Modeling Parallel Systems in ForSyDe-Atom*

George Ungureanu

November 2017

Abstract

This report 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 report is also meant to support the associated scientific publications with experimental results.

1 Goals

The reader is assumed to have been familiarized with the ForSyDe-Atom modeling framework. A good resource for that is the report "Getting started with ForSyDe-Atom" within the same repository. The main goals of this report 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**

*compiled with ForSyDe-Atom v0.2.2

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, 2016 or other recent books in Haskell.

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. 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>
```

```
{-# LANGUAGE PostfixOperators, TypeFamilies  #-}
```

3 An FFT network

This example shows how to instantiate an FFT butterfly process network using skeleton layer constructors operating on `Vector` types, and simulates the system's response to signals of different MoCs. It was inspired from **reekie95**

To be consistent with the current chapter, we shall separate the interconnection network expressed in terms of skeleton layer constructors, from the functional/behavioral/timing components further enforcing the practice of *orthogonalization* of concerns in system design. Also, to demonstrate the elegance of this approach, we shall construct two different functionally-equivalent systems which expose different propoerties: a) the butterfly pattern as a network of interconnected processes communicating through (vectors of) signals; b) the FFT as a process operating on a signal (of vectors). Each of these two cases will be fed with signals of different MoCs to further demonstrate the independence of the atom (i.e. constructor) composition from the semantics carried by atoms themselves.

```
module AtomExamples.Skeleton.FFT where
```

Following are the loaded modules. We import most modules qualified to explicitly show to which layer or class a certain constructor belongs to. While, from a system designer's perspective, loading the `Behavior` and `MoC` modules seems redundant since there exist equivalent helpers for each MoC for instantiating the desired processes, we do want to show the interaction between (and actually the separation of) each layer.

```
import           ForSyDe.Atom
import qualified ForSyDe.Atom.MoC             as MoC
import qualified ForSyDe.Atom.MoC.SY          as SY
import qualified ForSyDe.Atom.MoC.DE          as DE
import qualified ForSyDe.Atom.MoC.CT          as CT
import qualified ForSyDe.Atom.MoC.SDF         as SDF
import qualified ForSyDe.Atom.Skeleton.Vector as V
```

The `Data.Complex` module is needed for the complex arithmetic computations performed by a "butterfly" element. It provides the `cis` function for conversion towards a complex number, and the `magnitude` function for extracting the magnitude of a complex number.

```
import Data.Complex
>-- import Data.Ratio
>-- import System.Process
```

```
type C = Complex Double
```

3.1 The butterfly network

The Fast Fourier Transform (FFT) is an algorithm for computing the Discrete Fourier Transform (DFT), formulated by Cooley-Tukey as a "divide-and-conquer" algorithm computing (1), where N is the number of signal samples, also called *bins*.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad \text{where} \quad k = 0, ..., N-1 \tag{1}$$

This algorithm is often implemented in an iterative manner for reasons of efficiency. Figure **??** shows an example diagram for visualizing this algorithm, and we shall encode precisely this diagram as a network of interconnected processes (or functions).

We start by computing a vector of "twiddle" factors, which are complex numbers situated on the unit circle. For $n = 16$, the value of this vector is:

$$\langle W_{16}^0, W_{16}^4, W_{16}^2, W_{16}^6, W_{16}^1, W_{16}^5, W_{16}^3, W_{16}^7 \rangle \; quad \text{where} \quad W_N^m = e^{-2\pi m/N} \tag{2}$$

the actual vector is obtained by mapping the function `bW` over a vector of `indexes` and `bitreversing` the result.

$$\begin{aligned} &twiddles : \mathbb{N} \to \langle \mathbb{C} \rangle \\ &twiddles\ N = (\mathtt{reverse}_S \circ \mathtt{bitrev}_S \circ \mathtt{take}_S\ N/2)(\left(k \mapsto \frac{-2i\pi k}{N}\right) \Diamond \langle 0..\rangle) \end{aligned} \tag{3}$$

```
twiddles :: Integer → V.Vector C
twiddles bN = V.reverse $ V.bitrev $ V.take (bN `div` 2) $ V.farm11 bW V.indexes
  where bW k = (cis . negate) (−2 ∗ pi ∗ fromInteger (k − 1) / fromInteger bN)
```

$$\mathtt{fft}_S\ k\ vs = \mathtt{bitrev}_S((stage \diamondsuit kern) \diamondsuit vs) \tag{4}$$

where the constructors

$$stage\ N = \mathtt{concat}_S \circ (segment \diamondsuit twiddles) \circ \mathtt{group}_S\ N \tag{5}$$

$$segment\ t = \mathtt{unduals}_S \circ (butterfly\ t \Diamond) \circ \mathtt{duals}_S \tag{6}$$

$$butterfly\ w = ((x_0\ x_1 \mapsto x_0 + wx_1, x_0 - wx_1)\ \triangle) \oplus \tag{7}$$

```
fft :: (C → a → a → (a, a)) → Integer → V.Vector a → V.Vector a
fft butterfly k vs = (V.bitrev . V.pipe1 stage kern) vs
  where
    stage   w = V.concat . (V.farm21 segment (twiddles n)) . V.group w
    segment t = (V.unduals <>) . (V.farm22 (butterfly t) <>) . V.duals
    kern      = V.iterate k (∗2) 2 — segment lengths
    n         = V.length vs        — length of input
```

The `fft` network is instantiated with the function above. Its type signature is:

$$\mathtt{fft} : (\mathbb{C} \to \alpha^2 \to \alpha^2) \to \mathbb{N} \to \langle \alpha \rangle \to \langle \alpha \rangle \tag{8}$$

and, as it suggests, it is not concerned in the actual functionality implemented, rather it is just a skeleton (higher order function) which instantiates an interconnection network. Its arguments are:

- `vs` : $\langle \alpha \rangle$ the vector of input samples, where $L(\mathtt{vs}) = N$
- `k` : $\mathbb{N}$ the number of butterfly stages, which needs to satisfy the condition $2^{\mathtt{k}} = N$
- `butterfly` : $\mathbb{C} \to \alpha^2 \to \alpha^2$ which performs the computation suggested by Figure **??**, in different formats, depending on the test case.

```
butterfly1 bffunc w = MoC.comb22 (bffunc w)
```

The function wrapped by the `butterfly` network is `bffunc(U)` and needs to be specified differently depending on whether the MoC is timed or untimed. For timed MoCs, it is simply a function on values, but for untimed MoCs it needs to reflect the fact that it operates on multiple tokens, thus we express this in ForSyDe-Atom as a function on lists of values.

```
bffuncT w  x0   x1  = ( x0 + w * x1 ,  x0 - w * x1 )
bffuncU w [x0] [x1] = ([x0 + w * x1], [x0 - w * x1])
```

Now we can instantiate `fft` as a parallel process network with:

```
fft1SY  :: Integer → V.Vector (SY.Signal  C) → V.Vector (SY.Signal  C)
fft1DE  :: Integer → V.Vector (DE.Signal  C) → V.Vector (DE.Signal  C)
fft1CT  :: Integer → V.Vector (CT.Signal  C) → V.Vector (CT.Signal  C)
fft1SDF :: Integer → V.Vector (SDF.Signal C) → V.Vector (SDF.Signal C)

fft1SY  = fft (butterfly1          bffuncT )
fft1DE  = fft (butterfly1          bffuncT )
fft1CT  = fft (butterfly1          bffuncT )
fft1SDF = fft (butterfly1 (wrapped bffuncU))
  where wrapped bf w = MoC.ctxt22 (1,1) (1,1) (bf w)
```

3.2 Case 2: Signal of Vectors

The second case study treats `fft` as a function on vectors, executing with the timing semantics dictated by a specified MoC, i.e. wrapped in a MoC process constructor. This way, instead of exposing the parallelism at process network level, it treats it at data level, while synchronization is performed externally, as a block component.

For timed MoCs instantiating a process which performs `fft` on a vector of values is straightforward: simply pass the `bffunc` to the `fft` network, which in turn constitutes the function mapped on a signal by a `comb11` process constructor. We have shown in Section **??** how to systematically wrap the function on values with different behavior/-timing aspects. This time we will use the library-provided helpers to construct `comb11` processes with a default behavior.

```
butterfly2 w x0 x1 = ( x0 + w * x1 ,  x0 - w * x1 )
```

```
fft2SY  :: Integer → SY.Signal (V.Vector C) → SY.Signal (V.Vector C)
fft2DE  :: Integer → DE.Signal (V.Vector C) → DE.Signal (V.Vector C)
fft2CT  :: Integer → CT.Signal (V.Vector C) → CT.Signal (V.Vector C)

fft2SY k = MoC.comb11 (fft butterfly2 k)
fft2DE k = MoC.comb11 (fft butterfly2 k)
fft2CT k = MoC.comb11 (fft butterfly2 k)
```

– For untimed MoCs though, a more straightforward approach is to make use of the definition of actors as processes with a production/consumption rate for each output/input. In the case of SDF, we would simply have a process which consumes 2^k tokens from a signal (the input samples), and apply the `fft` function over them.

```
fft2SDF :: Integer → SDF.Signal C → SDF.Signal C
fft2SDF k = SDF.comb11 (2^k, 2^k, V.fromVector . fft butterfly2 k . V.vector)
```

4 Conclusion

This project is still under development...

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.