Signal a)`

Same as `unzipx`, but "sniffs" the first event to determine the length of the output vector. Might have unsafe behavior!

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = DE.signal [(0,v1),(2,v1),(6,v1),(8,v1),(9,v1)]
λ> s1
{ <1,2,3,4> @0s, <1,2,3,4> @2s, <1,2,3,4> @6s, <1,2,3,4> @8s, <1,2,3,4> @9s}
λ> unzipx' s1
<{ 1 @0s, 1 @2s, 1 @6s, 1 @8s, 1 @9s},{ 2 @0s, 2 @2s, 2 @6s, 2 @8s, 2 @9s},{ 3 @0s, 3 @2s, 3
    @6s, 3 @8s, 3 @9s},{ 4 @0s, 4 @2s, 4 @6s, 4 @8s, 4 @9s}>
```

Hybrid processes

`embedSY22`

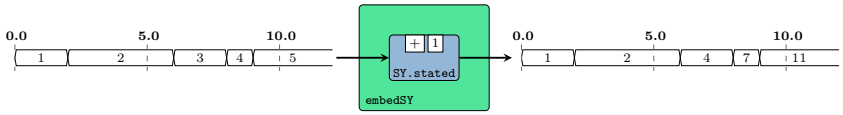
`::`	`(Signal a1 -> Signal a2 -> (Signal b1, Signal b2))`	`SY` process
`->`	`Signal a1`	first input DE signal
`->`	`Signal a2`	second input DE signal
`->`	`(Signal b1, Signal b2)`	two output `DE` signals

Embeds a `SY` process inside a `DE` environment. Internally, it synchronizes the input signals, translates them to SY, feeds them to a SY process and translates the result back to DE using the same input tags. Seen from outside, this process behaves like a DE process with "instantaneous response", even for feedback loops.

Constructors: `embedSY[1-4][1-4]`.

For the following example, see the difference between its output and the one of `stated22`

```
λ> let s = readSignal "{1@0, 2@2, 3@6, 4@8, 5@9}" :: DE.Signal Int
λ> embedSY11 (SY.stated11 (+) 1) s
{ 1 @0s, 2 @2s, 4 @6s, 7 @8s, 11 @9s}
```



4.9 ForSyDe.Atom.MoC.CT

```
module ForSyDe.Atom.MoC.CT (
    TimeStamp, Time, CT(CT, tag, phase, func), Signal, unit2, infinite,
    signal, checkSignal, delay, delay', comb22, reconfig22, constant2,
    infinite2, generate2, stated22, state22, moore22, mealy22, toDE1,
    sampDE2, zipx, unzipx, unzipx'
  ) where
```

The CT library implements the atoms holding the sematics for the continuous time computation model. It also provides a set of helpers for properly instantiating process network patterns as process constructors.

For working with time or timestamps please check the utilities provided by the `ForSyDe.Atom.MoC.Time` and `ForSyDe.Atom.MoC.TimeStamp` modules.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

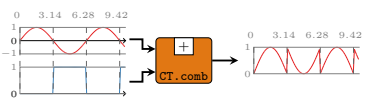
4.9.1 Continuous time (CT) event

According to Lee and Sangiovanni-Vincentelli, 1998, "[regarding metric time] at a minimum, T is an Abelian group, in addition to being totally ordered. A consequence is that t_2 - t_1 is itself a tag $\forall\ t_1,\ t_2 \in T$. In a slightly more elaborate model of computation, T has a metric. (...) A continuous-time system is a metric timed system Q where T is a continuum (a closed connected set)."

The continuous time (CT) MoC defines the closest behavior to what we could call "physical time", where signals cover the full span of a simulation as "functions of time" rather than "values". As such, we can state:

The CT MoC is abstracting the execution semantics and describes a system where computation is performed continuously over a (possibly infinite) span of time.

Below is an illustration of the behavior in time of the input and the output signals of a CT process:



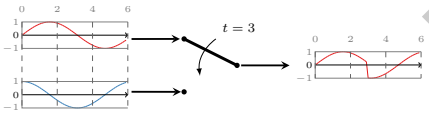
Our CT MoC is implemented as an enhanced version of DE with respect to the **CT MoC** definition, in the sense that:

1. tags t are also represented with `TimeStamps`, thus we can say that changes in a CT signal happen at discrete times (see below).

2. values are represented as functions over a continuous span of time *f(t)* rather than just a value or a series of values. The time domain is represented with rational numbers which, as compared to floating point numbers, do not suffer from inherent quantisation, being able to model true continuity, i.e. between any two arbitrary points in time there is an infinite amount of intermediate moments.

3. The event constructor has also a *phase* component ϕ, which is taken into consideration only when evaluating the event function, i.e. $f\ (t + \phi)$. This enables the modeling of "phase dispacements" of delay lines without altering the function itself (and thus increasing the complexity of the un-evaluated function graph). The phase needs to be reset during event synchronization.

These seemingly minor changes have deep implications in the expressiveness of a FoSyDe CT system and how we interpret it. Capturing the particularities of this MoC, we can formulate the following properties:



1. `CT` signals, due to their formation as streams of tagged events, represent *discrete* changes in a continuous function over time (e.g. analog signal). While the functions carried by events are infinite (have always happened and will always happen), being carried by events in a tag system suggests that changes occur at discrete times. A CT signal can be represented by the analog circuit above, where the inputs are continuous signals, but the switch is discrete. Like in the `DE` MoC, the absolute time 0 represent the time when the system started to be observed.

2. the previous property is also proven by the fact that the evaluation engine of ForSyDe-Atom is inherently discrete, i.e. evaluation is performed whenever a new event occurs, in a dataflow manner. Allowing infinitely small distances between tags would hinder the advancement of simulation time.

3. events carry *functions* and not *values*. In a lazy evaluation system like Haskell's, functions are kept symbolic until evaluation. This means that in a CT system computations are propagated as function graphs until a result is needed, e.g. a signal is plotted for arbitrary positions in time. This way intermediate quantization errors are eliminated, and the cost of higher plot resolution is the cost of evaluating the final results only.

4. needless to say, for each $t \in T$, a signal is able to return (e.g. plot) the exact value v for that particular t.

5. since itself the `CT` MoC is just an enhanced `DE` system, all atom evaluation properties are inherited from it: feedback loops need to advance time, atoms are forbidden to clean signals, and the conservative approach makes it ideal for parallel/distributed simulation.

6. since T is a total order, there is no need for an execution context (see section 4.5) and we can ignore the formatting of functions in `ForSyDe.Atom.MoC`, thus we can safely assume:

$$\dot{a} = a$$

`type TimeStamp = DiffTime`

Alias for the type representing discrete time. It is inherently quantizable, the quantum being a picosecond (10^{-12} seconds), thus it can be considered order-isomorphic with a set of integers, i.e. between any two timestamps there is a finite number of timestamps. Moreover, a timestamp can be easily translated into a rational number representing fractions of a second, so the conversion between timestamps (discrete time) and rationals (analog/continuous time) is straightforward.

This type is used in the explicit tags of the `DE` MoC (and subsequently the discrete event evaluation engine for simulating the `CT` MoC).

`type Time = Rational`

Type alias for the type to represent metric (continuous) time. Underneath we use `Rational` that is able to represent any t between $t_1 < t_2 \in T$.

```
data CT a
    =  CT
    >  tag ::  TimeStamp     start time of event
    >  phase ::  Time        phase. Models function delays
    >  func ::  Time -> a    function of time
```

The CT type, identifying a continuous time event and implementing an instance of the `MoC` class.

`instance Functor CT`

Allows for mapping of functions on a CT event.

`instance Applicative CT`

Allows for lifting functions on a pair of CT events.

`instance MoC CT`

Implenents the execution and synchronization semantics for the CT MoC through its atoms.

`instance Show a => Show (CT a)`

A non-ideal instance meant for debug purpose only. For each event it evaluates the function at the tag time *only*!

`instance Plottable a => Plot (Signal a)`

`CT` signals.

```
instance type Ret CT b = b
instance type Fun CT a b = a -> b
```

4.9.2 Aliases & utilities

A set of type synonyms and utilities are provided for convenience. The API type signatures will feature these aliases to hide the cumbersome construction of atoms and patters as seen in `ForSyDe.Atom.MoC`.

`type Signal a = Stream (CT a)`

Type synonym for a CT signal, i.e. "a signal of CT events"

```
unit2 :: ((TimeStamp, Time -> a1), (TimeStamp, Time -> a2))
      -> (Signal a1, Signal a2)
```

Wraps a (tuple of) pair(s) `(time, function)` into the equivalent unit signal(s), i.e. signal(s) with one event with the period `time` carrying `function`.

Helpers: `unit` and `unit[2-4]`.

`infinite :: (Time -> a) -> Signal a`
: Creates an infinite signal.

`signal :: [(TimeStamp, Time -> a)] -> Signal a`
: Transforms a list of tuples such as the ones taken by `event` into a CT signal

`checkSignal :: Stream (CT a) -> Stream (CT a)`
: Checks if a signal is well-formed or not, according to the CT MoC interpretation in `ForSyDe-Atom`.

4.9.3 `CT` process constuctors

The CT process constructors are basically specific instantiations of patterns defined in `ForSyDe.Atom.MoC`. Some might also be wrapping functions in an extended behavioural model.

In the examples below we have imported and instantiated the functions such as `e' pi'`, `sin'` and `cos'` from the collection of utilities in `ForSyDe.Atom.MoC.Time` and `ForSyDe.Atom.MoC.TimeStamp`. Also, for the sake of documentation the interactive examples are only dumping the CT signals in data files using the `dumpDat` utility defined in `ForSyDe.Atom.Utility.Plot`, according to the custom `cfg` structure. These files can be further plotted by any tool of choice, or using the plotting utilities provided in the `ForSyDe.Atom.Utility.Plot` module.

```
import ForSyDe.Atom.Utility.Plot
import ForSyDe.Atom.MoC.Time as Time
import ForSyDe.Atom.MoC.TimeStamp as TimeStamp
let pi'  = TimeStamp.pi
let exp' = Time.exp
let sin' = Time.sin
let cos' = Time.cos
let cfg  = defaultCfg {xmax=10, rate=0.1}
```

Simple

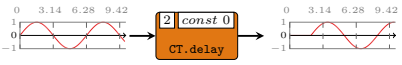
These are mainly direct instantiations of patterns defined in `ForSyDe.Atom.MoC`, using DE-specific utilities.

`delay`

`::`	`TimeStamp`	time delay
`->`	`(Time -> a)`	initial value
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

The `delay` process "delays" a signal with one event. Instantiates the `delay` pattern. In the CT MoC, this process can be interpreted as an ideal "delay line".

```
λ> let s  = infinite (sin')
λ> let s' = delay 2 (\_ -> 0) s
λ> dumpDat $ prepare cfg {labels=["delay"]} $ s'
Dumped delay in ./fig
["./fig/delay.dat"]
```

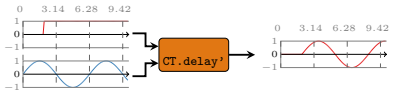


`delay'`

`::`	`Signal a`	signal "borrowing" the initial event
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

Similar to the previous, but this is the raw instantiation of the `delay` pattern. It "borrows" the first event from one signal and appends it at the head of another signal.

```
λ> let s  = infinite (sin')
λ> let s' = signal [(0, \_ -> 0), (2, \_ -> 1)]
λ> dumpDat $ prepare cfg {labels=["delayp"]} $ delay' s' s
Dumped delayp in ./fig
["./fig/delayp.dat"]
```



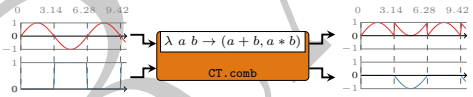
`comb22`

`::`	`(a1 -> a2 -> (b1, b2))`	function on values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	two output signals

`comb` processes map combinatorial functions on signals and take care of synchronization between input signals. It instantiates the `comb` pattern (see `comb22`).

Constructors: `comb[1-4][1-4]`.

```
λ> let s1 = infinite (sin')
λ> let s2 = signal [(0,\_->0),(pi',\_->1),(2*pi',\_->0),(3*pi',\_->1)]
λ> let o1 = comb11 (+1) s2
λ> let (o2_1, o2_2) = comb22 (\a b-> (a+b,a*b)) s1 s2
λ> dumpDat $ prepare cfg {labels=["comb11"]} o1
Dumped comb11 in ./fig
["./fig/comb11.dat"]
λ> dumpDat $ prepareL cfg {labels=["comb22-1","comb22-2"]} [o2_1, o2_2]
Dumped comb22-1, comb22-2 in ./fig
["./fig/comb22-1.dat","./fig/comb22-2.dat"]
```



`reconfig22`

`::`	`Signal (a1 -> a2 -> (b1, b2))`	signal carrying functions
`->`	`Signal a1`	first input signal carrying arguments
`->`	`Signal a2`	second input signal carrying arguments
`->`	`(Signal b1, Signal b2)`	two output signals

`reconfig` creates a CT adaptive process where the first signal carries functions and the other carry the arguments. It instantiates the `reconfig` atom pattern (see `reconfig22`).

Constructors: `reconfig[1-4][1-4]`.

```
λ> let s1 = infinite (sin')
λ> let sf = signal [(0,\_->(*0)),(pi',\_->(+1)),(2*pi',\_->(*0)),(3*pi',\_->(+1))]
λ> dumpDat $ prepare cfg {labels=["reconfig"]} $ reconfig11 sf s1
Dumped reconfig in ./fig
["./fig/reconfig.dat"]
```

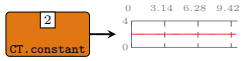
`constant2`

`::`	`(b1, b2)`	values to be repeated
`->`	`(Signal b1, Signal b2)`	generated signals

A generator for a constant signal. As compared with the `constant2`, it just constructs an infinite signal with constant value (i.e. a signal with one event starting from time 0).

Constructors: `constant[1-4]`.

```
λ> dumpDat $ prepare cfg {labels=["constant"]} $ constant1 2
Dumped constant in ./fig
["./fig/constant.dat"]
```



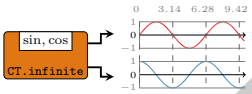
`infinite2`

::	`(Time -> b1, Time -> b2)`	values to be repeated
->	`(Signal b1, Signal b2)`	generated signals

A generator for an infinite signal. Similar to `constant2`.

Constructors: `infinite[1-4]`.

```
λ> let (inf1, inf2) = infinite2 (sin', cos')
λ> dumpDat $ prepareL cfg {labels=["infinite2-1", "infinite2-2"]} [inf1, inf2]
Dumped infinite2-1, infinite2-2 in ./fig
["./fig/infinite2-1.dat","./fig/infinite2-2.dat"]
```



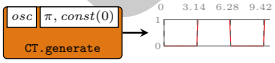
`generate2`

::	`(b1 -> b2 -> (b1, b2))`	function to generate next value
->	`((TimeStamp, Time -> b1), (TimeStamp, Time -> b2))`	kernel values tupled with their generation rate.
->	`(Signal b1, Signal b2)`	generated signals

A signal generator based on a function and a kernel value. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `generate[1-4]`.

```
λ> let { osc 0 = 1 ; osc 1 = 0 }
λ> dumpDat $ prepare cfg {labels=["generate"]} $ generate1 osc (pi', \_->0)
Dumped generate in ./fig
["./fig/generate.dat"]
```

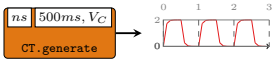


Another example simulating an RC oscillator:

```
λ> let vs = 2                                  -- Vs : supply voltage
λ> let r  = 100                                -- R : resistance
λ> let c  = 0.0005                             -- C : capacitance
λ> let vc t = vs * (1 - exp' (-t / (r * c))) -- Vc(t) : voltage charging through capacitor
λ> let ns v = vs + (-1 * v)                    -- next state : charging / discharging
λ> let rcOsc = generate1 ns (milisec 500, vc)
λ> dumpDat $ prepare cfg {labels=["rcosc"]} $ rcOsc
Dumped rcosc in ./fig
["./fig/rcosc.dat"]
```

`stated22`

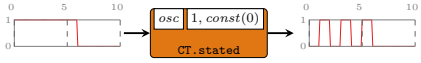
::	`(b1 -> b2 -> a1 -> a2 -> (b1, b2))`	next state function
->	`((TimeStamp, Time -> b1), (TimeStamp, Time -> b2))`	initial state values tupled with their initial delay
->	`Signal a1`	first input signal
->	`Signal a2`	second input signal
->	`(Signal b1, Signal b2)`	output signals



`stated` is a state machine without an output decoder. It is an instantiation of the `state` MoC constructor (see `stated22`).

Constructors: `stated[1-4][1-4]`.

```
λ> let { osc 0 a = a; osc _ a = 0 }
λ> let s1 = signal [(0,\_->1), (6,\_->0)]
λ> dumpDat $ prepare cfg {labels=["stated"]} $ stated11 osc (1,\_->0) s1
Dumped stated in ./fig
["./fig/stated.dat"]
```



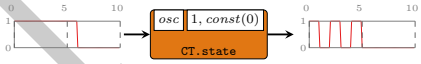
`state22`

`::`	`(b1 -> b2 -> a1 -> a2 -> (b1, b2))`	next state function
`->`	`((TimeStamp, Time -> b1), (TimeStamp, Time -> b2))`	initial state values tupled with their initial delay
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	output signals

`state` is a state machine without an output decoder, and the state non-transparent. It is an instantiation of the `state` MoC constructor (see `state22`).

Constructors: `state[1-4][1-4]`.

```
λ> let { osc 0 a = a; osc _ a = 0 }
λ> let s1 = signal [(0,\_->1), (6,\_->0)]
λ> dumpDat $ prepare cfg {labels=["state"]} $ state11 osc (1,\_->0) s1
Dumped state in ./fig
["./fig/state.dat"]
```



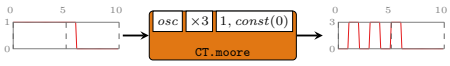
`moore22`

`::`	`(st -> a1 -> a2 -> st)`	next state function
`->`	`(st -> (b1, b2))`	output decoder
`->`	`(TimeStamp, Time -> st)`	initial state: tag and value
`->`	`Signal a1`	
`->`	`Signal a2`	
`->`	`(Signal b1, Signal b2)`	

`moore` processes model Moore state machines. It is an instantiation of the `moore` MoC constructor (see `moore22`).

Constructors: `moore[1-4][1-4]`.

```
λ> let { osc 0 a = a; osc _ a = 0 }
λ> let s1 = signal [(0,\_->1), (6,\_->0)]
λ> dumpDat $ prepare cfg {labels=["moore"]} $ moore11 osc (*3) (1,\_->0) s1
Dumped moore in ./fig
["./fig/moore.dat"]
```



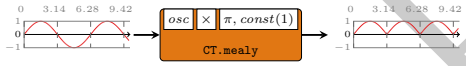
`mealy22`

```
::  (st -> a1 -> a2 -> st)          next state function
->  (st -> a1 -> a2 -> (b1, b2))    outpt decoder
->  (TimeStamp, Time -> st)         initial state: tag and value
->  Signal a1
->  Signal a2
->  (Signal b1, Signal b2)
```

`mealy` processes model Mealy state machines. It is an instantiation of the `mealy` MoC constructor (see `mealy22`).

Constructors: `mealy[1-4][1-4]`.

```
λ> let { osc (-1) _ = 1; osc 1 _ = (-1) }
λ> let s1 = infinite sin'
λ> dumpDat $ prepare cfg {labels=["mealy"]} $ mealy11 osc (*) (pi',\_->1) s1
Dumped mealy in ./fig
["./fig/mealy.dat"]
```

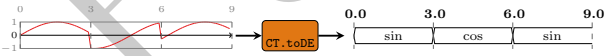


Interfaces

`toDE1 :: Signal a -> Signal (Time -> a)`

Translates a (set of) `CT` signal(s) into `DE` semantics without loss of information. In `DE`, the abstract function of time inferred by the `CT` event loses its abstraction and it is "dropped" to explicit form, under a lower layer. In other words the implicit time semantics are lost, the carried value simply becoming an ordinary function.

Constructors: `toDE[1-4]`.



`sampDE2`

```
::  Signal t              DE timestamp carrier
->  Signal a              CT input
->  Signal b              CT input
->  (Signal a, Signal b)  DE outputs
```

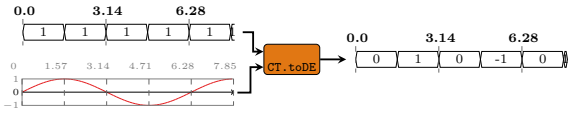
Synchronizes a (set of) `CT` signal(s) with a `DE` carrier which holds the timestamps at which the CT signal must be sampled, and outputs the respective (set of) `DE` signal(s).

Constructors: `sampDE[1-4]`.

```
λ> let s = CT.infinite (fromRational . sin')
λ> let c = DE.generate1 id (pi'/2, 1)
λ> takeS 6 $ sampDE1 c s
{ 0.0 @0s, 1.0 @1.570796326794s, 1.793238520564752e-12 @3.141592653588s, -1.0 @4.712388980382
    s, 0.0 @6.283185307176s, 1.0 @7.85398163397s}
```

`zipx :: Vector (Signal a) -> Signal (Vector a)`

Synchronizes all the signals contained by a vector and zips them into one signal of vectors. It instantiates the `zipx` skeleton.




```
λ> let s1 = CT.signal [(0,const 1), (2,const 2), (6,const 3)]
λ> let s2 = CT.signal [(0,const 1), (2,const 2), (4,const 3)]
λ> let v1 = V.vector [s1,s1,s2,s2]
λ> zipx v1
{ <1,1,1,1> @0s, <2,2,2,2> @2s, <2,2,3,3> @4s, <3,3,3,3> @6s}
```

See `zipx` from the `ForSyDe.Atom.MoC.DE` library for a comprehensive visual example.

`unzipx :: Integer -> Signal (Vector a) -> Vector (Signal a)`

Unzips the vectors carried by a signal into a vector of signals. It instantiates the `unzipx` skeleton. To avoid infinite recurrence, the user needs to provict the length of the output vector.

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = CT.signal [(0,const v1),(2,const v1),(6,const v1)]
λ> unzipx 4 s1
<{ 4 @0s, 4 @2s, 4 @6s},{ 3 @0s, 3 @2s, 3 @6s},{ 2 @0s, 2 @2s, 2 @6s},{ 1 @0s, 1 @2s, 1 @6s}>
```

See `unzipx` from the `ForSyDe.Atom.MoC.DE` library for a comprehensive visual example.

`unzipx' :: Signal (Vector a) -> Vector (Signal a)`

Same as `unzipx`, but "sniffs" the first event to determine the length of the output vector. Might have unsafe behavior!

```
λ> let v1 = V.vector [1,2,3,4]
λ> let s1 = CT.signal [(0,const v1),(2,const v1),(6,const v1)]
λ> unzipx' s1
<{ 4 @0s, 4 @2s, 4 @6s},{ 3 @0s, 3 @2s, 3 @6s},{ 2 @0s, 2 @2s, 2 @6s},{ 1 @0s, 1 @2s, 1 @6s}>
```

4.10 ForSyDe.Atom.MoC.SDF

```
module ForSyDe.Atom.MoC.SDF (
    SDF(SDF, val), Signal, Prod, Cons, signal, readSignal, delay,
    delay', comb22, reconfig22, constant2, generate2, stated22, state22,
    moore22, mealy22, toSY2, zipx, unzipx
  ) where
```

The `SDF` library implements the atoms holding the sematics for the synchronous data flow computation model. It also provides a set of helpers for properly instantiating process network patterns as process constructors.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

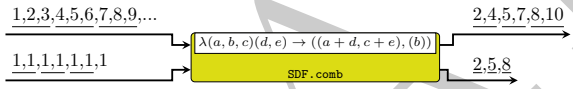
4.10.1 Synchronous data flow (SDF) event

The synchronous data flow (SDF) MoC is the first untimed MoC implemented by the forsyde-atom framework. On untimed MoCs, Lee and Sangiovanni-Vincentelli, 1998 states that: "when tags are partially ordered rather than totally ordered, we say that the system is untimed. Untimed systems cannot have the same notion of causality as timed systems [see SY]. (...) Processes defined in terms of constraints on the tags in the signals (...) have a *consistent cut* rather than *simultaneity*." Regarding SDF, it states that "is a special case of Kahn process networks Kahn and MacQueen, 1976. A dataflow process is a Kahn process that is also sequential, where the events on the self-loop signal denote the firings of the dataflow actor. The firing rules of a dataflow actor are partial ordering constraints between these events and events on the inputs. (...) Produced/consumed events are defined in terms of relations with the events in the firing signal. It results that for the same firing i, $e_i < e_o$, as an intuitive sort of causality constraint."

Based on the above insights, we can formulate a simplified definition of the forsyde-atom interpretation of SDF:

The SDF MoC is abstracting the execution semantics of a system where computation is performed according to firing rules where the production and the consumption rates are fixed.

Below is a *possible* behavior in time of the input and the output signals of a SDF process. Events sharing the same partial ordering in relation to one firing are overlined:



Implementing the SDF tag system implied a series of engineering decisions which lead to the following particularities:

1. signals represent FIFO channels, and tags are implicit from their position in the Stream structure. Internally, SDF signals have exactly the same structure as SY signals, whereas the partial ordering is imposed by the processes alone.
2. the SDF event constructor wraps only a value.
3. being an *untimed MoC*, the order between events is partial to the firings of processes. An SDF atom will fire only when there are enough events to trigger its inputs. Once a firing occurs, it will take care of partitioning the input or output signals.
4. SDF atoms *do* require a context: the consumption c and production p rates. Also, the functions passed as arguments reflect the fact that multiple events are handled during a firing.
5. the previous statement can be synthesized into the following execution context (see section 4.5), which also justifies the SDF implementation of Fun and for Ret:

$$\dot{a} = a^n, \text{ where } n \in \mathbb{N}$$

```
newtype SDF a
    =  SDF
    >  val ::  a
```

The CT type, identifying a discrete time event and implementing an instance of the MoC class. A discrete event explicitates its tag which is represented as an integer.

`instance Functor SDF`
: Allows for mapping of functions on a SDF event.

`instance Applicative SDF`
: Allows for lifting functions on a pair of SDF events.

```
instance Foldable SDF
instance Traversable SDF
```

`instance MoC SDF`
: Implenents the SDF semantics for the MoC atoms

`instance Read a => Read (SDF a)`
: Reads the value wrapped

`instance Show a => Show (SDF a)`
: Shows the value wrapped

`instance Plottable a => Plot (Signal a)`
: `SDF` signals.

```
instance type Ret SDF a = (Prod, [a])
instance type Fun SDF a b = (Cons, [a] -> b)
```

4.10.2 Aliases & utilities

A set of type synonyms and utilities are provided for convenience. The API type signatures will feature these aliases to hide the cumbersome construction of atoms and patters as seen in `ForSyDe.Atom.MoC`.

`type Signal a = Stream (SDF a)`
: Type synonym for a SY signal, i.e. "a signal of SY events"

`type Prod = Int`
: Type synonym for consumption rate

`type Cons = Int`
: Type synonym for production rate

`signal :: [a] -> Signal a`
: Transforms a list of values into a SDF signal with only one partition, i.e. all events share the same (initial) tag.

`readSignal :: Read a => String -> Signal a`
: Reads a signal from a string. Like with the `read` function from `Prelude`, you must specify the tipe of the signal.

```
λ> readSignal "{1,2,3,4,5}" :: Signal Int
{1,2,3,4,5}
```

These SY process constructors are basically specific instantiations of the patterns of atoms defined in `ForSyDe.Atom.MoC`. Some are also wrapping functions in an extended behavioural model.

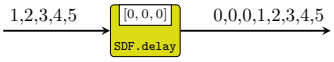
Simple

`delay`

`::`	`[a]`	list of initial values
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

The `delay` process "delays" a signal with initial events built from a list. It is an instantiation of the `delay` constructor.

```
λ> let s = signal [1,2,3,4,5]
λ> delay [0,0,0] s
{0,0,0,1,2,3,4,5}
```



`delay'`

`::`	`Signal a`	signal containing the initial tokens
`->`	`Signal a`	input signal
`->`	`Signal a`	output signal

Similar to the previous, but this is the raw instantiation of the `delay` pattern. It appends the contents of one signal at the head of another signal.

```
λ> let s1 = signal [0,0,0]
λ> let s2 = signal [1,2,3,4,5]
λ> delay' s1 s2
{0,0,0,1,2,3,4,5}
```

`comb22`

`::`	`((Cons, Cons), (Prod, Prod), [a1] -> [a2] -> ([b1], [b2]))`	function on lists of values, tupled with consumption / production rates
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	two output signals

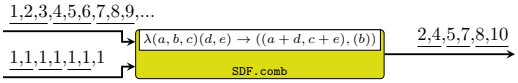
`comb` processes map combinatorial functions on signals and take care of synchronization between input signals. It instantiates the `comb` atom pattern (see `comb22`).

Constructors: `comb[1-4][1-4]`.

```
λ> let s1 = signal [1..]
λ> let s2 = signal [1,1,1,1,1,1,1]
λ> let f [a,b,c] [d,e] = [a+d, c+e]
λ> comb21 ((3,2),2,f) s1 s2
{2,4,5,7,8,10}
```

Incorrect usage (not covered by `doctest`):

```
λ> comb21 ((3,2),3,f) s1 s2
*** Exception: [MoC.SDF] Wrong production
```



`reconfig22`

`::`	`((Cons, Cons), (Prod, Prod))`	
`->`	`Signal ([a1] -> [a2] -> ([b1], [b2]))`	function on lists of values, tupled with consumption / production rates
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	two output signals

`reconfig` creates an SDF adaptive process where the first signal carries functions and the other carry the arguments. It instantiates the `reconfig` atom pattern (see `reconfig22`). According to our SDF definition, the production and consumption rates need to be fixed, so they are passed as parameters to the constructor, whereas the first signal carries adaptive functions only. For the adaptive signal it only makes sense that the consumption rate is always 1.

Constructors: `reconfig[1-4][1-4]`.

```
λ> let f1 a = [sum a]
λ> let f2 a = [maximum a]
λ> let sf = signal [f1,f2,f1,f2,f1,f2,f1]
λ> let s1 = signal [1..]
λ> reconfig11 (4,1) sf s1
{10,8,42,16,74,24,106}
```

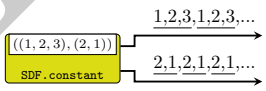
`constant2`

`::`	`([b1], [b2])`	values to be repeated
`->`	`(Signal b1, Signal b2)`	generated signals

A signal generator which repeats the initial tokens indefinitely. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `constant[1-4]`.

```
λ> let (s1, s2) = constant2 ([1,2,3],[2,1])
λ> takeS 7 s1
{1,2,3,1,2,3,1}
λ> takeS 5 s2
{2,1,2,1,2}
```



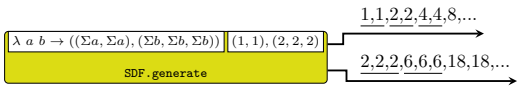
`generate2`

`::`	`((Cons, Cons), (Prod, Prod), [b1] -> [b2] -> ([b1], [b2]))`	function to generate next value, tupled with consumption / production rates
`->`	`([b1], [b2])`	values of initial tokens
`->`	`(Signal b1, Signal b2)`	generated signals

A signal generator based on a function and a kernel value. It is actually an instantiation of the `stated0X` constructor (check `stated22`).

Constructors: `generate[1-4]`.

```
λ> let f a b = ([sum a, sum a],[sum b, sum b, sum b])
λ> let (s1,s2) = generate2 ((2,3),(2,3),f) ([1,1],[2,2,2])
λ> takeS 7 s1
{1,1,2,2,4,4,8}
λ> takeS 8 s2
{2,2,2,6,6,6,18,18}
```



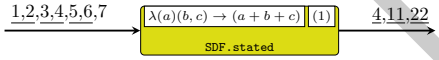
`stated22`

`::`	`((Cons, Cons, Cons, Cons), (Prod, Prod), [b1] -> [b2] -> [a1] -> [a2] -> ([b1], [b2]))`	next state function, tupled with consumption / production rates
`->`	`([b1], [b2])`	initial state partitions of values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	output signals

`stated` is a state machine without an output decoder. It is an instantiation of the `state` MoC constructor (see `stated22`).

Constructors: `stated[1-4][1-4]`.

```
λ> let f [a] [b,c] = [a+b+c]
λ> let s = signal [1,2,3,4,5,6,7]
λ> stated11 ((1,2),1,f) [1] s
{1,4,11,22}
```



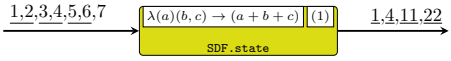
`state22`

`::`	`((Cons, Cons, Cons, Cons), (Prod, Prod), [b1] -> [b2] -> [a1] -> [a2] -> ([b1], [b2]))`	next state function, tupled with consumption / production rates
`->`	`([b1], [b2])`	initial partitions of values
`->`	`Signal a1`	first input signal
`->`	`Signal a2`	second input signal
`->`	`(Signal b1, Signal b2)`	output signals

`state` is a state machine without an output decoder. It is an instantiation of the `stated` MoC constructor (see `state22`).

Constructors: `state[1-4][1-4]`.

```
λ> let f [a] [b,c] = [a+b+c]
λ> let s = signal [1,2,3,4,5,6,7]
λ> state11 ((1,2),1,f) [1] s
{4,11,22}
```



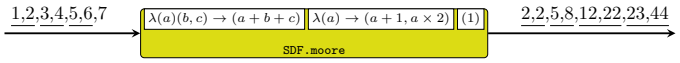
`moore22`

`::`	`((Cons, Cons, Cons), Prod, [st] -> [a1] -> [a2] -> [st])`	next state function, tupled with consumption / production rates
`->`	`(Cons, (Prod, Prod), [st] -> ([b1], [b2]))`	output decoder, tupled with consumption / production rates
`->`	`[st]`	initial state values
`->`	`Signal a1`	
`->`	`Signal a2`	
`->`	`(Signal b1, Signal b2)`	

`moore` processes model Moore state machines. It is an instantiation of the `moore` MoC constructor (see `moore22`).

Constructors: `moore[1-4][1-4]`.

```
λ> let ns [a] [b,c] = [a+b+c]
λ> let od [a]       = [a+1,a*2]
λ> let s = signal [1,2,3,4,5,6,7]
λ> moore11 ((1,2),1,ns) (1,2,od) [1] s
{2,2,5,8,12,22,23,44}
```



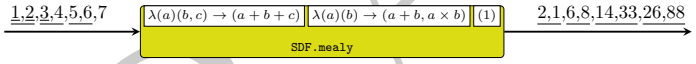
`mealy22`

`::`	`((Cons, Cons, Cons), Prod, [st] -> [a1] -> [a2] -> [st])`	next state function, tupled with consumption / production rates
`->`	`((Cons, Cons, Cons), (Prod, Prod), [st] -> [a1] -> [a2] -> ([b1], [b2]))`	outpt decoder, tupled with consumption / production rates
`->`	`[st]`	initial state values
`->`	`Signal a1`	
`->`	`Signal a2`	
`->`	`(Signal b1, Signal b2)`	

`mealy` processes model Mealy state machines. It is an instantiation of the `mealy` MoC constructor (see `mealy22`).

Constructors: `mealy[1-4][1-4]`.

```
λ> let ns [a] [b,c] = [a+b+c]
λ> let od [a] [b]   = [a+b,a*b]
λ> let s = signal [1,2,3,4,5,6,7]
λ> mealy11 ((1,2),1,ns) ((1,1),2,od) [1] s
{2,1,6,8,14,33,26,88}
```



Interfaces

`toSY2 :: Signal a -> Signal b -> (Signal a, Signal b)`

Transforms a (set of) `SDF` signal(s) into the equivalent `SY` signal(s). The only change is the event consructor. The partial order of DE is interpreted as SY's total order, based on the positioning of events in the signals (e.g. FIFO buffers) at that moment.

Constructors: `toSY[1-4]`.

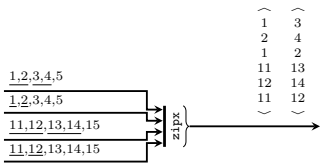
```
λ> let s = SDF.signal [1,2,3,4,5]
λ> toSY s
{1,2,3,4,5}
```

`zipx`

`::`	`Vector Cons`	consumption rates
`->`	`Vector (Signal a)`	vector of signals
`->`	`Signal (Vector a)`	signal of vectors

Consumes tokens from a vector of signals and merges them into a signal of vectors, with a production rate of 1. It instantiates the `zipx` skeleton.

```
λ> let s1 = SDF.signal [1,2,3,4,5]
λ> let s2 = SDF.signal [11,12,13,14,15]
λ> let v1 = V.vector [s1,s1,s2,s2]
λ> let r  = V.vector [2,1,2,1]
λ> v1
<{1,2,3,4,5},{1,2,3,4,5},{11,12,13,14,15},{11,12,13,14,15}>
λ> zipx r v1
{<1,2,1,11,12,11>,<3,4,2,13,14,12>}
```



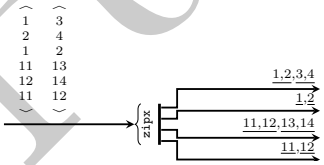
`unzipx`

`::`	`Vector Prod`	production rates (in reverse order)
`->`	`Signal (Vector a)`	signal of vectors
`->`	`Vector (Signal a)`	vector of signals

Consumes the vectors carried by a signal with a rate of 1, and unzips them into a vector of signals based on the user provided rates. It instantiates the `unzipx` skeleton.

OBS: due to the `recur` pattern contained by `unzipx`, the vector of production rates needs to be provided in reverse order (see `ForSyDe.Atom.Skeleton.Vector`).

```
λ> let s1 = SDF.signal [1,2,3,4,5]
λ> let s2 = SDF.signal [11,12,13,14,15]
λ> let v1 = V.vector [s1,s1,s2,s2]
λ> let r  = V.vector [2,1,2,1]
λ> let sz = zipx r v1
λ> v1
<{1,2,3,4,5},{1,2,3,4,5},{11,12,13,14,15},{11,12,13,14,15}>
λ> sz
{<1,2,1,11,12,11>,<3,4,2,13,14,12>}
λ> unzipx (V.reverse r) sz
<{1,2,3,4},{1,2},{11,12,13,14},{11,12}>
```



4.11 ForSyDe.Atom.MoC.Time

```
module ForSyDe.Atom.MoC.Time (
    Time, time, const, e, (*^*), pi, sin, cos, tan, atan, asin,
    acos, sqrt, exp, cosh, sinh, tanh, atanh, asinh, acosh, log
  ) where
```

Collection of utility functions for working with `Time`. While the `CT` MoC describes time as being a non-disjoint continuum (represented in ForSyDe-Atom with `Rational` numbers), most of the

functions here are non-ideal approximations or conversions from floating point equivalents. The trigonometric functions are imported from the numbers package, with a fixed `eps` parameter.

These utilities are meant to get started with using the CT MoC, and should be used with caution if result fidelity is a requirement. In this case the user should find a native `Rational` implementation for a particular function.

`type Time = Rational`
: Type alias for the type to represent metric (continuous) time. Underneath we use `Rational` that is able to represent any t between $t_1 < t_2 \in T$.

`time :: TimeStamp -> Time`
: Converts `TimeStamp` into `Time` representation.

`const :: a -> Time -> a`
: Returns a constant function.

`e :: Time`
: Euler's number in `Time` format. Converted from the `Prelude` equivalent, which is `Floating`.

`(*^*) :: Time -> Time -> Time`
: "Power of" function taking `Time`s as arguments. Converts back and forth to `Floating`, as it uses the `**` operator, so it is prone to conversion errors.

`pi :: Time`
: `Time` representation of the number π. Rational representation with a precision of `0.000001`.

`sin :: Time -> Time`
: Sine of `Time`. Rational representation with a precision of `0.000001`.

`cos :: Time -> Time`
: Cosine of `Time`. Rational representation with a precision of `0.000001`.

`tan :: Time -> Time`
: Tangent of `Time`. Rational representation with a precision of `0.000001`.

`atan :: Time -> Time`
: Arctangent of `Time`. Rational representation with a precision of `0.000001`.

`asin :: Time -> Time`
: Arcsine of `Time`. Rational representation with a precision of `0.000001`.

`acos :: Time -> Time`
: Arccosine of `Time`. Rational representation with a precision of `0.000001`.

`sqrt :: Time -> Time`
: Square root of `Time`. Rational representation with a precision of `0.000001`.

`exp :: Time -> Time`
: Exponent of `Time`. Rational representation with a precision of `0.000001`.

`cosh :: Time -> Time`
: Hyperbolic cosine of `Time`. Rational representation with a precision of `0.000001`.

`sinh :: Time -> Time`
: Hyperbolic sine of `Time`. Rational representation with a precision of `0.000001`.

`tanh :: Time -> Time`
: Hyperbolic tangent of `Time`. Rational representation with a precision of `0.000001`.

`atanh :: Time -> Time`
: Hyperbolic arctangent of `Time`. Rational representation with a precision of `0.000001`.

`asinh :: Time -> Time`
: Hyperbolic arcsine of `Time`. Rational representation with a precision of `0.000001`.

`acosh :: Time -> Time`
: Hyperbolic arccosine of `Time`. Rational representation with a precision of `0.000001`.

`log :: Time -> Time`
: Natural logarithm of `Time`. Rational representation with a precision of `0.000001`.

4.12 ForSyDe.Atom.MoC.TimeStamp

```
module ForSyDe.Atom.MoC.TimeStamp (
    TimeStamp, picosec, nanosec, microsec, milisec, sec, minutes, hours,
    toTime, pi,
  ) where
```

This module implements a timestamp data type, based on `Data.Time.Clock`.

`type TimeStamp = DiffTime`
: Alias for the type representing discrete time. It is inherently quantizable, the quantum being a picosecond (10^{-12} seconds), thus it can be considered order-isomorphic with a set of integers, i.e. between any two timestamps there is a finite number of timestamps. Moreover, a timestamp can be easily translated into a rational number representing fractions of a second, so the conversion between timestamps (discrete time) and rationals (analog/continuous time) is straightforward.

  This type is used in the explicit tags of the `DE` MoC (and subsequently the discrete event evaluation engine for simulating the `CT` MoC).

`picosec :: Integer -> TimeStamp`
: Specifies a timestamp in terms of picoseconds.

`nanosec :: Integer -> TimeStamp`
: Specifies a timestamp in terms of nanoseconds.

`microsec :: Integer -> TimeStamp`
: Specifies a timestamp in terms of microseconds.

`milisec :: Integer -> TimeStamp`
: Specifies a timestamp in terms of miliseconds.

`sec :: Integer -> TimeStamp`
: Specifies a timestamp in terms of seconds.

`minutes :: Integer -> TimeStamp`
: Specifies a timestamp in terms of minutes.

`hours :: Integer -> TimeStamp`
: Specifies a timestamp in terms of hours.

`toTime :: TimeStamp -> Rational`
: Converts a timestamp to a rational number, used for describing continuous time.

`pi :: TimeStamp`
: `TimeStamp` representation of the number π. Converted from the `Prelude` equivalent, which is `Floating`.

4.13 ForSyDe.Atom.Skeleton

```
module ForSyDe.Atom.Skeleton (
    Skeleton((=.=), (=*=), (=\=), (=<<=), first, last),  farm22,  reduce,
    reducei,  pipe,  pipe2
  ) where
```

This module exports a type class with the interfaces for the Skeleton layer atoms. It does *NOT* export any implementation of atoms not any constructor as composition of atoms.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

4.13.1 Atoms

`class Functor c => Skeleton c where`

Class containing all the Skeleton layer atoms.

This class is instantiated by a set of categorical types, i.e. types which describe an inherent potential for being evaluated in parallel. Skeletons are patterns from this layer. When skeletons take as arguments entities from the MoC layer (i.e. processes), the results themselves are parallel process networks which describe systems with an inherent potential to be implemented on parallel platforms. All skeletons can be described as composition of the three atoms below (`=«=` being just a specific instantiation of `=\=`). This possible due to an existing theorem in the categorical type theory, also called the Bird-Merteens formalism R. Bird and Moor, 1997:

factorization A function on a categorical type is an algorithmic skeleton (i.e. catamorphism) *iff* it can be represented in a factorized form, i.e. as a *map* composed with a *reduce*.

Consequently, most of the skeletons for the implemented categorical types are described in their factorized form, taking as arguments either:

- type constructors or functions derived from type constructors
- processes, i.e. MoC layer entities

Most of the ground-work on algorithmic skeletons on which this module is founded has been laid in R. Bird and Moor, 1997, Skillicorn, 1994 and it founds many of the frameworks collected in Fischer, Gorlatch, and Bischof, 2003.

Methods

`(=.=) :: (a -> b) -> c a -> c b`

Atom which maps a function on each element of a structure (i.e. categorical type), defined as:

$$\diamond : (\alpha \to \beta) \to \mathcal{S}(\alpha) \to \mathcal{S}(\beta)$$

`=.=` together with `=*=` form the map pattern.

`(=*=) :: c (a -> b) -> c a -> c b`

Atom which applies the functions contained by as structure (i.e. categorical type), on the elements of another structure, defined as:

$$\diamondsuit : \mathcal{C}(\alpha \to \beta) \to \mathcal{C}(\alpha) \to \mathcal{C}(\beta)$$

`=.=` together with `=*=` form the map pattern.

`(=\=) :: (a -> a -> a) -> c a -> a`

Atom which reduces a structure to an element based on an *associative* function, defined as:

$$\diamondsuit : (\alpha \times \alpha \to \alpha) \to \mathcal{C}(\alpha) \to \alpha$$

`(=«=)`

Skeleton which *pipes* an element through all the functions contained by a structure. N.B.: this is not an atom. It has an implicit definition which might be augmented by instances of this class to include edge cases.

$$\diamondsuit : \mathcal{C}(\alpha \to \alpha) \times \alpha \to \alpha$$
$$\diamondsuit = (\circ)\diamondsuit$$

As the composition operation is not associative, we cannot treat pipe as a true reduction. Alas, it can still be exploited in parallel since it exposes another type of parallelism: time parallelism.

`first :: c a -> a`

Returns the first element in a structure.N.B.: this is not an atom. It has an implicit definition which might be replaced by instances of this class with a more efficient implementation.

$$\texttt{first}_S : \mathcal{C}(\alpha) \to \alpha$$
$$\texttt{first}_S = (\lambda\ a\ b \to a)\diamondsuit$$

`last :: c a -> a`

Returns the last element in a structure.N.B.: this is not an atom. It has an implicit definition which might be replaced by instances of this class with a more efficient implementation.

$$\texttt{last}_S : \mathcal{C}(\alpha) \to \alpha$$
$$\texttt{last}_S = (\lambda\ a\ b \to b)\diamondsuit$$

`instance Skeleton Vector`

Ensures that `Vector` is a structure associated with the Skeleton Layer.

4.13.2 Skeleton constructors

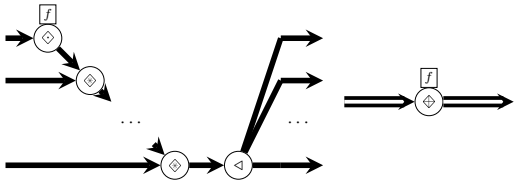
Patterns of in the skeleton layer are provided, like all other patterns in ForSyDe-Atom, as constructors. If the layer below this one is the `MoC` layer, i.e. the functions taken as arguments are processes, then these skeletons can be regarded as process network constructors, as the structures created are process networks with inherent potential for parallel implementation.

```
farm22 :: Skeleton c =>
          (a1 -> a2 -> (b1, b2)) -> c a1 -> c a2 -> (c b1, c b2)
```

`farm` maps a function on a vector. It is the embodiment of the `map` homomorphism, and its naming is inspired from the pattern predominant in HPC. Indeed, if we consider the layer below as being the `MoC` layer (i.e. the passed functions are processes), the resulting structure could be regarded as a "farm of data-parallel processes".

Constructors: `farm[1-8][1-4]`.

$$\diamondsuit : (V^n \to V^m) \to \mathcal{C}^n \to \mathcal{C}^m$$
$$f \diamondsuit (c_1, c_2, ...c_n) = f \diamondsuit c_1 \diamondsuit c_2 \diamondsuit ... \diamondsuit c_n\triangleleft$$



`reduce`

`::`	Skeleton c	
`=>`	`(a -> a -> a)`	associative function (*)
`->`	`c a`	structure
`->`	`a`	reduced element

Infix name for the `=\=` atom operator.

(*) if the operation is not associative then the network can be treated like a pipeline.

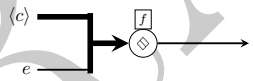
`reducei`

`::`	Skeleton c	
`=>`	`(a -> a -> a)`	associative function (*)
`->`	`a`	initial element of structure
`->`	`c a`	structure
`->`	`a`	reduced element

`reducei` is special case of `reduce` where an initial element is specified outside the reduced vector. It is implemented as a `pipe` with switched arguments, and the reduction function is constrained to be associative. It is semantically equivalent to the pattern depicted below.

(*) if the operation is not associative then the network is semantically equivalent to `pipe1` (see `pipe2`).

$$\texttt{reducei}_S : (\alpha \times \alpha \to \alpha) \to \alpha \to \mathcal{C}(\alpha) \to \alpha$$
$$\texttt{reducei}_S\ f\ e\ \langle c \rangle = f \diamondsuit \langle c \rangle \diamondsuit e$$



`pipe`

`::`	Skeleton c	
`=>`	`c (a -> a)`	vector of functions
`->`	`a`	kernel element
`->`	`a`	result

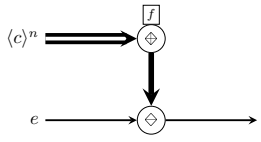
Infix name for the `=«=` skeleton operator.

```
pipe2 :: Skeleton c =>
    (a1 -> a2 -> a -> a) -> c a1 -> c a2 -> a -> a
```

The `pipe` constructors are a more generic form of the `=«=` (`pipe`) skeleton apt for successive partial application and create more robust parameterizable pipeline networks.

Constructors: `comb[1-8]`.

$$\texttt{pipe}_S : (\beta^n \to \alpha \to \alpha) \to \mathcal{C}(\beta^n) \to \alpha \to \alpha$$
$$\texttt{pipe}_S\ f\ \langle c \rangle^n\ e = f \diamondsuit \langle c \rangle^n \diamondsuit e$$



4.14 ForSyDe.Atom.Skeleton.Vector

```
module ForSyDe.Atom.Skeleton.Vector (
    Vector(Null, (:>)), null, unit, (<++>), vector, fromVector, indexes,
    isNull, (<:), farm22, reduce, prefix, suffix, pipe, (=/=), recur,
    cascade2, mesh2, length, index, fanout, fanoutn, generate, iterate,
    first, last, inits, tails, init, tail, concat, reverse, group,
    shiftr, shiftl, rotr, rotl, take, drop, takeWhile, filterIdx, odds,
    evens, stride, get, (<@), (<@!), gather1, (<@>), replace, scatter,
    bitrev, duals, unduals, zipx, unzipx
  ) where
```

This module defines the data type `Vector` as a categorical type, and implements the atoms for the `Skeleton` class. Algorithmic skeletons for `Vector` are mostly described in their factorized form, which ensures that they are catamorphisms (see the factorization theorem). Where efficiency or practicality is a concern, some skeletons are implemented as recurrences. One can still prove that they are catamorphisms through alternative theorems (see Skillicorn, 1994).

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

4.14.1 Vector data type

```
data Vector a
```

=	`Null`	Null element. Terminates a vector.
|	`a (:>) (Vector a)`	appends an element at the head of a vector.

The `Vector`, or at least its interpretation, is the exact equivalent of an infinite list, as defined in R. Bird and Moor, 1997. Its name though is borrowed from Reekie, 1995, since it is more suggestive in the context of process networks.

According to R. Bird and Moor, 1997, `Vector` should be implemented as following:

```
data Vector a = Null                    -- null element
              | Unit a                  -- singleton vector
              | Vector a <++> Vector a  -- concatenate two vectors
```

This construction suggests the possibility of splitting a `Vector` into multiple parts and evaluating it in parallel. Due to reasons of efficiency, and to ensure that the structure is flat and homogeneous, `Vector` is implemented using the same constructors as an infinite list like in R. S. Bird, 1987 (see below). When defining skeletons of vectors we will not use the real constructors though, but the theoretical ones defined above and provided as functions . This way we align ForSyDe-Atom's `Vector` type with the categorical type theory and its theorems.

Another particularity of `Vector` is that it instantiates the reduction atom `=\=` as a *right fold*, as it is the most efficient implementation in the context of lazy evaluation. As a consequence reduction is performed ***from right to left***. This is noticeable especially in the case of pipeline-based skeletons (where `pipe` itself is a reduction with the right-associative composition operator `.`) is performed from right to left, which comes in natural when considering the order of function composition. Thus for `reduce`-based skeletons (e.g. `prefix`, `suffix`, `recur`, `cascade`, `mesh`) the result vectors shall be read from end to beginning.

`instance Functor Vector`
: Provides an implementation for `=.=`.

`instance Applicative Vector`
: Provides an implementation for `=*=`.

`instance Foldable Vector`
: Provides an implementation for `=\=`.

`instance Skeleton Vector`
: Ensures that `Vector` is a structure associated with the Skeleton Layer.

`instance Eq a => Eq (Vector a)`

`instance Read a => Read (Vector a)`
: The vector 1 :> 2 :> Null is read using the string "<1,2>".

`instance Show a => Show (Vector a)`
: The vector 1 :> 2 :> Null is represented as <1,2>.

`instance Plottable a => Plottable (Vector a)`
: Vectors of plottable types

`instance Plottable a => Plot (Vector a)`
: vectors of coordinates

4.14.2 "Constructors"

Theoretical constructors for the `Vector` type, used in the definition of skeletons as catamorphisms.

`null :: Vector a`
: Constructs a null vector.

```
λ> null
<>
```

`unit :: a -> Vector a`
: Constructs a singleton vector.

```
λ> unit 1
<1>
```

`(<++>) :: Vector a -> Vector a -> Vector a`
: Constructs a vector by appending two existing vectors.

```
λ> unit 1 <++> unit 2
<1,2>
```

4.14.3 Utilities

`vector :: [a] -> Vector a`
: Converts a list to a vector.

`fromVector :: Vector a -> [a]`
: Converts a vector to a list.

`indexes :: Vector Integer`
: Creates the infinite vector:

```
<1,2,3,4,...>
```

Used mainly for operation on indexes.

`isNull :: Vector a -> Bool`
Returns `True` if the argument is a null vector.

`(<:) :: Vector a -> a -> Vector a`
Appends an element at the end of a vector.

4.14.4 Skeletons

Algorithmic skeletons on vectors are mainly presented in terms of compositions of the atoms associated with the `Skeleton` Layer. When defining them, we use the following operators:

$$\langle\rangle : \alpha \rightarrow \mathcal{V}(\alpha) \quad (1)$$
$$+\!\!+ : \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha) \quad (2)$$
$$v[n] : \mathcal{V}(\alpha) \times \texttt{Int} \rightarrow \alpha \quad (3)$$
$$v[2..] : \mathcal{V}(\alpha) \times \mathcal{V}(\texttt{Int}) \rightarrow \mathcal{V}(\alpha) \quad (4)$$
$$\infty : \alpha \rightarrow \mathcal{V}(\alpha) \quad (5)$$

where:

- (1) is the `unit` constructor, constructing a singleton vector.
- (2) is the `<++>` constructor, concatenating two vectors.
- (3) is the `<@!>` selector. The subscript notation is used to denote element at position n in a vector.
- (4) suggests an arbitrary selector which returns a vector with another one's elements, based on some indices. The shown example is an alternative notation for the `tail` skeleton.

Functional networks

This sub-category denotes skeletons (patterns) which are take functions as arguments. If the functions are `MoC` layer entities, i.e. processes, then these patterns are capable of constructing parallel process networks. Using the applicative mechanism, the designer has a high degree of freedom when customizing process networks through systematic partial application, rendering numerous possible usages for the same pattern. To avoid over-encumbering the figures, they depict small test cases, which might not expose the full potential of the constructors.

see the naming convention rules on how to interpret, use and develop your own constructors.

`farm22`

`::`	`(a1 -> a2 -> (b1, b2))`	function (e.g. process)
`->`	`Vector a1`	first input vector
`->`	`Vector a2`	second input vector
`->`	`(Vector b1, Vector b2)`	two output vectors

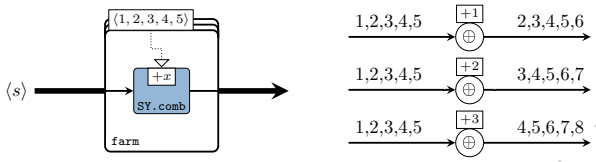
`farm` is simply the `Vector` instance of the skeletom `farm` pattern (see `farm22`). If the function taken as argument is a process, then it creates a farm network of data parallel processes.

Constructors: `farm[1-4][1-4]`.

```
λ> let v1 = vector [1,2,3,4,5]
λ> S.farm21 (+) v1 v1
<2,4,6,8,10>
λ> let s1 = SY.signal [1,2,3,4,5]
λ> let v2 = vector [s1,s1,s1]
λ> S.farm11 (comb11 (+1)) v2
<{2,3,4,5,6},{2,3,4,5,6},{2,3,4,5,6}>
λ> S.farm21 (\x -> comb11 (+x)) v1 v2
<{2,3,4,5,6},{3,4,5,6,7},{4,5,6,7,8}>
```

$$\texttt{farm}_S : (\alpha^m \to \beta^n) \to \langle\alpha\rangle^m \to \langle\beta\rangle^n$$

$$\texttt{farm}_S : \diamondsuit$$

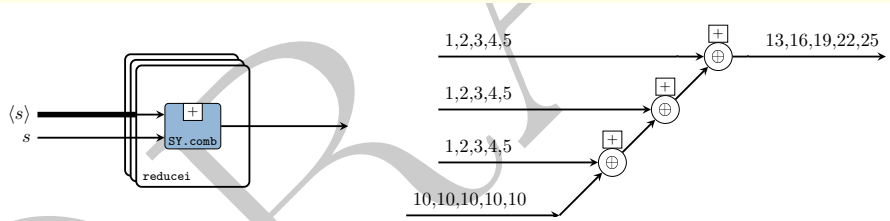


`reduce :: (a -> a -> a) -> Vector a -> a`

As the name suggests, it reduces a vector to an element based on an associative function. If the function is not associative, it can be treated like a pipeline.

`Vector` instantiates the skeletons for both `reduce` and `reducei`.

```
λ> let v1 = vector [1,2,3,4,5]
λ> S.reduce (+) v1
15
λ> let s1 = SY.signal [1,2,3,4,5]
λ> let s2 = SY.signal [10,10,10,10,10]
λ> let v2 = vector [s1,s1,s1]
λ> S.reduce (comb21 (+)) v2
{3,6,9,12,15}
λ> S.reducei (comb21 (+)) s2 v2
{13,16,19,22,25}
```



`prefix :: (b -> b -> b) -> Vector b -> Vector b`

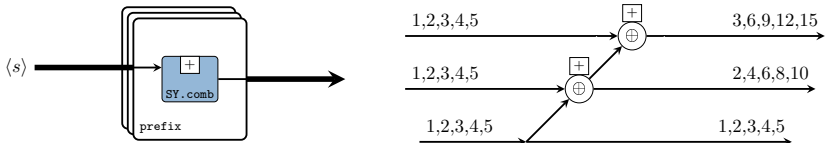
`prefix` peforms the *parallel prefix* operation on a vector. Equivalent process networks are constructed if processes are passed as arguments.

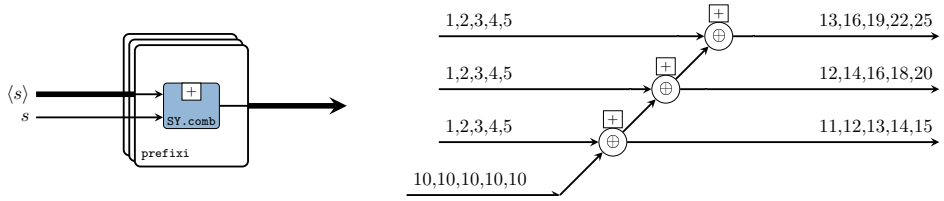
Similar to `reduce` and `reducei`, two versions `prefix` and `prefixi` are provided.

```
λ> let v1 = vector [1,2,3,4,5]
λ> prefix (+) v1
<15,14,12,9,5>
λ> let s1 = SY.signal [1,2,3,4,5]
λ> let s2 = SY.signal [10,10,10,10,10]
λ> let v2 = vector [s1,s1,s1]
λ> prefix (comb21 (+)) v2
<{3,6,9,12,15},{2,4,6,8,10},{1,2,3,4,5}>
λ> prefixi (comb21 (+)) s2 v2
<{13,16,19,22,25},{12,14,16,18,20},{11,12,13,14,15}>
```

$$\texttt{prefix}_S : (\alpha \times \alpha \to \alpha) \to \langle\alpha\rangle \to \langle\alpha\rangle$$

$$\texttt{prefix}_S\ f\ \langle a\rangle = (f\diamondsuit) \diamondsuit (\texttt{tails}_S \langle a\rangle)$$





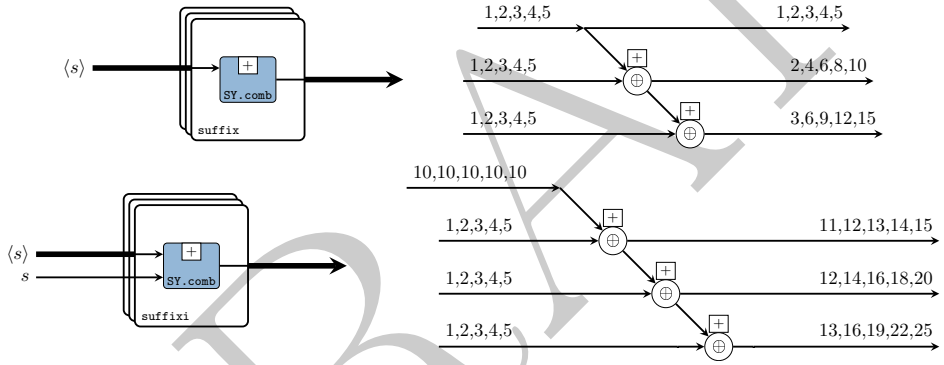
`suffix :: (b -> b -> b) -> Vector b -> Vector b`

`suffix` peforms the *parallel suffix* operation on a vector. Equivalent process networks are constructed if processes are passed as arguments.

Similar to `reduce` and `reducei`, two versions `suffix` and `suffixi` are provided.

```
λ> let v1 = vector [1,2,3,4,5]
λ> suffix (+) v1
<1,3,6,10,15>
λ> let s1 = SY.signal [1,2,3,4,5]
λ> let s2 = SY.signal [10,10,10,10,10]
λ> let v2 = vector [s1,s1,s1]
λ> suffix (comb21 (+)) v2
<{1,2,3,4,5},{2,4,6,8,10},{3,6,9,12,15}>
λ> suffixi (comb21 (+)) s2 v2
<{11,12,13,14,15},{12,14,16,18,20},{13,16,19,22,25}>
```

$$\texttt{suffix}_S : (\alpha \times \alpha \to \alpha) \to \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{suffix}_S\ f\ \langle a\rangle = (f\diamondsuit) \diamondplus (\texttt{inits}_S\langle a\rangle)$$



`pipe`

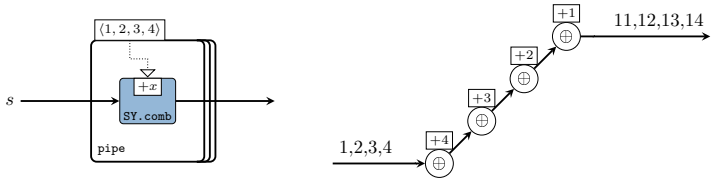
`::`	`Vector (a -> a)`	vector of functions
`->`	`a`	input
`->`	`a`	output

`pipe` creates a pipeline of functions from a vector. `pipe` simply instantiates the `=«=` atom whereas `pipeX` instantiate their omologi from the `ForSyDe.Atom.Skeleton` module (see `pipe2`).

OBS: the pipelining is done in the order dictated by the function composition operator: from right to left.

Constructors: `pipe[1-4]`.

```
λ> let v1 = vector [(+1),(+1),(+1)]
λ> S.pipe v1 1
4
λ> let s1 = SY.signal [1,2,3,4]
λ> let v2 = vector [1,2,3,4]
λ> S.pipe1 (\x -> comb11 (+x)) v2 s1
{11,12,13,14}
```



`(=/=) :: Vector (a -> a) -> a -> Vector a`

Infix operator for `recur`.

`recur`

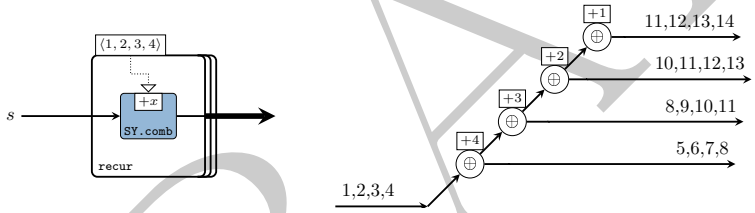
`::`	`Vector (a -> a)`	vector of functions
`->`	`a`	input
`->`	`Vector a`	output

`recur` creates a systolic array from a vector of functions. Just like `pipe` and `pipeX`, there exists a raw `recur` version with an infix operator `=/=`, and the enhanced `recurX` which is meant for systematic partial application of a function on an arbitrary number of vectors until the desired vector of functions is obtained.

Constructors: `(=/=)`, `recur`, `recuri`, `recur[1-4][1-4]`.

```
λ> let v1 = vector [(+1),(+1),(+1)]
λ> recur v1 1
<4,3,2>
λ> recuri v1 1
<4,3,2,1>
λ> let s1 = SY.signal [1,2,3,4]
λ> let v2 = vector [1,2,3,4]
λ> recur1 (\x -> comb11 (+x)) v2 s1
<{11,12,13,14},{10,11,12,13},{8,9,10,11},{5,6,7,8}>
```

$$\diamondsuit : \langle \alpha \to \alpha \rangle \to \alpha \to \langle \alpha \rangle$$
$$\langle f \rangle \diamondsuit i = (\circlearrowleft i) \Diamond (\texttt{tails}_S \langle f \rangle)$$



`cascade2`

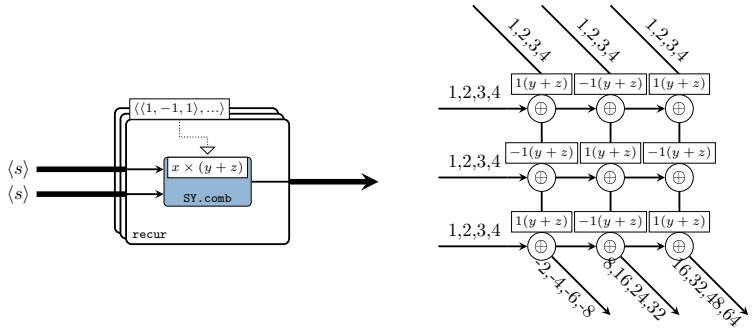
`::`	`(a2 -> a1 -> a -> a -> a)`	`function41` which needs to be applied to `function21`
`->`	`Vector (Vector a2)`	fills in the first argument in the function above
`->`	`Vector (Vector a1)`	fills in the second argument in the function above
`->`	`Vector a`	first input vector (e.g. of signals)
`->`	`Vector a`	second input vector (e.g. of signals)
`->`	`Vector a`	output

`cascade` creates a "cascading mesh" as a result of piping a vector into a vector of recur arrays.

Constructors: `cascade`, `cascade[1-4]`.

```
λ> let v1 = vector [1,2,3,4]
λ> cascade (+) v1 v1
<238,119,49,14>
λ> let s1 = SY.signal [1,2,3,4]
λ> let vs = vector [s1, s1, s1]
λ> cascade (comb21 (+)) vs vs
<{20,40,60,80},{10,20,30,40},{4,8,12,16}>
λ> let vv = vector [vector [1,-1,1], vector [-1,1,-1], vector [1,-1,1] ]
λ> cascade1 (\x -> comb21 (\y z-> x*(y+z))) vv vs vs
<{16,32,48,64},{8,16,24,32},{-2,-4,-6,-8}>
```

$$\texttt{cascade}_S : (\alpha \to \alpha \to \alpha) \to \langle \alpha \rangle \to \langle \alpha \rangle \to \langle \alpha \rangle$$
$$\texttt{cascade}_S\ p\ \langle s \rangle_1\ \langle s \rangle_2 = \texttt{scanf}_S \Diamond \langle s \rangle_2 \diamondsuit \langle s \rangle_1$$
$$\text{where}\quad \texttt{scanf}_S\ s_2\ \langle s \rangle_1 = p \Diamond \langle s \rangle_1 \diamondsuit s_2$$



`mesh2`

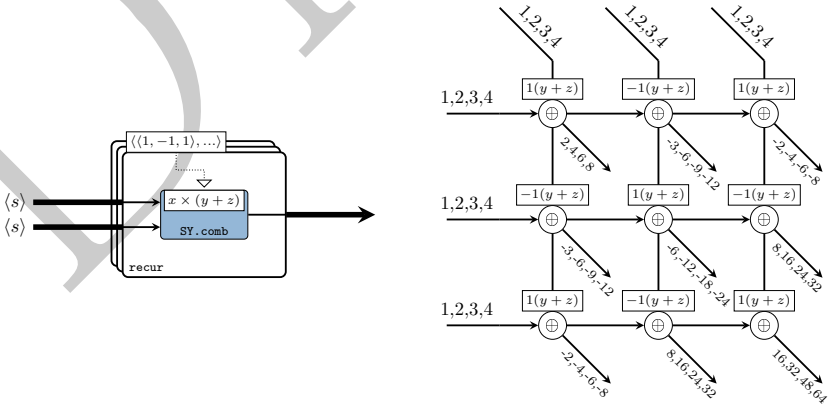
`::`	`(a2 -> a1 -> a -> a -> a)`	`function41` which needs to be applied to `function21`
`->`	`Vector (Vector a2)`	fills in the first argument in the function above
`->`	`Vector (Vector a1)`	fills in the second argument in the function above
`->`	`Vector a`	first input vector (e.g. of signals)
`->`	`Vector a`	second input vector (e.g. of signals)
`->`	`Vector (Vector a)`	output, a 2D vector

`mesh` creates a 2D systolic array as a result of piping a vector into a vector of 1D systolic arrays.

Constructors: `mesh`, `mesh[1-4]`.

```
λ> let v1 = vector [1,2,3,4]
λ> mesh (+) v1 v1
<<238,119,49,14>,<119,70,35,13>,<49,35,22,11>,<14,13,11,8>>
λ> let s1 = SY.signal [1,2,3,4]
λ> let vs = vector [s1, s1, s1]
λ> mesh (comb21 (+)) vs vs
<<{20,40,60,80},{10,20,30,40},{4,8,12,16}>,<{10,20,30,40},{6,12,18,24},{3,6,9,12}>,<{4,8,12,16},
     {3,6,9,12},{2,4,6,8}>>
λ> let vv = vector [vector [1,-1,1], vector [-1,1,-1], vector [1,-1,1]]
λ> mesh1 (\x -> comb21 (\y z-> x*(y+z))) vv vs vs
<<{16,32,48,64},{8,16,24,32},{-2,-4,-6,-8}>,<{8,16,24,32},{-6,-12,-18,-24},{-3,-6,-9,-12}>,
    <{-2,-4,-6,-8},{-3,-6,-9,-12},{2,4,6,8}>>
```

$$\texttt{mesh}_S : (\alpha \to \alpha \to \alpha) \to \langle\alpha\rangle \to \langle\alpha\rangle \to \langle\langle\alpha\rangle\rangle$$
$$\texttt{mesh}_S\ p\ \langle s\rangle_1\ \langle s\rangle_2 = \texttt{scanf}_S \diamondsuit \langle s\rangle_2 \diamondsuit \langle s\rangle_1$$
$$\text{where}\quad \texttt{scanf}_S\ s_2\ \langle s\rangle_1 = p \diamondsuit \langle s\rangle_1 \diamondsuit s_2$$



Queries

Queries return various information about a vector. They are also built as skeletons.

`length :: Num p => Vector a -> p`
returns the number of elements in a value.

```
λ> length $ vector [1,2,3,4,5]
5
```

$$\texttt{length}_S : \langle \alpha \rangle \rightarrow \texttt{Int}$$
$$\texttt{length}_S = (+^{\diamondsuit}) \circ (1^{\diamondsuit})$$

`index :: Vector a2 -> Vector Integer`
returns a vector with the indexes from another vector.

```
λ> index $ vector [1,1,1,1,1,1,1]
<1,2,3,4,5,6,7>
```

Generators

Generators are specific applications of the `prefix` or `suffix` skeletons.

`fanout :: t -> Vector t`
`fanout` repeats an element. As a process network it distributes the same value or signal to all the connected processes down the line. Depending on the target platform and the refinement decisions involved, it may be interpreted in the following implementations:

- global or shared memory in case of a massively parallel platform (e.g. GPU)
- a (static) memory or cache location in memory-driven architectures (e.g. CPU)
- a fanout in case of a HDL system
- a broadcast in case of a distributed system

`fanoutn :: (Num t, Ord t) => t -> a -> Vector a`
`fanoutn` is the same as `fanout`, but the length of the result is also provided.

```
λ> fanoutn 5 1
<1,1,1,1,1>
```

`generate :: (Num t, Ord t) => t -> (a -> a) -> a -> Vector a`
`generate` creates a vector based on a kernel function. It is just a restricted version of `recur`.

```
λ> generate 5 (+1) 1
<6,5,4,3,2>
```

$$\texttt{generate}_S : \texttt{Int} \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \langle \alpha \rangle$$
$$\texttt{generate}_S\ n\ f\ i = \langle f \rangle_{\times n}\ \diamondsuit\ i$$

`iterate :: (Num t, Ord t) => t -> (a -> a) -> a -> Vector a`
`iterate` is a version of `generate` which keeps the initial element as well. It is a restricted version of `recuri`.

```
λ> iterate 5 (+1) 1
<5,4,3,2,1>
```

Permutators

Permutators perform operations on the very structure of vectors, and make heavy use of the vector constructors.

`first :: Vector a -> a`

Instance of `first`

```
λ> S.first $ vector [1,2,3,4,5]
1
```

`last :: Vector a -> a`

Instance of `last`

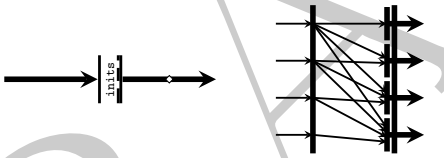
```
λ> S.last $ vector [1,2,3,4,5]
5
```

`inits :: Vector a -> Vector (Vector a)`

creates a vector of all the initial segments in a vector.

```
λ> inits $ vector [1,2,3,4,5]
<<1>,<1,2>,<1,2,3>,<1,2,3,4>,<1,2,3,4,5>>
```

$$\texttt{inits}_S : \langle\alpha\rangle \rightarrow \langle\langle\alpha\rangle\rangle$$
$$\texttt{inits}_S\ n = (sel^{\diamondsuit}) \circ (\langle\langle\rangle\rangle^{\oplus})$$
$$\text{where } sel\ x\ y = x +\!\!+ ((x[L]+\!\!+) \oplus y)$$

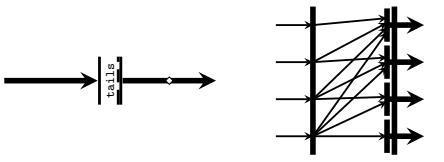


`tails :: Vector a -> Vector (Vector a)`

creates a vector of all the final segments in a vector.

```
λ> tails $ vector [1,2,3,4,5]
<<1,2,3,4,5>,<2,3,4,5>,<3,4,5>,<4,5>,<5>>
```

$$\texttt{tails}_S : \langle\alpha\rangle \rightarrow \langle\langle\alpha\rangle\rangle$$
$$\texttt{tails}_S\ n = (sel^{\diamondsuit}) \circ (\langle\langle\rangle\rangle^{\oplus})$$
$$\text{where } sel\ x\ y = ((+\!\!+y[1]) \oplus x) +\!\!+ y$$



`init :: Vector a -> Vector a`

Returns the initial segment of a vector.

```
λ> init $ vector [1,2,3,4,5]
<1,2,3,4>
```

$$\texttt{init}_S : \langle\alpha\rangle \rightarrow \langle\alpha\rangle$$
$$\texttt{init}_S = \texttt{inits}_S[L-1]$$

`tail :: Vector a -> Vector a`

Returns the tail of a vector.

```
λ> tail $ vector [1,2,3,4,5]
<2,3,4,5>
```

$$\texttt{tail}_S : \langle\alpha\rangle \rightarrow \langle\alpha\rangle$$
$$\texttt{tail}_S = \texttt{tails}_S[2]$$

`concat :: Vector (Vector a) -> Vector a`

concatenates a vector of vectors.

```
λ> concat $ vector [vector[1,2,3,4], vector[5,6,7]]
<1,2,3,4,5,6,7>
```

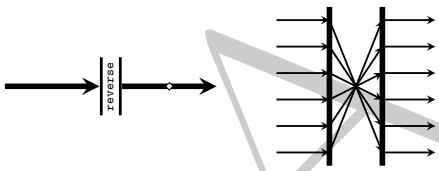
$$\texttt{concat}_S : \langle\langle\alpha\rangle\rangle \rightarrow \langle\alpha\rangle$$
$$\texttt{concat}_S = +\!\!+\diamond$$

`reverse :: Vector a -> Vector a`

reverses the elements in a vector.

```
λ> reverse $ vector [1,2,3,4,5]
<5,4,3,2,1>
```

$$\texttt{reverse}_S : \langle\alpha\rangle \rightarrow \langle\alpha\rangle$$
$$\texttt{reverse}_S\ n = (rev\diamond) \circ (\langle\rangle\diamond\!\!+)$$
$$\text{where } rev\ x\ y = y +\!\!+ x$$

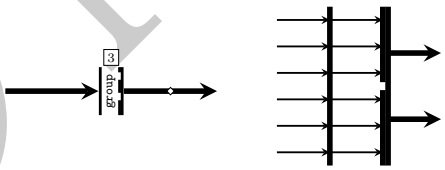


`group :: Integer -> Vector a -> Vector (Vector a)`

groups a vector into sub-vectors of n elements.

```
λ> group 3 $ vector [1,2,3,4,5,6,7,8]
<<1,2,3>,<4,5,6>,<7,8>>
```

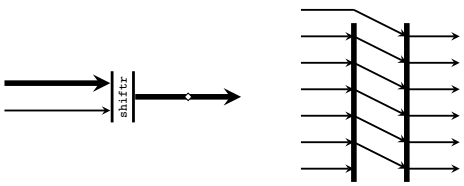
$$\texttt{group}_S : Int \rightarrow \langle\alpha\rangle \rightarrow \langle\langle\alpha\rangle\rangle$$
$$\texttt{group}_S\ n = (\texttt{take}_S\ n)\ \diamond\!\!+ \circ (\texttt{drop}_S\ n)\diamond$$



`shiftr :: Vector a -> a -> Vector a`

right-shifts a vector with an element.

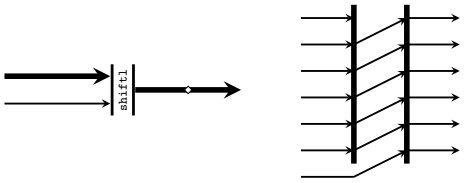
```
λ> vector [1,2,3,4] `shiftr` 8
<8,1,2,3>
```



`shiftl :: Vector a -> a -> Vector a`

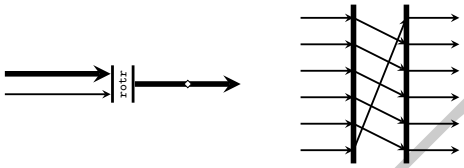
left-shifts a vector with an element.

```
λ> vector [1,2,3,4] `shiftl` 8
<2,3,4,8>
```



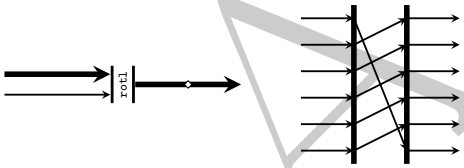
`rotr :: Vector a -> Vector a`
rotates a vector to the right.

```
λ> rotr $ vector [1,2,3,4]
<4,1,2,3>
```



`rotl :: Vector a -> Vector a`
rotates a vector to the left.

```
λ> rotl $ vector [1,2,3,4]
<2,3,4,1>
```



`take :: Integer -> Vector a -> Vector a`
takes the first n elements of a vector.

```
λ> take 5 $ vector [1,2,3,4,5,6,7,8,9]
<1,2,3,4,5>
```

$$\texttt{take}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{take}_S\ n = (sel\diamondsuit) \circ (\langle\rangle\diamondsuit)$$
$$\text{where } sel\ x\ y = \begin{cases} x +\!\!+ y & \text{if } index(x) < n \\ x & \text{otherwise} \end{cases}$$

`drop :: Integer -> Vector a -> Vector a`
drops the first n elements of a vector.

```
λ> drop 5 $ vector [1,2,3,4,5,6,7,8,9]
<6,7,8,9>
```

$$\texttt{drop}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{drop}_S\ n = (sel\diamondsuit) \circ (\langle\rangle\diamondsuit)$$
$$\text{where } sel\ x\ y = \begin{cases} x +\!\!+ y & \text{if } index(x) > n \\ y & \text{otherwise} \end{cases}$$

`takeWhile :: (a -> Bool) -> Vector a -> Vector a`
takes the first elements in a vector until the first element that does not fulfill a predicate.

```
λ> takeWhile (<5) $ vector [1,2,3,4,5,6,7,8,9]
<1,2,3,4>
```

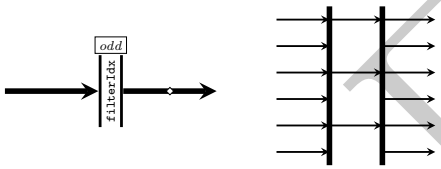
$$\texttt{takeWhile}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{takeWhile}_S\ f = \texttt{concat}_S \circ (sel\diamondsuit) \circ (\langle\langle\rangle\rangle^{\diamondsuit})$$
$$\text{where } sel\ x\ y = \begin{cases} x +\!\!+ y & \text{if } f(y[1]) = true \wedge \exists x[L] \\ x & \text{otherwise} \end{cases}$$

```
filterIdx :: (Integer -> Bool) -> Vector a -> Vector a
```

returns a vector containing only the elements of another vector whose index satisfies a predicate.

```
λ> filterIdx (\x -> x `mod` 3 == 0) $ vector [0,1,2,3,4,5,6,7,8,9]
<2,5,8>
```

$$\texttt{filterIdx}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{filterIdx}_S\ f = (sel\diamondsuit) \circ (\langle\rangle^{\diamondsuit})$$
$$\text{where } sel\ x\ y = \begin{cases} x +\!\!+ y & \text{if } f(index(x)) = true \\ y & \text{otherwise} \end{cases}$$




```
odds :: Vector a -> Vector a
```

$$\texttt{odds}_S = \texttt{filterIdx}_S(odd)$$

```
evens :: Vector a -> Vector a
```

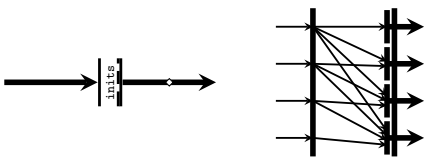
$$\texttt{odds}_S = \texttt{filterIdx}_S(even)$$

```
stride
    ::  Integer    first index
    ->  Integer    stride length
    ->  Vector a
    ->  Vector a
```

does a stride-selection on a vector.

```
λ> stride 1 3 $ vector [1,2,3,4,5,6,7,8,9]
<1,4,7>
```

$$\texttt{stride}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{stride}_S\ first\ stride = (sel\diamondsuit) \circ (\langle\rangle^{\diamondsuit})$$
$$\text{where } sel\ xy = \begin{cases} x +\!\!+ y & \text{if } (index(x) - first) \mod stride = 0 \\ y & \text{otherwise} \end{cases}$$




```
get :: Integer -> Vector a -> Maybe a
```

returns the n-th element in a vector, or `Nothing` if $n > l$.

```
λ> get 3 $ vector [1,2,3,4,5]
Just 3
```

$$\texttt{get}_S : \langle\alpha\rangle \to \alpha$$
$$\texttt{get}_S\ n = (sel\diamondsuit)$$
$$\text{where } sel\ x\ y = \begin{cases} x & \text{if } index(x) = n \\ y & \text{otherwise} \end{cases}$$

`(<@) :: Vector a -> Integer -> Maybe a`
: the same as `get` but with flipped arguments.

`(<@!) :: Vector p -> Integer -> p`
: unsafe version of `<@>`. Throws an exception if $n > l$.

`gather1`

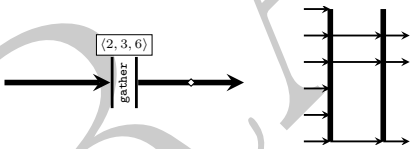
`::`	`Vector Integer`	vector of indexes
`->`	`Vector a`	input vector
`->`	`Vector (Maybe a)`	

selects the elements in a vector at the incexes contained by another vector.

The following versions of this skeleton are available, the number suggesting how many nested vectors it is operating upon: `gather[1-5]`

```
λ> let ix = vector [vector [1,3,4], vector [3,5,1], vector [5,8,9]]
λ> let v = vector [11,12,13,14,15]
λ> gather2 ix v
<<Just 11,Just 13,Just 14>,<Just 13,Just 15,Just 11>,<Just 15,Nothing,Nothing>>
```

$$\texttt{gather}_S : \langle Int\rangle\langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{gather}_S\ \langle ix\rangle\langle v\rangle = sel \diamondsuit \langle ix\rangle$$
$$\text{where } sel\ x = \langle v\rangle[x]$$



`(<@>)`

`::`	`Vector a`	input vector
`->`	`Vector Integer`	vector of indexes
`->`	`Vector (Maybe a)`	

the same as `gather1` but with flipped arguments

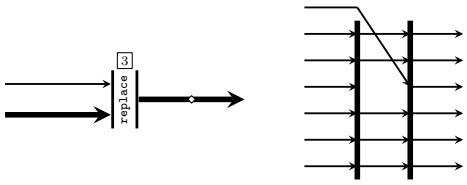
The following versions of this skeleton are available, the number suggesting how many nested vectors it is operating upon.

```
(<@>), (<<@>>), (<<<@>>>), (<<<<@>>>>), (<<<<<@>>>>>),
```

`replace :: Integer -> p -> Vector p -> Vector p`
: replaces the n-th element in a vector with another.

```
λ> replace 5 15 $ vector [1,2,3,4,5,6,7,8,9]
<1,2,3,4,15,6,7,8,9>
```

$$\texttt{replace}_S : \langle\alpha\rangle \to \langle\alpha\rangle$$
$$\texttt{replace}_S\ n\ r = (sel\diamondsuit) \circ (\langle\rangle\diamondsuit)$$
$$\text{where } sel\ x\ y = \begin{cases} \langle r\rangle +\!\!+ y & \text{if } index(x) = n \\ x +\!\!+ y & \text{otherwise} \end{cases}$$

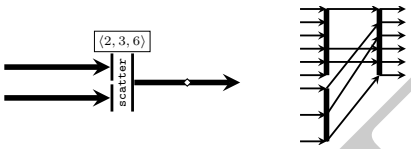


`scatter :: Vector Integer -> Vector p -> Vector p -> Vector p`

scatters the elements in a vector based on the indexes contained by another vector.

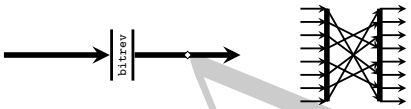
```
λ> scatter (vector [2,4,5]) (vector [0,0,0,0,0,0,0,0]) (vector [1,1,1])
<0,1,0,1,1,0,0,0>
```

$$\texttt{scatter}_S : \langle Int \rangle \langle \alpha \rangle \to \langle \alpha \rangle \to \langle \alpha \rangle$$
$$\texttt{scatter}_S\ \langle ix \rangle \langle host \rangle = (sel^{\diamondsuit}) \circ (\langle \rangle^{\oplus})$$
$$\text{where } sel\ x = \texttt{replace}_S\ index(x)\ x[1]\ \langle host \rangle$$



`bitrev :: Vector a -> Vector a`

performs a bit-reverse permutation.




```
λ> bitrev $ vector ["000","001","010","011","100","101","110","111"]
<"111","011","101","001","110","010","100","000">
```

`duals :: Vector b2 -> (Vector b2, Vector b2)`

splits a vector in two equal parts.

```
λ> duals $ vector [1,2,3,4,5,6,7]
(<1,2,3>,<4,5,6>)
```

`unduals :: Vector a -> Vector a -> Vector a`

concatenates a previously split vector. See also `duals`

Interfaces

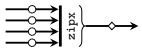
`zipx`

`::`	MoC e	
`=>`	`Vector ((Vector a -> Vector a -> Vector a) -> Fun e (Vector a) (Fun e (Vector a) (Ret e (Vector a))))`	vector of MoC-specific context wrappers for the function `<++>`
`->`	`Vector (Stream (e a))`	input vector of signals
`->`	`Stream (e (Vector a))`	output signal of vectors

`zipx` is a template skeleton for "zipping" a vector of signals. It synchronizes all signals (of the same MoC) in a vector and outputs one signal with vectors of the synced values. For each signal in the input vector it requires a function which *translates* a partition of events (see `ForSyDe.Atom.MoC`) into sub-vectors.

There exist helper intances of the `zipx` skeleton interface for all supported MoCs.

$$\texttt{zipx}_S : \langle \langle \dot{\alpha} \rangle \times \langle \dot{\alpha} \rangle \to \langle \dot{\alpha} \rangle \rangle \to \langle \mathcal{S}(\alpha) \rangle \to \mathcal{S}(\langle \alpha \rangle)$$
$$\texttt{zipx}_S\ \langle part \rangle = ((+^{\oplus}) \diamondsuit \langle part \rangle) \circ (\texttt{unit}_S{}^{\oplus})$$




```
unzipx :: MoC e =>
          (Vector a -> Vector (Ret e a))
          -> Integer -> Stream (e (Vector a)) -> Vector (Stream (e a))
```

`unzipx` is a template skeleton to unzip a signal carrying vectors into a vector of multiple signals. It required a function that *splits* a vector of values into a vector of event partitions belonging to output signals. Unlike `zipx`, it also requires the number of output signals. The reason for this is that it is impossible to determine the length of the output vector without "sniffing" the content of the input events, which is out of the scope of skeletons and may lead to unsafe behavior. The length of the output vector is needed in order to avoid infinite recurrence.

There exist helper intances of the `unzipx` skeleton interface for all supported MoCs.

$$\texttt{unzipx}_S : (\langle\alpha\rangle \to \langle\dot{\alpha}\rangle) \to Int \to \mathcal{S}(\langle\alpha\rangle) \to \langle\mathcal{S}(\alpha)\rangle$$
$$\texttt{unzipx}_S\ part\ n = ((\texttt{first}_S{}^{\oplus})^{\diamondsuit}) \circ (\langle\texttt{tail}_S\rangle_{\times n}{}^{\diamondsuit}) \circ (part^{\oplus})$$

fig/skel-vector-comm-unzipx.pdf

4.15 ForSyDe.Atom.Utility.Plot

```
module ForSyDe.Atom.Utility.Plot (
    Config(Cfg, verbose, path, title, rate, xmax, labels, fire, other),
    defaultCfg,  silentCfg,  noJunkCfg,  prepare,  prepareL,  prepareV,
    showDat,  dumpDat,  plotGnu,  heatmapGnu,  showLatex,  dumpLatex,
    plotLatex,  Plottable(toCoord),  Plot(sample, sample', takeUntil, getInfo),
    PInfo(Info, typeid, command, measure, style, stacking, sparse),  Samples,
    PlotData
  ) where
```

This module imports plotting and data dumping functions working with "plottable" data types, i.e. instances of the `Plot` and `Plottable` type classes.

4.15.1 User API

The following commands are frequently used as part of the normal modeling routine.

Configuration settings

```
data Config
```

=	Cfg	
>	verbose :: Bool	verbose printouts on terminal
>	path :: String	directory where all dumped files will be found
>	title :: String	base name for dumped files
>	rate :: Float	sample rate if relevant. Useful for explicit-tagged signals, ignored otherwise.
>	xmax :: Float	Maximum X coordinate. Mandatory for infinite structures, optional otherwise.
>	labels :: [String]	list of labels with the names of the structures plotted
>	fire :: Bool	if relevant, fires a plotting or compiling program.
>	other :: Bool	if relevant, dumps additional scripts and plots.

Record structure containing configuration settings for the plotting commands.

```
instance Show Config
```

```
defaultCfg :: Config
```

Default configuration: verbose, dump everything possible, fire whatever program needed. Check source for settings.

Example usage:

```
λ> defaultCfg {xmax = 15, verbose = False, labels = ["john","doe"]}
Cfg {verbose = False, path = "./fig", title = "plot", rate = 1.0e-2, xmax = 15.0, labels = ["
    john","doe"], fire = True, other = True}
```

```
silentCfg :: Config
```

Silent configuration: does not fire any program or print our unnecessary info. Check source for settings.

```
noJunkCfg :: Config
```

Clean configuration: verbose, does not dump more than necessary, fire whatever program needed. Check source for settings.

Data preparation

```
prepare
```

::	Plot a	
=>	Config	configuration settings
->	a	plottable data type
->	PlotData	structure ready for dumping

Prepares a single plottable data structure to be dumped and/or plotted.

```
prepareL :: Plot a => Config -> [a] -> PlotData
```

Prepares a list of plottable data structures to be dumped and/or plotted. See `prepare`.

```
prepareV :: Plot a => Config -> Vector a -> PlotData
```

Prepares a vector of plottable data structures to be dumped and/or plotted. See `prepare`.

Dumping and plotting data

```
showDat :: PlotData -> IO ()
```

Prints out the sampled contents of a `prepare`d data set.

`dumpDat :: PlotData -> IO [String]`
: Dumps the sampled contents of a `prepared` data set into separate `.dat` files.

`plotGnu :: PlotData -> IO ()`
: Generates a GNUplot script and `.dat` files for plotting the sampled contents of a `prepared` data set. Depending on the configuration settings, it also dumps LaTeX and PDF plots, and fires the script.

  OBS: needless to say that GNUplot needs to be installed in order to use this command. Also, in order to fire GNUplot from a ghci session you might need to install `gnuplot-x11`.

`heatmapGnu :: PlotData -> IO ()`
: Similar to `plotGnu` but creates a heatmap plot using the GNUplot engine. For this, the input needs to contain at least two columns of data, otherwise the plot does not show anything, i.e. the samples need to be lists or vectors of two or more elements.

  OBS: same dependencies are needed as for `plotGnu`.

`showLatex :: PlotData -> IO ()`
: Prints out a LaTeX environment from a `prepared` data set. This environment should be paste inside a `tikzpicture` in a document title which imports the ForSyDe-LaTeX package.

`dumpLatex :: PlotData -> IO [String]`
: Dumps a set of formatted data files with the extension `.flx` that can be imported by a LaTeX document which uses the ForSyDe-LaTeX package.

`plotLatex :: PlotData -> IO ()`
: Creates a standalone LaTeX document which uses the ForSyDe-LaTeX package, plotting a `prepared` data set. Depending on the configuration settings, the command `pdflatex` may also be invoked to compile a pdf image.

  OBS: A LaTeX compiler is required to run the `pdflatex` command. The ForSyDe-LaTeX package also needs to be installed according to the instructions on the project web page.

4.15.2 The data types

Below the data types involved are shown and the plottable structures are documented.

`class Plottable a where`
: This class gathers types which can be sampled and converted to a numerical string which can be read and interpreted by a plotter engine.

  Methods

  `toCoord :: a -> String`
  : Transforms the input type into a coordinate string.

`instance (Show a, Real a) => Plottable a`
: Real numbers that can be converted to a floating point representation

`instance Plottable TimeStamp`
: Time stamps

`instance Plottable a => Plottable (Vector a)`
: Vectors of plottable types

`instance (Show a, Plottable a) => Plottable (AbstExt a)`
: Absent-extended plottable types

`class Plot a where`
This class gathers all ForSyDe-Atom structures that can be plotted.

Methods

`sample :: Float -> a -> Samples`
Samples the data according to a given step size.

`sample :: a -> Samples ,`
Samples the data according to the internal structure.

`takeUntil :: Float -> a -> a`
Takes the first samples until a given tag.

`getInfo :: a -> PInfo`
Returns static information about the data type.

`instance Plottable a => Plot (Vector a)`
vectors of coordinates

`instance Plottable a => Plot (Signal a)`
`SY` signals.

`instance Plottable a => Plot (Signal a)`
`SDF` signals.

`instance Plottable a => Plot (Signal a)`
`DE` signals.

`instance Plottable a => Plot (Signal a)`
`CT` signals.

```
data PInfo
     =  Info
     >  typeid ::  String    id used usually in implicit tags
     >  command ::  String   LaTeX identifier
     >  measure ::  String   unit of measure
     >  style ::  String     style tweaking in the GNUplot script
     >  stacking ::  Bool     if the plot is stacking
     >  sparse ::  Bool       if the sampled data is sparse instead of dense
```

Static information of each plottable data type.

`instance Show PInfo`

`type Samples = [(String, String)]`
Alias for sampled data

`type PlotData = (Config, PInfo, [(String, Samples)])`
Alias for a data set `prepared` to be plotted.

4.16 ForSyDe.Atom.Utility.Tuple

```
module ForSyDe.Atom.Utility.Tuple (
    at22,  (||<),  (<>),  ($$)
  ) where
```

This module implements general purpose utility functions. It mainly hosts functions dealing with tuples. Utility are provided for up until 9-tuples. Follow the examples in the source code in case it does not suffice.

> **Reminder**
>
> Make sure to consult naming conventions in section 4.1.1 in order to interpret the names and type signatures correctly.

`at22 :: (a1, a2) -> a2`

The `at`xy functions return the y-th element of an x-tuple.

`ForSyDe.Atom.Utility` exports the constructors below. Please follow the examples in the source code if they do not suffice:

```
at21, at22,
at31, at32, at33,
at41, at42, at43, at44,
at51, at52, at53, at54, at55,
at61, at62, at63, at64, at65, at66,
at71, at72, at73, at74, at75, at76, at77,
at81, at82, at83, at84, at85, at86, at87, at88,
at91, at92, at93, at94, at95, at96, at97, at98, at99,
```

Example:

```
λ> at53 (1,2,3,4,5)
3
```

`(||<) :: (Functor f1, Functor f2) =>`
`f1 (f2 (a1, a2)) -> (f1 (f2 a1), f1 (f2 a2))`

This set of utility functions "unzip" nested n-tuples, provided as postfix operators. They are crucial for reconstructing data types from higher-order functions which input functions with multiple outputs. It relies on the nested types being instances of `Functor`.

The operator convention is `(|+<+)`, where the number of `|` represent the number of layers the n-tuple is lifted, while the number of `<` + 1 is the order n of the n-tuple.

`ForSyDe.Atom.Utility` exports the constructors below. Please follow the examples in the source code if they do not suffice:

```
   |<,    |<<,    |<<<,    |<<<<,    |<<<<<,    |<<<<<<,    |<<<<<<<,    |<<<<<<<<,
  ||<,   ||<<,   ||<<<,   ||<<<<,   ||<<<<<,   ||<<<<<<,   ||<<<<<<<,   ||<<<<<<<<,
 |||<,  |||<<,  |||<<<,  |||<<<<,  |||<<<<<,  |||<<<<<<,  |||<<<<<<<,  |||<<<<<<<<,
||||<, ||||<<, ||||<<<, ||||<<<<, ||||<<<<<, ||||<<<<<<, ||||<<<<<<<, ||||<<<<<<<<,
```

Example:

```
λ> :set -XPostfixOperators
λ> ([Just (1,2,3), Nothing, Just (4,5,6)] ||<<)
([Just 1,Nothing,Just 4],[Just 2,Nothing,Just 5],[Just 3,Nothing,Just 6])
```

`(<>) :: (a1 -> a2 -> b1) -> (a1, a2) -> b1`

Infix currying operators used for convenience.

The operator convention is `(<>+)`, where the number of `>` + 1 is the order n of the n-tuple.

`ForSyDe.Atom.Utility` exports the constructors below. Please follow the examples in the source code if they do not suffice:

```
<>, <>>, <>>>, <>>>>, <>>>>>, <>>>>>>, <>>>>>>>, <>>>>>>>>
```

Example:

```
λ> (+) <> (1,2)
3
```

`($$) :: (a1 -> b1, a2 -> b2) -> (a1, a2) -> (b1, b2)`
Infix function application operator for tuples.

The operator convention is `($+)`, where the number of `$` is the order n of the n-tuple. For Applying a function on nontuples we rely on `$` provided by `Prelude`.

`ForSyDe.Atom.Utility` exports the constructors below. Please follow the examples in the source code if they do not suffice:

```
$$, $$$, $$$$, $$$$$, $$$$$$, $$$$$$$, $$$$$$$$, $$$$$$$$$
```

Example:

```
λ> ((+),(-)) $$ (1,1) $$ (2,2)
(3,-1)
```

4.17 References

Bird, Richard S. (1987). *An introduction to the theory of lists*. Springer. ISBN: 3642873766.

Bird, Richard and Oege de Moor (1997). *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-507245-X.

Fischer, Jörg, Sergei Gorlatch, and Holger Bischof (2003). "Foundations of Data-parallel Skeletons". English. In: *Patterns and Skeletons for Parallel and Distributed Computing*. Ed. by Fethi A. Rabhi and Sergei Gorlatch. Springer London, pp. 1–27. ISBN: 978-1-85233-506-9. DOI: 10.1007/978-1-4471-0097-3_1.

Halbwachs, Nicholas et al. (1991). "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9, pp. 1305–1320.

Kahn, Gilles and David MacQueen (1976). "Coroutines and networks of parallel processes". In:

Lee, Edward A and Alberto Sangiovanni-Vincentelli (1998). "A framework for comparing models of computation". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 17.12, pp. 1217–1229.

Reekie, Hideki John (1995). "Realtime Signal Processing". PhD thesis. School of Electrical Engineering.

Sander, I. and A. Jantsch (2006). "System Modeling and Transformational Design Refinement in ForSyDe (Formal System Design)". In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 23.1, pp. 17–32. ISSN: 0278-0070. DOI: 10.1109/TCAD.2003.819898. URL: http://dx.doi.org/10.1109/TCAD.2003.819898.

Skillicorn, D. B. (1994). *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press. DOI: 10.1017/CBO9780511526626.

Ungureanu, George and Ingo Sander (2017). "A layered formal framework for modeling of cyber-physical systems". In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1715–1720.

Index