ASTRON
Netherlands Institute for Radio Astronomy

One-Click Design Flow Specification

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Daniel van der Schuur Eric Kooistra Harm Jan Pepping	ASTRON	
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: **Handtekening / Signature**	ASTRON	

Distribution list:

Group:	Others:
Andre Gunst Harm-Jan Pepping Daniel van der Schuur Erik Kooistra	Gijs Schoonderbeek

Document history:

Revision	Date	Author	Modification / Change
0.1	2014-02-12	Harm Jan	Creation
0.2	2014-02-26	Harm Jan	Added chapter 2 and 3.
0.3	2014-03-04	Harm Jan	Processed feedback of Eric and Daniel
0.4	2014-05-20	Harm Jan	Added chapter 7 Prototypes.
0.5	2014-06-04	Eric Kooistra	Updated oneclick tool flow figure 1. Added section 1.4 on clock accurate versus data driven.
0.51	2014-07-17	Eric Kooistra	Improved oneclick tool flow figure 1 regarding MyHDL and added a more elaborate description of it in section 1.3.

Table of contents:

Terminology:

DP	Data Path
Eop	End of packet
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HW	Hardware
IO	Input Output
IP	Intellectual Property
MM	Memory-Mapped
MISO	Master In Slave Out
MOSI	Master Out Slave In
RTL	Register Transfer Level
SISO	Source In Sink Out
Slice	FFT block size
Sop	Start of packet
SOSI	Source Out Sink In
ST	Streaming
SW	Software
Subband	Frequency bin of the PFB
UNB	Path to UniBoard Firmware directory.

References:

1. “APERTIF Filter bank Firmware Specification”, feb 2011, ASTRON-RP-474, E. Kooistra

1 Introduction

Development time of VHDL designs has considerably increased due to increased complexity of designs and devices. In order to decrease the development time a new form of design creation is considered. This document will introduce the context for such a revised design flow and it will serve as a communication platform to exchange and present ideas regarding this topic. The intended new design flow is currently named as “One-click design flow”.

1.1 Background

The development of large FPGA designs takes a lot of time. The one-click design flow will cover several optimizations in order to decrease development time. Apart from optimizing development time its aim is also to make programming of the supported hardware (UniBoard, UniBoard^2, Roach3…) available to a wider range of people. People without knowledge of VHDL should be able to create and simulate a design for the supported platforms. To summarize the three most important incentives for the one-click design flow:

- Decrease development time
- Create accessible design environment
- Support multiple platforms

An optimized design flow that uses a higher level language to automate the design flow could also include the possibility for functional modelling. In case all (VHDL) building blocks have a corresponding functional model it is possible to prototype and explore different design implementations in an early design stage. The functional model could already be used without a corresponding VHDL block being available.

1.2 Assumptions

1.2.1 Modelling language: Python

The scripting language Python will be used to facilitate the one-click design flow. Python has the capability to be used for functional modelling, automatic (testbench) code generation and test scripting.

1.2.2 Component based modelling

A design is always composed out of a set of components. Components can be I/O related, DSP related or platform specific. It is essential that the higher level description of a design contains all the necessary information of the design in order to auto generate the VHDL code. The high level description therefore requires a component-approach instead of a register based approach because some components don’t have registers. A register based approach would therefore be shortcoming. For every high level instantiated component a corresponding VHDL component should be instantiated as well.

1.2.3 Modelling on a structural level

The aim is NOT to generate RTL code from a functional Python description. The Python model will only be used to connect existing VHDL blocks. Automatic code generation shall be limited to the structural level of VHDL. The generated VHDL should be proper readable and understandable. It should not be necessary to look at the functional Python description to understand the structure of the generated VHDL.

1.2.4 Model reflects the actual implementation

The Python model must reflect the actual VHDL implementation in terms of building blocks and hierarchical levels. The Python model is a blueprint for the VHDL, but the generated VHDL could also function as a blueprint for a Python model.

1.2.5 Two internal bus types: MM and DP

To automate the design flow it is essential to use standardized busses. For accessing registers and memory spans of components a memory mapped (MM) interface shall be used (Memory Mapped bus). The data transfer between components is facilitated by means of a streaming (ST) protocol (DataPath = DP). Both busses have their own clock domain.

1.2.6 Multiple clock domains

The design flow should facilitate multiple clock domains. Initial a separate clock for the datapath and the memory mapped bus should be available. Later on support for multiple clock domains in the datapath must be provided.

1.2.7 Three realms in the design flow

As a result of the one-click design flow there will be three realms in which a design or a component can exist.

1.2.7.1 Modelling realm

In the realm of (functional) modelling only high level components can be modelled. These components are datablock oriented and have a generic port map. This realm serves two purposes. Based on a library with functional models a preliminary design can be made to verify several different types of algorithms and architectures. It is not an issue that for a certain model, the underlying VHDL component is not (yet) available as long as the functional behaviour is well described in a method. The second purpose is to create a model based design that is used to automatically generate the VHDL design that will result in a bit file to target a FPGA.

1.2.7.2 Simulation realm

The second realm is the simulation realm where Modelsim is used in conjunction with Python scripts. Here the RTL (written in VHDL) of the components are clock and bit accurately simulated and verified. Python will be used for verification and stimuli generation. This realm will be used for development of new VHDL componentns and it can be used for regression tests.

1.2.7.3 Hardware realm

The hardware realm is the place where a component or a design is mapped on an FPGA. Here a component is targeted and validated on hardware with the same python scripts that are used in the simulation realm.

1.3 Overview one-click design flow

Figure 1 gives a schematic representation of the one-click design flow and the role of Python and Modelsim.

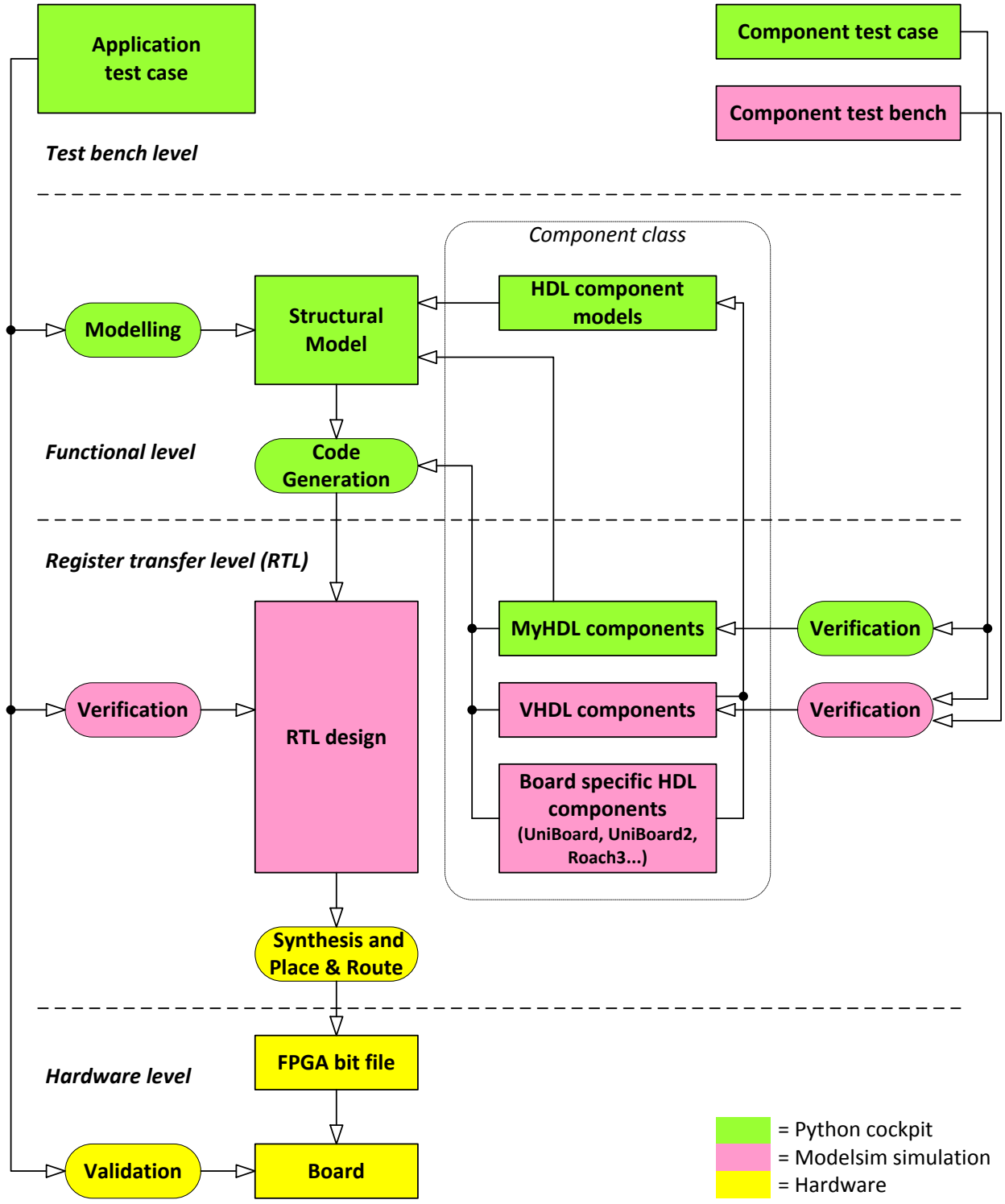

Figure 1 One-click design flow

1.3.1 Representation levels

The one-click design flow in Figure 1 shows the three realms of representing a design.

1. Functional level in Python
2. Register Transfer Level (RTL) in an HDL
3. Hardware level on the board

1.3.2 Applications, designs and components

The one-click design flow in Figure 1 distinguishes several kinds of components. At the lowest level the components are RTL building blocks that can be written in VHDL, Verilog or MyHDL. A design component consists of lower level components including the board specific components so that it can run on an FPGA. An application can consist of one design, but may also consists of multiple designs and different designs that run on one or more FPGA boards.

1.3.3 Structural model

At the functional level the functional model of a design is a structural model of the components that construct the design. For the MyHDL components the MyHDL RTL level code can directly also serve as functional model. For the VHDL components and board specific HDL components a behavioural model needs to be written manually in Python that represents the function of the HDL component in just sufficient detail. The structural model in one-click design flow in Figure 1 serves two purposes:

1. Testing
2. Code generation

The component class in Figure 1 provides methods to represent the components and to facilitate the testing and code generation.

1.3.4 Testing

The one-click design flow in Figure 1 distinguishes three levels of testing:

1. Modelling = functional testing in Python
2. Verification = RTL simulation in Modelsim
3. Validation = testing the design on the board

The one-click design flow in Figure 1 distinguishes two kinds of testing:

1. Via the MM interface using test cases in Python
2. At RTL using test benches in VHDL

At the top there is a test bench and a device under test (DUT). The DUT can be a low level component, a design or even an entire application that consists of multiple designs. The test bench provides the stimuli to the DUT and verifies the responses from the DUT. Furthermore the test bench can contain behavioural models of the DUT environment, e.g. a model of an ADC.

As shown in Figure 1 a Python design test case can apply at all three representation levels. In all cases the Python test case communicates with the DUT via the MM interface. A Python test case may also be applied to a low level component, provided that it has an MM interface. The validation on a board at hardware level and the verification at RTL level in Modelsim simulation using a test case in Python are already used for UniBoard1. On hardware the MM control is communicated via 1GbE and in simulation it is communicated via file IO with Modelsim. Verification using stimuli and responses in a VHDL test bench avoids the need for using a separate Python test case and is typically only used for lower level components that do not have an MM interface.

To support using a test case or test bench in a regression test it is necessary that it finishes automatically and that it is self-checking. This then allows to include the test in an regression test. A regression test

combines a set of tests that get run regularly or whenever the source code has been changed to provide an overall pass or failed result. This greatly helps to ensure and maintain the sanity of all components and designs.

1.3.5 Code generation

The structural model in Figure 1 contains the components that construct the design and the wires that connect the component to each other and to the FPGA IO. The VHDL components can directly be instantiated in the generated code. For the MyHDL components first VHDL needs to be generated. The code generation tasks then creates an RTL design in VHDL. Hence code generation in the one click design flow does not imply code compilation, instead it implies code construction by instantiating, parameterising and connecting existing HDL components.

1.3.6 Synthesis

The synthesis task in Figure 1 represents the entire back end tool flow in e.g. Altera Quartus or Xilinx ISE to get from an RTL design to a bit file that can be loaded onto an FPGA. Typically given some board specific constraints this back end tool flow can be fully automated.
.

1.4 Data driven or clock accurate

SA want designers to have access to the RTL without having to know VHDL, because this makes the FPGA development accessible for a broad community of engineers (like they already have in the Casper toolflow with Matlab/Simulink) and not only to digital engineers. RTL access in Python is what MyHDL provides. MyHDL simulates HDL at the RTL level, i.e. event driven with clocks. The parallel simulation in MyHDL is achieved using sensitivity lists, similar as in VHDL.

Astron wants to model the DSP at the data level, by passing along lists of valid data, so without the clock. The parallel simulation is not achieved by a sensitivity list and functions that are processed each time an input changes, but instead a function is only processed when it has sufficient data to calculate its next new output. Hence input data is only processed once. The functions may be mapped on software processes, threads or in a sequential loop. The current investigations focus on using software processes or threads, and on using pipes or queues.

MyHDL could be supported in the Astron scheme by treating the MyHDL components as just another source for creating RTL components as shown in Figure 1. Default at Astron we develop our RTL components in VHDL, but one may also start coding in MyHDL and then use the toVHDL() function of MyHDL to convert it to an VHDL component. The advantage of a MyHDL component is that the RTL description in Python can also serve as behavioral model in Python by simply adding an local clock source (assuming that the RTL description uses the standard MM and ST interfaces). For the VHDL components we manually need to create a model in Python. However this model is typically quite simple.

2 System Description

What should be described?

- What types of connection schemes should be provided: Are daisy chain or tree sufficient?
- Python methods Connect() vs Composite(). What should these functions return?
- Should it be possible to connect individual I/O-pins or only records/standardized ports?

Will feedback loops be supported?

Are MM busses connected implicitly or explicitly like the DP streams?

Will explicit assignment of the CLK and RST signals be supported?

2.1 Hierarchy

What level of hierarchy will be allowed/provided?

The use of hierarchy in the functional modelling realm is an important matter. One of the assumptions states that the functional model should be a complete reflection of the underlying VHDL design. As long as the functional model is used to generate a VHDL design there should be no hierarchical level introduced in the model other than that already exists in VHDL.
In case the model is only used for modelling it should be possible to introduce hierarchical levels. These introduced levels will only exist locally and will not be archived in a library structure.

2.2 Python component definition

How will a Python component class look?

The intended Python components classes should cover the following items in order to provide a complete reflection of the underlying VHDL instance.

- Provide methods for functional simulation
- Code generation
- Driver in Python control framework.

2.2.1 Reflect VHDL instance

2.2.1.1 Generics

The model should have attributes that reflect the values of the generics that configure a VHDL instance. The generic attributes should be considered read only and are defined when the components constructor is called.

2.2.1.2 Memory mapped registers

In case the VHDL model has an MM interface the content of these registers should also be reflected in attributes. These register attributes receive a default value in the constructor but can be modified during the lifetime of the object with methods. The read and write accessibility of the register attributes must reflect the read and write accessibility of the modelled VHDL component.

2.2.1.3 Datapath ports

The number of data in- and output ports should be reflected in attributes. If the number of ports depends on a generic attribute then this should be represented in the port attribute definition as well. There must also be

room to specify certain features of a data port, for instance: backpressure support or which fields are used for data transport (data or re & im).

2.2.1.4 Clocks

The connected clocks and resets can be derived from the availability of memory mapped registers and/or the availability of datapath ports. The connected clocks and resets should be reflected in an attribute.

2.2.2 Functional simulation

The class should provide a behavioral model (as a method) that determines the output(s) of the component based on a given input. This method is unique for every class and this behavioral model takes into account both the generic attributes for the instance as the values of the current register attributes.

2.2.3 Code generation

The Python component should have an attribute that points to the corresponding VHDL entity.

2.2.4 Python framework driver

A peripheral driver class should be implemented that facilitates reads and writes to the memory mapped registers in both simulation (verification) and in hardware (validation). A general driver for all memory mapped registers in all components would be desirable.

2.2.5 Component base class definition

Table 1 shows a possible component class definition that covers the aforementioned attributes and methods except the VHDL generics. Note the “Derived Attributes” that are literally derived from the attributes that are provided as arguments to the constructor.

Attribute name	**Description**
vhdl_reg_definition	String that contains the name(s) of the VHDL file(s) that contain(s) the memory mapped register definition(s). The constructor will create an attribute for every register that is defined (Could be a single attribute implemented as a dictionary?)
vhdl_comp_name	String that contains the name of the VHDL file that contains the toplevel of the VHDL component that is modelled.
(Derived Attributes)	
dp_clk	‘True’ when data input and/or output ports are available. Otherwise: ‘False’
mm_clk	‘True’ when a VHDL_reg_definition is defined, otherwise ‘False’.
reg_<component_name>	Dictionary that contains all available registers in the component.
Method name	**Description**
func()	A placeholder for the functional behavioral model of the component.
read_reg()	Method to read a register (from python model, simulation or hardware)
write_reg()	Method to write a register (from python model, simulation or hardware)

Table 1 Component Base Class Definition

To facilitate multiple input and output ports a separate base class is available for the data ports. An example for such a data port class is shown in Table 2.

Attribute name	**Description**
nof_ports	Number of ports
port_dir	Port direction: In or Out
data_width	Data width
clk	Label indicating the clock domain
data	Lists that represent the actual data on the in- or output ports.

Table 2 Data Port Class Definition

New components can inherit from the component base class. Besides that component specific attributes like ports can be added where appropriate. This will cover the VHDL generics that can be added as component specific attributes. Table 3 shows an example of a component class definition that describes a beamformer. Note the attributes that represent the corresponding VHDL generics and the func() method is now filled.

Attribute name	**Description**
d_in	Data port object that represents the data input. (Antenna inputs)
d_out	Data port object that represents the output. (Beamlet outputs)
g_quantize_ena	'True' to enable quantization, 'False' to disable quantization.
g_nof_weights	Specifies the number of weights.
Method name	**Description**
func()	d_out.data[0] = ∑reg_weights* d_in.data[d_in.nof_ports - 1: 0] Here the actual functional behavior is modelled. The output is defined as the sum of the weighted inputs.

Table 3 Beamformer class definition

Figure 2 shows a possible instantiation of a beamformer component that inherits from the base component and is complemented with beamformer specific attributes.

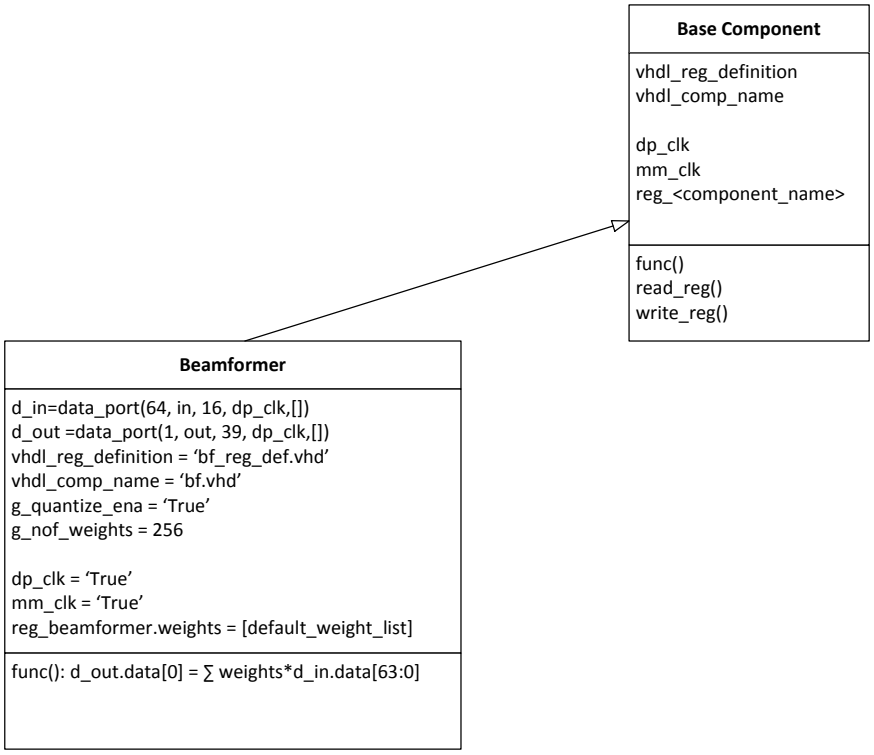

Figure 2 Possible beamformer object

3 Functional Modelling

The functional modeling realm should provide a way to model (part of) a design for preliminary architectural design research and provide a way to automatically generate a complete FPGA design (including testbenches for the simulation realm). In order to avoid that the functional modelling will become a functional simulation it is important to define the boundaries of the functional modelling realm. Here is a list of assumptions to this:

- Only standardized interfaces
- Data block accurate
- Time aware
- Clear syntax

In case the modelling realm is solely used for design generation the model should fit within the boundaries of the target FPGA. In that case it is important to be able to create a corresponding (auto-generated) VHDL design. In case the modelling realm is just used for modelling only it is not a necessity to be able to generate a corresponding VHDL design. In that case it is OK that the model exceeds the boundaries of a target FPGA. Of course design generation is not an option then.

3.1 Standardized interfaces

To automate the design flow and provide a proper way of modelling it is essential to use standardized busses. For accessing registers and memory spans of components a memory mapped interface shall be used (Memory Mapped bus). The data transfer between components is facilitated by means of a streaming protocol (DataPath). Both busses have their own clock domain. Note that these standardized busses have to be implemented at the VHDL RTL level. Functional modelling only requires that the interfaces are standardized in order to connect and access the components in a well known way.

3.1.1 Register map

Apart from the interfaces there must also be a standardized way of obtaining the register map of the components. The register definition originates in the VHDL and the functional model should refer to that definition.

3.2 Modelling resolution

The modelling resolution can be defined as the maximum number of samples that are transferred from one component to another at a single simulation cycle. In other words it defines the stepsize (in clock cycles) of the functional simulation with the assumption that there is valid data on every clock cycle. The corresponding quantity in the hardware realm is the HDL frame size (e.g. 256 clock cycles) that is marked by a SOP signal, an EOP signal and accompanied by a block sequence number. The usefulness of the HDL frame size is in the fact that the blocks that process data don't require counters to calculate the data. The start and end of an HDL frame is marked and there are always a known amount of samples in between. Based on this information and assuming the modelling resolution is independent from the HDL frame size three types of modelling resolutions can be distinguished:

- Multi Frame Resolution (Modelling resolution > HDL frame resolution). The modelling resolution is here a multiple of the corresponding HDL frame size. It theory the size could be extended to the complete simulation time in order to do a simulation in only one cycle.
- Single Frame Resolution (Modelling resolution = HDL frame resolution). In this case the modelling resolution corresponds to the frame size of the hardware realm.
- Sub Frame Resolution (Modelling resolution < HDL frame resolution). This resolution type offers cycle accurate simulations when set to 1.

Since most of the current available HDL components are HDL frame size oriented it is obvious that the Single Frame resolution is the type that is the easiest type to model in python. The Multi Frame and Sub Frame type require more administration with respect to transmitting and receiving data.

3.3 Block accurate

The components that are qualified for functional modelling must be data block oriented. This means that the functional models apply their functionality on a block of data and they provide blocks of data on their output. These blocks represent the blocks of data that are sent over the streaming datapath interfaces in the VHDL. In theory a blocksize of one data word is possible, but in practice the blocksize will, for instance, equals the number of points of the FFT.

3.3.1 Data exchange format

What is the best format for passing the data from one component to the next?

- Lists, Arrays
- What dimensions?

What are the limits of the data format candidates in terms of memory usage of the PC and speed of overall functional simulation? Should we use lists (range) or generators (xrange)?

3.4 Time awareness

Should the functional model include time as a controlled dimension (in order to facilitate feedback loops and MM accesses) ?
Within the functional model there should be the awareness of time to facilitate loopback functionality and to provide a way to change register settings over time. Having this time dimension available it becomes possible to create, for instance, a functional model of a beamformer system that is actively tracking a source, since the weights can be updated in time. Time awareness can be implemented by adding a block sequence number to the data blocks.

3.5 Clear syntax

Since there is a potential broad audience for this functional modelling system it is very important that the modelling syntax is clear and intuitive.

4 Code Generation

4.1 Memory Mapped bus

Replace the embedded processor (NIOS) by VHDL statemachine for address decoding.

For automatic code generation it is essential to have a generic and easy configurable memory mapped bus. Currently the Altera SOPC system is used to generate the bus. The bus is controlled by the NIOS II processor which runs a simple OS (unb_osy) that initiates bus accesses in response to incoming ethernet packets. The SOPC with the NIOS II processor facilitates the bus, the bus controller and the (single) bus master. Automated generation of the SOPC system requires interfering in the XML file that defines the SOPC system which is not straight forward. Other disadvantages are the vendor-dependency of the SOPC system and the chance of discontinuation or extensive upgrades of the system by Altera.

A replacement is therefore required and should meet the following:

- Vendor independent
- Compatible with current used MM interface at Astron
- Support multiple or single bus master?
- Support multiple slaves (256)
- Single read access
- Single write access
- No burst transfers
- Bus configuration (base-address, address span etc.) via VHDL Generics

How to deal with the address map and symbolic links. Store complete register map in FPGA?

Create the MM bus in VHDL:

- Instantiating the DP components (incl. port maps and connection section)
- Connecting the DP components
- Connect MM busses to MM master.

Create VHDL testbench.

5 Modelsim in-the-loop

Should we use blockgenerators and databuffers?
Or should we use HEX-files for data I/O?
Or should we use the foreign language interface?

6 Component control driver

The Python peripheral definition should also be included in the component class.

7 Prototypes

The first prototype, prototype X, shall demonstrate the functional modelling capabilities of the system and NOT the automatic code generation. It will consider a simple system as shown in Figure 3. A block generator provides two streams with blocks of data. The streams are multiplexed to a single stream in the dp_mux

component and this output stream is stored in a data buffer. The represented components are based on components that already exist in VHDL. For simplicity reasons no I/O components will be included. The model will not reflect any clocks, flow-control or memory mapped control.

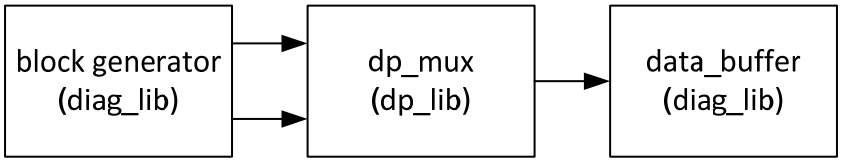

Figure 3 Schematic overview prototype X

The aim of prototype X is to demonstrate a suitable implementation for transferring data blocks from one component to another. There are three implementation options to consider:

1. Threads
2. Processes
3. Sequential

Both threads and processes offer convenient solutions for inter-component data transfer by means of queues or pipes.

7.1 Prototype X requirements

- For every connection (output-port to input-port) in the model the transferred data is stored and can be viewed when the modelling run is done.
- The length of the modelling run is defined upfront by specifying the number of blocks to process.
- A modelling run can always be interrupted. The calculated values (lists) that do exist at the moment of interruption can still be viewed.
- Each component will be modelled in a unique class that will inherit from the component class and ports class.