Getting Started with FORSYDE-ATOM*

George Ungureanu

March 2017

Abstract

This document 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.

1 Goals

The main goals of this document 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.

*compiled with FORSYDE-ATOM v0.2.1

Contents

2 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, 2007.

This document has been created using literate programming. This means that all code shown in the listings is compilable and executable. There are two types of code listing found in this document. This style

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

shows *source code* as it is found in the implementation files. Notice that in-line API documentation is also shown as comments. 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>
```

3 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`.

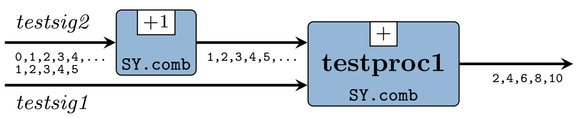


Figure 1: Simple process network as composition of processes

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. 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, "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.

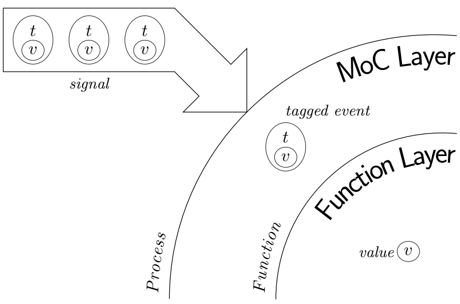


Figure 2: Layered structure of the processes in fig. 1

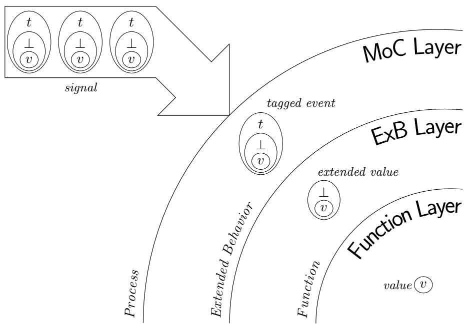


Figure 3: Layered structure of the processes describing absent events

As a last exercise for this section we would like to extend the behavior of the system in fig. 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. 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. 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. 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. 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.

3.1 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:

[1] ForSyDe-Atom v0.2.1

```
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
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
```

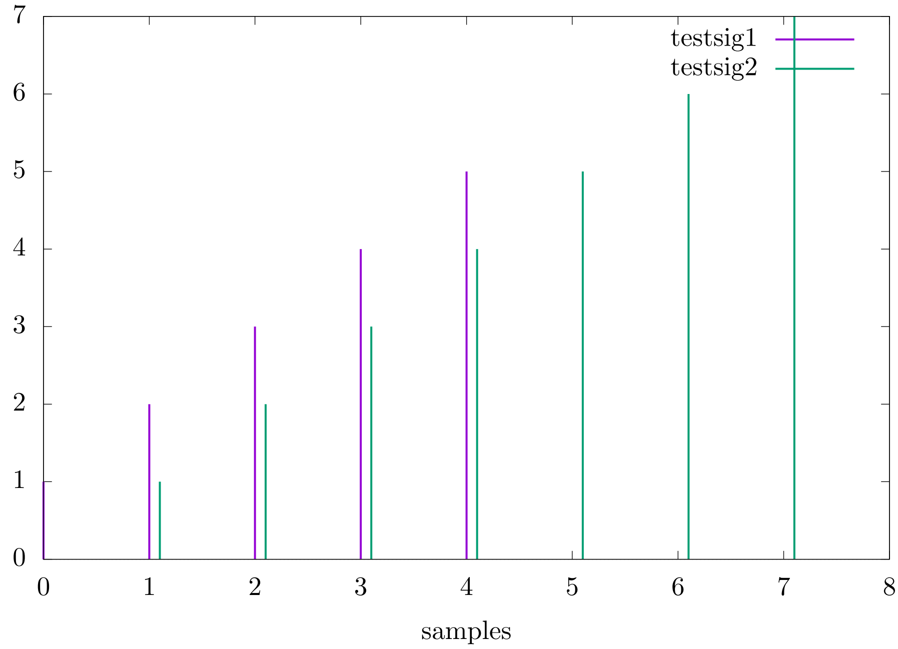
The function dumpDat dumps the data files in a path specified by the configuration object. Based on the `config` object instantiated earlier, after calling the following function you should see a new folder called `fig` in the current path, with two new `.dat` files.

```
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 3.1.

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

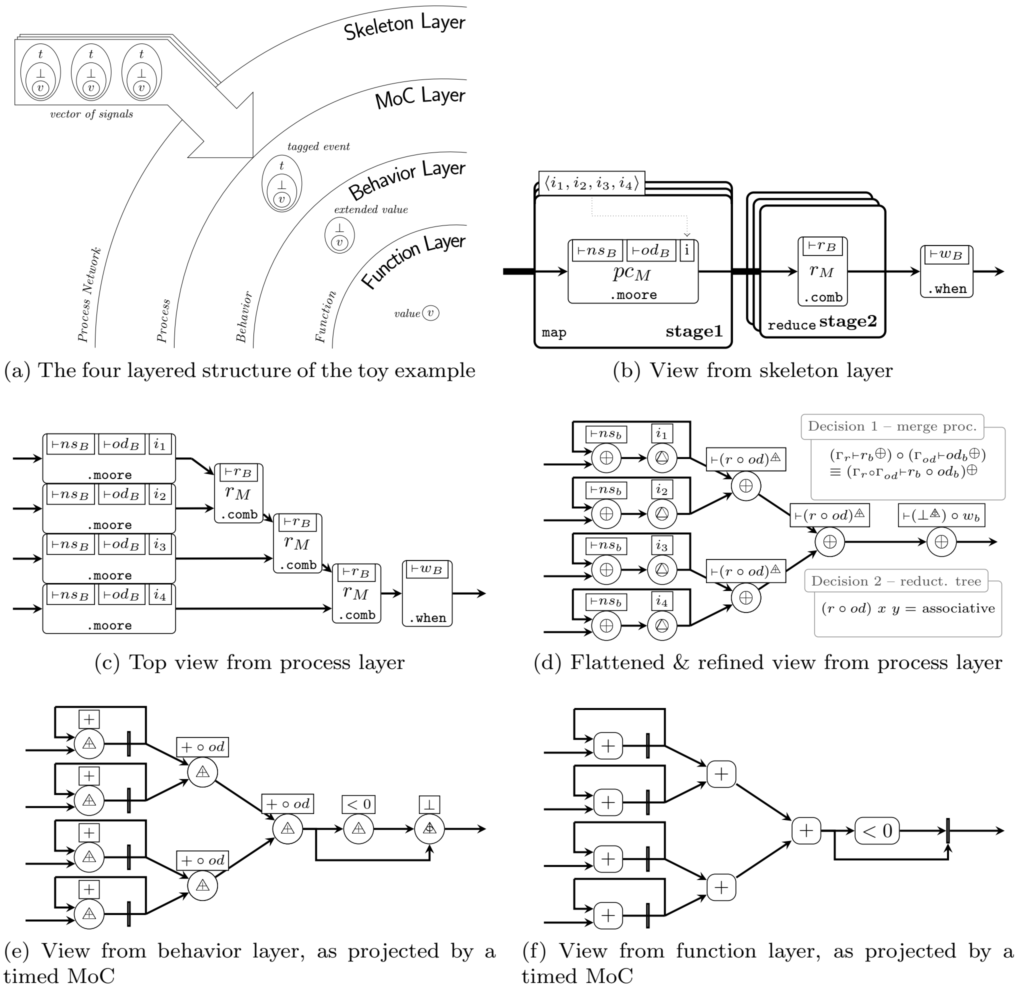
1.0	2.0	3.0	4.0	5.0			
0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0

Figure 4: SY signal plot in ForSyDe-LaTeX as a matrix of nodes

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.

4 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. 5b which exposes four layers, structured like in fig. 5a. This system is then fed vectors of signals describing different MoCs and its response is observed. In figs. 5c to 5f 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.



(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 5: Views and projections for the toy system

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. (1)–(5) and table 1. Do not bother much about this notation either, as this exact definition will appear in the code in a more "human readable" form.

$$
\begin{aligned}
&\texttt{toy} : \langle V \rangle \rightarrow \langle S \rangle \rightarrow S \\
&\texttt{toy}\langle i \rangle \langle s \rangle = \texttt{when}_{\text{M}}(\Gamma_w \vdash w_{\text{B}}) \circ \texttt{reduce}_{\text{S}}(r_{\text{M}}) \circ \texttt{map}_{\text{S}}(pc_{\text{M}}) \langle i \rangle \langle s \rangle
\end{aligned} \tag{1}
$$

where

$$\texttt{when}_{\text{M}}(\Gamma_w \vdash w_{\text{B}})(s) = ((\perp \triangle) \circ (\Gamma_w \vdash w_{\text{B}})) \oplus s \tag{2}$$

$$r_{\text{M}}(x, y) = \Gamma_r \vdash r_{\text{B}} \oplus (x, y) \tag{3}$$

$$\texttt{map}_{\texttt{s}}(pc_{\text{M}})\langle v \rangle \langle s \rangle = pc_{\text{M}} \diamond \langle v \rangle \diamondplus \langle s \rangle \tag{4}$$

$$pc_{\text{M}}(x, y) = \texttt{moore}_{\text{M}}(\Gamma_{ns} \vdash ns_{\text{B}}, \Gamma_{od} \vdash od_{\text{B}}, x)(y) \tag{5}$$

Table 1: CONTEXTS, FUNCTIONS AND INITIAL TOKENS FOR THE SYSTEM IN EQ. (1)

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

4.1 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.

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

```
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 . (\t -> 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. 5b and eq. (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 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 4.6 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}
```

4.2 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 4.1:

```
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
```

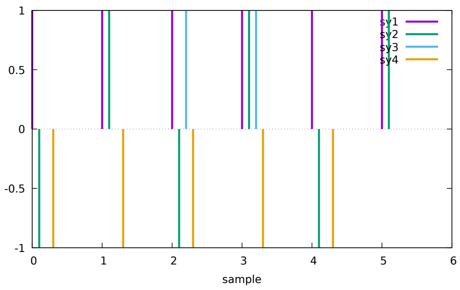
```
<{-1,0,1,2,3,4,5},{1,0,1,0,1,0,1},{-1,-1,-1,0,1,1},{1,0,-1,-2,-3,-4}>
λ> let latexIn = latexV 6 ["sy1","sy2","sy3","sy4"] vsy
λ> let latexS1 = latexV 7 ["sy1-1","sy2-1","sy3-1","sy4-1"] $ stage1SY isy vsy
λ> let gnuIn = plotV 6 ["sy1","sy2","sy3","sy4"] vsy
λ> let gnuS1 = plotV 7 ["sy1-1","sy2-1","sy3-1","sy4-1"]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

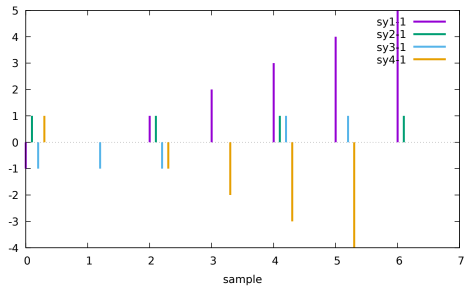
(a) `latexIn`

-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. 5b 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.

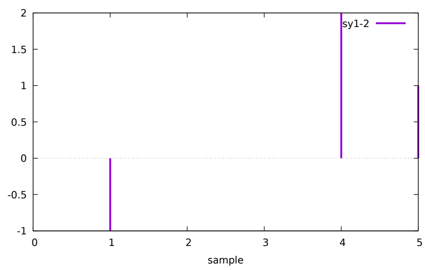
```
stage2SY :: V.Vector (SY.Signal (AbstExt Int))
         -> SY.Signal (AbstExt Int)
stage2SY = V.reduce rSY
  where
    rSY = SY.comb21 (ExB.res21 (+))
```

Again, let us print and plot the output signals using the test inputs and utilities defined in section 4.1.

```
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
```

0	-1	0	0	2	1

(a) `latexS2`



(b) `gnuS2`

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.

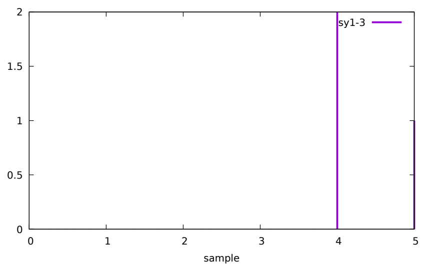
```
stage3SY :: SY.Signal (AbstExt Int)
         -> SY.Signal (AbstExt Int)
stage3SY = SY.filter (>=0)
>
toySY :: V.Vector (AbstExt Int)             -- ^ initial states
      -> V.Vector (SY.Signal (AbstExt Int)) -- ^ input
      -> SY.Signal (AbstExt Int)            -- ^ output
toySY i = stage3SY .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

We print and plot the system response to the test signals defined in section 4.1.

```
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`

4.3 DE instance

The DE instance of the `toy` looks exactly the same as the SY instance in section 4.2, 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.

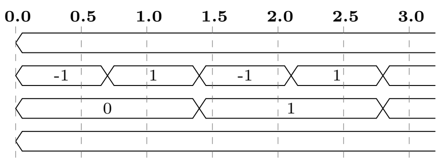
```
import ForSyDe.Atom
import ForSyDe.Atom.ExB.Absent (AbstExt)
import ForSyDe.Atom.ExB             as ExB
import ForSyDe.Atom.MoC.DE          as DE
import ForSyDe.Atom.Skeleton.Vector as V
```

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

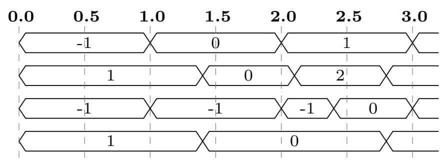
```
stage1DE :: V.Vector (TimeStamp, AbstExt Int)  -- ^ vector of initial states
         -> V.Vector (DE.Signal (AbstExt Int)) -- ^ vector of input signals
         -> V.Vector (DE.Signal (AbstExt Int)) -- ^ vector of output signals
stage1DE = V.farm21 pcDE
  where
    pcDE = DE.moore11 ns od
    ns   = 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.

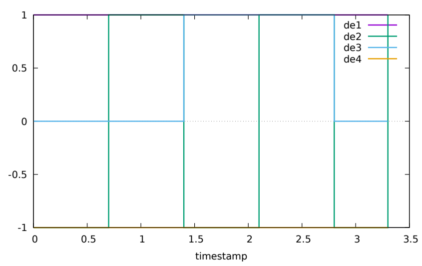
```
λ> ide
<(1s,-1),(1.399999999999s,1),(1s,-1),(1.399999999999s,1)>
λ> vde
<{ 1 @0s},{ -1 @0s, 1 @0.699999999999s, -1 @1.399999999999s, 1 @2.1s, -1 @2.799999999999s
    , 1 @3.5s},{ 0 @0s, 1 @1.399999999999s, 0 @2.799999999999s},{ -1 @0s}>
λ> fmap (takeS 6) $ stage1DE ide vde
<{ -1 @0s, 0 @1s, 1 @2s, 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
```



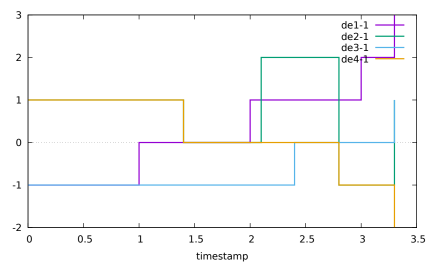
(a) latexIn



(b) latexS1



(c) gnuIn



(d) gnuS1

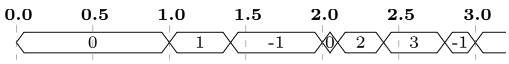
The second stage according to fig. 5b is also defined as a `reduce` network but this time we use the DE process constructor for the `comb` processes.

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

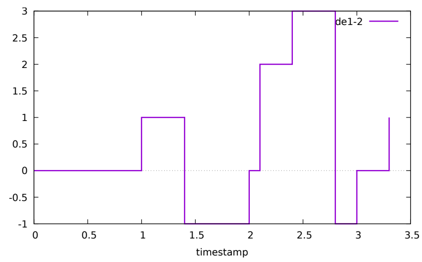
```
λ> let s2out = (stage2DE . stage1DE ide) vde
λ> takeS 10 s2out
{ 0 @0s, 1 @1s, -1 @1.399999999999s, 0 @2s, 2 @2.099999999998s, 3 @2.399999999999s, -1 @2
    .799999999998s, 0 @3s, 1 @3.399999999999s, 3 @3.499999999997s}
λ> let latexS2 = latex 3.3 ["de1-2","de2-2","de3-2","de4-2"] s2out
λ> let gnuS2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

```
stage3DE :: DE.Signal (AbstExt Int)
         -> DE.Signal (AbstExt Int)
stage3DE = deFilter (>=0)
```



(a) `latexS2`

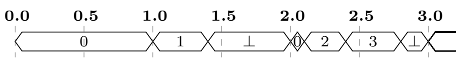


(b) `gnuS2`

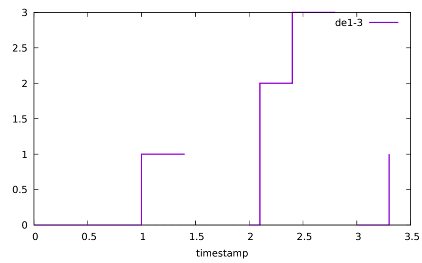
```
  where
    deFilter p s = DE.comb21 ExB.filter (predSig p s) s
    predSig  p s = DE.comb11 (ExB.res11 p) s

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`

4.4 CT instance

The CT instance of the `toy` looks exactly the same as the previous ones in sections 4.2 and 4.3, 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
```

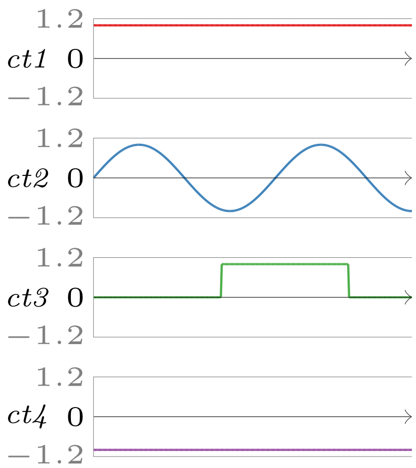
Again, `stage1` is a `farm` network of CT `moore` processes.

```
stage1CT :: V.Vector (TimeStamp, Time -> AbstExt Time)  -- ^ vector of initial states
         -> V.Vector (CT.Signal (AbstExt Time))         -- ^ vector of input signals
```

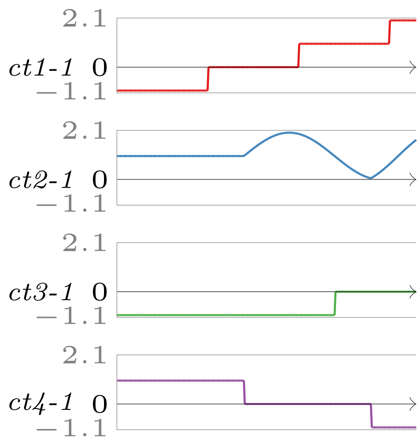
```
         -> V.Vector (CT.Signal (AbstExt Time))           -- ^ vector of output signals
stage1CT = V.farm21 pcCT
  where
    pcCT = CT.moore11 ns od
    ns   = ExB.res21 (+)
    od   = ExB.res11 id
```

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

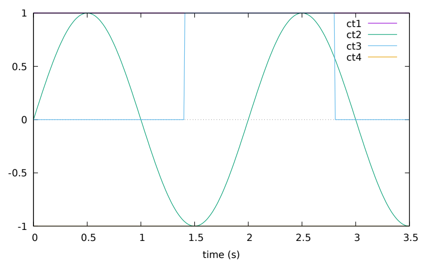
```
λ> let latexIn = latexV 3.5 ["ct1","ct2","ct3","ct4"] vct
λ> let latexS1 = latexV 3.5 ["ct1-1","ct2-1","ct3-1","ct4-1"] $ stage1CT ict vct
λ> let gnuIn = plotV 3.5 ["ct1","ct2","ct3","ct4"] vct
λ> let gnuS1 = plotV 3.5 ["ct1-1","ct2-1","ct3-1","ct4-1"] $ stage1CT ict vct
```



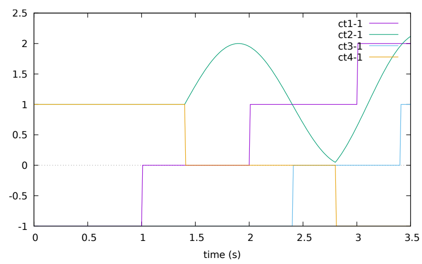
(a) `latexIn`



(b) `latexS1`



(c) `gnuIn`



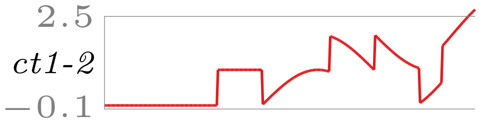
(d) `gnuS1`

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

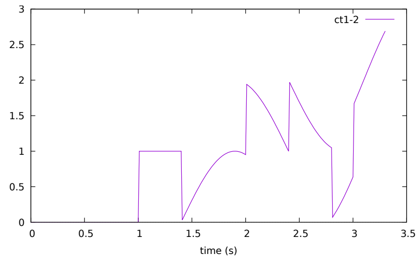
```
stage2CT :: V.Vector (CT.Signal (AbstExt Time))
         -> CT.Signal (AbstExt Time)
stage2CT = V.reduce rCT
  where
    rCT = CT.comb21 (ExB.res21 (+))
```

```
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

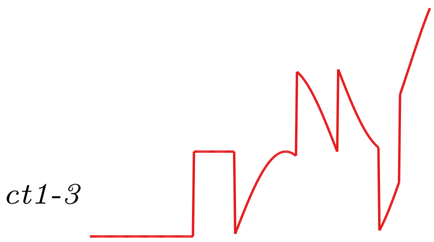


(b) gnuS2

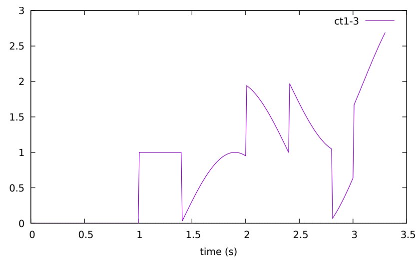
```
stage3CT :: CT.Signal (AbstExt Time)
         -> CT.Signal (AbstExt Time)
stage3CT = ctFilter (>=0)
  where
    ctFilter p s = CT.comb21 ExB.filter (predSig p s) s
    predSig  p s = CT.comb11 (ExB.res11 p) s

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

4.5 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 (\a 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 4.1:

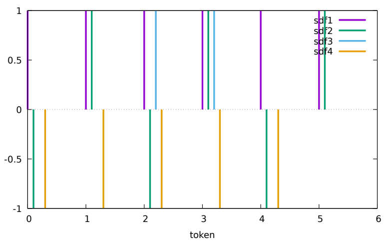
```
λ> isdf
<[-1],[1],[-1],[1]>
λ> vsdf
<{1,1,1,1,1,1},{-1,1,-1,1,-1,1},{0,0,1,1,0},{-1,-1,-1,-1,-1}>
λ> stage1SDF isdf vsdf
<{-1,1,3,5},{1,1,1,1},{-1,-1,1},{1,-1,-3}>
λ> let latexIn = latexV 6 ["sdf1","sdf2","sdf3","sdf4"] vsdf
λ> let latexS1 = latexV 7 ["sdf1-1","sdf2-1","sdf3-1","sdf4-1"] $ stage1SDF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

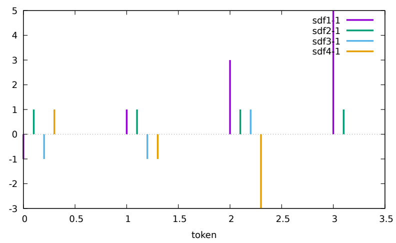
(a) `latexIn`

-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.

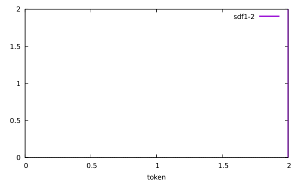
```
stage2SDF :: V.Vector (SDF.Signal (AbstExt Int))
         -> SDF.Signal (AbstExt Int)
stage2SDF = V.reduce rSDF
  where
    rSDF = SDF.comb21 ((1,1),1,rF)
    rF [x1] [y1] = [ExB.res21 (+) x1 y1]
```

Again, let us print and plot the output signals using the test inputs and utilities defined in section 4.1.

```
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
```

0.0 0.0 2.0

(a) `latexS2`



(b) `gnuS2`

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.

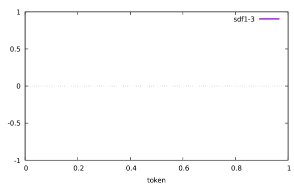
```
stage3SDF :: SDF.Signal (AbstExt Int)
         -> SDF.Signal (AbstExt Int)
stage3SDF = sdfFilter (>=0)
  where
    sdfFilter p s    = SDF.comb21 ((2,2),2,filterF) (predSig p s) s
    filterF pl sl    = zipWith ExB.filter pl sl
    predSig   p s    = SDF.comb11 (1,1,fmap (ExB.res11 p)) s
```

```
>
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`

4.6 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)
```

```haskell
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.

```haskell
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.

```haskell
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 (\[x1] [y1,y2] -> [ExB.res31 (\a b c -> a + b + c) x1 y1 y2])
odSDF = MoC.ctxt11 1     1 (\[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:

```haskell
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:

```haskell
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 (\[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.

```haskell
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:

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

The full definition of the `toy` system:

```haskell
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. 6.

```haskell
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. 6 are exactly the same as the ones presented in sections 4.2 to 4.5. 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.

-1.0			
1.0			
-1.0			
1.0			

(a) `iSDF`

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	

(b) `vSDF`

-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	

(c) `s1SDF`

0.0	0.0	2.0

(d) `s2SDF`

0.0	0.0

(e) `s3SDF`

-1.0
1.0
-1.0
1.0

(f) `iSY`

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	

(g) `vSY`

-1.0	0.0	1.0	2.0	3.0	4.0
1.0	0.0	1.0	0.0	1.0	0.0
-1.0	-1.0	-1.0	0.0	1.0	1.0
1.0	0.0	-1.0	-2.0	-3.0	-4.0

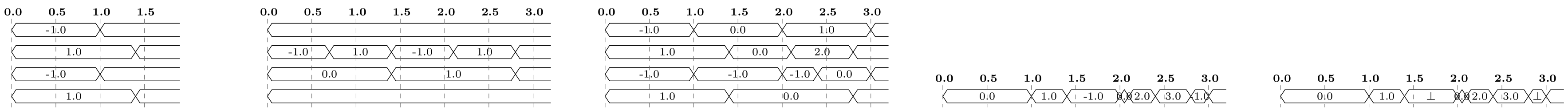
(h) `s1SY`

0.0	-1.0	0.0	0.0	2.0	1.0

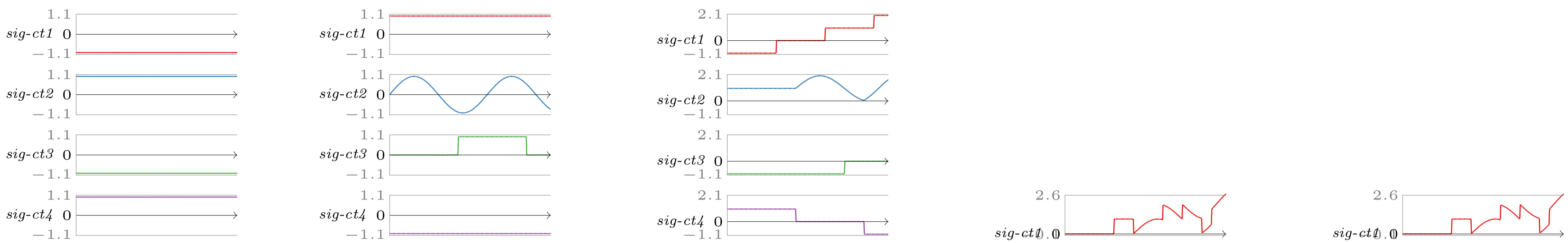
(i) `s2SY`

0.0	$\perp$	0.0	0.0	2.0	1.0

(j) `s3SY`



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



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

Figure 6: Inputs and outputs for the polymorphic `toy` system in section 4.6

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.

5 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:

```
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. 7.

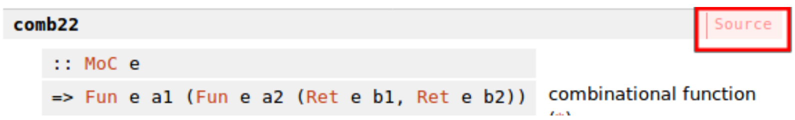


Figure 7: Screenshot from the API documentation. The link to the source code is marked with a red rectangle

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. 8.

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

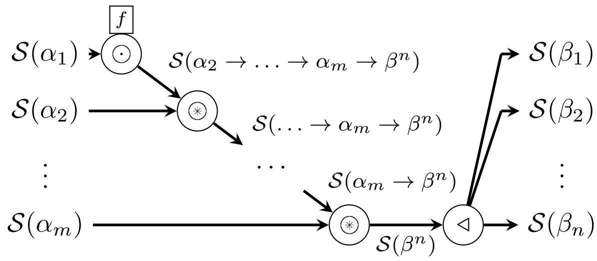


Figure 8: Composition of atoms forming the `comb` pattern

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

...
```

6 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.

References

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

Hutton, Graham (2007). *Programming in Haskell*. New York, NY, USA: Cambridge University Press. ISBN: 0521871727, 9780521871723.

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.

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.