A Layered Formal Framework for Modeling of Cyber-Physical Systems

George Ungureanu and Ingo Sander

KTH Royal Institute of Technology, Sweden

March 30, 2017

- Our vision for this framework is to unify and incorporate state-of-the-art modeling techniques spanning from different research fields.
- This presentation tries to demonstrate that this vision is not only fully justified, but also can elegantly be applied if we respect some overarching principles.

A Typical Sight...

```
short f(short int dir,long m,double *x,double *y) {
  long n,i,i1,j,k,i2,l,l1,l2;
  double c1,c2,tx,ty,t1,t2,u1,u2,z;
  n = 1;
  for (i=0;i<m;i++)
    n *= 2; i2 = n >> 1; j = 0;
  for (i=0;i<n-1;i++) {
    if (i < j) {
      tx = x[i]; ty = y[i];
      x[i] = x[j]; y[i] = y[j];
      x[j] = tx; y[j] = ty;
    }
    k = i2;
    while (k <= j) {
      j -= k;
      k >>= 1;
    }
    j += k;
  }
  c1 = -1.0; c2 = 0.0; l2 = 1;
  for (l=0;l<m;l++) {
    l1 = l2; l2 <<= 1;
    u1 = 1.0; u2 = 0.0;
    for (j=0;j<l1;j++) {
      for (i=j;i<n;i+=l2) {
        i1 = i + l1;
        t1 = u1 * x[i1] - u2 * y[i1];
        t2 = u1 * y[i1] + u2 * x[i1];
        x[i1] = x[i] - t1;
        y[i1] = y[i] - t2;
        x[i] += t1;
        y[i] += t2;
      }
      z =  u1 * c1 - u2 * c2;
      u2 = u1 * c2 + u2 * c1;
      u1 = z;
    }
    c2 = sqrt((1.0 - c1) / 2.0);
    if (dir == 1)
      c2 = -c2;
    c1 = sqrt((1.0 + c1) / 2.0);
  }

  if (dir == 1) {
    for (i=0;i<n;i++) {
      x[i] /= n;
      y[i] /= n;
    }
  }
  return(TRUE);
}
```


- Often when we try to understand the specifications of a problem we end up in the situation to *"read the source code"* due to various reasons, but mainly due to the lack of documentation, or unclear specifications.
- Although this C code is well structured, one might have difficulties in understanding *"what it does"*...

A Typical Sight...

```
short j = threadIdx . x; short i = j /2; short k = j %2; short h = i /2; short l = j %4; short m = i %2;
for ( short s=1; s<=n ; s++){
    p = M/(1<<(2*s) ) ;
    x = 2*(h+(h/(1<<2*(n-s)))*(1<<2*(n-s ))*3);
    r_tmp = (h%(1<<2*(n-s)))*(1<<2*(s-1));
    x0 = x+l*p ;
    sign1= m*(-2)+1;
    r0 = r_tmp + k*(N/4);
    x1 = x0+sign1*2*p ;
    SB[ x0 ] = SA[ x1 ] + sign1*SA[ x0 ] ;
    SB[ x0+1] = SA[ x1+1] + sign1*SA[ x0+1];
    short inx1 = x0-m*2*p + m;
    short inx2 = x0 + (!m)*2*p ;
    SA[ inx1 ] = SB[ inx1 ] ;
    SA[ inx2+m] = sign1*SB[ inx2 ]*SROT[2*r0+m] + SB[ inx2+1]*SROT[2*r0 +(!m) ] ;
    __syncthreads ( ) ;
    sign2 = k*(-2)+1;
    r1 = r_tmp*2;
    x2 = x0+sign2*p ;
    SB[ x0 ] = SA[ x2 ] + sign2*SA[ x0 ] ;
    SB[ x0+1] = SA[ x2+1] + sign2*SA[ x0+1];
    short inx3 = x0-k*p + k ;
    short inx4 = x0 + (! k)*p ;
    SA[ inx3 ] = SB[ inx3 ] ;
    SA[ inx4+k ] = sign2*SB[ inx4 ]*SROT[2*r1+k ] + SB[ inx4+1]*SROT[2*r1 +(!k ) ] ;
    __syncthreads ( );
}
```


- ... it gets even worse when you go to specialized platforms. This CUDA code performs the same function as the previous, but it is impossible to read.
- Now think from the perspective of the machine: although a computer will politely execute the code given, it has no idea *"what"* it does.
- And since we have the machine's perspective in mind, a full system specification would not be of much help, would it? Usually specifications are written using internal conventions and are meant for engineers rather than parsers.
- So this leads us to the question...

The Problem

- What is the best way to describe systems in order to make sense for both humans (engineers) and machines (design automation tools)?

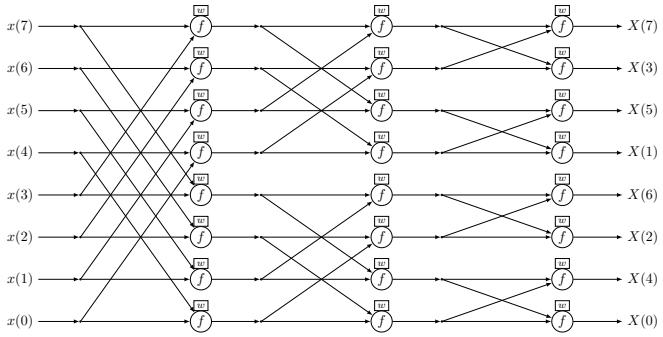
How can we describe systems understood by

humans

and

machines?

A Change of Perspective



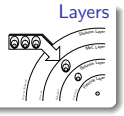
- ... what does *this* system perform? Evidently, the Discrete Fourier Transform using the Cooley-Tuckey algorithm.
- How did you, as humans understood its purpose? I presume due to the unmistakable butterfly pattern, which is taught in school in signal processing classes.
- The grand advantage of this specification format
  - it is not bound to any execution model. It describes *"what"* it does rather than *"how"* to do it.
  - if we provide clear semantics for the nodes and the edges, it is more likely that a machine is able to generate the previous code (or at lease platform code satisfies the correct functionality), than the other way around.
- An analyzable formal framework/language for application modeling is easier said than done! CPS: complex systems with a huge amount of interleaving concerns that need to be captured.
-

System Design Principles

Orthogonalization of concerns

computation · communication · function · behavior · contract · timing · synchronization

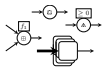
Layers



Capturing of concerns

- primitive building blocks
- clear core semantics
- side-effect free

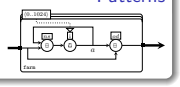
Atoms



Interactions of concerns

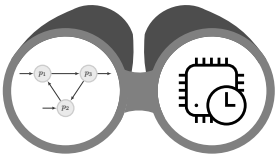
- composition & hierarchy
- interfaces

Patterns



- We propose three key principles that are crucial to just even dare tackling with the complexity of CPS.
- the first key principle is *orthogonalization of concerns*, i.e. separation of aspects in a CPS in order to handle complexity by the "divide-et-impera" rule.
  - for this, we introduce the concept of *layers* as separate environments for each aspect in order to exploit it
- another key principle is to correctly capture and express the concerns using an appropriate set of building blocks, enough to abstract the aspect's semantics.
  - we push orthogonalization to its limits by deconstructing the semantics of each layer to their core, in form of indivisible building blocks called *atoms*.
  - keep a reasonable number of symbols for handling complex behaviors.
- since by itself, separation is not enough in the context of interleaving concerns we need to express interaction
  - we define patterns for building complex behaviors from atoms.

The ForSyDe-Atom Modeling Framework



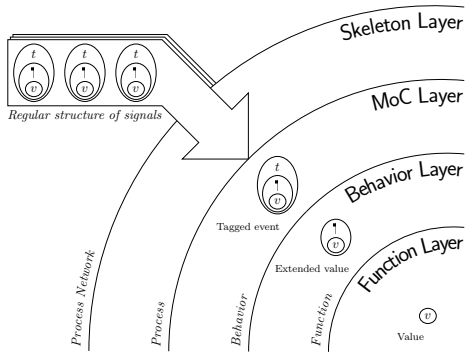
`https://github.com/forsyde`

ForSyDe-Atom

- EDSL in the functional programming language Haskell
- focus on capturing aspects of systems in a structured manner
- provide a proper front-end for formal analysis tools

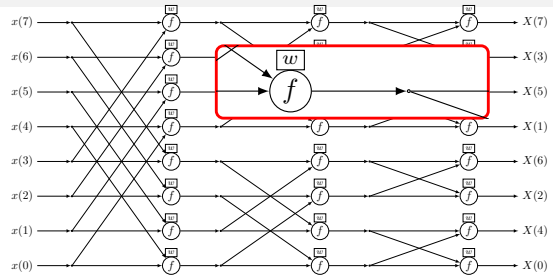
- we developed this framework under the name ForSyDe-Atom which is part of the ForSyDe project.

Layers in ForSyDe-Atom



- this is the layer structure presented in the paper: we analyzed CPS from the perspective of four concerns: the functionality, the behavior, the timing and synchronization and the structure of communication and computation.
- we make heavy use of concepts from functional programming for expressing and implementing this framework.
- the main idea is to "wrap" the system functionality (i.e. its data path) with layers abstracting different aspects. Each layer is associated with a data type which contains symbols and information relevant for the aspect we are trying to capture, and basically enable computation and analysis on that layer.
- I will go through these layers and demonstrate their usage on the previous FFT Example in a tutorial style...

The Function Layer: FFT Example

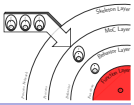


Functional description

$f(w, a, b) =$
$(a + wb, a - wb)$

ForSyDe-Atom code

```
f w a b =
  (a + w * b, a - w * b)
```



The Behavior Layer

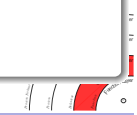
Example 1: Absent events

- synchronous languages
- $\top$ present and $\bot$ absent
- resolution $\Rightarrow$ clock calculus

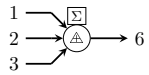
operands		resolution
$\top$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	*throw error*

Example 2: Multi-valued logic

- modeling CMOS : IEEE 1164 standard with 9 values logic
- digital modeling : 4-value logic subset *1, 0, Z, X*
- CAN transmission : SAE J1939 standard *T, F, Error, Not Installed*
- multiple sources : Belnap's relevance logic *true, false, both, neither*



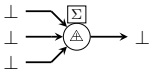
Behavior Atoms



The $\texttt{resolution}_B$ pattern

- "Lifts" a function in the behavior layer domain
- applies a resolution

Behavior Atoms



The $\texttt{resolution}_B$ pattern

- "Lifts" a function in the behavior layer domain
- applies a resolution

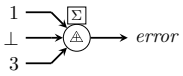
- operations on data (data path) + side-effect free
- in the FFT Example, the basic functional block is the butterfly function, captured by the block f.
  - two inputs, two outputs
  - takes a parameter w which is the "twiddle factor"
  - the mathematical formulation of f can be expressed in the left equation.
- the nice thing with choosing a functional programming language as host is that the code looks pretty similar like the maths.
- the function f tells what happens with the input data, but does not tell what happens if the data does not arrive, is corrupted, or in any case, if some error occurred and was propagated throughout the system. This set of events and their responses are captured in a layer above...

- tries to answer the question "what happens when a special event occurs"
- "special" events usually describe a state of the system which cannot be captured by the set of values
- example events: absent events or multi-valued logic.

- we need to extend the value types with symbols with clearly-defined semantics, and define *composable atomic operations* on those symbols.
- resolution pattern: simulates a normal evaluation cycle in the synchronous language Lustre.

- we need to extend the value types with symbols with clearly-defined semantics, and define *composable atomic operations* on those symbols.
- resolution pattern: simulates a normal evaluation cycle in the synchronous language Lustre.

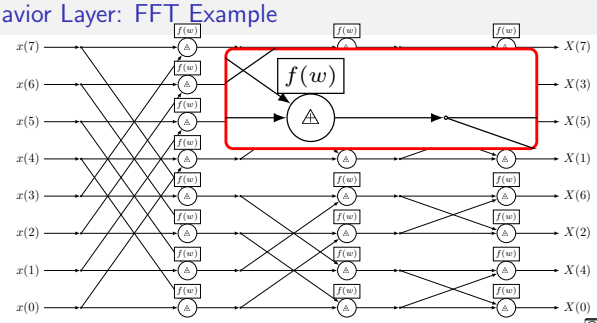
Behavior Atoms



The $\texttt{resolution}_B$ pattern

- "Lifts" a function in the behavior layer domain
- applies a resolution

The Behavior Layer: FFT Example

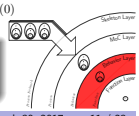


Functional description

$$f_B(w, a_B, b_B) = f(w) \triangleq (a_B, b_B)$$

ForSyDe-Atom code

```
import ForSyDe.Atom.Behavior as B
fB w = B.resolution22 (f w)
```



The MoC Layer

- **Models of Computation** founded in the tagged signal model[1]

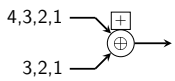
  "a common meta model for describing properties of concurrent systems (...)"

- Atoms *enabled by signals*
  ⇒ agnostic of their own MoC
- Supported MoCs:
  - Synchronous Data Flow (SDF)
  - Synchronous (SY)
  - Discrete Event (DE)
  - Continuous Time (CT)

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

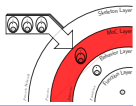
[1] E.A. Lee and A. Sangiovanni-Vincentelli. "A framework for comparing models of computation". In: *IEEE TCAD* 17.12 (Dec. 1998), pp. 1217–1229.

The Synchronous MoC



The $\texttt{comb}_M$ pattern

- synchronizes the input signals
- "Lifts" a function and applies it on each event



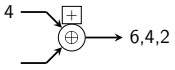
- we need to extend the value types with symbols with clearly-defined semantics, and define *composable atomic operations* on those symbols.
- resolution pattern: simulates a normal evaluation cycle in the synchronous language Lustre.

- coming back to the FFT Example, in order to "wrap" function f with a behavior, we use the *default* behavior atom, expressed with the operator $\triangleq$.
  - symbol convention in ForSyDe–Atom is inner operator for atom, outer shape for layer.
- both the mathematical expression and the figure suggest that $\triangleq$ takes the previously defined $f(w)$ as argument, and applies it on the wrapped inputs.
- the code makes use of partial application which avoids redundant argument calls.
- this system now expresses functionality and behavior, but tells nothing of how the data is handled upon arrival, or how the data even arrives. For this we need to capture timing and synchronization aspects defined in a layer above...

- MoC = classes of behaviors dictating the semantics of computation and concurrency in a network of processes.
- model describes signals as "sets of tagged events", where the set of tags itself is a partial order. This means that in our framework, the tags and the set constructors will play the main role and will act as enablers for the MoC layer.
- like all atoms in our framework, MoC atoms themselves are agnostic of the semantics they are implementing, and are presented as interfaces. The ones enabling atoms (and basically overloading the interfaces with the actual MoC semantics) are signals, i.e. a SY signal will overload an atom with SY semantics.

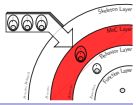
- One example is the SY MoC which is one of the most widely-used MoCs due to its similarities with the RTL logic.
- assumes computation is performed with 0 delay, and happens at certain synchronization points, where data is assumed to be available.
- here you see an example of a combinatorial process (pattern, it can be further be decomposed), which synchronizes two signals, applying function (+) on each pair of synchronous events.

The Synchronous MoC

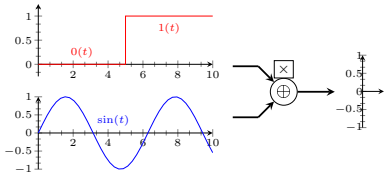


The comb_M pattern

- synchronizes the input signals
- "Lifts" a function and applies it on each event

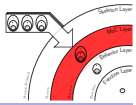


The Continuous MoC

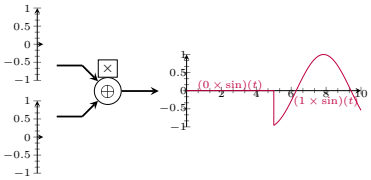


The comb_M pattern

- synchronizes the input signals
- "Lifts" a function and applies it on each event



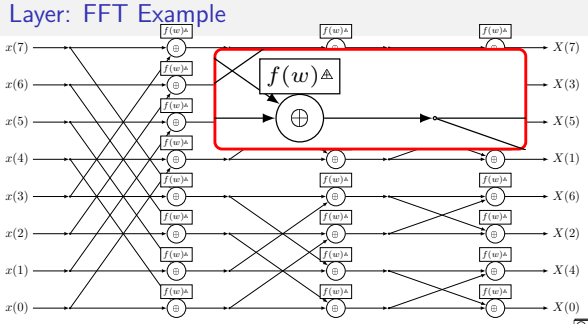
The Continuous MoC



The comb_M pattern

- synchronizes the input signals
- "Lifts" a function and applies it on each event

The MoC Layer: FFT Example

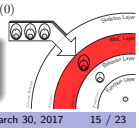


Functional description

$f_M(w, sa_M, sb_M) =$
$f_B(w) \oplus (sa_M, sb_M)$

ForSyDe-Atom code

```
import ForSyDe.Atom.MoC as M
fM w = M.comb22 (fB w)
```



- One example is the SY MoC which is one of the most widely-used MoCs due to its similarities with the RTL logic.
- assumes computation is performed with 0 delay, and happens at certain synchronization points, where data is assumed to be available.
- here you see an example of a combinatorial process (pattern, it can be further be decomposed), which synchronizes two signals, applying function $(+)$ on each pair of synchronous events.

- The CT MoC assumes that events are associated with an explicit tag, suggesting that they can "happen" at any physical instant in time.
- the combinatorial process' job is to synchronize these instants and output a signal respecting the time behavior.
- ForSyDe 's CT implementation is particularly useful thanks to the functional host. This way we can express functions over time as events, rather than sampled values. In this case the first signal has two events, and the second only one event. The output will be expressed as a function over time as well, and it will be evaluated only when plotted or sampled explicitly.

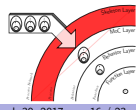
- The CT MoC assumes that events are associated with an explicit tag, suggesting that they can "happen" at any physical instant in time.
- the combinatorial process' job is to synchronize these instants and output a signal respecting the time behavior.
- ForSyDe 's CT implementation is particularly useful thanks to the functional host. This way we can express functions over time as events, rather than sampled values. In this case the first signal has two events, and the second only one event. The output will be expressed as a function over time as well, and it will be evaluated only when plotted or sampled explicitly.

- coming back to the FFT Example, enhancing f_B with a timing behavior is straightforward: we use the $\oplus$ pattern, which takes the previous f_B as argument, and now we can assume that the graph edges are signals of events of extended values.
- following the principle of function overloading associated with polymorphic types, two input SY signals would transform the $\oplus$ process into as SY process, whereas two CT signal would transform it into a CT process, a.s.o.
- while having introduced the needed blocks for describing and simulating CPS, modeling a FFT with only MoC layer elements (processes and signals) is not particularly great, nor scalable.
- we can easily observe that most interconnections follow repetitive patterns. Thankfully there is such a layer which exploits recursive structures...

The Skeleton Layer

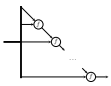
- *Algorithmic Skeletons*[2]
  - formalization of regular/recursive patterns
  - parallel computation and communication
- Algebra of types: *category theory*[3]
  - *types* expose parallelism
  - functions that exploit this potential
  - transformational framework ⇔ equational reasoning

[2] Murray I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.
[3] Richard Bird and Oege de Moor. *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

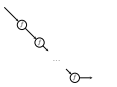


Computation Patterns

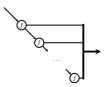
$\texttt{farm}_S = \diamondplus$



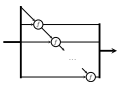
$\texttt{reduce}_S = \diamondsuit$



$\texttt{pipe}_S = \diamondsuit = (\circ)^{\diamondsuit}$



$\texttt{recur}_S = (f^{\diamondsuit}) \diamondplus \texttt{inits}_S$



$\texttt{prefix}_S = (f^{\diamondsuit}) \diamondplus \texttt{inits}_S$

Communication Patterns

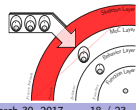
$\texttt{odds}_S$

$\texttt{evens}_S$

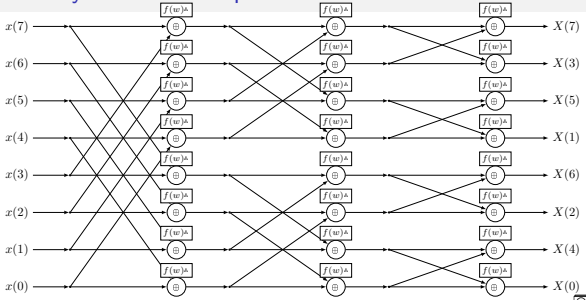
$\texttt{bitrev}_S$

$\texttt{concat2}_S$

$\texttt{split}_S$



The Skeleton Layer: FFT Example

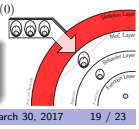


Functional description

$$f_M(w, sa_M, sb_M) = f_B(w) \oplus (sa_M, sb_M)$$

ForSyDe-Atom code

```
import ForSyDe.Atom.MoC as M
fM w = M.comb22 (fB w)
```



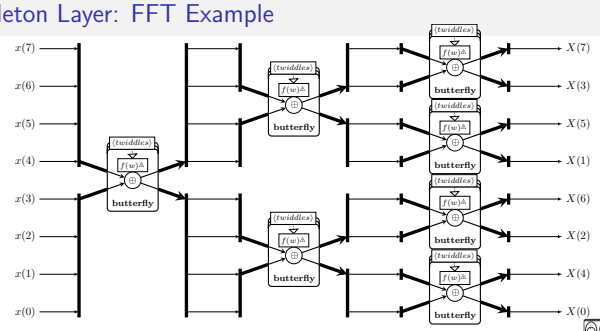
- This layer is concerned with algorithmic skeletons = high level structures capturing computation/communication which has the potential to be executed in parallel.
- Algorithmic skeletons are founded in an algebra of types called *category theory*, which describes a class of types that through their construction they have the potential to be evaluated in parallel.
  - example: *vector*
- This theory also provides a framework based on equational reasoning for semantic preserving transformations from specification model to efficient implementations.

- The *factorization* theorem states that a function on a categoric type is a catamorphism iff it can be expressed in terms of a *map* and a *reduce*.
- This theorem justifies the choice of these two skeletons as atoms in the ForSyDe–Atom framework.
- We describe an extensive library of patterns in terms of these two atoms.

- The patterns do not necessarily need to describe parallel computation. Communication patterns such as permutations can also be defined in terms of *map* and *reduce*.
- Instead of wrapping an arbitrary function, these patterns operate on the type constructors themselves.

- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. first, we notice that all the processes in our FFT network are replicas of the same block f_M. While the unequal segmentation at each stage forbids us to group all processes in a column into a `farm` pattern, we can do that at least for each stage.
  2. the *butterfly* pattern now takes a vector of twiddles as argument and distributes it across processes. It also now operates on two vectors of signals.

The Skeleton Layer: FFT Example



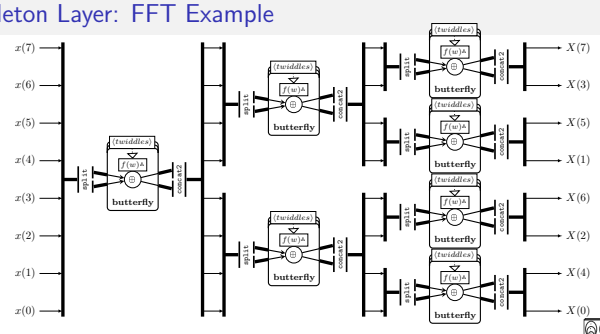
Functional description

$$butterfly_S(w, vsa_S, vsb_S) = f_M \Diamond \langle twiddles \rangle \Diamond (vsa_S, vsb_S)$$

ForSyDe-Atom code

```
import ForSyDe.Atom.Skeleton.Vector as V
butterfly = V.farm32 fM twiddles
```


The Skeleton Layer: FFT Example



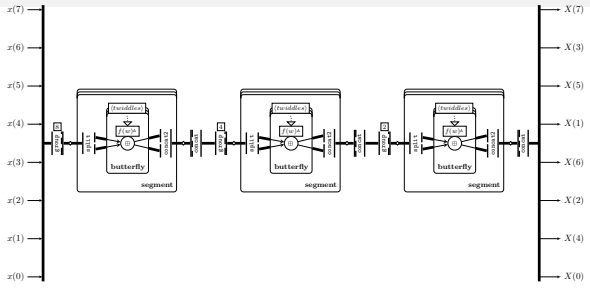
Functional description

$$segment_S(vs) = (\texttt{cat}_S \circ butterfly_S \circ \texttt{split}_S)(vs)$$

ForSyDe-Atom code

```
segment = V.concat2 . butterfly
        . V.split
```


The Skeleton Layer: FFT Example



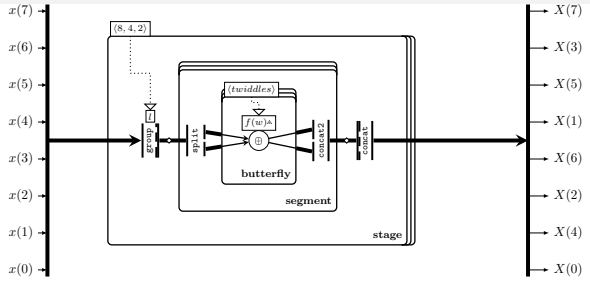
Functional description

$$stage_S(l, vs) = (\texttt{concat}_S \circ (segment_S^{\Diamond}) \circ \texttt{group}_S(l))(vs)$$

ForSyDe-Atom code

```
stage l = V.concat . V.farm11 segment .
    V.group l
```


The Skeleton Layer: FFT Example



Functional description

$$\mathit{fft}_S(vs) = stage_S \Diamond (\langle 2, 4, 8 \rangle, vs)$$

ForSyDe-Atom code

```
fft = V.pipe2 stage (vector[2,4,8])
```

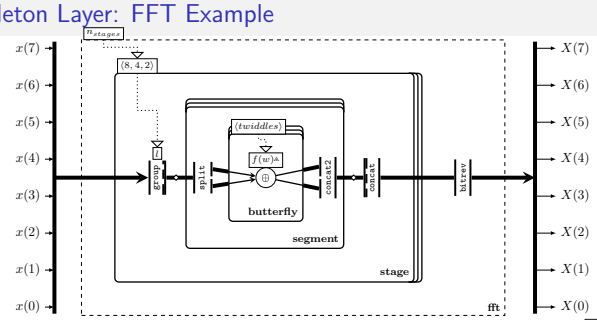

- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. we now need to capture the fact that for each segment the vector is split and merged back do form the butterfly communication pattern.
  2. this can be easily done by adding the `split` and `cat` patterns. But now we can observe that for each stage, the new split-butterfly-cat is the same, and it is replicated with a factor of $2^{(n-1)}$

- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. this can be easily done by adding the `split` and `cat` patterns. But now we can observe that for each stage, the new split-butterfly-cat is the same, and it is replicated with a factor of $2^{(n-1)}$
  2. we can further capture this property by embedding each FFT stage into a `farm` pattern, and making sure we group the input vectors into smaller vectors of a specified length.

- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. what we are left with is a pipeline of `N` stages, where each stage differs from the others only through its partition size for the input vector. Well, this partition is just a parameter taken by the $\texttt{group}_S$ pattern, thus we can capture the whole pipeline using a $\texttt{pipe}_S$ pattern which distributes the partition sizes to each stage accordingly.
  2. one last addition we make is applying the bit-reversal permutator pattern on the output, in order to get the FFT bins in order. We can also generate the vector of lengths from one parameter, which is the number of stages fro the FFT network.

- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. what we are left with is a pipeline of `N` stages, where each stage differs from the others only through its partition size for the input vector. Well, this partition is just a parameter taken by the $\texttt{group}_S$ pattern, thus we can capture the whole pipeline using a $\texttt{pipe}_S$ pattern which distributes the partition sizes to each stage accordingly.
  2. one last addition we make is applying the bit-reversal permutator pattern on the output, in order to get the FFT bins in order. We can also generate the vector of lengths from one parameter, which is the number of stages fro the FFT network.

The Skeleton Layer: FFT Example

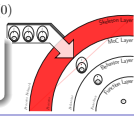


Functional description

$$\mathit{fft}_S(N, vs) = \texttt{bitrev}_S \circ \mathit{stage}_S \diamond (\texttt{widths}_S(N), vs)$$

ForSyDe-Atom code

```
fft n = V.bitrev .
        V.pipe2 stage (widths n)
```



- Coming back to the FFT Example. We need to make use of these computation/communication patterns to reach a simpler, parameterizable form.
  1. what we are left with is a pipeline of `N` stages, where each stage differs from the others only through its partition size for the input vector. Well, this partition is just a parameter taken by the $\texttt{group}_S$ pattern, thus we can capture the whole pipeline using a $\texttt{pipe}_S$ pattern which distributes the partition sizes to each stage accordingly.
  2. one last addition we make is applying the bit-reversal permutator pattern on the output, in order to get the FFT bins in order. We can also generate the vector of lengths from one parameter, which is the number of stages fro the FFT network.

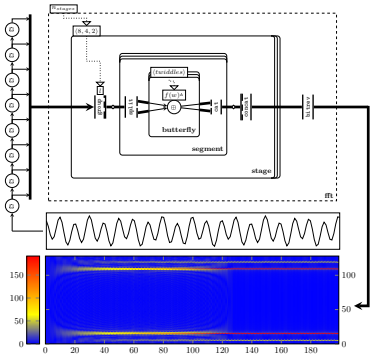
FFT Example : Code


```
1  import ForSyDe.Atom.Behavior as B
2  import ForSyDe.Atom.MoC as M
3  import ForSyDe.Atom.Skeleton.Vector as V
4
5  fft n = V.bitrev . V.pipe2 stage (widths n)
6    where
7      stage l    = V.concat . V.farm11 segment . V.group
8      segment    = (V.concat2 <>) . (butterfly <>) . V.split
9      butterfly = V.farm32 fM twiddles
10     fM w       = M.comb22 (B.default22
11                    (\a b -> a + w * b, a - w * b))
```

Code available at https://github.com/forsyde/forsyde-atom

- ...and we were able to do that in just 11 lines of code (including the library imports).
- $+$ 3 additional lines for defining `twiddles` and `widths` (found in the public repository).
- coming back to the original question which drove our research:
  - how can this code be useful for humans?
    - well-structured, red keywords are patterns and the prefix is their layer.
    - parametrizable patterns: avoid boilerplate code.
    - associated with a visual structure which can be expanded/collapsed and projected to show different aspects of the system.
  - how can this code be useful for machines?
    - each pattern can be deconstructed into a small set of symbols with known and well-defined semantics.
    - translates into an intermediate structure that enables design transformations and design space exploration.
    - acts as an executable specification...

FFT Example : Executable Specification



- .. an example simulation testbench for the previous code
- describe one double-sine input signal of a certain MoC (say SY).
- inject this signal into a network of delay processes, to get the correct vector of input samples.
- we instantiate a 128-bin FFT (i.e. $N = 7$ stages).
- the output shows the two stable sine components after the 128th sample, i.e. after the delay network has been filled.

Conclusions & Future Work

Framework for modeling CPS

- frontend enabling formal design flows, driven by:
  - **orthogonalization** of concerns: four separate **layers** of interleaving concerns.
  - **capturing** of concerns: **atoms** as symbols with clear semantics.
  - **interaction** of concerns: **patterns** of atom as hierarchical blocks.
- step-by-step demonstration on a typical FFT Example.

ForSyDe–Atom

- shallow-EDSL, proof-of-concept for the presented framework.
- publicly available at `https://github.com/forsyde/forsyde-atom`

Future work

- development of existing layers and exploration of new aspects (e.g. validation, adaptivity).
- deep-EDSL and full compiler flow.