Design of Sensor Signal Processing with ForSyDe

Modeling, Validation and Synthesis

George Ungureanu
School of EECS
KTH Royal Institute of Technology
ugeorge@kth.se

Timmy Sundström
Business Area Aeronautics
Saab AB

Anders Åhlander
Business Area Surveillance
Saab AB

Ingo Sander
School of EECS
KTH Royal Institute of Technology

Ingemar Söderquist
Business Area Aeronautics
Saab AB

Version 0.2

Version History:

0.1	2019/07/08	first released version
0.2	2019/08/09	added section “Model Synthesis to VHDL”

This report was partially funded by the Swedish Governmental Agency for Innovation Systems, NFFP7 project: Correct . by construction design methodology #2017-04892

Design of Sensor Signal Processing with ForSyDe

George Ungureanu
ugeorge@kth.se

Timmy Sundström
timmy.sundstrom@saabgroup.com

Anders Åhlander
anders.ahlander@saabgroup.com

Ingo Sander
ingo@kth.se

Ingemar Söderquist
ingemar.soderquist@saabgroup.com

Abstract

This document serves as a report and as a step-by-step tutorial for modeling, simulating, testing and synthesizing complex heterogeneous systems in ForSyDe, with special focus on parallel and concurrent systems. The application under test is a radar signal processing chain for an active electronically scanned array (AESA) antenna provided by Saab AB. Throughout this report the application will be modeled using several different frameworks, gradually introducing new modeling concepts and pointing out similarities and differences between them.

Contents

1 Introduction

In order to develop more cost-efficient implementation methods for complex systems, we need to understand and exploit the inherent properties derived from the specification of the target applications and, based on these properties, be able to explore the design space offered by alternative platforms. Such is the case of the application studied in this report: the active electronically scanned array (AESA) radar is a versatile system that is able to determine both position and direction of incoming objects, however critical parts of its signal processing has significant demands on processing and memory bandwidth, making it well out-of reach from the general public usage. We believe that a proper understanding of the temporal and spatial properties of the signal processing chain can lead to a better exploration of alternative solutions, ideally making it an affordable appliance in the context of current technology limitations. Nevertheless, expressing behaviors and (extra-functional) properties of systems in a useful way is far from a trivial task and it involves respecting some key principles:

- the language(s) chosen to represent the models need(s) to be *formally defined* and *unambiguous* to be able to provide a solid foundation for analysis and subsequent synthesis towards implementation.
- the modeling paradigm should offer the *right* abstraction level for capturing the *needed* properties (Lee 2015). An improper model might either abstract away essential properties or over-specify them in a way that makes analysis impossible. In other words it is the engineer's merit to find the right model for the right "thing being modeled".
- the models, at least during initial development stages, need to be *deterministic* with regard to defining what *correct* behavior is (Lee 2018).
- at a minimum, the models need to be *executable* in order to verify their conformance with the system specification. Ideally they should express operational semantics which are traceable across abstraction levels, ultimately being able to be synthesized on the desired platform (Sifakis 2015).

ForSyDe is a design methodology which envisions "correct-by-construction system design" through formal or rigorous methods. Its associated modeling frameworks offer means to tackle the challenges enumerated above by providing well-defined composable building blocks which capture extra-functional properties in unison with functional ones. ForSyDe-Shallow is a domain specific language (DSL) shallow-embedded in the functional programming language Haskell, meaning that it can be used only for modeling and simulation purposes. It introduced the concept of *process constructors* (Sander and Jantsch 2004) as building blocks that capture the semantics of computation, concurrency and synchronization as dictated by a certain model of computation (MoC). ForSyDe-Atom is also a shallow-embedded (set of) DSL which extends the modeling concepts of ForSyDe-Shallow to systematically capture the interacting extra-functional aspects of a system in a disciplined way as interacting *layers* of minimalistic languages of primitive operations called *atoms* (Ungureanu and Sander 2017). ForSyDe-Deep is a deep-embedded DSL implementing a synthesizable subset of ForSyDe, meaning that it can parse the structure of process networks written in this language and operate on their abstract syntax: either simulate them or further feed them to design flows. Currently ForSyDe-Deep is able to generate GraphML structure files and synthesizable VHDL code.

This documents presents alternatives ways to modelling the AESA radar signal processing chain and, using these models, gradually introducing one concept at a time and pointing towards reference documentation. The final purpose is to refine, synthesize, and replace parts of the behavioral model down to VHDL implementation on FPGA hardware platforms, and co-simulate these design artifacts along with the initial high-level model. The report itself is written using literate programming, which means that all code snippets contained are *actual compiled code* alternating with documentation text. Following the report might be difficult without some initial clarification. The remaining parts of section 1 will present a guide to using this document, as well as an introduction to the AESA application. In section 2 a high-level, functionally complete ForSyDe-Atom model of the application is thoroughly presented with respect to the specification, and tested against a set of known input data. In section 3 an equivalent model written in ForSyDe-Shallow is briefly presented and tested, to show the main similarities and differences between the two modeling APIs. In section 4 is introduced the concept of property checking for the purpose of validation of ForSyDe designs. We formulate a set of properties in the QuicCheck DSL for each component of the AESA model which are validated against a number of randomly-generated tests. In section 5.1 we focus on refining the behavior of the initial (high-level) specification model to lower level ones, more suitable for (backend) implementation synthesis, followed by section 5.3 where we formulate new properties for validating some of these refinements. All refinements in section 5 happen

in the domain(s) of the ForSyDe-Atom DSL. In section 6 we switch the DSL to ForSyDe-Deep, which benefits from automatic synthesis towards VHDL: in each subsection the refinements are stated, the refined components are modeled, properties are formulated to validate them, and in sections 6.3, 6.4 VHDL code is generated and validated.

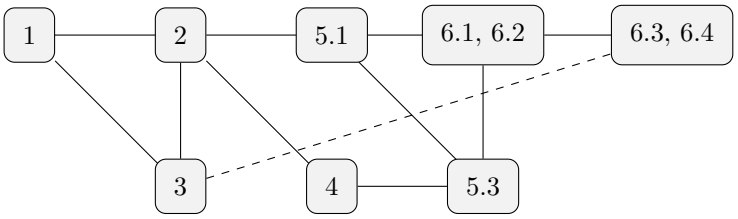


Figure 1: Reading order dependencies

Figure 1 depicts a reading order suggestion, based on information dependencies. The dashed line between section 3 and sections 6.3, 6.4 suggests that understanding the latter is not directly dependent on the former, but since ForSyDe-Deep syntax is derived from ForSyDe-Shallow, it is recommended to get acquainted with the ForSyDe-Shallow syntax and its equivalence with the ForSyDe-Atom syntax.

1.1 Application Specification

An AESA, see picture below, may consist of thousands of antenna elements. The relative phases of the pulses of the antenna's different antenna elements can be set to create a constructive interference in the chosen main lobe bearing. In this way the pointing direction can be set without any moving parts. When receiving, the direction can be steered by following the same principle, as seen the Digital Beam Former below. One of the main advantages of the array antennas is the capacity to extract not only temporal but also spatial information, i.e. the direction of incoming signals.

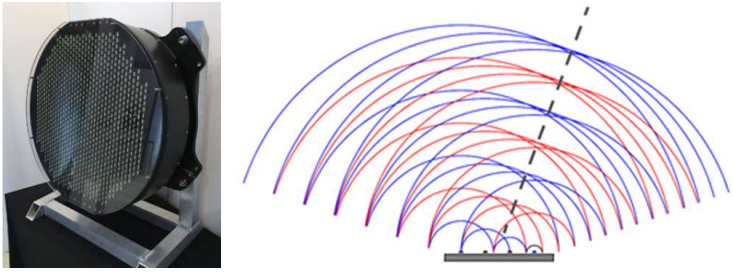


Figure 2 shows a simplified radar signal processing chain that is used to illustrate the calculations of interest. The input data from the antenna is processed in a number of steps.

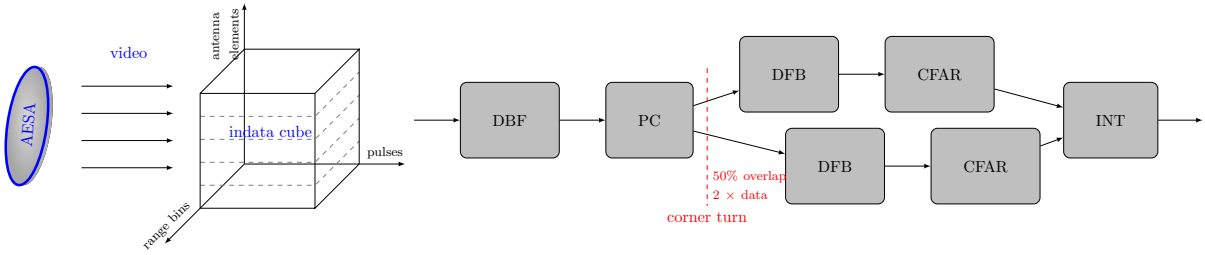


Figure 2: Overview of the video processing chain

In this report we assume one stream per antenna element. The indata is organized into a sequence of "cubes", each corresponding to a certain integration interval. Each sample in the cube represents a particular antenna element, pulse and range bin. The data of the cube arrives pulse by pulse and each pulse arrives range bin by range bin. This is for all elements in parallel. Between the Pulse Compression

(PC) and Doppler Filter Bank (DFB) steps there is a corner turn of data, i.e. data from all pulses must be collected before the DFB can execute.

The different steps of the chain, the antenna and the input data format are briefly described in the following list. For a more detailed description of the processing chain, please refer to section 2.

- *Digital Beam Forming (DBF)*: The DBF step creates a number of simultaneous receiver beams, or "listening directions", from the input data. This is done by doing weighted combinations of the data from the different antenna elements, so that constructive interference is created in the desired bearings of the beams. The element samples in the input data set are then transformed into beams.
- *Pulse Compression (PC)*: The goal of the pulse compression is to collect all received energy from one target into a single range bin. The received echo of the modulated pulse is passed through a matched filter. Here, the matched filtering is done digitally.
- *Doppler Filter Bank incl. envelope detection (DFB)*: The DFB gives an estimation of the target's speed relative to the radar. It also gives an improved signal-to-noise ratio due to a coherent integration of indata. The pulse bins in the data set are transformed into Doppler channels. The envelope detector calculates the absolute values of the digital samples. The data is real after this step.
- *Constant False Alarm Ratio (CFAR)*: The CFAR processing is intended to keep the number of false targets at an acceptable level while maintaining the best possible sensitivity. It normalizes the video in order to maintain a constant false alarm rate when the video is compared to a detection threshold. With this normalization the sensitivity will be adapted to the clutter situation in the area (around a cell under test) of interest.
- *Integrator (INT)* The integrator is an 8-tap unsigned integrator that integrates channel oriented video. Integration shall be done over a number of FFT batches of envelope detected video. Each Doppler channel, range bin and antenna element shall be integrated.

The following table briefly presents the dimensions of the primitive functions associated with each processing stage, where data sets are measured in number of elements processed organized in channels × range bins × pulses; data precision is measured in bit width; and approximative performance is given in MOPS. These dimensions represent the scaled-down version of a realistic AESA radar signal processing system, with 16 input antenna channels forming into 8 beams.

Table 1: Dimensions of the AESA case study covered in the report

Block	In Data Set	Out Data Set	Type		Precision		Approx.
			In	Out	In	Out	perform.
DBF	$16 \times 1 \times 1$	$8 \times 1 \times 1$	$\mathbb{C}$	$\mathbb{C}$	16	20	2688
PC	$1 \times 1024 \times 1$	$1 \times 1024 \times 1$	$\mathbb{C}$	$\mathbb{C}$	20	20	4608
DFB	$1 \times 1 \times 256$	$1 \times 1 \times 256$	$\mathbb{C}$	$\mathbb{R}$	20	20	7680
CFAR	$1 \times 1024 \times 256$	$1 \times 1024 \times 256$	$\mathbb{R}$	$\mathbb{R}$	20	16	360
INT	$1 \times 1 \times 1$	$1 \times 1 \times 1$	$\mathbb{R}$	$\mathbb{R}$	16	16	24576
AESA	$16 \times 1024 \times 256$	$2(8 \times 1024 \times 256)$	$\mathbb{C}$	$\mathbb{R}$	–	–	39912

1.2 Using This Document

PREREQUISITES: the document assumes that the reader is familiar with the functional programming language Haskell, its syntax, 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. The reader also needs to be familiar with some basic ForSyDe modeling concepts, such as *process constructor*, *process* or *signal*. We recommend going through at least the online getting started tutorial on ForSyDe-Shallow or the one on ForSyDe-Atom, and if possible, consulting the (slightly outdated) book chapter on ForSyDe (Sander, Jantsch, and Attarzadeh-Niaki 2017).

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

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

shows *source code* as it is found in the implementation files, where the line numbers correspond to the position in the source file. 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 (e.g. `2`).

The way this document is meant to be parsed efficiently is to load the source files themselves in an interpreter and test the exported functions gradually, while reading the document at the same time. Due to multiple (sometimes conflicting) dependencies on external packages, for convenience the source files are shipped as *multiple* Stack packages each creating an own sandbox on the user's local machine with all dependencies and requirements taken care of. Please refer to the project's `README` file for instructions on how to install and compile or run the Haskell files.

At the beginning of each chapter there is meta-data guiding the reader what tools and packages are used, like:

Package	aesa-atom-0.1.0	path: `./aesa-atom/README.md`

This table tells that the package where the current chapter's code resides is `aesa-atom`, version 0.1.0. This table might contain information on the main dependency packages which contain important functions, and which should be consulted while reading the code in the current chapter. If the main package generates an executable binary or a test suite, these are also pointed out. The third column provides additional information such as a pointer to documentation (relative path to the project root, or web URL), or usage suggestions.

It is recommended to read the main package's `README` file which contains instructions on how to install, compile and test the software, before proceeding with following a chapter. Each section of a chapter is written within a library *module*, pointed out in the beginning of the respective section by the line:

```haskell
module ForSyDe.X.Y where
```

The most convenient way to test out all functions used in module `ForSyDe.X.Y` is by loading its source file in the sandboxed interpreter, i.e. by running the following command from the project root:

```
stack ghci src/ForSyDe/X/Y.lhs
```

An equally convenient way is to create an own `.hs` file somewhere under the project root, which imports and uses module `ForSyDe.X.Y`, e.g.

```haskell
-- MyTest.hs
import ForSyDe.X.Y

myData = [1,2,3,4,5] :: [Int]
myTest = functionFromForSyDeXY myData
```

This file can be loaded and/or compiled from within the sandbox, e.g. with `stack ghci MyTest.hs`.

2 High-Level Model of the AESA Signal Processing Chain in ForSyDe-Atom

This section guides the reader throughout translating "word-by-word" the provided specifications of the AESA signal processing chain into a concrete, functionally complete, high-level executable model in ForSyDe-Atom. This first attempt focuses mainly on the top-level functional behavior of the system, exerting the successive transformations upon the input video cubes, as suggested by Figure 2. We postpone the description/derivation of more appropriate time behaviors for later sections. Enough AESA system details are given in order to understand the model. At the end of this section we simulate this model against a realistic input set of complex antenna data, and test if it is sane (i.e. provides the expected results).

Package	aesa-atom-0.1.0	path: `./aesa-atom/README.md`
Deps	forsyde-atom-0.2.2	url: `https://forsyde.github.io/forsyde-atom/api/`
	forsyde-atom-extensions-0.1.1	path: `./forsyde-atom-extensions/README.md`
Bin	aesa-cube	usage: `aesa-cube --help`

Historically, ForSyDe-Atom has been a spin-off of ForSyDe-Shallow which has explored new modeling concepts, and had a fundamentally different approach to how models are described and instantiated. ForSyDe-Atom introduced the concept of *language layer*, extending the idea of process constructor, and within this setting it explored the algebra of algorithmic skeletons, employed heavily within this report. As of today, both ForSyDe-Shallow and ForSyDe-Atom have similar APIs and are capable of modeling largely the same behaviors. The main syntax differences between the two are covered in section 3, where the same high-level model is written in ForSyDe-Shallow.

2.1 Crash course in ForSyDe-Atom

Before proceeding with the modeling of the AESA processing chain, let us consolidate the main ForSyDe concepts which will be used throughout this report: *layer*, *process constructor* and *skeleton*. If you are not familiar with ForSyDe nomenclature and general modeling principles, we recommend consulting the documentation pointed out in section 1 first.

As seen in section 1.1, most of the AESA application consists of typical DSP algorithms such as matrix multiplications, FFT, moving averages, etc. which lend themselves to *streaming parallel* implementations. Hence we need to unambiguously capture the two distinct aspects of the AESA chain components:

- *streaming behavior* is expressed in ForSyDe through processes which operate on signals encoding a temporal partitioning of data. Processes are instantiated exclusively using *process constructors* which can be regarded as templates inferring the semantics of computation, synchronization and concurrency as dictated by a certain model of computation (MoC) (Sander and Jantsch (2004),Lee and Seshia (2016)).
- *parallelism* is expressed through parallel patterns which operate on structured types (e.g. vectors) encoding a spatial partitioning of data. These patterns are instantiated as skeletons, which are templates inferring the semantics of distribution, parallel computation and interaction between elements, as defined by the algebra of catamorphisms (Fischer, Gorlatch, and Bischof 2003).

In order to capture these two[1] interacting aspects in unison as a complete system, we describe them in terms of two distinct, orthogonal *layers* within the ForSyDe-Atom framework. A language layer is a (very small) domain specific language (DSL) heavily rooted in the functional programming paradigm, which is able to describe one aspect of a cyber-physical system (CPS). Layers interact with one another by virtue of the abstraction principle inherent to a functional programming language (Backus 1978): each layer defines at least one higher order function that is able to take a function (i.e. abstraction) from another layer as an argument and to lift it within its own domain. To picture the interaction between layers, consider Figure 2 where, although we use the same `farm` vector skeleton and `comb` synchronous (SY) process constructor, the two different compositions describe two different (albeit semantically equivalent)

[1] and other aspects, not covered in this report

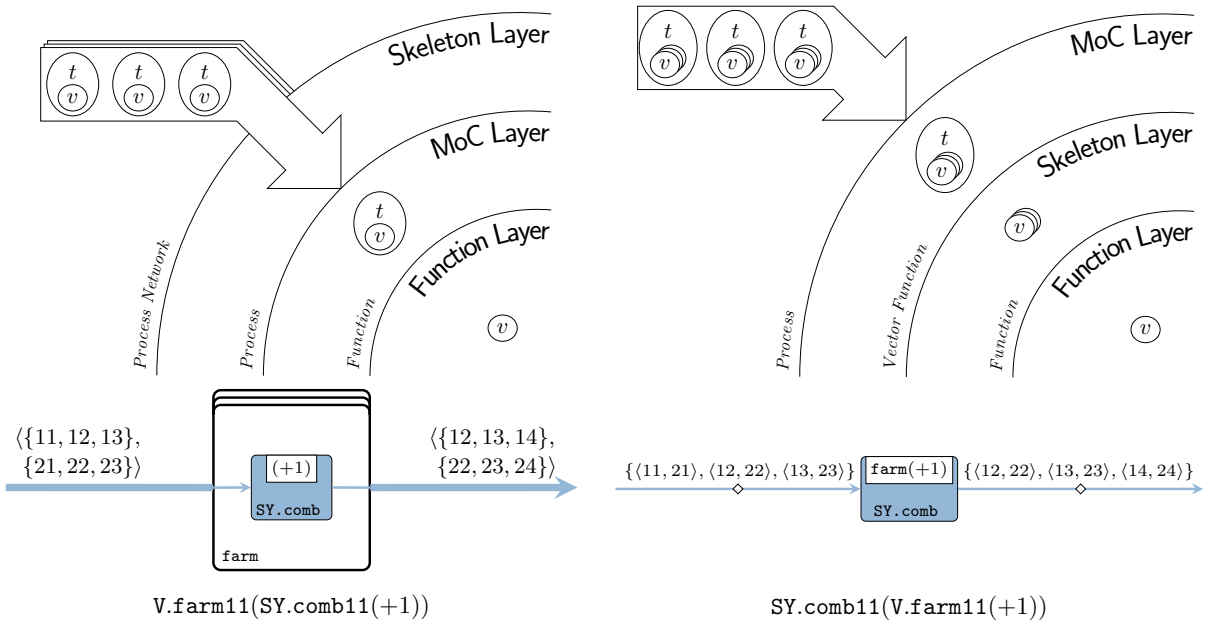


Figure 3: Depiction of layer usage: (left) skeleton networks of processes; (right) processes of skeleton functions

systems: the first one is instantiating a farm network of SY processes operating on a vector of signals in parallel; whereas the second one is instantiating a single SY process operating on o signal where each event is carrying a vector of values.

2.2 A High-Level Model

This section presents a high-level behavioral model of the AESA signal processing chain presented in section 1.1. This model follows intuitive and didactic way to tackle the challenge of translating the *textual* specifications into an *executable* ForSyDe specification and is not, in any circumstance, the only way to model the application. As of section 2.1 we represent each stage in the AESA chain as a *process* acting upon (complete) *cubes* of data, i.e. as processes of skeleton functions. At this phase in the system design we do not care *how* the cubes are being formed or how the data is being carried around, bur rather on *what* transformations are applied on the data cubes during each subsequent stage of the AESA pipeline. The purpose of the model is to provide a executable *reference* for the AESA system functionality, that can later be derived to more efficient descriptions.

```
15 {-# LANGUAGE PackageImports #-} -- allows explicit import of modules from custom
16                                 -- libraries instead of standard ones. Will be taken
17                                 -- out once the extensions are merged upstream.
```

The code for this section is written in the following module, see section 1.2 on how to use it:

```
22 module AESA.CubesAtom where
```

As the AESA application uses complex numbers, we use Haskell's `Complex` type.

```
27 import Data.Complex
```

The only timed behavior exerted by the model in this section is the causal, i.e. ordered, passing of cubes from one stage to another. In order to enable a simple, abstract, and thus analyzable "pipeline" behavior this passing can be described according to the *perfect synchrony hypothesis*, which assumes the processing of each event (cube) takes an infinitely small amount of time and it is ready before the next synchronization point. This in turn implies that all events in a system are synchronized, enabling the description of fully deterministic behaviors over infinite streams of events. These precise execution semantics are captured by the *synchronous reactive (SY) model of computation (MoC)* (Lee and Seshia

(2016),Benveniste et al. (2003)), hence we import the SY library from the *MoC layer* of ForSyDe-Atom, see (Ungureanu and Sander 2017), using an appropriate alias.

```
43 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
```

For describing parallel operations on data we use algorithmic skeletons (Fischer, Gorlatch, and Bischof (2003),Skillicorn (2005)), formulated on ForSyDe-Atom's in-house Vector data type, which is a shallow, lazy-evaluated implementation of unbounded arrays, ideal for early design validation. Although dependent, bounded, and even boxed (i.e. memory-mapped) alternatives exist, such as FSVec or REPA Arrays, for the scope of this project the functional validation and (by-hand) requirement analysis on the properties of skeletons will suffice. We also import the `Matrix` and `Cube` utility libraries which contain type synonyms for nested `Vectors` along with their derived skeletons, as well a `DSP` which contain commonly used DSP blocks defined in terms of vector skeletons.

```
59 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector        as V
60 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.Matrix as M
61 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.Cube   as C
62 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.DSP
```

Finally, we import the local project module defining different coefficients for the AESA algorithms, presented in detail in section 2.2.5.

```
67 import AESA.Coefs
```

2.2.1 Type Aliases and Constants

The system parameters are integer constants defining the size of the application. For a simple test scenario provided by Saab AB, we have bundled these parameters in the following module, and we shall use their variable names throughout the whole report:

```
75 import AESA.Params
76 import Data.Ratio
```

For ease of documentation we will be using type synonyms (aliases) for all types and structures throughout this design:

- `Antenna` denotes a vector container for the antenna elements. Its length is equal to the number of antennas in the radar N_A.
- After Digital Beamforming (DBF), the antenna elements are transformed into N_B beams, thus we associate the `Beam` alias for the vector container wrapping those beams.
- `Range` is a vector container for range bins. All antennas have the same number of range bins N_b, rendering each Antenna $\times$ Range a perfect matrix of samples for every pulse.
- `Window` stands for a Doppler window of N_{FFT} pulses.

```
94 type Antenna     = Vector -- length: nA
95 type Beam        = Vector -- length: nB
96 type Range       = Vector -- length: nb
97 type Window      = Vector -- length: nFFT
```

Finally we provide two aliases for the basic Haskell data types used in the system, to stay consistent with the application specification.

```
102 type CpxData  = Complex Float
103 type RealData = Float
```

2.2.2 Video Processing Pipeline Stages

In this section we follow each stage described in section 1.1, and model them as a processes operating on cubes (three-dimensional vectors) of antenna samples.

2.2.2.1 Digital Beamforming (DFB)

The DBF receives complex in data, from N_A antenna elements and forms N_B simultaneous receiver beams, or "listening directions", by summing individually phase-shifted in data signals from all elements. Considering the indata video cube, the transformation applied by DBF, could be depicted as in Figure 4.

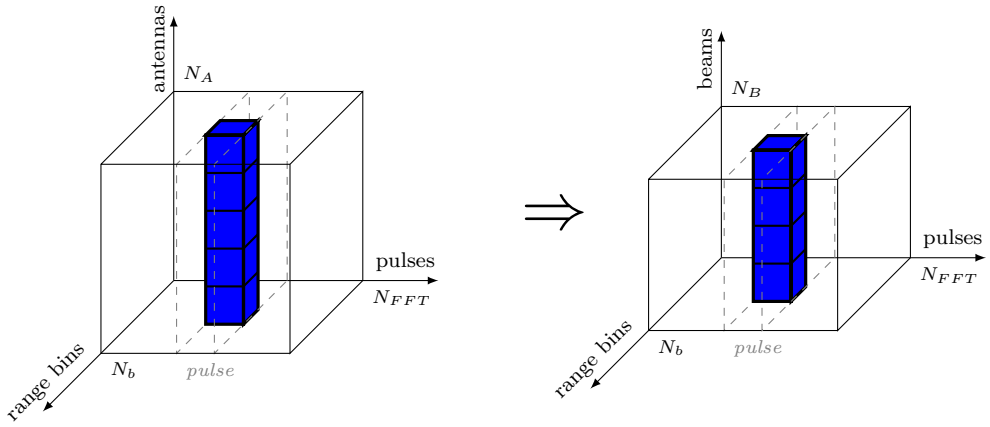


Figure 4: DBF on video structure

Considering the application specification in section 1.1 on the input data, namely *"for each antenna the data arrives* pulse by pulse*, and each pulse arrives* range bin by range bin*"*, we can assume that the video is received as `Antenna (Window (Range a))` cubes, meaning that the inner vectors are the range bins. However, Figure 4 shows that the beamforming function is applied in the antenna direction, so we need to transpose the cube in such a way that `Antenna` becomes the inner vector, i.e. `Window (Range (Antenna a))`. We thus describe the DBF stage as a combinational SY process `comb` acting upon signals of `Cubes`, namely mapping the beamforming function f_{DBF} on each column of each *pulse matrix* (see Figure 4).

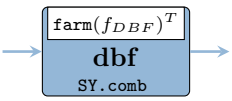


Figure 5: DBF stage process

```
dbf :: Signal (Antenna (Window (Range CpxData)))
    -> Signal (Window  (Range  (Beam  CpxData)))
dbf = SY.comb11 (M.farm11 fDBF . C.transpose)
```

The beamforming function is specified like in eq. 1, where e_k, $\forall k \in [1, N_A]$ denotes the samples from antenna elements, and respectively b_i, $\forall i \in [1, N_B]$ are samples for each beam. This is in fact a of matrix-vector multiplication, thus we implement eq. 1 at the highest level of abstraction simply as matrix/vector operations like in eq. 2.

$$b_i(n) = \sum_{k=1}^{N_A} e_k(n) \cdot \alpha_{ki} \qquad \forall i \in [1, N_B] \tag{1}$$

```
fDBF :: Antenna CpxData -- ^ input antenna elements
     -> Beam    CpxData -- ^ output beams
fDBF antennaEl = beams
  where
    beams      = V.reduce (V.farm21 (+)) beamMatrix
    beamMatrix = M.farm21 (*) elMatrix beamConsts
    elMatrix   = V.farm11 V.fanout antennaEl
    beamConsts = mkBeamConsts dElements waveLength nA nB :: Matrix CpxData
```

Function	Original module	Package
`farm11`, `reduce`, `length`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`farm11`, `farm21`	`ForSyDe.Atom.Skeleton.Vector.Matrix`	forsyde-atom-extensions
`transpose`	`ForSyDe.Atom.Skeleton.Vector.Cube`	forsyde-atom-extensions
`comb11`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`mkBeamConsts`	`AESA.Coefs`	aesa-atom
`dElements`, `waveLenth`, `nA`, `nB`	`AESA.Params`	aesa-atom

$$\underset{\text{elMatrix}}{\begin{bmatrix} e_1 & e_1 & \dots \\ e_2 & e_2 & \dots \\ \vdots & \vdots & \ddots \\ e_{N_A} & e_{N_A} & \dots \end{bmatrix}} \times \underset{\text{beamConsts}}{\begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1N_B} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2N_B} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{N_A1} & \alpha_{N_A2} & \dots & \alpha_{N_AN_B} \end{bmatrix}} \Rightarrow$$
$$\Rightarrow \underbrace{\begin{bmatrix} e_1\alpha_{11} & e_1\alpha_{12} & \dots & e_1\alpha_{1N_B} \\ e_2\alpha_{21} & e_2\alpha_{22} & \dots & e_2\alpha_{2N_B} \\ \vdots & \vdots & \ddots & \vdots \\ e_{N_A}\alpha_{N_A1} & e_{N_A}\alpha_{N_A2} & \dots & e_{N_A}\alpha_{N_AN_B} \end{bmatrix}}_{\text{beamMatrix}} \Big\} \sum_{rows} = \underset{\text{beam}}{\begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}} \tag{2}$$

In fact the operation performed by the f_{DBF} function in eq. 1, respectively in eq. 2 is nothing else but a dot operation between a vector and a matrix. Luckily, `ForSyDe.Atom.Skeleton.Vector.DSP` exports a utility skeleton called `dotvm` which instantiates exactly this type of operation. Thus we can instantiate an equivalent f'_{DBF} simply as

```
200 fDBF' antennaEl = antennaEl `dotvm` beamConsts
201   where
202     beamConsts = mkBeamConsts dElements waveLength nA nB :: Matrix CpxData
```

The expanded version `fDBF` was given mainly for didactic purpose, to get a feeling for how to manipulate different skeletons to obtain more complex operations or systems. From now on, in this section we will only use the library-provided operations, and we will study them in-depth only later in section 5.

2.2.2.2 Pulse Compression (PC)

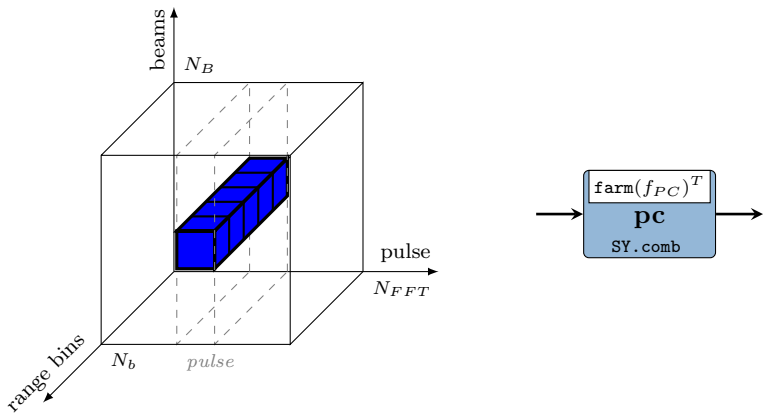


Figure 6: PC: direction of application on video structure (left); process (right)

In this stage the received echo of the modulated pulse, i.e. the information contained by the range bins, is passed through a matched filter for decoding their modulation. This essentially applies a sliding window,

or a moving average on the range bin samples. Considering the video cube, the PC transformation is applied in the direction shown in Figure 6, i.e. on vectors formed from the range of every pulse.

The PC process is mapping the f_{PC} on each row of the pulse matrices in a cube, however the previous stage has arranged the cube to be aligned beam-wise. This is why we need to re-arrange the data so that the innermost vectors are `Ranges` instead, and we do this by simply `transpose`-ing the inner `Range` × `Beam` matrices into `Beam` × `Range` ones.

```
225 pc :: Signal (Window (Range (Beam  CpxData)))
226    -> Signal (Window (Beam  (Range CpxData)))
227 pc = SY.comb11 (V.farm11 (V.farm11 fPC . M.transpose))
228 --              ^ == (M.farm11 fPC . V.farm11 M.transpose)
```

Here the function f_{PC} applies the `fir` skeleton on these vectors (which computes a moving average if considering vectors). The `fir` skeleton is a utility formulated in terms of primitive skeletons (i.e. `farm` and `reduce`) on numbers, i.e. lifting arithmetic functions. We will study this skeleton later in this report and for now we take it "for granted", as conveniently provided by the `DSP` utility library. For this application we also use a relatively small average window (5 taps).

```
237 fPC :: Range CpxData -- ^ input range bin
238     -> Range CpxData -- ^ output pulse-compressed bin
239 fPC = fir (mkPcCoefs pcTap)
```

Function	Original module	Package
`comb11`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`farm11`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`farm11`, `transpose`	`ForSyDe.Atom.Skeleton.Vector.Matrix`	forsyde-atom-extensions
`fir`	`ForSyDe.Atom.Skeleton.Vector.DSP`	forsyde-atom-extensions
`mkPcCoefs`	`AESA.Coefs`	aesa-atom

2.2.2.3 Corner Turn (CT) with 50% overlapping data

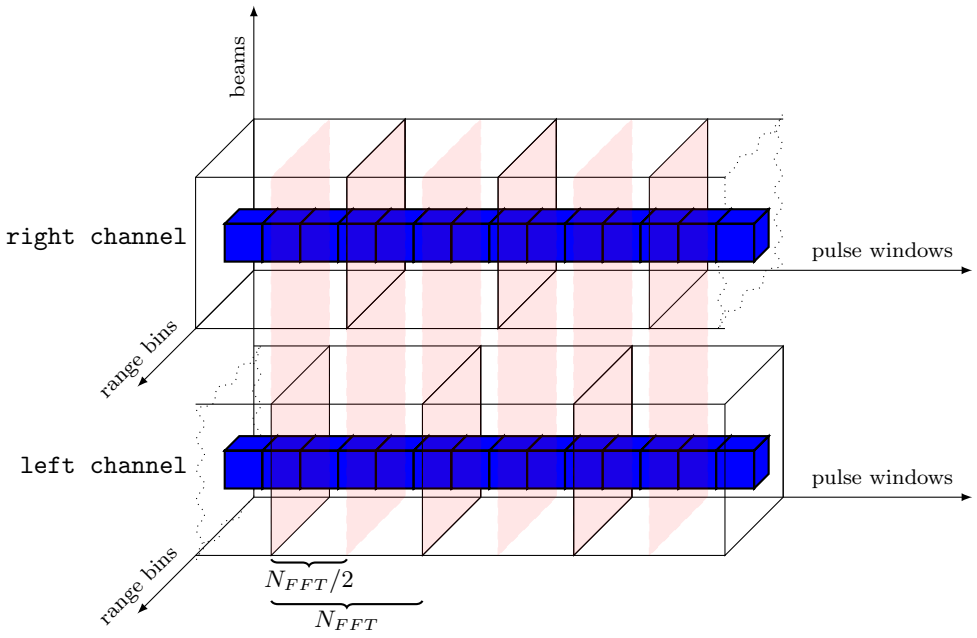


Figure 7: Concurrent processing on 50% overlapped data

During the CT a rearrangement of data must be performed between functions that process data in "different" directions, e.g. range and pulse, in order to be able to calculate the Doppler channels further

in the processing pipeline. The process of corner turning becomes highly relevant when detailed time behavior of the application is being derived or inferred, since it demands well-planned design decisions to make full use of the underlying architecture. At the level of abstraction on which we work right now though, it is merely a matrix `transpose` operation, and it can very well be postponed until the beginning of the next stage. However, a much more interesting operation is depicted in Figure 7: in order to maximize the efficiency of the AESA processing the datapath is split into two concurrent processing channels with 50% overlapped data.

Implementing such a behavior requires a bit of "ForSyDe thinking". At a first glance, the problem seems easily solved considering only the cube structures: just "ignore" half of the first cube of the right channel, while the left channel replicates the input. However, there are some timing issues with this setup: from the left channel's perspective, the right channel is in fact "peaking into the future", which is an abnormal behavior. Without going too much into details, you need to understand that *any type of signal "cleaning", like dropping or filtering out events, can cause serious causality issues in a generic process network, and thus it is* ***illegal*** *in ForSyDe system modeling.* On the other hand we could *delay* the left channel in a deterministic manner by assuming a well-defined *initial state* (e.g. all zeroes) while it waits for the right channel to consume and process its first half of data. This defines the history of a system where all components start from *time zero* and eliminates any source of "clairvoyant"/ambiguous behavior.

To keep things simple, we stay within the same time domain, keeping the perfect synchrony assumption, and instantiating the left channel building mechanism as a simple Mealy finite state machine. This machine splits an input cube into two halves, stores one half and merges the other half with the previously stored state to create the left channel stream of cubes.

Figure 8: Left channel data builder process

```
287 overlap :: Signal (Window (Beam (Range CpxData)))
288         -> Signal (Window (Beam (Range CpxData)))
289 overlap = SY.mealy11 nextState outDecode initState
290   where
291     nextState _ cube = V.drop (nFFT `div` 2) cube
292     outDecode s cube = s <++> V.take (nFFT `div` 2) cube
293     initState        = (V.fanoutn (nFFT `div` 2) . V.fanoutn nB . V.fanoutn nb) 0
```

Function	Original module	Package
`mealy11`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`drop`, `take`, `fanoutn`, `(<++>)`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`nFFT`, `nA`, `nB`	`AESA.Params`	aesa-atom

OBS! Perhaps considering all zeroes for the initial state might not be the best design decision, since that is in fact "junk data" which is propagated throughout the system and which alters the expected behavior. A much safer (and semantically correct) approach would be to model the initial state using *absent events* instead of arbitrary data. However this demands the introduction of a new layer and some quite advanced modeling concepts which are out of the scope of this report[2]. For the sake of simplicity we now consider that the initial state is half a cube of zeroes and that there are no absent events in the system. As earlier mentioned, it is *illegal* to assume any type of signal cleaning during system modeling, however this law does not apply to the *observer* (i.e. the testbench), who is free to take into consideration whichever parts of the signals it deems necessary. We will abuse this knowledge in order to show a realistic output behavior of the AESA signal processing system: as "observers", we will ignore the effects of the initial state propagation from the output signal and instead plot only the useful data.

[2]For more information on absent semantics, check out Chapter 3 of (Lee and Seshia 2016)

2.2.2.4 Doppler Filter Bank (DFB)

During the Doppler filter bank, every window of samples, associated with each range bin is transformed into a Doppler channel and the complex samples are converted to real numbers by calculating their envelope.

The `dfb` process applies the the following chain of functions on each window of complex samples, in three consecutive steps:

- scale the window samples with a set of coefficients to decrease the Doppler side lobes from each FFT output and thereby to increase the clutter rejection.
- apply an N_{FFT}-point 2-radix decimation in frequency Fast Fourier Transform (FFT) algorithm.
- compute the envelope of each complex sample when phase information is no longer of interest. The envelope is obtained by calculating the absolute value of the complex number, converting it into a real number.

Function	Original module	Package
`farm11`,`farm21`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`fft`	`ForSyDe.Atom.Skeleton.Vector.DSP`	forsyde-atom-extensions
`mkWeightCoefs`	`AESA.Coefs`	aesa-atom
`nS`, `nFFT`	`AESA.Params`	aesa-atom

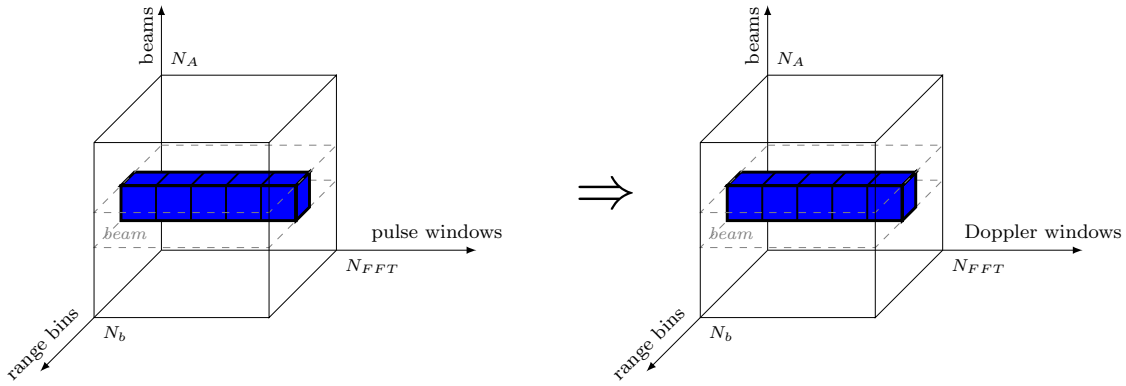


Figure 9: Doppler Filter Bank on video structure

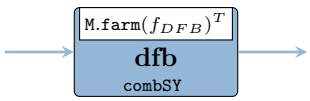


Figure 10: DFB process

```
350 dfb :: Signal (Window (Beam  (Range  CpxData )))
351     -> Signal (Beam   (Range (Window RealData)))
352 dfb = SY.comb11 (M.farm11 fDFB . C.transpose)
353
354 fDFB :: Window CpxData -> Window RealData
355 fDFB = V.farm11 envelope . fft nS . weight
356   where
357     weight     = V.farm21 (*) (mkWeightCoefs nFFT)
358     envelope a = let (i, q) = (realPart a, imagPart a)
359                  in sqrt (i * i + q * q)
```

2.2.2.5 Constant False Alarm Ratio (CFAR)

The CFAR normalizes the data within the video cubes in order to maintain a constant false alarm rate with respect to a detection threshold. This is done in order to keep the number of false targets at an acceptable level by adapting the normalization to the clutter situation in the area (around a cell under test) of interest. The described process can be depicted as in Figure 11 which suggests the stencil data accessing pattern within the video cubes.

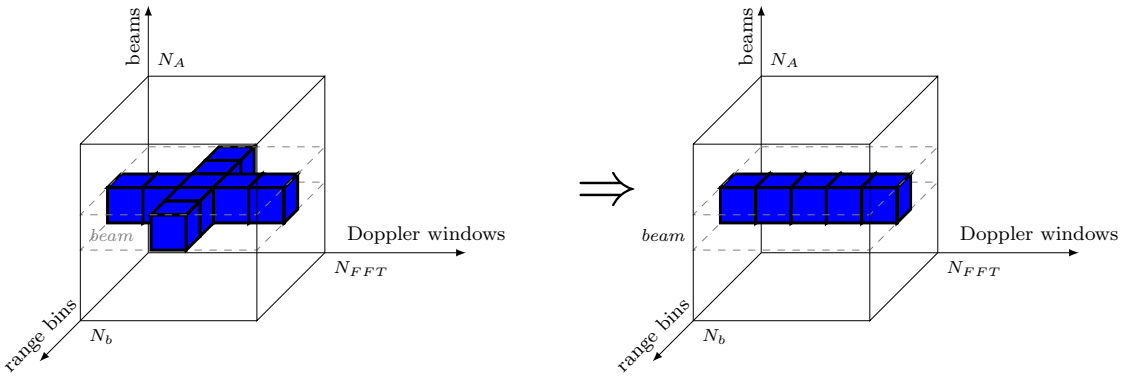


Figure 11: Constant False Alarm Ratio on cubes of complex samples

```
373 cfar :: Signal (Beam (Range (Window RealData)))
374       -> Signal (Beam (Range (Window RealData)))
375 cfar = SY.comb11 (V.farm11 fCFAR)
```

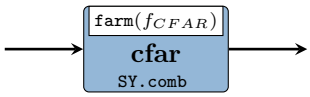


Figure 12: CFAR process

The `cfar` process applies the f_{CFAR} function on every $N_b \times N_{FFT}$ matrix corresponding to each beam. The f_{CFAR} function normalizes each Doppler window, after which the sensitivity will be adapted to the clutter situation in current area, as seen in Figure 13. The blue line indicates the mean value of maximum of the left and right reference bins, which means that for each Doppler sample, a swipe of neighbouring bins is necessary, as suggested by Figure 11. This is a typical pattern in signal processing called stencil, which will constitute the main parallel skeleton within the f_{CFAR} function.

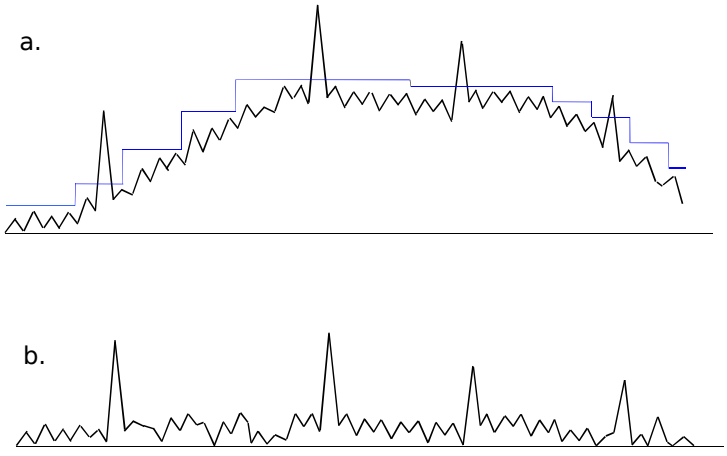


Figure 13: The signal level within one pulse window: a) before CFAR; b) after CFAR

```haskell
fCFAR :: Range (Window RealData) -> Range (Window RealData)
fCFAR rbins = V.farm41 (\m -> V.farm31 (normCfa m)) md rbins lmv emv
  where
    md  = V.farm11 (logBase 2 . V.reduce min) rbins
    emv = (V.fanoutn (nFFT + 1) dummy) <++> (V.farm11 aritMean neighbors)
    lmv = (V.drop 2 $ V.farm11 aritMean neighbors) <++> (V.fanout dummy)
    ------------------------------------------------
    normCfa m a l e = 2 ** (5 + logBase 2 a - maximum [l,e,m])
    aritMean :: Vector (Vector RealData) -> Vector RealData
    aritMean  = V.farm11 (/n) . V.reduce addV . V.farm11 geomMean . V.group 4
    geomMean  = V.farm11 (logBase 2 . (/4)) . V.reduce addV
    ------------------------------------------------
    dummy     = V.fanoutn nFFT (-maxFloat)
    neighbors = V.stencil nFFT rbins
    ------------------------------------------------
    addV      = V.farm21 (+)
    n         = fromIntegral nFFT
```

Function	Original module	Package
`farm[4/3/1]1`, `reduce`, `<++>`, `drop`, `fanout`, `fanoutn`, `stencil`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`comb11`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`maxFloat`	AESA.Coefs	aesa-atom
`nb`, `nFFT`	`AESA.Params`	aesa-atom

The f_{CFAR} function itself can be described with the system of eq. 3, where

- MD is the minimum value over all Doppler channels in a batch for a specific data channel and range bin.
- EMV and LMV calculate the early and respectively late mean values from the neighboring range bins as a combination of geometric and arithmetic mean values.
- eb and lb are the earliest bin, respectively latest bin for which the CFAR can be calculated as EMV and LMV require at least N_{FFT} bins + 1 guard bin before and respectively after the current bin. This phenomenon is also called the "stencil halo", which means that CFAR, as defined in eq. 3 is applied only on $N'_b = N_b - 2N_{FFT} - 2$ bins.
- bins earlier than eb, respectively later than lb, are ignored by the CFAR formula and therefore their respective EMV and LMV are replaced with the lowest representable value.
- 5 is added to the exponent of the CFAR equation to set the gain to 32 (i.e. with only noise in the incoming video the output values will be 32).

$$\begin{cases} CFAR(a_{ij}) = 2^{(5+\log_2 a_{ij}) - \max(EMV(a_{ij}), LMV(a_{ij}), MD(a_{ij}))} \\ EMV(a_{ij}) = \dfrac{1}{N}\displaystyle\sum_{k=0}^{N-1}\left(\log_2\left(\frac{1}{4}\sum_{l=0}^{3} a_{(i-2-4k-l)j}\right)\right) \\ LMV(a_{ij}) = \dfrac{1}{N}\displaystyle\sum_{k=0}^{N-1}\left(\log_2\left(\frac{1}{4}\sum_{l=0}^{3} a_{(i+2+4k+l)j}\right)\right) \\ MD(a_{ij}) = \log_2\left(\min_{k=1}^{N}(a_{ik})\right) \end{cases} \quad (3)$$

$$\forall i \in [eb, lb], j \in [1, N] \text{ where } \begin{cases} N = N_{FFT} \\ eb = N_{FFT} + 1 \\ lb = N_b - N_{FFT} - 1 \end{cases}$$

The first thing we calculate is the MD for each Doppler window (row). For each row of `rbins` (i.e. range

bins of Doppler windows) we look for the minimum value (`reduce min`) and apply the binary logarithm on it.

Another action performed over the matrix `rbins` is to form two stencil "cubes" for EMV and LMV respectively, by gathering batches of N_{FFT} Doppler windows like in eq. 4, computing them like in eq. 5.

$$\overset{\text{rbins}}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N_{FFT}} \\ a_{21} & a_{22} & \cdots & a_{2N_{FFT}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N_b1} & a_{N_b2} & \cdots & a_{N_bN_{FFT}} \end{bmatrix}} \xrightarrow{\texttt{stencil}} \overset{\text{neighbors}}{\begin{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N_{FFT}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N_{FFT}1} & a_{N_{FFT}2} & \cdots & a_{N_{FFT}N_{FFT}} \end{bmatrix} \\ \begin{bmatrix} a_{21} & a_{22} & \cdots & a_{2N_{FFT}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(N_{FFT}+1)1} & a_{(N_{FFT}+1)2} & \cdots & a_{(N_{FFT}+1)N_{FFT}} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} a_{(N_b-N_{FFT})1} & a_{(N_b-N_{FFT})2} & \cdots & a_{(N_b-N_{FFT})N_{FFT}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N_b1} & a_{N_b2} & \cdots & a_{N_bN_{FFT}} \end{bmatrix} \end{bmatrix}} \tag{4}$$

Each one of these neighbors matrices will constitute the input data for calculating the EMV and LMV for each Doppler window. EMV and LMV are calculated by applying the mean function `arithMean` over them, as shown (only for the window associated with the eb bin) in eq. 5. The resulting `emv` and `lmv` matrices are padded with rows of the minimum representable value `-maxFloat`, so that they align properly with `rbins` in order to combine into the 2D farm/stencil defined at eq. 3. Finally, `fCFAR` yields a matrix of normalized Doppler windows. The resulting matrices are not transformed back into sample streams by the parent process, but rather they are passed as single tokens downstream to the INT stage, where they will be processed as such.

$$\begin{bmatrix} a_{11} & \cdots & a_{1N_{FFT}} \\ \vdots & \ddots & \vdots \\ a_{N_{FFT}1} & \cdots & a_{N_{FFT}N_{FFT}} \end{bmatrix} \xrightarrow{\texttt{group}} \begin{bmatrix} \begin{bmatrix} a_{11} & \cdots & a_{1N_{FFT}} \\ \vdots & \ddots & \vdots \\ a_{41} & \cdots & a_{4N_{FFT}} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} a_{(N_{FFT}-4)1} & \cdots & a_{(N_{FFT}-4)N_{FFT}} \\ \vdots & \ddots & \vdots \\ a_{N_{FFT}1} & \cdots & a_{N_{FFT}N_{FFT}} \end{bmatrix} \end{bmatrix} \tag{5}$$

$$\xrightarrow{\texttt{farm(geomMean)}} \begin{bmatrix} \log_2 \frac{1}{4} \sum_{i=1}^{4} a_{i1} & \cdots & \log_2 \frac{1}{4} \sum_{i=1}^{4} a_{iN_{FFT}} \\ \vdots & \ddots & \vdots \\ \log_2 \frac{1}{4} \sum_{i=N_{FFT}-4}^{N_{FFT}} a_{i1} & \cdots & \log_2 \frac{1}{4} \sum_{i=N_{FFT}-4}^{N_{FFT}} a_{iN_{FFT}} \end{bmatrix}$$

$$\xrightarrow{(/\texttt{N}_{\texttt{FFT}})\circ\texttt{reduce}(+)} \begin{bmatrix} EMV(a_{eb,1}) & \cdots & EMV(a_{eb,N_{FFT}}) \end{bmatrix}$$

2.2.2.6 Integrator (INT)

During the last stage of the video processing chain each data sample of the video cube is integrated against its 8 previous values using an 8-tap FIR filter, as suggested by the drawing in Figure 14.

```
int :: Signal (Beam (Range (Window RealData)))
    -> Signal (Beam (Range (Window RealData)))
    -> Signal (Beam (Range (Window RealData)))
int right left = firNet mkIntCoefs $ SY.interleave right left
```

Before integrating though, the data from both the left and the right channel need to be merged and interleaved. This is done by the process `interleave` below, which is a convenient utility exported by the SY library, hiding a domain interface. When considering only the data structures, the `interleave`

process can be regarded as an up-sampler with the rate 2/1. When taking into consideration the size of the entire data set (i.e. token rates × structure sizes × data size), we can easily see that the overall required system bandwidth (ratio) remains the same between the PC and INT stages, i.e. $\frac{2\times N_B\times N_b\times N_{FFT}\times size(\texttt{RealData})}{N_B\times N_b\times N_{FFT}\times size(\texttt{CpxData})} = 1/1$.

Function	Original module	Package
`farm21`,`farm11`,`fanout`	ForSyDe.Atom.Skeleton.Vector.Cube	forsyde-atom-extensions
`fir'`	ForSyDe.Atom.Skeleton.Vector.DSP	forsyde-atom-extensions
`comb21`,`comb11`, `delay`, `interleave`	ForSyDe.Atom.MoC.SY	forsyde-atom
`mkFirCoefs`	AESA.Coefs	aesa-atom

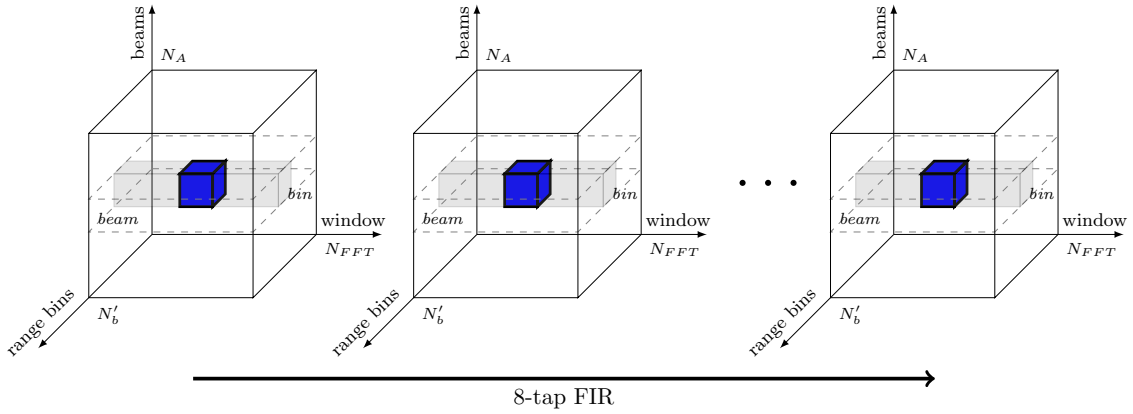


Figure 14: Integration on cubes of complex samples

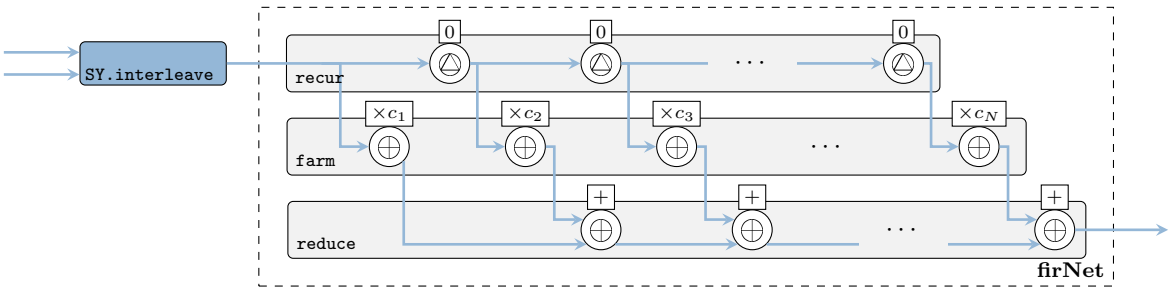


Figure 15: INT network

The 8-tap FIR filter used for integration is also a moving average, but as compared to the `fir` function used in section 2.2.2.2, the window slides in time domain, i.e. over streaming samples rather than over vector elements. To instantiate a FIR system we use the `fir'` skeleton provided by the ForSyDe-Atom DSP utility libraries, which constructs the the well-recognizable FIR pattern in Figure 15, i.e. a recur-farm-reduce composition. In order to do so, `fir'` needs to know *what* to fill this template with, thus we need to provide as arguments its "basic" operations, which in our case are processes operating on signals of matrices. In fact, `fir` itself is a *specialization* of the `fir'` skeleton, which defines its basic operations as corresponding functions on vectors. This feature derives from a powerful algebra of skeletons which grants them both modularity, and the possibility to transform them into semantically-equivalent forms, as we shall soon explore in section 6.

```
581 firNet :: Num a => Vector a -> SY.Signal (Cube a) -> SY.Signal (Cube a)
582 firNet coefs = fir' addSC mulSC dlySC coefs
583   where
584     addSC   = SY.comb21 (C.farm21 (+))
```

```
585     mulSC c = SY.comb11 (C.farm11 (*c))
586     dlySC   = SY.delay  (C.fanout 0)
```

2.2.3 System Process Network

The AESA process network is formed by "plugging in" together all components instantiated in the previous sections, and thus obtaining the system description in Figure 16. We do not transpose the output data, because the Doppler windows are the ones we are interested in plotting as the innermost structures.

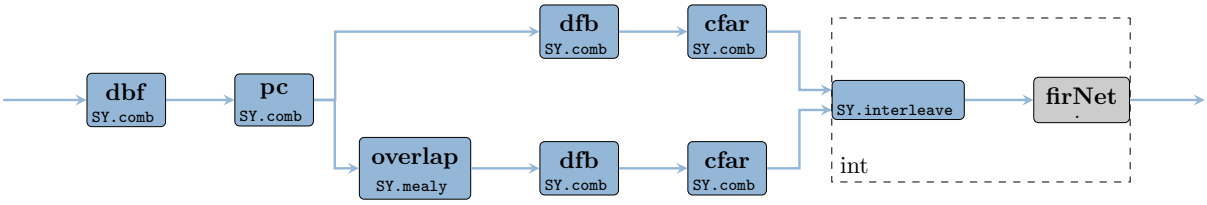


Figure 16: The AESA process network instance

```
597 aesa :: Signal (Antenna (Window (Range  CpxData )))
598      -> Signal (Beam    (Range  (Window RealData)))
599 aesa video = int rCfar lCfar
600   where
601     rCfar = cfar $ dfb oPc
602     lCfar = cfar $ dfb $ overlap oPc
603     oPc   = pc $ dbf video
```

2.2.4 System Parameters

Here we define the size constants, for a simple test scenario provided by Saab AB. The size `nA` can be inferred from the size of input data and the vector operations.

```
6  module AESA.Params where
7
8  nA    =   16 :: Int
9  nB    =    8 :: Int
10 nb    = 1024 :: Int
11 nFFT  =  256 :: Int
12 nS    =    8 :: Int -- 2^nS = nFFT; used for convenience
13 pcTap =    5 :: Int -- number of FIR taps in the PC stage
14
15 freqRadar  = 10e9 :: Float -- 10 Ghz X-band
16 waveLength = 3e8 / freqRadar
17 dElements  = waveLength/2
```

2.2.5 Coefficient Generators

Here we define the vectors of coefficients used throughout the AESA design. We keep this module as independent as possible from the main design and export the coefficients both as ForSyDe-Atom vectors but also as Haskell native lists, so that other packages can make use of them, without importing the whole ForSyDe-Atom chain.

```
8  {-# LANGUAGE PackageImports #-}
9  module AESA.Coefs where
11 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
12 import ForSyDe.Atom.Skeleton.Vector.Matrix as M
```

```
import ForSyDe.Atom.Skeleton.Vector.DSP
import Data.Complex
```

The `mkBeamConst` generator creates a matrix of α_{ij} beam constants used in the digital beamforming stage section 5.1.2.1. These beam constants perform both phase shift and tapering according to eq. 6, where c_k performs tapering and φ_{kl} perform phase shifting. For tapering we use a set of Taylor coefficients generated with our in-house utility `taylor`. The phase shift shall be calculated according to eq. 7, where d is the distance between the antenna elements. θ_l is the angle between the wave front of the current beam and normal of the antenna elements and λ is the wavelength of the pulse.

$$\alpha_{kl} = c_k e^{j\varphi_{kl}},\ \forall k \in [0, N_A - 1], l \in [0, N_B - 1] \tag{6}$$

$$\varphi_{kl} = \frac{(k - 9.5) \cdot 2\pi \cdot d \sin\theta}{\lambda} \tag{7}$$

```
mkBeamConsts :: RealFloat a
             => a                    -- ^ distance between radar elements
             -> a                    -- ^ radar signal wavelength
             -> Int                  -- ^ Number of antenna elements
             -> Int                  -- ^ Number of resulting beams
             -> Matrix (Complex a)
mkBeamConsts d lambda nA nB = M.farm21 mulScale taperingCf phaseShiftCf
  where
    -- all coefficients are normalized, i.e. scaled with 1/nA'
    mulScale x y   = x * y / nA'
    -- tapering coefficients, c_k in Eq. (4)
    taperingCf     = V.farm11 (V.fanoutn nB) taylorCf
    -- phase shift coefficients, e^(j*phi_kl) in Eqs.(4) and (5)
    phaseShiftCf   = V.farm11 (\k -> V.farm11 (mkCf k) thetas) antennaIxs
    mkCf k theta_l = cis $ (k - 9.5) * 2 * pi * d * sin theta_l / lambda
    --------------
    -- Taylor series: nA real numbers; 4 nearly constant adjacent side lobes;
    -- peak sidelobe level of -30dB
    taylorCf = taylor nA 4 (-30)
    -- theta_l spanning nB angles from 0 to pi
    thetas   = V.farm11 (\t -> pi/3 + t * (pi - 2*pi/3)/(nB'-1)) beamIxs
    --------------
    nA'       = fromIntegral nA
    nB'       = fromIntegral nB
    antennaIxs = vector $ map realToFrac [0..nA-1]
    beamIxs    = vector $ map realToFrac [0..nB-1]

-- Can be used without importing the ForSyDe.Atom libraries.
mkBeamConsts' :: RealFloat a => a -> a-> Int -> Int -> [[Complex a]]
mkBeamConsts' d l nA nB = map fromVector $ fromVector $ mkBeamConsts d l nA nB
```

Function	Original module	Package
`farm11`,`farm21`, `fanout`, (from-)`vector`	ForSyDe.Atom.Skeleton.Vector	forsyde-atom
`taylor`	ForSyDe.Atom.Skeleton.Vector.DSP	forsyde-atom-extensions

The `mkPcCoefs` generator for the FIR filter in section 5.1.2.2 is simply a n-tap Hanning window. It can be changed according to the user requirements. All coefficients are scaled with $1/n$ so that the output does not overflow a possible fixed point representation.

```
mkPcCoefs :: Fractional a => Int -> Vector a
mkPcCoefs n = V.farm11 (\a -> realToFrac a / realToFrac n) $ hanning n
```

Function	Original module	Package
`hanning`	`ForSyDe.Atom.Skeleton.Vector.DSP`	forsyde-atom-extensions

```
79
80  -- Can be used without importing the ForSyDe.Atom libraries.
81  mkPcCoefs' :: Fractional a => Int -> [a]
82  mkPcCoefs' n = fromVector $ mkPcCoefs n
```

We use also a Hanning window to generate the complex weight coefficients for decreasing the Doppler side lobes during DFB in section 2.2.2.4. This can be changed according to the user requirements.

```
88  mkWeightCoefs :: Fractional a => Int -> Vector a
89  mkWeightCoefs nFFT = V.farm11 realToFrac $ hanning nFFT

91  -- Can be used without importing the ForSyDe.Atom libraries.
92  mkWeightCoefs' :: Fractional a => Int -> [a]
93  mkWeightCoefs' nFFT = fromVector $ mkWeightCoefs nFFT
```

For the integrator FIR in section 5.1.2.5 we use a normalized square window.

```
97  mkIntCoefs :: Fractional a => Vector a
98  mkIntCoefs = vector mkIntCoefs'

100 -- Can be used without importing the ForSyDe.Atom libraries.
101 mkIntCoefs' :: Fractional a => [a]
102 mkIntCoefs' = [1/8,1/8,1/8,1/8,1/8,1/8,1/8,1/8]

104 -- The maximum floating point number representable in Haskell.
105 maxFloat :: Float
106 maxFloat = x / 256
107  where n = floatDigits x
108        b = floatRadix x
109        (_, u) = floatRange x
110        x = encodeFloat (b^n - 1) (u - n)
```

2.3 Model Simulation Against Test Data

As a first trial to validate that our AESA high-level model is "sane", i.e. is modeling the expected behavior, we test it against realistic input data from an array of antennas detecting some *known* objects. For this we have provided a set of data generator and plotter scripts, along with an executable binary created with the `CubesAtom` module presented in section 2.2. Please read the project's `README` file on how to compile and run the necessary software tools.

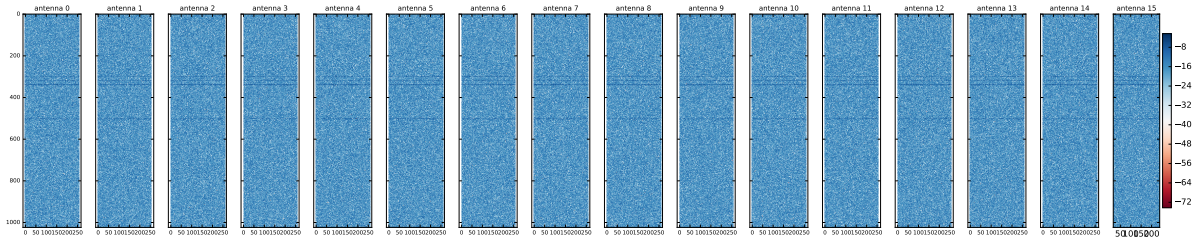
The input data generator script replicates a situation in which 13 distinct objects, either near each other or far apart, as seen in tbl. 12, are detected drowned into -18dB worth of noise by the 16 AESA antenna elements (see section 2.2.4). In in Figure 17a can be seen a plot with the absolute values of the complex samples in the first video indata cube, comprising of 16 antenna (pulse × range) matrices.

In Figure 17b a cube consisting of 8 beam (Doppler × range) matrices from the AESA signal processing output is shown. We can see the 13 objects detected with different intensities across the 8 beams. As they are all positioned at approximately the same angle relative to the antenna (i.e. $\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$) we can see the maximum correlation values are reflected in beam 2.

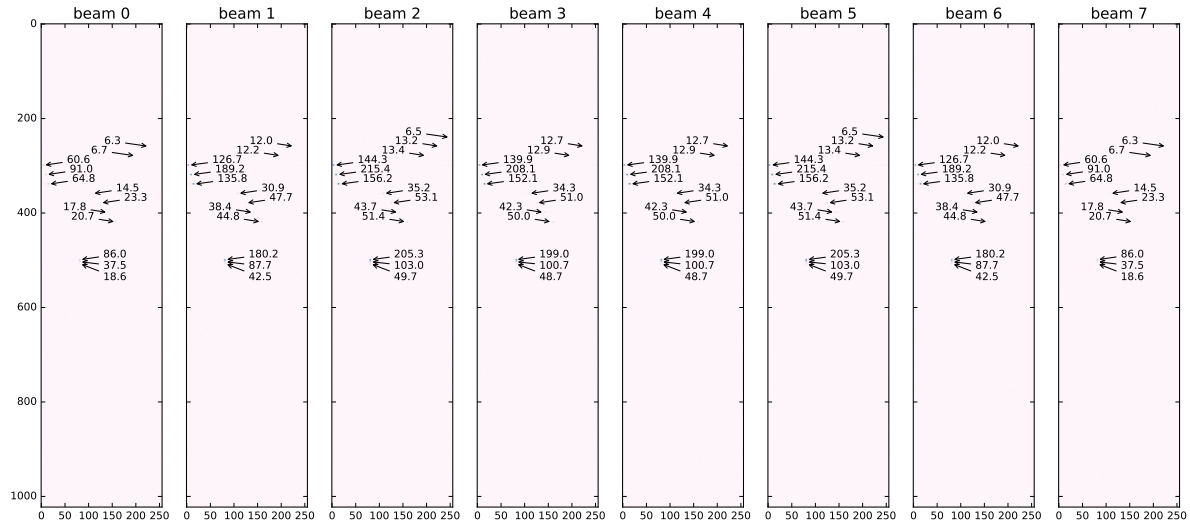
Table 12: Objects reflected in the generated AESA indata

#	Distance (m)	Angle (θ)	Rel. Speed (m/s)	Rel. Power
1	12e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	0.94	-6
2	13e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	$5 \cdot 0.94$	-6
3	14e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	$10 \cdot 0.94$	-6
4	15e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-0.94	-2
5	16e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	$-2 \cdot 0.94$	-2

#	Distance (m)	Angle (θ)	Rel. Speed (m/s)	Rel. Power
6	17e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-3 · 0.94	-2
7	18e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-20 · 0.94	-4
8	19e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-23 · 0.94	-4
9	20e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-26 · 0.94	-4
10	21e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-29 · 0.94	-4
11	25e3	$\frac{\pi}{3} + 2(\pi - \frac{2\pi}{3})/7$	-15 · 0.94	-2
12	25.4e3	$\frac{\pi}{3} + 2.1(\pi - \frac{2\pi}{3})/7$	-15 · 0.94	-4
13	25.2e3	$\frac{\pi}{3} + 2.2(\pi - \frac{2\pi}{3})/7$	-15 · 0.94	-3



(a) Absolute values for one input video cube with antenna data



(b) One output cube with radar data

Figure 17: AESA data plots

2.4 Conclusion

In this section we have shown how to write a fully-functional high-level model of an AESA radar signal processing chain in ForSyDe-Atom. In the process we have exemplified basic modeling concepts such as *layers*, *process constructors* and *skeletons*. For simplicity we have employed only two widely-used layers, representing time (through the MoC layer) and parallelism (through the Skeleton layer) aspects in a system. Within the MoC layer we used only the SY MoC capturing the passage of data (structures) in a synchronous pipeline fashion. Within the Skeleton layer we used algorithmic skeletons on vector data types to capture the inherent parallel interaction between elements, mainly to build algorithms on cubes. As of Figure 3 from section 2.1, this layer setup is represented by right picture (processes of skeleton functions). However in section 2.2.2.6 we have given "an appetizer" on how to instantiate regular, parameterizable process network structures using the *same* skeletons, thanks to the concept of layers.

In section 5 we transform this model to capture some more refined notions of time and data passage, as well as more complex use of skeletons and patterns, gradually reaching enough system details to start considering the synthesis of the model in section 6 to a hardware platform. Until then we will introduce an alternative modeling framework, as well as practical tools to verify the conformance of the ForSyDe model and all its subsequent refinements to the given specification.

3 Alternative Modeling Framework: ForSyDe-Shallow

This section follows step-by-step the same approach as section 2, but this time using the ForSyDe-Shallow modeling framework. The purpose is to familiarize the reader to the syntax of ForSyDe-Shallow should the designer prefer it instead of ForSyDe-Atom. This section is also meant to show that, except for minor syntactic differences, the same modeling concepts are holding and the user experience and API design are very similar.

Package	aesa-shallow-0.1.0	path: `./aesa-shallow/README.md`
Deps	forsyde-shallow-0.2.2	url:`http://hackage.haskell.org/package/forsyde-shallow`
	forsyde-atom-extensions-0.1.1	path: `./forsyde-shallow-extensions/README.md`
	aesa-atom-0.1.1	path: `./aesa-atom/README.md`
Bin	aesa-shallow	usage: `aesa-shallow --help`

ForSyDe-Shallow is the flagship and the oldest modeling language of the ForSyDe methodology (Sander and Jantsch 2004). It is a domain specific language (DSL) shallow-embedded into the functional programming language Haskell and uses the host's type system, lazy evaluation mechanisms and the concept of higher-order functions to describe the formal modeling framework defined by ForSyDe. At the moment of writing this report, ForSyDe-Shallow is the more "mature" counterpart of ForSyDe-Atom. Although some practical modeling concepts such as *layers*, *patterns* and *skeletons* have been originally developed within ForSyDe-Atom, they are now well-supported by ForSyDe-Shallow as well. In the future the modeling frameworks such as ForSyDe-Atom and ForSyDe-Shallow are planned to be merged into a single one incorporating the main language features of each one and having a similar user experience as ForSyDe-Shallow.

3.1 The High-Level Model

The behavioral model of this section is exactly the same as the one presented in in section 2.2, and thus we will not go through all the details of each functional block, but rather list the code and point out the syntax differences.

The code for this section is written in the following module, see section 1.2 on how to use it:

```
10 {-# LANGUAGE PackageImports #-}    -- you can ignore this line for now
```

The code for this section is written in the following module, see section 1.2 on how to use it:

```
15 module AESA.CubesShallow where
```

3.1.1 Imported Libraries

The first main difference between ForSyDe-Shallow and ForSyDe-Atom becomes apparent when importing the libraries: ForSyDe-Shallow does not require to import as many sub-modules as its younger counterpart. This is because the main library `ForSyDe.Shallow` exports all the main language constructs, except for specialized utility blocks, so the user does not need to know where each function is placed.

```
26 import ForSyDe.Shallow
```

For the AESA model we make use of such utilities, such as n-dimensional vectors (i.e. matrices, cubes) and DSP blocks, thus we import our local extended `ForSyDe.Shallow.Utilities` library.

```
33 -- | explicit import from extensions package. Will be imported
34 -- normally once the extensions are merged into forsyde-shallow
35 import "forsyde-shallow-extensions" ForSyDe.Shallow.Core.Vector
36 import "forsyde-shallow-extensions" ForSyDe.Shallow.Utility
```

Finally, we import Haskell's `Complex` type, to represent complex numbers.

```
42 import Data.Complex
```

To keep the model consistent with the design in section 2.2 we import the *same* parameters and coefficient generator functions from the `aesa-atom` package, as presented in sections 2.2.4, 2.2.5.

```
48 import AESA.Params
49 import AESA.CoefsShallow  -- wraps the list functions exported by 'AESA.Coefs' into
50                           -- 'ForSyDe.Shallow' data types.
```

3.1.2 Type Synonyms

We use the same (local) aliases for types representing the different data structure and dimensions.

```
57 type Antenna     = Vector       -- length: nA
58 type Beam        = Vector       -- length: nB
59 type Range       = Vector       -- length: nb
60 type Window      = Vector       -- length: nFFT
61 type CpxData     = Complex Float
62 type RealData    = Float
```

3.1.3 Video Processing Pipeline Stages

This section follows the same model as section 2.2.2 using the ForSyDe-Shallow modeling libraries. The digital beamforming (DBF) block presented in section 2.2.2.1 becomes:

```
70 dbf :: Signal (Antenna (Window (Range CpxData)))
71     -> Signal (Window  (Range  (Beam  CpxData)))
72 dbf = combSY (mapMat fDBF . transposeCube)
73
74 fDBF :: Antenna CpxData -- ^ input antenna elements
75       -> Beam    CpxData -- ^ output beams
76 fDBF antennas  = beams
77   where
78     beams      = reduceV (zipWithV (+)) beamMatrix
79     beamMatrix = zipWithMat (*) elMatrix beamConsts
80     elMatrix   = mapV (copyV nB) antennas
81     beamConsts = mkBeamConsts dElements waveLength nA nB
```

The second main difference between the syntax of ForSyDe-Shallow and ForSyDe-Atom can be noticed when using the library functions, such as process constructors: function names are not invoked with their module name (alias), but rather with a suffix denoting the type they operate on, e.g. `transposeCube` is the equivalent of `C.transpose`. Another difference is that constructors with different numbers of inputs and outputs are not differentiated by a two-number suffix, but rather by canonical name, e.g. `combSY` is the equivalent of `SY.comb11`. Lastly, you might notice that some function names are completely different. That is because ForSyDe-Shallow uses names inspired from functional programming, mainly associated with operations on lists, whereas ForSyDe-Atom tries to adopt more suggestive names with respect to the layer of each design component and its associated jargon, e.g. `mapMat` is the equivalent of `M.farm11`, or `zipWithV` is equivalent with `V.farm21`.

We go further with the description of the pulse compression (PC) block, as described previously in section 2.2.2.1 becomes. The same variation in naming convention can be noticed, but the syntax is still the same.

```
101 pc :: Signal (Window (Range (Beam  CpxData)))
102    -> Signal (Window (Beam  (Range CpxData)))
103 pc = combSY (mapV (mapV fPC . transposeMat))
104
105 fPC :: Range CpxData -- ^ input range bin
106     -> Range CpxData -- ^ output pulse-compressed bin
107 fPC = mav (mkPcCoefs 5)
```

The same goes for `overlap` state machine use in the corner turn (CT) stage, as well as the Doppler filter bank (DFB) and the constant false alarm ratio (CFAR) stages from sections 2.2.2.3, 2.2.2.4, 2.2.2.5 respectively.

```
overlap :: Signal (Window (Beam (Range CpxData)))
        -> Signal (Window (Beam (Range CpxData)))
overlap = mealySY nextState outDecode initState
  where
    nextState _ cube = dropV (nFFT `div` 2) cube
    outDecode s cube = s <+> takeV (nFFT `div` 2) cube
    initState        =  copyCube nb nB (nFFT `div` 2) 0

dfb :: Signal (Window (Beam  (Range  CpxData )))
    -> Signal (Beam   (Range (Window RealData)))
dfb = combSY (mapMat fDFB . transposeCube)

fDFB :: Window CpxData -> Window RealData
fDFB = mapV envelope . onComplexFloat (fft nFFT) . weight
  where
    weight     = zipWithV (*) (mkWeightCoefs nFFT)
    envelope a = let (i, q) = (realPart a, imagPart a)
                 in realToFrac $ sqrt (i * i + q * q)
    onComplexFloat f = mapV (fmap realToFrac) . f . mapV (fmap realToFrac)
```

The `fft` function from ForSyDe-Shallow is provided as a *monomorphic* utility function, and not necessarily as a skeleton. This is why, although slightly more efficient, it is not as flexible as the skeleton counterpart, and thus we need to wrap it inside our custom data type converter `onComplexFloat` to be able to "plug it" into our system, i.e. there are no data type mismatches.

```
cfar :: Signal (Beam (Range (Window RealData)))
     -> Signal (Beam (Range (Window RealData)))
cfar = combSY (mapV fCFAR)

fCFAR :: Range (Window RealData) -> Range (Window RealData)
fCFAR rbins = zipWith4V (\m -> zipWith3V (normCfa m)) md rbins lmv emv
  where
    md  = mapV (logBase 2 . reduceV min) rbins
    emv = (copyV (nFFT + 1) dummy) <+> (mapV aritMean neighbors)
    lmv = (dropV 2 $ mapV aritMean neighbors) <+> (copyV (nFFT*2) dummy)
    ------------------------------------------------
    normCfa m a l e = 2 ** (5 + logBase 2 a - maximum [l,e,m])
    aritMean  = mapV (/n) . reduceV addV . mapV geomMean . groupV 4
    geomMean  = mapV (logBase 2 . (/4)) . reduceV addV
    ------------------------------------------------
    dummy     = copyV nFFT (-maxFloat)
    neighbors = stencilV nFFT rbins
    ------------------------------------------------
    addV      = zipWithV (+)
    n         = fromIntegral nFFT
```

For the integration stage (INT) presented in section 2.2.2.3 we extended the `ForSyDe.Shallow.Utility` library with a `fir'` skeleton and an `interleaveSY` process, similar to the ones developed for ForSyDe-Atom[3]. Thus we can implement this stage exactly as before:

```
int :: Signal (Beam (Range (Window RealData)))
    -> Signal (Beam (Range (Window RealData)))
    -> Signal (Beam (Range (Window RealData)))
-- int r l = firNet $ interleaveSY r l
```

[3]at this moment it is enough to know that both are implemented in terms of existing skeletons or process constructors. More complex behaviors such as these two will be made explicit later in section 5. Interested readers can consult the implementations in the `forsyde-shallow-extensions` package.

```
--   where
--     firNet = zipCubeSY . mapCube (firSY mkIntCoefs) . unzipCubeSY
int r l = fir' addSC mulSC dlySC mkIntCoefs $ interleaveSY r l
  where
    addSC   = comb2SY (zipWithCube (+))
    mulSC c = combSY  (mapCube (*c))
    dlySC   = delaySY (repeatCube 0)
```

Function	Original module	Package
mapV, zipWith, zipWith[4/3], reduceV, lengthV, dropV, takeV, copyV, (<+>), stencilV	ForSyDe.Shallow.Core.Vector	forsyde-shallow
mapMat, zipWithMat, transposeMat	ForSyDe.Shallow.Utility.Matrix	forsyde-shallow
transposeCube	ForSyDe.Shallow.Utility.Cube	forsyde-shallow-extensions
fir	ForSyDe.Shallow.Utility.FIR	forsyde-shallow
fft	ForSyDe.Shallow.Utility.DFT	forsyde-shallow
mav	ForSyDe.Shallow.Utility.DSP	forsyde-shallow-extensions
combSY, comb2SY, mealySY, delaySY	ForSyDe.Shallow.MoC.Synchronous	forsyde-shallow
interleaveSY	ForSyDe.Shallow.Utility	forsyde-shallow-extensions
mkBeamConsts,mkPcCoefs, mkWeightCoefs, mkFirCoefs, maxFloat	AESA.Coefs	aesa-atom
dElements, waveLength, nA, nB nFFT, nb	AESA.Params	aesa-atom

3.1.4 The AESA Process Network

The process network is exactly the same as the one in section 2.2, but instantiating the locally defined components.

```
aesa :: Signal (Antenna (Window (Range  CpxData )))
     -> Signal (Beam    (Range  (Window RealData)))
aesa video = int rCfar lCfar
  where
    rCfar = cfar $ dfb oPc
    lCfar = cfar $ dfb $ overlap oPc
    oPc   = pc $ dbf video
```

3.2 Model Simulation Against Test Data

Similarly to the ForSyDe-Atom implementation, we have provided a runner to compile the model defined in section 2.2 within the `AESA.CubesShallow` module into an executable binary, in order to tests that it is sane. The same generator an plotter scripts can be used with this binary, so please read the project's `README` file on how to compile and run the necessary software tools.

We use the same generated input data reflecting the 13 objects from tbl. 12, plotted in Figure 17a. This time the AESA radar processing output shows the picture in Figure 18.

Comparing Figure 18 with Figure 17b one can see that the same objects have been identified, albeit with slightly different correlation values. This is because we use single-precision floating point `Float` as the base number representation in our model, which are well-known to be very sensitive to slight variations in arithmetic operations, as far as the order of evaluation is concerned. The output data graphs reflect the fact that ForSyDe-Shallow and ForSyDe-Atom have different implementation, but that is the only conclusion that can be drawn from this.

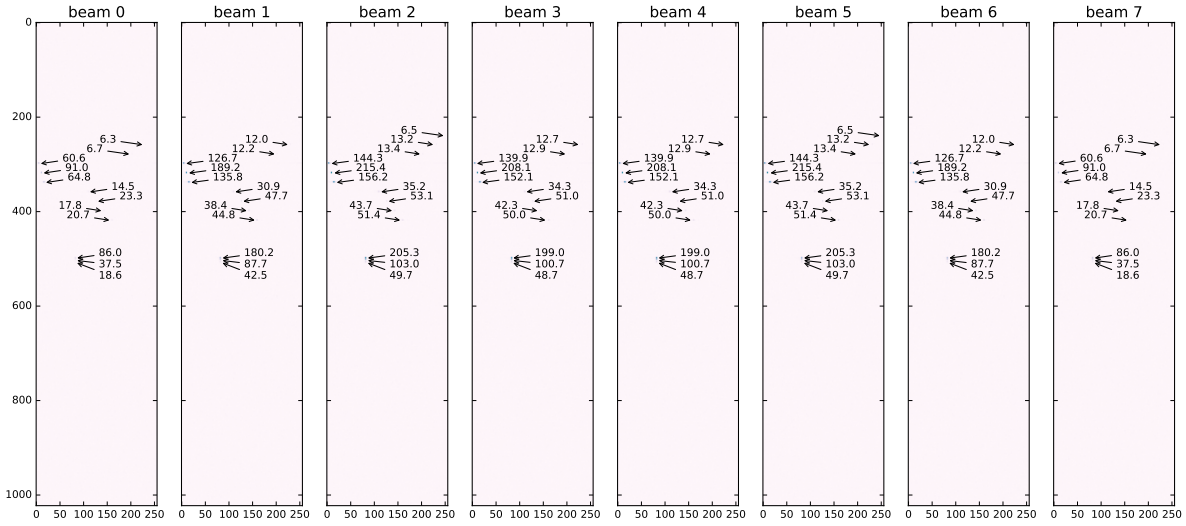


Figure 18: One output cube with radar data

3.3 Conclusion

In this section we have presented an alternative implementation of the AESA signal processing high-level model in the ForSyDe-Shallow modeling framework. We have deliberately implemented the *same* model in order to highlight the API differences between ForSyDe-Atom and ForSyDe-Shallow.

The next sections will only use ForSyDe-Atom as the main modeling framework, however users should hopefully have no trouble switching between their modeling library of choice. The insights in the API of ForSyDe-Shallow will also be useful in section 6 when we introduce ForSyDe-Deep, since the latter has similar constructs.

4 Validating a ForSyDe Model Against the Specification

This section presents a practical and convenient method for verifying the conformance of a ForSyDe model against a set of specification properties. In order to do so we make use of the QuickCheck framework, which offers an EDSL for specifying system model properties and a random test-case generation engine for validating a design under test (DUT) against these properties. Although the verification is not exhaustive, it is "smart" enough to quickly identify and correct discrepancies between specification and model implementation, making QuickCheck an ideal tool in a system designer's toolbox.

Package	aesa-atom-0.1.0	path: `./aesa-atom/README.md`
Deps	forsyde-atom-0.2.2	url: `https://forsyde.github.io/forsyde-atom/api/`
	forsyde-atom-extensions-0.1.1	path: `./forsyde-atom-extensions/README.md`
	QuickCheck-2.13.1	url: `http://hackage.haskell.org/package/QuickCheck`
Suite	tests-cube	usage: `stack test :tests-cube`

QuickCheck (Claessen and Hughes 2011) is a DSL embedded in Haskell for specifying denotational properties of functions, and a test case generator engine based on type algebras to obtain a good coverage of a DUT. As it is based on type algebras, it uses the insights from the DUT's type constructors to build automatic or user-guided strategies for identifying edge or violating cases within a relatively small number of test cases. Due to its random nature it is capable of finding failing sequences of inputs, which are then *shrinked* to define minimum violating cases, making it very useful in development iterations.

A very good article motivating QuickCheck's usefulness in system design is found on Joe Nelson's blog, from which we extract the following paragraph: "Proponents of formal methods sometimes stress the notion of specification above that of implementation. However it is the inconsistencies between these two independent descriptions of the desired behavior that reveal the truth. We discover incomplete understanding in the specs and bugs in the implementation. Programming does not flow in a single direction from specifications to implementation but evolves by cross-checking and updating the two. Property-based testing quickens this evolution. Of course the only way to truly guarantee properties of programs is by mathematical proof. However property-based tests approximate deductive confidence with less work by checking properties for a finite number of randomized inputs called test cases."

DISCLAIMER: this section assumes that the reader is familiar with QuickCheck's syntax and has gone through a few hands-on tutorials. For more information we recommend checking out either the article of Hughes (2007), John Nelson's blog article on QuickCheck, or the (slightly outdated) official QuickCheck tutorial.

4.1 Formal Notation

Properties in the QuickCheck DSL are formulated as *pre-condition* $\Rightarrow$ *statement*, where pre-conditions are defining generators for legal inputs under which the statement is evaluated.

In order to ease the understanding of the QuickCheck code, for each property we will also provide a short-hand mathematical notation using the *pre-condition* $\Rightarrow$ *statemen* format, using the following conventions:

Notation	Meaning
$\langle \alpha \rangle$	`Vector a`
$\lvert v \rvert$	`length v`
$\overline{s}$	a signal with events with the type of `s`
$\langle a, b, c, ... \rangle$	`vector [a,b,c,...]`
$\{a, b, c, ...\}$	`signal [a,b,c,...]`
$\Sigma(s)$ or $\Sigma(v)$	the (ordered) sequence with the elements from a signal `s` or a vector `v`

4.2 Properties

In this subsection we formulate a handful of properties whose purpose is:

1. to test that the model implementation from section 2.2 does not violate in any circumstance these "contracts"; and
2. ensure that any future (iterative) model refinement does not alter or violate these "contracts".

Below you find the code written in a runnable module found at `aesa-atom/test`, which constitute the `:tests-cube` suite:

```
14 {-# LANGUAGE PackageImports #-}
15 module SpecCube where
```

4.2.1 Imports

A couple of modules need to be imported before we can proceed. The QuickCheck and Framework modules a provide the test DSL as well as a couple of handy utilities for writing and handling test suites.

```
24 import Test.QuickCheck as QC
25 import Test.QuickCheck.Function
26 import Test.Framework
27 import Test.Framework.Providers.QuickCheck2 (testProperty)
```

We import some relevant ForSyDe-Atom modules, mainly to get access to the internal structure of ForSyDe types such as `Vector` or `Signal`.

```
32 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
33 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
34 import ForSyDe.Atom.Skeleton.Vector.Matrix (Matrix, size)
35 import ForSyDe.Atom.Skeleton.Vector.DSP (fir)
```

Obviously, we need to import the AESA designs modules as well.

```
39 import AESA.CubesAtom
40 import AESA.Coefs
41 import AESA.Params
```

Finally, we import some in-house data generators which will be used in formulating the pre-conditions below, as well as some Haskell data type libraries. The generators are further documented in section 4.2.4.

```
47 import Generators
48 import Data.List as L
49 import Data.Complex
```

4.2.2 Formulations

The first property we want to check is that the main function in DBF (see section 2.2.2.1), f_{DBF} will always yield n_B beam samples, no matter what or how many inputs it has. Using the notation from section 4.1 we can formalize this property as follows:

$$\forall v \in \langle \mathbb{C} \rangle : |v| > 0 \Rightarrow |f_{DBF}(v)| = n_B \tag{8}$$

Using our custom generator `nonNullVector` (see section 4.2.4) which satisfies $\forall v \in \langle \alpha \rangle : |v| > 0$. we translate eq. 8 to QuickCheck code:

```
65 prop_dbf_num_outputs = forAll (nonNullVector arbitrary)
66                        $ \v -> V.length (fDBF v) == nB
```

This property ensures that the `Beam` dimension is respected, but what about the `Range` and `Pulse` dimensions of the indata video cube? These dimensions must not be altered. This is easy to prove, since the vector on those dimensions undergo a set of nested `farm` transformations, which *ensure by definition*

that they do not alter the structure of the input data. However, let us be skeptical and assume that the library might be faulty, since `farm` is such an essential skeleton in the AESA system. A property which verifies that `farm` does not alter the structure of its input type, and thus any n-dimensional vector (e.g. `Matrix`, `Cube`) undergoing a `farm11` keeps its original dimensions, can be formulated as:

$$\forall v \in \langle \mathbb{C} \rangle \Rightarrow |\texttt{farm11}(f, v)| = |v| \tag{9}$$

```
prop_generic_farm_structure :: Fun Int Int -> Vector Int -> Bool
prop_generic_farm_structure f v = V.length v == V.length (farm11 (apply f) v)
```

Notice that if there are no special pre-conditions for the input data, e.g. $\forall v \in \langle \mathbb{C} \rangle$, we don't need to invoke our own generator with the `forAll` keyword, but rather just specify the input type and QuickCheck will automatically call the default `arbitrary` generator for that particular type. Also, we enable QuickCheck to generate *arbitrary unary functions* along with arbitrary data, by using its `Fun a b = Fun {apply :: a -> b}` function wrapper.

Another property we want to make sure is not violated during any stage in the refinement flow is that the DBF block does not produce overflown numbers, i.e. the beam coefficients are well-scaled. At this abstraction level, we do not really consider the number representation, and for what it's worth we can assume that our type `CpxData` is in fact $\mathbb{C} = a + bi$ where $a, b \in \mathbb{R}$. However, we *know* from the specifications that the input values $\forall a \in \mathbb{C} : a >= -1 - i \wedge a < 1 + i$ and that we eventually need to find efficient implementation for these number representations. The engineering intuition/experience tells that a more efficient representation would deal only with decimal numbers and would not need to be concerned with the integer part (e.g. Qn fixed point representation). Thus, at the functional level we need to ensure that the outputs themselves remain within the $[-1 - i, 1 + i)$ value pool as well, in order to avoid overflow in an arbitrary number representation.

For the function f_{DBF}, the property ensuring legal value bounds would be formulated as:

$$\forall v \in \langle \mathbb{C} \rangle, a \in v, b \in f_{DBF}(v) : |v| > 0 \wedge a \in [-1 - i, 1 + i) \Rightarrow b \in [-1 - i, 1 + i) \tag{10}$$

which translates into QuickCheck code[4] to:

```
prop_dbf_value_range = forAll (nonNullVector decimalCpxNum)
                       $ \v -> all (withinRangeComplex (-1) 1) $ V.fromVector (fDBF v)
```

Recall that in section 2.2.2.1 we have said that f_{DBF} is the equivalent of a simple vector-matrix dot operation, and provided a simplified definition f'_{DBF} using the library-provided `dotvm` function. But *are* f_{DBF} and f'_{DBF} really equivalent? We test this out by formulating a property:

$$\forall v \in \langle \mathbb{C} \rangle \Rightarrow f_{DBF}(v) = f'_{DBF}(v) \tag{11}$$

```
prop_dbf_func_equiv = forAll (nonNullVector arbitrary)
                      $ \v -> fDBF v == fDBF' v
```

Next we target the PC component (see section 2.2.2.2). The problem of dimension preserving has been solved by the previous `prop_generic_farm_structure` however, keeping our skeptical attitude, we want to check that the library-provided `fir` function performing a moving average, does not alter the dimensions of the input data

$$\forall v \in \langle \mathbb{C} \rangle \Rightarrow |f_{PC}(v)| = |v| \tag{12}$$

```
prop_pc_num_outputs :: Vector CpxData -> Bool
prop_pc_num_outputs v = V.length v == V.length (fPC v)
```

or that it is indeed having the right response to an impulse sequence:

$$\forall v \in \langle \mathbb{C} \rangle, i = \langle 1, 0, ... \rangle \Rightarrow \texttt{fir}(v, i) = v \tag{13}$$

```
prop_pc_fir_response :: Vector CpxData -> Bool
prop_pc_fir_response v = and $ zipWith (==) coefs response
```

[4] the `decimalCpxNum` generator and `withinRangeComplex` utility are defined in the `Generators` module, see section 4.2.4

```
147   where
148     coefs    = fromVector v
149     response = L.reverse $ fromVector $ fir v impulse
150     impulse  = V.vector $ L.reverse $ 1 : replicate 100 0
```

Furthermore we need to check that the FIR coefficients are scaled correctly and the outputs are within the legal range to avoid future possible overflows.

$$\forall v \in \langle \mathbb{C} \rangle, a \in v, b \in f_{PC}(v) : |v| > 0 \wedge a \in [-1-i, 1+i) \Rightarrow b \in [-1-i, 1+i) \tag{14}$$

```
158 prop_pc_value_range = forAll (nonNullVector decimalCpxNum)
159                       $ \v -> all (withinRangeComplex (-1) 1) $ V.fromVector (fPC v)
```

Checking the `overlap` process in section 2.2.2.2 is a quite tricky to check due to the structure of the data: we would need to access and compare data wrapped in nested vectors with large dimensions, meaning that it will be a very slow process. The 50% overlap will be able to be tested more easily using a random test generator further in section 5. For now we test that the cube dimensions are preserved by the `overlap` function:

$$\forall c \in \langle\langle\langle \mathbb{C} \rangle\rangle\rangle, o \in \texttt{overlap}(\overline{c}) : |c| = (n_b, n_B, n_{FFT}) \Rightarrow |o| = (n_b, n_B, n_{FFT}) \tag{15}$$

```
171 prop_ct_dimensions = forAll (sizedCube nFFT nB nb arbitrary)
172                     $ \c -> dimensionsMatch $ overlap $ SY.signal [c]
173   where
174     dimensionsMatch s = let c = L.head $ SY.fromSignal $ s
175                             z = V.length c
176                             y = V.length $ V.first c
177                             x = V.length $ V.first $ V.first c
178                         in z == nFFT && y == nB && x == nb
```

From DFB onward we deal with Doppler values which have a higher range, so we are no longer concerned with testing for overflowing output data, until we take a decision in the refinement process to fix a particular number representation. However, we would still like to test that the format and dimensions of the intermediate cubes is the same, thus we formulate:

```
186 prop_dfb_num_outputs = forAll (sizedVector nFFT arbitrary)
187                       $ \v -> V.length v == V.length (fDFB v)

189 prop_cfar_num_outputs = forAll (nonNullVector $ nonNullVector arbitrary)
190                        $ \m -> size m == size (fCFAR m)
```

The DFB contains a FFT function which is hard-coded to work for n_{FFT} samples thus we need to fix the input size accordingly. For the CFAR output, we use the `Matrix size` utility.

We have tested that the `fir` implementation gives the correct impulse response, but what about our `fir'` network instantiation `firNet` defined in section 2.2.2.6? Does it act like a proper FIR filter? We test it by giving it an "impulse cube" signal, and test that the response is the expected one:

$$\forall v \in \langle \mathbb{R} \rangle, c_1, c_0 \in \langle\langle \mathbb{R} \rangle\rangle, e_1 \in c_1, e_0 \in c_0 : e_1 = 1 \wedge e_0 = 0 \Rightarrow \texttt{firNet}(v, \{c_1, c_0, c_0, ...\}) = \overline{s_v} \tag{16}$$

where $\overline{s_v}$ is the response signal whose events are events are cubes containing the coefficients in v. The generator `impulseSigOfCubes` is, again, defined in section 4.2.4.

```
209 prop_int_fir_response :: V.Vector Int -> Property
210 prop_int_fir_response cf = forAll (impulseSigOfCubes $ V.length cf)
211                           $ \i -> correctResponse (firNet cf i)
212  where
213     correctResponse r = and $ zipWith checkEq coefsL (toLists r)
214     checkEq i = all (all (all (==i)))
215     coefsL    = L.reverse $ V.fromVector cf
216     toLists   = map (map (map V.fromVector . V.fromVector) . V.fromVector) . fromSignal
```

4.2.3 Main function

Finally we gather all the properties defined in this section in a bundle of tests called "Cube HL Model Tests", using the utilities provided by the `Test.Framework` library. `withMaxSuccess` determines how many random tests will be generated per property during one run.

```
tests :: [Test]
tests = [
  testGroup " Cube HL Model Tests "
    [ testProperty "GENERIC farm does not alter the input structure         "
      (withMaxSuccess 100 prop_generic_farm_structure)
    , testProperty "DBF  right number of outputs                            "
      (withMaxSuccess 100 prop_dbf_num_outputs)
    , testProperty "DBF  legal value range                                  "
      (withMaxSuccess 200 prop_dbf_value_range)
    , testProperty "DBF  equivalence with simple dot product operation      "
      (withMaxSuccess 200 prop_dbf_func_equiv)
    , testProperty "PC   right number of outputs                            "
      (withMaxSuccess 100 prop_pc_num_outputs)
    , testProperty "PC   right unit impulse response                        "
      (withMaxSuccess 100 prop_pc_fir_response)
    , testProperty "PC   legal value range                                  "
      (withMaxSuccess 200 prop_pc_value_range)
    , testProperty "CT   both channels have cubes of the same dimensions    "
      (withMaxSuccess 100 prop_ct_dimensions)
    , testProperty "DFB  right number of outputs                            "
      (withMaxSuccess 100 prop_dfb_num_outputs)
    , testProperty "CFAR right number of outputs                            "
      (withMaxSuccess 100 prop_cfar_num_outputs)
    , testProperty "INT  right unit impulse response                        "
      (withMaxSuccess  70 prop_int_fir_response)
    ]
  ]

main :: IO()
main = defaultMain tests
```

4.2.4 Data Generators

The data generators used to formulate pre-conditions in this section, as well as a couple of utility functions are defined in this in-house module found at `aesa-atom/tests`. The documentation for each function is provided as in-line comments.

```
{-# LANGUAGE PackageImports #-}
module Generators where

import Test.QuickCheck as QC
import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
import qualified ForSyDe.Atom.Skeleton.Vector.Matrix as M
import qualified ForSyDe.Atom.Skeleton.Vector.Cube as C
import ForSyDe.Atom.MoC.Stream
import Data.Complex
import Data.List as L

-- | Generator for complex numbers within the range $[-1-i,1+i)$
decimalCpxNum :: Gen (Complex Float)
decimalCpxNum = do
```

```
25    realPart <- choose (-1,0.99999999999)
26    imagPart <- choose (-1,0.99999999999)
27    return (realPart :+ imagPart)

30  -- | Generates non-null vectors, i.e. which satisfy 'forall v . length v > 0'.
31  nonNullVector :: Gen a -> Gen (Vector a)
32  nonNullVector a = do
33    ld <- listOf a `suchThat` (not . L.null)
34    return $ V.vector ld

36  -- | Generator for vector of fixed size.
37  sizedVector :: Int -> Gen a -> Gen (Vector a)
38  sizedVector n a = do
39    v <- QC.vectorOf n a
40    return $ V.vector v

42  -- | Generator for cube of fixed size.
43  sizedCube :: Int -> Int -> Int -> Gen a -> Gen (C.Cube a)
44  sizedCube z y x  = sizedVector z . sizedVector y . sizedVector x

46  -- | Generator for a SY signal
47  sigOfSmallCubes :: Gen a -> Gen (SY.Signal (C.Cube a))
48  sigOfSmallCubes g = do
49    x <- choose (2, 20)  -- do not choose too large dimensions otherwise
50    y <- choose (2, 20)  -- the tests will take too long... small tests are
51    z <- choose (2, 20)  -- good enough
52    sigData <- listOf1 $ sizedCube z y x g
53    return (SY.signal sigData)

55  -- | Generator for an impulse signal of (small) cubes:
56  impulseSigOfCubes :: Int -> Gen (SY.Signal (C.Cube Int))
57  impulseSigOfCubes n = do
58    x <- choose (2, 20)  -- do not choose too large dimensions otherwise
59    y <- choose (2, 20)  -- the tests will take too long... small tests are
60    z <- choose (2, 20)  -- good enough
61    impulse <- sizedCube z y x $ elements [1]
62    trail   <- sizedCube z y x $ elements [0]
63    return (SY.signal (impulse : replicate n trail))

65  -- | The default 'Arbitrary' instance for the different ForSyDe-Atom types used in
66  -- the report.
67  instance Arbitrary a => Arbitrary (V.Vector a) where
68    arbitrary = do
69      x <- arbitrary
70      return (V.vector x)
71
72  instance Arbitrary a => Arbitrary (Stream a) where
73    arbitrary = do
74      x <- arbitrary
75      return (stream x)
76
77  instance Arbitrary a => Arbitrary (SY.SY a) where
78    arbitrary = do
79      x <- arbitrary
80      return (SY.SY x)
81
82  instance Arbitrary a => Arbitrary (SDF.SDF a) where
83    arbitrary = do
84      x <- arbitrary
85      return (SDF.SDF x)

87  -- | Utility which tests whether a complex number is within the range $[-1-i,1+i)$
```

```haskell
withinRangeComplex :: Ord a => a -> a -> Complex a -> Bool
withinRangeComplex a b c
  | realPart c <  a = False
  | imagPart c <  a = False
  | realPart c >= b = False
  | imagPart c >= b = False
  | otherwise = True
```

4.3 Running the Test Suite. Conclusion

The file shown in section 4.2 can be compiled using any means available. We have included it as a test suite for the `aesa-atom` package. Please refer to the package's `README` file for instructions on how to compile an run the tools.

As expected (in a public report) the test suite output looks like below:

```
aesa-atom-0.1.0.0: test (suite: tests-cube)

 Cube HL Model Tests :
  GENERIC farm does not alter the input structure         : [OK, passed 100 tests]
  DBF  right number of outputs                            : [OK, passed 100 tests]
  DBF  legal value range                                  : [OK, passed 200 tests]
  DBF  equivalence with simple dot product operation      : [OK, passed 200 tests]
  PC   right number of outputs                            : [OK, passed 100 tests]
  PC   right unit impulse response                        : [OK, passed 100 tests]
  PC   legal value range                                  : [OK, passed 200 tests]
  CT   both channels have cubes of the same dimensions    : [OK, passed 100 tests]
  DFB  right number of outputs                            : [OK, passed 100 tests]
  CFAR right number of outputs                            : [OK, passed 100 tests]
  INT  right unit impulse response                        : [OK, passed 70 tests]

         Properties   Total
 Passed  11           11
 Failed  0            0
 Total   11           11

aesa-atom-0.1.0.0: Test suite tests-cube passed
```

However it should not be taken for granted that during the development process the printed output is *this* flawless. As mentioned in this section's introduction, like any other software project, a ForSyDe model grows organically with consecutive iterations between system specification and model implementation, often held accountable by different parties or groups. Along with traditional unit tests (not covered by this report), property checking is a powerful tool which shortens these iteration cycles by setting a formal "legal frame" within which each (sub-)component in a a model needs to operate. Apart from helping in understanding the specification better, these properties ensure that further in the refinement process we do not mistakenly introduce new flaws. This will be demonstrated in the next sections.

5 Refining the Model Behavior. A Streaming Interpretation of AESA

In this section we refine the behavior of the high-level AESA model presented in section 2 focusing on a more fine-grained timed (i.e. streaming) aspects of the computation. We do this in an attempt to expose the inherent parallelism of the AESA application, by describing it as parallel networks of concurrent, independent processes operating on streaming elements rather than monolithic blocks operating on cubes. This perspective should pave the way for a more efficient exploration of the available design space in future refinement decisions. We present an alternative streaming-oriented model of the AESA system, validate it against test data and verify the properties previously formulated.

Package	aesa-atom-0.1.0	path: `./aesa-atom/README.md`
Deps	forsyde-atom-0.2.2	url: `https://forsyde.github.io/forsyde-atom/api/`
	forsyde-atom-extensions-0.1.1	path: `./forsyde-atom-extensions/README.md`
	QuickCheck-2.13.1	url: `http://hackage.haskell.org/package/QuickCheck`
Suite	tests-stream	usage: `stack test :tests-stream`
Bin	aesa-stream	usage: `aesa-stream --help`

The high-level behavioral model of the AESA signal processing chain presented in this section is semantically equivalent to the one presented in section 2, however it exposes a more fine-grained temporal behavior of the individual operations performed on the complex indata samples. While most design choices are driven by a didactic purpose to consolidate the new modeling concepts presented in section 2.1, they can also be justified by the need to capture the essential properties in a formal way, in order to be exploitable in future stages of a design flow towards efficient implementations. The main design approach is to exploit the relative independence between each data path associated with each antenna element or beam, and to model these paths as skeletons (e.g. farms) of (chains of) processes. Each design decision will infer a re-partitioning of the indata cubes between the time (i.e. causality) and space dimensions following the design patters depicted earlier in Figure 3, namely:

- *skeletons of processes* which: 1) express parallelism at the process level; 2) depicts processes as operating on elementary streams of data, e.g. originating from each antenna element in particular, and skeletons as the structured interactions between these streams; 3) expose a fine-grained modular view allowing to quantify the potential for *load distribution*, since each "operation" (i.e. process) clearly captures the aspect of precedence constraints.
- *process of skeletons* which : 1) express parallelism at the datum level; 2) depicts processes as operating on structures of data (e.g. vectors, matrices or cubes); 3) expose a monolithic view of processes where precedence constraints are expressed "outside" of the algorithm, and where the algorithm itself expresses potential for *data parallelism*.

5.1 The High-Level Model

The code for this section is written in the following module, see section 1.2 on how to use it:

```
7 {-# LANGUAGE PackageImports #-} --can be ignored
8 module AESA.StreamsAtom where
```

5.1.1 Libraries and Aliases

We import exactly the same libraries as in section 2.2, so we don't need to explain what each one does. However, in this model we introduce a new MoC. For describing streaming behavior of the application our design will use a heterogeneous approach, using a combination of *synchronous reactive (SY)* processes (Lee and Seshia 2016; Benveniste et al. 2003), where the main assumption is that all events in the system are synchronized; and *synchronous data flow (SDF)* processes (Lee and Seshia 2016; Lee and Parks 1995), where the temporal behavior is formulated in terms of partial order constraints between events. We also

use the Boolean dataflow model, which is an extension to SDF to allow for dynamic behaviors (Buck and Lee 1993). We import the SY, SDF and BDF libraries described in the *MoC layer*, see (Ungureanu and Sander 2017), using an appropriate alias for each.

```
27 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
28 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
29 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.BDF as BDF
30 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector        as V
31 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.Matrix as M
32 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.DSP
33 import AESA.Coefs
34 import AESA.Params
35 import Data.Complex

37 import qualified AESA.CubesAtom as CAESA
```

We use the same internal aliases to name the different types employed in this model:

```
41 type Antenna  = Vector -- length: nA
42 type Beam     = Vector -- length: nB
43 type Range    = Vector -- length: nb
44 type Window   = Vector -- length: nFFT
45 type CpxData  = Complex Float
46 type RealData = Float
```

5.1.2 Video Processing Stages

In this section we follow each stage earlier described in described in section 2.2, and exploit the initial assumption on the order of events stating: *"For each antenna the data arrives* pulse by pulse*, and each pulse arrives* range bin by range bin*. This happens* for all antennas in parallel*, and all complex samples are synchronized with the same sampling rate, e.g. of the A/D converter."*

This allows us to "unroll" the indata video cubes into N_A parallel synchronous streams, each stream being able to be processed as soon as it contains enough data. This unrolling is depicted in Figure 19 as streaming the pulses as soon as they arrive: range bin by range bin. We say that we partition the data *in time* rather than *in space*, which is a more appropriate partition judging by the assumption above.

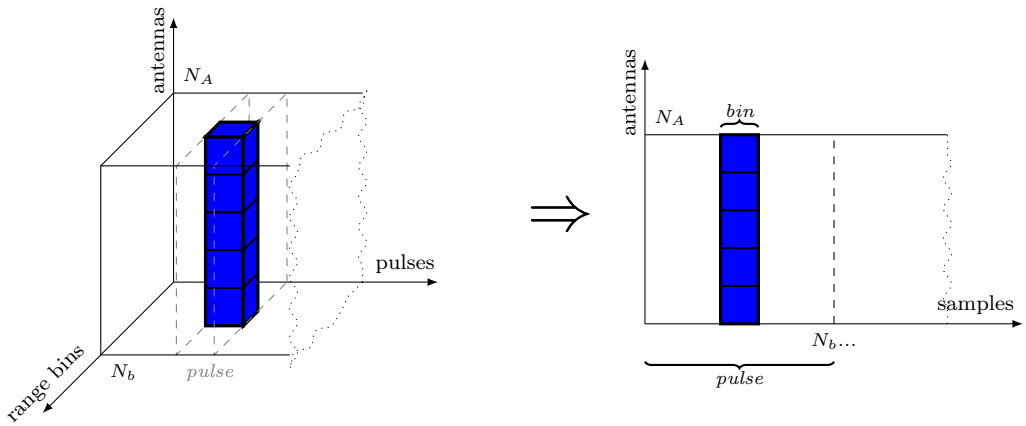


Figure 19: Video cube unrolling

5.1.2.1 Digital Beamforming (DBF)

The role of the DBF stage is explained in section 2.2.2.1. Depicted from a streaming point of view, DBF would like in Figure 20.

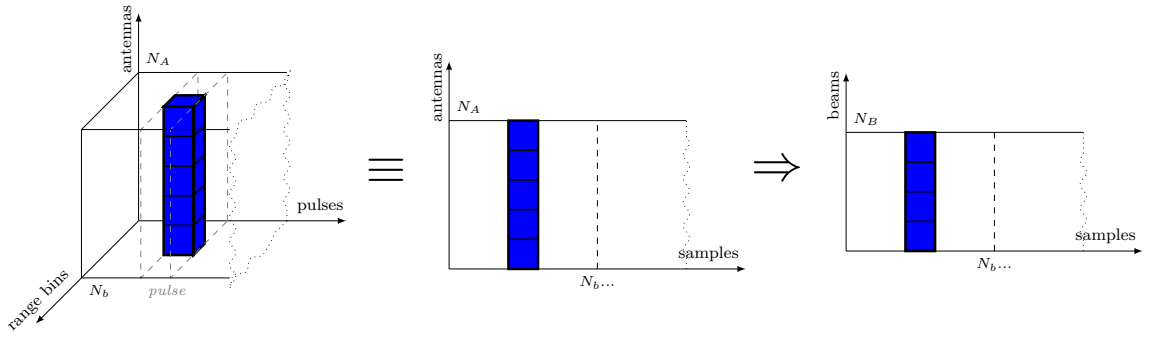


Figure 20: Digital Beam Forming on streams of complex samples

As can be seen in Figure 20, a beam can be formed *as soon as* all antennas have produced a complex sample. The parallel streams of data coming from each antenna element are represented as a *vector of synchronous (SY) signals*, i.e. vector of signals where each event is synchronous with each other. This allows us to depict the dataflow interaction between the streams during digital beamforming as the process network in Figure 21, where an $\oplus$ represents a combinational process `comb`.

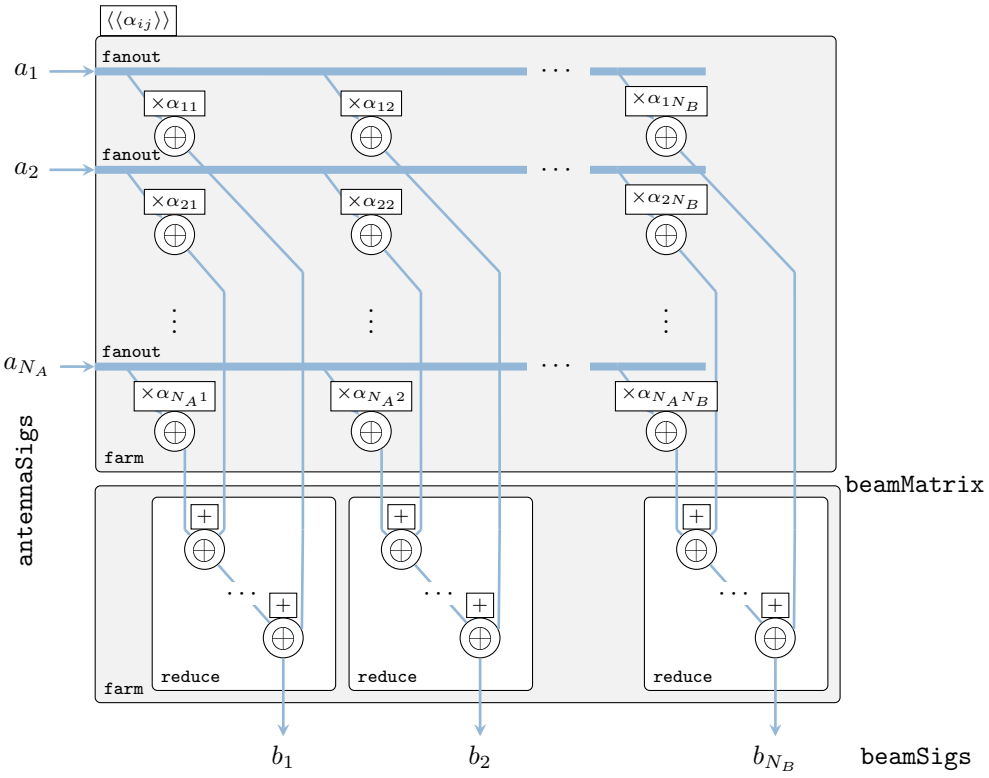


Figure 21: DBF network

```
dbf :: Antenna (SY.Signal CpxData)
    -> Beam    (SY.Signal CpxData)
dbf antennaSigs = beamSigs
  where
    beamSigs   = V.reduce (V.farm21 (SY.comb21 (+))) beamMatrix
    beamMatrix = M.farm21 (\c -> SY.comb11 (*c)) beamConsts sigMatrix
    sigMatrix  = V.farm11 V.fanout antennaSigs
    beamConsts = mkBeamConsts dElements waveLength nA nB :: Matrix CpxData
```

Function	Original module	Package
`farm11`, `reduce`, `length`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`farm21`	`ForSyDe.Atom.Skeleton.Vector.Matrix`	forsyde-atom-extensions
`comb11`, `comb21`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`mkBeamConsts`	`AESA.Coefs`	aesa-atom
`dElements`, `waveLength`, `nA`, `nB`	`AESA.Params`	aesa-atom

The previous code listing, depicted in Figure 21, is actually showing the *exact same* "internals" as the vector-matrix dot product presented in section 2.2.2.1. However, the elementary operations, instead of regular arithmetic operations $\times$ and $+$, are *processes* applying these operations on SY streams. As such, the `fanout` skeleton distributes one signal to a whole row (i.e. vector) of processes, the matrix `farm` applies pair-wise a matrix of partially applied processes on this matrix of signals, and `reduce` creates a reduction network of binary processes pair-wise applying the function $+$ on all events in signals. Practically the DBF network transforms N_A synchronous signals originating from each antenna element into N_B synchronous signals for each beam. The internal structure of this transformation exposes multiple degrees of potential distribution on parallel synchronous resources.

5.1.2.2 Pulse Compression (PC)

The role of the DBF stage is explained in section 2.2.2.2. The sliding window, or moving average (MAV), is now applied on the range bin samples in the order of their arrival.

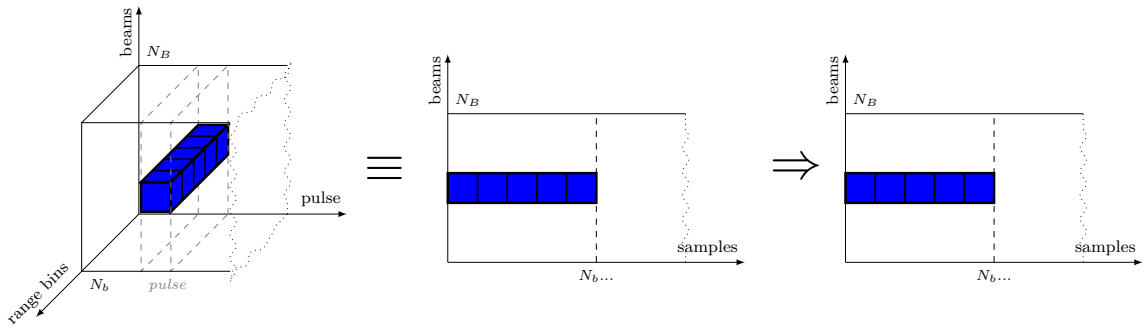


Figure 22: Pulse Compression on streams of complex samples

In Figure 22 we can see that, in order to apply the MAV algorithm on all the range bins of every pulse, we need to accumulate N_b samples and process them in batches. Intuitively this can be done by processing each beam with a *synchronous dataflow* (SDF) actor which, with each firing, consumes N_b samples and produces N_b samples. Note that at this stage of modeling we value intuition and the capturing of the right application properties rather than target implementation efficiency. We will tackle this problem later in the design stages (see section 6) where we will try to transform the model toward more efficient implementation models (with respect to some constraint, e.g. throughput) which preserve these behavioral properties.

```
132 pc :: Beam ( SY.Signal CpxData)
133    -> Beam (SDF.Signal CpxData)
134 pc = V.farm11 (procPC . SY.toSDF)
```

Function	Original module	Package
`farm11`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`toSDF`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`comb11`	`ForSyDe.Atom.MoC.SDF`	forsyde-atom
`fir`	`ForSyDe.Atom.Skeleton.Vector.DSP`	forsyde-atom-extensions
`mkPcCoefs`	`AESA.Coefs`	aesa-atom
`nb`	`AESA.Params`	aesa-atom

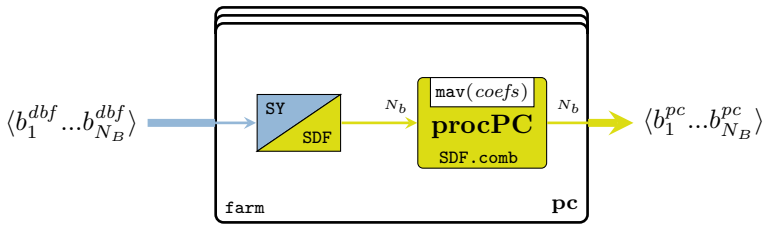


Figure 23: PC stage process

Following the reasoning above, we instantiate the PC video processing stage as a `farm` of SDF processes `procPC` as depicted in Figure 23. Notice that before being able to apply the SDF actors we need to translate the SY signals yielded by the DBF stage into SDF signals. This is done by the `toSDF` interface which is an injective mapping from the (timed) domain of a SY MoC tag system, to the (untimed) co-domain of a SDF MoC tag system. For more on tag systems please consult (Lee and Sangiovanni-Vincentelli 1998).

```
154 procPC :: Fractional a => SDF.Signal a -> SDF.Signal a
155 procPC = SDF.comb11 (nb, nb, V.fromVector . fir (mkPcCoefs pcTap) . V.vector)
```

The `procPC` actor consumes and produces `nb` tokens each firing, forms a `Vector` from these tokens, and applies the same `fir` function used in section 2.2.2.2 on these vectors. Also notice that the type signature for `procPC` is left polymorphic as to be more convenient later when we formulate properties over it.

5.1.2.3 Corner Turn (CT) with 50% overlap

The role of the CT stage is explained in section 2.2.2.3. In this model we make use of the knowledge that for each beam sample arrives in order, one range bin at a time, in the direction of consumption suggested in Figure 24, and "fill back in" the video cube in the direction of production. In order to maximize the efficiency of the AESA processing the datapath is split into two concurrent processing channels with 50% overlapped data, as shown in Figure 7 from section 2.2.2.3.

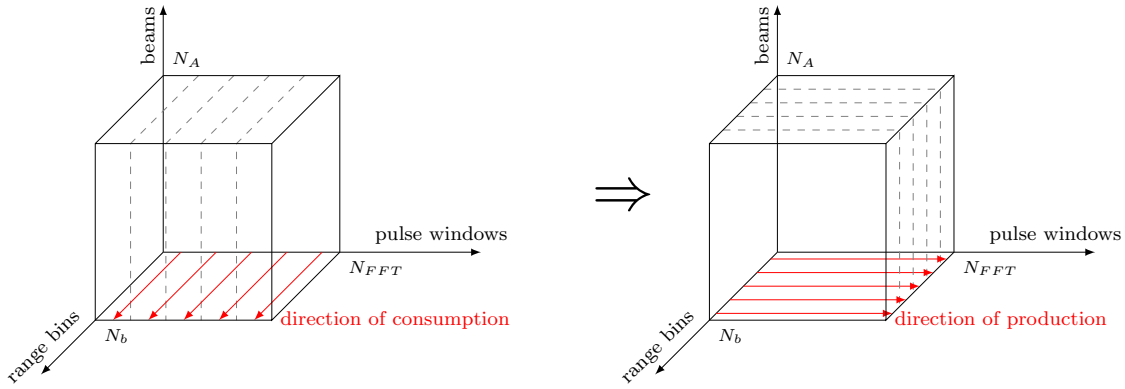


Figure 24: Building matrices of complex samples during CT

```
175 ct :: Beam (SDF.Signal CpxData)
176    -> (Beam (SDF.Signal CpxData),
177        Beam (SDF.Signal CpxData))
178 ct = V.farm12 procCT
```

Our CT network thus maps on each beam signal a corner turn process which, under the SDF execution semantics, consumes $N_{FFT} \times N_b$ ordered samples, interprets them as a matrix, transposes this matrix, and produces $N_b \times N_{FFT}$ samples ordered in the direction suggested in Figure 24.

```
185 procCT :: SDF.Signal a -> (SDF.Signal a, SDF.Signal a)
186 procCT sig = (corner rightCh, corner leftCh)
187   where
```

```
188     leftCh      = sig
189     (_,rightCh) = BDF.switch selectSig sig
190     selectSig   = SDF.delay (replicate (nb * nFFT `div` 2) True)
191                   $ SDF.constant1 [False]
192     corner      = SDF.comb11 (nFFT * nb, nb * nFFT,
193                               fromMatrix . M.transpose . matrix nb nFFT)
```

Function	Original module	Package
`farm12`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
(`from-`)`matrix`, `transpose`	`ForSyDe.Atom.Skeleton.Vector.Matrix`	forsyde-atom-extensions
`comb11`, `delay`	`ForSyDe.Atom.MoC.SDF`	forsyde-atom
`nb`, `nFFT`	`AESA.Params`	aesa-atom

Recall that, in order to achieve 50% overlapping between the two output channels, in the initial high-level model in section 2.2.2.3 we "delayed" the left channel with half a cube of "useless data" which we later ignored in the testbench. While it is possible to do the same trick here, i.e. delay the left channel with $N_b \times N_{FFT}/2$ samples per beam, which is the equivalent of half a cube, we agreed upon it being a simulation artifice to avoid undefined behavior, but not really reflecting the target implementation. The end artifact would start streaming the left channels only after the first half cube. We can think of three ways how to model such a behavior *without stepping outside the data flow paradigm*[5]:

1. "starting" the *entire* system (i.e. considering time 0) once we *know* the values for half a cube. This means that the behavior of the AESA system does not include the acquisition of the first half video cube, which would be described in its *history* (i.e. initial state, e.g. passed as top-level argument).
2. *using absent semantics*, which would be the *correct* way to define such an abstract behavior, but would render the description of the whole system more complicated, since every block would then need to be aware of the "absence" aspect of computation.
3. *using a dynamic dataflow MoC*, which is a bit more risky, because in general most of these MoCs are undecidable and might lead to deadlocks if not used properly, but are less abstract and much closer to what you would expect from a dynamic "start/stop" mechanism.

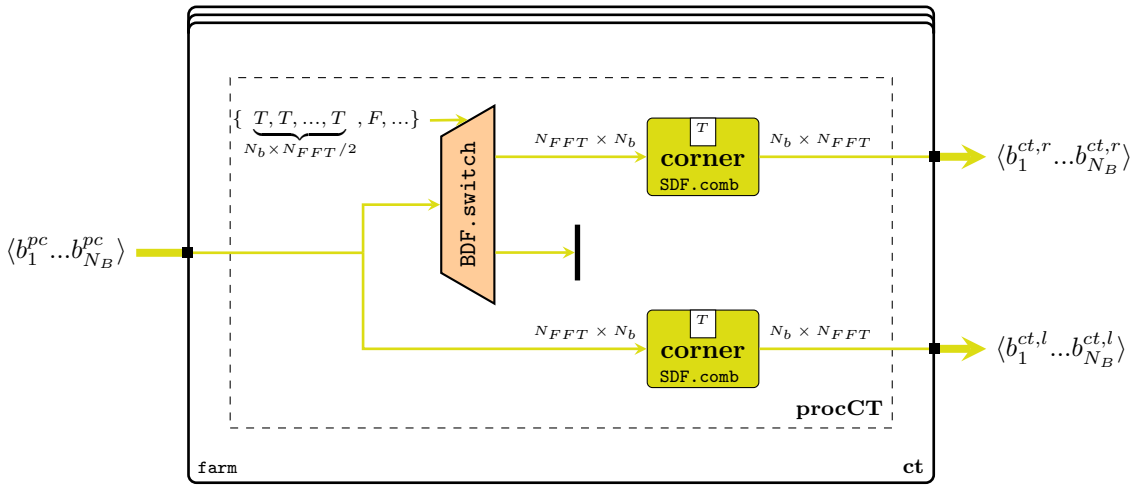


Figure 25: CT network

Since we are already considering a refined behavior model of the AESA system, we have chosen the third approach, as hinted in Figure 25. We use the Boolean dataflow (BDF) MoC introduced by Buck and Lee (1993) to express the dynamic switching behavior. BDF *extends* SDF with two actors `switch` and `select` which are able to redirect the flow of data to/from separate channels based on an input selection signal carrying Boolean tokens. We use the BDF `switch` process which uses the *hard-coded* `selectSig`

[5]this type of behavior is quite naturally expressed in other paradigms, such as communicating sequential processes (CSP), rendezvous or Petri Nets. Currently ForSyDe does not support such MoCs and they are out of the scope of this report.

signal to redirect the first $N_b \times N_{FFT}/2$ tokens for each beam (the equivalent of half a cube of indata) to a "null" channel, and only after that to start streaming into the right channel.

OBS: in general it is advised to *avoid* BDF, which does not benefit from static analysis methods for schedulability and deadlock detection, in favor of a more analyzable one, such as scenario aware dataflow (SADF) (Stuijk et al. 2011). However, in our case we base our modeling choice based on the knowledge that, lacking any type of feedback composition, the AESA signal processing system cannot cause any deadlock in any of its possible states. Even so, hard-coding the selection signal can also be considered a safe practice, because it is possible to derive a fixed set of SDF scenarios based on a known reconfiguration stream.

5.1.2.4 Doppler Filter Bank (DFB) and Constant False Alarm Ratio (CFAR)

For these two processing stages presented in sections 2.2.2.4, 2.2.2.5, we do not change the functional description at all, but rather the granularity of their timed description. We continue to work over the premise that signals are streaming samples of data, which have recently been arranged to arrive in *pulse order*, as suggested by Figure 26. As such, both stages are modeled as farms of SDF processes operating over beam streams, and for each stream they consume the necessary data, apply their (vector) function, and produce their respective data.

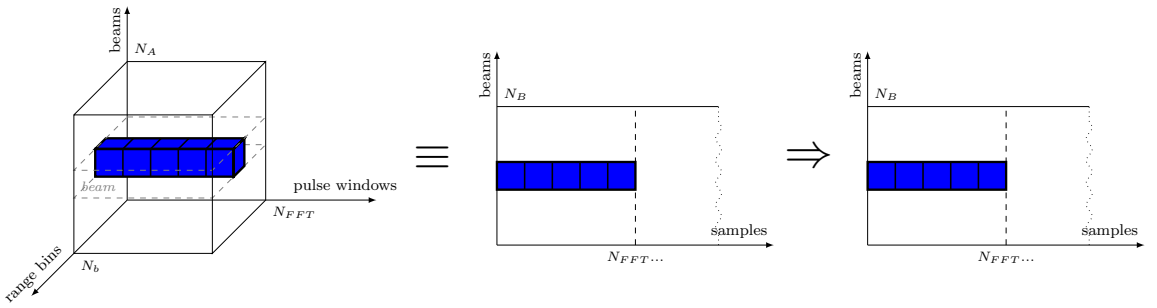


Figure 26: Doppler Filter Bank on streams of complex samples

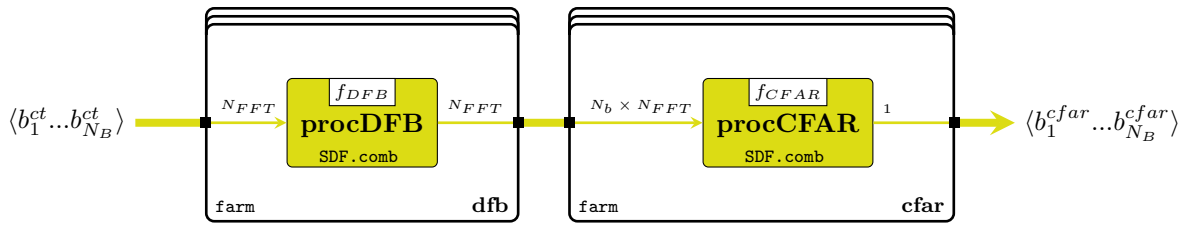


Figure 27: DFB + CFAR networks

```
dfb :: Beam (SDF.Signal CpxData)
    -> Beam (SDF.Signal RealData)
dfb = V.farm11 procDFB

cfar :: Beam (SDF.Signal RealData)
     -> Beam (SDF.Signal (Range (Window RealData)))
cfar = V.farm11 procCFAR

procDFB  = SDF.comb11 (nFFT, nFFT, fromVector . CAESA.fDFB . vector)
procCFAR = SDF.comb11 (nb * nFFT, 1, (:[]) . CAESA.fCFAR . M.matrix nFFT nb)
```

Function	Original module	Package
`farm11`,(`from-`)`vector`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`matrix`	`ForSyDe.Atom.Skeleton.Vector.Matrix`	forsyde-atom-extensions
`nb`, `nFFT`	`AESA.Params`	aesa-atom

Notice how we reuse the *exact same* functions f_{DFB} and f_{CFAR} imported from the `AESA.CubesAtom` previously defined in section 2.2. This way we are sure that we have not introduced coding errors. Also, notice that after the CFAR process we do not unwrap the matrices any longer, but produce them as *individual tokens*. This is because the subsequent INT block, as seen in the next paragraph, operates much more efficiently on matrices, and thus we spare some intermediate wrapping/unwrapping.

OBS: each function composing f_{DFB} and f_{CFAR} is itself inherently parallel, as it is described in terms of parallel skeletons. We could have "lifted" these skeletons as far as associating a process for each elementary arithmetic operation, following the example set by the DBF network in section 5.1.2.1. The two representations (if carefully modeled) are semantically equivalent and, thanks to the chosen formal description, can be transformed (ideally being aided by a computer/tool in the future) from one to another. In fact there are multiple degrees of freedom to partition the application on the time/space domains, thus a particular representation should be chosen in order to be more appropriate to the target platform model. However in this report we are not planning to refine these blocks even further, so for the purpose of simulation and visualization, this abstraction level is good enough for now. The skeleton lifting to the process network level is left as an exercise for the reader. The interested reader is also recommended to read the skeletons chapter in the technical manual (Ungureanu 2018) to see how the `fft` skeleton used in f_{DFB} is defined, and how it behaves during different instantiations, as network of processes or as function.

5.1.2.5 Integrator (INT)

During the last stage of the video processing chain each data sample of the video cube is integrated against its 8 previous values using an 8-tap FIR filter.

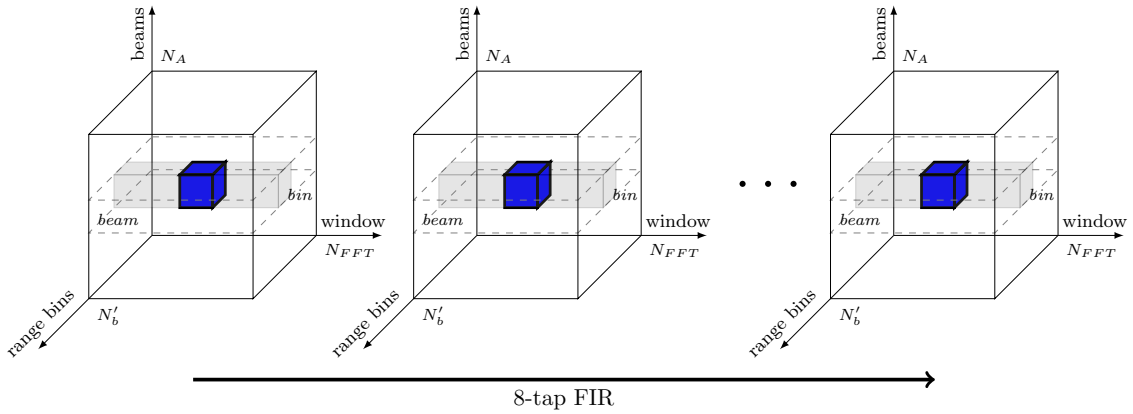


Figure 28: Integration on cubes of complex samples

The integration, re-drawn in Figure 28, like each stage until now, can be modeled in different ways based on how the designer envisions the partitioning of the data "in time" or "in space". This partitioning could be as coarse-grained as streams of cubes of samples, or as fine-grained as networks of streams of individual samples. For convenience and for simulation efficiency[6] we choose a middle approach: video cubes are represented as farms (i.e. vectors) of streams of matrices, as conveniently bundled by the previous DFB stages. We pass the responsibility of re-partitioning and interpreting the data accordingly to the downstream process, e.g. a control/monitoring system, or a testbench sink.

```
329 int :: Beam (SDF.Signal (Range (Window RealData)))
330     -> Beam (SDF.Signal (Range (Window RealData)))
331     -> Beam (SY.Signal  (Range (Window RealData)))
332 int = V.farm21 procINT
```

Please review section 2.2.2.6 concerning the up-sampler `interleave` used as a SY utility process. Here, since you have been already introduced to the SDF MoC, we can "unmask" how an interleaving process

[6] we try to avoid unnecessary transposes (i.e. type traversals) which, at the moment, are not very efficiently implemented.

actually operates underneath. It is in fact a SDF actor which consumes one token from each input and interleaves them at the output.

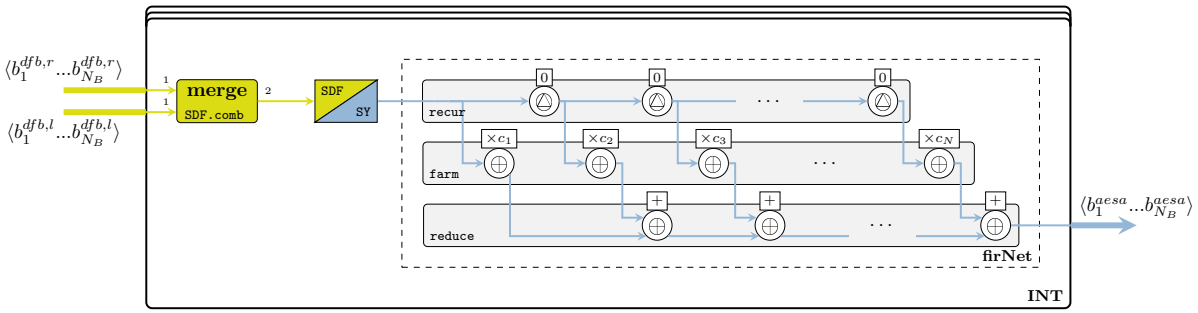


Figure 29: INT network

```
procINT :: Fractional a => SDF.Signal (Matrix a) -> SDF.Signal (Matrix a) -> SY.Signal (Matrix a)
procINT cr = firNet mkIntCoefs . SDF.toSY1 . merge cr
  where
    merge   = SDF.comb21 ((1,1), 2, \[r] [l] -> [r, l])
```

As for the FIR network, we prefer working in the SY MoC domain, which describes more naturally a streaming n-tap filter, hence we use translate back using the `toSY` MoC interface.

```
firNet :: Num a => Vector a -> SY.Signal (Matrix a) -> SY.Signal (Matrix a)
firNet coefs = fir' addSM mulSM dlySM coefs
  where
    addSM   = SY.comb21 (M.farm21 (+))
    mulSM c = SY.comb11 (M.farm11 (*c))
    dlySM   = SY.delay  (M.fanout 0)
```

Function	Original module	Package
`farm21`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`farm21,farm11,fanout`	ForSyDe.Atom.Skeleton.Vector.Matrix	forsyde-atom-extensions
`fir'`	ForSyDe.Atom.Skeleton.Vector.DSP	forsyde-atom-extensions
`comb21,comb11`	`ForSyDe.Atom.MoC.SDF`	forsyde-atom
`mkIntCoefs`	AESA.Coefs	aesa-atom

5.1.3 System Process Network

Finally, when putting all the blocks together in an equation, we obtain the system `aesa'` in Figure 30.

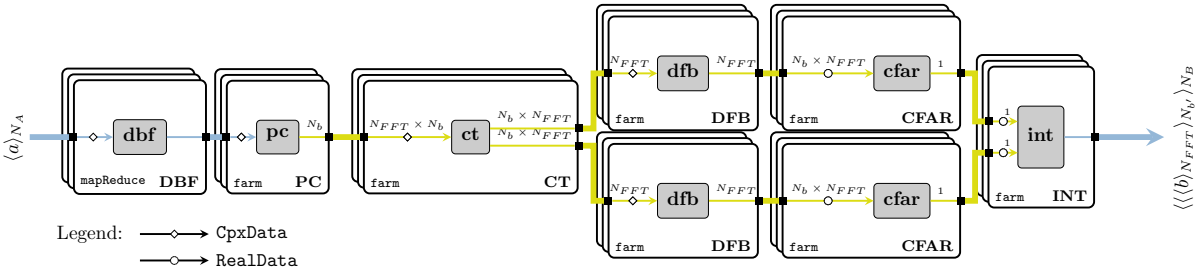


Figure 30: AESA network as black-box components

```
aesa' :: Antenna (SY.Signal CpxData) -> Beam (SY.Signal (Range (Window RealData)))
aesa' video = int rCfar lCfar
  where
    lCfar     = cfar $ dfb lCt
```

```
    rCfar      = cfar $ dfb rCt
    (rCt,lCt) = ct $ pc $ dbf video
```

At least for the sake of code elegance if not for more, we refine the system `aesa'` in Figure 30 to avoid unnecessary merging-splitting of vectors between stages, by *fusing* the `farm` skeletons between the PC and INT stages, like in Figure 31. While for simulation this transformation does not make much of a difference (thanks to Haskell's lazy evaluation mechanisms), for future synthesis stages it might be a valuable insight, e.g. it increases the potential of parallel distribution. As both models are semantically equivalent an automated or tool-assisted transformation process should be trivial.

```
aesa :: Antenna (SY.Signal CpxData) -> Beam (SY.Signal (Range (Window RealData)))
aesa = V.farm11 pcToInt . dbf

pcToInt beam = let (rb,lb) = procCT $ procPC $ SY.toSDF beam
                   lCFAR   = procCFAR $ procDFB lb
                   rCFAR   = procCFAR $ procDFB rb
               in  procINT rCFAR lCFAR
```

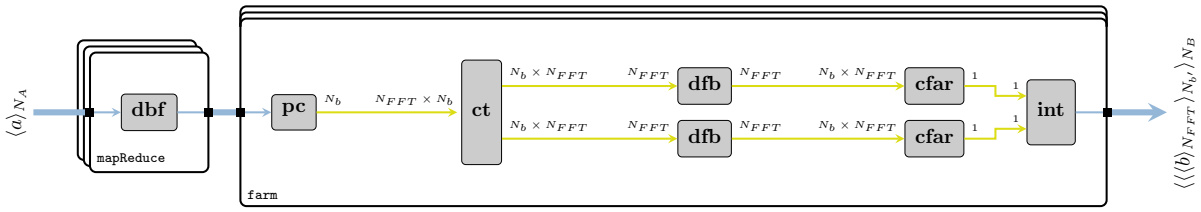


Figure 31: AESA network when fusing the related `farm`s

5.2 Model Simulation Against Test Data

Just like before in section 2.3 we test the compiled model built from the blocks defined in section 5.1 within the `AESA.StreamsAtom` module against the same generated input data. Please refer to the the project's `README` file on how to compile and run the necessary software tools.

The 13 objects described in tbl. 12 and plotted in Figure 17a are again identified in the AESA radar processing output plotted in Figure 32. Again we notice a slight difference in the cross-correlation values due to a different chaining of floating point operations, but in general the detected objects are within the same range.

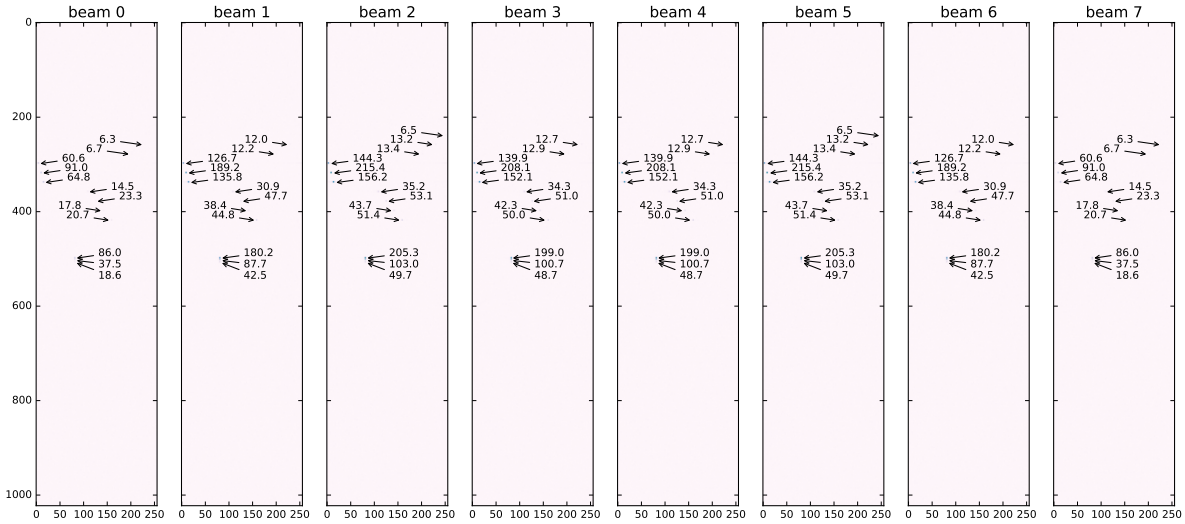


Figure 32: One output cube with radar data

5.3 Checking System Properties

In section 4 we preached the need for ways of verifying that an (executable) model satisfies a set of specification properties, and that those properties are not being violated during the refinement process. We presented `QuickCheck` as a practical and easy means for formulating properties and validating them against randomly-generated test cases. Following the same mindset, in this subsection we will try to validate through simulation[7] that the semantics of the main blocks in the AESA signal processing system defined in section 5.1 preserve the semantics of their initial high-level description from section 2.2, with respect to the properties formulated in section 4.2.

Below you find the code written in a runnable module found at `aesa-atom/test`, which constitute the `:tests-cube` suite:

```
18 {-# LANGUAGE PackageImports #-}
19 module SpecStream where
```

5.3.1 Imports

We import the necessary libraries, such as QuickCheck the two AESA model implementations `AESA.CubesAtom` and `AESA.StreamsAtom`, some ForSyDe-Atom libraries and a couple of other utility libraries. We use the data generators defined in the `Generators` module defined in section 4.2.4.

```
29 import Test.QuickCheck as QC
30 import Test.QuickCheck.Function
31 import Test.Framework
32 import Test.Framework.Providers.QuickCheck2 (testProperty)
33
34 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
35 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
36 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
37 import ForSyDe.Atom.Skeleton.Vector.Matrix as M (transpose)
38 import ForSyDe.Atom.Skeleton.Vector.Cube   as C (Cube, transpose, transpose')
39
40 import AESA.CubesAtom   as AESAC
41 import AESA.StreamsAtom as AESAS
42 import AESA.Params
43
44 import Generators
45 import Data.List as L
46 import Data.Complex
```

5.3.2 Properties

Perhaps the most noticeable transformation has been performed on the DBF module, since all the primitive operations performed by the vector-matrix dot product have become synchronous processes instead. In order to show that the two models are in fact sematically equivalent *with respect to the values and their structured order*, but not necessarily to the causal order or time implications, we formulate a property which degrades the signals and vectors to lists and compares them. Lists do not capture any temporal or spatial semantics, but they do have a clearly-defined *ordering* relation, which is equally valuable in our case. The formulations below follow the notation from section 4.1, and we denote the vector dimensions `Rn` for *range*, `Wn` for *window*, `An` for *antenna*, `Bm` for *beam* and `Ps` for *pulse*.

$$\forall c \in \langle\langle\langle\mathbb{C}\rangle_{\mathtt{Rn}}\rangle_{\mathtt{Wn}}\rangle_{\mathtt{An}} \Rightarrow \Sigma_{\mathtt{An}\to\mathtt{Wn}\to\mathtt{Rn}}(\mathtt{dbf}_{\mathrm{cube}}(\bar{c})) = \Sigma_{\mathtt{An}\to\mathtt{Wn}\to\mathtt{Rn}}(\mathtt{dbf}_{\mathrm{stream}}(\bar{c})) \tag{17}$$

```
67 prop_dbf_transf_equiv = forAll (sigOfSmallCubes arbitrary) $ \sc -> same (cbRes sc) (stRes sc)
68   where
```

[7] actually for most of the properties in this section there are theories which can prove the preservation of semantics during certain transformations (e.g. the Bird-Merteens formalism) without the need to resort to simulation, but they are out of the scope of this report. This report rather focuses on common practices and tools for disciplined design instead.

```
    -- Compare all the values in the resulting nested lists
    same c s = (all . all) id $ (zipWith . zipWith) (==) c s
    -- Results of the cube version of DBF as nested lists. Outputs are re-arranged
    cbRes sc = toLists $ V.farm11 unroll $ SY.unzipx $ SY.comb11 C.transpose' $ AESAC.dbf sc
    -- Results of the stream version of DBF as nested lists. Inputs are re-arranged
    stRes sc = toLists $ AESAS.dbf $ V.farm11 unroll $ SY.unzipx sc
    -- Transforms a vector of signals into a list of lists
    toLists  = map SY.fromSignal . V.fromVector
    -- Flattens a signal of matrices into a signal of ordered samples
    unroll   = SY.signal . concatMap (concatMap fromVector . fromVector) .  SY.fromSignal
```

Let us now test the transformed PC block for the same property. Although the innermost operation is still `fir`, we would like to make sure that changing the MoC domain of the process from SY (on vectors) to SDF (on tokens) did affect in any way the ordering (and values) of the data.

$$\forall c \in \langle\langle\langle\mathbb{C}\rangle_{\mathtt{Rn}}\rangle_{\mathtt{Wn}}\rangle_{\mathtt{Bm}}, r \in c_{\mathtt{Rn}} : |r| = N_b \Rightarrow \Sigma_{\mathtt{Bm}\to\mathtt{Wn}\to\mathtt{Rn}}(\mathtt{pc}_{\mathrm{cube}}(\overline{c})) = \Sigma_{\mathtt{Bm}\to\mathtt{Wn}\to\mathtt{Rn}}(\mathtt{pc}_{\mathrm{stream}}(\overline{c})) \tag{18}$$

```
-- We need an custom generator for cubes with 'nb'-sized range vectors
sigOfRnCubes :: Gen (SY.Signal (C.Cube (Complex Float)))
sigOfRnCubes = do
  rangeL  <- elements [nb]  -- range vector needs to have exactly 'nb' samples
  windowL <- choose (2, 10)
  beamL   <- choose (2, 10)
  sigData <- listOf1 $ sizedCube beamL windowL rangeL arbitrary
  return (SY.signal sigData)

prop_pc_transf_equiv = forAll sigOfRnCubes $ \sc -> same (cbRes sc) (stRes sc)
  where
    -- Compare all the values in the resulting nested lists
    same c s = (all . all) id $ (zipWith . zipWith) (==) c s
    -- Results of the cube version of DBF as nested lists. Outputs are re-arranged
    cbRes sc = syToLst $ V.farm11 unroll $ SY.unzipx
               -- inverse the transposes during DBF and PC, align cubes with streams
               $ SY.comb11 M.transpose $ AESAC.pc $ SY.comb11 C.transpose sc
    -- Results of the stream version of DBF as nested lists. Inputs are re-arranged
    stRes sc = sdfToLst $ AESAS.pc $ V.farm11 unroll $ SY.unzipx sc
    -- Transforms a vector of signals into a list of lists
    syToLst  = map SY.fromSignal  . V.fromVector
    sdfToLst = map SDF.fromSignal . V.fromVector
    -- Flattens a signal of matrices into a signal of ordered samples
    unroll   = SY.signal . concatMap (concatMap fromVector . fromVector) .  SY.fromSignal
```

N.B.: after a few iterations we realized that the property `prop_pc_transf_equiv` would not hold $\forall c \in \langle\langle\langle\mathbb{C}\rangle\rangle\rangle$ because of the consumption rules of `SDF.comb11`. The streaming version of PC would not produce the same result as the cube version had the `Rn` dimension not been an integer multiple of the consumption rate N_b, simply because if there are not enough tokens at the input, a SDF actor does not execute, whereas a scalable vector operation is evaluated regardless of how many elements it operates on. We thus had to adjust the property in eq. 18 accordingly having this insight, which means that we *restrict* the legal inputs pool to fit more to the specifications in section 2.2.4, otherwise we cannot guarantee the property above.

When evaluating the corner turning (CT) it is now easier to verify the 50% overlap because we have access to the streams directly. We thus formulate the property

$$\forall \overline{a} \text{ large enough}, v \in \langle\alpha\rangle_{\mathtt{Wn}} : |v| = N_{FFT} \wedge \mathtt{ct}(\overline{a}) = (\overline{v}_{\mathrm{right}}, \overline{v}_{\mathrm{left}}) \Rightarrow v_{\mathrm{right}}[i] = v_{\mathrm{left}}[i + \frac{N_{FFT}}{2}] \tag{19}$$

```
-- We need a generator for signals of size larger than 'nFFT/2'
largeSigs :: Gen (SDF.Signal Int)
largeSigs = do
  n <- choose (nb * nFFT `div` 2, nb * nFFT)
  sigData <- vectorOf n arbitrary
  return (SDF.signal sigData)
```

```
prop_ct_50_overlap = forAll largeSigs $ \s -> over (AESAS.procCT s)
  where
    over (rc,lc) = all id $ SDF.fromSignal
                   $ SDF.comb21 ((nFFT,nFFT), nFFT `div` 2, overF) rc lc
    overF rl ll  = zipWith (==) rl (L.drop (nFFT `div` 2) ll)
```

For the rest of the blocks, DFB, CFAR and INT we can follow the model set by `prop_pc_transf_equiv` to formulate properties testing for functional equivalence. We only mention these properties as formulas, and we leave the writing of the QuickCheck code as an exercise to the reader.

$$\forall c \in \langle\langle\langle\mathbb{C}\rangle_{\texttt{Wn}}\rangle_{\texttt{Rn}}\rangle_{\texttt{Bm}}, w \in c_{\texttt{Wn}} : |w| = N_{FFT} \Rightarrow \Sigma_{\texttt{Bm}\to\texttt{Rn}\to\texttt{Ps}}(\texttt{dfb}_{\text{cube}}(\overline{c})) = \Sigma_{\texttt{Bm}\to\texttt{Rn}\to\texttt{Ps}}(\texttt{dfb}_{\text{stream}}(\overline{c})) \quad (20)$$

$$\forall c \in \langle\langle\langle\mathbb{C}\rangle_{\texttt{Ps}}\rangle_{\texttt{Rn}}\rangle_{\texttt{Bm}}, w \in c_{\texttt{Wn}}, r \in c_{\texttt{Rn}} : |w| = N_{FFT} \wedge |r| = N_b \Rightarrow \Sigma_{\texttt{Bm}\to\texttt{Rn}\to\texttt{Ps}}(\texttt{cfar}_{\text{cube}}(\overline{c})) = \Sigma_{\texttt{Bm}\to\texttt{Rn}\to\texttt{Ps}}(\texttt{cfar}_{\text{stream}}(\overline{c})) \quad (21)$$

$$\forall c_1, c_2 \in \langle\langle\langle\mathbb{C}\rangle\rangle\rangle \Rightarrow \Sigma(\texttt{int}_{\text{cube}}(\overline{c_1}, \overline{c_2})) = \Sigma(\texttt{int}_{\text{stream}}(\overline{c_1}, \overline{c_2})) \quad (22)$$

5.3.3 Main function. Test Suite Results

We gather the QuickCheck properties defined above into one runnable suite:

```
tests :: [Test]
tests = [
  testGroup " Stream HL Model Tests "
    [ testProperty "DBF transformation preserves the order of elements "
      (withMaxSuccess 50 prop_dbf_transf_equiv)
    , testProperty "PC transformation preserves the order of elements  "
      (withMaxSuccess 20 prop_pc_transf_equiv)
    , testProperty "CT 50% overlap                                     "
      (withMaxSuccess 20 prop_ct_50_overlap)
    ]
  ]

main :: IO()
main = defaultMain tests
```

which we execute, as per the instructions in the project's `README` file. Since we are generating random tests on cubes of noticeable dimensions, the execution time is also quite long, hence the smaller number of test cases generated per suite. The expected printout is:

```
aesa-atom-0.1.0.0: test (suite: tests-stream)

 Stream HL Model Tests :
  DBF transformation preserves the order of elements : [OK, passed 50 tests]
  PC transformation preserves the order of elements  : [OK, passed 20 tests]
  CT 50% overlap                                     : [OK, passed 20 tests]

         Properties  Total
 Passed  3           3
 Failed  0           0
 Total   3           3

aesa-atom-0.1.0.0: Test suite tests-stream passed
```

This means that, at least when concerning the functionality of the AESA signal video processing chain, we have not introduced any unwanted flaws with our model refinement process. We also have a better understanding of the conditions and restrictions within which the new refined model operates as expected.

5.4 Conclusion

In this section we have presented an alternative model of the AESA signal processing chain which refines its behavior in order to expose its streaming characteristics inferred from the specification statement *"For*

each antenna the data arrives pulse by pulse*, and each pulse arrives* range bin by range bin. *This happens* for all antennas in parallel*, and all complex samples are synchronized with the same sampling rate, e.g. of the A/D converter."* While still regarded from a high abstraction level (e.g. SDF actors are assumed to execute "instantaneously" once they have enough data to operate on), the refined processing blocks now describe a more fine-grained causal ordering between individual samples arriving in streams. This enables their potential of better resources exploitation (e.g. hardware/software pipelining, or distribution) during future mapping, scheduling and synthesis design stages. We have (unit) tested this refined model against the high-level "golden model" presented in section 2, by confronting the two models' responses against the same set of input stimuli. We have also formulated a couple of properties to ensure that the functional description of some of the blocks is still the expected one, and we have not introduced hidden bugs, or rather flaws in the model description.

In the following section we plan to synthesize one functional block down to VHDL code targeting FPGA hardware implementation. This means that we will continue gradually refining the (sub-)system model until we reach enough detail level to start the code synthesis process. As each refinement process is prone to introducing hidden mistakes or logical flaws due to its nonsemantic-preserving nature, we will monitor the correctness of the resulting artifact with the help of the existing, as well as newly-formulated, properties.

6 Model Synthesis to VHDL

In this section we choose one sub-system in the AESA signal processing stage and gradually refine to synthesizable VHDL code, by applying a series of semantic- and nonsemantic-preserving transformations. In order to validate the resulting design we gradually wrap each refined component in order to co-simulate it with the original model, test it against the same input data as the previous sections, and formulate design properties that ensure the desired behavior is preserved. As input model we use the refined streaming behavior from section 5 and we focus only on its PC stage. Throughout this section we will introduce another framework in the ForSyDe ecosystem: ForSyDe-Deep, which is able to parse an input ForSyDe program and translate it to different backends.

Package	aesa-deep-0.1.0	path: `./aesa-deep/README.md`
Deps	forsyde-atom-0.2.2	url: `https://forsyde.github.io/forsyde-atom/api/`
	forsyde-atom-extensions-0.1.1	path: `./forsyde-atom-extensions/README.md`
	forsyde-deep-0.2.1	url: `http://hackage.haskell.org/package/forsyde-deep` note: 0.2.1 is found in a local experimental branch.
	QuickCheck-2.13.1	url: `http://hackage.haskell.org/package/QuickCheck`
Suite	tests-deep	usage: `stack test :tests-deep`
Bin	aesa-deep	usage: `aesa-deep --help`

The system in section 5, although exposing a more fine-grained streaming behavior for each data channel, is still far from a physical realization. In order to reach a suitable implementation one has to take into consideration a multitude of aspects from one or several target platforms (architectures), which form the *design space*. Mapping a behavior onto a platform often involves a (series of) transformation(s) that do not fully preserve the original semantics any longer, yet still offer a good realization of the intended behavior. We call these *nonsemantic-preserving* transformations (Raudvere, Sander, and Jantsch 2008).

In this section we will focus on synthesizing the PC signal processing stage, as specified in section 1.1 and modeled in section 5.1, into an FPGA component. We do that by applying four subsequent refinement phases, guided by design *decisions* to reach a correct and/or efficient implementation. Each decision is tested against a set of properties as introduced in section 4, and each refined component is co-simulated within the entire AESA high-level model.

6.1 Refinement 1: Untimed MAV to Timed FIR

The first refinement comes from the observation that the PC block from section 5.1, although operating on streams, exposes an instantaneous, untimed behavior over fixed sized-vectors of numbers. In other words, the SDF process associated with a channel's PC stage is more concerned that a moving average (MAV) algorithm is applied on a vector of N_b samples as soon as they arrive, but it is not really concerned on *how* the MAV is performed. This type of under-specification can derive a family of possible implementations, but does not bide well with the register transfer level (RTL) style of modeling specific to hardware implementations, which requires a more specific total order between actions. For example, translating the "instantaneous" MAV function (i.e. `fir (mkPcCoef pcTap)`) word-by-word into RTL would create a terribly un-efficient and wasteful circuit! Luckily, we have already seen a similar, more fine-grained streaming behavior for the same MAV function in section 2.2.2.6, and respectively section 5.1.2.5, namely the n-tap systolic FIR structure created with the `fir'` skeleton[8].

6.1.1 Model

In this first refinement phase we translate the SDF process `procPC` from section 5.1.2.2 into systolic network of SY processes `procPC'` much more appropriate and efficient for RTL-based implementations.

```
26 {-# LANGUAGE PackageImports #-}
27 module AESA.PC.R1 where
```

[8] in fact both `fir` and `fir'` derive from the same catamorphism. For a formal proof check (Ungureanu et al. 2019).

Customary, we import the needed modules from the `ForSyDe.Atom` suite:

```
31 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
32 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
33 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
34 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector.DSP (fir')
```

To make sure we use exactly the same coefficients and constants as the ones used with the high-level model in section 5.1, we import them directly from the `aesa-atom` package.

```
40 import AESA.Coefs (mkPcCoefs)
41 import AESA.Params (pcTap, nb)
```

Lastly, we need to import the `Complex` type.

```
45 import Data.Complex
```

First we define the coefficients which will be used throughout all the refinement phases. If you read the API documentation of `fir'` you will see that the order of application of FIR tap processes is *from right to left*, thus we need to reverse the order of PC coefficients.

```
52 coefsR1 :: Fractional a => Vector a
53 coefsR1 = V.reverse $ mkPcCoefs pcTap
```

Then we define the interface for the PC' stage, which is the same as for PC in section 5.1.2.2. This means that, as far as the type checker is concerned, we can simply replace `pc` with `pc'` in the original high level model and simulate as part of the entire AESA signal processing chain.

```
60 pc' :: Vector ( SY.Signal (Complex Float))
61     -> Vector (SDF.Signal (Complex Float))
62 pc' = farm11 (SY.toSDF . procPC')
```

the `procPC'` process is, as mentioned earlier, a SY `fir'` process network which defines a particular timed (in the causality sense) behavior for the SDF `procPC` in section 5.1.2.2.

OBS: for didactic purpose, we define `procPC'` as an instance of a custom process constructor `pcFIR` defined below. The role of `pcFIR` is to decouple whatever happens in the *function layer*[9] from the layers above. This will come in handy in the next refinement phase, which affects *only* the function layer, and thus we can reuse the same code patterns from this module.

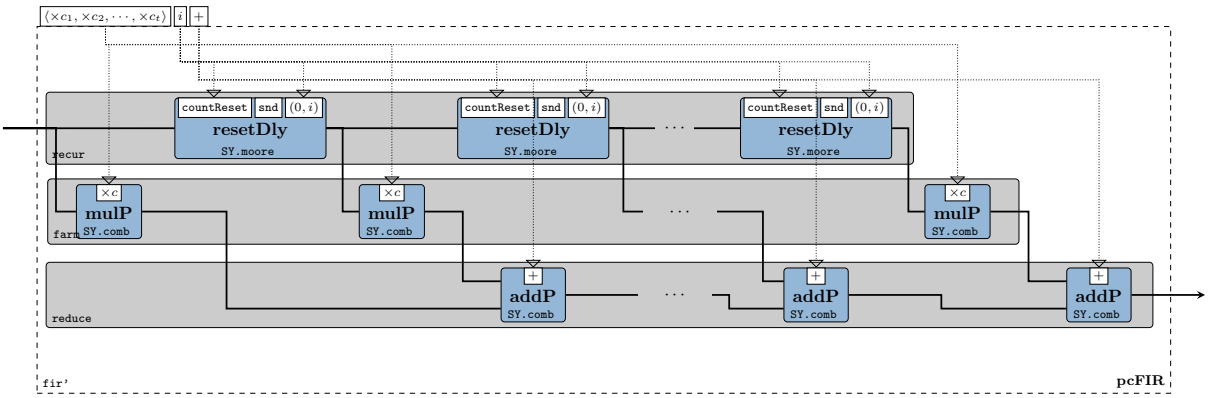


Figure 33: The internals of the `pcFIR` process constructor

```
78 pcFIR :: (a -> a -> a)              -- ^ /function layer/ addition
79       -> (a -> a -> a)              -- ^ /function layer/ multiplication
80       -> a                          -- ^ /function layer/ initial state
81       -> Vector    a                -- ^ vector of coefficients
82       -> SY.Signal a                -- ^ input signal
83       -> SY.Signal a                -- ^ output signal
84 pcFIR sum mul initial coefs = fir' sumP mulP resetDly coefs
```

[9] recall what layers are in section 2.1

```
  where
    sumP     = SY.comb21 sum
    mulP c   = SY.comb11 (mul c)
    resetDly = SY.moore11 countReset snd (0,initial)
    countReset (c,_) p | c == nb-1 = (0,initial)
                       | otherwise = (c+1,p)
```

When we change the time domain of the process we lose the information on the partial order between events. In SY there is no MoC-induced mechanism that tells us *when* N_b events have been consumed/processed: this mechanism needs to be hand-crafted. *One* solution is to embed a count-reset Moore state machine inside each delay element associated with every FIR tap. This way, each Moore machine stores a counter value $\in [0, N_b)$ along with the previous complex sample, and would reset its state after after propagating N_b samples.

Finally, we define `procPC'` by filling in the function layer entities needed by the `pcFIR` process constructor.

```
procPC' :: Fractional a
        => SY.Signal a
        -> SY.Signal a
procPC' = pcFIR (+) (*) 0 coefsR1
```

Function	Original module	Package
`mealy11`, `comb11`, `comb21`, `toSDF`	`ForSyDe.Atom.MoC.SY`	forsyde-atom
`farm11`	`ForSyDe.Atom.Skeleton.Vector`	forsyde-atom
`fir'`	`ForSyDe.Atom.Skeleton.Vector.DSP`	forsyde-atom-extensions
`mkPcCoefs`	`AESA.Coefs`	aesa-atom
`pcTap`, `nb`	`AESA.Params`	aesa-atom

6.1.2 Simulation

As mentioned above, the PC' component can be "plugged in" and used with the AESA signal processing chain. Please refer to the project's `README` file, respectively the binary help menu instructions, on how to execute the model containing the PC' block against the same test input data as the previous high-level models. For the sake of space we do not include the output image, however we encourage the reader to try it out and plot it herself. For now we can only assure you that it looks similar to Figure 32, except for the detection values, whose (slight) differences are induced by floating point calculation errors.

6.1.3 Properties

The main property we want to test now is that the process `procPC'` defined above is *sequence equivalent* with the original `procPC` from section 5.1.2.2 and hence, as of `prop_pc_transf_equiv` defined in eq. 17, the PC' stage is an exact *functional* replacement for the high-level PC operating on video cubes.

We define a new module as part of these refinements' test suite:

```
{-# LANGUAGE PackageImports #-}
module TestR1 where
```

We need to be able to pack/unpack signals, so we import the `SY` and respectively `SDF` modules.

```
import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
```

We import the QuickCheck DSL, along with a couple of local utilities, mainly arbitrary data type generators.

```
import Test.QuickCheck
import Generators (largeSdfSigs, rationals)
```

Finally, we import the DUTs: the SDF `procPC` from the high-level streamed model and the SY `procPC'` from Refinement 1.

```
27 import AESA.StreamsAtom as M1
28 import AESA.PC.R1 as R1
```

The property we need to test is

$$\forall c \in \mathbb{C} \Rightarrow \Sigma(\texttt{procPC}(\overline{c})) = \Sigma(\texttt{procPC}'(\overline{c})) \tag{23}$$

In order to test sequence equivalence between the two processes we need to take into account two very important matters:

1. a SDF process' output signal contains a number of elements which is a multiple of its production rate, whereas a SY process does not have this restriction. Sequence equivalence means that both output signals have the same initial segments, property which is handled by the `zip` list function.
2. floating point numbers *cannot* be used to test for equality, especially now since the order of operations is different. Luckily we have defined both `procPC` and `procPC'`, as well as their coefficient vectors as polymorphic, thus as a practical alternative `Complex Float` (i.e. $\mathbb{C}$), we can use `Rational` numbers (i.e. $\mathbb{Q}$) which have a strict notion of equality. Sequence equivalence over $\mathbb{Q}$ is sufficient to later demonstrate through inference eq. 23.

```
54 prop_refine1_equiv = forAll (largeSdfSigs rationals) $ \s -> equiv s (SDF.toSY1 s)
55   where
56     equiv sdf sy = all (\(a,b) -> a == b) $ zip
57                    (SDF.fromSignal $ M1.procPC sdf)
58                    ( SY.fromSignal $ R1.procPC' sy)
```

When running the test suite (refer to the `README` instructions) `prop_refine1_equiv` passes all tests, which means that the two behaviors are compatible and PC' can be used as source for further refinements.

6.2 Refinement 2: Floating Point to Q19

Due to it widespread usage with general purpose computing, IEEE floating point number representation is the de-facto standard for non-integer numerical applications. However, although common and even taken for granted in CPUs, floating point units are in general much larger and more expensive circuits than their integer counterparts. Moreover, judging by the application specification in section 1.1, synthesizing floating point arithmetics on FPGA would be overkill, because:

- we do not need a large span of numbers. On the contrary, the specification says that at the PC stage the samples are within $[-1-i, 1+i)$, i.e. only decimals;
- according to tbl. 1, for the calculations within PC we do not need a precision higher than 20 bits.

Driven by these statements, we decide that a much more appropriate number representation for synthesizing the PC signal processing stage to FPGA is the fixed point Q19 representation which, under the hood, consists simply in integer calculations. We thus refine further the PC' stage as to operate on complex fixed point Q19 numbers instead of complex floating point numbers, or more specific *refine only its function layer* to operate on this type of numbers.

6.2.1 Model

We code the second refinement in its own module.

```
28 {-# LANGUAGE PackageImports #-} --can be ignored
29 module AESA.PC.R2 where
```

We import the ForSyDe-Atom libraries that are used:

```
33 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
34 import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
35 import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
```

We import the `Complex` type. However, unlike previously, we now use our in-house `ForSyDe.Deep.Complex` module. This module is in fact re-exporting `Data.Complex` along with some additional instances which makes it synthesizable, plus a couple of custom arithmetic operators `(+:)`, `(-:)`, `(*:)` which do not force the base type of `Complex a` to be a part of `RealFloat` class (see Data.Complex API), but rather any `Num` type.

```
45 import ForSyDe.Deep.Complex
```

We import our in-house fixed point data types. This module exports the usual types such as `Fixed8`, `Fixed16`, `Fixed32` and `Fixed64`. For this report we have extended the `forsyde-deep` package with a custom (synthesizable) `Fixed20` type, hard-coded across the full compiler stack. Future versions of ForyDe-Deep will most likely support arbitrary-precision fixed point types using type-level numerals instead, similar to `FSVec`, but for simple demonstration purposes `Fixed20` suffices at the moment.

```
54 import ForSyDe.Deep.Fixed
```

Finally, we import the functions defined in the previous refinement stage.

```
58 import AESA.PC.R1 (coefsR1, pcFIR)
```

We translate the floating point FIR coefficients from the first refinement phase to Q19 complex numbers.

```
63 coefsR2 = V.farm11 ((:+0) . realToFixed20) coefsR1 :: Vector (Complex Fixed20)
```

For `procPC''` we use the previously defined `pcFIR` process constructor whose function layer entities are now replaced with operations over `Complex Fixed20` numbers. This eases the validation process: we now need to test only that the function layer refinements respect the specifications, and not the whole system.

```
70 procPC'' :: SY.Signal (Complex Fixed20)
71          -> SY.Signal (Complex Fixed20)
72 procPC'' = pcFIR (+:) (*:) (0:+0) coefsR2
```

The PC" stage process (network) is the same one as in the previous refinement, except that it uses the `procPC''` as the base process.

```
77 pc'' :: Vector ( SY.Signal (Complex Fixed20))
78      -> Vector (SDF.Signal (Complex Fixed20))
79 pc'' = farm11 (SY.toSDF . procPC'')
```

6.2.2 Simulation

In order to be able to "plug in" PC" into the AESA signal processing system, we need to wrap it so that its type signature is the one expected. We thus define the following wrapper which translates the floating point numbers fed into PC to fixed point Q19 and back:

```
88 wrapR2 f = farm11 (SDF.comb11 (1,1,(fmap . fmap) fixed20ToReal))
89            . f . farm11 (SY.comb11 (fmap realToFixed20))
```

which is then used to wrap `pc''` so that it looks like `pc`.

```
93 wrappedPC'' :: Vector ( SY.Signal (Complex Float))
94              -> Vector (SDF.Signal (Complex Float))
95 wrappedPC'' = wrapR2 pc''
```

The effect of the refined PC" signal processing stage can be observed by simulating the AESA application instantiating `wrappedPC''`. Please refer to the project's `README` file on how to execute the program. When plotting the results against the same input data, we can see that the same 13 objects are detected, albeit having different numerical values.

6.2.3 Properties

In the second refinement phase the properties target mainly the translations between the floating point and fixed point representations. We define a new module with the usual preamble:

```haskell
{-# LANGUAGE PackageImports #-}
module TestR2 where

import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
import "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector (farm11)
import Test.QuickCheck
import ForSyDe.Deep.Complex
import ForSyDe.Deep.Fixed

import Generators (sySignals, decimalCpxNum, decimalCpxRat)
import AESA.PC.R1 as R1
import AESA.PC.R2 as R2
import AESA.Params (pcTap)
```

The first property we check is that even when wrapped (i.e. going through two translations), the `procPC''` does not overflow its legal input/output value range, i.e. $[-1-i, 1+i)$. For this we write a local process wrapper `wrapProc`.

$$\forall v \in \langle\mathbb{C}\rangle, a \in v, b \in \texttt{procPC}''(v) : |v| > 0 \wedge a \in [-1-i, 1+i) \Rightarrow b \in [-1-i, 1+i) \tag{24}$$

```haskell
prop_refine2_values = forAll (sySignals decimalCpxNum)
                      $ \s -> all (withinRangeComplex (-1) 1)
                              $ SY.fromSignal $ wrapProc R2.procPC'' $ s
  where
    wrapProc p = SY.comb11 (fmap fixed20ToReal) . p . SY.comb11 (fmap realToFixed20)
    withinRangeComplex a b c
      | realPart c <  a = False
      | imagPart c <  a = False
      | realPart c >= b = False
      | imagPart c >= b = False
      | otherwise = True
```

The second property we want to test is that the cumulative quantization error stays within a constant, accepted range. Again, we cannot use floating point numbers as samples for comparison, due to the dynamic quantization error of their own representation. As before, we fall back to rationals which constitute a good model. The cumulative error for one PC" channel, supposing that both multiplications and addition results are truncated back to Q19, can be calculated with:

$$\epsilon = \epsilon_{Q19} * N_{tap} = 2^{-19} \times N_{tap}$$

```haskell
prop_refine2_error = forAll (sySignals decimalCpxRat)
                     $ \s -> all acceptable $ zip (rationalPcVals s) (fixed20PcVals s)
  where
    rationalPcVals s = SY.fromSignal $ procPCRationals s
    fixed20PcVals  s = SY.fromSignal $ procPC'' $ SY.comb11 (fmap realToFixed20) s
    procPCRationals  = pcFIR (+:) (*:) (0:+0) (farm11 ((:+0) . realToFrac) coefsR1)
    ------------------------------------
    acceptable (rat, f20) = let rp = abs (realPart rat - (fixed20ToReal $ realPart f20))
                                ip = abs (imagPart rat - (fixed20ToReal $ imagPart f20))
                            in rp <= epsilon && ip <= epsilon
    epsilon = (realToFrac pcTap) * 2^^(-19)
```

Notice that we have instantiated a clone `procPC''` which works on rational numbers called `procPCRationals` using the `pcFIR` process constructor. We needed this process to compare the rational outputs against the fixed point ones.

6.3 Refinement 3: Deep Language Embedding

In this refinement phase we translate the PC" system defined in the previous phase to a language more appropriate for synthesis. Up until now we have used the shallow-embedded DSLs of ForSyDe, namely

ForSyDe-Atom and ForSyDe-Shallow. In this section we will start using ForSyDe-Deep, a deep-embedded DSL, capable of extracting a system's internal structure as a netlist and synthesizing it to different backends (at present GraphML and VHDL).

6.3.1 Crash course in ForSyDe-Deep

ForSyDe-Deep was originally created as a replacement for ForSyDe-Shallow. It provides a sub-set of the ForSyDe-Shallow language and, in principle, is based on the same modeling concepts. However its deep-embedded features (such as netlist traversing and language parsing) makes it much more verbose than its shallow counterparts. This is why its syntax appeals less to new users, hence it became it a fall-back framework for synthesis purposes only. Nevertheless translating from a shallow ForSyDe model to a deep one is straightforward, once you understand the principles listed as follows. For more on ForSyDe-Deep modeling, refer to the beginner tutorials on the ForSyDe homepage.

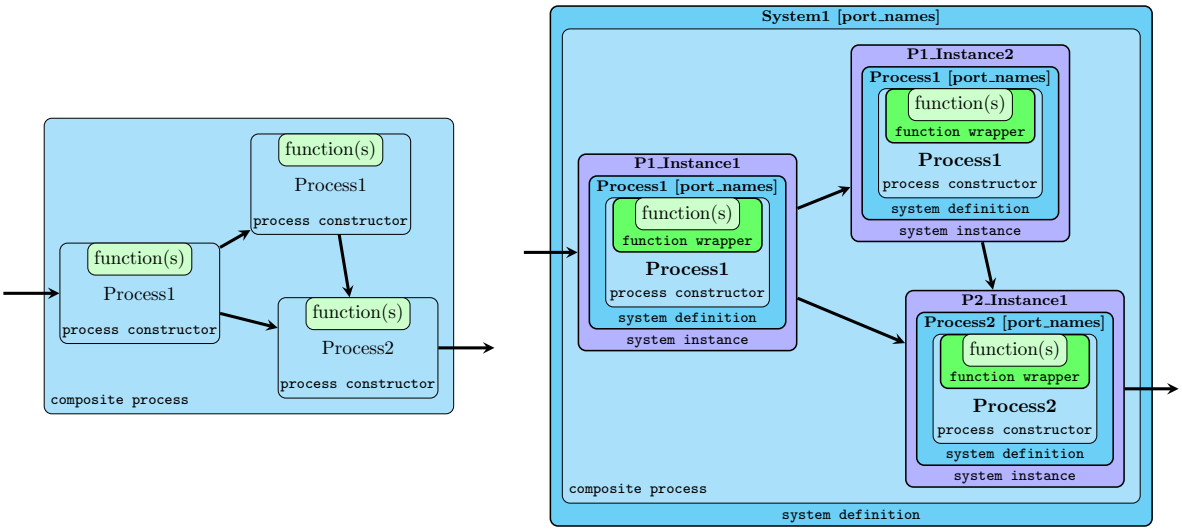


Figure 34: The difference between a shallow ForSyDe system (left) and a deep one (right)

- process functions are parsed to their AST using a language extension of Haskell called Template-Haskell. The function parser is able to recognize a subset of the Haskell language written between the so-called banana brackets `[d| function |]`, and needs to *explicitly specify its type signature*. Whatever code is written within the banana brackets is parsed as-is and needs to be self-contained, i.e. information cannot be inferred from global/system-wide definitions. In order to instantiate a parsable ForSyDe-Deep function from a Haskell function, it needs to be wrapped into a `ProcFun` type, using one of the function wrappers provided by the API.
- processes are instantiated using the ForSyDe-Deep equivalents of the main ForSyDe-Shallow SY process constructors. The main difference is that they need a string identifier as well as `ProcFun`-wrapped functions instead of regular Haskell functions.
- in order to be able to simulate, parse or synthesize a process, it needs to be wrapped within a `SysDef` type, similarly to parsable functions, using one of the API-provided system definition wrappers. These system wrappers, apart from a fully applied process, need also a unique string identifier for the new system, as well as name identifiers for all its input and output ports.
- in order to use a defined system hierarchically as a component, it needs to be instantiated back to a composable function. A process instance needs a unique name identifier, as well as a `SysDef` object.

N.B.: That is quite a mouthful of wrappers and, as you will see in the PC example below, it implies a quite some unpleasant verbosity, especially for one used to the elegant compactness of Haskell programs. Future iterations of the ForSyDe language will most likely have a unified syntax and translation from shallow (for efficient simulation) to deep (for structure parsing and synthesis) versions will be done automatically. For now the current (type-extended) version of ForSyDe-Deep suffices for demonstration purposes since the modeling principles are the same no matter the frontend language.

6.3.2 Model

We create a new module for this refinement phase. The `TemplateHaskell` and `FlexibleContexts` language extensions are mandatory.

```
{-# LANGUAGE PackageImports, TemplateHaskell, FlexibleContexts #-}
module AESA.PC.R3 where
```

We import some ForSyDe-Atom types, only for co-simulation wrapping.

```
import qualified "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY  as SY
import qualified "forsyde-atom-extensions" ForSyDe.Atom.MoC.SDF as SDF
import qualified "forsyde-atom-extensions" ForSyDe.Atom.Skeleton.Vector as V
```

We import the ForSyDe-Deep libraries. The `ForSyDe.Deep.Skeletons` library belongs to the package extension built for this case study and contains a couple of deep-embedded (synthesizable) skeletons, as presented in sections 2, 5.

```
import ForSyDe.Deep
import ForSyDe.Deep.Skeleton as Sk
```

We need some additional type libraries. `Data.Param.FSVec` defines fixed-size vectors as compared to the shallow vectors used until now, e.g. `ForSyDe.Atom.Skeleton.Vector`. Fixed-size vectors have their length captured in the type signature as type-level numerals imported from `Data.TypeLevel.Num`. For example `(FSVec D8 a)` denotes a fixed-size vector of type a, with the size of 8 elements. We also use list functions, thus we can tell their functions apart by loading the two libraries using different aliases.

```
import Data.List as L
import Data.Param.FSVec as FSV
import Data.TypeLevel.Num hiding ((+), (-), (==))
```

Finally, we import some functions from the previous refinement phase, as well as some system constants.

```
import AESA.PC.R2 (coefsR2, wrapR2)
import AESA.Params (nb)
```

For starters, let us define the adder process used in the `fir'` skeleton. According to the crash course in section 6.3.1 we need to gradually wrap its elements until it becomes a system definition `SysDef`. First, we need to declare a `ProcFun` element from a Haskell addition function. As already mentioned, whatever is written between the banana brackets needs to be self-contained. This means that the `Num` type instance of `Complex` which overloads the `(+)` operator is not of much use here. Complex number addition needs to be explicitly written so that the parser know what to synthesize (N.B. type class support is listed among the "todo" items of future ForSyDe language versions).

```
addFun :: ProcFun (Complex Fixed20 -> Complex Fixed20 -> Complex Fixed20)
addFun = $(newProcFun
            [d|addf :: Complex Fixed20 -> Complex Fixed20 -> Complex Fixed20
               addf (x :+ y) (x' :+ y') = (x + x') :+ (y + y')  |])
```

The addition process is created by passing an identifier and the newly defined `ProcFun` to the process constructor `zipWithSY` (i.e. the equivalent of `SY.comb21` in ForSyDe-Atom).

```
addProc :: Signal (Complex Fixed20)
        -> Signal (Complex Fixed20)
        -> Signal (Complex Fixed20)
addProc = zipWithSY "addProc" addFun
```

From this process we create the component identified as `"add"`, with two input ports `"i1"` and `"i2"` and an output port `"o1"`.

```
addSys :: SysDef (  Signal (Complex Fixed20)
                 -> Signal (Complex Fixed20)
                 -> Signal (Complex Fixed20) )
addSys = newSysDef addProc "add" ["i1","i2"] ["o1"]
```

Similarly, we define the `"mul"` component, but this time in one go. Notice again the expanded complex number multiplication. Instead of the normal multiplication operator `(*)` we use the in-house `fixmul20` function, which multiplies two Q19 numbers and truncates the result back to Q19. In the shallow simulation `fixmul20` ≡ `(*)`, but in the deep version `fixmul20` synthesizes to a custom VHDL `std_vector` function instead of the regular VHDL `*` operator.

```
mulSys :: SysDef (  Signal (Complex Fixed20)
                 -> Signal (Complex Fixed20)
                 -> Signal (Complex Fixed20) )
mulSys = newSysDef (zipWithSY "mulProc" mulFun) "mul" ["i1","i2"] ["o1"]
  where
    mulFun = $(newProcFun
                [d|mulf :: Complex Fixed20 -> Complex Fixed20 -> Complex Fixed20
                   mulf (x :+ y) (x' :+ y') = (fixmul20 x x' - fixmul20 y y') :+
                                              (fixmul20 x y' + fixmul20 y x') |])
```

The reset-delay FSM used as a delay element in the `fir'` pattern needs also to be defined as a component. We define the `"rDelay"` component using the `mooreSY` process constructors, taking two `ProcFun`s for the next state function `countReset` and output decoder `propagate`. We fix the counter value type to be `Int16`. The globally-defined `nb` (i.e. the number of range bins N_b) cannot be recognized as-is inside the banana brackets, and for now all we can do is to write the number explicitly. (N.B. support for partial function application is listed among the "todo" items of future ForSyDe language versions).

```
rDelaySys :: SysDef (Signal (Complex Fixed20) -> Signal (Complex Fixed20))
rDelaySys = newSysDef (mooreSY "rDelayProc" countReset propagate(0, 0 :+ 0))
            "rDelay" ["i1"] ["o1"]
  where
    countReset = $(newProcFun
                    [d|cntf :: (Int16,Complex Fixed20) -> Complex Fixed20
                            -> (Int16,Complex Fixed20)
                       cntf(c,_) p = if c == 1024-1
                                     then (0, 0 :+ 0)
                                     else (c+1,p) |])
    propagate  = $(newProcFun
                    [d|prpf :: (Int16,Complex Fixed20) -> Complex Fixed20
                       prpf (_,p) = p |])
```

Because partial application is not yet supported we need to instantiate the vector of coefficients into a farm of `constSY` signal generators. We thus create the "system constructor":

```
constSys :: ProcId -> Complex Fixed20 -> SysDef (Signal (Complex Fixed20))
constSys name c = newSysDef (constSY "const" c) name ["i1"] ["o1"]
```

which we then use as argument for the `app11` skeleton (which is in fact a `farm11` tailored for partial application only) to instantiate a farm of constant signal generators, i.e. a vector of constant signals. The coefficients are interpreted into a `FSVec` from their shallow counterparts defined in the second refinement phase.

```
coefsR3 = Sk.app11 "coef" constSys coefs
  where
    coefs = $(vectorTH (V.fromVector coefsR2 :: [Complex Fixed20]))
```

OBS: ForSyDe-Deep skeletons, as compared to their shallow counterparts, take care of *creating instances* (see Figure 34) of defined systems as well as coupling them together. This is why they need components as arguments instead of simple functions.

Now, for didactic purpose, let us define ForSyDe-Deep equivalent of the `fir'` skeleton using the base catamorphisms *recur* (or its specialization *generate*), *farm* and *reduce*. We call this process network constructor `deepFIR` and, as you can see, it is defined similarly to its shallow counterpart, with a few exceptions: 1) it expects a base identifier, to create unique IDs for all the generated component instances; 2) it takes `SysDef` components instead of processes as arguments; 3) for coefficients it requires a fixed-size vector of strictly positive size, containing constants.

```
deepFIR :: (SysFun (a -> a -> a), SysFun (c -> a -> a), SysFun (a -> a),
            Pos s', Succ s' s)
        => ProcId                -- ^ system ID
        -> SysDef (a -> a -> a)  -- ^ process/operation replacing '+'
        -> SysDef (c -> a -> a)  -- ^ process/operation replacing '*'
        -> SysDef (a -> a)       -- ^ delay process
        -> FSVec s c             -- ^ vector of coefficients
        -> a                     -- ^ input signal/structure
        -> a                     -- ^ output signal/structure
deepFIR name addSys mulSys dlySys coefs =
  Sk.reduce addName addSys . Sk.farm21 mulName mulSys coefs . Sk.generate dlyName n dlySys
  where n = lengthT coefs
        dlyName = name L.++ "_dly_"
        addName = name L.++ "_add_"
        mulName = name L.++ "_mul_"
```

We finally can define a component for the deep equivalent of `procPC` using the `deepFIR` process network constructor, by passing the above-defined components as arguments.

```
procPCSys :: SysDef (  Signal (Complex Fixed20)
                    -> Signal (Complex Fixed20) )
procPCSys = newSysDef (deepFIR "fir" addSys mulSys rDelaySys coefsR3)
            "FIR" ["i1"] ["o1"]
```

and the deep equivalent of the PC process, statically defining its size as a type-level numeral equal to N_b

```
pc3 :: FSVec D8 (Signal (Complex Fixed20))
    -> FSVec D8 (Signal (Complex Fixed20))
pc3 = Sk.farm11 "pc" procPCSys
```

with its associate system definition. Unfortunately, at the moment processes of type `FSVec a (Signal a) -> ...` are not part of the `SysFun` class, thus we cannot create directly a `SysDef` components from `pc3`. What wee *can* do however is to wrap `pc3` inside a `zipx . pc . unzipx` pattern which merely transposes the vector and signal domains, and the synthesizer will simply ignore it. (N.B. vector-based components will be supported in future iterations of ForSyDe).

```
sysPC3 = newSysDef (zipxSY "zip" . pc3 . unzipxSY "unzip") "PC3" ["i1"] ["o1"]
```

6.3.3 Simulation. Synthesis

Similarly to the previous refinement stages, we further wrap the PC component in order to co-simulate it with the rest of the AESA system. To simulate a ForSyDe-Deep `SysDef` component we use the `simulate` function

```
wrappedPC3 :: V.Vector ( SY.Signal (Complex Float))
           -> V.Vector (SDF.Signal (Complex Float))
wrappedPC3 = wrapR2 (wrapR3 (simulate sysPC3))
```

where `wrapR3` translates between the shallow ForSyDe-Atom types and the component simulator types.

```
wrapR3 sim = V.vector . L.map SDF.signal . L.transpose . L.map FSV.fromVector . sim
             . L.map (FSV.unsafeVector d8) . L.transpose . L.map SY.fromSignal . V.fromVector
```

Please refer to the project's `README` file for how to execute the AESA system alongside the deep `pc3` component. The plotted output for running the system against the radar indata generated by the 13 objects is shown in Figure 35.

The $PC^{(3)}$ component is not only able to be simulated, but we can also parse its internal structure. Here we call the command which dumps the internal structure of `sysPC3` as hierarchical GraphML files, as seen in the plots in Figure 36.

```
graphmlPC3 = writeGraphML sysPC3
```

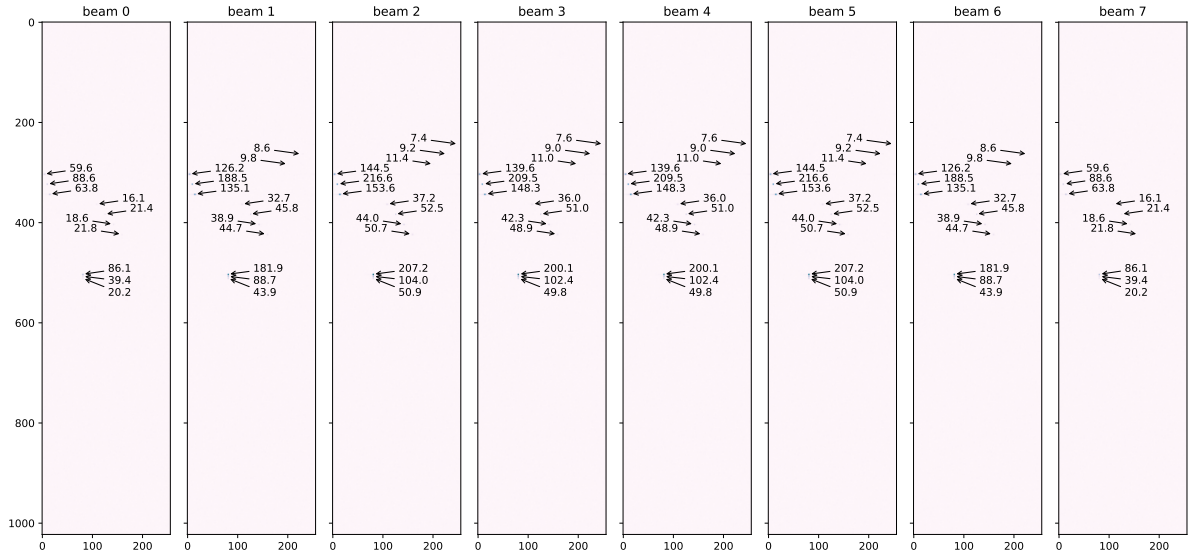


Figure 35: One output cube with radar data

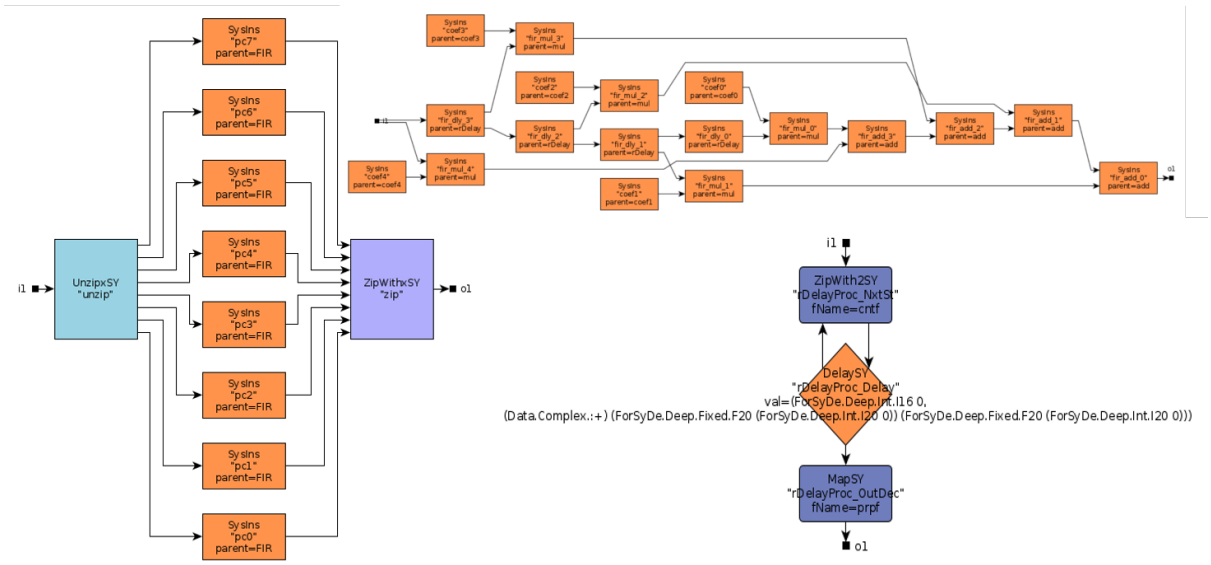


Figure 36: Dumped GraphML structure: PC (left); FIR (above right); rDelay (below right).

Of course, the whole point of writing ForSyDe-Deep is to synthesize to backend code, in this case to synthesizable VHDL code. The following command generates a VHDL project folder with the PC architecture files.

```
280 vhdlPC3 = writeVHDLOps (defaultVHDLOps {debugVHDL = VHDLVerbose}) sysPC3
```

Furthermore, one can simulate the VHDL files using ModelSim (or alternatively Ghdl), provided their binaries are found in the system `PATH`. The following function can be wrapped as `wrappedPC3` instead of `simulate sysPC3`.

```
286 vhdlSim = writeAndModelsimVHDL Nothing sysPC3
```

Finally, we synthesize `sysPC3` to FPGA using the Quartus II tool suite (provided its binaries are found in `PATH`), e.g.:

```
291 quartusPC3 = writeVHDLOps vhdlOps sysPC3
292   where vhdlOps     = defaultVHDLOps{execQuartus=Just quartusOps}
293         quartusOps = checkSynthesisQuartus
```

After synthesis it generated the RTL structures in Figure 37, with the specifications in tbl. 25.

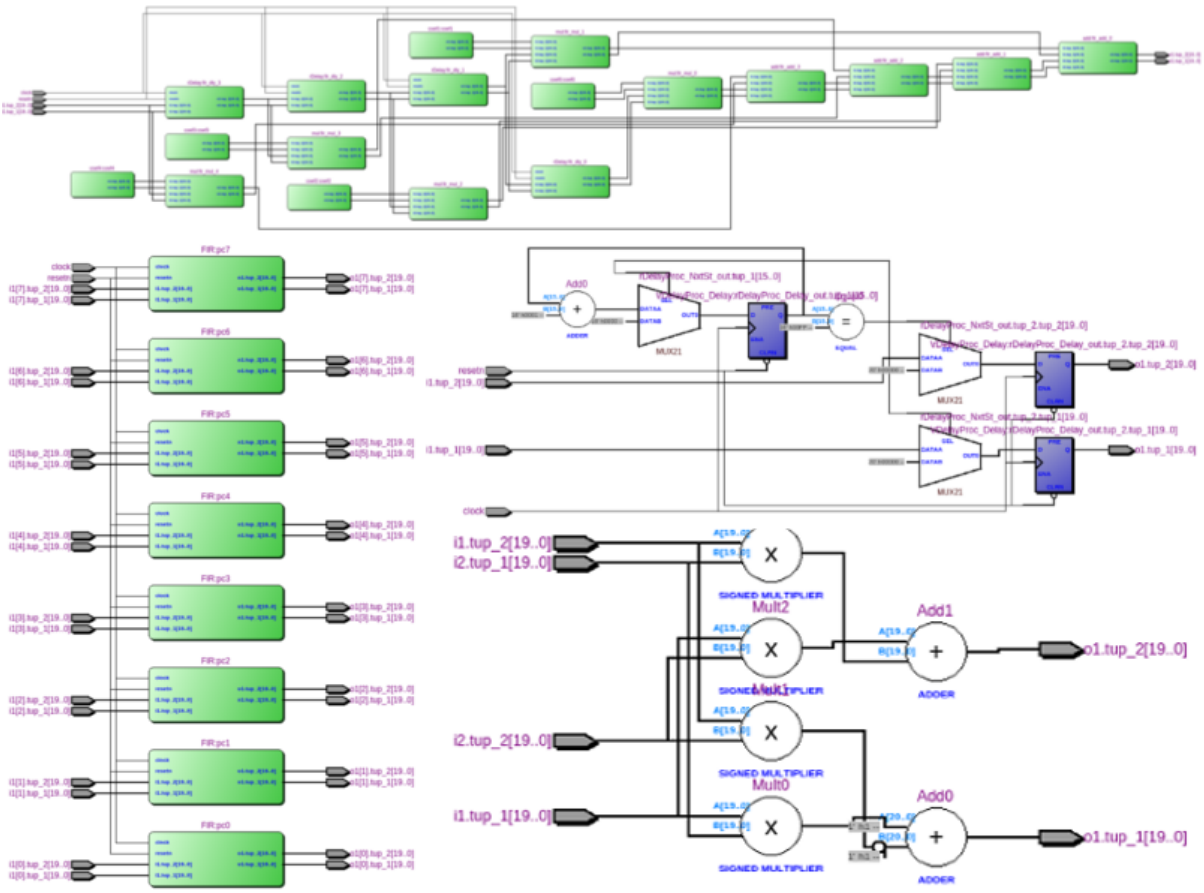


Figure 37: Screenshots of synthesized RTL components: FIR (above); PC (below right); rDelay and mul (below left).

Table 25: Specifications of generated FPGA design

Top-level Entity Name	PC3
Family	Cyclone IV GX
Total logic elements	3,014
Total combinational functions	3,014
Dedicated logic registers	976
Total registers	976

Table 25: Specifications of generated FPGA design

Total memory bits	0
Embedded Multiplier 9-bit elements	160
Total PLLs	0

6.3.4 Properties

The main property we want to check now is that the ForSyDe-Deep version of $\text{PC}^{(3)}$ is the same as PC" defined in the second refinement phase.

```
{-# LANGUAGE PackageImports #-}
module TestR3 where

import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
import Test.QuickCheck
import ForSyDe.Deep

import Generators (largeSySigs, cpxFixed20)
import AESA.PC.R2 as R2
import AESA.PC.R3 as R3
```

Since fixed point numbers can be compared exactly, we simply use the processes as they are, operating on signals of `Fixed20` numbers to test the property

$$\forall c \in \mathbb{C} \Rightarrow \Sigma(\texttt{procPC}''(\bar{c})) = \Sigma(\texttt{simulate}(\texttt{procPCSys})(\bar{c})) \tag{25}$$

```
prop_refine3_equiv = forAll (largeSySigs cpxFixed20)
                       $ \s -> all (\(a,b) -> a == b) $ zip
                               (SY.fromSignal $ R2.procPC'' s)
                               (simulate R3.procPCSys $ SY.fromSignal s)
```

Since the test passes flawlessly we can conclude that in the third refinement phase we have achieved semantic equivalence between the components written in the two different languages.

6.4 R4: Balancing the FIR Reduction

The final refinement phase consists in performing some simple semantic-preserving transformations on the previous model in order to optimize the efficiency of the generated circuit.

Two (rather evident) optimizations are considered in this phase:

- as seen in Figures 36, 37, as well as in Figure 33, the final stage of the `pcFIR`-instantiated process network consists of a reduction pattern. This pattern is instantiated recursively, by taking the result of the previous addition and applying it to the next one, thus generating a linear ($O(n)$) reduction tree. However, the base operation performed, i.e. addition, is commutative, and a well-known catamorphism theorem (see Ungureanu et al. (2019) for more details) states that the reduction can in fact be performed as a balanced logarithmic ($O(\log n)$) tree which means, in the case of digital hardware designs, a shorter combinational critical path.
- having a counter in each delay element, although elegant and modular, is a terrible waste of silicon when considering that all counters are counting the same thing: how many samples have passed. A more reasonable design would take advantage of the identical part in all the `"rDelay"` components (i.e. the counter), and separate from the non-identical part (i.e. the delay register).

The second optimization is identified and performed automatically by the Quartus compiler, hence the specifications in tbl. 25 includes only one counter circuit for all the delay elements. Therefore we will only focus on reduction tree balancing and leave the functional decoupling of the reset-counter as an exercise.

6.4.1 Model

The module for the forth refinement module is the following.

```
{-# LANGUAGE PackageImports, TemplateHaskell, FlexibleContexts #-}
module AESA.PC.R4 where
```

We import the same library as before:

```
import ForSyDe.Deep
import ForSyDe.Deep.Skeleton as Sk
import Data.List as L
import Data.Param.FSVec as FSV
import Data.TypeLevel.Num hiding ((+), (-), (==))

import AESA.PC.R2 (wrapR2)
import AESA.PC.R3 (wrapR3, coefsR3, addSys, mulSys, rDelaySys)
```

To instantiate the logarithmic reduction tree, it is enough to replace the `reduce` skeleton with the `logReduce` skeleton in the previous `deepFIR` process network constructor.

```
balancedFIR name addSys mulSys dlySys coefs =
  Sk.logReduce  (name L.++ "_add_") addSys
  . Sk.farm21   (name L.++ "_mul_") mulSys coefs
  . Sk.generate (name L.++ "_dly_") n dlySys
  where n = lengthT coefs
```

The new `procPC` system definition is created using the skeleton above.

```
procPCSys' :: SysDef (  Signal (Complex Fixed20)
                     -> Signal (Complex Fixed20) )
procPCSys' = newSysDef (balancedFIR "fir" addSys mulSys rDelaySys coefsR3)
             "FIR" ["i1"] ["o1"]
```

We finally create the refined version of the AESA $PC^{(4)}$ signal processing stage

```
pc4 :: FSVec D8 (Signal (Complex Fixed20))
    -> FSVec D8 (Signal (Complex Fixed20))
pc4 = Sk.farm11 "pc" procPCSys'
```

which is then declared as a component.

```
sysPC4 = newSysDef (zipxSY "zip" . pc4 . unzipxSY "unzip") "PC4" ["i1"] ["o1"]
```

6.4.2 Simulation. Synthesis

Similar to previous refinement phases, we wrap the $PC^{(4)}$ in order to co-simulate it with the high-level AESA model. As expected, the AESA simulation using this component gives exactly the same results as the previous simulation, shown in Figure 35.

```
wrappedPC4 = wrapR2 (wrapR3 (simulate sysPC4))
```

Dumping the internal structure of $PC^{(4)}$ shows in Figure 38 a more compact FIR structure, with a shorter combinational depth, as well as one `"rCount"` component fanning out into all the delay elements.

```
graphmlPC4 = writeGraphMLOps (defaultGraphMLOps {yFilesMarkup = True})  sysPC4
```

The VHDL code can be generated

```
vhdlPC4 = writeVHDL sysPC4
```

simulated

```
vhdlSim = writeAndModelsimVHDL Nothing sysPC4
-- vhdlSim' = writeAndGhdlVHDL Nothing sysPC4 -- alternatively
```

or synthesized.

```
quartusPC4 = writeVHDLOps vhdlOps sysPC4
  where vhdlOps     = defaultVHDLOps{execQuartus=Just quartusOps}
        quartusOps = checkSynthesisQuartus
```

The generated circuit has exactly the same size as the previous one (tbl. 26), however, the combinational depth is visibly smaller, as seen in the RTL plot in Figure 39.

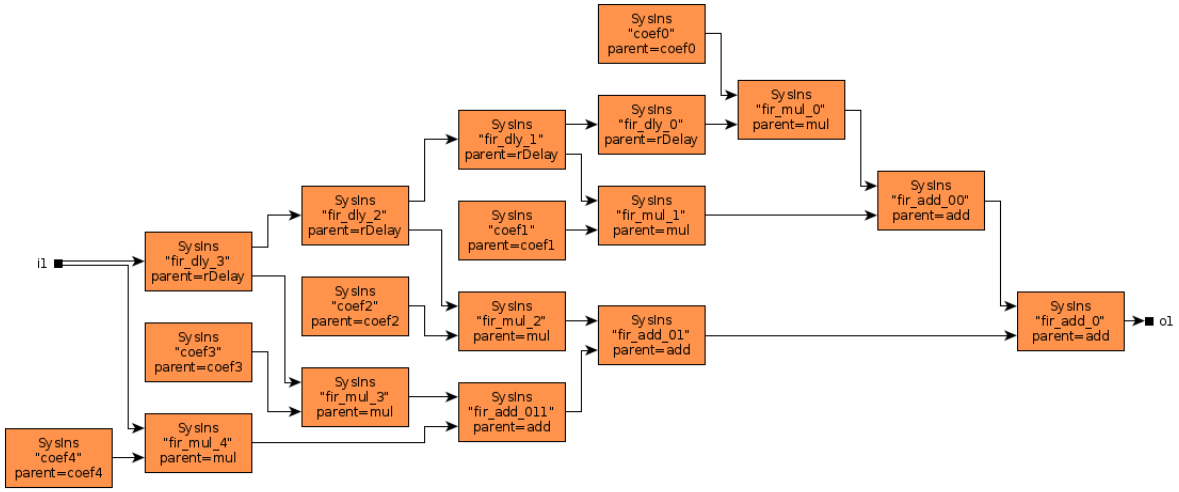


Figure 38: Dumped GraphML structure of the new FIR component

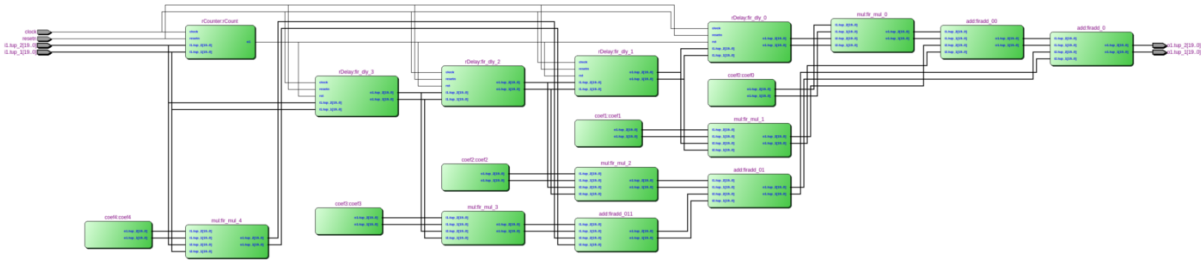


Figure 39: Screenshot of the RTL view of FIR

Table 26: Specifications of generated FPGA design

Top-level Entity Name	PC4
Family	Cyclone IV GX
Total logic elements	3,014
Total combinational functions	3,014
Dedicated logic registers	976
Total registers	976
Total memory bits	0
Embedded Multiplier 9-bit elements	160
Total PLLs	0

6.4.3 Properties

Here we test that the ForSyDe-Deep version of $PC^{(4)}$ is the same as $PC^{(3)}$ defined in the third phase.

```
{-# LANGUAGE PackageImports #-}
module TestR4 where

import "forsyde-atom-extensions" ForSyDe.Atom.MoC.SY as SY
```

```
10 import Test.QuickCheck
11 import ForSyDe.Deep

13 import Generators (largeSySigs, cpxFixed20)
14 import AESA.PC.R3 as R3
15 import AESA.PC.R4 as R4
```

The property

$$\forall c \in \mathbb{C} \Rightarrow \Sigma(\texttt{procPCSys}(\overline{c})) = \Sigma(\texttt{procPCSys}'(\overline{c})) \tag{26}$$

is tested by the QuickCheck program

```
26 prop_refine4_equiv = forAll (largeSySigs cpxFixed20)
27                       $ \s -> all (\(a,b) -> a == b) $ zip
28                               (simulate R3.procPCSys  $ SY.fromSignal s)
29                               (simulate R4.procPCSys' $ SY.fromSignal s)
```

As all tests are passing (check the project's `README` file on how to run tests), hence we can conclude that $\text{PC}^{(4)}$ is a good replacement for the AESA PC stage.

6.5 Conclusions

In this section we have shown a series of design transformations on one of the high-level model components of AESA down to synthesizable VHDL code. Along the way we have co-simulated the transformed components alongside the original AESA high-level model, as well as gradually validated the final generated design artifact is a proper implementation for its behavioural model.

The scope of this section was two-fold: on the one hand we have shown a practical and didactic example and consolidated the design methodology introduced in the previous chapters; and on the other hand we have provided a brief glance over the current status of some of the tools and their position in the ForSyDe ecosystem, as well as directions for future development. As mentioned several times throughout this document the ForSyDe ecosystem is actively being developed and the overall vision is to learn from past and current experiences in order to achieve a correct-by-construction design flow.

Although all the designs shown in this report, both original and intermediate, were written manually, the modeling frameworks have provided us with the means to understand which transformations can be fully- or partially-automated in the future. For example refinement 3 should be invalidated by a unified high-level, layered modeling language; refinement 4 could be done automatically once a program analyzer understands that the core reduction operation is commutative; refinement 2 could be fully- or partially-automated once a clear design specification or constraint language can be formulated *and* analyzed. Refinement 1 though, as well as the transformation from the cube version of AESA in section 2 to the streamed version in section 5 is not so trivial to automate. This is because the nonsemantic-preserving transformations involved admit multiple (maybe better) solutions. Choosing between alternative solutions, although aided by verification and validation techniques, often relies on designer experience. In the future some of these decision could be automated or computer-aided by design space exploration techniques once we understand how to formulate the search problems.

7 Acknowledgements

The authors would like to acknowledge Per Ericsson for his contributions in designing the AESA signal processing chain (in Figure 2) and in writing the specifications.

References

Backus, John. 1978. "Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs." *Communications of the ACM* 21 (8): 613–41. https://doi.org/10.1145/359576.359579.

Benveniste, Albert, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. "The Synchronous Languages 12 Years Later." *Proceedings of the IEEE* 91 (1): 64–83.

Buck, J.T., and E.A. Lee. 1993. "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model." In *IEEE International Conference on Acoustics Speech and Signal Processing*, nil. https://doi.org/10.1109/icassp.1993.319147.

Claessen, Koen, and John Hughes. 2011. "Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs." *ACM SIGPLAN* 46 (4): 53.

Fischer, Jörg, Sergei Gorlatch, and Holger Bischof. 2003. "Foundations of Data-Parallel Skeletons." In *Patterns and Skeletons for Parallel and Distributed Computing*, edited by Fethi A. Rabhi and Sergei Gorlatch, 1–27. Springer London. https://doi.org/10.1007/978-1-4471-0097-3_1.

Hughes, John. 2007. "QuickCheck Testing for Fun and Profit." In, 4354:1–32. https://doi.org/10.1007/978-3-540-69611-7_1.

Hutton, Graham. 2016. *Programming in Haskell*. 2nd ed. New York, NY, USA: Cambridge University Press.

Lee, Edward. 2015. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models." *Sensors* 15 (3): 4837–69. https://doi.org/10.3390/s150304837.

Lee, Edward A. 2018. "Models of Timed Systems." In *Formal Modeling and Analysis of Timed Systems*, edited by David N. Jansen and Pavithra Prabhakar, 17–33. Cham: Springer International Publishing.

Lee, Edward A, and Thomas M Parks. 1995. "Dataflow Process Networks." *Proceedings of the IEEE* 83 (5): 773–801.

Lee, Edward A, and Alberto Sangiovanni-Vincentelli. 1998. "A Framework for Comparing Models of Computation." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17 (12): 1217–29.

Lee, Edward A., and Sanjit A. Seshia. 2016. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Second Edition. MIT Press. http://leeseshia.org.

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

Raudvere, Tarvo, Ingo Sander, and Axel Jantsch. 2008. "Application and Verification of Local Nonsemantic-Preserving Transformations in System Design." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27 (6): 1091–1103.

Sander, I., and A. Jantsch. 2004. "System Modeling and Transformational Design Refinement in Forsyde." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23 (1): 17–32. https://doi.org/10.1109/tcad.2003.819898.

Sander, Ingo, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. 2017. "ForSyDe: System Design Using a Functional Language and Models of Computation." In *Handbook of Hardware/Software Codesign*, edited by Soonhoi Ha and Jürgen Teich, 99–140. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-017-7267-9_5.

Sifakis, Joseph. 2015. “System Design Automation: Challenges and Limitations” 103 (November): 2093–2103.

Skillicorn, David B. 2005. *Foundations of Parallel Programming*. 6. Cambridge University Press.

Stuijk, Sander, Marc Geilen, Bart Theelen, and Twan Basten. 2011. “Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications.” In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*.

Ungureanu, George. 2018. *ForSyDe-Atom User Manual*. KTH Royal Institute of Technology. https://forsyde.github.io/forsyde-atom/assets/manual.pdf.

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)*, 1715–20. IEEE.

Ungureanu, George, Timmy Sundström, Anders Åhlander, Ingo Sander, and Ingemar Söderquist. 2019. “Formal Design, Co-Simulation and Validation of a Radar Signal Processing System.” In *Proceedings of FDL 2019: Forum on Specification & Design Languages*.